SPIKINGPHOENIX: SEMANTICS-PRESERVING EXECUTION OPTIMIZATION FOR DIRECT SNN TRAINING ON GPUS

Anonymous authors
Paper under double-blind review

ABSTRACT

Direct multi-step Spiking Neural Network (SNN) training remains costly on graphics processing units (GPUs) after temporal fusion because backward propagation spans recurrent state, backward-required forward quantities, and persistent minibatch updates. Existing systems optimize these costs separately, leaving their composition under a fixed training update unresolved. We present SpikingPhoenix, which reorganizes nested recurrent and minibatch execution boundaries without changing that update. SEQUENCE pairs forward and reverse scans, SEGMENT-K rematerializes discarded membrane state within recurrent backward, and GRAPH-V2 replays the device-side state transitions required by each minibatch update. The transformed paths match the reference exactly on discrete observables and within data-type-specific tolerances on floating-point observables. Across representative Spiking Transformers, including QKFormer and MaxFormer, SpikingPhoenix achieves 1.41–$2.73\times$ end-to-end training speedups over matched SpikingJelly runs on complete CIFAR-10 schedules. On ImageNet-1K, SEQUENCE further achieves 1.37-1.45× geometric-mean speedups across T=2,4,8.

1 INTRODUCTION

Direct multi-step Spiking Neural Network (SNN) training commonly uses backpropagation through time (BPTT) with surrogate gradients on graphics processing units (GPUs) (Fang et al., 2023). Each minibatch couples recurrent membrane dynamics, backward-required forward quantities, and persistent optimizer, normalization, scheduler, and mixed-precision state. Recurrent kernels determine temporal scan granularity, whereas minibatch execution determines which update-state transitions occur before control returns to the host. Because the associated state lifetimes differ, runtime depends not only on which operators execute but also on how temporal work and persistent updates are grouped.

Prior systems optimize distinct parts of this execution stack. SpikingJelly provides multi-step execution, parallelizes temporal work for stateless layers, and fuses CUDA kernels for stateful neurons (Fang et al., 2023); Temporal Fusion further reduces temporal overhead across multiple steps (Li et al., 2024). Activation checkpointing and SNN memory methods reduce retained state through recomputation (Chen et al., 2016; Jain et al., 2020; Liang et al., 2022; Huang et al., 2026). Triton and PyTorch compilation restructure operator execution (Tillet et al., 2019; Ansel et al., 2024), while CUDA Graphs replay captured device work at stable addresses (Jin et al., 2026). These techniques separately reduce recurrent overhead, retained forward state, or host dispatch.

Yet equivalent work can move across an execution boundary without changing the reference training update. Backward-required state can be rematerialized inside a recurrent kernel or after runtime re-entry. Likewise, a local operator rewrite can execute outside or within graph replay. This raises two linked questions: how can direct SNN training be reorganized while preserving recurrent and minibatch-update semantics, and how do the resulting boundaries change equivalent-work cost and local-transformation gain?

An execution boundary is a runtime region that encloses a unit of work and its state transitions before control returns to the surrounding runtime. For recurrent scans, it sets temporal launch granularity and transient-state lifetime. For rematerialization, it determines whether regenerated forward state

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

remains inside recurrent backward or requires runtime re-entry; for graph replay, it determines which persistent update transitions are captured. Preserving training semantics requires agreement with the reference recurrent computation and persistent minibatch-state transitions.

SpikingPhoenix transforms two nested boundaries under this semantic constraint. The recurrent boundary encloses forward recurrence, reverse temporal adjoints, and backward-required state: SEQUENCE realizes paired scans, while SEGMENT-K retains periodic post-reset states and rematerializes only the prefix required at each reverse position. The enclosing minibatch boundary contains the recurrent path and persistent device-side update transitions; GRAPH-V2 replays the device-side minibatch update as a state-complete transaction while keeping scheduler ordering explicit on the host.

The transformed recurrent and replay paths satisfy the declared semantic criteria. Controlled placement and rewrite studies then expose boundary dependence: moving the same rematerialization outside recurrent backward multiplies temporal launch count, and query, key, and value (QKV) rewrite gains change with the enclosing execution mode. The speedups transfer across spiking Transformer architectures, GPUs, ImageNet-1K workloads, and complete CIFAR-10 schedules.

Our main contributions are summarized as follows:

- We formalize the reference update and nested recurrent and minibatch execution boundaries for direct multi-step SNN training, with criteria for semantic agreement, execution cost, and local-rewrite gain.
- We restructure recurrent execution with paired forward and adjoint scans and boundary-local rematerialization (SEQUENCE and SEGMENT-K), exposing a controllable trade-off between retained membrane state and rematerialization work.
- We extend minibatch replay to the device-side state transitions that determine subsequent training state (GRAPH-V2), while keeping learning-rate scheduling explicit on the host.

2 Related Work

Recurrent execution. SLAYER and EXODUS provide CUDA implementations of temporal backpropagation with gradients consistent with BPTT (Shrestha & Orchard, 2018; Bauer et al., 2023). SpikingJelly's multi-step execution and custom CUDA kernels fuse work across time, and Temporal Fusion further reduces temporal overhead across multiple steps (Liang et al., 2022; Fang et al., 2023; Li et al., 2024). Spyx uses JAX/XLA compilation, while FlashRNN develops recurrent kernels for hardware-efficient recurrence beyond SNNs (Heckel & Nowotny, 2024; Pöppel et al., 2025). SpikingPhoenix uses the fused recurrent path as the execution boundary for backward-required state and rematerialization, then composes that boundary with minibatch replay. SEQUENCE realizes the paired forward and reverse scans within this boundary.

Temporal learning formulations. Online Training Through Time, Sparse Local Temporal Training, Rate-Based Backpropagation, and fixed-point training reduce temporal cost by changing gradient propagation or the optimization formulation (Xiao et al., 2022; Meng et al., 2023; Yu et al., 2024; Feng et al., 2025). EventProp derives continuous-time gradients at spike events, while SparseProp exploits event and network sparsity for sparse recurrent SNN simulation and BPTT (Wunderlich & Pehle, 2021; Engelken, 2023). SpikingPhoenix instead keeps the discrete-time surrogate-gradient recurrence and reference minibatch update fixed while changing GPU execution.

State retention and rematerialization. Checkpointing and Checkmate trade retained tensors for recomputation (Chen et al., 2016; Jain et al., 2020). SNN systems additionally use temporal recomputation, reversibility, and spike compression (Liang et al., 2022; Zhang & Zhang, 2024; Hu et al., 2024; Huang et al., 2026). SEGMENT-K stores periodic membrane boundaries and rematerializes required prefixes inside recurrent backward execution, separating retained state from rematerialization placement.

Compilation and graph replay. Nimble, Triton, and PyTorch 2 reduce scheduling or kernel overhead through ahead-of-time scheduling, custom kernels, or compilation (Kwon et al., 2020; Tillet et al., 2019; Ansel et al., 2024). CUDA Graphs provide the replay primitive for device work at stable addresses, and GraCE broadens graphability through compiler transformations and selective

deployment (Jin et al., 2026; Ghosh et al., 2026). GRAPH-V2 uses the device-side minibatch update for direct SNN training as the replay boundary: recurrent resets, normalization state, gradient participation, optimizer state, and dynamic mixed precision remain inside replay; scheduler ordering stays on the host.

3 Training Semantics and Execution Boundaries

3.1 Direct Training Semantics

Following the standard charge–fire–reset convention for directly trained SNNs (Fang et al., 2023), consider a sequence of T time steps. Let H_t denote the membrane potential after charging but before firing, S_t the emitted spike, and V_t the membrane potential after reset. We write one neuron step as

$$\begin{aligned} H_t &= f(V_{t-1}, X_t), \\ S_t &= \Theta(H_t - V_{\mathrm{th}}), \\ V_t &= \mathcal{R}(H_t, S_t), \end{aligned} \tag{1}$$

where X_t is the input, f the charging map, V_{th} the firing threshold, Θ the Heaviside step function with $\Theta(0) = 1$ under the SpikingJelly convention, and $\mathcal{R}$ the hard or soft reset map. Appendix A specifies the exact charge rules for the evaluated leaky integrate-and-fire (LIF) neuron with fixed parameters, its integrate-and-fire (IF) specialization, and the implemented reset cases. Detaching the reset leaves the forward equations unchanged and changes only the surrogate derivative of the reset branch.

Direct training uses the hard threshold during the forward pass and substitutes a surrogate derivative during backward. Let σ denote the surrogate function, with $\sigma'_t := \sigma'(H_t - V_{\mathrm{th}})$, and let $\bar{S}_t^{\mathrm{net}}$ denote the adjoint arriving from network consumers of S_t other than the reset equation, and let $\bar{V}_t$ denote the total adjoint at the state after reset. For the elementwise IF and LIF recurrence considered here, all reset Jacobians below use the surrogate derivative for the spike function. Define $j_t^{\mathcal{R}} := \partial V_t / \partial H_t$, $j_t^V := \partial H_t / \partial V_{t-1}$, and $j_t^X := \partial H_t / \partial X_t$. For loss L, define $\bar{H}_t := \partial L / \partial H_t$ and $\bar{X}_t := \partial L / \partial X_t$. With $\odot$ denoting elementwise multiplication, reverse propagation through one time step is

$$\begin{aligned} \bar{H}_t &= \bar{S}_t^{\mathrm{net}} \odot \sigma'_t + j_t^{\mathcal{R}} \odot \bar{V}_t, \\ \bar{X}_t &= j_t^X \odot \bar{H}_t, \\ \bar{V}_{t-1} &= \bar{V}_{t-1}^{\mathrm{ext}} + j_t^V \odot \bar{H}_t. \end{aligned} \tag{2}$$

Here $\bar{V}_{t-1}^{\mathrm{ext}}$ denotes any adjoint injected directly at the state after reset outside the temporal recurrence. Intermediate V_t are not exposed by the evaluated operator interface, so this term is zero except at an exposed boundary. Any adjoint at the final state initializes $\bar{V}_T$ before the reverse scan. Appendix A derives the explicit reset and charge derivatives. Separating $\bar{S}_t^{\mathrm{net}}$ from the reset branch avoids counting the reset dependency twice.

Let z collect persistent minibatch state: model parameters and buffers, exposed recurrent and normalization state, optimizer and scheduler state, gradient presence, mixed-precision scale and overflow state, and consumed random number generator state. For inputs x and targets y, write the reference update as $z^+ = \mathcal{U}_{\mathrm{ref}}(x, y; z)$. The membrane trajectory remains transient unless exposed beyond the update. This update defines training semantics: a transformed execution is semantics-preserving when discrete observables match exactly and floating-point observables satisfy data-type-specific tolerances. Appendix C reports the criteria and deviations.

Execution boundary. We use execution boundary for a runtime region that encloses a unit of work and the state transitions it performs before control returns to the surrounding runtime. For fixed logical work W, let $C(W; \mathcal{B})$ denote measured cost under boundary $\mathcal{B}$. For local transformation τ, let $\mathrm{Thr}(W; \mathcal{B})$ denote throughput and define its gain as

$$G(\tau \mid \mathcal{B}) := \frac{\mathrm{Thr}(\tau(W); \mathcal{B})}{\mathrm{Thr}(W; \mathcal{B})}. \tag{3}$$

3.2 State Retention and Rematerialization

Reverse propagation in Eq. 2 requires forward quantities entering the surrogate and reset Jacobians. They may be retained from the forward scan or rematerialized before reverse use. Retention sets the state carried into backward; rematerialization adds recurrent work and can change launch topology when it crosses an execution boundary. Section 4 defines these policies.

3.3 Process Memory and Graph Replay

We report process memory separately because it also includes activations, optimizer storage, allocator history, static buffers, and storage private to the graph. The experiments record peak PyTorch allocation M_{alloc}, allocator reservation M_{reserved}, and GPU process peak M_{process} reported by the NVIDIA Management Library (NVML).

CUDA Graph replay replaces repeated host issuance with replay of captured device work. A replay requires stable captured topology and storage addresses, although values in persistent tensors may change between updates. Changes that alter the captured shape or storage topology require a new capture. Section 4.3 defines the device-side training state and update transitions enclosed by GRAPH-V2 replay.

4 Execution-Boundary Transformations

Direct training is transformed at two nested execution boundaries. The recurrent boundary encloses forward recurrence, reverse temporal adjoints, and backward-required state; the minibatch boundary encloses this path with persistent device-side update transitions. SEQUENCE and SEGMENT-K realize the recurrent transformation, whereas GRAPH-V2 realizes state-complete replay.

4.1 Recurrent Forward and Adjoint Scans (SEQUENCE)

For these fixed-parameter IF and LIF neurons, SEQUENCE executes the T forward neuron steps as one recurrent scan and the temporal adjoints as one reverse scan. Writing $\mathcal{F}_{\mathrm{seq}}$ and $\mathcal{B}_{\mathrm{seq}}$ for the forward and reverse recurrent scans, Eqs. 1 and 2 give

$$\begin{aligned}(S_{1:T}, V_T, \mathcal{C}) &= \mathcal{F}_{\mathrm{seq}}(X_{1:T}, V_0),\\ (\bar{X}_{1:T}, \bar{V}_0) &= \mathcal{B}_{\mathrm{seq}}(\mathcal{C}, \bar{S}^{\mathrm{net}}_{1:T}, \bar{V}_T),\end{aligned} \tag{4}$$

where $\mathcal{C}$ contains the forward quantities consumed by the reverse scan. When these quantities are retained, the forward scan keeps $\mathcal{C}$ until the reverse scan consumes it. Section 4.2 defines alternative retention and rematerialization policies for supplying $\mathcal{C}$.

The evaluated accelerated kernels cover IF and LIF configurations with fixed parameters; Appendix B specifies backend dispatch, and Appendix D reports forward and backward timing across Torch, CuPy, and Triton.

4.2 Boundary-Local Prefix Rematerialization (SEGMENT-K)

The reverse scan can obtain its required forward quantities under three policies. FULL retains every pre-reset membrane quantity required by Eq. 2. RECOMPUTE retains the initial state and regenerates the needed prefix from the sequence start for each reverse position. SEGMENT-K retains periodic post-reset boundaries in 32-bit floating point (FP32) and regenerates each needed prefix from the left boundary of its segment. Figure 1 illustrates the concrete schedule for $T = 16$ and $K = 4$.

Let K be the segment length, S_V the bytes per membrane state, $Q = \lceil T/K \rceil$ the segment count, and k_j the length of segment j. Let P denote membrane state retained for backward and R the

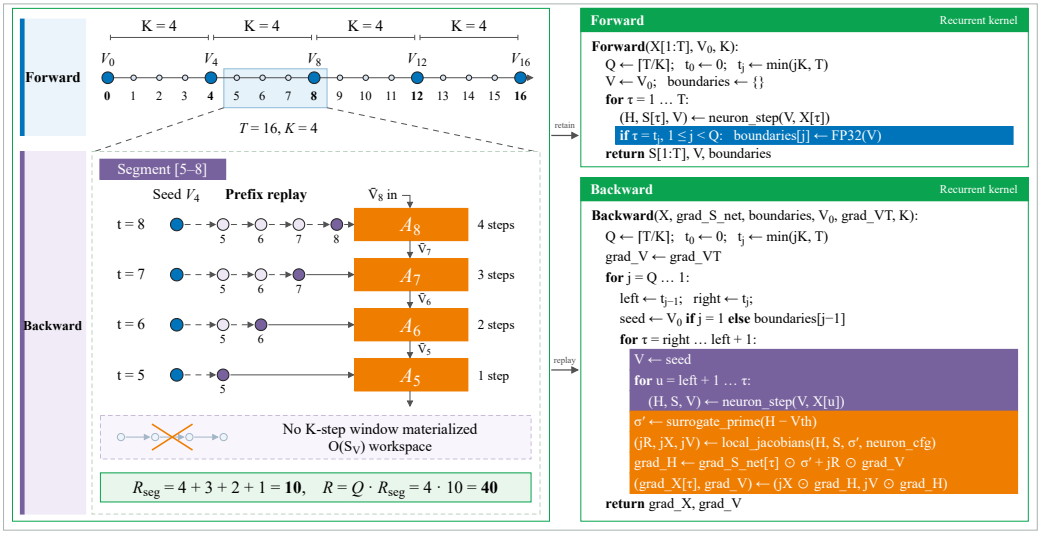

Figure 1: SEGMENT-K rematerialization in the recurrent backward pass. Forward stores periodic FP32 post-reset boundaries; backward replays and consumes the required prefix at each reverse position. $T = 16$, $K = 4$.

backward-only rematerialization updates. The policies obey

$$\begin{aligned} P_{\text{full}} &= \Theta(TS_V), & R_{\text{full}} &= 0, \\ P_{\text{recompute}} &= \Theta(S_V), & R_{\text{recompute}} &= \frac{T(T+1)}{2}, \\ P_{\text{segment}} &= \Theta(QS_V), & R_{\text{segment}} &= \sum_{j=1}^{Q} \frac{k_j(k_j+1)}{2} = \Theta(TK). \end{aligned} \tag{5}$$

SEGMENT-K trades retained recurrent state for additional recurrence work. A segment prefix is scanned through a constant number of live membrane vectors, so logical membrane workspace remains $O(S_V)$ and no K-step membrane trajectory is materialized. Unlike external rematerialization, which re-enters the recurrent module, SEGMENT-K keeps prefix rematerialization and reverse propagation inside the recurrent backward kernel, so recurrent kernel launches do not scale with Q.

Formally, let $K \in \{1, \ldots, T-1\}$, $Q = \lceil T/K \rceil$, $t_0 = 0$, $t_j = \min(jK, T)$, and $I_j = \{t_{j-1} + 1, \ldots, t_j\}$. Backward visits segments in reverse order. Given the left boundary $V_{t_{j-1}}$, local inputs X_{I_j}, upstream spike adjoints $\bar{S}^{\text{net}}_{I_j}$, and adjoint $\bar{V}_{t_j}$ at the right boundary, the recurrent kernel realizes

$$\left(\bar{X}_{I_j}, \bar{V}_{t_{j-1}}\right) = \mathcal{B}_j\left(V_{t_{j-1}}, X_{I_j}, \bar{S}^{\text{net}}_{I_j}, \bar{V}_{t_j}\right). \tag{6}$$

For reverse position $t \in I_j$, $\mathcal{B}_j$ replays only $t_{j-1} + 1{:}t$, applies Eq. 2, and discards the replay state before the next position. Segment j performs $k_j(k_j+1)/2$ rematerialized neuron updates with $O(S_V)$ logical membrane workspace. Starting from a stored boundary, the kernel rematerializes the same forward quantities in exact arithmetic. Appendix C reports finite-precision deviations, and Appendix A derives the rematerialization equivalence and state accounting.

4.3 State-Complete Minibatch Replay (GRAPH-V2)

A matched raw CUDA Graph control captures replay while leaving selected minibatch update-state transitions outside the graph. GRAPH-V2 enlarges the replay boundary to include the corresponding device-side transitions. Replay is state-complete for an update when it encloses every device-side state transition whose value can affect subsequent training state. Learning-rate scheduling remains host controlled. Figure 2 contrasts the raw-graph control with this state-complete replay.

The mutable state in the evaluated update comprises model parameters, batch normalization (BN) running statistics, gradient presence, AdamW state, and automatic mixed precision (AMP) scaler state. Device topology and storage addresses remain fixed, while host learning-rate scheduling writes the next learning rate into persistent device tensors referenced by the captured optimizer.

AdamW distinguishes a materialized zero gradient from `grad=None`. The set of parameters with materialized gradients is part of the capture signature. For parameter θ_i, define the active-gradient set

$$\mathcal{A} = \{i \mid \theta_i.\texttt{grad} \neq \texttt{None}\}. \tag{7}$$

GRAPH-V2 discovers $\mathcal{A}$ with a backward pass that omits the optimizer step, restores the module state from before discovery, and allocates stable gradient buffers for parameters in $\mathcal{A}$. The resulting capture signature preserves both gradient presence and the stable storage expected by the optimizer.

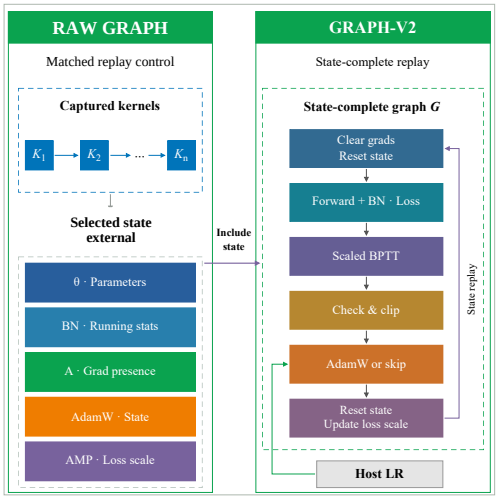

Figure 2: Replay boundaries. The raw-graph control leaves selected update-state transitions external; GRAPH-V2 incorporates them into state-complete replay. $\mathcal{A}$ denotes the active-gradient set.

Changes in the active gradient set require recapture. After each scheduler call, the next learning rate is copied into the persistent device scalar referenced by the captured optimizer. Appendix B specifies capture and the replayed state transitions, while Appendix C reports one-update, long-run, and dynamic-AMP state evolution.

5 Experiments

5.1 Evaluation Protocol and Controls

We evaluate Spikformer (Zhou et al., 2022), QKFormer (Zhou et al., 2024), MaxFormer (Fang et al., 2026), and MS-QKFormer (Fang et al., 2026) under SpikingJelly 2.0.0.dev0 (SJ), SEQUENCE, and GRAPH-V2 on two GPU stacks. Matched conditions use paired runs, and we report paired ratios and uncertainty where applicable. We use 16-bit floating point (FP16), and B denotes the per-run batch size. Stabilized microbenchmarks exclude setup and capture from the timed interval, whereas complete wall time includes initialization, capture, training, validation, checkpointing, and termination. Appendix G specifies the hardware and complete measurement protocols.

The RTX 6000D hosts mechanism studies of recurrent execution, state retention and rematerialization, graph replay, QKV composition, semantic fidelity, and complete training. SpikingJelly 2.0.0.dev0 provides the training baseline, and MemOpt (Huang et al., 2026) provides the memory baseline. Backend substitutions, a raw CUDA Graph control, external rematerialization, and local QKV rewrites vary the execution mechanism under matched workloads. Complete Transformer schedules use the released CIFAR-10 recipes without model-specific retuning (Zhou et al., 2022; 2024; Fang et al., 2026); SJ and GRAPH-V2 share model and optimization settings. ImageNet-1K tests recurrent execution across batch size and temporal horizon with graph replay and local rewrites disabled.

5.2 Architecture, Platform, and Workload Transfer

SEQUENCE is faster than SJ in all eight architecture–platform pairs in Table 1. GRAPH-V2 further increases throughput in every pair, although its incremental gain varies more across architectures and GPUs.

Table 1: CIFAR-10 throughput and speedups across architectures and platforms. All models use depth 4, embedding dimension 384, $T = 4$, FP16 and values are medians of three paired runs.

			Throughput (img/s) ↑			Speedup		
Platform	Model	B	SJ	SEQUENCE	GRAPH-V2	SEQ/SJ	G-V2/SJ	G-V2/SEQ
RTX 6000D	Spikformer	128	1018.8	1503.6	2262.2	1.476×	2.220×	1.505×
	QKFormer	64	546.2	808.9	1536.0	1.481×	2.812×	1.899×
	MaxFormer	128	1207.0	1561.8	1968.2	1.294×	1.631×	1.260×
	MS-QKFormer	128	990.6	1239.4	1426.0	1.251×	1.440×	1.151×
RTX 5090	Spikformer	128	1315.5	1890.3	2331.4	1.437×	1.772×	1.233×
	QKFormer	64	660.5	978.5	1614.8	1.481×	2.445×	1.650×
	MaxFormer	128	1448.4	2015.7	2130.6	1.392×	1.471×	1.057×
	MS-QKFormer	128	1320.1	1524.2	1583.3	1.155×	1.199×	1.039×

On ImageNet-1K, the four architecture families use configurations of approximately 16 million parameters with 224×224 inputs. We sweep batch size at $T = 4$ and temporal horizon at $B = 16$, with three paired runs per condition; graph replay and local rewrites are disabled. SEQUENCE is faster in every comparison; across $T = 2, 4, 8$, the geometric-mean SEQUENCE/SJ speedup ranges from $1.373\times$ to $1.446\times$. Across LIF controls with fixed parameters, backward execution accounts for 73.7–81.2% of the measured recurrent latency reduction. A separate control with fixed product TB leaves SpikingPhoenix FULL nearly constant across $T = 2$–32 while the Torch path grows with the serial temporal loop. Appendix D.4 reports the recurrent-kernel and backend controls. Appendix E reports absolute ImageNet throughput and paired uncertainty.

5.3 Rematerialization Placement and Memory–Throughput Trade-Offs

On spike-element-wise ResNet-18 (SEW-ResNet18), the external-rematerialization control uses the same rematerialization as SEGMENT-K but moves it outside recurrent backward. Both satisfy the numerical criteria in Appendix C; in the Nsight profile at the fixed $T = 64$, $K = 8$ partition, external rematerialization increases temporal launches from 6 to 72.

For the SpikingJelly memory baseline, Table 2 uses the Balanced and Memory operating points selected by MemOpt on the primary Triton backend. Appendix D reports the selected configuration, search time, and full context.

Table 2: Throughput and process memory across recurrent state policies. SEW-ResNet18, $T = 16$, $B = 64$, FP16. Throughput is in img/s; process peak is in GiB.

	SJ		SEQUENCE		GRAPH-V2	
Policy	Thr.↑	Peak↓	Thr.↑	Peak↓	Thr.↑	Peak↓
FULL	770.9	5.686	1052.5	8.643	1138.4	9.332
SEGMENT-2	–	–	1036.2	7.180	1164.3	7.570
SEGMENT-4	–	–	1045.6	6.410	1185.1	6.955
SEGMENT-8	–	–	1057.0	5.889	1196.0	6.956
MemOpt-balanced	531.9	4.666	–	–	–	–
MemOpt-memory	537.6	4.490	–	–	–	–

Within SEQUENCE, increasing the segment length lowers process memory while throughput remains close to FULL. The MemOpt operating points reach lower absolute process memory at substantially lower throughput, placing the methods at different memory–throughput operating points. Under GRAPH-V2, reducing recurrent state no longer produces proportional reductions in the measured process peak, consistent with graph and nonrecurrent storage accounting for a larger share of the live set. Appendix D reports the complete K matrix, uncertainty, placement study, and profiler controls.

5.4 State-Complete Replay and Amortization

We compare eager execution, a raw CUDA Graph control, and GRAPH-V2 on spike-element-wise ResNet-18 (SEW-ResNet18) (Fang et al., 2021). Dynamic AMP checks exercise loss-scale growth, overflow, skipped optimizer steps, and recovery; Appendix F reports the full state traces.

For one-time setup cost C_{setup}, eager latency ℓ_{eager}, and replay latency ℓ_{replay}, the update count that amortizes capture is

$$N_{\text{BE}} = \left\lceil \frac{C_{\text{setup}}}{\ell_{\text{eager}} - \ell_{\text{replay}}} \right\rceil . \tag{8}$$

Table 3: Update latency and graph amortization. SEW-ResNet18, $T = 4$, $B = 128$, FP16, five runs. Latency excludes setup; break-even includes setup.

Execution	Latency (ms)↓	vs. eager (×)↑	vs. raw (×)↑	Setup (s)↓	Break-even updates↓	Peak (GiB)↓
SJ eager	73.984	1.000	–	0	–	4.424
SJ + raw graph	27.809	2.660	1.000	5.47	119	3.230
GRAPH-V2	26.068	2.838	1.067	2.11	45	3.344

The raw-graph control accounts for most of the steady-state latency reduction. GRAPH-V2 adds a smaller steady-state gain while amortizing setup earlier. With dynamic AMP, the overflow, skip, and scaler-recovery trajectory matches eager execution with similar latency and amortization. Appendix F reports the full traces, and Appendix D reports the interaction with recurrent state policies.

5.5 Boundary-Conditioned Local Rewrites

We evaluate two local projection rewrites under the same model parameters and training protocol. Packed-QKV combines compatible projection branches; Token-QK denotes the fused local treatment evaluated only in MS-QKFormer. Appendix E reports the FP32 agreement checks and paired confidence intervals.

Under SEQUENCE, every paired 95% confidence interval for the rewrite ratio lies above unity. Under GRAPH-V2, the intervals for MaxFormer packed-QKV and MS-QKFormer Token-QK lie below unity, while the MS-QKFormer packed-QKV interval includes unity. Changing the enclosing execution mode changes local rewrite gain, including cases that cross unity, as represented by $G(\tau \mid \mathcal{B})$ in Eq. 3.

Table 4: Complete CIFAR-10 training under released recipes. SJ and GRAPH-V2 share model and optimization settings; epochs include cooldown.

Architecture	B	Epochs	SJ wall (h)	GRAPH-V2 wall (h)	End-to-end (×)↑
Spikformer	128	310	4.391	2.089	2.102
QKFormer	64	410	10.965	4.017	2.730
MaxFormer	128	410	4.997	3.094	1.615
MS-QKFormer	128	410	6.042	4.273	1.414

Complete-schedule transfer. Across complete schedules, GRAPH-V2 best accuracy is within 0.33 pp of SJ, and GRAPH-V2 remains faster end to end for all four models. Appendix E reports final and best accuracy together with train-loop speedups; end-to-end wall time additionally includes validation, checkpointing, initialization, and other work outside the accelerated update.

Figure 3 summarizes the placement, composition, replay, and state-retention evidence.

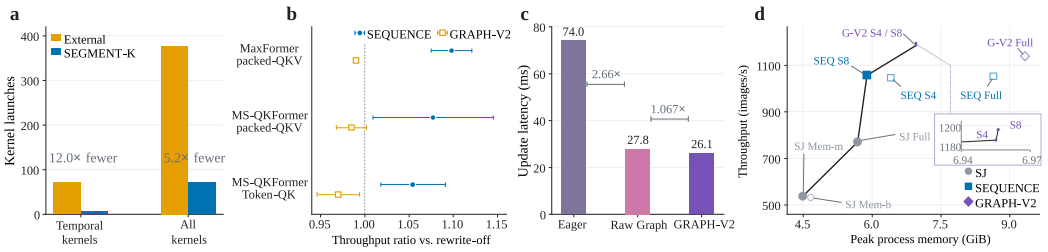

Figure 3: Execution-boundary effects across controlled studies. (a) External versus SEGMENT-K launches. (b) QKV rewrite ratios. (c) Eager, raw-graph control, and GRAPH-V2 update latency. (d) Memory–throughput operating points; SJ denotes SpikingJelly.

5.6 Semantic Fidelity across Boundaries

Let $\mathcal{O}_{\text{num}}$ and $\mathcal{O}_{\text{disc}}$ denote numerical and discrete observables. For each q, let q_{exec} and q_{ref} be its transformed and reference values; for $q \in \mathcal{O}_{\text{num}}$, let D_q be the comparison metric and ϵ_q its tolerance. Semantic agreement holds when

$$\text{Pass} \iff \left[\bigwedge_{q \in \mathcal{O}_{\text{num}}} D_q(q_{\text{exec}}, q_{\text{ref}}) \le \epsilon_q \right] \wedge \left[\bigwedge_{q \in \mathcal{O}_{\text{disc}}} q_{\text{exec}} = q_{\text{ref}} \right]. \tag{9}$$

With dynamic AMP, unchanged model parameters θ and optimizer state o do not imply an unchanged complete update state z: a skipped optimizer step can satisfy

$$(\theta^+, o^+) = (\theta, o) \not\Rightarrow z^+ = z,$$

because the loss-scale state can still change. SEGMENT-K satisfies the criterion in Eq. 9 across FP32, FP16, and BF16; validation spans initial, middle, and terminal temporal positions and the external-reconstruction control, with spike outputs, recurrent states, and input gradients evaluated under data-type-specific tolerances. GRAPH-V2 agrees with eager execution on persistent training state after one update and over long trajectories, including dynamic mixed-precision scale growth, overflow, skipped optimizer steps, and recovery. Appendix C reports the observables, tolerances, and complete case counts.

6 Discussion

Because the transformed executions satisfy the declared semantic criteria, the placement controls isolate execution organization under the same learning rule. Rematerialization costs rise when work crosses the recurrent backward boundary, while a local QKV rewrite can lose its gain under graph replay. Execution boundaries determine which launches and state transitions are amortized together, making an execution boundary a relevant unit of optimization. Reduced recurrent retention also exposes graph, allocator, optimizer, and nonrecurrent storage in process memory. Our evaluation covers fixed-parameter IF and LIF recurrence, fixed tensor shapes, and single-GPU NVIDIA execution; learnable neuron parameters, dynamic shapes, and distributed synchronization add untested state or control transitions.

7 Conclusion

Direct SNN training interleaves temporal recurrence, backward-required state, and persistent minibatch updates. SpikingPhoenix reorganizes these computations and state transitions within recurrent and minibatch execution boundaries while preserving the reference training semantics under the declared criteria. Crossing a boundary changes runtime cost and can reverse the marginal value of a local rewrite. Acceleration persists across architectures, GPUs, ImageNet workload sweeps, and complete CIFAR-10 schedules. Execution placement is a measurable systems variable in direct SNN training, alongside the computation being optimized.

Reproducibility Statement

Appendix G provides complete training and measurement protocols, recorded software versions, configuration details, and the provenance and aggregation procedure for the supplementary traces and artifacts used to produce the reported summaries. Appendix C reports semantic checks for recurrent rematerialization and state-complete device-side replay of the minibatch update.

AI Use Statement

Generative AI tools were used to assist with language editing and code consistency review. They were not used to generate experimental measurements. The authors are responsible for the technical claims, results, citations, and final text.

References

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM international conference on architectural support for programming languages and operating systems, volume 2*, pp. 929–947, 2024.

Felix C Bauer, Gregor Lenz, Saeid Haghighatshoar, and Sadique Sheik. Exodus: Stable and efficient training of spiking neural networks. *Frontiers in Neuroscience*, 17:1110444, 2023.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Rainer Engelken. Sparseprop: Efficient event-based simulation and training of sparse recurrent spiking neural networks. volume 36, pp. 3638–3657, 2023.

Wei Fang, Zhaofei Yu, Yanqi Chen, Tiejun Huang, Timothée Masquelier, and Yonghong Tian. Deep residual learning in spiking neural networks. volume 34, pp. 21056–21069, 2021.

Wei Fang, Yanqi Chen, Jianhao Ding, Zhaofei Yu, Timothée Masquelier, Ding Chen, Liwei Huang, Huihui Zhou, Guoqi Li, and Yonghong Tian. Spikingjelly: An open-source machine learning infrastructure platform for spike-based intelligence. *Science Advances*, 9(40):eadi1480, 2023.

Yuetong Fang, Deming Zhou, Ziqing Wang, Hongwei Ren, Lusong Li, Renjing Xu, et al. Spiking neural networks need high-frequency information. volume 38, pp. 103404–103425, 2026.

Wanjin Feng, Xingyu Gao, Wenqian Du, Hailong Shi, Peilin Zhao, Pengcheng Wu, and Chunyan Miao. Efficient parallel training methods for spiking neural networks with constant time complexity. 2025.

Abhishek Ghosh, Ajay Nayak, Ashish Panwar, and Arkaprava Basu. {GraCE}: Unlocking {CUDA} graphs with compiler support for {ML} workloads. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 1927–1947, 2026.

Kade M Heckel and Thomas Nowotny. Spyx: A library for just-in-time compiled optimization of spiking neural networks. *arXiv preprint arXiv:2402.18994*, 2024.

JiaKui Hu, Man Yao, Xuerui Qiu, Yuhong Chou, Yuxuan Cai, Ning Qiao, Yonghong Tian, Bo Xu, and Guoqi Li. High-performance temporal reversible spiking neural networks with o(l) training memory and o(1) inference cost. In *Forty-first International Conference on Machine Learning*, 2024.

Yifan Huang, Wei Fang, Zecheng Hao, Zhengyu Ma, and Yonghong Tian. Towards lossless memory-efficient training of spiking neural networks via gradient checkpointing and spike compression. In *International Conference on Learning Representations*, volume 2026, pp. 66530–66554, 2026.

Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. volume 2, pp. 497–511, 2020.

Hongyi Jin, Bohan Hou, Guanjie Wang, Ruihang Lai, Jinqi Chen, Zihao Ye, Yaxing Cai, Yixin Dong, Xinhao Cheng, Zhihao Zhang, Yilong Zhao, Yingyi Huang, Lijie Yang, Jinchen Jiang, Gabriele Oliaro, Jianan Ji, Xupeng Miao, Vinod Grover, Todd C. Mowry, Zhihao Jia, and Tianqi Chen. Event tensor: A unified abstraction for compiling dynamic megakernel. In *Proceedings of Machine Learning and Systems*, volume 8, 2026.

Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. volume 33, pp. 8343–8354, 2020.

Yanchen Li, Jiachun Li, Kebin Sun, Luziwei Leng, and Ran Cheng. Towards scalable gpu-accelerated snn training via temporal fusion. In *International Conference on Artificial Neural Networks*, pp. 58–73. Springer, 2024.

Ling Liang, Zhaodong Chen, Lei Deng, Fengbin Tu, Guoqi Li, and Yuan Xie. Accelerating spatiotemporal supervised training of large-scale spiking neural networks on gpu. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 658–663. IEEE, 2022.

Qingyan Meng, Mingqing Xiao, Shen Yan, Yisen Wang, Zhouchen Lin, and Zhi-Quan Luo. Towards memory-and time-efficient backpropagation for training spiking neural networks. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6143–6153. IEEE, 2023.

Korbinian Pöppel, Maximilian Beck, and Sepp Hochreiter. Flashrnn: I/o-aware optimization of traditional rnns on modern hardware. In *International Conference on Learning Representations*, volume 2025, pp. 63895–63919, 2025.

Sumit B Shrestha and Garrick Orchard. Slayer: Spike layer error reassignment in time. volume 31, 2018.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

Timo C Wunderlich and Christian Pehle. Event-based backpropagation can compute exact gradients for spiking neural networks. *Scientific Reports*, 11(1):12829, 2021.

Mingqing Xiao, Qingyan Meng, Zongpeng Zhang, Di He, and Zhouchen Lin. Online training through time for spiking neural networks. volume 35, pp. 20717–20730, 2022.

Chengting Yu, Lei Liu, Gaoang Wang, Erping Li, and Aili Wang. Advancing training efficiency of deep spiking neural networks through rate-based backpropagation. volume 37, pp. 115786–115815, 2024.

Hong Zhang and Yu Zhang. Memory-efficient reversible spiking neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 16759–16767, 2024.

Chenlin Zhou, Han Zhang, Zhaokun Zhou, Liutao Yu, Liwei Huang, Xiaopeng Fan, Li Yuan, Zhengyu Ma, Huihui Zhou, and Yonghong Tian. Qkformer: Hierarchical spiking transformer using qk attention. volume 37, pp. 13074–13098, 2024.

Zhaokun Zhou, Yuesheng Zhu, Chao He, Yaowei Wang, Shuicheng Yan, Yonghong Tian, and Li Yuan. Spikformer: When spiking neural network meets transformer. 2022.

A Neuron Dynamics, Temporal Adjoint, and Rematerialization

This appendix collects the neuron equations, surrogate gradient derivatives, rematerialization analysis conditioned on the execution boundary, and recurrent state and work accounting used in the evaluation. Symbols follow the charge–fire–reset convention used by SpikingJelly: H_t is the membrane potential after charging and before firing, S_t the spike, and V_t the membrane potential after reset (Fang et al., 2023). The evaluated recurrence includes integrate-and-fire (IF) and leaky integrate-and-fire (LIF) neurons.

A.1 IF/LIF Dynamics and Reverse Adjoint

For an LIF neuron with fixed parameters and membrane time constant τ_m, SpikingJelly uses the following discrete charge rule when input decay is enabled and hard reset uses reset potential V_{reset}:

$$\begin{aligned} H_t &= V_{t-1} + \frac{1}{\tau_m}\left[X_t - (V_{t-1} - V_{\text{reset}})\right] \\ &= \left(1 - \frac{1}{\tau_m}\right) V_{t-1} + \frac{1}{\tau_m} X_t + \frac{1}{\tau_m} V_{\text{reset}}. \end{aligned} \tag{10}$$

When input decay is disabled, the implementation uses

$$\begin{aligned} H_t &= V_{t-1} - \frac{1}{\tau_m}(V_{t-1} - V_{\text{reset}}) + X_t \\ &= \left(1 - \frac{1}{\tau_m}\right) V_{t-1} + X_t + \frac{1}{\tau_m} V_{\text{reset}}. \end{aligned} \tag{11}$$

For soft reset, the same implementation uses zero as the charge baseline and then subtracts the threshold after firing. For an IF neuron, $H_t = V_{t-1} + X_t$. Eqs. 10–11 reproduce the evaluated SpikingJelly LIF discretization with fixed parameters. Other LIF formulations may use a distinct leak or rest potential.

The evaluated SpikingJelly implementation uses the arctangent surrogate. With surrogate parameter α, its backward derivative is

$$\sigma'(x) = \frac{\alpha}{2\left[1 + \left(\frac{\pi}{2}\alpha x\right)^2\right]}, \qquad \alpha = 2 \text{ in the main validation.} \tag{12}$$

Forward firing remains the hard threshold Θ. Eq. 12 is used only in backward.

For threshold V_{th}, $S_t = \Theta(H_t - V_{\text{th}})$ and

$$V_t = \mathcal{R}(H_t, S_t) = \begin{cases} (1 - S_t)H_t + S_t V_{\text{reset}}, & \text{hard reset,} \\ H_t - V_{\text{th}} S_t, & \text{soft reset.} \end{cases} \tag{13}$$

Let $\sigma'_t := \sigma'(H_t - V_{\text{th}})$. All reset derivatives below use σ'_t as the surrogate derivative of the spike function. Define the reset Jacobian as $j_t^{\mathcal{R}} := \partial V_t / \partial H_t$. For hard reset with a non-detached reset branch,

$$\begin{aligned} j_t^{\mathcal{R}} &= \frac{d}{dH_t}\left[(1 - S_t)H_t + S_t V_{\text{reset}}\right] \\ &= 1 - S_t + (V_{\text{reset}} - H_t)\sigma'_t, \end{aligned} \tag{14}$$

whereas non-detached soft reset gives

$$j_t^{\mathcal{R}} = \frac{d}{dH_t}(H_t - V_{\text{th}} S_t) = 1 - V_{\text{th}}\sigma'_t. \tag{15}$$

With `detach_reset=True`, the spike is treated as constant only on the reset branch. The four cases are therefore

$$j_t^{\mathcal{R}} = \begin{cases} 1 - S_t + (V_{\text{reset}} - H_t)\sigma'_t, & \text{hard, non-detached reset,} \\ 1 - S_t, & \text{hard, detached reset,} \\ 1 - V_{\text{th}}\sigma'_t, & \text{soft, non-detached reset,} \\ 1, & \text{soft, detached reset.} \end{cases} \tag{16}$$

Thus detaching the reset leaves the forward trajectory unchanged and removes only the derivative through the spike on the reset branch, matching the standard surrogate gradient treatment (Fang et al., 2023; Bauer et al., 2023).

To write backpropagation through time (BPTT) without double-counting the reset path, let $\bar{S}_t^{\mathrm{net}}$ denote the adjoint arriving from network consumers of S_t other than Eq. 13, and let $\bar{V}_t$ denote the total adjoint at the state after reset. All derivatives below are elementwise because the represented IF and LIF recurrence couples time steps through each neuron's own membrane state. Reverse accumulation at the state before firing is

$$\bar{H}_t = \bar{S}_t^{\mathrm{net}} \odot \sigma_t' + j_t^{\mathcal{R}} \odot \bar{V}_t. \tag{17}$$

For the LIF charge rules with fixed parameters above, the local elementwise derivatives are

$$j_t^V := \frac{\partial H_t}{\partial V_{t-1}} = 1 - \frac{1}{\tau_m}, \qquad j_t^X := \frac{\partial H_t}{\partial X_t} = \begin{cases} \tau_m^{-1}, & \text{input decay,} \\ 1, & \text{no input decay.} \end{cases} \tag{18}$$

The corresponding input adjoint is

$$\bar{X}_t := \frac{\partial L}{\partial X_t} = j_t^X \odot \bar{H}_t. \tag{19}$$

If $\bar{V}_{t-1}^{\mathrm{ext}}$ denotes any adjoint injected directly at V_{t-1} outside the temporal recurrence, the total adjoint after reset propagated to the previous time step is

$$\bar{V}_{t-1} = \bar{V}_{t-1}^{\mathrm{ext}} + j_t^V \odot \bar{H}_t, \qquad t = 1, \dots, T, \tag{20}$$

with $\bar{V}_T$ initialized by any external adjoint on the exposed final state before the reverse scan. Intermediate V_t are not exposed by the evaluated operator interface, so $\bar{V}_t^{\mathrm{ext}} = 0$ for $t < T$. Eqs. 17–20 are the explicit form of Eq. 2.

The primary numerical comparison instantiates a LIF neuron with fixed parameters, $\tau_m = 2.0$, input decay enabled, $V_{\mathrm{th}} = 1.0$, hard reset to zero, `detach_reset=False`, the arctangent surrogate in Eq. 12, and full BPTT. Additional supported validations vary reset, detach-reset, and input-decay fields while remaining within the evaluated IF and LIF configurations with fixed parameters. Learnable recurrent parameters such as parametric LIF neurons introduce additional parameter, state, and gradient dependencies and lie outside the evaluated recurrence domain.

A.2 Boundary-Local Temporal Rematerialization

For integer segment length $K \in \{1, \dots, T\}$, define

$$Q = \left\lceil \frac{T}{K} \right\rceil, \qquad t_0 = 0, \qquad t_j = \min(jK, T), \quad j = 1, \dots, Q, \qquad I_j = \{t_{j-1} + 1, \dots, t_j\}. \tag{21}$$

For a fixed input sequence, write the one-step neuron map as $F_t(V) = \mathcal{R}(f(V, X_t), \Theta(f(V, X_t) - V_{\mathrm{th}}))$. Starting from the left segment boundary, repeated application of this deterministic map reconstructs the forward trajectory over that segment in exact arithmetic:

$$V_{t_{j-1}}^{\mathrm{seg}} = V_{t_{j-1}}^{\mathrm{ref}} \quad \Longrightarrow \quad (H_t^{\mathrm{seg}}, S_t^{\mathrm{seg}}, V_t^{\mathrm{seg}}) = (H_t^{\mathrm{ref}}, S_t^{\mathrm{ref}}, V_t^{\mathrm{ref}}), \quad t \in I_j. \tag{22}$$

The rematerialized trajectory therefore supplies the surrogate and reset Jacobians used by the reverse recurrence. Given the upstream network spike adjoints and the adjoint at the right boundary, Eq. 19 determines the local input gradients and adjoint at the left boundary:

$$\begin{gathered} V_{t_{j-1}}^{\mathrm{seg}} = V_{t_{j-1}}^{\mathrm{ref}}, \quad \bar{S}_{I_j}^{\mathrm{net,seg}} = \bar{S}_{I_j}^{\mathrm{net,ref}}, \quad \bar{V}_{t_j}^{\mathrm{seg}} = \bar{V}_{t_j}^{\mathrm{ref}} \\ \Longrightarrow \quad \left(\bar{X}_{I_j}^{\mathrm{seg}}, \bar{V}_{t_{j-1}}^{\mathrm{seg}}\right) = \left(\bar{X}_{I_j}^{\mathrm{ref}}, \bar{V}_{t_{j-1}}^{\mathrm{ref}}\right). \end{gathered} \tag{23}$$

Applying the segment operator in Eq. 6 in reverse segment order recovers the full reverse recurrence in exact arithmetic. FULL, RECOMPUTE, and SEGMENT-K therefore differ in retained state and rematerialization work. Appendix C characterizes deviations caused by finite precision of the implemented paths.

A.3 Retained State and Rematerialization Work

Let S_V denote the byte footprint of one membrane state tensor. FULL retains the temporal trajectory before reset used by its direct reverse scan, so its retained recurrent state term is $\Theta(TS_V)$. RECOMPUTE removes that temporal tensor: for reverse position t, it scans the recurrence from the initial boundary through t, uses the resulting local state for the adjoint update, and discards the scan variables. Its number of rematerialized neuron updates is

$$R_{\text{recompute}} = \sum_{t=1}^{T} t = \frac{T(T+1)}{2}. \tag{24}$$

The logical membrane workspace of this nested scan is $O(S_V)$: it carries only the current scan state and the local state needed by the reverse update, without materializing another length-T membrane trajectory. Required inputs, spike adjoints, and gradient outputs are excluded from this membrane term.

For SEGMENT-K, let the partition in Eq. 21 have segment lengths $k_j = |I_j|$, with $\sum_j k_j = T$ and $k_j \le K$. The implementation stores the $Q-1$ interior membrane states after reset in FP32, up to an additive initial and final state term,

$$P_{\text{segment}} = (Q-1)S_V + O(S_V) = \Theta\left(\frac{T}{K} S_V\right). \tag{25}$$

During reverse processing of a segment, the state required at local reverse position r is regenerated by scanning the corresponding prefix from the stored left boundary. Thus segment j executes

$$R_j = \sum_{r=1}^{k_j} r = \frac{k_j(k_j+1)}{2} \tag{26}$$

rematerialized neuron updates, and the complete backward rematerialization work is

$$R_{\text{segment}} = \sum_{j=1}^{Q} \frac{k_j(k_j+1)}{2} \le \frac{T(K+1)}{2} = O(TK). \tag{27}$$

When all but the final segment have length K, this is $\Theta(TK)$. As in RECOMPUTE, logical membrane rematerialization workspace is $O(S_V)$ because each prefix scan overwrites a constant number of membrane vectors. These terms instantiate Eq. 5, while membrane workspace remains independent of K.

For comparison, conventional checkpointing that materializes a window can retain $\Theta(TS_V/K)$ boundaries while also materializing a $\Theta(KS_V)$ rematerialization window, yielding the familiar $T/K + K$ storage balance (Chen et al., 2016). SEGMENT-K instead consumes each rematerialized prefix immediately, yielding the $O(S_V)$ logical membrane workspace above. The segmented path is used for $K < T$. Configurations without an interior segment boundary use the FULL state path.

The recurrent state terms above are not predictions of the complete process peak. Let τ index execution within one update and decompose its live set schematically as

$$M_{\text{process}} = \max_{\tau} \left[M_{\text{rec}}(\tau) + M_{\text{nonrec}}(\tau) + M_{\text{opt}}(\tau) + M_{\text{graph}}(\tau) + M_{\text{alloc}}(\tau) \right]. \tag{28}$$

Reducing recurrent boundaries can therefore stop reducing NVML peak memory once another live set becomes dominant. This is the regime observed for SEGMENT-4 and SEGMENT-8 under GRAPH-V2 in Appendix D. All reported SEGMENT-K boundaries use FP32 precision.

B Recurrent and Minibatch Execution

This appendix specifies the recurrent state policies and the replay transition over the device-side portion of the optimizer update used by SpikingPhoenix. Their measured memory, placement, and replay effects appear in Appendices D–F.

The accelerated IF kernels and LIF kernels with fixed parameters implement the dynamics in Appendix A. SEQUENCE realizes Eq. 4 through registered Torch operators or accelerated recurrent kernels. The recurrent state policy determines membrane lifetime and rematerialization, while minibatch graph execution determines the repeated issuance boundary and GRAPH-V2 replays the device-side update transaction while preserving persistent training state.

Table 5: Recurrent state policies. S_V is one membrane-state tensor and $Q = \lceil T/K \rceil$; replay updates count backward rematerialization only.

Policy	Retained membrane state	Replay updates	Logical workspace	Backward use
FULL	$\Theta(TS_V)$	0	$O(S_V)$	saved trajectory
RECOMPUTE	$\Theta(S_V)$	$T(T+1)/2$	$O(S_V)$	prefix from sequence start
SEGMENT-K	$\Theta(QS_V)$	$\sum_j k_j(k_j+1)/2$	$O(S_V)$	prefix from segment boundary

B.1 Recurrent Execution and State Policies

FULL, RECOMPUTE, and SEGMENT-K share the recurrence and reverse adjoint in Appendix A. They differ in which membrane states survive forward and in the recurrence work used to supply backward state.

SEGMENT-K implements the operator conditioned on the execution boundary derived in Appendix A: periodic FP32 boundaries persist across forward, while prefix rematerialization and its local adjoint consumption remain inside one recurrent backward kernel. Table 5 summarizes the resulting persistence, replay work, and workspace contracts for all recurrent state policies.

B.2 State-Complete Minibatch Replay

Minibatch graph execution moves repeated device work with fixed shapes from host issuance to replay at stable addresses. GRAPH-V2 uses the device-side portion of one optimizer update as the replay boundary so that persistent device state evolves with the captured work. Captured operations use stable buffers, while scheduler actions ordered by the host complete the logical update around replay.

Before capture, the gradient presence set $\mathcal{A}$ from Eq. 7 must be known because it affects tensor storage and the optimizer transition. If the gradient is absent, a PyTorch-style optimizer skips that parameter,

$$\theta_i.\texttt{grad} = \texttt{None} \Longrightarrow (\theta_i, o_i)^+ = (\theta_i, o_i), \tag{29}$$

where o_i denotes optimizer state for parameter i. A materialized zero gradient instead participates in the optimizer step,

$$\theta_i.\texttt{grad} = \mathbf{0} \Longrightarrow (\theta_i, o_i)^+ = \mathsf{Opt}_i(\theta_i, o_i; \mathbf{0}), \tag{30}$$

which can change decoupled weight decay, moment state, and state associated with optimizer steps under AdamW. A discovery pass without an optimizer step determines $\mathcal{A}$, after which the state from before discovery is restored before capture.

Algorithm 1 summarizes capture and the one-step state comparison.

Algorithm 1 GRAPH-V2 capture and state validation.

Require: reference update $\mathcal{U}_{\text{ref}}$, update state z, batch with fixed shapes (x, y)
$z_0 \leftarrow \textsc{Snapshot}(z)$
$\mathcal{A} \leftarrow \textsc{DiscoverGradientPresence}(x, y, z)$
$z \leftarrow \textsc{Restore}(z_0)$
$b \leftarrow \textsc{AllocateStableGradAndLearningRateBuffers}(\mathcal{A})$
$G \leftarrow \textsc{WarmAndCaptureDeviceTransaction}(x, y, z, b)$
$z \leftarrow \textsc{Restore}(z_0)$
$z_{\text{graph}} \leftarrow \textsc{ReplayDeviceTransaction}(G, x, y, z)$
$z_{\text{graph}} \leftarrow \textsc{ApplyHostScheduler}(z_{\text{graph}})$
$z_{\text{ref}} \leftarrow \mathcal{U}_{\text{ref}}(x, y; z_0)$
if $\textsc{Validate}(z_{\text{graph}}, z_{\text{ref}})$ **then**
    return G
else
    return reference execution
end if

The captured device transaction contains recurrent reset, gradient clearing, forward execution and BatchNorm updates, loss, scaled backward, device unscale and nonfinite detection, optional clipping of the gradient norm, capturable fused AdamW, recurrent reset after the update, and dynamic update of the loss scale. Scheduler ordering remains managed by the host. After each scheduler call, changed learning rate values are copied into the persistent device tensors referenced by the captured optimizer. Random number generator state is restored before discovery and capture when the measured update consumes it. Buffers at stable addresses and graph pools provide replay storage but are not persistent model state.

BatchNorm gives a concrete counterexample to treating a time step replay as a minibatch update. For running mean μ_t^{run}, batch mean at time step t $\widehat{\mu}_t$, and BatchNorm momentum α_{BN}, applying BatchNorm at each time step gives

$$\mu_t^{\mathrm{run}} = (1-\alpha_{\mathrm{BN}})\mu_{t-1}^{\mathrm{run}} + \alpha_{\mathrm{BN}}\widehat{\mu}_t, \qquad t = 1, \dots, T. \tag{31}$$

The evaluated minibatch path flattens $[T, N, C, H, W]$ to $[TN, C, H, W]$ and applies one BatchNorm update. For temporal slices of equal size, $\widehat{\mu}_{1:T} = T^{-1}\sum_{t=1}^{T}\widehat{\mu}_t$, so

$$\begin{aligned} \mu_{\mathrm{ref}}^{+} &= (1-\alpha_{\mathrm{BN}})\mu_0^{\mathrm{run}} + \alpha_{\mathrm{BN}}\widehat{\mu}_{1:T}, \\ \mu_T^{\mathrm{run}} &= (1-\alpha_{\mathrm{BN}})^T\mu_0^{\mathrm{run}} + \alpha_{\mathrm{BN}}\sum_{t=1}^{T}(1-\alpha_{\mathrm{BN}})^{T-t}\widehat{\mu}_t, \qquad \mu_T^{\mathrm{run}} \neq \mu_{\mathrm{ref}}^{+} \text{ in general.} \end{aligned} \tag{32}$$

The two paths weight the initial running mean and observed batch means differently. Tensor operations applied per time step therefore need not preserve the minibatch transition. Running variance follows this difference in update boundaries.

AMP gives a second example. Let $\boldsymbol{g}$ denote unscaled gradients, ξ_{AMP} the scaler state, and $c \in \{\mathrm{finite}, \mathrm{nonfinite}\}$ the check for finite gradients. Suppressing scheduler details,

$$(\theta', o') = \mathsf{Opt}(\theta, o; \boldsymbol{g}), \qquad (\theta^+, o^+, \xi_{\mathrm{AMP}}^+) = \begin{cases} (\theta', o', \mathsf{Scale}_{\mathrm{finite}}(\xi_{\mathrm{AMP}})), & c = \mathrm{finite}, \\ (\theta, o, \mathsf{Scale}_{\mathrm{nonfinite}}(\xi_{\mathrm{AMP}})), & c = \mathrm{nonfinite}. \end{cases} \tag{33}$$

A nonfinite gradient event can therefore leave parameters and optimizer state unchanged while still advancing scaler state:

$$(\theta^+, o^+) = (\theta, o) \not\Rightarrow z^+ = z, \qquad \mathsf{Scale}_{\mathrm{nonfinite}}(\xi_{\mathrm{AMP}}) \neq \xi_{\mathrm{AMP}} \text{ in general.} \tag{34}$$

BatchNorm running state, gradient presence, optimizer state, AMP scaler state, and scheduler state jointly define the complete minibatch transition. GRAPH-V2 places the device portion inside the captured transaction and preserves scheduler ordering around it.

Execution domain. A captured transaction has fixed tensor shapes and a stable gradient participation set. Changes to either define a new graph signature and trigger recapture. The accelerated recurrent kernels cover the evaluated IF configurations and LIF configurations with fixed parameters, while other recurrence and backend combinations use the registered Torch execution path. Appendix G specifies the timing protocols.

C Semantic Fidelity Validation

This appendix reports numerical validation for recurrent rematerialization and replay of the device-side minibatch update transaction. Recurrent tests compare spike outputs, membrane states, and input gradients; replay tests compare persistent training state after the optimizer update. Task behavior over long runs is reported in Appendix E.

C.1 Observables and Numerical Criteria

Numerical agreement follows Eq. 9. Numerical observables use data-type-specific tolerances, while discrete observables require exact agreement. Membrane and recurrent state tensors use maximum absolute difference; input gradients use relative L_2 error; persistent update state uses direct tensor,

counter, or discrete-state comparison. Table 6 records the data-type-specific thresholds and observed maxima.

Table 6: SEGMENT-K numerical validation by data type. Ninety cases, 30 per type; "Grad." is relative input-gradient L_2 error.

Dtype	Tol.	Spike diff.	State max.	Grad. rel. L_2
FP32	2×10^{-6}	0	2.98×10^{-7}	2.77×10^{-7}
FP16	8×10^{-3}	0	4.88×10^{-4}	1.16×10^{-5}
BF16	6×10^{-2}	0	1.95×10^{-3}	3.77×10^{-5}

C.2 Segmented Recurrent Validation

Tests at initial, middle, and terminal simulation positions exercise boundary propagation over different dependency lengths. All tested backend and policy combinations meet the output, final-state, and input-gradient criteria. SpikingPhoenix FULL and SEGMENT-K agree on the recorded observables, while Table 6 reports finite-precision deviations from the Torch baseline. The external checkpoint control extends this comparison to long horizons.

C.3 Long-Run Replay and Dynamic AMP

GRAPH-V2 is compared with eager execution after the complete optimizer update. Checks after one update and over long runs cover outputs, loss, gradients, model and optimizer state, scheduler state, and counters; dynamic mixed precision additionally exercises scale growth, overflow, optimizer skipping, and recovery. The persistent observables agree with eager execution throughout these trajectories. Table 7 reports the coverage and recorded maxima.

Table 7: Numerical validation of recurrent and minibatch state transitions.

Validation	Scope	Cases/ steps	Agreement	Observed result
Typed lowering	mixed	216	216	supported typed specifications accepted
Segmented parity	FP32/16/BF16	90	90	SEGMENT vs. FULL reported differences are zero
Temporal position	mixed	192	192	max input gradient rel. $L_2 = 1.09 \times 10^{-7}$
External checkpoint	FP16	120	120	tested observables meet the recurrent criteria
Replay after one update	4 configurations	4	4	all five output/update deltas $= 0$
Long-run replay	updates 1/10/100/1000	16	16	model/optimizer deltas $= 0$; scheduler state agrees
AMP/control trajectories	10 scenarios	92	92	state at each step agrees through overflow and recovery

The validation covers fixed-parameter IF and LIF recurrence under FULL and SEGMENT-K, and GRAPH-V2 replay with the fixed shapes and persistent update state used in the reported experiments.

D State Retention and Rematerialization Placement

Table 2 reports the complete $T = 16$ state–execution operating points. Here we analyze the interaction between state policy and replay, the memory baseline context, rematerialization placement,

and recurrent kernel attribution on the primary RTX 6000D platform. All SpikingPhoenix results use version 1.0.8.31.

D.1 State Policy under Graph Replay

Table 8 crosses recurrent state policy with minibatch execution. Let $R_{p,g}^{(r)}$ be throughput in paired run r, where $p \in \{\mathrm{FULL}, K\}$ denotes the state policy and $g \in \{\mathrm{seq}, \mathrm{graph}\}$ the execution mode. The ratio of graph to sequence throughput within each policy is

$$\rho_{p,\mathrm{G/S}}^{(r)} = \frac{R_{p,\mathrm{graph}}^{(r)}}{R_{p,\mathrm{seq}}^{(r)}}. \tag{35}$$

To measure whether graph gain changes under segmentation, define the multiplicative interaction factor

$$\Gamma_K^{(r)} = \frac{\rho_{K,\mathrm{G/S}}^{(r)}}{\rho_{\mathrm{FULL,G/S}}^{(r)}} = \frac{R_{K,\mathrm{graph}}^{(r)} / R_{K,\mathrm{seq}}^{(r)}}{R_{\mathrm{FULL,graph}}^{(r)} / R_{\mathrm{FULL,seq}}^{(r)}}. \tag{36}$$

Thus $\Gamma_K = 1$ corresponds to multiplicative separability. Across segment sizes, the interaction factors remain close to one; only the $K = 4$ interval lies entirely above one. For $K = 8$, process memory falls by 25.5% relative to FULL+GRAPH-V2 while throughput remains slightly higher, so recurrent-state materialization still contributes to the complete process peak after replay.

Table 8: State-policy and replay interaction at $T = 16$. 95% CIs.

	Graph effect	**Interaction**
K	Gain CI	Ratio CI
2	1.124 [1.047,1.201]	1.039 [0.998,1.080]
4	1.134 [1.062,1.206]	1.048 [1.033,1.063]
8	1.132 [1.113,1.150]	1.046 [0.998,1.094]

D.2 Memory Baseline

SpikingJelly 2.0.0.dev0 MemOpt provides the memory baseline on the primary Triton backend. Both selected profiles use the same checkpoint/compression structure and no temporal split. Their lower process memory comes with markedly lower throughput than Phoenix SEGMENT-8; Table 9 records the operating points and configuration time.

Table 9: MemOpt operating points on the Triton backend. Throughput is relative to SJ.

Profile	Config. time (s)	NVML (GiB) ↓	Throughput/SJ (×) ↑
Balanced	3.86	4.666	0.690
Memory	60.05	4.490	0.697

The rematerialization-placement study uses the same temporal partition and changes only where rematerialization executes.

D.3 Rematerialization Placement

A pair is eligible for the recurrent mechanism summary when both candidates have coefficient of variation (CV) $\leq 5\%$ and $P_{90}/P_{10} \leq 1.10$; Appendix G specifies the complete aggregation protocol. Table 10 extends the direct $K = 8$ comparison to $T = 64$. Both executions satisfy this criterion at $T = 16$ and $T = 32$. The SEGMENT-8 $T = 64$ point remains faster in median latency but has larger variation across runs, so the geometric mean mechanism summary uses the stable $T = 16$ and $T = 32$ pairs.

Table 11 reports every placement pair at long horizons. All pairs satisfy the FP16 recurrent criteria, while four also satisfy the timing stability threshold used for aggregation across runs. Repeated module reentry drives the large latency gap in this stress test. SEGMENT-K retains more peak allocated memory because it stores periodic recurrent boundaries. Figure 4 pairs these placements with the Nsight launch profile.

A separate Nsight profile at $T = 64, K = 8$ localizes the rematerialization placement gap to execution structure. Here $Q = T/K = 8$ temporal segments are applied to $N_{\mathrm{mod}} = 3$ recurrent modules.

Table 10: SEGMENT-8 latency across temporal horizons. FP16, batch 64, five runs; speedup is SpikingJelly/SEGMENT-8. Eligibility requires CV $\leq 5\%$ and $P_{90}/P_{10} \leq 1.10$.

		Latency		**Stability**		**Memory**	
T	Execution	Median (ms)↓	Pair speedup	CV	P_{90}/P_{10}	Peak alloc. (MiB)↓	Eligible
16	SpikingJelly 2.0.0.dev0 full	16.560	–	1.7%	1.038	68.68	✓
16	SEGMENT-8	10.201	1.623	3.6%	1.081	69.78	✓
32	SpikingJelly 2.0.0.dev0 full	16.216	–	0.6%	1.013	71.68	✓
32	SEGMENT-8	10.340	1.568	1.1%	1.020	73.56	✓
64	SpikingJelly 2.0.0.dev0 full	16.370	–	1.7%	1.036	77.68	✓
64	SEGMENT-8	10.325	1.585	8.4%	1.165	81.12	–

Table 11: Reconstruction placement across T and K. Allocation ratio is SEGMENT-K/external checkpointing; eligibility requires CV $\leq 5\%$ and $P_{90}/P_{10} \leq 1.10$.

		Latency (ms)		**Relative cost**		
T	K	External checkpoint	SEGMENT-K	Speedup (×)↑	Allocation ratio↓	Eligible
128	2	681.95	10.383	65.68	1.242	✓
128	4	349.67	10.515	33.25	1.207	–
128	8	189.38	10.287	18.41	1.187	–
128	16	103.87	10.234	10.15	1.171	–
256	2	1309.44	10.412	125.77	1.401	–
256	4	687.92	10.286	66.88	1.350	✓
256	8	349.55	10.216	34.22	1.320	–
256	16	191.66	10.328	18.56	1.300	–
512	2	1726.48	10.349	166.82	1.598	–
512	4	1351.88	10.394	130.06	1.531	✓
512	8	667.53	10.124	65.93	1.493	–
512	16	359.17	10.522	34.14	1.468	✓

Checkpointing outside the recurrent module invokes each module once in the original forward, once in checkpoint replay, and once in temporal backward for every segment. In contrast, SEGMENT-K keeps the segment loop inside one forward and one backward temporal kernel per module:

$$N_{\text{launch,temp}}^{\text{external}} = 3N_{\text{mod}}Q = 72, \qquad N_{\text{launch,temp}}^{\text{segment}} = 2N_{\text{mod}} = 6. \tag{37}$$

The profile matches the predicted launch structure. At this fixed partition, moving rematerialization outside the recurrent boundary increases temporal launches from 6 to 72, while total profiled step launches rise from 72 to 377. Expressing the temporal rematerialization through repeated module re-entry incurs substantially more launch work.

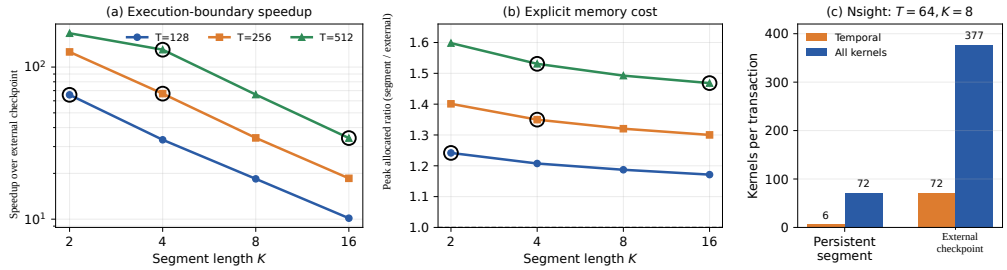

Figure 4: External versus recurrent rematerialization. Panels (a)–(b) report speedup and peak allocation across T and K; rings mark pairs satisfying the stability criterion. Panel (c) reports Nsight launches at $T = 64, K = 8$.

D.4 Recurrent Kernel Controls

We additionally measure a forward–backward workload using LIF with fixed parameters with $N_{\text{state}} = 2^{20}$ FP32 recurrent states and $T \in \{8, 16, 32, 64, 128\}$. Each point uses five independent runs, 20 warm-up iterations, and 50 individually synchronized retained repetitions. Table 12 compares the Torch, CuPy, and Triton implementation families. The distribution across SpikingPhoenix Triton runs is bimodal at $T = 16$ and broader at $T = 32$, so those points are reported individually and no cross-process aggregate is formed.

Table 12: LIF forward/backward latency across five runs. Parameters are fixed; entries are medians of run medians, and the final column is the Phoenix Triton CV.

	Torch (ms)		**CuPy (ms)**		**Triton (ms)**		
T	SJ	Phoenix	SJ	Phoenix	SJ	Phoenix	Phoenix Triton CV
8	13.335	13.038	4.947	4.083	4.813	2.387	1.36%
16	24.902	24.017	4.987	4.030	4.757	1.188	41.83%
32	42.246	41.304	5.045	4.206	4.896	2.469	17.38%
64	78.680	75.711	5.851	4.885	5.738	3.187	0.33%
128	260.838	250.049	7.532	6.150	7.469	5.864	2.17%

The split timing in Table 13 localizes most of the absolute latency reduction to backward execution. For reference latency L^{ref} and Phoenix latency L^{phx}, the reported backward share is

$$\text{Share}_{\text{bwd}} = \frac{L^{\text{ref}}_{\text{bwd}} - L^{\text{phx}}_{\text{bwd}}}{(L^{\text{ref}}_{\text{fwd}} + L^{\text{ref}}_{\text{bwd}}) - (L^{\text{phx}}_{\text{fwd}} + L^{\text{phx}}_{\text{bwd}})}. \tag{38}$$

Across the five temporal lengths, backward contributes 73.7–81.2% of the measured reduction in split timing. Because forward and backward kernels are optimized together, this statistic localizes the gain to a component of execution rather than to one isolated kernel rewrite.

Table 13: Triton forward/backward attribution over five runs.

T	Forward speedup	Backward speedup	Backward share
8	1.478	2.419	81.2%
16	2.590	4.336	74.7%
32	1.453	2.160	81.1%
64	1.590	1.819	74.2%
128	1.333	1.424	73.7%

D.5 ResNet and Temporal Scaling Controls

A Spiking ResNet-18 workload provides a model control for temporal scaling beyond the three-block recurrent benchmark. The timed region is `zero_grad`→reset→forward→cross-entropy→backward

Table 14: Spiking ResNet-18 temporal scaling at batch size 8. Timings exclude the optimizer; speedup is relative to SpikingJelly, and memory is PyTorch peak allocation.

	Latency (ms) ↓			**Speedup vs. SJ** (×)		**Peak allocated (GiB)** ↓		
T	SJ	FULL	RECOMP.	FULL	RECOMP.	FULL	RECOMP.	Rec./FULL
2	62.155	41.159	40.982	1.510	1.517	0.553	0.418	0.756
4	62.376	40.840	40.979	1.527	1.522	0.936	0.664	0.709
8	62.084	41.225	40.984	1.506	1.515	1.707	1.159	0.679
16	70.323	44.526	43.859	1.579	1.603	3.248	2.152	0.663
32	80.804	64.907	–	1.245	–	6.329	–	–

and excludes the optimizer step. Each aggregate point contains five independent runs. Table 14 reports the FP16 results at fixed shapes with explicitly selected FULL and RECOMPUTE policies. RECOMPUTE remains close to FULL in latency for $T \leq 16$ while requiring less peak allocation. The $T = 32$ row is available only for FULL under this workload.

A control with fixed TB keeps TB constant and separates temporal recurrence from growing sample count. Under FP16, SpikingPhoenix FULL varies by less than 1% across $T = 2$–32, while Torch execution grows with the serial temporal loop. The geometric mean speedups for the individual backends are $0.990\times$ for Torch, $1.149\times$ for CuPy, and $1.469\times$ for Triton, placing the measured advantage in the accelerated backend paths. A separate inference probe over a long temporal horizon with stepwise execution reverses the ordering, restricting the recurrent advantage to the evaluated training execution.

E Transfer and Complete-Training Evidence

This appendix complements the reported operating points with training repeated across seeds, paired architecture–platform uncertainty, correspondence with complete schedules, composition of local architecture rewrites, and the RECOMPUTE endpoint.

E.1 SEW-ResNet Repeated-Seed Training

Each mode uses one SEW-ResNet18 (Fang et al., 2021) training recipe, $T = 4$, batch size 128, FP16, cosine learning rate schedule, and seeds 340–342. Both SpikingPhoenix modes use FULL recurrent state materialization. The effective throughput statistic is computed from total processed training images divided by total training seconds. Table 15 reports the resulting per-seed records; this rate includes first-epoch graph setup and differs slightly from the arithmetic mean of epoch throughput.

Table 15: SEW-ResNet18 training over 100 epochs for three seeds. Effective throughput includes setup; NVML reports process peak memory.

		Accuracy (%)		**Training cost**		**Memory**
Mode	Seed	Final	Best	Effective img/s ↑	Train time (min) ↓	NVML peak (GiB) ↓
SpikingJelly	340	88.31	88.35	1668	49.87	3.484
	341	87.51	87.90	1690	49.22	3.434
	342	88.18	88.36	1697	49.03	3.484
SpikingPhoenix FULL SEQUENCE	340	87.56	88.09	2494	33.36	4.926
	341	87.07	87.59	2488	33.45	4.922
	342	88.59	88.98	2477	33.58	5.059
SpikingPhoenix FULL GRAPH-V2	340	88.05	88.37	4595	18.11	6.021
	341	86.99	87.25	4611	18.04	5.986
	342	87.17	87.27	4600	18.09	6.053

Across seeds, both Phoenix modes reduce training time, while run-to-run accuracy variation exceeds the mean paired shift. GRAPH-V2 further reduces sustained time but raises process peak memory. Figure 5 plots the corresponding time–accuracy trajectories.

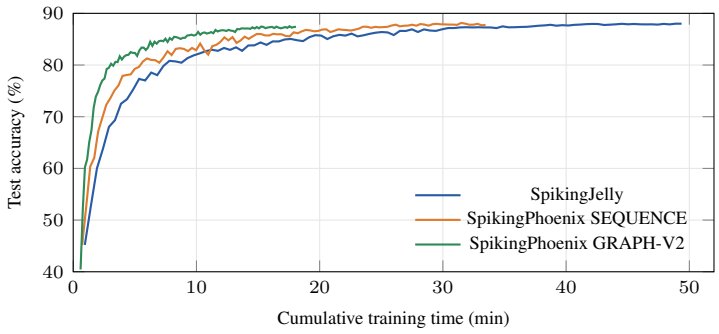

Figure 5: SEW-ResNet18 accuracy versus cumulative training time over three seeds. Curves are epoch aligned; horizontal separation reflects training-time differences.

E.2 Architecture–Platform Paired Statistics

Table 1 reports absolute throughput and point estimates for both GPU platforms and software stacks. Table 16 reports the paired 95% confidence intervals used to quantify runtime variation across runs. QKFormer uses batch size 64. The remaining architectures use batch size 128. All conditions use $T = 4$, FP16 with AMP, FULL recurrent state, and local architecture rewrites disabled.

Table 16: Paired speedup intervals across architectures and platforms. Effects use two-sided 95% CIs.

		SEQ/SJ	**GRAPH-V2/SJ**	**GRAPH-V2/SEQ**
Platform	Architecture	× CI	× CI	× CI
RTX 6000D	Spikformer	1.476 [1.448,1.504]	2.221 [2.167,2.274]	1.505 [1.485,1.524]
	QKFormer	1.481 [1.413,1.549]	2.812 [2.738,2.886]	1.899 [1.859,1.939]
	MaxFormer	1.294 [1.256,1.332]	1.631 [1.598,1.663]	1.260 [1.238,1.283]
	MS-QKFormer	1.251 [1.205,1.298]	1.439 [1.421,1.458]	1.151 [1.119,1.182]
	Geo. mean	1.372 –	1.957 –	1.427 –
RTX 5090	Spikformer	1.437 [1.211,1.663]	1.772 [1.768,1.776]	1.237 [1.048,1.425]
	QKFormer	1.482 [1.354,1.610]	2.446 [2.293,2.598]	1.651 [1.525,1.777]
	MaxFormer	1.392 [1.375,1.408]	1.471 [1.460,1.482]	1.057 [1.051,1.063]
	MS-QKFormer	1.155 [1.083,1.227]	1.200 [1.130,1.270]	1.039 [1.029,1.049]
	Geo. mean	1.360 –	1.663 –	1.224 –

The recurrent gain is nearly unchanged across the two qualified stacks, whereas the incremental GRAPH-V2 gain is more stack dependent. Table 16 contains the paired uncertainty and geometric mean across architectures. Raw repetitions from each run are retained in the corresponding experiment artifacts.

E.3 ImageNet-1K Workload Transfer

We additionally evaluate transfer of recurrent execution on the ImageNet-1K training workload using 224×224 inputs and matched configurations of approximately 16 million parameters for the four Transformer architecture families. Spikformer, QKFormer, MaxFormer, and MS-QKFormer are evaluated under matched SJ and SEQUENCE execution in two sweeps: batch size $B \in \{4, 8, 16, 32\}$ at $T = 4$, and temporal horizon $T \in \{2, 4, 8\}$ at fixed $B = 16$. Each condition uses three paired runs with seeds 340–342, 50 warm-up updates, and 200 measured complete optimizer updates. Graph replay and local architecture rewrites are disabled, leaving recurrent execution as the varying implementation.

Table 17 reports architecture scale through parameter count together with absolute GPU compute and end-to-end operating points measured over the timed phase. SEQUENCE improves every architecture–batch condition, and the end-to-end measurements closely track the GPU compute result across all four ImageNet architecture families.

We next hold $B = 16$ fixed and vary $T \in \{2, 4, 8\}$ under the same paired-run protocol. Table 18 reports transfer across temporal horizon and the corresponding absolute reduction in update time.

The independent $T = 4$ condition reproduces the operating point from the batch sweep. Across temporal horizons, every architecture remains above unity while the absolute reduction in complete update latency changes only modestly. SEQUENCE therefore remains faster across the tested temporal horizons without a fixed or T-proportional multiplicative gain. Combined with the batch sweep, the same advantage appears across architecture, batch size, and temporal horizon on ImageNet-1K.

Table 17: ImageNet-1K transfer across batch size. $T = 4$, three paired runs; columns report absolute SJ/SEQUENCE throughput, speedup, two-sided 95% Student-t CIs, and E2E throughput.

Architecture scale			**GPU compute (img/s) ↑**				**Measured E2E (img/s) ↑**		
Architecture	Params (M)	B	SJ	SEQ	SEQ/SJ	CI	SJ	SEQ	SEQ/SJ
Spikformer	16.83	4	19.96	29.93	1.500	[1.417,1.583]	19.91	29.83	1.499
		8	40.44	57.42	1.420	[1.385,1.455]	40.32	57.17	1.418
		16	78.47	111.77	1.424	[1.356,1.493]	78.07	110.96	1.421
		32	155.19	220.20	1.419	[1.382,1.457]	153.80	217.48	1.414
QKFormer	16.47	4	16.48	24.76	1.503	[1.433,1.572]	16.45	24.69	1.501
		8	32.66	49.08	1.503	[1.433,1.573]	32.58	48.90	1.501
		16	66.31	97.23	1.466	[1.423,1.510]	66.03	96.62	1.463
		32	129.71	173.58	1.338	[1.310,1.366]	128.73	171.85	1.335
MaxFormer	16.24	4	18.89	27.45	1.453	[1.403,1.503]	18.85	27.36	1.451
		8	37.94	55.10	1.452	[1.418,1.486]	37.83	54.87	1.450
		16	74.97	106.19	1.417	[1.335,1.498]	74.61	105.45	1.414
		32	150.05	203.91	1.359	[1.317,1.401]	148.75	201.52	1.355
MS-QKFormer	16.47	4	16.99	25.49	1.500	[1.439,1.561]	16.96	25.42	1.499
		8	34.14	49.47	1.449	[1.384,1.514]	34.05	49.28	1.447
		16	68.70	96.94	1.413	[1.252,1.573]	68.40	96.33	1.410
		32	133.86	176.19	1.316	[1.286,1.347]	132.78	174.41	1.314

Table 18: ImageNet-1K transfer across temporal horizons. $B = 16$; speedup is SEQUENCE/SJ, with paired 95% confidence intervals. Geo. mean and saved time summarize each temporal horizon.

T	Architecture	Speedup (×)	95% CI	Geo. mean (×)	Saved (ms/update)
2	Spikformer	1.433	[1.389,1.477]	1.446	67.9
	QKFormer	1.474	[1.422,1.526]		
	MaxFormer	1.417	[1.377,1.457]		
	MS-QKFormer	1.461	[1.418,1.503]		
4	Spikformer	1.440	[1.328,1.552]	1.426	66.6
	QKFormer	1.447	[1.388,1.506]		
	MaxFormer	1.365	[1.222,1.508]		
	MS-QKFormer	1.454	[1.412,1.495]		
8	Spikformer	1.387	[1.356,1.419]	1.373	63.1
	QKFormer	1.368	[1.305,1.430]		
	MaxFormer	1.380	[1.350,1.410]		
	MS-QKFormer	1.357	[1.300,1.414]		

E.4 Complete-Schedule Diagnostics

The complete schedules remain on the primary RTX 6000D platform and compare short-protocol estimates with sustained acceleration through complete optimization schedules. Training-loop ratios track the short protocol closely; end-to-end speedups are lower because validation, checkpointing, initialization, and other work remain outside the accelerated update. Table 19 compares these three scopes; GRAPH-V2 remains in replay through overflow and scaler recovery.

Table 19: Short-protocol and complete-schedule correspondence. Entries are GRAPH-V2/SJ speedups.

Architecture	Short	Training loop	E2E
Spikformer	2.221	2.189	2.102
QKFormer	2.812	2.815	2.730
MaxFormer	1.631	1.645	1.615
MS-QKFormer	1.439	1.429	1.414

Table 4 reports complete wall time and accuracy.

E.5 QKV Rewrite Crossover

The composition study re-enables the local QKV and Token-QK rewrites under SEQUENCE and GRAPH-V2. MaxFormer and MS-QKFormer use CIFAR-10, $T = 4$, batch size 128, FP16 with AMP, FULL recurrent state materialization, and generic Phoenix fusion disabled. Each treatment uses three independent runs with cyclically rotated order. Local effect ratios are paired within each run and summarized with two-sided 95% Student-t CIs.

Packed-QKV combines compatible projection branches under fixed model parameters and training protocol. MS-QKFormer additionally evaluates a fused Token-QK treatment. Table 20 reports the paired effects; the FP32 fused Token-QK comparison shows exact output agreement and negligible Q/K-gradient deviation.

Table 20: QKV/Token-QK rewrite crossover at $T = 4$. Effects and 95% CIs compare each rewrite with *off* within the same execution mode.

Model	Execution	Local rewrite	img/s ↑	Effect (×)	CI
MaxFormer	SEQUENCE	off	1568.1	–	–
		packed-QKV	1721.5	1.098	[1.075,1.121]
	GRAPH-V2	off	1970.2	–	–
		packed-QKV	1950.6	0.990	[0.989,0.991]
MS-QKFormer	SEQUENCE	off	1224.5	–	–
		packed-QKV	1317.9	1.077	[1.009,1.145]
		fused Token-QK	1291.0	1.054	[1.018,1.091]
	GRAPH-V2	off	1428.1	–	–
		packed-QKV	1406.4	0.985	[0.968,1.002]
		fused Token-QK	1385.2	0.970	[0.946,0.994]

Under SEQUENCE, all tested rewrites improve over the corresponding off treatment. Under GRAPH-V2, two rewrites become significantly adverse and the remaining packed-QKV treatment is statistically indistinguishable from off. The marginal value of the local rewrite changes with the enclosing execution mode. Launch, layout, memory traffic, and device compute contributions remain aggregated in these end-to-end measurements.

E.6 RECOMPUTE Endpoint

At the shorter $T = 4$ horizon, RECOMPUTE approaches the SpikingJelly process memory point while retaining most of the FULL throughput gain. Table 21 quantifies this endpoint. The $T = 16$ SEGMENT-K study in Appendix D evaluates bounded rematerialization over a longer temporal sequence.

Table 21: RECOMPUTE endpoint on SEW-ResNet18 at $T = 4$. Throughput ratios use paired 95% CIs; NVML reports process peak memory.

Policy	img/s ↑	vs. SJ CI	NVML (GiB) ↓	Mem./SJ
SpikingJelly	1653.5	1.000 –	3.482	1.000
Phoenix FULL	2563.5	1.550 [1.478,1.622]	4.910	1.410
Phoenix RECOMPUTE	2467.6	1.493 [1.463,1.524]	3.538	1.016

F Graph Replay Cost and Amortization

Table 3 reports eager, raw graph, and GRAPH-V2 update latency. This appendix adds graph memory decomposition, setup amortization, and the dynamic AMP extension. Appendix C reports the corresponding validation of update state.

F.1 Graph Memory Decomposition

The memory probe resets allocator peaks after setup, executes one replay, and then measures 20 steady replays. Table 22 separates peak allocation, steady live allocation, allocator reservation, and NVML process peak. GRAPH-V2 adds about 63–66 MiB of steady live allocation relative to SEQUENCE. The larger increase at the process level tracks allocator reservation and storage private to the graph.

Table 22: SEQUENCE and GRAPH-V2 memory decomposition at batch size 128. The upper block reports allocator memory; the lower block reports NVML peak, GRAPH-V2/SEQUENCE ratio, and capture time.

		Peak allocated		**Current allocated**		**Reserved**	
Architecture	Local transform	SEQ	GRAPH-V2	SEQ	GRAPH-V2	SEQ	GRAPH-V2
MaxFormer	off	7.365	7.531	0.168	0.233	8.719	8.900
MaxFormer	packed-QKV	7.380	7.542	0.169	0.232	8.674	9.193
MS-QKFormer	off	8.693	8.862	0.171	0.234	9.959	11.344
MS-QKFormer	fused Token-QK	8.805	8.969	0.170	0.234	10.090	10.758

		NVML process peak (GiB)		**Ratio**	**Capture**
Architecture	Local transform	SEQ	GRAPH-V2	GRAPH-V2/SEQ	ms
MaxFormer	off	9.414	9.664	1.027	136.31
MaxFormer	packed-QKV	9.369	9.957	1.063	117.48
MS-QKFormer	off	10.822	12.107	1.119	151.58
MS-QKFormer	fused Token-QK	10.785	11.521	1.068	131.50

The graph memory effect changes with architecture and local rewrite. Fused Token-QK lowers the MS-QKFormer graph reservation relative to its off configuration, whereas packed-QKV increases MaxFormer reserved graph memory while changing steady live allocation little. Complete training NVML peaks also include DataLoader activity, validation, static minibatch buffers, and allocator history, so allocator statistics and process peaks describe different storage boundaries.

F.2 Setup Amortization and Dynamic AMP

Table 3 establishes the replay hierarchy. This appendix derives the setup break-even criterion in Eq. 8 and extends GRAPH-V2 to dynamic mixed precision, including scaler growth, overflow, optimizer skipping, and recovery. For N updates with equal work, replay amortizes setup when

$$C_{\text{setup}} + N\ell_{\text{replay}} \le N\ell_{\text{eager}}$$
$$N \ge \frac{C_{\text{setup}}}{\ell_{\text{eager}} - \ell_{\text{replay}}}, \tag{39}$$

Table 24: Qualified platform stacks. RTX 6000D is the mechanism-study platform; RTX 5090 is the transfer stack.

Platform	GPU	Python	PyTorch	CUDA	Triton	Phoenix	SpikingJelly source
Primary	RTX 6000D	3.14.6	2.13.0+cu132	13.2	3.7.1	1.0.8.31	2.0.0.dev0
Secondary	GeForce RTX 5090	3.12.5	2.12.0+cu132	13.2	3.7.0	1.0.8.31	frozen v2 runtime port

where C_{setup}, ℓ_{eager}, and ℓ_{replay} are defined with the main-text criterion. Taking the smallest integer N gives Eq. 8. Substitution yields the break-even counts reported in Table 3 and Table 23; setup and latency use consistent time units.

Minibatch graph execution supplies the dominant steady-state latency gain relative to host-issued execution, while GRAPH-V2 defines the device-side update transaction and reduces the updates needed to amortize setup. Dynamic AMP preserves the replayed update through overflow, skipped steps, and scaler recovery with only a small latency and setup change.

Table 23: GRAPH-V2 mixed-precision update on SEW-ResNet18. $T = 4$, $B = 128$, five runs.

Execution	Latency (ms)	vs. eager (×)	Setup (s)	Break-even
Autocast	26.068	2.838	2.11	45
Dynamic AMP	26.147	2.844	2.47	52

G Platforms, Protocols, and Reproducibility

G.1 Platform Provenance and Measurement Boundaries

The detailed mechanism studies and complete schedules use the primary NVIDIA RTX 6000D platform. The transfer study across architecture families is additionally replicated on an NVIDIA GeForce RTX 5090 with a distinct software stack. Table 24 records the two software environments. Changes in gain magnitude across platforms are interpreted as transfer across the complete stack rather than effects of hardware alone because the Python, PyTorch, and Triton versions differ slightly.

On RTX 5090, SpikingJelly provenance is tied to the frozen SpikingJelly v2 source and runtime port manifest, which also records the symmetric model port transformations. The documented QKFormer correction that reshapes the temporal axis before LIF is applied in both execution ports.

Memory is reported at three boundaries: PyTorch peak allocated memory, allocator reserved memory, and NVML process peak. Microbenchmark latency after stabilization excludes setup and capture, which are reported separately. Complete process wall time includes initialization, capture, training, validation, checkpointing, and termination. Training loop time sums the training intervals for each epoch and excludes subsequent validation.

G.2 Benchmark Protocol Matrix

Within each paired comparison, execution modes share architecture, data pipeline, optimization, precision, and random seed unless the named study axis changes that field. Complete Transformer schedules reproduce the released CIFAR-10 recipes without retuning. ImageNet sweeps vary one workload axis at a time. Studies of state policy and local architecture rewrites vary only the named treatment. Table 25 gives the exact workload, temporal horizon, batch size, repetition count, precision, and measurement window for every study.

Table 25: Benchmark protocol matrix. Warm-up and measurement units are study specific.

Study	Platform	Configuration	Runs / window
Architecture family	6000D, 5090	4 Transformers; $T = 4$; $B = 64/128$; FP16	3; 3 stabilized + 3 measured
ImageNet batch	6000D	4 Transformers, 224^2; $T = 4$; $B = 4/8/16/32$; FP16	3; 50 / 200 updates
ImageNet temporal	6000D	4 Transformers, 224^2; $T = 2/4/8$; $B = 16$; FP16	3; 50 / 200 updates
State matrix	6000D	SEW-ResNet18; $T = 16$; $B = 64$; FP16	3; 20 / 100 updates
Fixed LIF	6000D	2^{20} states; $T = 8$–128; FP32	5; 20 / 50 iterations
Graph replay	6000D	SEW-ResNet18; $T = 4$; $B = 128$; FP16	5; fixed updates
QKV rewrite	6000D	MaxFormer, MS-QKFormer; $T = 4$; $B = 128$; FP16	3; 3 stabilized + 3 measured
Complete schedules	6000D	4 Transformers; $T = 4$; $B = 64/128$; FP16	1 pair; 310/410 epochs
SEW long train	6000D	SEW-ResNet18; $T = 4$; $B = 128$; FP16	3 seeds; 100 epochs

G.3 Paired Statistics and Stability Criteria

For positive measurements a_r and b_r in paired run r, the paired ratio is

$$\rho_r = \frac{a_r}{b_r}. \tag{40}$$

Throughput ratios place the candidate over the baseline, latency speedups place the baseline over the candidate, and memory ratios place the candidate over the baseline. A memory reduction is reported as $1 - M_{\text{candidate}}/M_{\text{baseline}}$. For n paired ratios, the two-sided 95% Student-t confidence interval is

$$\bar{\rho} \pm t_{0.975,n-1}\frac{s_\rho}{\sqrt{n}}, \qquad \bar{\rho} = \frac{1}{n}\sum_{r=1}^{n}\rho_r. \tag{41}$$

When positive mechanism ratios are summarized across temporal points, their geometric mean is used.

The CV and P_{90}/P_{10} stability filter applies only to the controlled recurrent mechanism summaries. Each run contributes its median latency, and a pair is eligible when both candidates satisfy CV $\leq 5\%$ and $P_{90}/P_{10} \leq 1.10$. Results from the short protocol for the architecture families are reported from all paired runs without this filtering.

G.4 Complete-Schedule Throughput Protocol

For the 100-epoch SEW study, effective throughput over the training loop is

$$R_{\text{train}} = \frac{\sum_{e=1}^{E} N_e}{\sum_{e=1}^{E} t_e}. \tag{42}$$

This rate divides total processed work by measured training time, includes GRAPH-V2 setup in the first epoch, and excludes the subsequent test phase. Complete Transformer schedules use one shared-seed execution pair per architecture and the released recipes. Epoch counts include the original 10-epoch cooldown: 310 for Spikformer $(300 + 10)$ and 410 for QKFormer, MaxFormer, and MS-QKFormer $(400 + 10)$. GRAPH-V2 is captured at the first update and then replayed under FULL with dynamic AMP.

G.5 Reproducibility Artifact

The experiment artifact is maintained separately from the manuscript source package and contains source/protocol snapshots, per-run configurations, timing and memory traces, normalized summaries, and aggregation scripts. Raw ArchitectureFamily repetitions for both platforms, local rewrite runs, and complete-schedule epoch logs remain in the corresponding result packages; the manuscript reports the paired summaries derived from these records.