RESEARCH-ARTICLE

Advancing Matrix Operations for High-Performance and Memory-Efficient Automata Processing on GPUs

ZHENLIN WU, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China

TIANAO GE, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China

JIAJIA LI, NC State University, Raleigh, NC, United States

XINYU CHEN, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China

HONGYUAN LIU, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, Guangdong, China

Published: 16 December 2025
Online AM: 04 November 2025
Accepted: 16 October 2025
Revised: 16 October 2025
Received: 16 June 2025

Advancing Matrix Operations for High-Performance and Memory-Efficient Automata Processing on GPUs

ZHENLIN WU, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China
TIANAO GE, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China
JIAJIA LI, North Carolina State University at Raleigh, Raleigh, United States
XINYU CHEN, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China
HONGYUAN LIU, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China

Finite state automata are essential in various domains, such as pattern matching and data analytics, where high throughput is critical. Recent work has explored representing automata execution as matrix algebra and leveraging CPU Basic Linear Algebra Subprograms (BLAS) libraries. While promising, this approach faces bottlenecks in memory usage, data locality, and redundant computation. This work systematically identifies these bottlenecks and develops automata-specific optimizations to address them.

We focus on GPUs due to their high compute capabilities and widespread availability. To overcome these challenges, we propose three key techniques to enhance computational and memory efficiency: (1) transition matrix deduplication to reduce memory usage, (2) interleaved state renumbering to improve GPU thread utilization, and (3) state vector caching to eliminate redundant computations. Detailed evaluation shows that the proposed solution matches GPU-based automata engines in performance (up to 6.54× speedup) while using less than 2% of their memory footprint, and outperforms state-of-the-art domain-specific accelerators (up to 965×) for automata workloads.

CCS Concepts: • **Computing methodologies → Massively parallel algorithms**; • **Theory of computation → Formal languages and automata theory**; • **Computer systems organization → Single instruction, multiple data**; • **Software and its engineering → Massively parallel systems;**

Additional Key Words and Phrases: Automata processing, GPU, sparse matrix-vector multiplication (SpMV)

New Paper, Not an Extension of a Conference Paper.
This material is based upon work supported by the National Natural Science Foundation of China (#62302416).
Authors' Contact Information: Zhenlin Wu, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China; e-mail: zwu065@connect.hkust-gz.edu.cn; Tianao Ge, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China; e-mail: tge601@connect.hkust-gz.edu.cn; Jiajia Li, North Carolina State University at Raleigh, Raleigh, North Carolina, United States; e-mail: jiajia.li@ncsu.edu; Xinyu Chen, The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China; e-mail: xinyuchen@hkust-gz.edu.cn; Hongyuan Liu (corresponding author), The Hong Kong University of Science and Technology - Guangzhou Campus, Guangzhou, China; e-mail: hongyuanliu@hkust-gz.edu.cn.

ACM 1544-3973/2025/12-ART156
https://doi.org/10.1145/3774656

ACM Reference Format:
Zhenlin Wu, Tianao Ge, Jiajia Li, Xinyu Chen, and Hongyuan Liu. 2025. Advancing Matrix Operations for High-Performance and Memory-Efficient Automata Processing on GPUs. *ACM Trans. Arch. Code Optim.* 22, 4, Article 156 (December 2025), 26 pages. https://doi.org/10.1145/3774656

1 Introduction

Finite automata frequently serve as core compute kernels in a wide range of applications, including pattern matching [68, 69], data analytics [24, 34, 39, 42], mining [68, 69], bioinformatics [18, 52], machine learning [62], network traffic inspection [74], and intrusion detection [3–5, 49, 51, 77]. As automata workloads become increasingly complex and grow in scale, their inherent parallelism [21, 29, 45, 75, 78] offers significant opportunities to achieve higher throughput using parallel computer architectures [26, 37, 54–56, 58]. **Graphics Processing Units** (**GPUs**), in particular, stand out as a compelling choice due to their massively parallel processing capabilities and widespread adoption across various domains [60] . Significant studies have been made in characterizing and optimizing automata workloads for GPUs [29, 44, 47, 63, 78], with numerous specialized techniques designed to efficiently exploit parallelism (Table 1). However, most solutions rely on task-specific codebases and lack a general interface for portability and reuse, posing a challenge when integrating new optimization techniques.

Our Vision: A Basic Linear Algebra Subprograms (BLAS)-Based Abstraction. To unify and generalize GPU-based AP, we adopt a matrix algebra abstraction inspired by GraphBLAS [23, 25], in which state transitions are expressed as **sparse matrix-vector** (**SpMV**) multiplications. Three key reasons behind our vision are: (1) Automaton transitions can be encoded as matrices, and the current state set as a sparse vector, enabling state updates via SpMV. (2) **Basic Linear Algebra Subprograms** (**BLAS**) is a standardized interface with optimized implementations across CPU, GPU, and emerging hardware, making BLAS a promising foundation for a general-purpose automata engine. (3) Previous methods [29, 44, 78] suffer from a lack of memory efficiency due to the need for extensive worklist maintenance. This abstraction is not unique to this work. Prior automata studies have also mapped automata to matrix operations [27, 64], but they are based on multi-core CPUs and exploit the associativity of matrix multiplication to realize task-level parallelism through a sequence of matrix multiplications applied to a vector. In contrast, AP on GPUs requires fine-grained data-parallel SpMV for each input symbol. Our primary goal is to address key research questions when adopting matrix-based approaches to the specific needs of AP workloads: **(RQ1)** Can vendor-provided GPU BLAS libraries effectively support the computation demands of automata workloads? **(RQ2)** What are the primary performance bottlenecks that hinder the effectiveness of these workloads? **(RQ3)** How can we mitigate these bottlenecks to improve the overall efficiency of AP on GPUs?

Challenges in Adopting Standard BLAS Subprograms to Automata. To answer these research questions, we prototype AP using vendor-optimized SpMV kernels. Our initial prototype using a vendor-provided sparse BLAS library (cuSPARSE [10]) reveals two fundamental challenges. **Challenge 1:** Each input symbol triggers a short-lived SpMV kernel, resulting in high control overhead from frequent kernel launches. **Challenge 2:** Even with single-kernel execution, we observe three remaining inefficiencies: (1) *memory redundancy* from storing structurally similar transition matrices across symbols, (2) *poor data locality* because nonzeros processed together by a warp are often scattered in memory, and (3) *redundant computation* from repeatedly updating identical state vectors.

Table 1. Comparison of GPU-based **Automata Processing** (**AP**) Optimizations, which Focus on Thread Mapping, Storage, and Parallelism

Schemes	Thread Mapping	Automata Storage	Parallelism
iNFAnt [21]	States	Transition Table	Inputs, Automata
NFA-CG [78]	State Group	Transition Table	Inputs, State Groups
GPU-NFA [44]	Active States	CSR + Compressed Matchset	Inputs, Active States
*ng*AP [29]	Active State/Symbol	CSR + Matchset + Memoization	Symbols, Active States
Our Work	**Nonzeros**	**Transition Matrices**	**Nonzeros**

In contrast, our work employs a BLAS-based abstraction to achieve both high performance and improved memory efficiency.

Our Customized Design to Advance the Efficiency of Sparse Matrix Operations. We propose AutomataBLAS, a GPU-based automata engine. To avoid the kernel launch overhead in Challenge 1, we design a custom **Cooperative Thread Array** (**CTA**)-level SpMV kernel, which allows each thread block to process a group of automata independently without host-side synchronization. To address Challenge 2, we propose holistic solutions consisting of three novel optimizations:

- **Transition Matrix Deduplication** reduces memory consumption by eliminating redundant transition matrices.
- **Interleaved State Renumbering** improves thread utilization by co-locating states that are often active together, enhancing data locality.
- **State Vector Caching** reduces redundant computation by memoizing frequently seen state vectors and their corresponding transition matrices.

In summary, this work makes the following contributions:

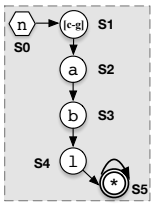
- We perform a systematic evaluation of vendor BLAS libraries on automata workloads, identifying two key challenges that restrict their effectiveness.
- We design **AutomataBLAS**, a lightweight yet general engine that addresses these challenges through a CTA-level SpMV kernel to eliminate control overhead and three automata-specific optimizations targeting memory redundancy, data locality, and redundant computation.
- Our design outperforms GPU-NFA [44] and NFA-CG [78] by up to **6.54**× and **2.94**×, respectively, while using only **0.08%** and **1.66%** of the memory of *ng*AP [29] and GPU-NFA. Compared to a recent sparse matrix accelerator [41], AutomataBLAS achieves **56.33**× speedup.

2 Background and Motivation

This section provides an overview of non-deterministic finite automata, the matching process, and analyzes the performance challenges for a BLAS-based implementation on GPUs.

2.1 Automata and Matching Process

Non-deterministic Finite Automaton (NFA). NFA is formally defined as a quintuple $(Q, \Sigma, Q_0, \delta, F)$, where Q represents a finite set of states, Σ denotes the alphabet that the states may accept, Q_0 signifies the set of *starting states*, and F is the set of *reporting states*, $Q_0, F \subseteq Q$. The transition function $\delta(S, \alpha)$ defines the states activated when a set $S \subseteq Q$ matches the input symbol $\alpha \in \Sigma$. Figure 1(a) shows an NFA accepting the pattern `/n[c-g]abl.+/`, with states represented as nodes. Here, $Q = \{S_0, S_1, S_2, S_3, S_4, S_5\}$, Σ is the ASCII character set, $Q_0 = S_0$ (the hexagon), and $F = S_5$ (the double-circle). Each directed edge indicates a transition δ, with



(a) An NFA.

Iter.	Symbol	Active States	Matched States	Reporting States
0-7	automata	S0		
8	n	S0	S0	
9	f	S0, S1	S1	
10	a	S0, S2	S2	
11	b	S0, S3	S3	
12	l	S0, S4	S4	
13	a	S0, S5	S5	S5
14	s	S0, S5	S5	S5

(b) The NFA matching process.

Fig. 1. **(a)** An NFA that matches pattern $/n[c-g]abl.+/$. ".+" matches any character one or more times; **(b)** An example of the matching process on "automatanfablas".

accepted symbol α or ranges shown in the source node (e.g., $[c-g]$ at state S_1). States accepting all symbols in Σ are marked as "*". [1] Each state has a matchset representing the symbols it accepts, and transitions occur only on symbols from this set. Compared to **deterministic finite automata** (**DFAs**), **non-deterministic finite automata** (**NFAs**) are better suited for GPUs due to their ability to have multiple active states simultaneously, allowing for greater parallelism.

In this study, we concentrate on a specific class of NFAs known as homogeneous NFAs, which are widely utilized in various automata representations, such as ANML [2] and MNRL [13]. A homogeneous NFA is defined by the constraint that each state is limited to accepting a fixed subset of input symbols. More formally, for every state $q \in Q$, there exists a subset $A_q \subseteq \Sigma$ such that the transition function $\delta(q, \alpha)$ is non-empty if and only if the symbol α is included in A_q. In other words, all transitions directed towards a given state must share the same subset of Σ. Despite this constraint, homogeneous NFAs maintain the full expressive power of traditional NFAs. In particular, Glushkov's construction [20, 32] allows for the transformation of any regular expression into a homogeneous NFA. In applications that involve NFA processing, multiple homogeneous NFAs are typically used, with each one representing a distinct pattern that the application seeks within the input streams. This makes them suitable for various AP engines [29, 44, 70] and domain-specific accelerators [57, 58] .

Matching Process. The matching process matches the input stream with all NFAs in each application iteratively, starting by activating the starting states. There are two types of starting states [2, 13]: (1) "all-input", which are *always active* during the matching process and allow patterns to begin at any position in the input stream; and (2) "start-of-data", which restrict patterns to the start only at the beginning of the input stream. To process a symbol, the *active* states are checked against the input symbol; if the symbol belongs to the matchset of a state, that state becomes a *matched* state. For example, S_0 is an "all-input" state that remains always active throughout the matching process. The matched states then activate their neighbors in the NFAs and this process continues until all symbols in the input stream are consumed. When a reporting state transitions to a matched state, a report is generated with matched states and corresponding input locations, indicating that an interesting pattern has been detected.

Figure 1(b) shows how the NFA in Figure 1(a) matches the input stream "automatanfablas". S_0 is inactive until the 8th iteration, accepts "n", then activates S_1. This proceeds until all symbols are processed and a report is generated.

[1] "*" indicates zero or more occurrences of the preceding element in the regular expression [1]. For example, "ab*c" matches "ac", "abc", and so on.

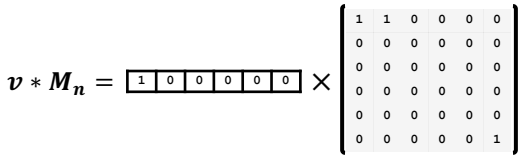


Fig. 2. An illustrative example of state vector and transition matrix multiplication for symbol "n".

2.2 BLAS-based Automata Processing

This section revisits how prior works [27, 64] represent AP as matrix-vector multiplications. While prior CPU-focused works exploit the associativity of matrix-vector multiplication for parallelism on multi-core CPUs, they lack fine-grained parallelism, making them unsuitable for GPUs. In contrast, this work focuses on fine-grained parallelism in matrix-vector multiplication on GPUs and examines bottlenecks in AP.

$$v_{\text{output}} = v \times M_{\text{a}} \times M_{\text{u}} \times M_{\text{t}} \times \ldots \times M_{\text{s}} \tag{1}$$

Section 2.1 shows that each matching iteration maintains active states using a boolean vector v of size n, where n is the number of states in the NFA. If a state is active, the corresponding bit in the vector is 1; otherwise, it is 0. We use a transition matrix for each symbol in the alphabet for the NFA. A transition matrix M_α for a symbol α is an $n \times n$ matrix. If a state S_i accepts the symbol α and transitions to S_j, then $M_\alpha[i][j] = 1$. A special case occurs in the starting state S_0: since it is an "always-active" starting state, thus $M_\alpha[0][0] = 1$. To compute the state vector in the next iteration when the incoming symbol is α, we multiply M_α by the current state vector v. This can be expressed as $v_{\text{next}} = v_{\text{current}} \times M_\alpha$. Thus, Equation (1) indicates how the final matching results between the NFA and the input stream "`automatanfablas`" are computed. Figure 2 presents the transition matrix for symbol "n" and the input state vector for the NFA in Figure 1(a) at iteration 8. Starting with the state vector `100000`, applying matrix M_n for binary **Vector-Matrix Multiplication** (**VMM**), results in `110000`, thereby activating states S_0 and S_1 at iteration 9.

2.3 Challenges of Adopting Standard BLAS to Automata

GPU vendors such as NVIDIA provide optimized libraries (e.g., cuSPARSE [10]) for efficient sparse BLAS operations. We prototype AP as SpMV multiplication using cuSPARSE as the start point of our study, treating automata as one large NFA and repeatedly invoking `cusparseSpMV` per input symbol (**RQ1**). However, our evaluation (Section 4.3) reveals that the implementation is inefficient. To more thoroughly answer the research question (**RQ2**), we analyze the performance bottlenecks identified within this implementation and present two major challenges, which will be elaborated on in the following discussion.

Challenge 1: Control overhead dominates the runtime. The baseline launches a separate kernel for each input symbol in streaming processing, requiring host-side preparation of auxiliary structures (including output reset, the descriptor setup, and creation of the external buffer for each incoming sparse matrix) mandated by cuSPARSE. The large number of device-level synchronizations and frequent GPU kernel launches introduce non-negligible overhead relative to the lightweight SpMV kernel computation. We quantify this *host-side overhead* as $H = (T_{\text{wall}} - \sum T_{\text{kernel}})/T_{\text{wall}}$, where T_{wall} is the wall-clock time of processing a fixed number of symbols on the host, and $\sum T_{\text{kernel}}$ is the sum of per-symbol kernel durations measured with CUDA events. One-time initializations (e.g., handle creation) are excluded; the dominant component of the host-side overhead is the GPU kernel launch latency (including a separate reporting result counting kernel), which occurs at processing every symbol. Figure 3 shows that over 40% of total time is spent on the host side for

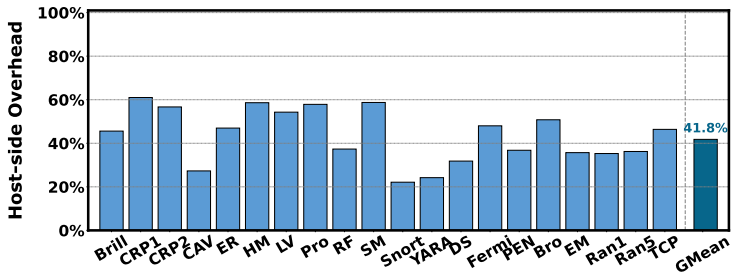


Fig. 3. Host-side overhead $H = (T_{\text{wall}} - \sum T_{\text{kernel}})/T_{\text{wall}}$ for `cusparseSpMV` in steady-state AP, with one-time initializations excluded in T_{wall}.

more than half of the benchmarks (average 41.8%), indicating that even after removing one-time costs, kernel launch and synchronization remain a significant fraction of runtime. cuSPARSE, focused on large-scale, device-level SpMV tasks, fails with CTA-level support, making it unsuited to the small, frequent automata computations. Custom kernels are required for CTA-level SpMV to reduce synchronization overhead.

Challenge 2: Inefficiencies in memory usage, data locality, and computation persist with CTA-level SpMV. Although CTA-level SpMV could mitigate control overhead, several critical inefficiencies remain, significantly limiting performance: (1) *Memory redundancy:* Transition matrices exhibit substantial redundancy across input symbols, resulting in a worst-case memory footprint proportional to $|\Sigma| \times |S|^2$, often exceeding GPU memory capacities (as detailed in Table 3, further discussed in Section 3.2). (2) *Poor data locality:* Default state numbering does not consider GPU memory access patterns, causing nonzeros processed by the same warp to scatter across memory. This leads to uncoalesced accesses and suboptimal warp efficiency. (3) *Redundant computation:* Computations on frequently recurring state vectors yield identical results across different input symbols, yet these computations are repeatedly executed, incurring unnecessary overhead (see Figure 9 for detailed results).

Why prior work is insufficient in the BLAS formulation? The four bottlenecks we identify are not intrinsic to AP. They arise when NFAs are cast into BLAS primitives. Traditional GPU AP systems use transition tables and worklists, which avoid some of these issues by design, but they do not operate through BLAS interfaces and thus cannot address the bottlenecks that appear only after the BLAS mapping (detailed in Table 2).

3 AutomataBLAS: Optimized Design for Efficient Sparse Matrix-based Automata Processing

This section introduces AutomataBLAS, an AP engine built with customized CTA-level BLAS.

3.1 Eliminating Kernel Launch Overhead via CTA-level Sparse Matrix-Vector Multiplication

Automata applications match input streams against multiple NFAs. Two properties motivate our design:

— NFAs operate independently and require no host synchronization, allowing each to use its own transition matrix.
— SpMV operations can run consecutively on each small NFA over the input, avoiding costly host-side overhead.

Table 2. BLAS-induced Bottlenecks, Why Prior AP Systems Did Not Fix Them, and our Solutions

Bottlenecks	Why prior AP techniques unsuitable to BLAS	Our solutions
High per-symbol launch cost	BLAS uses per-call kernels	Fuse and batch SpMV in a kernel
Duplicate transition matrices	AP engines compress tables, but BLAS materializes per-symbol matrices	Deduplicate matrices and map symbols to groups
Poor memory locality	Worklists process only active states; BLAS lacks frontier filtering	Interleaved state renumbering for locality
Redundant state vector computation	Prior work uses dynamic worklists, making reuse tracking difficult.	Cache state vectors and skip hits

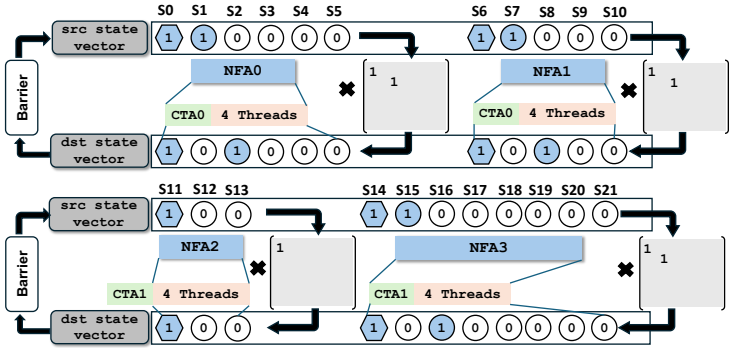


Fig. 4. Our CTA-level SpMV for AP, with "f" as the input symbol (see Figure 1(b)) in the current iteration. NFA0 represents the example NFA in Figure 1(a). Blue circles indicate active states. Group 0 includes NFA0 and NFA1; Group 1 includes NFA2 and NFA3.

These properties allow us to perform many small, parallel SpMV operations. While libraries like CUB and CUTLASS [8, 61] offer CTA-level primitives for intra-kernel parallelism, sparse libraries such as cuSPARSE only provide device-side control and host APIs, without support for batching these SpMV operations in a single kernel.

To address two challenges listed in Section 2.3 and enable an efficient BLAS-based automata system, a suite of holistic solutions are presented as follows:

— **Solution to Challenge 1**: To address the large host-side overhead, we design a CTA-level synchronization scheme. Each CTA manages SpMV independently for its assigned group of NFAs, enabling efficient parallel execution without costly host-level synchronization. Figure 4 illustrates the matching process: a single kernel launch with multiple CTAs, each independently matching input streams to assigned NFAs. CTAs update local state vectors via transition matrices as symbols arrive, with intra-CTA barriers ensuring consistency. This example shows 2 CTAs with 4 threads each; CTA-0 processes NFA0 and NFA1, CTA-1 handles NFA2 and NFA3. We adapt Bell and Garland's CSR-based SpMV [17] to the CTA level, allowing each thread block to process NFAs independently with low overhead. Each thread in the CTAs computes one output in the state vector by multiplying the input vector by a matrix row. This CTA-level design addresses Challenge 1 by eliminating control overhead. It not only streamlines processing but also enhances overall efficiency, as each automaton can execute its tasks without waiting for others to complete their operations.

— **Solution to Challenge 2**: Building on the CTA-level SpMV, we apply three complementary optimizations to further improve (*i*) memory usage by eliminating redundant transition

Table 3. Analysis of Total Transition Matrix Count and the Memory Usage (**MB**) of Transition Matrices

App.		Brill	CRP1	CRP2	CAV	ER	HM	LV	Pro	RF	SM
Original TMs	CSR	167.34	79.35	205.63	2,475.75	588.44	110.94	117.96	38.56	1,806.90	102.42
	Dense	1,048.64	1,482.67	10,353.10	485,210.00	18,882.50	1,581.05	3,225.02	73.06	62,968.90	402.78
Unique TMs	CSR	20.88	2.08	9.53	2,371.19	114.95	3.46	5.81	3.30	565.09	3.39
	Dense	132.27	28.96	363.98	465,591.00	3,663.73	30.88	62.99	6.05	19,897.30	14.17
App.		Snort	YARA	DS	Fermi	PEN	Bro	EM	Ran1	Ran5	TCP
Original TMs	CSR	290.96	1,193.59	101.69	72.77	47.60	2.74	12.88	13.01	13.16	22.40
	Dense	6,971.71	41,236.30	1,360.20	159.27	279.89	4.62	58.96	41.55	42.03	85.21
Unique TMs	CSR	42.15	713.61	30.26	10.37	11.42	0.19	1.91	1.65	1.63	2.69
	Dense	899.56	24,643.00	401.73	22.49	66.49	0.33	8.67	5.28	5.20	10.18

"Original_TMs" stands for all transition matrices including all duplicates. For a comprehensive overview of the benchmarks, please refer to Table 4 in Section 4.1.

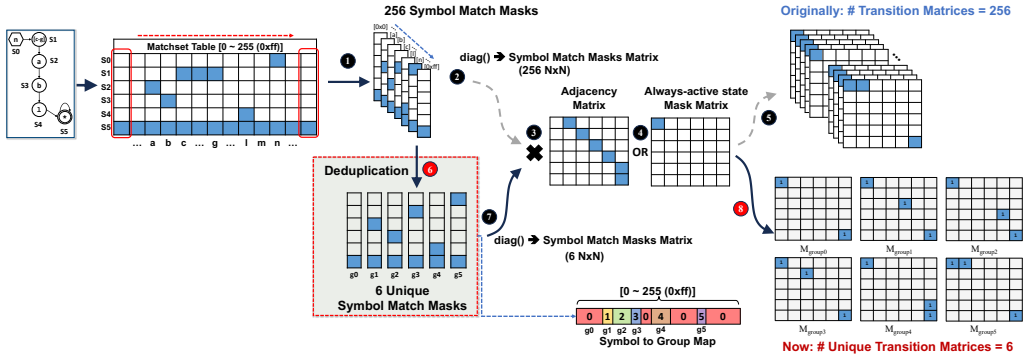


Fig. 5. Example of our transition matrix deduplication scheme using the NFA in Figure 1(a). For each input symbol (0–255), we first create a vector marking which states match the symbol (**Symbol Match Mask**), then turn it into a diagonal matrix by placing the vector's elements on the diagonal. Multiplying this diagonal mask with the adjacency matrix yields the corresponding symbol-specific transition matrix (dashed arrows). We observe that many symbols yield identical matrices; we group them and keep only the unique ones (solid arrows), producing 6 unique matrices in this example. A **Symbol-to-Group Map** records the mapping from each symbol to its corresponding unique matrix.

matrices (Section 3.2), (*ii*) data locality through interleaved state renumbering (Section 3.3), and (*iii*) computation efficiency via state-vector caching (Section 3.4).

3.2 Optimization #1: Reducing Memory Footprint via Transition Matrix Deduplication

To address the memory redundancy in **Challenge 2**, we observe that many transition matrices across input symbols share the same value, which can be eliminated through deduplication.

Analysis of Transition Matrix Construction. Table 3 shows the matrix memory usage in our CTA-level GPU baseline. The original matrices, even in CSR format (e.g., 1.8 GB for RF), often exceed typical GPU L2 cache capacity. This design limits data reuse due to the transition matrix's variation with incoming symbols. To address this issue, we first analyze the transition matrix construction process and illustrate how we identify matrix duplicates as shown in Figure 5. Figure 5 shows how to create 256 transition matrices for the NFA in Figure 1(a), each with 36 entries (6×6, $|S| = 6$). To construct a transition matrix, a value of 1 in position (i, j) of the boolean transition matrix M_α is determined by two conditions: (1) State S_i accepts the symbol α. (2) State S_i is connected to S_j in the NFA graph *or* state S_i is an always active state;

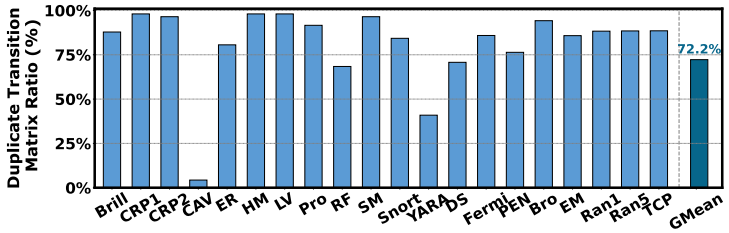


Fig. 6. Percentage of transition matrix deduplication.

As shown in Figure 5(❶) , each state is associated with a 256-bit *matchset*, then all states together can form a **Matchset Table**, where each row indicates symbols accepted by a state and each column represents states accepting a specific symbol. A formal representation of the NFA's transition matrix construction process, expressed in matrix language, is provided below:

$$M_{transition} = (M_{sym\_mask} \times M_{NFA\_adj}) \oplus M_{aas}. \tag{2}$$

We illustrate each item as follows:

— M_{sym_mask} represents the **Symbol Match Mask Matrix**. As shown in the figure, the original 256 symbol match mask matrices are derived from their corresponding **Symbol Match Masks** through diagonalization. Each symbol match mask is **one column** of the matchset table. Taking the match mask $[1, 0, 0, 0, 0, 1]^T$ for symbol "n" as an example, its *diagonalized* mask matrix is $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$ (Figure 5(❷)).

— M_{NFA_adj} represents the adjacency matrix of the NFA graph with encoded topology information.

— M_{aas} is also a diagonal matrix where the positions of the 1's indicate the **always active states** (**aas**). This information must be included in the final transition matrix to ensure these states activate for any input symbol.

As illustrated in Figure 5, the first condition is achieved by multiplying the diagonalization of a **Symbol Match Mask** (M_{sym_mask} in Figure 5(❷)) with the adjacency matrix of the NFA (M_{NFA_adj} in Figure 5(❸)). The second condition is done by performing a bitwise OR operation between the result matrix of the first step and the **Always-active State Mask Matrix** (M_{aas} in Figure 5(❹)). Then, a transition matrix is constructed (Figure 5(❺)).

Observing a large portion of duplicate **Symbol Match Masks**, our key insight is to *keep only one copy of each, thereby reducing the number of transition matrices.*

Transition Matrix Deduplication. We assign unique group IDs to each **Symbol Match Mask** (Figure 5(❻)). In this example, there are six unique masks, resulting in an equal number of unique transition matrices after applying the same transformation procedure (Figure 5(❽)). To ensure correctness after deduplication, each symbol is linked to its group using a **Symbol-to-Group Map**.

Table 3 shows the count of unique transition matrices and their total memory consumption achieved after the adoption of our deduplication strategy. Figure 6 presents the ratio of duplicate transition matrices across 20 evaluated benchmarks. We observe an average duplicate transition matrix ratio of 72.2%.

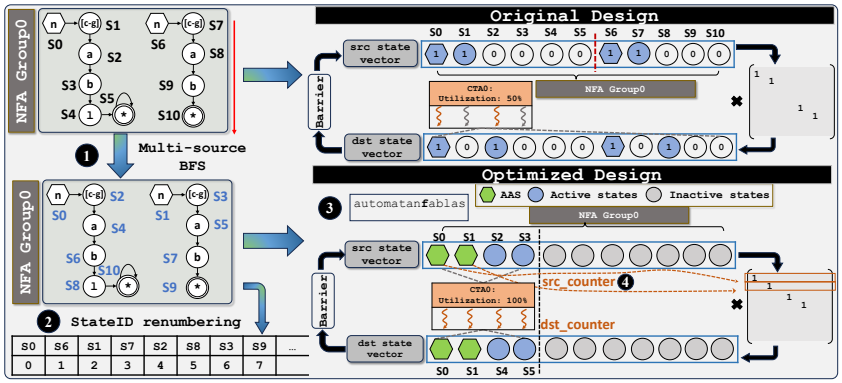


Fig. 7. An example of our interleaved state renumbering design to enhance data locality. "f" is the current input symbol.

3.3 Optimization #2: Improving Thread Utilization via Interleaved State Renumbering

To mitigate poor memory locality and warp inefficiency identified in **Challenge 2**, we redesign state numbering to improve memory coalescing.

In our CTA-level SpMV, each NFA state corresponds to a position in the state vector, with multiple NFAs potentially grouped as one, as described in Section 3.1. The position of the state vector is set to one when the state is active. During SpMV, threads in each CTA are assigned to state vector positions, *leading to thread underutilization since threads mapped to positions with a value of 0 have no tasks.*

Interleaved State Renumbering. Our key idea is to *reorganize frequently activated states so that they are stored as close together as possible.* This reorganization transforms the computation into a **Sparse Matrix Sparse Vector** (**SpMSpV**) Multiplication subprogram, where the matching process typically progresses from "shallow" states to "deep" states. Prior work [43] indicates that state activation frequency strongly correlates with **breadth-first search** (**BFS**) layers. To group frequently activated states in the state vector–ensuring all threads in a CTA have useful work–we renumber states using a BFS traversal order (Figure 7(❶)). We perform a multi-source BFS traversal for all NFAs in a group simultaneously, starting from each NFA's always-active state. As shown in Figure 7(❷), we introduce an extra NFA alongside the toy NFA depicted in Figure 1(a) to represent an NFA group, using the same input stream ("automatanfablas"), while processing the symbol "f" in the current iteration. A stack stores ID pairs in traversal order, such as [(S_0, 0), (S_6, 1), (S_1, 2), ...], providing an interleaved ordering. Each pair's value represents the new state ID for the original state. With an overall sparsity of 96.61% (geometric mean) observed across 20 benchmarks, derived from columns "#States", "#Always-active", and "#Start" in Table 4, this scheme utilizes a sparse format for the state vector instead of a dense array, with active states dynamically inserted on demand, reducing memory overhead and improving efficiency.

Maintain the Sparse Vector. Based on the newly generated order, we divide the locations in the sparse vector into three categories as shown in Figure 7: (1) aas, if applicable; (2) active states excluding aas; and (3) inactive states. In our implementation, the state vector layout places the aas on the left, other active states in the middle, and inactive states on the right, as shown in Figure 7(❸). In this layout, the new IDs within the aas region are strictly consecutive, while the IDs of the active states tend to be consecutive or close to the aas IDs.

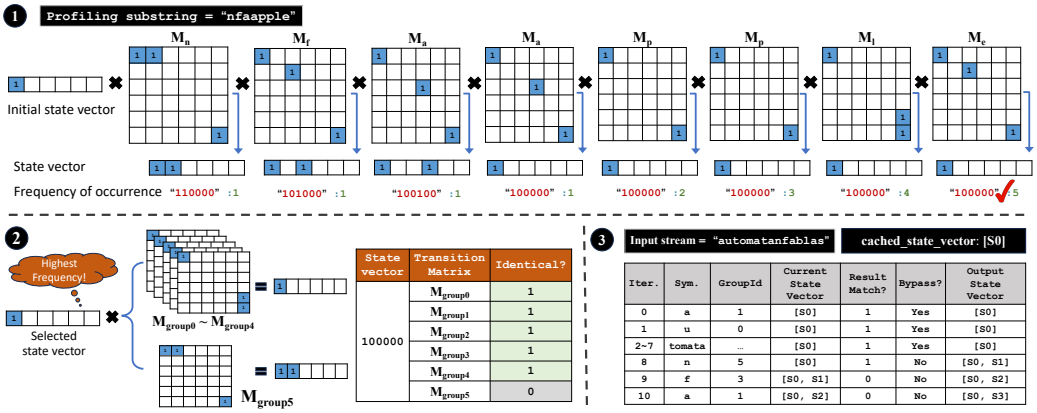


Fig. 8. An example of our state vector caching scheme: (1) Finding the highest frequency state vector via offline profiling; (2) Identify matrices resulting in the same result as the cached vector via offline pre-computation; (3) An illustrative example of the iterative runtime execution over the same input stream "automatanfablas".

Each thread in a CTA fetches state indices from the leftmost segment (the aas portion) of the input state vector. A shared counter (Figure 7(❸)) tracks the right boundary of active states, preventing access to inactive states (shown in gray). We maintain two state vectors (input and output) with the same layout. Since the aas portion is constant, we pre-write its results to the output vector. Thus, during each iteration, threads read the aas (green) and active regions (blue) but only update the active region in the output. We also use a shared *deduplication bitset* in on-chip memory to avoid duplicate state IDs in the output's active region. This strategy reduces unnecessary atomic operations on the boundary counter and further improves overall efficiency.

Figure 7 shows how the original always-active State IDs, 0 and 6, are renumbered to 0 and 1, improving spatial locality for accessing the state vector and transition matrices. In this iteration, their successors, States 2 and 3, are also active. This reorganization allows all four threads in CTA-0 to achieve coalesced access to the state vector, ensuring each thread performs meaningful work without accessing zero values, unlike the original design.

3.4 Optimization #3: Eliminating Redundant Computations via State Vector Caching

To reduce the redundant computation noted in **Challenge 2**, we cache and reuse frequently occurring state vectors when they yield the same results across multiple input symbols. The approach is inspired by two observations: First, *the resultant state vector often remains the same across many iterations, appearing repeatedly*. Second, *multiplying a transition matrix with a state vector frequently produces an identical state vector*. These insights help bypass redundant multiplications, thereby boosting efficiency.

State Vector Caching. To maximize caching opportunities–given that caching requires considerable memory–we propose the most frequent state vector selection approach. We analyze the 1 KB prefix of the real input data in each benchmark application for profiling, following the approach used in prior work [43]. We record the frequency of each output state vector for each NFA. Figure 8(❶) illustrates this process. We utilize a `state_vector_freq_map` to track state vector frequencies, with keys as bitsets and values as occurrence counts. For example, with the input stream "nfaapple" and initial state vector `100000`, we precompute the results of the vector multiplied by the corresponding transition matrices for the same NFA in Figure 1(a) and identify four unique pairs. The state vector `100000`, occurring five times, is selected for caching.

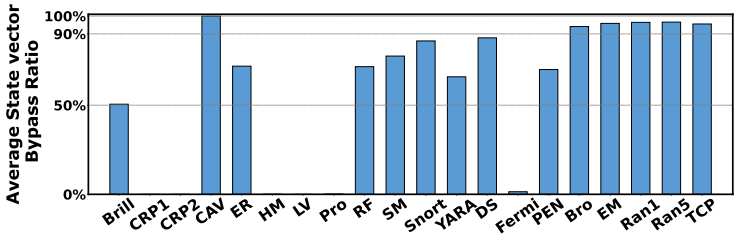


Fig. 9. Average vector-matrix computation bypass ratio across all NFAs based on profiling.

Next, we determine if each transition matrix multiplied by the selected state vector produces the same vector as shown in Figure 8(❷). This is achieved through offline pre-computation, storing results in a bitvector (`record_matrix_group_map`), where the vector's dimension corresponds to the number of unique transition matrices. A matrix that produces the same state vector is marked as 1; otherwise, it is marked as 0. Assuming that there is only one NFA within the NFA group, we check the multiplication results for six unique matrices with the state vector `100000`. Matrices 0 to 4 generate the same vector, resulting in the bitvector `111110` for runtime use. The selected state vector, stored as a sparse array ("`[S0]`" in sparse format, "sparse" means that only non-zero elements are explicitly stored), is also transferred to the GPU.

Runtime Computation with Cached Results. During runtime, each iteration requires checking if the current state vector matches the cached result. Since each GPU thread corresponds to a single state and cannot independently determine a match, the threads must work *collectively*. We use a boolean flag in shared memory for this purpose. Each thread fetches its state index and performs a binary search in the cached state vector (`cached_state_vector`). If the index is not found, the flag is set to `false`. After a CTA-level barrier, a thread checks the flag's status. If the flag is false, indicating a failed comparison, we perform the regular multiplication computation. If true, we then check if multiplying the transition matrix for the incoming symbol α by the state vector yields the same state vector. This step uses `record_matrix_group_map` and **Symbol-to-Group Map** (as described in Section 3.2). If the bit in `record_matrix_group_map` at index `Symbol_to_Group_map`[α] is 1, the state vector result remains unchanged, allowing us to reuse input values in the next iteration without extra computations. Otherwise, standard matrix-vector multiplication should be performed. Figure 8(❸) shows the runtime results for each iteration under our state vector caching strategy, which efficiently bypasses matrix-vector multiplication in 8 of the 11 iterations, as detailed in the table. We define the *bypass ratio* as the ratio of iterations in which matrix-vector multiplication is skipped. Figure 9 shows that over 70% of matrix-vector multiplications can be skipped for most of these evaluated applications in each NFA group for the profiled 1 KB input.

In addition, we conduct an in-depth sensitivity analysis to evaluate the impact of different profiling strategies on online processing performance. This analysis allows us to understand the nuances of how each profiling approach affects the overall performance of our matrix-based automata framework. Refer to Section 4.6 for further discussion.

4 Evaluation

4.1 Evaluation Methodology

System Configurations. Our experiments use an NVIDIA RTX 3090 GPU (Ampere architecture [6], 24 GB memory, 6 MB L2 cache, 82 SMs) with an Intel Xeon 4214R CPU (12 cores, 128 GB memory) on Ubuntu 20.04 (kernel 5.15.0). AutomataBLAS is compiled with GCC 9.4, CUDA 12.1 with `-O3` flag. GPU kernel time is measured via CUDA events, averaged over three runs, excluding

Table 4. Characteristics of Evaluated NFA Applications

Suite	Idx	Application	Abbr.	#States	#Always-active	#Start	#Report	#CC	Max CC Size	Avg CC Aize
AutomataZoo [67]	1	Brill	Brill	115,549	5,946	0	5,946	5,946	40	19.43
	2	CRISPR_CasOFFinder	CRP1	74,000	4,000	0	2,000	2,000	37	37.00
	3	CRISPR_CasOT	CRP2	202,000	4,000	0	2,000	2,000	101	101.00
	4	ClamAV	CAV	2,374,717	33,171	0	33,667	33,171	22,075	71.59
	5	EntityResolution	ER	413,352	10,000	0	10,000	10,000	75	41.34
	6	Hamming_N1000_l18_d3	HM	108,000	2,000	0	2,000	1,000	108	108.00
	7	Levenshtein_l19d3	LV	109,000	4,000	0	4,000	1,000	109	109.00
	8	Protomata	Pro	24,103	1,302	8	1,321	1,309	123	18.41
	9	RandomForest_20_400_200	RF	992,000	16,000	0	16,000	16,000	62	62.00
	10	SeqMatch_BIBLE_w6_p6	SM	51,570	0	10,314	1,719	1,719	30	30.00
	11	Snort	Snort	202,043	2,507	644	3,246	2,486	4,509	81.27
	12	YARA	YARA	1,047,528	23,537	2	23,583	23,530	1,017	44.52
ANMLZoo [65]	13	Dotstar	DS	96,438	2,837	0	2,838	2,837	95	33.99
	14	Fermi	Fermi	40,783	0	2,399	2,399	2,399	17	17.00
	15	PowerEN	PEN	40,513	2,857	0	3,456	2,857	52	14.18
Regex [16]	16	Bro217	Bro	2,312	187	0	187	187	84	12.36
	17	ExactMatch	EM	12,439	297	0	297	297	87	41.88
	18	Ranges1	Ran1	12,464	297	0	297	297	96	41.97
	19	Ranges05	Ran5	12,621	299	0	299	299	94	42.21
	20	TCP	TCP	19,704	756	0	767	738	391	26.70

CC stands for "connected component".

Table 5. Overview of the Evaluated Schemes

Schemes	Descriptions
GPU-NFA [44]	GPU AP with hot-cold states scheduling
NFA-CG [78]	GPU AP based on "compatible groups"
AutomataBLAS	Our CTA-level SpMV design with three proposed optimizations
VASim [66]	NFA processing engine on CPUs
Spada [41]	A SpGEMM accelerator adapted to NFA processing
cuSPARSE-SpMV [10]	A naïve implementation using cuSPARSE library
***ng*AP** [29]	Batch-mode GPU AP engine with batch size x bytes
AsyncAP [45]	Enable matching from different input locations asynchronously

automata loading and input preparation, which are amortized over long input streams. Throughput is the number of input symbols processed per second. We also use an NVIDIA RTX 4090 GPU (Ada Lovelace architecture [7], 24 GB memory, 72 MB L2 cache, 128 SMs) for portability studies.

Benchmarks. We evaluate 20 applications sourced from three benchmark suites: AutomataZoo [67], ANMLZoo [65], and Regex [16]. Table 4 details the characteristics of the evaluated applications, all using a 1 MB input stream from their benchmark suites.

Evaluated Schemes. Table 5 summarizes the evaluated schemes used in our experiments. Our extensive evaluation is comprised of nine key components, each serving a crucial role in our overall assessment. This allows us to delve deeply into various facets of the topic, ensuring that our analysis is both thorough and comprehensive. Our evaluation methodology is organized as follows:

— First, in our evaluation of GPU implementations, we focus on comparing our work with two prior open-source data-driven designs: GPU-NFA [44] and NFA-CG [78]. Each of these three studies operates under the "one-symbol-at-a-time" paradigm, which is known as streaming mode execution. This alignment allows for a fair comparison, with GPU-NFA serving as the baseline for the overall performance analysis (Section 4.2).
— Second, we evaluate the performance of our tailored BLAS-based approach by comparing it to a CPU-based automata simulator, namely VASim [66]. This analysis is designed to evaluate how effectively our method addresses automata workloads in contrast to a specialized automata accelerator. This comparison helps us understand the strengths and limits of our approach within the context of AP.

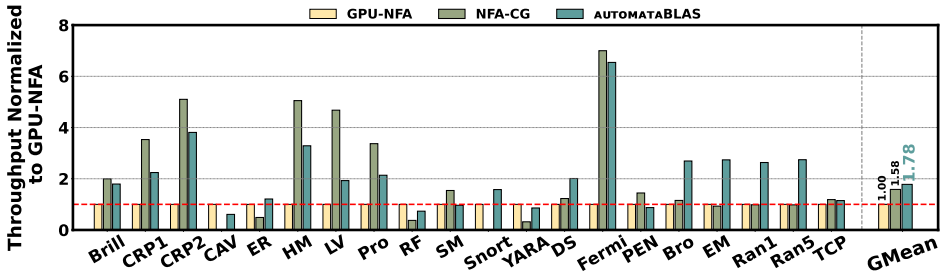


Fig. 10. Performance comparison with existing GPU-based automata engines under streaming mode processing. NFA-CG [78] fails with Snort and CAV.

— Third, to demonstrate the effectiveness of our proposed strategies, we evaluate the performance of BLAS-based domain-specific accelerators in managing NFA workloads. By leveraging their highly optimized BLAS routines, we show that without automata-specific optimizations, even advanced BLAS accelerators are significantly less efficient than our GPU-based designs. We use a state-of-the-art SpMM accelerator (SPADA [41]), adapting it to the NFA processing and comparing the performance with our proposal.
— Fourth, we utilize CUSPARSE-SPMV [10] as a baseline used in our breakdown analysis (Section 4.3).
— Fifth, to showcase the memory efficiency of our approaches, we compare them against *ng*AP [29], which utilizes "symbol level parallelism" for batch processing, constrained by a limited batch size to ensure fairness (Section 4.4). Batch size, in this context, is the number of symbols that NFAs process at a time. Notably, *ng*AP demands significant memory space to accommodate the extensive intermediate matching results generated during batch processing.
— Sixth, we compare our work with AsyncAP [45], which also implements batch mode processing by matching from different input locations in the input stream (Section 4.5).
— Seventh, we perform sensitivity analysis on state vector selection, the CTA count, and block size configuration, optimizing for the best performance (Section 4.6).
— Eighth, we analyze the preprocessing overhead to present a detailed picture of our approach's practicality (Section 4.7).
— Lastly, we still consider GPU-NFA as our comparative work and leverage a different GPU to investigate the portability of our proposed optimizations (Section 4.8).

4.2 Performance Results

This section compares our performance with existing works. We first compare our performance with GPU automata frameworks, then evaluate AUTOMATABLAS improvements over VASIM, and finally evaluate our results against a BLAS-based accelerator, SPADA.

Comparison with GPU-based Automata Engines. As seen in Figure 10, throughput results are normalized to GPU-NFA . Key observations include: (1) On average, AUTOMATABLAS with optimization schemes achieves a 1.78× performance improvement over GPU-NFA . (2) AUTOMATABLAS outperforms GPU-NFA in 15 of the 20 applications, with a maximum speedup of 6.54× on Fermi. This boost is attributed to the fact that the reporting states in these applications typically occur at deeper iteration levels. While GPU-NFA enhances memory efficiency by loading intermediate data into GPU registers, the increasing depth of traversal generates a number of intermediate results that cannot be fully accommodated in the limited register file resources. Consequently, GPU-NFA experiences memory inefficiency. Conversely, AUTOMATABLAS enhances data locality and mitigates

Table 6. Throughput (MB/s) Comparison with a CPU NFA Simulator, and a DSA for SpMM

App.	Brill	CRP1	CRP2	CAV	ER	HM	LV	Pro	RF	SM
VASim	0.079	0.086	0.036	0.028	0.054	0.065	0.025	0.532	0.015	0.133
Spada	0.006	0.009	0.003	0.001	0.002	0.005	0.004	0.026	0.006	0.023
Ours	0.886	1.008	0.870	0.677	1.926	0.892	0.398	1.112	0.779	1.054
App.	**Snort**	**YARA**	**DS**	**Fermi**	**PEN**	**Bro**	**EM**	**Ran1**	**Ran5**	**TCP**
VASim	0.322	0.025	0.538	0.040	0.556	8.701	6.391	6.184	6.146	2.492
Spada	0.005	0.003	0.058	0.013	0.049	1.182	0.990	0.996	0.976	0.149
Ours	0.833	0.678	2.201	1.141	0.943	3.949	4.699	4.326	4.516	1.590

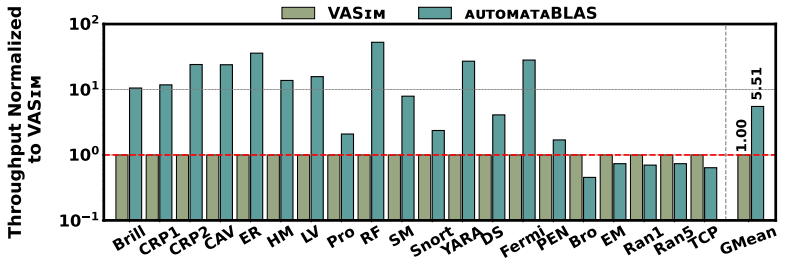


Fig. 11. Performance comparison of AutomataBLAS with VASim [66].

these issues; On Fermi, we achieve an average reduction of over 80% in matrix memory usage using our **Opt.#1**, as shown in Figure 6, alongside a remarkable reduction in total global memory requests of over 10× from our **Opt.#2** (see Figure 15). Both of these techniques significantly optimize memory efficiency. (3) Some applications, such as CAV, RF, SM, and PEN, the performance for our approach lag behind GPU-NFA because they match states at shallow levels, enabling GPU-NFA to perform most computations in registers. (4) In contrast to NFA-CG, AutomataBLAS excels in 10 of the 20 applications. For the smaller benchmark set (Regex and ANMLZoo), we achieve a maximum speedup of 2.94× (Bro), and 2.70× on YARA in the larger set (AutomataZoo). NFA-CGintroduces the idea of *compatible groups*, wherein states that fall within the same group cannot be active simultaneously. This enables the assignment of a compatible group to a single thread, which improves overall thread utilization. However, this method for computing compatible groups is quite costly, with a time complexity that reaches at least quadratic in the number of NFA states. Therefore, NFA-CGexcels in CRP1, CRP2, HM, LV with fewer compatible groups but faces high preprocessing overhead (e.g., **2,288.21**s vs. **17**s for LV) and fails in CAV and Snort. Overall, these results demonstrate that AutomataBLAS performs well against the compared works, and a more in-depth analysis of the performance improvement sources is detailed in Section 4.3.

Comparison with CPU Automata Engine (VASim). As shown in Table 6, VASim achieves higher throughput than AutomataBLAS in some smaller applications like Bro, due to limited parallelism and lower computational intensity. However, Figure 11 shows that AutomataBLAS outperforms VASim in 15 large applications, with over 50× throughput increase on RF and an average 5.51× boost, exhibiting superior efficiency for large-scale automata.

Comparison with BLAS Accelerator (Spada). We begin the illustration with the motivation for the evaluation, followed by the methodology:

— **Motivation of this comparison.** We conduct this experiment to investigate the behavior of our NFA workloads on the general-purpose sparse matrix accelerator, Spada [41], which is

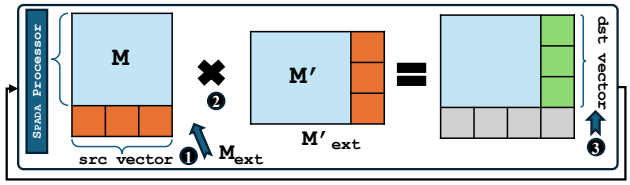


Fig. 12. Overview of adapting SPADA to NFA processing.

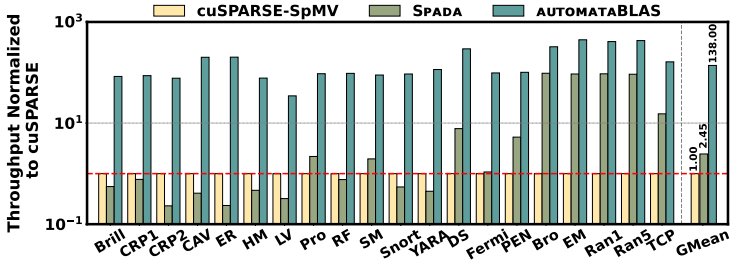


Fig. 13. Performance comparison of AUTOMATABLAS with SPADA [41] for NFA processing.

not specifically optimized for NFA processing. This comparison highlights that even under optimistic hardware assumptions for SPADA, AUTOMATABLAS achieves significant gains on commodity GPUs, underscoring its practical performance advantage and adaptability across architectures. Our SPADA comparison serves two purposes. First, it reveals that automata-specific optimizations such as matrix deduplication, interleaved state renumbering, and state vector caching are critical for high performance. Even when our compute paradigm is mapped to a DSA, the lack of these optimizations leads to substantial inefficiencies. Second, it provides insights for DSA designers. If a sparse matrix accelerator is to be repurposed for automata workloads, it will likely face similar issues with redundant memory usage, poor locality, and redundant computation. Our results indicate that addressing these issues is essential for competitive performance.

— **Methodology.** The current SPADA [41] core supports only BLAS operations involving a matrix and itself or its transpose. To enable NFA processing without changing the SPADA internals, we adapt our inputs to match its interface. In each iteration, we construct an extended matrix M_{ext} (Figure 12❶) that embeds both the symbol transition matrix and the current state vector. We then multiply M_{ext} with its transpose inside the SPADA core (Figure 12❷) and extract the updated state vector from the output (Figure 12❸) for the next iteration. To estimate compute-bound throughput, we count SPADA core cycles only and exclude preprocessing time. Even under these favorable assumptions for the BLAS accelerator, our GPU design achieves an average speedup of 56.33× and up to 965× on ER (see Figure 13 and Table 6), highlighting the importance of application-specific optimizations.

4.3 Effectiveness of Optimizations

To evaluate our proposed approaches, each technique is added incrementally as detailed in Table 7, with throughput normalized to baseline implementation using `cuSPARSE-SpMV` shown in Figure 14. To minimize kernel launch overhead in the `cuSPARSE-SpMV` implementation, we introduce an optimized version using CUDA Graphs [9]. The results indicate that this technique can improve the `cuSPARSE-SpMV` baseline by an average of 1.5×. Our main observations are made as follows: (1) By

Table 7. Incremental Evaluation of AutomataBLAS.

Schemes	Solutions				
	Single CPU Launch	CTA-level SpMV	Opt. #1	Opt. #2	Opt. #3
cuSPARSE-SpMV					
cuSPARSE-SpMV-CUDA Graphs	✓				
AutomataBLAS-baseCOO	✓	✓			
AutomataBLAS-baseCSR	✓	✓			
AutomataBLAS-O^1 (CSR)	✓	✓	✓		
AutomataBLAS-O^1O^2 (CSR)	✓	✓	✓	✓	
AutomataBLAS-$O^1O^2O^3$ (CSR)	✓	✓	✓	✓	✓

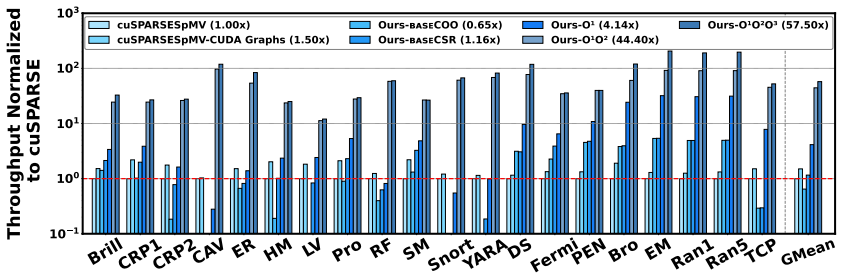


Fig. 14. Overall throughput results normalized to the naïve implementation using cuSPARSE SpMV.

implementing CTA-level SpMV, AutomataBLAS-baseCSR delivers a 1.16× speedup compared to `cuSPARSE-SpMV`, mitigating kernel launch overhead. Unlike cuSPARSE-SpMV, which involves n kernel launches for n symbols, our method requires just one kernel launch. As a result, the kernel launch overhead becomes negligible as n increases. For an 1 MB input, AutomataBLAS incurs a kernel launch overhead of only 1.3×10^{-5}s, representing just 2.2×10^{-6}% of `cuSPARSE-SpMV`'s overhead (7.78s) on average. (2) AutomataBLAS-O^1 achieves an average speedup of 4.14× across 20 benchmarks, thanks to enhanced memory efficiency from matrix deduplication. Taking `Fermi` as an instance, we demonstrate that our **Opt.#1** leads to an average reduction exceeding 80% in matrix memory usage, illustrated in Figure 6, and delivers a performance enhancement of 6.49× with a largely reduced memory footprint. Our deduplication results in a L2 cache hit rate increase from 72% to 80% on average, compared to `cuSPARSE-SpMV`. Furthermore, we have evaluated the potential benefits of applying the L2 cache residency control [11]. In our CTA-level design, transition matrices for each NFA group are sequentially ordered by symbol, with each CTA accessing a distinct matrix segment (see Section 3.1). Accessing larger matrix segments increases L2 miss chances. To reduce this on the RTX 3090 GPU, we allocate 4 MB of its 6 MB L2 cache to the most frequently accessed portion of the longest segment. However, this yields only a 2% performance gain, likely due to our high L2 cache hit rate and irregular access patterns. This suggests that reducing memory footprint is more effective. (3) AutomataBLAS-baseCOO, AutomataBLAS-baseCSR, and AutomataBLAS-O^1 struggle with `CAV`, `YARA`, and `Snort` due to NFA size imbalances, resulting in performance degradation, which can be further improved by AutomataBLAS-O^1O^2. (4) AutomataBLAS-O^1O^2 significantly boosts throughput by 44.4×, improving data locality and thread utilization via state renumbering, accessing the useful states in a coalesced manner. For instance, AutomataBLAS-O^1O^2 achieves a reduction of over 33× and 20× in the total number of global requests and global transactions (see Figure 15), respectively. This shows our **Opt.#2**'s effectiveness in substantially reducing excessive memory accesses relative to AutomataBLAS-O^1 and AutomataBLAS-baseCSR. (5) AutomataBLAS-$O^1O^2O^3$ outperforms `cuSPARSE-SpMV` by 57.5× and AutomataBLAS-O^1O^2 by 1.35× (geometric mean). Notably, `EM` achieves a 2.24× speedup over AutomataBLAS-O^1O^2 via bypassing 95% of computations (Figure 9) with our state vector caching scheme.

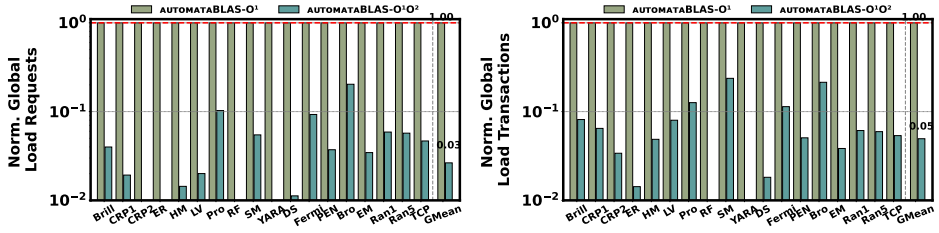


Fig. 15. Profiled memory results between AutomataBLAS-O¹ and AutomataBLAS-O¹O².

Table 8. Complexity Analysis with *ng*AP [29]

Schemes	Time Complexity	Space Complexity
*ng*AP [29]	$O(n^2m)$	$O(\|\Sigma\|^d * m + (m+E) + n \times m)$
AutomataBLAS	$O(nm)$	$O(\|\Sigma\| \times (m+E))$

n: #input symbols, *m*: #NFA states, *E*: #edges in NFA graph, *d*: memoized prefix length introduced in *ng*AP.

Overall, this analysis validates our schemes as effective solutions and answers the research questions (**RQ1**, **RQ2**, and **RQ3**).

4.4 Comparison with *ng*AP

AP systems are generally categorized into two execution models: stream mode and batch mode. Stream mode systems process one symbol at a time, updating automata states immediately, offering low per-symbol latency and reduced memory footprint, fitting latency-sensitive or memory-limited scenarios.

In contrast, batch mode collects multiple input symbols and processes them together in a single iteration to expose greater parallelism. While batch mode can maximize hardware utilization when sufficient resources are available, it introduces input buffering latency, increases memory consumption, and complicates the handling of partial pattern matches that span across batch boundaries. Recent work, such as *ng*AP [29], adopts a batch mode design. We analyze the time and space complexity of *ng*AP and make a comparison with our approach as shown in Table 8. For time complexity, our NFA processing maintains $O(nm)$ complexity by handling one symbol at a time, unlike *ng*AP, which incurs $O(n^2m)$ complexity due to repeated pattern matching at different locations of the input symbols. On the other hand, *ng*AP requires $O(|\Sigma|^d * m + (m+E) + n \times m)$ space, with terms from the memoization table, NFA topology, and the worklist to accommodate the intermediate results. Therefore, our worst-case complexity analysis in Table 8 shows that *ng*AP incurs redundant computations.

Table 8 indicates an upper bound of time/space complexity for each approach. However, actual performance is input-dependent. Deduplication and repeated state vectors reduce realized operations below the bound.

(1) **Work-parallelism trade-off.** Throughput depends on

$$T \approx \frac{\text{work}}{\text{available parallelism}} + \text{overhead}.$$

AutomataBLAS reduces per-symbol work in streaming mode, focusing on NFA-level parallelism. In contrast, *ng*AP leverages *symbol-level* parallelism, which introduces additional work but achieves higher throughput when the batch size (K) is large and sufficient memory is available.

Table 9. Comparison with *ng*AP [29] for Different Input Batch Sizes (Bytes), Including Performance and GPU Memory Usage (MBs)

App.		Brill	CRP1	CRP2	CAV	ER	HM	LV	Pro	RF	SM
Norm. Perf.	*ng*AP-8	0.30	0.73	0.50	1.40	0.40	0.74	0.35	0.49	0.71	0.84
	*ng*AP-32	0.77	2.33	1.76	4.64	0.94	3.25	0.86	1.47	0.98	0.84
	*ng*AP-128	1.93	6.04	3.70	10.84	2.34	4.47	1.44	3.50	1.13	0.84
GPU Mem.	*ng*AP-128	**18,798**	**23,289**	**23,294**	**17,150**	**16,866**	**17,164**	**20,784**	**17,624**	**20,526**	**16,760**
	Ours	**41**	**4**	**10**	**2,573**	**190**	**6**	**11**	**7**	**832**	**3**
App.		**Snort**	**YARA**	**DS**	**Fermi**	**PEN**	**Bro**	**EM**	**Ran1**	**Ran5**	**TCP**
Norm. Perf.	*ng*AP-8	0.08	0.97	0.29	0.46	0.47	0.33	0.31	0.32	0.29	0.45
	*ng*AP-32	0.10	2.50	0.45	0.46	0.64	0.93	1.00	0.99	0.98	0.79
	*ng*AP-128	0.09	4.18	0.70	0.46	0.65	2.05	2.18	2.13	2.21	1.30
GPU Mem.	*ng*AP-128	**17,276**	**15,561**	**16,768**	**17,906**	**17,906**	**16,702**	**16,754**	**16,754**	**16,752**	**16,756**
	Ours	**99**	**80**	**3**	**2**	**15**	**1**	**3**	**3**	**3**	**5**

Performance results are normalized to AutomataBLAS.

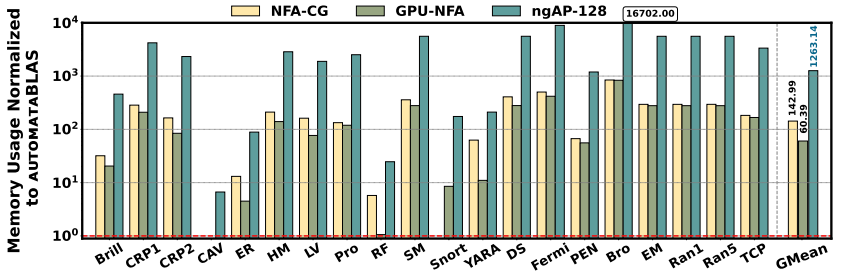


Fig. 16. Memory usage comparison across GPU-based engines against AutomataBLAS.

(2) **When each wins.** Batch mode (*ng*AP) typically performs better with large K, high active-state density, or significant symbol repetition, particularly with adequate memory and relaxed latency. Streaming mode (AutomataBLAS) is preferable when memory is constrained, latency matters, K is small, or inputs are sparse.

We have evaluated AutomataBLAS against three different batch sizes of *ng*AP (denoted as *ng*AP-*x*), with results summarized in Table 9. We make two observations:

— For small batch sizes, AutomataBLAS matches or exceeds *ng*AP's performance except `CAV`. Particularly, in applications such as `Snort` and `Fermi`, AutomataBLAS consistently outperforms *ng*AP. On `Snort`, we observe over a 10× performance improvement across different batch sizes. *ng*AP relies on aas for prefetching, absent in `Fermi` (refer to Table 4), whereas AutomataBLAS works without such assumptions.
— *ng*AP boosts throughput with larger batch sizes (e.g., 128 symbols) but incurs significant memory overhead. Figure 16 presents that *ng*AP-128 requires over **1,263**× (up to over **16,700**×) more memory on average across 20 applications. Table 9 shows *ng*AP-128 offers 1.44× higher throughput, but uses over **20 GB** GPU memory, compared to only **11 MB** used by AutomataBLAS for `LV`.

Overall, batch mode achieves high throughput at the cost of high memory consumption and relaxed latency constraints. Our stream mode approach offers a balanced alternative: *competitive performance, superior memory efficiency, lower latency, and better worst-case complexity across diverse automata workloads.*

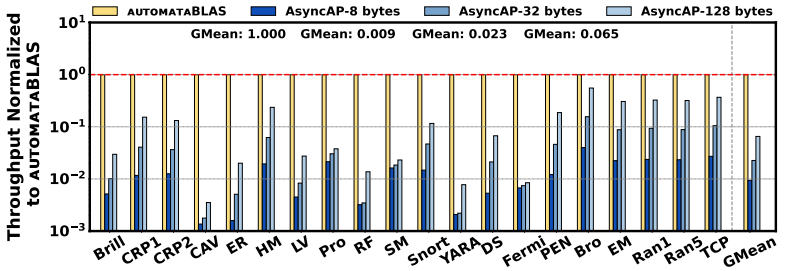


Fig. 17. Performance comparison of AsyncAP [45] against AutomataBLAS on an RTX 3090 GPU.

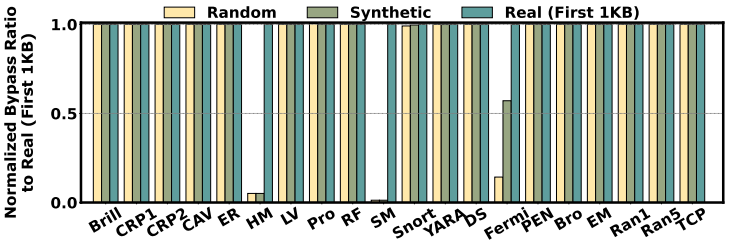


Fig. 18. Sensitivity of normalized bypass ratio on most frequent state vector selection methods: random input, synthetic input, and 1 KB prefix profiling.

4.5 Comparison with AsyncAP

As AsyncAP [45] is developed for batch processing like *ng*AP, we have included an experiment to compare its performance with AutomataBLAS at different batch sizes (AsyncAP-*x*). Batch sizes are consistent with those in the *ng*AP comparison (see Table 9). We make the following observations: With batch sizes ranging from 8 bytes to 128 bytes, AutomataBLAS consistently outperforms AsyncAP as shown in Figure 17, and achieves average speedups between 100× and 15.4×. The primary factor contributing to AsyncAP's slower performance is its reliance solely on symbol parallelism without employing state-level parallelism. When the batch size is reduced, parallelism decreases significantly, resulting in poorer performance. On the other hand, *ng*AP uses both state-level and symbol-level parallelism to enhance efficiency. Overall, the experiment validates the proposed approaches for our BLAS-based automata framework.

4.6 Sensitivity Studies

Sensitivity to State Vector Selection. We conducted a sensitivity analysis on state vector selection strategies: (1) "Random" creates 1 KB of symbols randomly from real input; (2) "Synthetic" generates 1 KB with the same frequency distribution as the first 1 KB of real input; (3) our design strategy from Section 3.4. As shown in Figure 18, all three strategies yield substantial bypass ratios across most applications, indicating that our caching scheme is *generally robust* to state vector selection. Among them, our "1 KB prefix" consistently achieves slightly higher bypass ratios, reflecting higher-quality profiling and making it a preferred default.

Sensitivity to Block Size and CTA Count. We have carried out sensitivity analysis experiments to assess how thread block size impacts performance. **(1) Methodology:** We assign one NFA per CTA and vary the block size from 32 to 1024. Figure 19 presents the normalized performance results. **(2) Results:** Our findings consistently show that performance degraded as the block size increased

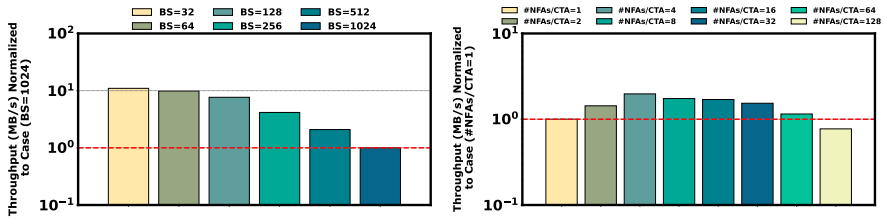


Fig. 19. Performance sensitivity to thread block size and the CTA count. Results shown in the figure are averaged with the geometric mean across all applications.

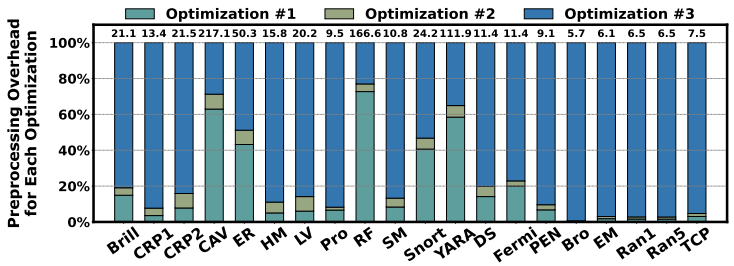


Fig. 20. Analysis of preprocessing overhead on an Intel Xeon 4214R CPU, with total time (seconds) showing above each bar.

across all applications tested under this setting. **(3) Analysis:** Our profiling shows performance decline mainly from increased `stall_sync` at CTA barriers. Threads have a very low utilization with one NFA per CTA (e.g., less than 2% for `Brill`). This behavior amplified `__syncthreads()` overhead as block size grows, suggesting the kernel is latency-bound due to threads waiting at synchronization points.

In our CTA-level design, NFAs are grouped for each CTA processing, with granularity affected by CTA count changes. **(1) Methodology:** We evaluate this by varying NFAs per group from 1 to 128, altering CTA counts as total NFAs remain fixed. **(2) Results:** Performance generally improves as NFAs per CTA increase from 1 to 4 but declines beyond 8, as shown in the right figure of Figure 19. This indicates a tradeoff between thread utilization and overall occupancy. **(3) Analysis:** Our profiling results reveal that a low #NFA/CTA ratio results in high occupancy but poor thread utilization, limiting performance. Increasing the ratio boosts performance, but a too-large ratio indicates too few CTA counts, yielding a low occupancy (e.g., under 10% for `Brill` when #NFA/CTA>=64), which becomes a new bottleneck.

4.7 Analysis of Preprocessing Overhead

We have thoroughly analyzed the preprocessing overhead of our three optimizations: (1) *transition matrix deduplication*, (2) *interleaved state renumbering*, and (3) *state vector caching*. As shown in Figure 20, the average execution time across 20 benchmarks is 37.33 seconds, indicating that the overhead is small and can be performed once before online streaming begins. The `CAV` application, the most time-consuming, requires under 3.6 minutes for the entire preprocessing. For **Opt.#1**, which includes **Symbol2Group Map Generation** and **Unique Matrix Construction**, 17 of the 20 applications finish in under one minute, with `CAV` taking the longest due to its largest matrix size. **Opt.#3** involves profiling the first 1 KB of input and takes up to one minute for `CAV` because

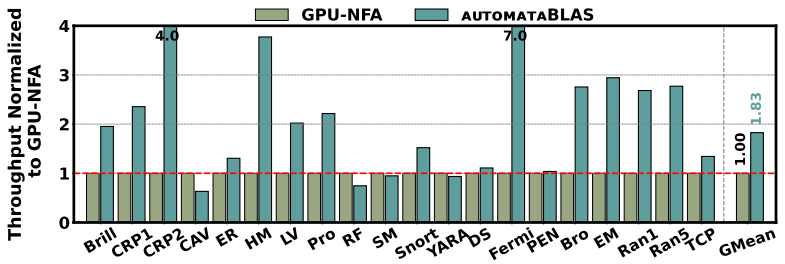


Fig. 21. Performance comparison on RTX 4090 GPU.

of its highest computational demand. Although **Opt.#3** has the highest overhead percentage, it is negligible as it is a one-time cost amortized over multiple streaming executions.

4.8 Portability Studies

We further compare AutomataBLAS and GPU-NFA on an RTX 4090 GPU, where AutomataBLAS outperforms GPU-NFA by 1.83× (see Figure 21). Performance on the RTX 4090 yields a higher speedup (1.83× vs. 1.78×) than on the RTX 3090 (Figure 10) across 20 applications, demonstrating AutomataBLAS's better scalability primarily due to the higher number of SMs (128 vs. 82) and increased integer capabilities [6, 7].

5 Related Work

This section reviews literature on parallel AP and sparse linear algebra acceleration, covering techniques for systems and accelerators.

Parallel Automata Processing. Several studies have developed efficient AP on GPUs. iNFAnt by Cascarano et al. [21] lacks data movement optimizations. iNFAnt2 [65] improves it with an alphabet-oriented transition table. NFA-CG by Zu et al. [78] uses compatible groups to enhance utilization with heavy preprocessing cost, and poor scaling for large-scale automata. GPU-NFA [44] optimizes data movement by loading topology data of aas into GPU registers. AsyncAP [45] targets symbol-level parallelism but faces workload imbalances and inefficient memory use for transition tables. *ng*AP [29] enhances data locality with new worklist scheduling techniques, with significant memory overhead due to their worklist management and prefix memoization. Vu *et al.* [64] propose a matrix-based approach for automata using the matrix operations to enable task-level parallelism on a GPGPU. HybridSA [38] exploits a bit-parallel design and distributes regular expression matching between CPUs and GPUs. BitGen [28] is a GPU-accelerated regex engine that compiles regexes into bitstream programs. However, not all automata can be represented as regular expressions efficiently (e.g., Hamming/Levenshtein automata). In contrast, our approach improves matrix operations for efficient GPU AP.

High-performance Sparse Linear Algebra. Various methods have been explored to accelerate SpMV, such as workload balancing [14] and machine learning [71] for kernel selection and format generation. Niu et al. [46] proposed a 2D block technique using multiple sparse formats on GPUs, also applied in SpMSpV [36]. Many recent works leverage sparse linear algebra to accelerate graph problems, with GraphBLAS [25] and GraphBLAST [73] being notable examples. Other general graph computing libraries like CombBLAS [15, 19] and GraphPad [12] also include optimizations for SpMSpV. Li et al. [40] dynamically select between SpMV and SpMSpV based on vector sparsity in large-scale graph computations. While these works provide insights and could serve as starting

points like how cuSPARSE-SpMV is used in this work, the challenges specific to AP using BLAS operations are addressed uniquely in our study.

Domain-specific Accelerators for Sparse Matrices. Many domain-specific accelerators [30, 31, 72, 76] targeting the enhancement of SpMV and SpMSpV performance have emerged, addressing their inherently irregular computational demands. For example, EIE [33] enhances sparse neural network inference; SIGMA [50] accelerates sparse neural network training; Sadi *et al.* [53] optimize graph SpMV workloads. Tensaurus [59] speeds up sparse-dense tensor multiplications including SpMV, with the co-design of a new sparse storage format and hardware architecture. Cerberus [35] explores SpMV acceleration design space for diverse input workloads, and OuterSPACE [48] and GaaS-X [22] support SpMV kernels. Spada [41] presents a SpGEMM accelerator equipped with adaptive dataflow, aiming to enhance the performance of sparse matrix workloads. Despite the achieved advancements, none of the aforementioned work caters to accelerating NFA processing tasks. Our method leverages BLAS operations for AP, paving the way for DSAs to support automata workloads.

6 Conclusion

We introduce AutomataBLAS, a high-performance, memory-efficient GPU AP engine that employs a BLAS-based approach. By systematically evaluating the feasibility of using existing GPU-based standard BLAS libraries, identifying key bottlenecks, and proposing automata-specific optimizations, AutomataBLAS achieves up to **6.54**× (geometric mean: **1.78**×) higher throughput than existing GPU solutions (GPU-NFA) and reduces memory usage to as low as **0.006%** (geometric mean: **0.08%**) compared to *ng*AP. Additionally, we achieve a significant speedup over a state-of-the-art BLAS accelerator for automata workloads. This work reveals the potential of BLAS methods for efficient automata execution and offers insights for designing future AP engines with BLAS support.

Acknowledgments

We thank the anonymous reviewers for their detailed feedback, which significantly improved the quality of the paper.

References

[1] 2025. Regular Expression. Retrieved from https://en.wikipedia.org/wiki/Regular_expression
[2] 2018. ANML Documentation. Retrieved from http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html
[3] 2018. Clamav Net. Retrieved from https://www.clamav.net/
[4] 2019. The Zeek Network Security Monitor. Retrieved from https://www.zeek.org
[5] 2019. YARA: The Pattern Matching Swiss Knife for Malware Researchers. Retrieved from https://virustotal.github.io/yara/
[6] 2021. NVIDIA AMPERE GA102 GPU ARCHITECTURE. Retrieved from https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf
[7] 2023. NVIDIA ADA GPU ARCHITECTURE. Retrieved from https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf
[8] 2025. Cooperative primitives for CUDA C++. Retrieved from https://nvidia.github.io/cccl/cub/
[9] 2025. CUDA Graphs. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?#cuda-graphs
[10] 2025. cuSPARSE. Retrieved from https://docs.nvidia.com/cuda/cusparse/
[11] 2025. Tuning CUDA Applications for NVIDIA Ampere GPU Architecture. Retrieved from https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html#increased-l2-capacity-and-l2-residency-controls
[12] Michael J. Anderson, Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Theodore L. Willke, and Pradeep Dubey. 2016. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS).*

[13] Kevin Angstadt, Jack Wadden, Westley Weimer, and Kevin Skadron. 2017. *MNRL and MNCaRT: An Open-Source, Multi-Architecture State Machine Research and Execution Ecosystem*. Technical Report CS2017-01. University of Virginia.
[14] Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M. Tsai, and Weichung Wang. 2020. Load-balancing sparse matrix vector product kernels on GPUs. *ACM Transactions on Parallel Computing (TOPC)* 7, 1 (2020), 1–26.
[15] Ariful Azad, Oguz Selvitopi, Md Taufique Hussain, John R. Gilbert, and Aydın Buluç. 2021. Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 33, 4 (2021), 989–1001.
[16] Michela Becchi, Mark Franklin, and Patrick Crowley. 2008. A workload for evaluating deep packet inspection architectures. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*.
[17] Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. Technical Report. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.
[18] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. 2018. Searching for potential gRNA Off-target sites for CRISPR/Cas9 using automata processing across different platforms. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[19] Aydın Buluç and John R. Gilbert. 2011. The combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 469–509.
[20] Pascal Caron and Djelloul Ziadi. 2000. Characterization of glushkov automata. *Theoretical Computer Science* 233, 1-2 (2000), 75–90.
[21] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnt: NFA pattern matching on GPGPU devices. *SIGCOMM Computer Communication Review (CCR)* 40, 5 (2010), 20–26.
[22] Nagadastagiri Challapalle, Sahithi Rampalli, Linghao Song, Nandhini Chandramoorthy, Karthik Swaminathan, John Sampson, Yiran Chen, and Vijaykrishnan Narayanan. 2020. GaaS-X: Graph analytics accelerator supporting sparse data representation using crossbar architectures. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*.
[23] Jou-An Chen, Hsin-Hsuan Sung, Xipeng Shen, Nathan Tallent, Kevin Barker, and Ang Li. 2022. Bit-GraphBLAS: Bit-level optimizations of matrix-centric graph processing on GPU. In *Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.
[24] Cristiana Chitic and Daniela Rosu. 2004. On validation of XML streams using finite state machines. Association for Computing Machinery. DOI: 10.1145/1017074.1017096
[25] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *(TOMS)* 45, 4 (2019), 1–25.
[26] Paul Dlugosch, Dave Brown, Paul Glendenning, Leventhal Leventhal, and Harold Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 25, 12 (2014), 3088–3098.
[27] Zhe Fu and Jun Li. 2019. High speed regular expression matching engine with fast pre-processing. *China Communications* 16, 2 (2019), 177–188.
[28] Tianao Ge, Xiaowen Chu, and Hongyuan Liu. 2025. Interleaved bitstream execution for multi-pattern regex matching on GPUs. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. DOI: 10.1145/3725843.3756052
[29] Tianao Ge, Tong Zhang, and Hongyuan Liu. 2024. ngAP: Non-blocking large-scale automata processing on GPUs. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[30] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures. *ACM SIGMETRICS Performance Evaluation Review* 50, 1 (2022), 33–34.
[31] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. SparseP: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)* 6, 1 (2022), 1–49.
[32] Victor Mikhaylovich Glushkov. 1961. The abstract theory of automata. *Russian Mathematical Surveys* 16, 5 (1961), 1.
[33] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.
[34] Bingsheng He, Qiong Luo, and Byron Choi. 2006. Cache-conscious automata for XML filtering. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 18, 12 (2006), 1629–1644.
[35] Soojin Hwang, Daehyeon Baek, Jongse Park, and Jaehyuk Huh. 2024. Cerberus: Triple mode acceleration of sparse matrix and vector multiplication. *ACM Transactions on Architecture and Code Optimization (TACO)* 21, 2 (2024), 1–24.

[36] Haonan Ji, Huimin Song, Shibo Lu, Zhou Jin, Guangming Tan, and Weifeng Liu. 2022. Tilespmspv: A tiled algorithm for sparse matrix-sparse vector multiplication on GPUs. In *Proceedings of the 51st International Conference on Parallel Processing (ICPP)*.

[37] Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. Software-hardware codesign for efficient in-memory regular pattern matching. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*.

[38] Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2024. HybridSA: GPU acceleration of multi-pattern regex matching using bit parallelism. *Proceedings of the ACM on Programming Languages* OOPSLA 8 (2024), 1699–1728.

[39] Vincent T. Lee, Justin Kotalik, Carlo C. Del Mundo, Armin Alaghi, Luis Ceze, and Mark Oskin. 2017. Similarity search on automata processors. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[40] Min Li, Yulong Ao, and Chao Yang. 2020. Adaptive SpMV/SpMSpV on GPUs for input vectors of varied sparsity. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 32, 7 (2020), 1842–1853.

[41] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating sparse matrix multiplication with adaptive dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[42] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron. 2012. Parabix: Boosting the efficiency of text processing on commodity processors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[43] Hongyuan Liu, Mohamed Ibrahim, Onur Kayiran, Sreepathi Pai, and Adwait Jog. 2018. Architectural support for efficient large-scale automata processing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[44] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. Why GPUs are slow at executing NFAs and how to make them faster. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[45] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2023. Asynchronous automata processing on GPUs. *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* 7, 1 (2023), 1–27.

[46] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on GPUs. In *Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[47] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. 2017. Demystifying automata processing: GPUs, FPGAs or micron's AP? In *Proceedings of the International Conference on Supercomputing (ICS)*.

[48] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[49] Vern Paxson. 1999. Bro: A system for detecting network intruders in real-time. *Computer Networks* 31, 23–24 (1999), 2435–2463.

[50] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[51] Martin Roesch. 1999. Snort - Lightweight intrusion detection for networks. In *Proceedings of the USENIX Conference on System Administration (LISA)*.

[52] Indranil Roy and Srinivas Aluru. 2014. Finding motifs in biological sequences using the micron automata processor. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*.

[53] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SpMV operation for large and highly sparse matrices using scalable multi-way merge parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[54] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. FlexAmata: A universal and efficient adaption of applications to spatial automata processing accelerators. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[55] Elaheh Sadredini, Reza Rahimi, Lenjani Marzieh, Stan Mircea, and Skadron Kevin. 2020. Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[56] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mohsen Imani, and Kevin Skadron. 2021. Sunder: Enabling low-overhead and scalable near-data pattern matching acceleration. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[57] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. A scalable and efficient in-memory interconnect architecture for automata processing. *IEEE Computer Architecture Letters (CAL)* 18, 2 (2019), 87–90.

[58] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. eAP: A scalable and efficient in memory accelerator for automata processing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[59] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[60] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. 2020. Summarizing CPU and GPU Design Trends with Product Data. arXiv:1911.11313. Retrieved from https://arxiv.org/abs/1911.11313

[61] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. *CUTLASS*. Retrieved from https://github.com/NVIDIA/cutlass

[62] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. 2016. Towards machine learning on the automata processor. In *Proceedings of the International Conference on High Performance Computing (HiPC)*.

[63] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2011. Parallelization and characterization of pattern matching using GPUs. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*.

[64] Kien Chi Vu. 2020. *Accelerating Bit-based Finite Automaton on a GPGPU Device*. Master's thesis. Japan Advanced Institute of Science and Technology.

[65] Jack Wadden, Vinh Dang, Nathan Brunelle, Tom Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. 2016. ANMLZoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*.

[66] Jack Wadden and Kevin Skadron. 2016. *VASim: An open virtual automata simulator for automata processing application and architecture research*. Technical Report. Univeristy of Virginia.

[67] Jack Wadden, Tom Tracy II, Elaheh Sadredini, Lingzi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Matthew Wallace, Jeffrey Udall, Mircea Stan, and Kevin Skadron. 2018. AutomataZoo: A modern automata processing benchmark suite. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*.

[68] Ke Wang, Yanjun Qi, Jeffrey J. Fox, Mircea R. Stan, and Kevin Skadron. 2015. Association rule mining with the micron automata processor. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*.

[69] Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2016. Sequential pattern mining with the micron automata processor. In *Proceedings of the ACM International Conference on Computing Frontiers*.

[70] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[71] Guoqing Xiao, Tao Zhou, Yuedan Chen, Yikun Hu, and Kenli Li. 2024. Machine learning-based kernel selector for SpMV optimization in graph analysis. *ACM Transactions on Parallel Computing (TOPC)* 11, 2 (2024), 1–25.

[72] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[73] Carl Yang, Aydın Buluç, and John D. Owens. 2022. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Transactions on Mathematical Software (TOMS)* 48, 1 (2022), 1–51.

[74] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. 2006. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the Symposium on Architecture for Networking and Communications Systems (ANCS)*.

[75] Xiaodong Yu and Michela Becchi. 2013. GPU acceleration of regular expression matching for large datasets: Exploring the implementation space. In *Proceedings of the International Conference on Computing Frontiers (CF)*.

[76] Xiaoyu Zhang, Zerun Li, Rui Liu, Xiaoming Chen, and Yinhe Han. 2024. GAS: General-purpose in-memory-computing accelerator for sparse matrix multiplication. *IEEE Transactions on Computers (TOC)* 73, 6 (2024), 1427–1441.

[77] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps intrusion prevention on a single server. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[78] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

Received 16 June 2025; revised 16 October 2025; accepted 16 October 2025