SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2026, Vol. 69, Iss. 9, 192104:1–192104:18
https://doi.org/10.1007/s11432-025-5044-3

MPL-schedule: a dynamic priority-based scheduling scheme for spiking neural networks

Xiaofan ZHAO[1], Xin DU[2*], Di YU[1], Linshan JIANG[3], Gang PAN[1] & Shuiguang DENG[1*]

[1]*College of Computer Science and Technology, Zhejiang University, Hangzhou 310000, China*
[2]*School of Software Technology, Zhejiang University, Ningbo 315000, China*
[3]*Institute of Data Science, National University of Singapore, Singapore 117602, Singapore*

Received 1 June 2025/Revised 28 December 2025/Accepted 7 February 2026/Published online 31 August 2026

Abstract Spiking neural networks (SNNs) are gaining attention for energy-efficient, event-driven computing due to their ability to encode information through discrete spikes over multiple time steps. Their use of a biologically plausible neuron model makes them well-suited for asynchronous and sparse data processing in edge workloads. However, this temporal and stateful behavior introduces burst-driven execution patterns and inter-step dependencies, posing significant challenges for GPU-based deployment. Existing GPU schedulers, optimized for dense and feedforward DNNs, fail to fully exploit the irregularity and sparsity of SNNs, leading to poor resource utilization and unpredictable latency under dynamic workloads. To address these challenges, we propose MPL-schedule, a multi-preemptive, priority-aware scheduling scheme that dynamically coordinates heterogeneous SNN tasks on GPUs. Our approach formulates the scheduling process as a reward maximization problem under resource constraints, integrating adaptive priority computation, overhead-aware time-slicing, and state-preserving preemption. We evaluate MPL-schedule across diverse scenarios, including task complexities, spike arrival patterns, and time-slice configurations. Compared to state-of-the-art baselines, our method improves throughput by up to 15.0% and energy efficiency by up to 20.3%, while sustaining over 95.8% GPU utilization.

Keywords task scheduling, spiking neural networks, adaptive time-slicing, GPU preemption, trace-driven evaluation

Citation Zhao X F, Du X, Yu D, et al. MPL-schedule: a dynamic priority-based scheduling scheme for spiking neural networks. Sci China Inf Sci, 2026, 69(9): 192104, https://doi.org/10.1007/s11432-025-5044-3

1 Introduction

Modern data-centric systems—from cloud platforms to edge devices—increasingly rely on machine learning workloads that demand low latency and high energy efficiency [1]. While deep neural networks (DNNs) dominate many such applications, their dense and continuous activation patterns result in high power consumption and limited responsiveness under tight resource budgets. These limitations have spurred interest in spiking neural networks (SNNs), which encode information as discrete temporal spikes [2, 3]. SNNs, considered the third generation of neural models, leverage sparse, event-driven spike communication and bio-plausible dynamics like leaky integrate-and-fire (LIF) to capture temporal correlations in data streams [4]. However, their temporal behavior—spanning multiple simulation steps with evolving membrane potentials—poses unique challenges to system-level scheduling and GPU-based deployment [5].

Unlike conventional DNNs, which benefit from workload regularity and support static execution plans, SNNs require schedulers to accommodate bursty, highly irregular execution patterns tied to spike events. The execution load of each task may fluctuate significantly based on recent spiking activity, leading to tightly coupled neuron states and variable temporal dependencies [6]. This makes naive GPU scheduling strategies—such as round-robin kernel launches or static job queues—ill-suited for SNN tasks. In practice, failing to respond to these fluctuations leads to low GPU utilization, increased context-switching overhead, and degraded responsiveness under concurrent SNN loads.

However, existing GPU resource management frameworks typically designed for traditional workloads face difficulties in accommodating the irregular and latency-sensitive nature of spike-driven computations [7]. Common techniques, such as dynamic optimizations or generic priority-based scheduling, improve performance for mixed workloads but often fail to provide the precise timing control critical to SNN execution. These challenges intensify when multiple concurrent SNN tasks compete for GPU resources, each with distinct computational demands,

* Corresponding author (email: xindu@zju.edu.cn, dengsg@zju.edu.cn)

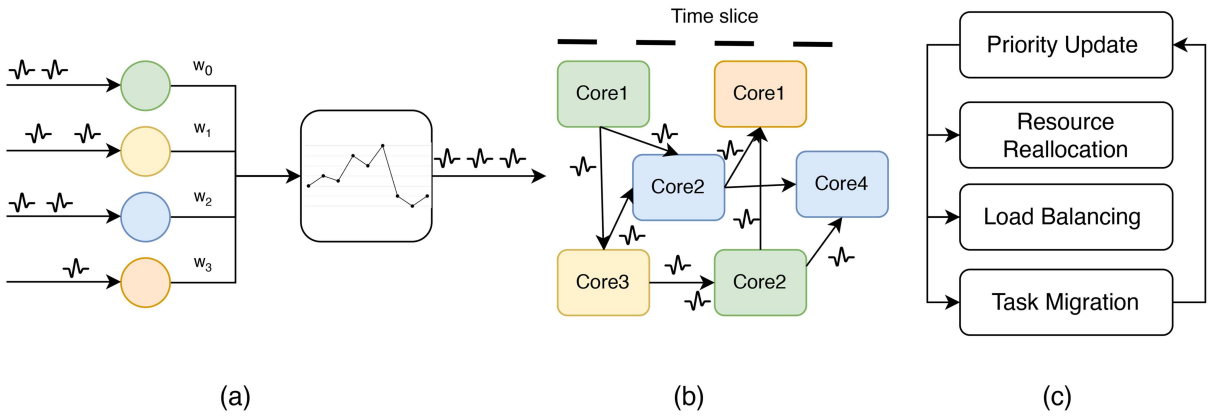

Figure 1 (Color online) Overview of SNN-based GPU task scheduling. (a) Neuron state updates. Tasks receive input, update their states, and trigger execution. (b) Task propagation. Tasks are propagated to different GPU cores or time slices with scheduling delays. (c) Plasticity adaptation. Tasks undergo dynamic priority adjustment and resource reallocation based on current states.

latency constraints, and resource contention patterns [8]. Frequent preemptions and context switching exacerbate overheads, making efficient SNN scheduling even more difficult [9].

To address these challenges, we propose MPL-schedule, a multi-preemptive, priority-based scheduling scheme specifically tailored for coordinating heterogeneous SNN tasks on GPU platforms. MPL-schedule explicitly accounts for the unique characteristics of SNN workloads—event-driven spikes and stateful neuron dynamics. In SNNs, computation triggered by spike arrivals leads to highly irregular yet latency-critical execution patterns, significantly influencing scheduling decisions [10]. By dynamically responding to spike arrival times, MPL-schedule minimizes idle periods and maximizes GPU utilization. Moreover, as neuron membrane potentials and internal states dictate firing behavior, the scheduler continually tracks evolving computational requirements, dynamically adjusting task priorities to maintain timing constraints and optimize resource allocation [4]. Figure 1 illustrates our scheduling scheme, highlighting three key stages: (a) neuron state updates; (b) task propagation; (c) plasticity adaptation. This cyclic workflow induces sparse event-driven computations and tight temporal dependencies, as commonly emphasized in simulation neuroscience and spiking network modeling [11, 12]. Our contributions are summarized as follows.

(1) We present a unified modeling and optimization scheme tailored to heterogeneous SNN workloads. By framing scheduling as an online reward maximization problem, it jointly considers deadline urgency, task complexity, energy cost, context-switching overhead, and adjustment latency.

(2) We design an adaptive time-slicing mechanism with lightweight state-preservation routines and theoretical guarantees. By adjusting slices in response to task dynamics and historical feedback, MPL-schedule achieves a balance between responsiveness and efficiency while minimizing preemption overhead.

(3) We conduct extensive experimental evaluations using realistic SNN workloads inspired by typical data-intensive and latency-sensitive analytics scenarios. The evaluations demonstrate that MPL-schedule consistently achieves improved throughput, energy efficiency, and GPU resource utilization compared to existing scheduling methods.

2 Background and related work

This work builds upon prior research in GPU-based SNN simulation and heterogeneous system scheduling. Below, we review representative studies and highlight the gaps that motivate MPL-schedule.

GPU-based SNN simulation. Simulating SNNs on GPUs is challenging due to sparse, event-driven, and temporally correlated spike activity, which induces irregular control flow and memory access patterns. Fidjeland et al. [13, 14] proposed NeMo and improved kernel and memory layouts for large-scale SNN simulation on GPUs. Subsequent work further explored GPU-side optimizations for spike-driven dynamics and scalability, including high-performance SNN simulation on GPGPUs [15], GPU-cluster SNN simulation frameworks [16], and system-level hybrid CPU-GPU simulation for neuromorphic computing chips under architectural constraints [17]. Knight et al. [18] provided an extensive performance/energy evaluation, showing that GPU-based simulation can be competitive with existing HPC and neuromorphic solutions for large cortical models.

High-level simulator interfaces such as PyNN [19] and PCSIM [20] improve usability and portability, but do

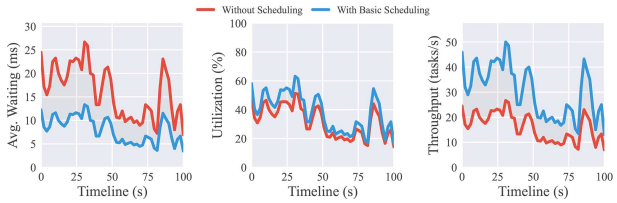

Figure 2 (Color online) Performance comparison between basic scheduling and non-scheduling scenarios under Poisson patterns.

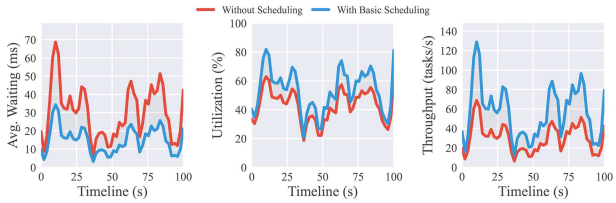

Figure 3 (Color online) Performance comparison between basic scheduling and non-scheduling scenarios under bursty patterns.

not target adaptive runtime GPU resource management across concurrent spike-driven workloads. Overall, prior GPU-SNN efforts establish the feasibility and benefits of acceleration, yet largely overlook adaptive scheduling mechanisms that explicitly model spike-driven progress variability and stateful preempt-resume overhead in multi-tenant settings.

Scheduling in heterogeneous systems. Task scheduling in heterogeneous systems ranges from traditional real-time policies such as EDF [21,22] and SRTF (often formalized/analyzed as SRPT) [23] to DAG-based mapping heuristics such as HEFT [10]. These classical methods typically assume stable, predictable task costs, which can break down for event-driven workloads whose activity fluctuates at run time.

Recent GPU scheduling systems target multi-tenant accelerators with improved isolation and latency control. Gandiva [24] enables fast suspend-resume and time-sharing for deep-learning workloads. TimeGraph [7] provides real-time GPU scheduling support, and Chimera [9] proposes collaborative preemption for multitasking on shared GPUs to control and bound preemption overhead. More recent work has revisited GPU preemption mechanisms and practical context-switch overhead. For example, GPreempt improves the generality and efficiency of GPU preemption, and Orion provides interference-aware, fine-grained GPU sharing for ML applications [25,26]. However, these systems are not designed for spike-driven SNN workloads and do not model task progress and checkpoint state evolution under temporally sparse and bursty executions.

To address these gaps, we propose MPL-schedule, a runtime that monitors real-time spike activity, preserves neuron states across preemptions, and allocates GPU resources using overhead-aware policies. Unlike prior work, it sustains high utilization under bursty loads while meeting service-level objectives for SNN-based applications.

3 Challenge and motivation

Using GPUs for dynamic, event-driven SNNs presents significant challenges due to their temporal irregularity and state-dependent behavior. Unlike conventional machine learning models, SNNs rely on temporal neuron states and discrete spike events, making their computational requirements highly unpredictable and dependent on prior interactions. This time-aware behavior creates substantial difficulties for traditional scheduling algorithms, which assume more stable workloads with predictable execution patterns [27].

A key challenge lies in the highly variable nature of SNN workloads, driven by bursty spikes and fluctuating task complexities. For instance, a burst of spikes can suddenly amplify computational demands, leading to temporary GPU overload and severe resource contention. Without adaptive scheduling, such dynamics can lead to resource underuse or overload. Furthermore, context-switching costs in SNN tasks—due to the necessity of saving and restoring detailed neuron states, spike buffers, and synaptic parameters—add considerable overhead, complicating efficient scheduling [15].

To demonstrate why adaptive scheduling policies are critical for managing SNN workloads, we conducted preliminary experiments under two representative workload patterns: Poisson arrivals and bursty arrivals. Figure 2 (Poisson) and Figure 3 (bursty) compare system performance between scenarios with and without a basic scheduling mechanism. In each figure, the three panels from left to right report waiting time, GPU resource utilization, and throughput.

In the absence of scheduling, the system exhibits clear symptoms of inefficiency and performance instability. Waiting time becomes markedly larger, with bursty arrivals producing the most pronounced spikes, while basic scheduling reduces the average waiting time by about 50% under both patterns. Resource usage is also less regulated without coordination: GPU utilization frequently drops below 40%, indicating substantial underuse even when tasks are present, whereas basic scheduling raises the average utilization by roughly 10% points and mitigates utilization drops. The throughput results further confirm the severity of this degradation. Without scheduling, the average throughput is about 29.55 tasks/s under Poisson arrivals and 33.74 tasks/s under bursty arrivals, while basic

scheduling increases it to approximately 39.93 and 45.68 tasks/s, respectively. Since both schemes are evaluated under identical time-varying arrivals, their time series naturally follow the same workload-driven trend, and the scheduling benefit mainly manifests as a shifted operating point and suppressed congestion spikes.

These results indicate that uncoordinated execution can simultaneously cause queue buildup and resource underutilization, and that workload variability directly translates into unstable end-to-end performance. The fact that even a basic scheduler yields substantial gains motivates the need for a more responsive and adaptive scheduling framework that can continuously track workload dynamics and mitigate performance degradation under fluctuating SNN workloads [4].

4 Problem formulation

Scheduling heterogeneous SNN tasks presents challenges due to dynamic resource demands and temporal complexity. We develop a unified model capturing both system constraints and task dynamics, and extract key structural properties to guide algorithm design.

4.1 System model

Consider a GPU cluster with M nodes, denoted by $\mathcal{G} = \{G_1, G_2, \ldots, G_M\}$. Each node G_m manages multiple resource types through a capacity vector $\boldsymbol{C}_m = \left[C_m^{(1)}, C_m^{(2)}, \ldots, C_m^{(K)}\right]$, where each element represents a specific resource (compute cores, memory bandwidth). This model supports fine-grained control and reflects GPU heterogeneity.

The system evolves in discrete time slots $t \in \{1, 2, \ldots, T\}$, where T denotes the total scheduling horizon. At each time slot t, resource allocations are determined using two key decision variables.

- $x_{ijm}(t) \in \{0, 1\}$. Indicates whether task T_i uses resource type j on node m.
- $R_{ijm}(t) \geqslant 0$. Specifies the amount of resource type j allocated to task T_i on node m at time t.

In these definitions, $i \in \{1, \ldots, N\}$ indexes the tasks, $j \in \{1, \ldots, K\}$ indexes the resource types, and $m \in \{1, \ldots, M\}$ indexes the nodes. Resource allocations must satisfy

$$\sum_{i=1}^{N} x_{ijm}(t)\, R_{ijm}(t) \;\leqslant\; C_m^{(j)}, \quad \forall\, j,\, m,\, t. \tag{1}$$

Additionally, hardware limitations create coupling between resources

$$\phi_{j,k}\big(R_{ijm}(t),\, R_{ikm}(t)\big) \;\leqslant\; \kappa_{j,k}, \quad \forall\, i,\, j,\, k,\, m,\, t, \tag{2}$$

where $\phi_{j,k}(\cdot)$ captures resource interactions and $\kappa_{j,k}$ bounds their combined usage.

4.2 LIF task dynamics and workload characteristics

LIF task behavior follows the event-driven dynamics of the Leaky integrate-and-fire model ((A1); see Appendix A). Each task $T_i \in \mathcal{T}$ emits a spike once its membrane potential $V_i(t)$ surpasses the firing threshold V_{th}, after which it resets to V_{reset}. This spiking behavior leads to highly variable task loads, significantly complicating resource scheduling in multi-tenant GPU environments.

To quantify each task's computational load, we note that spiking activity triggers both membrane updates and synaptic operations. We model this as

$$\tau_i(t) \;=\; \alpha_i \sum_{n=1}^{N_i} S_i^n(t) \;+\; \beta_i \sum_{j=1}^{N_i} \sum_{k=1}^{N_i} w_{jk}^i\, S_i^j(t), \tag{3}$$

where N_i is the number of neurons in task T_i, $S_i^n(t)$ is the spike indicator of the n-th neuron (1 if a spike occurs at time t, 0 otherwise), and w_{jk}^i represents the synaptic weight from neuron j to neuron k in task T_i. The constants α_i and β_i capture scaling factors for membrane updates and synaptic updates, respectively, making $\tau_i(t)$ an overall measure of computational load at time t. Here, (α_i) and (β_i) capture membrane and synaptic updates, respectively. The dynamic nature arises from changing spike patterns and network activity, creating three key characteristics that influence scheduling.

- Time-varying resource demands due to spike-driven computation.
- Non-uniform workload distribution across neurons.

- Strong temporal dependencies between execution steps.

These properties reflect key behaviors of SNN workloads: fluctuating computational intensity, carryover states, and mixed firing synchrony. As such, any static resource allocation policy is likely to struggle with either oversubscription or underutilization at different times [8]. Hence, an adaptive scheduling strategy becomes essential to dynamically manage the evolving demands of spiking tasks within GPU environments.

4.3 Optimization problem

We formulate scheduling as a constrained reward-maximization problem balancing progress and overhead

$$\max_{x,\,\boldsymbol{R}} \quad \sum_{t=1}^{T_{\max}} \sum_{i\in\mathcal{A}(t)} \Big[\gamma_i(t)\, f_i\big(R_i(t)\big) \;-\; O_i(t)\Big] \tag{4a}$$

$$\text{s.t.} \quad \sum_{i=1}^{N} x_{ijm}(t)\, R_{ijm}(t) \;\leqslant\; C_m^{(j)}, \quad \forall\, j,\, m,\, t, \tag{4b}$$

$$\phi_{j,k}\Big(R_{ijm}(t),\, R_{ikm}(t)\Big) \;\leqslant\; \kappa_{j,k}, \quad \forall\, i,\, j,\, k,\, m,\, t, \tag{4c}$$

$$\sum_{m=1}^{M} \sum_{t=1}^{T} x_{ijm}(t) \;\geqslant\; \tau_i, \quad \forall\, i,\, j, \tag{4d}$$

$$x_{ijm}(t) \in \{0,1\}, \quad R_{ijm}(t) \geqslant 0, \quad \forall\, i,\, j,\, m,\, t. \tag{4e}$$

Here, $f_i\big(R_i(t)\big)$ measures task progress under allocated resources, while $O_i(t)$ captures overhead from context switching and communication. The priority gradient $\gamma_i(t)$ enables dynamic adaptation to changing conditions. We let $T_{\max}$ be an upper bound on the total scheduling horizon, and $\mathcal{A}(t)$ be the set of active tasks at time t. This formulation explicitly accounts for resource constraints (4b), hardware coupling (4c), and task completion requirements (4d).

In the implementation, $O_i(t)$ is not a fixed constant but a runtime estimate derived from measured switching costs. When task T_i is preempted, and its context has been saved and restored at time t, the scheduler measures the actual switching cost $O_i^{\text{switch}}(t)$ for this preemption and updates a per-task estimate $O_i(t)$ using (5)

$$O_i(t) \leftarrow (1-\lambda)\, O_i(t^-) + \lambda\, O_i^{\text{switch}}(t). \tag{5}$$

$O_i(t^-)$ is the previous estimate for task T_i, and $\lambda \in (0,1]$ is a small smoothing factor that controls how fast new observations override the history. The updated $O_i(t)$ then enters the objective in (4a) as the overhead term for task T_i. We define the measured switching cost $O_i^{\text{switch}}(t)$ as the elapsed time of one preemption-resume cycle: saving the task state to the state map, enforcing the required synchronization, restoring the saved state, and relaunching the kernel to resume execution. In practice, $O_i^{\text{switch}}(t)$ is obtained by placing lightweight timestamps at the start and end of this save-sync-restore-relaunch sequence. Therefore, the measured overhead naturally varies with the task state size and the instantaneous memory-bandwidth pressure.

This measurement-driven interface also supports other spiking neuron models beyond LIF. In practice, instantiating a different neuron model mainly requires recalibrating the task-specific statistics that the scheduler consumes, such as the checkpoint footprint, the per-step execution cost, and the switching cost of a preempt-resume cycle measured with the same boundary. The scheduling logic remains unchanged, while these calibrated statistics may shift across neuron models and implementations. When the observed switching cost becomes non-negligible, the controller tends to use longer time slices to avoid excessive preemptions.

4.4 Insights from problem structure

The problem exhibits structural properties that guide algorithm design. First, the presence of binary variables $x_{ijm}(t)$ makes the problem non-convex and NP-hard [28]. This rules out classical optimization methods and necessitates more sophisticated approaches.

Analyzing the temporal structure of the problem yields critical insights. While decisions in one time slot affect future states through task progress and overhead, they do not completely determine future allocations. This partial decoupling suggests that online learning methods could be effective, particularly when combined with careful state tracking.

The coupling of resources through constraints (4c) creates complex interdependencies that must be managed dynamically. However, the problem also exhibits beneficial sparsity properties—in any time slot, only a subset of tasks can be active due to resource limits. This sparsity can be exploited for efficient implementation. Together with SNN stochasticity, these properties motivate the following scheduling principles.

- Dynamic priority computation to handle time-varying tasks.
- Adaptive time-slicing to balance responsiveness and overhead.
- Online optimization to learn effective allocation patterns.
- Explicit handling of resource coupling constraints.

The next section presents our MPL-schedule algorithm, which incorporates these insights while maintaining practical efficiency. Our design explicitly addresses the theoretical challenges identified here through a combination of priority-driven scheduling and online learning.

5 MPL-schedule design

As shown in Section 4, effective SNN scheduling demands both theoretical guarantees and practical efficiency. We first present the theoretical foundations of MPL-schedule and then explain how they guide its design choices.

5.1 Core design principles and theoretical basis

MPL-schedule optimizes task allocation based on theoretical principles that balance performance objectives and system constraints. We derive these properties analytically and use them to inform our system design.

Consider the gradient of our objective function at time t:

$$\nabla q(x(t), y(t)) = \begin{cases} x_l(t)((f_r^{k^*})'(y_{(l,r)}^{k^*}(t)) - \beta_{k^*}), & k = k^*, \\ x_l(t)(f_r^k)'(y_{(l,r)}^k(t)), & \text{otherwise.} \end{cases} \tag{6}$$

This gradient structure leads to our first key theorem.

Theorem 1 (Optimality conditions)**.** At optimality, resource allocations $R^*(t)$ and scheduling indicators $x^*(t)$ must satisfy

$$\gamma_i(t)\nabla f_i(R_i^*(t)) = \sum_j \lambda_m^j(t) + \sum_k \mu_{j,k}(t)\nabla\phi_{j,k}(R_{ijm}^*(t), R_{ikm}^*(t)), \tag{7}$$

where $\lambda_m^j(t)$ and $\mu_{j,k}(t)$ are Lagrange multipliers for resource and coupling constraints.

Proof. These conditions follow from the Lagrangian's first-order stationarity. Due to non-convexity, they are necessary but not sufficient for global optimality.

These optimality conditions motivate three core principles in our algorithm design: priority computation should balance resource demand and coupling constraints, resource allocation should follow gradient directions when feasible, and overhead from state changes should be explicitly accounted for.

Based on these principles, we implement priority computation as

$$\gamma_i(t) = \alpha \cdot \frac{1}{d_i - t + \epsilon} + \beta \cdot \frac{w_i}{\tau_i^{\text{rem}}(t) + \epsilon} - \delta \cdot E_i(t). \tag{8}$$

Here, α, β, δ are tunable weights that control the balance among deadline urgency $(d_i - t)$, remaining work (τ_i^{rem}), and energy usage (E_i). Here, the small constant ϵ is introduced to prevent numerical instability from division by zero. The three weights control the trade-off among urgency (α), remaining workload (β), and energy penalty (δ). Increasing α prioritizes tight-slack tasks to reduce deadline misses, increasing β favors draining backlogged heavy tasks to improve throughput, and increasing δ penalizes energy-hungry executions to improve energy efficiency, potentially at the cost of longer latency. In practice, these objectives may conflict, so the weights are tuned to the expected mix of latency-sensitive and energy-sensitive workloads. The convergence properties of this approach are guaranteed.

Lemma 1 (Step size bound)**.** For time step t and feasible allocation $y(t) \in \mathcal{Y}$:

$$\|y(t+1) - y^*\|^2 \leqslant \|y(t) - y^*\|^2 - 2\eta_t \nabla J(y(t))^{\mathrm{T}}(y(t) - y^*) + \eta_t^2 \|\nabla J(y(t))\|^2, \tag{9}$$

where y^* is the optimal solution and η_t is the step size.

This lemma lets us bound the distance to optimality after each update. The bound guides our choice of step size η_t, which we set as

$$\eta_t = \frac{\operatorname{diam}(\mathcal{Y})}{\|\nabla q(x(t), y(t))\|\sqrt{T}}. \tag{10}$$

These theoretical results form the foundation for our practical scheduling mechanisms, which we detail in the following sections.

5.2 Priority computation and resource allocation

The analysis yields a two-phase scheduling process: task priorities are computed using (8), followed by resource allocation guided by those priorities.

To efficiently address non-convexity, resource allocation is divided into two steps. First, the initial allocation is performed in proportion to task priorities

$$R_{ijm}^{\text{init}}(t) = C_m^{(j)} \cdot \frac{\gamma_i(t)}{\sum_{k\in\mathcal{A}(t)} \gamma_k(t)}. \tag{11}$$

Then, allocations are refined via gradient ascent under coupling constraints

$$R_{ijm}(t) = \Pi_{\mathcal{Y}} \left[R_{ijm}^{\text{init}}(t) + \eta_t \nabla q(x(t), y(t)) \right]. \tag{12}$$

The projection $\Pi_{\mathcal{Y}}$ enforces feasibility. Lemma 1 provides a convergence bound for this refinement step.

Theorem 2 (Convergence rate)**.** With step size $\eta_t = \frac{\eta}{\sqrt{t}}$ where $\eta = \frac{\operatorname{diam}(\mathcal{Y})}{G\sqrt{T}}$, the allocation error decreases as

$$R_T \leqslant \operatorname{diam}(\mathcal{Y}) G \sqrt{T}. \tag{13}$$

To accelerate convergence, the refinement is parallelized across resource types and structured into three components: a priority update using (8), an initial allocation based on the updated priorities, and a parallel refinement step that follows the gradients. This design enables rapid adaptation to workload dynamics while preserving theoretical guarantees. Parallelization across resource types yields $O(\log(K))$ time per iteration, enhancing scalability.

5.3 Adaptive time-slicing and state management

Efficient execution of heterogeneous SNN tasks on GPUs requires managing the interplay of temporal dependencies, workload variability, and resource constraints: each simulation step depends on prior states and newly emitted spikes that propagate downstream, producing uncertainty and time-varying workloads; different neuron groups exhibit highly uneven activity, with some generating dense spikes that demand intensive computation while others remain largely inactive; and finite GPU compute and memory resources must be dynamically shared across tasks with diverse latency, energy, and complexity requirements.

These factors necessitate adaptive scheduling that updates priorities in real time, adjusts time slices to shifting demands, and balances preemption overhead against performance gains. Adaptive time-slice management is essential for handling variable SNN workloads. We propose a time-slicing mechanism guided by theoretical bounds and empirical demands.

For each task T_i, the time slice $\theta_i(t)$ is computed as

$$\theta_i(t) = \theta_{\min} + (\theta_{\max} - \theta_{\min}) \mathrm{e}^{-\kappa\sigma_i(t)}, \tag{14}$$

where $\sigma_i(t)$ is the slack ratio, and κ controls adaptation speed. This mechanism has important theoretical properties.

Lemma 2 (Time-slice monotonicity)**.** The adaptive time-slice $\theta_i(t)$ decreases monotonically with slack ratio $\sigma_i(t)$:

$$\frac{\partial\theta_i(t)}{\partial\sigma_i(t)} = -(\theta_{\max} - \theta_{\min})\kappa \mathrm{e}^{-\kappa\sigma_i(t)} \leqslant 0. \tag{15}$$

This property ensures urgent tasks receive longer time slices, improving deadline adherence. To quantify the efficiency of this mechanism, we define the scheduling overhead ratio

Definition 1 (Overhead ratio)**.** The scheduling overhead ratio $\rho_i(t)$ for task T_i at time t is

$$\rho_i(t) = \frac{O_i^{\text{switch}}(t)}{\theta_i(t)}, \tag{16}$$

where $O_i^{\text{switch}}(t)$ represents context switching costs.

Our adaptive time-slicing mechanism provides rigorous theoretical guarantees to control scheduling overhead effectively.

Theorem 3 (Time-slice optimality)**.** The adaptive time-slice mechanism minimizes total scheduling overhead while meeting deadlines. For any task T_i:

$$\mathbb{E}[\rho_i(t)] \leqslant \min\left\{\frac{O_i^{\max}}{\theta_{\min}}, \frac{1}{\sigma_i(t)}\right\}, \tag{17}$$

where $O_i^{\max}$ is the maximum switching overhead.

To support this time-slicing mechanism, we implement state management routines that save membrane potentials and spike histories during preemption, track task progress and update slack ratios, and monitor switching overhead to provide feedback for subsequent scheduling decisions.

6 MPL-schedule implementation

We now present the practical implementation of the MPL-schedule scheme, along with a system architecture designed to operationalize its core scheduling mechanisms.

6.1 MPL-schedule architecture

As shown in Figure 4, MPL-schedule consists of four core components that enable priority-aware scheduling, adaptive resource allocation, and overhead-aware execution. The task manager, priority computation module, resource allocator, and execution monitor collaboratively realize the scheme's priority-driven scheduling and adaptive resource management.

The task manager handles incoming tasks and continuously monitors their lifecycle and execution status. As new tasks arrive, the task manager meticulously records their characteristics and requirements, laying the foundation for effective scheduling decisions.

This information is seamlessly passed to the priority computation module, which lies at the heart of MPL-schedule's adaptive scheduling strategy. Using (8), this module dynamically computes each task's priority based on deadline urgency, remaining workload, and energy consumption. The resulting priority scores serve as the driving force behind the subsequent resource allocation and scheduling steps.

Armed with the computed priorities, the resource allocator embarks on a two-phase process to optimally distribute the available GPU resources among the active tasks. In the initial phase, the allocator uses a proportional allocation scheme, which ensures that tasks with higher priority receive a larger share of the computational capacity. This theoretically grounded approach enables the system to swiftly respond to the most critical tasks, minimizing the risk of deadline misses. In the second phase, the allocator iteratively refines initial assignments via gradient-guided adjustments (12), driving the resource allocation toward global optimality.

The execution monitor continuously tracks performance metrics to evaluate and refine scheduling decisions and resource allocations. This information enables the scheme to make informed decisions, adjusting priorities and allocation strategies in response to changing workload demands and system conditions.

To facilitate this monitoring, MPL-schedule maintains an overhead log, capturing detailed records of context-switch and inter-task communication costs. It also uses a dynamic priority queue to reorder tasks and a state map to store neuron states and spike history for preemption. These components improve responsiveness and reduce scheduling overhead. For each task T_i, the overhead log records the measured switching costs $O_i^{\text{switch}}(t)$ at every preemption. At runtime, we keep a per-task estimate $O_i(t)$ as a smoothed average of these measurements, so the objective uses a recent estimate of the actual preemption cost instead of a fixed constant. The feedback controller monitors the relation between the new $O_i^{\text{switch}}(t)$ and the current $O_i(t)$ and, when a task shows persistently high switching cost, it enlarges that task's time slice within $[\theta_{\min}, \theta_{\max}]$ and reduces how often it is preempted. This limits the fraction of GPU time that is spent on repeated context switches.

Among the various metrics monitored, three stand out as particularly crucial for assessing the system's performance and guiding dynamic adjustments.

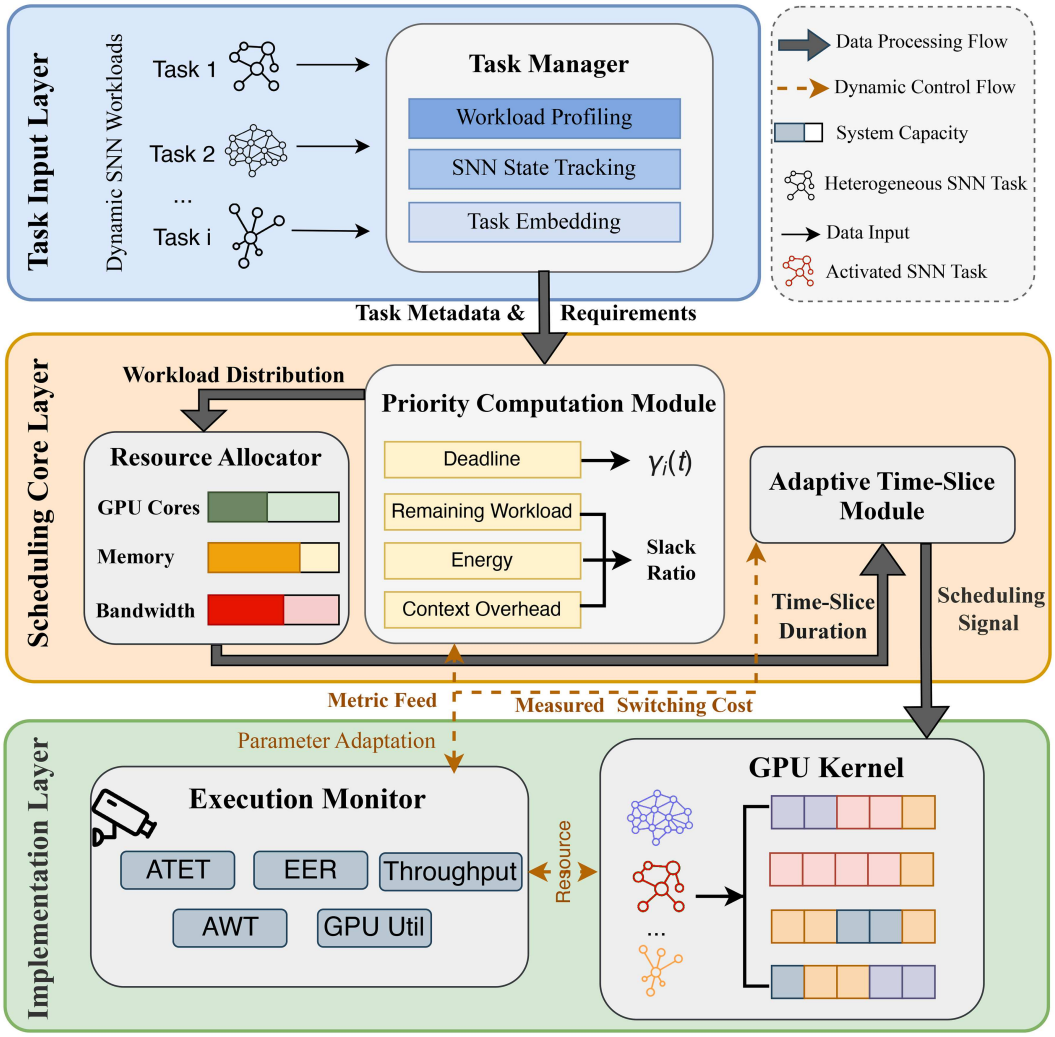

Figure 4 (Color online) System architecture of the MPL-schedule scheme.

Average task execution time (ATET):

$$\mathrm{ATET} = \frac{1}{N}\sum_{i=1}^{N}\left(t_i^{\mathrm{comp}} - t_i^{\mathrm{start}}\right). \tag{18}$$

ATET quantifies the system's responsiveness and efficiency. By tracking the average time taken from the moment a task starts executing until its completion, MPL-schedule can identify potential bottlenecks or resource contention issues. A higher ATET indicates that tasks are taking longer to complete, suggesting a need for adjustments in resource allocation or priority assignment.

Energy efficiency ratio (EER):

$$\mathrm{EER} = \frac{\sum_{i=1}^{N}\sum_{t=1}^{\tau_i} S_i(t)}{E_{\mathrm{total}}}. \tag{19}$$

EER is a critical metric for assessing the system's energy efficiency, which is a key consideration in GPU-based SNN simulations. By measuring the ratio of the total number of spikes processed to the total energy consumed, MPL-schedule can evaluate how effectively it is utilizing the available resources. A higher EER indicates that the system is processing more spikes per unit of energy, suggesting a more efficient resource allocation strategy.

Average waiting time (AWT):

$$\mathrm{AWT} = \frac{1}{N}\sum_{i=1}^{N}\left(t_i^{\mathrm{start}} - t_i^{\mathrm{arrive}}\right). \tag{20}$$

AWT measures the average time tasks spend waiting in the queue before they start executing. This metric is crucial for assessing the system's ability to handle high-priority or deadline-sensitive tasks. A lower AWT indicates that tasks are being scheduled and executed promptly, minimizing the risk of missing critical deadlines.

These metrics, along with others such as GPU utilization and throughput, form the basis of MPL-schedule's dynamic adaptation mechanism. The performance monitor continuously collects and analyzes these metrics, comparing them against predefined target values or thresholds. When significant deviations are detected, the monitor triggers the feedback control loop, which adjusts key system parameters to steer the performance back towards the desired state. The feedback control loop dynamically adjusts key parameters influencing scheduling decisions and resource allocations.

• Priority weights (α, β, δ). These weights determine the relative importance of factors such as deadline urgency, workload intensity, and energy consumption in the priority calculation (8). By dynamically adjusting these weights based on the observed metrics, MPL-schedule can adapt its scheduling strategy to prioritize the most critical aspects of performance at any given time.

• Time-slice bounds $(\theta_{\min}, \theta_{\max})$. The minimum and maximum time-slice durations (14) control the granularity of the scheduling decisions and the frequency of context switches. If the monitor detects excessive context switching overhead or resource underutilization, it can adjust these bounds to find a better balance between responsiveness and efficiency.

• Resource coupling parameters $(\kappa_{j,k})$. These parameters (2) capture the interdependencies between different resource types and constrain the resource allocation process. By dynamically tuning these parameters based on the observed resource utilization patterns, MPL-schedule can adapt to the specific characteristics of the workload and the underlying hardware, ensuring a more efficient utilization of the available resources.

These metrics guide dynamic adjustments that help sustain performance under changing workloads and system states.

6.2 MPL-schedule pipeline

To support the efficient operation of these components, MPL-schedule maintains a set of carefully designed data structures. The priority queue dynamically ranks tasks based on their current priorities. This helps the scheduler promptly dispatch urgent tasks and reduce deadline misses. Meanwhile, the state map stores runtime states of tasks, including membrane potentials, spike history, and progress. This supports seamless task preemption and resumption as priorities and resource availability change. The overhead log records context-switch and communication costs, informing priority computation and resource allocation. The MPL-schedule core execution pipeline, as described in Algorithm 1, embodies the seamless integration of these components and data structures. The step that measures the switching cost $O_i^{\text{switch}}(t)$ writes this value into the overhead log for the current task, so that $O_i(t)$ and the feedback controller both rely on the same runtime information about preemption cost. To remain robust when the overhead estimate temporarily lags behind reality, if $O_i^{\text{switch}}(t)$ stays high for several consecutive preemptions, MPL-schedule caps the overhead term using the empirical upper bound $O_i^{\max}$ and increases $\theta_i(t)$ for T_i to reduce further preemptions. At each time step, the pipeline updates active tasks, computes priorities, allocates resources, and monitors performance. This iterative process allows MPL-schedule to adaptively optimize scheduling and manage GPU resources for heterogeneous SNN workloads.

7 Experimental results

We evaluate MPL-schedule under diverse workloads to demonstrate its effectiveness in managing heterogeneous SNN tasks. We first outline our experimental setup, including three workload patterns simulating distinct arrival and complexity scenarios. We then compare MPL-schedule with baseline algorithms commonly used in GPU and real-time scheduling. The evaluation covers five key metrics, including throughput, energy efficiency, GPU utilization, completion time, and waiting time. It further analyzes deadline-related trade-offs through priority-weight sensitivity.

7.1 Experimental setup

We use four workload traces to evaluate MPL-schedule under different conditions. Each workload consists of a series of heterogeneous LIF tasks with different computational intensities.

Algorithm 1 MPL-schedule execution pipeline.

Require: Task set $\mathcal{T}$, resource constraints $\mathcal{C}$, maximum time steps $T_{\max}$.
Ensure: A record $\mathcal{S}$ of scheduling decisions and task performance metrics.
Create an empty structure $\mathcal{S}$ for storing per-time-step results;
Initialize task priorities $\gamma_i(0)$ for all tasks $T_i \in \mathcal{T}$;
for $t = 1$ to $T_{\max}$ **do**
  Update active task set $\mathcal{A}(t) \subseteq \mathcal{T}$;
  for all $T_i \in \mathcal{A}(t)$ (in parallel) **do**
    Compute priority $\gamma_i(t)$ using (8);
    Initialize resource allocation using (11);
    Refine resource allocation using (12);
  end for
  Schedule tasks according to final resource allocation $R_i^*(t)$;
  Monitor task execution metrics (e.g., completion ratio, energy usage);
  Dynamically adjust time slice $\theta_i(t)$ for each T_i using (14);
  for all $T_i \in \mathcal{A}(t)$ **do**
    if T_i is preempted **then**
      Save partial states of T_i;
      Update task progress and slack ratio $\sigma_i(t)$;
      Measure $O_i^{\text{switch}}(t)$ as the elapsed time of saving state, synchronization, restoring state, and kernel relaunch for T_i;
    end if
  end for
  Store $\{\gamma_i(t), R_i^*(t), \theta_i(t), \sigma_i(t), O_i^{\text{switch}}(t)\}_{T_i \in \mathcal{A}(t)}$ into $\mathcal{S}(t)$;
end for
return $\mathcal{S}$.

7.1.1 *Workload characteristics*

Workload 1 (Poisson arrival). It consists of LIF tasks with varying complexity ratios and neuron counts from 300 to 1050, in steps of 150. Task arrivals follow a Poisson process, with exponentially distributed inter-arrival times. It evaluates how well MPL-schedule handles stochastic arrivals with diverse computational loads.

Workload 2 (Bursty arrival). Workload 2 has the same task composition as Workload 1 in terms of complexity ratios and neuron counts. However, the task arrivals are clustered into bursts, with short periods of high arrival rates followed by relatively idle periods. It simulates real-world patterns with concentrated bursts of task arrivals. It evaluates MPL-schedule's performance in handling sudden resource demands and maintaining stability under significant fluctuations.

Workload 3 (Mixed arrival). To simulate more diverse environments, Workload 3 combines Poisson, bursty, and uniform arrival patterns. The task complexity and neuron counts remain consistent with the previous workloads. This mixed arrival pattern stresses scheduling algorithms, testing MPL-schedule's ability to adjust resource allocations and priorities under interleaved arrival types.

Workload 4 (Trace-driven arrival). Besides the synthetic workloads, we also include a trace-driven workload derived from the N-MNIST dataset using a convolutional LIF network implemented in SpikingJelly. We treat each input sample as one inference task and record its arrival timestamp and measured GPU execution time (per-task compute cost) when running a stream of samples. We then replay this trace in the simulator to complement the synthetic arrivals. This trace exhibits sparse and irregular arrivals due to event sparsity in N-MNIST, providing a more data-driven workload instance for evaluation.

7.1.2 *Experimental environment and task configuration*

All experiments are conducted on a small-scale GPU cluster composed of four compute nodes, each equipped with four NVIDIA 4090 GPUs (24 GB memory per GPU), 512 GB system RAM, and a 100 Gbps InfiniBand interconnect.

For the software configuration, all SNN tasks use LIF models adapted from the open-source SpikingJelly library. Each neuron group tracks membrane potentials, spike events, and total spike counts, and the GPU routines update LIF states at every time step and generate spikes when the membrane potential crosses a threshold.

7.1.3 *Evaluation protocol*

To assess performance under diverse scenarios, we evaluate MPL-schedule and baseline methods across five metrics—throughput, energy efficiency, GPU utilization, completion time, and waiting time. We further examine how load levels, complexity, and time-slice duration affect scheduling performance. The complexity ratio ranges from 0.3 to 0.7 (step 0.1), and neuron counts vary from 300 to 1050 (step 150). By controlling neuron count, complexity, and time-slice duration, we isolate the effect of each factor on scheduling performance. This enables fair comparison between MPL-schedule and baseline schedulers under varied conditions.

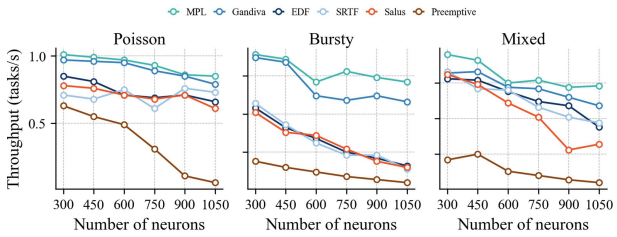

Figure 5 (Color online) Throughput under Poisson, bursty, and mixed arrival patterns at different neuron counts.

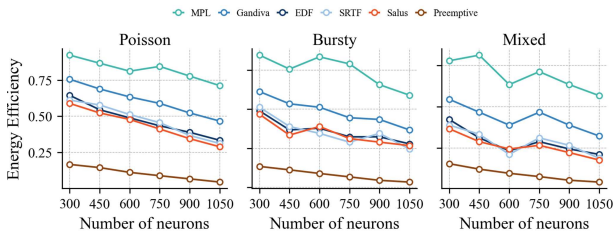

Figure 6 (Color online) Energy efficiency comparison under Poisson, bursty, and mixed arrival patterns.

7.2 Baselines

We evaluate MPL-schedule against five widely recognized scheduling algorithms: Gandiva, earliest deadline first (EDF), shortest remaining time first (SRTF), Salus, and a basic preemptive scheduler. These baselines encompass diverse scheduling paradigms, from resource multiplexing to deadline-driven preemption, providing comprehensive insights into MPL-schedule's relative advantages.

Gandiva [24] uses time-sharing and suspend-resume mechanisms to improve cluster utilization and provide early feedback for deep-learning jobs. We adapt it to track spike-driven activity so that partial state preservation can handle event-based neuronal computations.

SRTF [23] prioritizes tasks with fewer estimated steps remaining, preempting longer ones. For SNNs, we approximate each task's remaining execution by its current spiking activity to favor those with small pending workloads.

Salus [29] leverages spatial multiplexing and partial state preservation to quickly switch among deep learning models. We extend these ideas to accommodate SNN computations by restoring partial membrane states after context switches, thus co-locating multiple SNN tasks on a single GPU.

EDF [22] enforces a deadline-oriented strategy by preempting tasks with looser deadlines when a more urgent SNN task arrives. We assign explicit deadlines based on how many simulation steps must be completed before the task becomes time-critical.

Preemptive [30, 31] uses a fixed time-slice in a first-come–first-served manner and preserves partial SNN states upon switching. Although simple, it highlights overhead incurred by frequent context switches without advanced optimization or dynamic priority reassignment.

By comparing these baselines, we study how MPL-schedule handles deadlines, energy usage, throughput, and changing arrivals in spiking scenarios. The following sections present our experimental traces, performance metrics, and evaluations across multiple workload conditions.

7.3 Overall performance analysis

Throughput comparison. Figure 5 presents throughput trends as neuron count increases from 300 to 1050 (step 150) with a fixed complexity ratio of 0.4. MPL-schedule maintains a clear lead in all three arrival patterns. At 300 neurons, its throughput exceeds Gandiva and EDF by about 5%–8% and surpasses SRTF or preemptive by a wider margin. As the neuron count increases, the adaptive time slice of MPL helps to retain stable throughput, while the baselines exhibit larger drops. This pattern remains consistent for Poisson, bursty, and mixed arrivals. The results indicate that MPL's overhead-aware mechanism reduces contention and preserves concurrency, which leads to a 10.0%–15.0% improvement over methods that rely on simpler or more rigid scheduling decisions.

Energy efficiency. Figure 6 illustrates how the energy efficiency of each scheduler responds to increasing neuron counts in the same complexity ratio of 0.4. MPL sustains the highest values in Poisson, bursty, and mixed scenarios. In Poisson with 300 neurons, it outperforms Gandiva and EDF by 10.1%–20.3% and maintains its lead at larger scales. Under bursty arrivals, MPL's overhead-aware preemptions help avoid idle periods when spikes occur, which yields better energy usage. In mixed workloads, MPL continues to maintain a noticeable gap, while Salus and others show moderate efficiency that weakens under heavier loads. Preemptive sits at the bottom because repeated context switches lead to greater energy overhead. This advantage results from MPL's adaptive reallocation and priority adjustment, which reduce idle time and energy waste.

GPU utilization. Figure 7 shows that MPL-schedule keeps GPU utilization above 95.8% from 300 to 1050 neurons. By adapting resource allocation to workload intensity and arrival bursts, MPL keeps the device busy during spikes and avoids long idle gaps. SRTF and Gandiva also maintain high utilization, since both reschedule jobs to keep GPUs occupied. Note that our utilization metric counts any GPU-busy interval, including both effective SNN

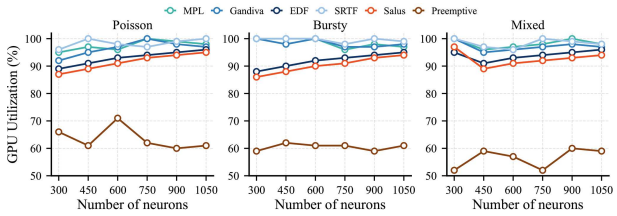

Figure 7 (Color online) GPU utilization across different neuron counts under Poisson, bursty, and mixed arrivals.

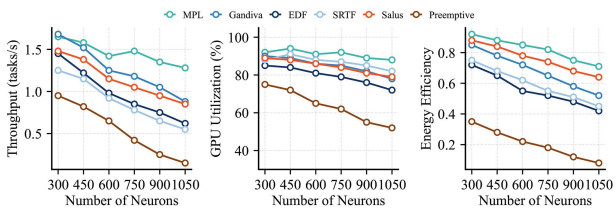

Figure 8 (Color online) Scheduling performance on the trace-driven N-MNIST workload with LIF dynamics.

Table 1 Completion time and waiting time (ms) under synthetic workloads. Bold values indicate the best results, and underlined values indicate the second-best results.

Workload	Scheduler	Neuron count, JCT/wait					
		300	450	600	750	900	1050
Poisson	MPL	12.98/4.93	47.11/18.23	**91.31**/32.75	**148.06**/53.49	**218.41**/79.23	**298.12**/114.65
	Gandiva	12.44/**3.82**	47.65/**12.10**	94.79/**26.03**	155.20/**38.54**	226.43/**66.21**	312.61/**76.84**
	EDF	13.92/5.12	53.48/16.45	100.22/30.74	164.71/48.65	238.11/73.65	324.93/111.45
	SRTF	**12.23**/4.05	**46.78**/14.24	93.46/29.35	152.88/43.01	222.11/71.06	303.44/98.54
	Salus	14.28/4.76	55.01/18.89	104.53/36.41	168.31/55.42	243.61/81.33	331.30/117.43
	Preemptive	63.51/27.24	342.11/135.21	692.43/284.65	1104.31/448.21	1547.23/608.41	2231.11/884.23
Bursty	MPL	14.20/5.43	**50.41**/19.82	**98.65**/38.41	**162.10**/64.31	**231.84**/92.41	**315.61**/124.78
	Gandiva	14.89/**4.42**	57.12/**14.89**	107.43/**28.32**	178.56/**44.21**	256.32/**65.11**	342.50/**89.24**
	EDF	17.51/6.81	66.89/23.61	123.51/41.22	195.42/57.23	272.11/92.65	365.12/108.31
	SRTF	**14.16**/4.92	52.12/17.20	102.32/33.65	169.51/50.11	240.21/78.65	325.77/101.55
	Salus	18.23/6.10	69.10/25.10	127.84/43.11	198.65/61.41	280.41/95.34	371.42/113.84
	Preemptive	88.24/36.41	392.41/158.32	752.41/305.23	1215.11/482.41	1684.32/652.11	2451.20/941.11
Mixed	MPL	13.72/5.08	**48.42**/18.41	**95.61**/37.41	**152.11**/58.21	**224.51**/87.41	**306.41**/121.21
	Gandiva	14.12/**3.74**	51.02/**13.41**	100.22/**27.12**	164.53/**41.44**	235.10/**65.23**	324.77/**84.56**
	EDF	15.65/5.05	58.21/15.31	110.53/33.11	176.43/52.12	255.43/79.11	342.66/115.42
	SRTF	**13.56**/4.31	49.82/16.20	98.54/31.62	158.66/48.77	230.12/72.53	318.53/99.10
	Salus	16.23/5.42	61.12/17.51	114.33/36.41	182.50/56.41	258.11/83.12	349.50/122.41
	Preemptive	78.43/29.50	372.10/145.42	721.56/288.54	1152.41/452.33	1521.43/612.44	2341.21/912.44

Table 2 Completion time and waiting time (ms) on the trace-driven N-MNIST workload.

Scheduler	Neuron count, JCT/wait					
	300	450	600	750	900	1050
MPL	38.52/**8.22**	85.54/**36.53**	185.57/**78.84**	315.82/**122.26**	**425.57/168.53**	**515.09/205.05**
Gandiva	42.52/11.83	110.25/65.27	258.54/155.51	425.58/285.88	615.22/450.55	750.57/565.02
EDF	48.46/16.53	130.57/90.54	305.82/205.24	490.57/355.59	680.83/520.22	850.58/710.03
SRTF	**36.85**/24.87	**78.83**/68.56	**178.21**/145.89	**298.54**/242.08	445.03/385.55	550.87/480.01
Salus	40.16/13.24	95.82/45.57	215.03/105.03	365.88/178.02	520.54/265.58	640.09/320.04
Preemptive	155.59/78.53	580.52/380.26	950.57/620.54	1400.22/1150.03	2150.06/1850.28	2800.54/2400.07

computation and runtime overhead, such as state save/restore and slice-boundary stalls. MPL-schedule reduces the overhead share through overhead-aware preemption and adaptive time slices, indicating that more GPU-busy time is translated into effective SNN progress rather than repeated preempt-resume operations.

Completion and waiting times. Tables 1 and 2 report completion time and waiting time under synthetic and trace-driven workloads, respectively. The synthetic workloads include Poisson, bursty, and mixed arrivals, and MPL-schedule achieves lower completion times than Gandiva and EDF in most settings, especially beyond 600 neurons or under bursty arrivals.

For the synthetic workloads, Gandiva sometimes obtains shorter waiting times by favoring faster queue admission, while MPL-schedule provides more stable completion under heavy SNN workloads. On the trace-driven N-MNIST workload, MPL-schedule achieves both low completion time and consistently shorter waiting time at larger neuron counts, confirming its advantage under realistic sparse event-driven arrivals.

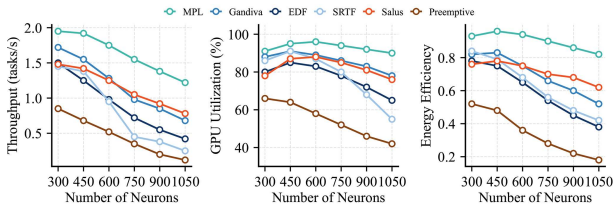

Figure 9 (Color online) Scheduling performance on the trace-driven N-MNIST workload with IF dynamics.

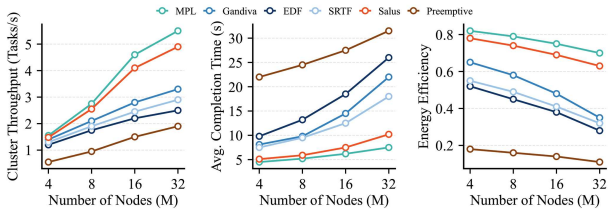

Figure 10 (Color online) Scaling performance from 4 to 32 nodes under the N-MNIST anchor workload.

Trace-driven evaluation on N-MNIST. Figure 8 reports performance on the N-MNIST workload. GPU utilization on this trace seldom exceeds 85%–90% and shows clear fluctuations, which is consistent with a bandwidth-bound, sparse SNN workload. MPL-schedule maintains the highest throughput, outperforming Gandiva by up to 45% at 1050 neurons, while Salus remains within about 10% of MPL in energy efficiency by exploiting spatial memory sharing. This suggests that memory sharing helps amortize overhead, and MPL's adaptive prioritization further improves effective GPU usage.

Table 2 summarizes the completion time and waiting time on the trace-driven N-MNIST workload. At 750 neurons, SRTF achieves a 5.4% lower average completion time than MPL-schedule by aggressively prioritizing short tasks, but its average waiting time is nearly 2× higher, revealing pronounced unfairness for long-running jobs. In contrast, MPL-schedule balances efficiency and fairness: it keeps completion time close to the best-performing baseline while cutting average waiting time by up to 10× compared with the preemptive baseline under congestion. This provides a more stable trade-off on realistic stochastic SNN workloads.

IF instantiation on the N-MNIST trace. To quantify model diversity under a realistic workload, we instantiate MPL-schedule with integrate-and-fire (IF) dynamics on the same N-MNIST trace. We keep the network structure and trace-driven task generation unchanged, and only replace the per-step neuron-update kernel. Using the same measurement interface, we re-profile the per-step kernel cost and the preempt-resume switching cost, and then run the identical scheduling pipeline. Since IF alters the per-step update behavior (and thus the switch-to-progress ratio) while leaving arrivals identical, Figure 9 shows that MPL-schedule preserves its advantage on this IF instantiation: at the highest load of 1050 neurons, it delivers 1.56× higher throughput than the strongest baseline and 1.32× higher energy efficiency, indicating that more GPU busy time is converted into effective SNN progress rather than preempt–resume overhead. This aligns with the utilization trend, where MPL-schedule maintains a utilization margin of roughly 12 percentage points at 1050 neurons. Overall, these results demonstrate that our measurement-driven interface can adapt MPL-schedule to different neuron dynamics without modifying the core pipeline, while still improving efficiency and stability under trace-driven arrivals.

7.4 Analysis and scalability

Cluster scalability. Besides running on the physical 4-node cluster, we use a calibrated trace-driven simulator to evaluate scalability beyond the testbed limits. The simulator is anchored to the heaviest N-MNIST setting from the real system. To assess performance at scale, we expand the cluster size from M=4 to M=32 nodes while scaling the total neuron count proportionally (from 1050 to 8400) to maintain a consistent workload per node. Inter-node spike-broadcast and synchronization costs are incorporated into the calibrated trace-driven simulator. As shown in Figure 10, MPL-schedule scales efficiently, achieving a 3.5× aggregate throughput increase while limiting average completion time growth to 66%. In contrast, Gandiva and EDF suffer over 160% JCT degradation due to accumulated synchronization overhead. At 32 nodes, MPL-schedule delivers 1.67× higher throughput than Gandiva and retains 85% of its baseline energy efficiency. These results demonstrate that MPL-schedule's overhead-aware strategy remains robust against the communication bottlenecks inherent in large-scale SNN simulations.

Sensitivity to priority weights. We obtain a baseline setting (α, β, δ) via a coarse grid search on a held-out workload slice, targeting a balanced trade-off among deadline miss rate, throughput, and energy efficiency. To quantify the trade-offs implied by (8), we further conduct a small sensitivity study by sweeping (α, β, δ) over four representative profiles while keeping all other settings (workload trace, resource limits, and preemption mechanism) unchanged: deadline-centric $(0.60, 0.30, 0.10)$, throughput-centric $(0.20, 0.60, 0.20)$, energy-centric $(0.20, 0.20, 0.60)$, and a balanced baseline $(0.40, 0.40, 0.20)$.

Figure 11 reports the resulting performance trade-offs under the mixed-arrival workload. As expected, prioritizing deadline urgency (deadline-centric) yields the lowest miss rate at the expense of raw throughput. Conversely, the Throughput-centric profile maximizes tasks per second but incurs a higher miss rate. Notably, the energy-centric

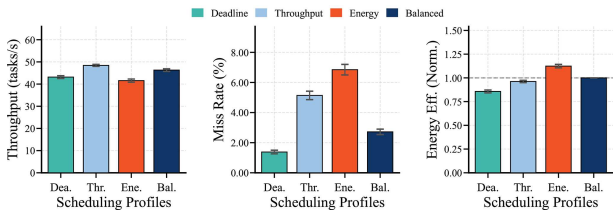

Figure 11 (Color online) Sensitivity analysis of priority weights (α, β, δ) showing trade-offs between performance metrics.

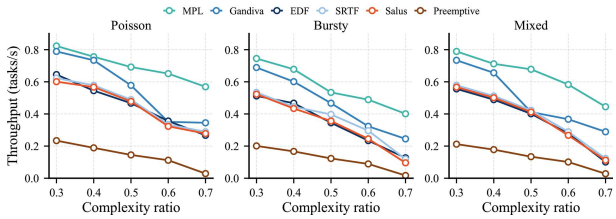

Figure 12 (Color online) Throughput analysis of complexity ratio under Poisson, bursty, and mixed arrival patterns.

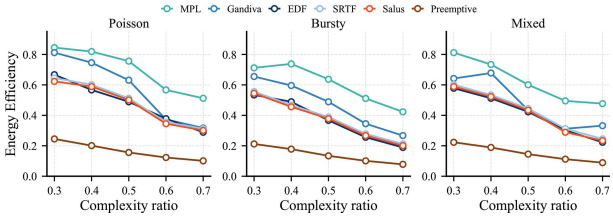

Figure 13 (Color online) Energy efficiency across complexity ratios.

Table 3 Normalized GPU utilization under different complexity ratios.

Scheduler	Poisson			Bursty			Mixed		
	0.30	0.50	0.70	0.30	0.50	0.70	0.30	0.50	0.70
MPL	**1.00**	0.95	**0.92**	0.99	**0.94**	**0.84**	**1.00**	**0.97**	**0.89**
Gandiva	0.99	0.94	0.89	0.99	0.88	0.81	0.98	0.95	0.87
SRTF	**1.00**	**0.98**	0.90	**1.00**	0.82	0.84	**1.00**	0.96	0.88
EDF	0.98	0.91	0.78	0.94	0.80	0.72	0.91	0.82	0.74
Salus	0.89	0.85	0.78	0.95	0.78	0.74	0.93	0.81	0.76
Preemptive	0.62	0.49	0.37	0.50	0.34	0.17	0.48	0.41	0.30

configuration improves the normalized energy efficiency significantly above the baseline (indicated by the dashed line), while the balanced profile maintains a stable middle ground across all metrics. This demonstrates that MPL-schedule can flexibly adapt to diverse service-level objectives by tuning the priority weights.

Complexity ratio. Figures 12 and 13 show how throughput and energy efficiency vary as the complexity ratio increases from 0.3 to 0.7. MPL-schedule consistently outperforms the baselines across all arrival patterns by converting more GPU-busy time into useful SNN computation rather than switching or waiting. Under Poisson arrivals, overhead-aware allocation preserves throughput for lower-complexity tasks. Under bursty and mixed arrivals, adaptive preemption helps handle large tasks without starving smaller ones, sustaining higher efficiency beyond a complexity ratio of 0.5.

GPU utilization under different complexity ratios. Table 3 reports normalized GPU utilization at three representative complexity ratios. MPL-schedule, SRTF, and Gandiva approach full occupancy at 0.30, but SRTF drops noticeably under bursty traffic at higher complexity ratios. At 0.50 and 0.70, Gandiva and Salus show more idle time, while preemptive stays below 0.50 because frequent switching fragments execution. MPL-schedule sustains high utilization by prioritizing computation-heavy tasks and adapting allocation to workload intensity.

Resource utilization over different time slices. Figure 14 shows resource utilization across time slices from 1 to 10 at complexity ratio 0.4 and 450 neurons. MPL-schedule remains close to full utilization under Poisson, bursty, and mixed arrivals, indicating that its allocation policy is robust to changes in slice length. In contrast, other schedulers achieve acceptable utilization mainly at smaller slices, but degrade as the slice grows due to weaker adaptation to bursty and heterogeneous SNN tasks.

Context-switch overhead and waiting time over different time slices. Figures 15 and 16 show the impact of time-slice length on context-switch overhead and average waiting time. As the slice increases from 1 to 10, switching overhead generally decreases, with MPL-schedule showing the most consistent reduction. At a slice of 10 under Poisson arrivals, MPL reduces the overhead to about 0.31, while preemptive remains above 4.5. This advantage comes from overhead-aware preemption, which avoids repeated save–restore operations when switching costs become high. Meanwhile, MPL keeps waiting time relatively low across slice durations by adapting priorities

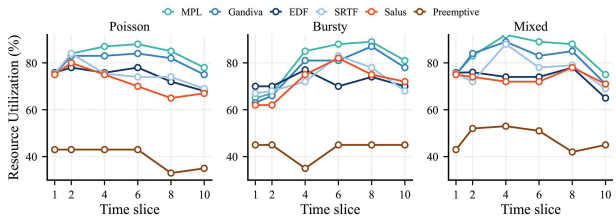

Figure 14 (Color online) Resource utilization at different time slices.

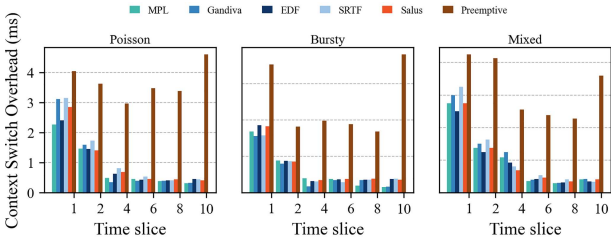

Figure 15 (Color online) Context-switch overhead under three arrival patterns.

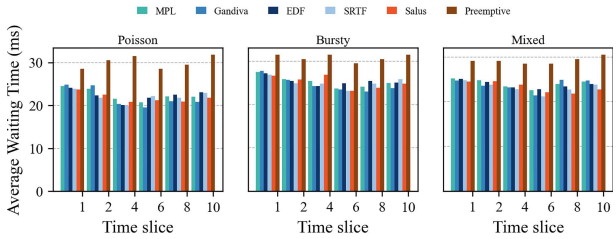

Figure 16 (Color online) Average waiting time under three arrival patterns.

and reallocating resources according to workload dynamics. Overall, MPL-schedule achieves a better trade-off between fewer preemptions and controlled queueing delay.

8 Conclusion

This work presents MPL-schedule, a multi-preemptive, priority-aware scheduling framework for heterogeneous SNN workloads on GPUs. By combining overhead-aware resource allocation with adaptive priority assignment and time slicing, MPL-schedule improves throughput, energy efficiency, and responsiveness under dynamic arrivals and diverse task complexities. Theoretical analysis establishes optimality conditions and convergence guarantees for the underlying optimization. Experiments under Poisson, bursty, mixed, and trace-driven N-MNIST workloads show that MPL-schedule demonstrates clear advantages over state-of-the-art baselines, and a time-slice sensitivity study confirms robustness across a wide range of load levels. Our design mainly targets contention-heavy, multi-tenant SNN services with heterogeneous spike patterns and latency requirements. In low-load regimes or nearly homogeneous batch workloads, the scheduling policy has limited room to improve system-level metrics beyond avoiding regression. Future work includes extending the evaluation to additional neuron models and larger-scale physical deployments, and integrating MPL-schedule with SNN compilation and backend systems.

Acknowledgements This work was supported by National Key Research and Development Program of China (Grant No. 2022YFB4500100), National Natural Science Foundation of China (Grant Nos. 62502443, 62125206), and the Zhejiang Provincial Natural Science Foundation of China (Grant No. LD24F020014).

References

1 Satyanarayanan M. The emergence of edge computing. Computer, 2017, 50: 30–39
2 Maass W. Networks of spiking neurons: the third generation of neural network models. Neural Netws, 1997, 10: 1659–1671
3 Roy K, Jaiswal A, Panda P. Towards spike-based machine intelligence with neuromorphic computing. Nature, 2019, 575: 607–617
4 Wu Y, Deng L, Li G, et al. Direct training for spiking neural networks: faster, larger, better. In: Proceedings of the 40th AAAI Conference on Artificial Intelligence, Honolulu, 2019. 1311–1318
5 Davies M, Srinivasa N, Lin T H, et al. Loihi: a neuromorphic manycore processor with on-chip learning. IEEE Micro, 2018, 38: 82–99
6 Zhang T, Morris J, Stewart K, et al. HyperSpikeASIC: accelerating event-based workloads with hyperdimensional computing and spiking neural networks. IEEE Trans Comput-Aided Des Integr Circuits Syst, 2023, 42: 3997–4010
7 Kato S, Lakshmanan K, Rajkumar R, et al. TimeGraph: GPU scheduling for real-time multi-tasking environments. In: Proceedings of the 2011 USENIX Annual Technical Conference, Portland, 2011. 17–30
8 Ye Z, Gao W, Hu Q, et al. Deep learning workload scheduling in GPU datacenters: a survey. ACM Comput Surv, 2024, 56: 1–38
9 Park J J K, Park Y, Mahlke S. Chimera: collaborative preemption for multitasking on a shared GPU. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, Istanbul, 2015. 593–606
10 Topcuoglu H, Hariri S, Wu M-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distrib Syst, 2002, 13: 260–274
11 Gerstner W, Sprekeler H, Deco G. Theory and simulation in neuroscience. Science, 2012, 338: 60–65
12 Gerstner W, Kistler W M. Spiking Neuron Models: Single Neurons, Populations, Plasticity. Cambridge: Cambridge University Press, 2002
13 Fidjeland A K, Roesch E B, Shanahan M P, et al. NeMo: a platform for neural modelling of spiking neurons using GPUs. In: Proceedings of IEEE International Conference on Application-Specific Systems, Architectures and Processors, Boston, 2009. 137–144

14 Fidjeland A K, Shanahan M P. Accelerated simulation of spiking neural networks using GPUs. In: Proceedings of International Joint Conference on Neural Networks, Barcelona, 2010. 1–8
15 Qu P, Zhang Y, Fei X, et al. High performance simulation of spiking neural network on GPGPUs. IEEE Trans Parallel Distrib Syst, 2020, 31: 2510–2523
16 Qu P, Lin H, Pang M, et al. ENLARGE: an efficient SNN simulation framework on GPU clusters. IEEE Trans Parallel Distrib Syst, 2023, 34: 2529–2540
17 Zhang H, Ho N M, Polat D Y, et al. Simeuro: a hybrid CPU-GPU parallel simulator for neuromorphic computing chips. IEEE Trans Parallel Distrib Syst, 2023, 34: 2767–2782
18 Knight J C, Nowotny T. GPUs outperform current HPC and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. Front Neurosci, 2018, 12: 941
19 Davison A P, Brüderle D, Eppler J M, et al. PyNN: a common interface for neuronal network simulators. Front Neuroinform, 2009, 2: 11
20 Pecevski D, Natschläger T, Schuch K. PCSIM: a parallel simulation environment for neural circuits fully integrated with Python. Front Neuroinform, 2009, 3: 11
21 Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard-real-time environment. J ACM, 1973, 20: 46–61
22 Andrews M. Probabilistic end-to-end delay bounds for earliest deadline first scheduling. In: Proceedings of IEEE Conference on Computer Communications, Tel Aviv, 2000. 603–612
23 Bansal N, Harchol-Balter M. Analysis of SRPT scheduling: investigating unfairness. In: Proceedings of ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Cambridge, 2001. 279–290
24 Xiao W, Bhardwaj R, Ramjee R, et al. Gandiva: introspective cluster scheduling for deep learning. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation, Carlsbad, 2018. 595–610
25 Fan R, Ren T, Xie M, et al. GPreempt: GPU preemptive scheduling made general and efficient. In: Proceedings of the 2025 USENIX Annual Technical Conference, Boston, 2025. 263–272
26 Strati F, Ma X, Klimovic A. Orion: interference-aware, fine-grained GPU sharing for ML applications. In: Proceedings of the 19th European Conference on Computer Systems, Athens, 2024. 1–17
27 She X, Long Y, Mukhopadhyay S. Fast and low-precision learning in GPU-accelerated spiking neural network. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition, Florence, 2019. 450–455
28 Hillar C J, Lim L H. Most tensor problems are NP-hard. J ACM, 2013, 60: 1–39
29 Yu P, Chowdhury M. Salus: fine-grained GPU sharing primitives for deep learning applications. In: Proceedings of Machine Learning and Systems, Austin, 2020. 98–111
30 Corbató F J, Merwin-Daggett M, Daley R C. An experimental time-sharing system. In: Proceedings of Spring Joint Computer Conference, San Francisco, 1962. 335–344
31 Kleinrock L. Time-shared systems. J ACM, 1967, 14: 242–261

Appendix A Preliminaries on spiking neural networks and GPU execution

This appendix outlines the fundamentals of SNNs and their execution on GPU-based platforms. Although the experiments adopt the LIF neuron model, the concepts broadly extend to other spiking neuron models employing discrete spike events.

Appendix A.1 Fundamentals of spiking neural networks

Unlike traditional DNNs, where information is conveyed through continuous activation functions, SNNs rely on discrete spikes to encode and transmit signals [3]. In many SNN models, each neuron maintains an internal state, commonly referred to as its membrane potential, which evolves under incoming synaptic inputs. Once this potential crosses a threshold, the neuron emits a spike and subsequently resets [12].

Multiple neuron models exist to capture spiking behaviors with varying degrees of biophysical detail. More detailed models can mimic ion-channel dynamics but are often computationally intensive. By contrast, the LIF model provides a simplified yet effective approximation

$$\tau_m \frac{\mathrm{d}V_i(t)}{\mathrm{d}t} = -\big[V_i(t) - V_{\mathrm{rest}}\big] + z_m\, I_i(t), \tag{A1}$$

where τ_m is the membrane time constant, $V_i(t)$ is the membrane potential of the i-th neuron at time t, V_{rest} is the resting potential, z_m denotes the membrane resistance, and $I_i(t)$ is the input current. Once $V_i(t)$ surpasses the firing threshold V_{th}, the neuron emits a spike and resets its potential to V_{reset} [12]. Although we adopt LIF for practical demonstrations, the broader scheme discussed in this paper remains relevant to other spiking neuron formulations that share the same discrete event-driven paradigm.

Despite variations in neuron or synapse models, SNNs share two fundamental characteristics. First, they are event-driven: computational steps hinge upon the occurrence and timing of spikes. Second, they are stateful: a neuron's membrane potential at any instant depends on its spike history. These properties allow SNNs to explicitly model temporal dynamics, making them highly suitable for applications in neuroscience, cognitive computing, and low-power neuromorphic systems [2, 3, 12].

Appendix A.2 GPU-based execution and model scheduling

GPUs can substantially accelerate SNN simulations by exploiting massive parallelism and high memory bandwidth [15]. In typical GPU-based setups, the SNN computation is decomposed into kernels performing the following tasks.

- *Neuron state updates.* Each thread (or thread block) updates membrane potentials for assigned neurons. If a neuron surpasses its firing threshold, the kernel records a spike event [15].
- *Spike delivery and synaptic integration.* Upon spike generation, downstream neuron potentials are incrementally updated according to synaptic connections, typically using sparse data structures optimized for irregular connectivity [16].
- *Global synchronization.* After each simulation step, a barrier-style synchronization is required to ensure that spikes emitted in the current step are consistently visible to the next step, preserving correct step-to-step causality in time-stepped GPU SNN simulation.

Techniques such as kernel batching, memory coalescing, and overlapping data transfers can minimize overhead, enhance throughput, and improve GPU utilization. However, spiking activity varies significantly over time, generating bursty or highly sparse computational demands across neuron populations [6]. This variability introduces additional complexity when SNN workloads or mixed GPU jobs are executed concurrently.

In this context, model scheduling refers to coordinating and ordering the execution of SNN kernels and associated memory operations across shared GPU resources. Large SNN simulations typically consist of multiple kernels running in quick succession, each handling distinct neuron groups or synaptic computations [18]. When multiple concurrent tasks or SNN models must execute simultaneously, the scheduler must address.

(1) How GPU cores and memory bandwidth are allocated among active kernels or tasks.

(2) When to launch or preempt kernels, especially if competing tasks have differing priorities or deadlines.

(3) How to efficiently handle state preservation, as each neuron's membrane potential and spike-related buffers must persist consistently across time steps [18].

Therefore, effective model scheduling must go beyond simple kernel queueing, addressing concurrency control, resource partitioning, and the overhead of neuron state management during suspension and resumption. Because SNN neurons carry state information across simulation steps, naive scheduling (e.g., round-robin kernel execution) can lead to imbalanced GPU resource usage or unacceptable delays for latency-sensitive tasks. Furthermore, the inherent burstiness of spiking activity means kernels can alternate between high-resource demands and near-idle states. A sophisticated scheduling approach can adapt to these fluctuations, thereby significantly enhancing both throughput (i.e., total completed simulation steps per unit time) and responsiveness (timely updates of critical neuron populations) [6].