A Details of Memory Cost Breakdown

Figure 2 illustrates the memory breakdown of stored feature maps (input tensors and internal states of all layers), model weights, gradients, optimizer states, and temporary runtime variables when training SNNs or ANNs on ImageNet (Deng et al., 2009). We evaluate SEW ResNet-34 (Fang et al., 2021a) and Spikformer (Zhou et al., 2023) using the same settings as our main experiments (Appendix C), with ResNet-34 (He et al., 2016) and ViT (Dosovitskiy et al., 2021) mirroring the settings of SEW ResNet-34 and Spikformer, respectively. SEW ResNet-34 and Spikformer are implemented using SpikingJelly (Fang et al., 2023a), and the LIF model with CuPy backend is adopted; For ResNet-34 and ViT, we use torchvision implementations (Paszke et al., 2019). We run the experiments on a single NVIDIA A100 GPU (80 GB, CUDA 12.2).

The memory usage for weights, gradients, and optimizer states can be easily computed by summing the sizes of all tensors of these kinds. To quantify the size of stored feature maps, we measure the allocated memory after the forward pass before the backward pass starts, and subtract the sizes of weights and optimizer states from the value. For runtime variables, we first identify the critical layer l^* with the highest peak memory. In other words, the network-level peak memory occurs during backpropagation on layer l^*. The difference between the peak allocated memory at layer l^* and the allocated memory at the start of the layer's backward pass reflects runtime variable costs. Note that gradient sizes are slightly overestimated, as not all gradients are ready when global peak memory is reached. Detailed results are shown in Table 6.

Table 6: Detailed memory breakdown of different networks when trained on ImageNet. Memory costs are measured in MB.

Network	Stored Features	Weights	Gradients	Optimizer States	Runtime Variables
ResNet-34	936.49	83.15	83.15	83.15	38.25
ViT	1629.20	100.05	100.05	200.11	91.19
SEW ResNet-34	8373.98	83.15	83.15	83.15	40.50
Spikformer	33372.02	113.26	113.26	226.52	496.00

B Memory Evolution Curves

Figure 3 and Figure 5 demonstrate the memory cost evolution within one training iteration of Spiking VGG on CIFAR10-DVS. To get these curves, we record the allocated memory at the start, peak, and end of each target layer's forward pass (FP) and backward pass (BP). The resulting sequence, arranged in the temporal order of events, is

$$\begin{aligned}\Big[&\mathcal{M}^{\text{start}}_{\text{FP}^1}, \mathcal{M}^{\text{peak}}_{\text{FP}^1}, \mathcal{M}^{\text{end}}_{\text{FP}^1}, \mathcal{M}^{\text{start}}_{\text{FP}^2}, \mathcal{M}^{\text{peak}}_{\text{FP}^2}, \mathcal{M}^{\text{end}}_{\text{FP}^2}, \dots, \\ &\mathcal{M}^{\text{start}}_{\text{BP}^2}, \mathcal{M}^{\text{peak}}_{\text{BP}^2}, \mathcal{M}^{\text{end}}_{\text{BP}^2}, \mathcal{M}^{\text{start}}_{\text{BP}^1}, \mathcal{M}^{\text{peak}}_{\text{BP}^1}, \mathcal{M}^{\text{end}}_{\text{BP}^1}\Big],\end{aligned} \tag{6}$$

where FP^l and BP^l denote the forward and backward pass of layer l, respectively. The global peak memory can be defined as $\mathcal{M}^{\text{peak}} = \max\left(\{\mathcal{M}^{\text{peak}}_{\text{FP}^l}\}_l \bigcup \{\mathcal{M}^{\text{peak}}_{\text{BP}^l}\}_l\right)$.

C Details of the Main Experiments

The main experiment is implemented using PyTorch (Paszke et al., 2019) and SpikingJelly (Fang et al., 2023a).

Sequential CIFAR-10 Sequential CIFAR-10 (Fang et al., 2023b; Chen et al., 2024) is a sequence classification task derived from the standard CIFAR-10 benchmark (Krizhevsky, 2009). It is widely used for evaluating SNNs' capability to learn long-term temporal patterns. In this task, the CIFAR-10 images are fed into the model column by column, mimicking the way humans scan pictures from left to right. Each sample is a sequence with $T = 32$ elements, and each element contains 32 RGB pixels.

There are $50,000$ training samples, $10,000$ test samples, and 10 classes. Following the practice in PSN (Fang et al., 2023b), we augment the training data with random mixup (Zhang et al., 2018), random cutmix (Yun et al., 2019), random horizontal flipping, TrivialAugment (Müller & Hutter, 2021), predefined data normalization, and random erasing. An 8-layer 1D convolutional SNN is employed (SCNN) (Fang et al., 2023b). Hyperparameters and running environment are listed in Table 7.

DVS128 Gesture DVS128 Gesture (Amir et al., 2017) is an event-based gesture recognition dataset recorded by a DVS128 camera. It contains 11 gesture classes performed by 29 subjects under 3 illumination conditions with spatial resolution 128×128. For experiments, we follow the standard split provided in SpikingJelly (Fang et al., 2023a): 1,176 training samples and 288 test samples. Each recording is integrated into $T = 16$ frames, and no extra augmentations are applied. We use 7B-Net, a small-scale SEW ResNet (Fang et al., 2021a), as the backbone. See Table 7 for other hyperparameters and the running environment.

CIFAR10-DVS CIFAR10-DVS (Li et al., 2017) is a neuromorphic vision classification task created by recording CIFAR-10 images (Krizhevsky, 2009) through a Dynamic Vision Sensor (DVS) (Lichtsteiner et al., 2008). The dataset is composed of $10,000$ samples, each represented as an event stream with 2 channels and 128×128 resolution. Following the protocol of temporal effective batch normalization (TEBN) (Duan et al., 2022) and PSN (Fang et al., 2023b), we partition the dataset into $9,000$ training samples and $1,000$ test samples, downsample the resolution to 48×48, and integrate each event stream into $T = 10$ frames. The data augmentation pipeline incorporates random resized cropping, random horizontal flipping, and Neuromorphic Data Augmentation (NDA) (Li et al., 2022). We adopt a Spiking VGG11 architecture, following the practice of TEBN (Duan et al., 2022) and PSN (Fang et al., 2023b). Refer to Table 7 for hyperparameters and running environment.

ImageNet ImageNet-1k (Deng et al., 2009) is a large-scale visual recognition benchmark containing about 1.28 million training samples and 50,000 validation samples across 1,000 classes. Training on the entire ImageNet dataset is computationally expensive, so we use its $\frac{1}{32}$ subset instead, whose samples are evenly distributed across all 1000 classes. Since the peak memory cost during training is independent of the sample size, the memory footprint we report can faithfully reflect full-dataset training conditions. Each image is resized to 224×224 resolution. We utilize SEW ResNet-34 (Fang et al., 2021a), Spikformer (Zhou et al., 2023) and QKFormer (Zhou et al., 2024) architectures. The SEW residual connections in these architectures bring non-binary integer activation values (Fang et al., 2021a); for these activations, we compress them into 8-bit unsigned integers (uint8) rather than bits to avoid accuracy loss. For experiments using SEW ResNet-34, we use the same data augmentation pipeline as in the original work (Fang et al., 2021a); for both Spikformer and QKFormer, we augment data using the procedure in the original QKFormer work (Zhou et al., 2024). Hyperparameters and running environments are provided in Table 7.

Table 7: Hyperparameter settings and running environment configurations for the main experiments.

	Sequential CIFAR-10	DVS128 Gesture	CIFAR10-DVS	ImageNet	
				SEW	Transformer
λ	0.5	0.5	0.25	0.5	0.5
V_{th}	1.0	1.0	1.0	1.0	1.0
Optimizer	SGD(0.9)	SGD(0.9)	SGD(0.9)	SGD(0.9)	AdamW
L2 Reg.	0	0	5×10^{-4}	0	5×10^{-2}
Init. LR	0.1	0.1	0.1	0.1	0.001
Scheduler	Cosine	Step(0.1, 64)	Cosine	Cosine	Cosine
Loss	CE	CE	TET	TET	Smooth CE
Batch Size	128	16	32	32	32
T	32	16	10	4	4
k	2	2	2	2	2
CUDA Version	12.3	12.3	12.3	12.2	12.2
Device	1× 4090	1× 4090	1× 4090	1× A100	1× A100

D EXPERIMENTS OF MULTI-GPU TRAINING

The experiments in Table 1 of the main text are conducted on a single GPU. To further validate the scalability of the proposed framework, we conduct multi-GPU training experiments on ImageNet (Deng et al., 2009) using QKFormer (Zhou et al., 2024). The experimental setup follows Appendix C, except that 1, 2, or 3 NVIDIA A100 GPUs are used for distributed data parallel (DDP) training. We set a per-device batch size of 32. Table 8 reports the time and memory costs. Here, the batch time cost refers to the average time per training iteration for a single GPU. Throughput accounts for all GPUs, measured as the total number of training samples processed per second. The peak allocated memory is the maximum of peak allocated memory across all devices. Generally, in multi-GPU settings, our method achieves substantial memory efficiency improvements while incurring a moderate increase in training time, which is consistent with the single-GPU cases.

Table 8: Time and memory efficiency when training a QKFormer on ImageNet using multiple GPUs.

#GPUs	Neuron	Method	Throughput (samples / s) $\uparrow$	Peak. Alloc. Mem. (MB) $\downarrow$
1	SJLIF	BPTT	86.15	44571.33
	MELIF	O_4	76.51 (0.89×)	5219.93 (0.12×)
2	SJLIF	BPTT	168.43	44679.28
	MELIF	O_4	151.54 (0.90×)	5323.13 (0.12×)
3	SJLIF	BPTT	235.45	44679.28
	MELIF	O_4	211.01 (0.90×)	5323.13 (0.12×)

E ADDITIONAL MEMORY OPTIMIZATION TECHNIQUES

Low-Memory Optimizer (LOMO) Low-memory optimization (LOMO) (Lv et al., 2024b) reduces the memory cost of gradients by updating $\mathbf{W}^l$ once its gradient $\mathbf{G}^l$ is computed, instead of waiting until all gradients are available. Unlike the stateless original LOMO (Lv et al., 2024b), we retain optimizer states (e.g., those of Adam (Kingma & Ba, 2015)) to match the baseline cases. LOMO ensures that at most one gradient tensor resides in memory at a time, reducing $\sum_l \mathcal{M}_{\mathbf{G}^l}$ in Equation (4) to $\max_l \mathcal{M}_{\mathbf{G}^l}$.

Automatic Mixed Precision (AMP) Automatic mixed precision (AMP) training (Micikevicius et al., 2018) can be optionally enabled to reduce overall memory usage and accelerate training by utilizing 16-bit floats for activations and gradients. The loss is scaled to prevent underflow and ensure numerical stability.

F ACCURACY RESULTS

We report additional validation accuracy results in Table 9. Note that these experiments are designed to validate the mathematical equivalence of our method with standard BPTT rather than to maximize performance, so we do not apply advanced training tricks like random temporal delete (Fang et al., 2021a). We train 300, 192, and 100 epochs for Sequential CIFAR-10, DVS128 Gesture and CIFAR10-DVS, respectively. For $\frac{1}{32}$ ImageNet, we report validation accuracy at the fifth epoch to reduce training cost, which is sufficient to demonstrate the equivalence. The results show that MELIF attains accuracy nearly identical to SJLIF across all benchmarks, with minor discrepancies arising only from different backend numerical behaviors. In most cases, the optimization pipeline itself does not affect accuracy. However, O_3 and O_4 for Spikformer and QKFormer slightly influence accuracy due to the temporal segment partitioning on weight layers. Appendix G discusses this issue in detail. **These small deviations stem purely from numerical computation rather than from any approximation in the gradient computation. The gradients produced by our method remain free from systematic bias.**

Table 9: Comparison of validation accuracy (%). For $\frac{1}{32}$ ImageNet, we report the validation accuracy at epoch 5. For SJLIF conditions, we report mean $\pm$ std over three runs. For MELIF conditions, we report the results on a single run using a fixed seed.

Task	Network	SJLIF	MELIF				
		BPTT	BPTT	O_1	O_2	O_3	O_4
Sequential CIFAR-10	SCNN	82.53±0.25	82.36	82.36	82.36	82.36	82.36
DVS128 Gesture	7B-Net	95.08±0.87	95.14	95.14	95.14	95.14	95.14
CIFAR10-DVS	Spiking VGG	85.98±0.25	86.10	86.10	86.10	86.10	86.10
$\frac{1}{32}$ ImageNet	SEW ResNet-34	3.46±0.21	3.50	3.50	3.50	3.50	3.50
	Spikformer	1.03±0.16	0.90	0.90	0.90	1.10	1.10
	QKFormer	1.06±0.12	1.20	1.20	1.20	1.10	1.10

G Potential Sources of Numerical Discrepancies

Numerical discrepancies in gradients may arise when temporal GC segment partitioning is applied to layers with learnable parameters. Without temporal partitioning, the gradient is first computed for each time step $t \in \{1, \dots, T\}$ and batch sample $n \in \{1, \dots, N\}$, and then summed over the temporal and batch dimensions:

$$\mathbf{G} = \sum_{t=1}^{T} \sum_{n=1}^{N} \mathbf{G}_{t,n}, \tag{7}$$

where $\mathbf{G}_{t,n}$ denotes the gradient contribution at time step t from sample n. In contrast, when the temporal dimension is partitioned ($k = 2$ for example), the accumulation is performed in two stages:

$$\mathbf{G}^{(1)} = \sum_{t=1}^{\frac{T}{2}} \sum_{n=1}^{N} \mathbf{G}_{t,n}, \quad \mathbf{G}^{(2)} = \sum_{t=\frac{T}{2}+1}^{T} \sum_{n=1}^{N} \mathbf{G}_{t,n}, \tag{8}$$

followed by a final aggregation:

$$\mathbf{G} = \mathbf{G}^{(1)} + \mathbf{G}^{(2)}. \tag{9}$$

Although mathematically equivalent to the unpartitioned case, these operations differ in numerical practice because **floating-point addition is not associative**. As a result, reordering the accumulation of gradient terms leads to slight deviations in the final gradient values. This explains the minor accuracy deviations observed in Table 9 for Spikformer and QKFormer at O_3 and O_4.

H Time Cost of Memory Optimization

Table 10 reports the time cost of the memory optimization pipeline at each optimization level. For SCNN, the overhead increases moderately with higher optimization levels, reflecting the additional computations from spatial and temporal segment partitioning and greedy segment restoration. For QKFormer, the jump in time cost from O_3 to O_4 is much more pronounced, primarily due to the transformer's greater depth, which increases profiling costs and the number of segments to iterate over. Importantly, this overhead is incurred only once before training and is negligible relative to the total training time.

Table 10: Time (in seconds) spent by the memory optimization pipeline at each optimization level.

	O_1	O_2	O_3	O_4
SCNN	1.08	26.32	30.63	75.41
QKFormer	1.13	44.91	78.58	564.26

I TUTORIAL

We provide a brief tutorial on using the proposed automatic memory optimization pipeline, taking the training of Spiking VGG on CIFAR10-DVS as an example. The model can be defined using PyTorch (Paszke et al., 2019) and SpikingJelly (Fang et al., 2023a) as shown in the code below.

```
class VGGBlock(nn.Module):
    def __init__(
        self, in_plane, out_plane, T,
        neuron_type, preceding_avg_pool=False, **kwargs
    ):
        super().__init__()
        proj_bn = []
        if preceding_avg_pool:
            proj_bn.append(nn.AvgPool2d(2))
        proj_bn += [
            nn.Conv2d(in_plane, out_plane, 3, 1, 1),
            nn.BatchNorm2d(out_plane),
        ]
        self.proj_bn = SeqToANNContainer(*proj_bn)
        kwargs["T"] = T
        self.neuron = get_neuron(neuron_type, **kwargs)

    def forward(self, x_seq):
        return self.neuron(self.proj_bn(x_seq))

class CIFAR10DVSVGG(nn.Module):
    def __init__(self, T, neuron_type, dropout=0.25, **kwargs):
        super().__init__()
        self.features = nn.Sequential(
            VGGBlock(2, 64, T, neuron_type, False, **kwargs),
            VGGBlock(64, 128, T, neuron_type, False, **kwargs),
            VGGBlock(128, 256, T, neuron_type, True, **kwargs),
            VGGBlock(256, 256, T, neuron_type, False, **kwargs),
            VGGBlock(256, 512, T, neuron_type, True, **kwargs),
            VGGBlock(512, 512, T, neuron_type, False, **kwargs),
            VGGBlock(512, 512, T, neuron_type, True, **kwargs),
            VGGBlock(512, 512, T, neuron_type, False, **kwargs),
            layer.AvgPool2d(2, step_mode="m"),
        )
        d = int(48 / 2 / 2 / 2 / 2)
        l = [nn.Dropout(dropout)] if dropout > 0 else []
        l.append(nn.Linear(512 * d * d, 10))
        self.classifier = nn.Sequential(*l)
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(
                    m.weight, mode='fan_out', nonlinearity='relu'
                )

    def forward(self, input):
        # input.shape = [N, T, C, H, W]
        input = input.transpose(0, 1).contiguous()
        # [T, N, C, H, W]
        x = self.features(input)
        x = torch.flatten(x, 2)  # [T, N, D]
        x = self.classifier(x)
        return x
```

Users can define spatial partitioning rules by implementing the `__spatial_split__` method, which returns a tuple of submodules corresponding to the spatial subsegments of a layer. For instance, a VGG block can be split into a convolution-plus-batch-norm segment and a spiking neuron segment.

```
class VGGBlock(nn.Module):
    def __spatial_split__(self):
        return self.proj_bn, self.neuron
```

To define temporal partitioning rules, users should implement the `__tc_init_states__` and `__tc_forward__` methods. `__tc_init_states__` returns a list of initial hidden states, while `__tc_forward__` takes a chunk of input tensors along with the initial hidden states, and then returns the corresponding outputs and updated hidden states. The stateless layer container `SeqToANNContainer` is the simplest case, where no hidden states are required and the temporally chunked forward pass is just the same as the container's original forward pass.

```
class SeqToANNContainer(layer.SeqToANNContainer):
  """Stateless layer container that supports temporal chunking"""
  def __tc_init_states__(self, x_seq):
      return []

  def __tc_forward__(self, xc):
      return [self.forward(xc),]
```

A more complex example is the `NeuronMaxPool` block:

```
class NeuronMaxPool(nn.Module):
    def __init__(self, neuron_type, **kwargs):
        super().__init__()
        self.neuron = get_neuron(neuron_type, **kwargs)
        self.pool = SeqToANNContainer(
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )

    def forward(self, x_seq):
        return self.pool(self.neuron(x_seq))

    def __tc_init_states__(self, x_seq):
        device, dtype = x_seq.device, x_seq.dtype
        return [torch.zeros([], device=device, dtype=dtype)]

    def __tc_forward__(self, xc, v):
        sc, v = self.neuron.multistep_state_update(xc, v)
        yc = self.pool(sc)
        return yc, v
```

which means that the hidden state (the neuron's membrane potential) is initialized to zero, and the temporally forward pass consists of a multi-step state update of the neuron followed by max pooling. In this example, there is only one input, one hidden state, and one output. However, multiple inputs, hidden states, and outputs are also supported. Finally, the `memory_optimization` function can be called to apply the automatic pipeline.

```
net = CIFAR10DVSVGG(T, neuron_type, dropout, **kwargs)
net = memory_optimization(
    net,
    instance=(VGGBlock,),
    dummy_input=torch.rand(32, T, 2, 48, 48),
```

```
    compress_x=True,
    level=4,
    verbose=True,
    temporal_split_factor=2,
)
```

where `instance` specifies the layer types to apply gradient checkpointing, `dummy_input` is a sample input tensor for profiling, `compress_x` indicates whether to compress input spikes, `level` sets the optimization level, and `temporal_split_factor` is the k factor that controls the granularity of temporal partitioning. After optimization, the model can be trained using standard procedures without further modification.

Note that the pipeline will automatically check whether spike compression is applicable at each GC segment based on the input distribution. Users can also manually specify spike compressors by setting the module's `x_compressor` attribute. For instance, for a layer in a SEW residual block (Fang et al., 2021a) whose input is non-binary integer tensors, we can compress the input to 8-bit unsigned integers (uint8):

```
class SEWBlock(nn.Module)
    def __init__(self, c_in, c_mid, neuron_type, **kwargs):
        super().__init__()
        self.conv = nn.Sequential(
            Conv3x3(c_in, c_mid, neuron_type, **kwargs),
            Conv3x3(c_mid, c_in, neuron_type, **kwargs),
        )
        self.conv[0].x_compressor = "Uint8SpikeCompressor"

    def forward(self, x: torch.Tensor):
        out = self.conv(x)
        out = out + x
        return out
```

J The Effect of Spike Compression on Training Memory

Table 11 reports the peak memory usage corresponding to Figure 5. The majority of memory saving comes from layer-wise GC (BPTT vs. O_1, compression disabled), while spike compression alone only provides marginal memory savings (O_1, compression disabled vs. O_1, compression enabled). However, as shown in Figure 5, spike compression reduces the memory footprint of activations, thus substantially lowering the instantaneous memory usage of deeper layers. This reduction creates the headroom for stronger spatio-temporal partitioning, leading to larger memory savings at higher optimization levels (O_3 and O_4). In summary, **spike compression is not the main source of memory efficiency, but an enabling factor that allows spatio-temporal partitioning to further reduce peak memory.**

Table 11: Peak allocated memory (MB) of Spiking VGG when training on CIFAR10-DVS with spike compression enabled or disabled ($T = 10$, batch size is 32). See Figure 5.

Compression	BPTT	O_1 (+GC)	O_3 (+partitioning)	O_4 (+restoration)
✓	/	2887.75	2349.39	2349.39
✗	6131.07	2892.63	2530.66	2530.66

K Detailed Runtime Profiling

Table 12 reports the the forward and backward runtime for each layer in Spiking VGG. Note that the fully connected classification head is omitted since no change is applied to it across all optimization

levels. GC introduces additional backward computation roughly equivalent to a single extra local forward pass. Spike compression and decompression further add small overheads to both forward and backward passes, but the increase is negligible relative to the total runtime, demonstrating the efficiency of bit compression; note that Conv0 do not apply input spike compression. In this example, spatial partitioning is applied only to Conv1, while temporal partitioning is skipped since it does not yield additional memory benefits (see Algorithm 2). As a result, virtually no extra computational cost. Finally, greedy segment restoration significantly reduces the computation load of both passes. Conv3 and Conv5 are reverted to standard BPTT blocks, and their forward and backward runtimes return to BPTT level.

Table 12: Layer-wise runtime profiling for Spiking VGG on CIFAR10-DVS (MELIF, $T = 10$, batch size is 32). Results are averaged over 200 iterations with 10 warmup iterations and reported in milliseconds.

Condition	Stage	Conv0	Conv1	Conv2	Conv3	Conv4	Conv5	Conv6	Conv7
BPTT	fwd	2.48	5.96	4.16	5.22	2.80	3.98	1.04	0.91
	bwd	4.28	12.82	8.57	10.13	5.58	8.29	2.26	1.95
+ GC	fwd	2.52	5.99	4.20	5.26	2.81	4.01	1.07	0.92
	bwd	6.80	18.86	12.78	15.42	8.45	12.33	3.27	2.89
O_1	fwd	2.51	6.19	4.51	5.46	3.04	4.11	1.18	1.02
	bwd	6.81	19.08	13.21	15.63	8.71	12.46	3.41	3.04
O_3	fwd	2.49	**6.15**	4.48	5.38	3.05	4.09	1.17	0.99
	bwd	6.72	**19.02**	13.24	15.62	8.56	12.22	3.39	2.98
O_4	fwd	2.50	6.16	4.49	**5.18**	3.03	**3.99**	1.16	1.00
	bwd	6.76	19.01	13.23	**10.08**	8.55	**8.24**	3.34	2.97

We further investigate how training time cost scales with the number of checkpointed layers and temporal splits. Since GC performs one fixed-cost recomputation per segment, the total overhead increases as the number of GC segments grows. However, the scaling is not strictly linear, since recomputation cost varies across layers. As shown in the left plot of Figure 8, the overhead grows as more GC segments are added, but the increments are uneven. From Table 12, we know that spatial partitioning has almost no effect on training speed. In contrast, temporal partitioning actually reduces temporal parallelism, thereby slowing down training (especially for models with a large T). To illustrate this effect, we measure per-batch training time cost of DH-SFNN on SHD. Training time per batch increases nonlinearly with the the temporal partitioning factor k.

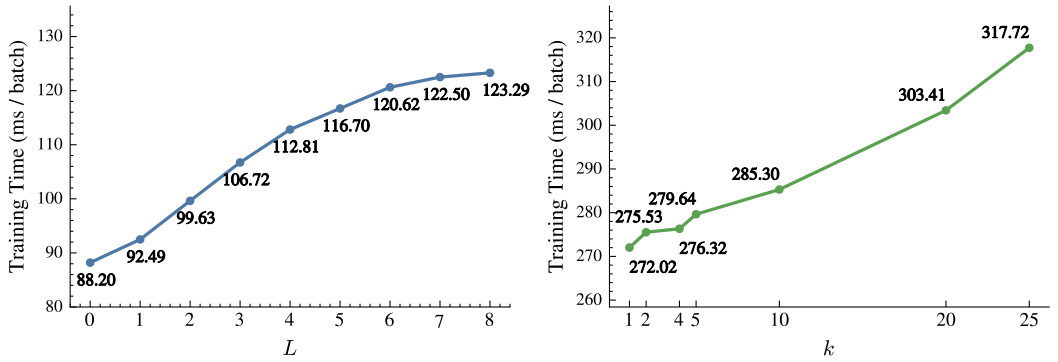

Figure 8: Left: taining time per batch (forward + backward) of Spiking VGG on CIFAR10-DVS as a function of the number of GC segments. The first L layers adopt GC, where $L = 0$ corresponds to standard BPTT. $T = 10$, and batch size is 32. Right: training time per batch of DH-SFNN on SHD as a function of the temporal partitioning factor k under O_3. $k = 1$ indicates no temporal partitioning. $T = 100$, and batch size is 128. Experiments are run on a single 4090 (24 GB).

L FINE-GRAINED ABLATION STUDY

Table 2 shows the ablation results of optimization levels on CIFAR10-DVS. To better understand the impact of each GC adjustment strategy, we conduct fine-grained ablations on QKFormer for ImageNet. As shown in Table 13, spatial partitioning provides substantial memory reduction with almost no impact on training throughput. Temporal partitioning also reduces memory usage, though it introduces a slight slowdown due to reduced temporal parallelism. When combined, the two strategies complement each other effectively, lowering peak memory to 5219 MB. Greedy restoration further improves throughput while preserving memory savings. With all three strategies jointly applied, throughput reaches 76.51 samples/s, the highest among all variants; notably, it even exceeds the condition without temporal partitioning because there are more GC segments restored to standard BPTT blocks. Overall, these ablations clarify the individual and collective contributions of the three components to memory and computational efficiency, confirming the intended synergistic effect of the full pipeline.

Table 13: Ablation study of QKFormer for ImageNet on the impact of spatial partitioning, temporal partitioning, and greedy restoration. All conditions adopt MELIF, GC and spike compression.

Spatial Partition	Temporal Partition	Greedy Restoration	Throughput (sample/s)	Memory (MB)	Annotation
			66.13	7726.55	O_1
✓			66.02	6834.48	O_2
	✓		63.82	6920.25	
		✓	73.17	7725.87	
✓	✓		64.01	5219.93	O_3
✓		✓	73.07	6833.87	
	✓	✓	70.24	6920.25	
✓	✓	✓	76.51	5219.93	O_4

M LOSSLESS SPIKE COMPRESSORS

As discussed in Section 4.3, we adopt bit representation as the default lossless spike compressor due to its superior speed and memory efficiency. To validate this choice, we compare it with two alternatives: sparse representation (storing indices of non-zero elements) and lossless bit-stream compressor (e.g., ANS from nvCOMP [1]). Experiments are performed on Sequential CIFAR-10 using SCNN ($T = 32$, batch size is 128) on an NVIDIA GeForce RTX 4090. The results in Table 14 show that bit compression consistently achieves the lowest memory footprint and highest throughput under both O_1 and O_4. Sparse representation yields slightly higher memory consumption and lower speed, while ANS provides moderate compression gains but is substantially slower.

For a more direct comparison, we evaluate compressed size and compression-decompression time cost across a range of firing rates ρ, using a float32 spike tensor of 10^7 elements (38.14 MB) as input. Time costs are averaged over 100 trials following 20 warm-up runs. As Table 15 and Table 16 show, bit compression consistently produces a fixed-size 1.19 MB representation regardless of sparsity, whereas sparse representation's memory efficiency decreases rapidly as ρ increases. ANS achieves small compressed sizes at low sparsity but is over an order of magnitude slower. Notably, firing rates in modern activation-based SNNs typically fall within 0.02 to 0.35 (Zhou et al., 2024), a regime in which bit representation performs effectively. These results confirm that bit representation is both efficient and effective.

N LIMITATIONS, FUTURE WORK, AND SOCIAL IMPACTS

While the proposed memory optimization pipeline achieves significant memory reduction with broad compatibility and preserved accuracy, several limitations remain. First, GC inevitably introduces computational overhead due to the recomputation of intermediate features during backward pass, and spike compression also slightly adds computational burden. Although we alleviate these by greedily

[1] https://developer.nvidia.com/nvcomp

Table 14: Comparison of lossless spike compressors (bit, sparse, ANS) on Sequential CIFAR-10 (SCNN, $T = 32$, batch size is 128).

Compressor	Throughput (sample / s)		Memory (MB)	
	O_1	O_4	O_1	O_4
bit	496.79	474.98	4768.11	5138.76
sparse	527.53	516.99	4454.24	5020.27
ANS	509.42	497.71	2029.53	2542.50

Table 15: Compressed memory (MB) of spike compressors across firing rates ρ

Compressor	$\rho = 0.01$	$\rho = 0.1$	$\rho = 0.2$	$\rho = 0.5$	$\rho = 0.8$	$\rho = 0.9$
bit	1.19	1.19	1.19	1.19	1.19	1.19
sparse	0.76	7.63	15.27	38.14	61.03	68.66
ANS	0.30	1.68	2.82	5.14	6.61	6.95

Table 16: Compression time cost (sec) of spike compressors across firing rates ρ

Compressor	$\rho = 0.01$	$\rho = 0.1$	$\rho = 0.2$	$\rho = 0.5$	$\rho = 0.8$	$\rho = 0.9$
bit	0.1234	0.1257	0.1262	0.1216	0.1250	0.1215
sparse	0.1538	0.1568	0.1616	0.2210	0.2965	0.3119
ANS	2.6097	2.4416	2.4407	2.5761	2.6305	2.6603

restoring low-memory-impact GC segments to standard BPTT segments and implementing efficient Triton kernels for compression and decompression, the overall training speed can be reduced to about $0.8\times$ that of the baseline in the worst case. Second, our experiments mainly focus on visual and audio classification benchmarks, which is a common practice in SNN research. While the pipeline is theoretically applicable to other modalities, its effectiveness on tasks such as language modeling remains to be validated. Thirdly, the proposed method is specially designed for layer-wise training setting (Table 5), which is common for modern medium- to large-scale SNNs. Its applicability to strict online learning settings, where model parameters are updated immediately at each time step, is limited. Fourth, the current pipeline follows user-specified partitioning schemes to reduce the search space. Automatic generation of partitioning rules are not yet supported.

Future work can address these limitations in several directions. One direction is to evaluate the framework on large-scale SNNs for language tasks, and to design optimization strategies tailored to language backbones. This would broaden the applicability and further demonstrate the generalizability of the pipeline. Another promising direction is to automatically generate spatio-temporal partitioning rules, enabling fully automated memory optimization without manual intervention. To this end, more principled optimization approaches, such as dynamic programming (Gruslys et al., 2016), could be employed to search for efficient partitioning schemes.

By reducing the memory cost of SNN training while retaining BPTT-level accuracy and broad compatibility, our method lowers the hardware barriers for scaling up SNNs. The pipeline facilitates the deployment of energy-efficient SNNs on resource-constrained platforms, including mobile and edge IoT devices. Such advances can democratize access to neuromorphic computing, promote sustainable AI solutions, and ultimately contribute to reduced energy consumption in intelligent systems. We do not see any negative societal impacts from this work.

O USE OF LARGE LANGUAGE MODELS

We utilized large language models (LLMs) to refine phrasing, correct spelling and grammar, and enhance the clarity of expressions. Additionally, LLMs were employed to assist in result visualization, such as providing initial code templates or optimizing figure layout suggestions. However, the core ideas, methodological design, code framework development, and key contributions of this paper were independently conceived and completed by the authors, without relying on LLMs for substantive support.