# Hopps: Leveraging Sparsity to Accelerate Automata Processing

Xingran Du
Massachusetts Institute of Technology
Cambridge, MA, USA
xrdu@csail.mit.edu

Joel S. Emer
Massachusetts Institute of Technology
Cambridge, MA, USA
NVIDIA
Westford, MA, USA
emer@csail.mit.edu

Daniel Sanchez
Massachusetts Institute of Technology
Cambridge, MA, USA
sanchez@csail.mit.edu

## Abstract

Automata processing (AP) is a key kernel in data analytics and scientific computing. AP workloads process a stream of symbols with many automata (FSMs) in parallel, e.g., pattern-matching network traffic against many malicious strings.

The need for high-performance AP has sparked the design of specialized accelerators. But prior AP accelerators are inefficient: AP workloads have substantial sparsity, but accelerators exploit no or limited sparsity. Specifically, each AP workload can be expressed as the concurrent traversal of all automata, which are encoded as graphs. But state-of-the-art accelerators store these graphs *uncompressed*, using bitsets. This allows the use of specialized memory crossbars that provide high parallelism and efficiency when graphs are dense. But many graphs are highly sparse, making crossbar-based accelerators inefficient.

We present Hopps, the first automata processing accelerator that exploits sparse data representations. Hopps combines two types of processing units: one represents data uncompressed, which achieves high throughput but is space-inefficient, while the other uses a compressed-sparse representation, which achieves high space efficiency but lower and more variable throughput. To use Hopps well, we present a novel automata mapping algorithm that maps most work to high-throughput units, while keeping a large fraction of state in space-efficient units. Hopps's hybrid design relaxes several constraints in crossbar-based designs, allowing for more efficient high-throughput units (e.g., by using a large number of smaller crossbars). Thus, by making the uncommon case cheap, Hopps makes the common case even faster.

We evaluate Hopps on AutomataZoo [44] benchmarks. Hopps outperforms prior state-of-the-art accelerators Impala and SpAP by gmean 2.5× and 2.2× when using equal area.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.
*ASPLOS '25, Rotterdam, Netherlands*
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1080-3/2025/03
https://doi.org/10.1145/3676642.3736126

***CCS Concepts:*** • **Computer systems organization → Architectures**.

***Keywords:*** Sparsity; Accelerators; Automata Processing

**ACM Reference Format:**
Xingran Du, Joel S. Emer, and Daniel Sanchez. 2025. Hopps: Leveraging Sparsity to Accelerate Automata Processing. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676642.3736126

## 1 Introduction

Automata processing has wide applications in network intrusion and malware detection [2, 7, 54, 56], bioinformatics [4, 30], data mining [31, 39], and machine learning [19, 34]. An automata processing workload analyzes a stream of input symbols to find matching patterns, and reports the matching patterns. Each pattern is encoded as an automaton (i.e., a finite-state machine), and its state-transition diagram is represented as a graph. Though each automaton graph is small (typically from less than ten to a few thousand nodes), the workload uses many automata (e.g., thousands), which provides ample parallelism: at its core, the workload is the concurrent traversal of many small graphs.

Despite its importance, automata processing is slow on CPUs or GPUs, calling for specialized accelerators. Specifically, the state-of-the-art GPU AP implementation [12] processes inputs at <200MB/s for all but one of the AutomataZoo [44] benchmarks; CPU implementations are ~10× worse. In particular, the GPU implementation of the network intrusion detection workload Snort achieves 115MB/s, short of the 100s of Gb/s bandwidth of modern SmartNICs [14]. As a result, commercial systems now include pattern-matching accelerators [1, 20, 28].

Prior work has proposed automata-processing accelerators that achieve high performance. The fastest AP accelerators use memory-based *crossbars*, using DRAM [10] or SRAM [41] technology. These crossbars store the adjacency matrices of each automaton using bitsets (with one bit per memory cell), which enable finding the next set of active nodes from the current set in a single, parallel access.

The problem is that these prior accelerators use an *inefficient data representation*: automata graphs are *sparse*, meaning that a significant fraction of entries in the adjacency matrix are zeros. Sparsity in the adjacency matrix arises due to the structure of patterns (e.g., they often consist of a chain of subpatterns), so each node has only one to two neighbors on average. Crossbar-based accelerators either do not exploit this sparsity, or only exploit coarse-grained, structural sparsity, e.g., by packing the adjacency matrix into blocks stored in multiple crossbars. Since crossbars store data using *uncompressed* bitsets, they become inefficient as sparsity grows: they achieve high efficiency only when graphs are dense, but become inefficient when the vast majority of entries in the adjacency matrix are zeros.

In this paper, we propose an AP accelerator that exploits sparse data representations. Prior work has proposed specialized accelerators that exploit sparsity for other domains, in linear algebra [16, 24], graph analytics [8, 26], and deep learning [13, 52, 53]. The key feature of these accelerators is the use of *compressed* data representations that store only nonzero values, reducing memory footprint and avoiding ineffectual computation. Prior GPU implementations have used compressed representations in software [12, 22, 23], but with very different optimization goals (discussed in Sec. 8). *Our core contribution is to bring the machinery of sparse accelerators to bear in the AP domain, leveraging compressed data representations for the first time.*

The key benefit of using compressed data representations is space efficiency: storing the adjacency matrix *compressed* (e.g., in Compressed Sparse Row format [33]) requires space linear to the number of nonzeros, rather than quadratic to the number of automaton states. For example, an automaton graph with 200 nodes where each node has two neighbors on average takes 200×200 bits uncompressed, but requires 400 entries compressed—100× fewer elements. Higher space efficiency means that, for a given amount of area, the accelerator can support more concurrent patterns. When workloads have more automata than can fit in the accelerator, automata are processed in *batches*, as shown in Fig. 1: the accelerator takes multiple passes over the input symbol stream, processing one batch of automata per pass. Thus, using more space-efficient representations means that the accelerator needs fewer batches, which translates to higher performance.

The drawback of using compressed representations is that they are costlier to process: they require a variable number of cycles to traverse and preclude the use of crossbars. Simply using a compressed representation for the whole AP workload does not improve performance, as the slower processing speed negates the benefits of higher space efficiency. Our key insight is that we can use a *hybrid* design that reaps the benefits of sparsity while avoiding a high sparsity tax.

We present *Hopps*, the first automata processing accelerator that extensively exploits sparsity to achieve high performance per area. Hopps combines hardware units that use uncompressed and compressed formats, respectively, to achieve both low traversal latency and high storage efficiency. Because in many workloads, a large fraction of nodes are never accessed, and the computation is concentrated on a few frequently accessed nodes [21], this hybrid approach makes the uncommon case cheap, and the common case fast.

Specifically, Hopps combines two types of units, called Tortoise and Hare units. Tortoise units aggressively compress the automata graph, reducing area overhead, while incurring longer latency for processing compressed data. Hare units, on the other hand, store the frequently accessed nodes using an uncompressed representation and process them at high throughput. This approach yields benefits beyond simply getting the best of each type of unit: because infrequent nodes are offloaded to Tortoise units, Hare units *become more efficient* than those of prior work, as they can be sized more aggressively (e.g., using smaller crossbars).

Our execution model effectively combines both types of units to keep throughput close to running all nodes in the Hare units. The Hare units speculatively run ahead, and only roll back their execution on the rare case where the Tortoise units encounter frequent nodes. To this end, we develop a novel partitioning algorithm that identifies the frequent and infrequent nodes in the automata, based on the topology of automata graphs, each node's likelihood of matching with input symbols, and a small profiling input. We also propose a compression scheme for the match set to reduce its storage, achieving higher area efficiency.

We evaluate Hopps on the AutomataZoo [44] benchmark suite. Hopps achieves 2.5× and 2.2× better gmean performance per area than two prior accelerators, Impala and SpAP.

In summary, we make the following contributions:

- We frame automata processing as sparse computation, and analyze its optimization opportunities.
- We design the first automata processing accelerator that extensively exploits sparsity in automata graphs.
- We propose a novel approach that combines uncompressed and compressed representations to achieve both high throughput and high space efficiency.
- We evaluate these techniques in depth, showing their performance gains.

## 2 Background

### 2.1 Automata computation

Fig. 2 shows an example where a single automaton (specifically, an NFA in homogeneous representation [6]) processes an input stream. The automaton's state-transition diagram is a directed graph; Fig. 2 (top right) shows the graph's *adjacency matrix*. Graph *nodes* are labeled by numbers, and each edge (e.g., node 1 to node 2) corresponds to a 1 in the adjacency matrix (e.g., row 1, column 2). Each node also has a set of symbols it *matches*, indicated by the 1s in the *match set* and the letters on the nodes.

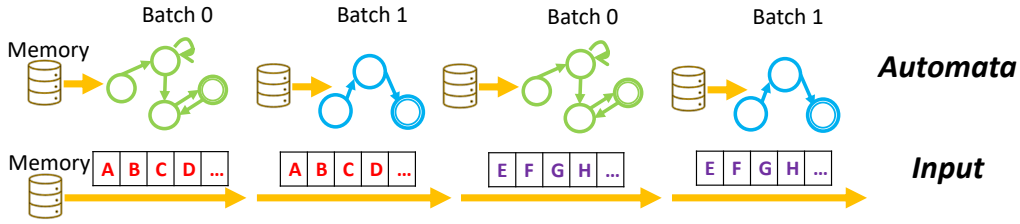
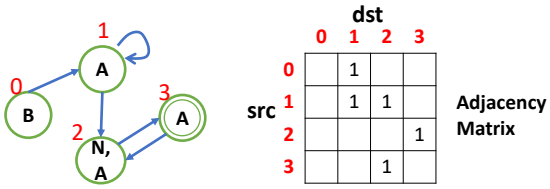



**Figure 1.** Execution timeline when using batching, which is used when automata do not fit in the accelerator. The accelerator takes multiple passes through the input symbols, processing a batch of automata in each pass.




| Input | Frontier | Matched |
|---|---|---|
| B | 0 | 0 |
| A | 1 | 1 |
| N | 1,2 | 2 |
| A | 3 | 3 |

**Figure 2.** Example automaton.

```
1  # Tensors of 1-bit (Boolean) values
2  bool A[C][N][N]; # Adjacency Matrix
3  bool M[C][S][N]; # Match Set
4  bool F[C][N]; # Frontier
5  bool F_next[C][N]; # Next Frontier
6  # Tensor of log(S)-bit values (typically bytes)
7  byte I[L]; # Input Symbols
8
9  for l in [0, L):
10   for cc in [0, C):
11     for src in [0, N):
12       for dst in [0, N):
13         symb = I[l]
14         matched = F[cc][src] & M[cc][symb][src]
15         F_next[cc][dst] = matched & A[cc][src][dst]
16     F[cc] = F_next[cc]
```

**Listing 1.** Automata processing loop nest.

The algorithm maintains a *frontier* of currently active nodes (initialized to the start node, 0). Then, for each input symbol, nodes in the frontier that match with the symbol activate their out-neighbors. In the example, node 0 in the frontier matches with the first input symbol B, so it activates its neighbor node 1, which is now the only node in the next frontier. Then, for input A, node 1 matches, and activates nodes 1 and 2, which become the next frontier. For the next input N, both nodes 1 and 2 compare their match sets with N, and only node 2 matches, so it activates its neighbor node 3. Node 3 matches with the next input A, and it is a *report* node (indicated by the double circle), so the match is reported.

Listing 1 shows the loop nest for an automata workload, which processes a vector of input symbols `I` for all $C$ automata in the workload. Each automaton has at most $N$ nodes, and is specified by its $N \times N$ adjacency matrix `A`, and $S \times N$ match set `M`, where $S$ is the size of the set of valid symbols. For each input symbol, and for each automaton, the workload finds whether each source node matches with the input symbol (`matched`), then traverses the adjacency matrix `A` to add each matching node's neighbors to the next frontier `F_next`.

Since processing each symbol depends on the previous ones, the latency of processing a single symbol (one iteration on line 9) is key to performance. Typically, the input `I` is streamed in from the host, while automata tensors `A` and `M` are stored on-chip. Storing many automata on-chip and processing them in parallel improves performance; when automata do not fit on-chip, batching is used, as shown in Fig. 1 (batching is equivalent to tiling the loops in lines 9–10).

### 2.2 Opportunities to exploit sparsity

Exploiting sparsity for the adjacency matrix `A` can reduce its storage overhead and improve performance. Fortunately, there is abundant sparsity in the automaton computation.

***The frontier is sparse.*** At any given time, only a tiny fraction of nodes across all automata are in the frontier (i.e., active). For example, the Snort benchmark from AutomataZoo [44] has 202043 total nodes, but the frontier size averaged across the input trace is around 230 nodes, which is only 0.1%. In addition, some nodes are far more likely to be in the frontier and capture most of the computation; we call them *frequent* nodes, and the rest *infrequent* nodes.

***The adjacency matrix is sparse.*** The average degree of automaton graphs is around 1–2, meaning that all but 1–2 entries per row in the adjacency matrix are zeros. With the same average degree, the sparsity (fraction of zeros) of the adjacency matrix increases with its size. For a 256-node graph, the adjacency matrix is around 99% sparse; prior work often use $256 \times 256$ crossbars [35, 37, 41].

Despite abundant sparsity in automata processing, prior accelerators use *uncompressed* representations for tensors, leading to storage inefficiencies (Sec. 8 discusses sparsity in software implementations). In an *uncompressed* representation for the adjacency matrix, i.e., a 2D array, the storage needed is fixed (quadratic to the number of nodes) which is undesirable when sparsity is high and storage is only needed for nonzeros (roughly linear to the number of nodes).

| | CA [41] | eAP [38] | Impala [37] | SpAP [21] | Hopps: Hare | Hopps: Tortoise |
|---|---|---|---|---|---|---|
| Frontier representation | bit vector | bit vector | bit vector | bit vector | bit vector | node list |
| Crossbar size | 256×256 | 96×96 | 256×256 | 232 × 16 + 24 × 2304 | 32×32 | 64 src nodes |
| Can one automaton span multiple crossbars? | ✗ | ✗ | ✓ | ✗ | ✓ | |
| Adjacency matrix compressed? | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Node activity-aware? | ✗ | ✗ | ✗ | ✓ | ✓ | |

**Table 1.** Prior specialized AP accelerators do not use compressed representations to exploit sparsity.

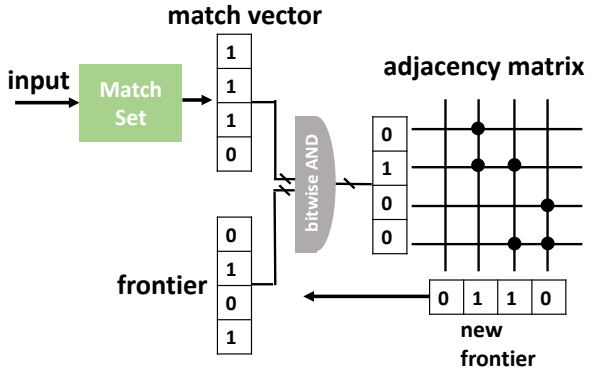



**Figure 3.** Example showing the operation of a memory-based crossbar (using the same matrix as Fig. 2).

### 2.3 Prior crossbar-based accelerators

Prior AP accelerators store adjacency matrices in memory crossbars implemented using DRAM (e.g., AP chip [10]) or SRAM (e.g., Cache Automaton [41]). Crossbars allow finding all neighbors of the frontier nodes in parallel in one cycle.

Fig. 3 shows the operation of a crossbar. The crossbar is implemented using a memory array, e.g., an SRAM with 8-transistor cells, as proposed in Cache Automaton [41]. Each horizontal line (wordline) corresponds to a source node, and each vertical line (bitline) corresponds to a destination node. Each cell in the array stores a bit of the adjacency matrix; a value of 1 denotes an edge from the source to the destination node. Wordlines are driven with all the (matched) source nodes, and cells storing a 1 pull up their bitline, so the next frontier is produced at the bitlines in a single operation.

This design achieves high parallelism, but leaves compression opportunities unexploited. For example, Cache Automaton uses 256 × 256 crossbars, leading to a high fraction of zeros. State-of-the-art accelerators adopt this design with additional optimizations, e.g., to pack several automata in the crossbar; nonetheless, this still leaves most bits as zeroes.

Table 1 summarizes relevant prior work. Cache Automaton (CA) [41] uses 256 × 256 bits SRAM arrays to store adjacency matrices, while eAP [38] folds the 256 × 256 adjacency matrix to fit in a 96 × 96 crossbar with added circuitry.

Impala [37] uses 256 × 256 arrays, while also supporting larger (1024 × 1024) adjacency matrices by composing 256 × 256 crossbars diagonally. Impala and eAP can exploit coarse-grain, structural sparsity by using the fact that ones in adjacency matrices are often clustered near the diagonal; however, they fail to exploit fine-grained sparsity, which Hopps's Tortoise units do exploit.

SpAP [21] is an execution mode designed for the AP chip [10], which uses DRAM crossbars and limits the number of neighbors for most nodes to 16, but allows some nodes to have high connectivity (up to 2304 nodes). SpAP increases throughput by processing frequent and infrequent nodes in two phases, skipping symbols where no node is active when processing infrequent nodes. To achieve this, SpAP needs to store timestamps of frequent nodes activating infrequent nodes, and it *cannot* classify nodes as infrequent if they ever need to activate frequent nodes again. SpAP's solution is conservative. In contrast, Hopps supports more fine-grained synchronization between frequent and infrequent nodes, in addition to supporting the SpAP execution model, allowing additional optimization opportunities.

All these architectures store their adjacency matrices uncompressed in crossbars. In contrast, our Hare units use small 32 × 32 crossbars to capture the diagonal pattern of ones in the adjacency matrix, efficiently achieving high access throughput, while our Tortoise units use a compressed format that exploits fine-grained sparsity and achieves linear instead of quadratic storage scaling.

We evaluate our system against Impala and SpAP (Sec. 7).

## 3 Hopps Architecture

We design an architecture that effectively exploits sparsity to deliver high performance, with the philosophy of making the common case fast, and making the uncommon case cheap. Fig. 4 shows an overview of Hopps. Hopps is a spatial architecture with two types of execution units: Hare (H) and Tortoise (T) units. Hare units process frequently accessed nodes at high thoughput (1 symbol/cycle), while Tortoise units store infrequently accessed nodes compressed, and process them at lower throughput. Tortoise units proceed asynchronously, and let Hare units run ahead, but interrupt them rarely, when synchronizing execution is needed.

Hardware units are logically laid out in columns, with local connections among them. Rows of units can be added as needed to take the desired amount of chip area. A configurable local interconnect within and between H units and T units supports composing them to run the same automaton.

Input symbols are streamed from the host system and broadcast across units, consumed by match set (M) storage to produce the match vectors which are inputs to H and T units. Each Tortoise unit implements a small input buffer to

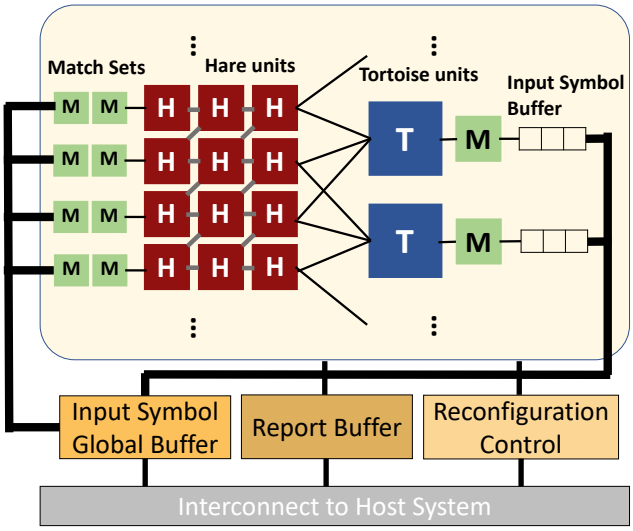



**Figure 4.** Hopps system diagram: Hare units (H), Tortoise units (C), and match sets (M) have local connections; units are laid out in columns and replicated in the vertical dimension.

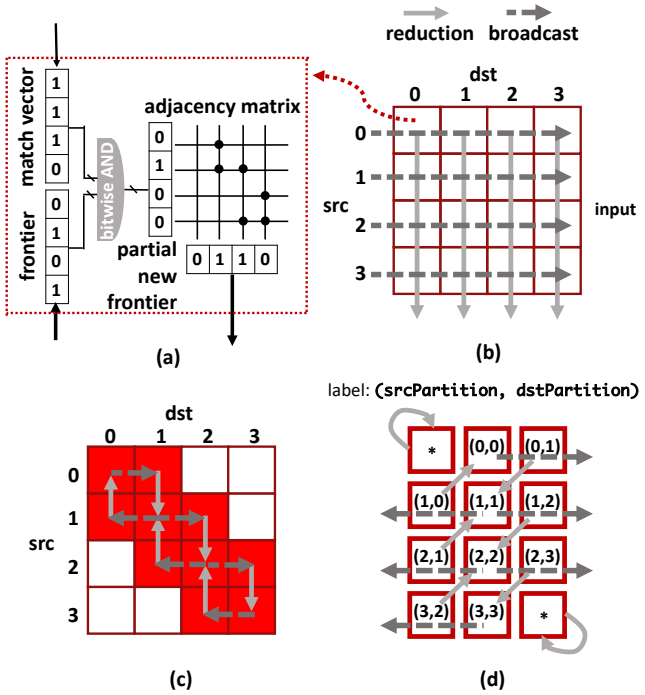



**Figure 5.** Hare unit: (a) single Hare unit; (b) communication pattern when processing a partitioned adjacency matrix; (c) communication pattern if only near-diagonal blocks of the matrix are stored; (d) layout and communication between Hare units processing one partitioned adjacency matrix.

accomodate throughput variation across units. Hopps also implements a small global report buffer to stream reports to the host system as they are generated. On each reconfiguration, SRAM data bits and interconnect configuration bits are streamed from the host. Sec. 5 details I/O and integration with the host system; in this section, we focus on the design and data representation used in Hare and Tortoise units.

### 3.1 Hare unit

Sec. 2 described how some prior work uses SRAM-based crossbars to store adjacency matrices uncompressed and traverse them with high parallelism. Hare units use the same approach (with a smaller SRAM size) *within* each unit. In addition, Hopps employs a novel technique to efficiently compose multiple Hare units to process larger automata.

**3.1.1 Single Hare unit.** Fig. 5(a) shows the hardware within each Hare unit. For each symbol, the match vector (bitmask indicating which nodes match with the current input symbol) is read. When the unit is used alone, the SRAM output directly becomes the new frontier. Otherwise, the output is sent across units to compute the new frontier, which is then received by the unit to continue with the next iteration. Each iteration can be performed in one cycle.

**3.1.2 Composing multiple Hare units for one automaton.** We use a 32×32 crossbar in each unit, as many automata only have a small number of frequently activated nodes. When an automaton has more than 32 frequent nodes, we can compose multiple Hare units to process it.

Fig. 5(b) illustrates how a $128 \times 128$ adjacency matrix can be split into $4 \times 4$ blocks of $32 \times 32$ nodes. Nodes are split into 4 partitions, and each block stores the connectivity between a pair of source and destination partition. When each block is assigned to a processing unit, the unit sends its result bit vector vertically (across blocks with the same destination partition) and reduced using bitwise OR to produce the new frontier for that partition. Then, the new frontier is broadcast horizontally (across blocks with the same source partition) to continue the next iteration.

Unfortunately, this design is inefficient when the adjacency matrix is sparse. By carefully ordering the nodes in each automaton (using standard fill-reducing reordering algorithms [9]) the 1s in the adjacency matrix can often be clustered near the diagonal. Therefore, units far from the diagonal would store only zeros and be completely ineffectual.

Our approach avoids this by storing a narrow band of blocks near the diagonal. In Fig. 5(c), the colored blocks represent the region we store, while the white blocks do not need to be stored. Now, only adjacent hardware units need to communicate if reduction and broadcast happens at the blocks on the diagonal. For most automata, the diagonal band is no wider than 3 blocks.

Fig. 5(d) illustrates how Hare units operate when laid out as a grid with a width of 3, and a height of the number of partitions in the original adjacency matrix (4 in the example). The label in each unit represents the ID of its source and destination partition. In each iteration, the partitioned frontier is broadcast among all units in the same row, and

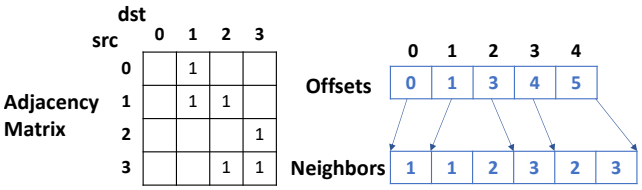



**Figure 6.** Adjacency matrix in CSR format.

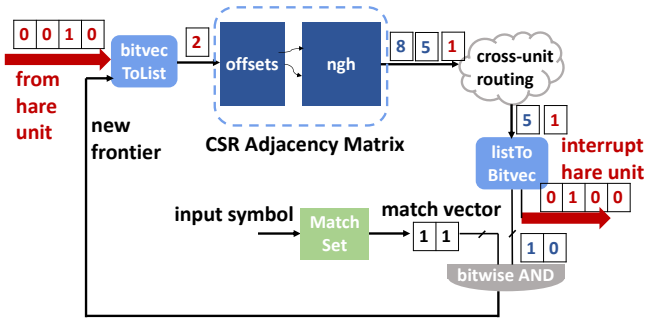



**Figure 7.** Tortoise unit.

each unit independently computes the result bit vector. Then, units with the same destination partition (diagonal) perform a reduction (bitwise OR) to form the next frontier for that partition. The unused units ("∗") can be used independently, e.g., for automata with less than 32 frequent nodes.

The broadcast and reduction are both simple operations among adjacent units, and the communication distance does not grow with the size of the adjacency matrix (number of rows in (d)), only with the diagonal band size (number of columns in (d)), which is determined by the adjacency pattern and is usually small (3). Since connections are local, it is easy to achieve a throughput of 1 symbol/cycle.

## 3.2 Tortoise unit

**3.2.1 Compressed adjacency matrix.** Using a compressed format for the adjacency matrix improves space efficiency. But efficiently traversing compressed formats requires hardware support, and even then, it is slower than a crossbar.

The Tortoise unit stores the adjacency in Compressed Sparse Row (CSR) format, shown in Fig. 6. The `Neighbors` array contiguously stores the lists of neighbor nodes for each source node, and the `Offsets` array records the starting position of each neighbors list. To retrieve the neighbors of node `src`, we first access the `Offsets` array to read indices `Offsets[src]` and `Offsets[src+1]`. Then, we fetch the range of `Neighbors` elements demarcated by these indices.

In the automaton computation, for each input symbol, the adjacency matrix needs to be traversed for all matching source nodes, which cannot be done in one cycle due to these data-dependent accesses. The Tortoise unit achieves a high compression ratio, but has lower throughput, so it is suitable for processing the long tail of infrequently activated nodes.

**3.2.2 Hardware.** Fig. 7 shows the structure of each Tortoise unit. Tortoise units are composable, like Hare units: each unit uses small memories to store the adjacency matrix, and if an automaton requires more entries, its adjacency matrix is partitioned by source node ID and processed by multiple Tortoise units in parallel.

Computation begins when a Hare unit finds one or more infrequent nodes that are mapped to a Tortoise unit. Hare units send these nodes to the Tortoise unit, encoded as a bit vector. Then, the Tortoise unit proceeds asynchronously: the pipeline in Fig. 7 first reads this source bitvector and reads the CSR adjacency matrix, producing a neighbor ID per cycle. The list of nodes are then converted to a bitvector at the throughput of one node ID per cycle, yielding the next frontier. This bitvector is ANDed with the match set to produce the next frontier; if it is not empty, it is used to start the next iteration.

The Tortoise unit communicates with other units at two points. First, when multiple Tortoise units are used to process the same automaton, each Tortoise unit holds all the neighbor IDs for a slice of sources, so neighbors must be routed to other Tortoise units based on the partition they belong to (shown as *cross-unit routing* in Fig. 7). Second, the Tortoise unit may find a *frequent* node in its next frontier, i.e., a node that is mapped to a Hare unit. The Tortoise unit interrupts the Hare unit in this case, as we will see in detail later. Thankfully, our partitioning algorithm makes these interrupts rare: traversals that reach Tortoise units often die off in Tortoise units (Sec. 3.4). As a result, even though Tortoise units are slower than Hare ones, Hopps can keep throughput close to that of the Hare units.

## 3.3 Match set compression

As discussed in Sec. 2.1, automata processing requires storing the adjacency matrix and match set for each automaton. In crossbar-based accelerators, the adjacency matrix takes up more than half of the storage. However, after optimizing the representation of the adjacency matrix, the match set now dominates, limiting overall compression ratio. Thus, we also use a compressed representation for the match set.

Prior work [35–37] has also identified this overhead, and has proposed transforming automata to use smaller symbols (e.g., 4-bit instead of 8-bit symbols). Since each match set has one entry per symbol, this yields much smaller match sets. However, this requires processing multiple symbols per cycle to make up for the smaller symbols and extra nodes in the new automata. This approach is orthogonal to our work; here, we adopt a simple compression technique that sufficiently reduces match set size and demonstrates the full advantage of our main contributions.

The green array in Fig. 8 (left) shows the uncompressed match set. Each cycle, the input symbol is used as an index to read a match vector, which is a bit vector corresponding to whether each node matches with the indexing symbol.

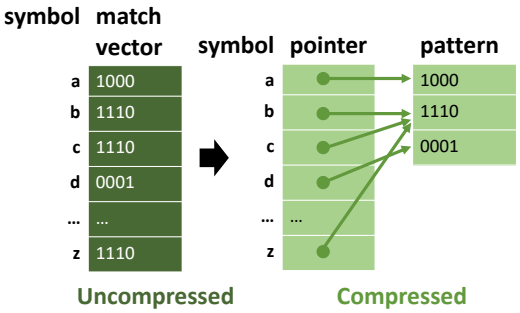



**Figure 8.** Match sets are compressed by deduplicating unique match vectors.

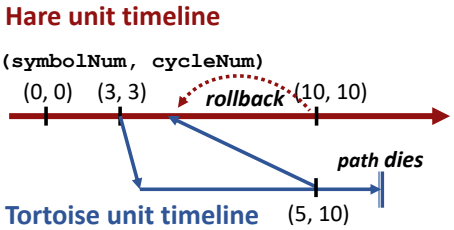



**Figure 9.** Synchronization between Hare and Tortoise units. Most of the execution paths offloaded to Tortoise units die off in Tortoise units.

The length of the match vector is typically the size of the adjacency matrix block or the frontier, since AND operations need to be performed among the bit vectors.

Unlike the adjacency matrix, the match set has limited sparsity (1s are frequent), so a CSR representation is ineffective. Fortunately, we find through profiling that multiple symbols often share the same match vector. For example, some symbols (e.g., non-alphanumeric characters) are never used, and some symbols (e.g., white space) are matched by the same nodes. Therefore, we deduplicate these patterns to save space using a two-level structure, shown in Fig. 8 (right): the *pattern* memory stores these unique bit vectors, and the *pointer* memory has one entry per symbol and stores a pointer to the right pattern bitvector. Our experiments show this approach leads to over 2× storage reduction.

Our implementation uses separate, fixed-size SRAMs for each of these memories. We size them for the common case; when an automaton would need larger memories (because it has more unique patterns than supported in one pattern SRAM), this is treated as a mapping restriction: source nodes are partitioned across more processing units to ensure that each match set has a fitting number of unique patterns.

Reading the compressed match set takes two (pipelined) SRAM accesses, but since the match set accesses only depend on the input symbol, we can prefetch the symbol stream and pipeline the accesses, in parallel with accessing the adjacency matrix. Therefore, the latency and throughput described in Sec. 3.2 and Sec. 3.1 still hold.

### 3.4 Execution

We develop an execution model to efficiently combine the advantages of both types of units, exploiting the fact that Hare units offloading the computation of infrequent nodes to Tortoise units is rare, and synchronizations due to Tortoise units encountering frequent nodes are even rarer.

Fig. 9 shows the execution timeline of a Hare unit and a Tortoise unit working together. The Hare unit runs at 1 symbol/cycle, and at cycle 3 (symbol 3), its frontier has a frequent node with an infrequent neighbor. The Tortoise unit is then activated, and starts reading infrequent neighbors from the CSR matrix. It processes symbols in parallel with the Hare unit, albeit at a lower speed. Next, at cycle 10, the Tortoise unit is now processing the 5th symbol, and it finds frequent nodes in its new frontier, interrupting the Hare unit. On receiving a frontier and a `symbolNum` timestamp from the Tortoise unit at cycle 10, the Hare unit saves its current frontier at symbol 10, and restarts execution with the received frontier at symbol 5. When it reaches symbol 10 again, it merges (ORs) the current frontier with the saved one, and resumes processing subsequent symbols (symbol 11). After sending the frontier at cycle 10, the Tortoise unit continues execution until the frontier is empty, i.e., the path dies. If another frequent-infrequent transition happens before the Tortoise unit is available, the Hare unit stalls.

The overall latency for an input stream is the longer of the two timelines. Frequent-to-infrequent transitions are rare, and only a fraction of them transition back to frequent. Therefore, Hare unit stalls and rollbacks are rare, and throughput is comparable to running all nodes using Hare units.

Hopps hardware has the following features to support this execution model. We detect the presence of an infrequent neighbor by keeping a bitmask of the frontier nodes. As shown in Fig. 10, we apply the bitmask to the matched frontier, and send it to the Tortoise unit if it is not all zeroes. We also relabel node IDs at compile time such that all frequent nodes have lower IDs than infrequent nodes, so in the Tortoise unit, we can detect frequent nodes by comparing with a threshold value. Fig. 7 shows how hardware creates a separate frontier with frequent node 1, and sends it to the Hare unit.

### 3.5 Interconnect

As shown in Fig. 4, Hare and Tortoise units and match set storage are placed logically as a long column. Hopps only requires configurable local connections between the units.

Hopps can be designed with different provisioning of hardware units (explored in Sec. 7.3) and spatial layouts. Sec. 6 details the provisioning and spatial layout used for our main results. Our examples in this section (Fig. 4 and Fig. 5) show how Hare units and their match sets can be laid out when their ratio is 3:2 (the ratio in our evaluated configuration is 7:3).

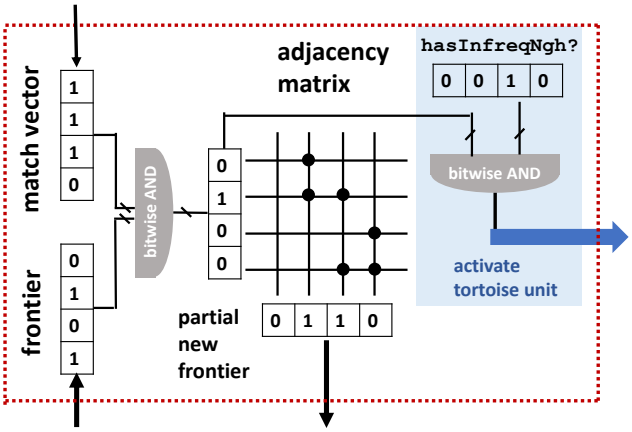



**Figure 10.** Extra hardware in Hare units for activating Tortoise units.

In Fig. 4, each Tortoise unit is connected to a match set, and each row of Hare units is connected to two different match sets, which can be used by units in the row. This is because Hare units in the same row may be composed to process the same automaton, in which case they need the same match set, or the units may operate independently, so that they need different match sets. In Fig. 5(d)'s example, the second and third rows use one match set per row, while the first and fourth rows use two match sets per row because the "$*$" units are used for different automata.

Hare units are connected only to local Tortoise units, because we map infrequent nodes to Tortoise units by maintaining the same order of automata as in Hare units. If the storage requirement for frequent and infrequent nodes mismatches with the provisioning ratio between Hare and Tortoise units, some Hare or Tortoise units can be left underutilized to maintain a short distance between frequent and infrequent nodes in the same automaton.

## 4 Compiling Automata to Hopps

We develop a partitioning heuristic to determine whether nodes should be mapped to Hare or Tortoise units. We further describe how automata adjacency matrices are mapped to units to achieve high utilization.

### 4.1 Frequent/infrequent node partitioning

As we have seen in Sec. 3.4, making sure there is low activity in the infrequent paths is important for performance, but we also want to map as many nodes as possible to the Tortoise units to achieve better compression ratio. Prior work SpAP [21] developed a heuristic for predicting low-activity nodes using a profiling input of 1% of the test trace length. Inspired by their approach, we use a 1% profiling trace to collect a set of enabled nodes, and we compute the topological depth of each node in the automaton graph. We further observe that many workloads have nodes with a self-loop that match with all possible symbols except a few special ones (e.g. \n, \r), and when misclassified as infrequent nodes, they will form long cold paths that are unlikely to end if they are ever enabled. Therefore, we assign **high** priority to nodes that are enabled in the profiling trace, **medium** priority to nodes that have high match set occupancy ($\geq$ 253 out of 256 possible symbols) and their neighbors, and **low** priority to the rest. Within each priority class, nodes are sorted by increasing topological depth.

Then, for each workload, we start by marking all nodes infrequent, and calculate the number of batches needed for mapping all automata. The algorithm then greedily moves nodes from the infrequent set to the frequent set as long as the number of batches decreases or stays the same, to maximally utilize the Hare units. If a workload has too high a fraction of enabled nodes ($\geq$ 80%), or if all nodes are either enabled or have high match set occupancy, we classify all nodes as frequent to avoid the performance penalty caused by potentially frequent synchronizations between Hare and Tortoise units.

In addition, because Hopps supports a wider range of node partitioning and execution schemes than prior accelerators, we can also combine SpAP's optimization with our execution model, further improving performance. Specifically, nodes that are very unlikely to be activated (*doomed* nodes) can be mapped in separate batches, where symbols are skipped if no node is activated. In each automaton, doomed nodes are required to have higher topological depth (assigned in the same way as in SpAP) than any other nodes, so there is never any transition from doomed nodes back to other nodes. When processing these doomed-node batches, for simplicity, we map all nodes to the Hare units. In each automaton, we mark as doomed nodes all nodes with higher topological depths than a cutoff value defined by the max depth of all high-priority nodes and top 30% of medium-priority nodes. The remaining nodes are partitioned into the frequent and infrequent sets following the heuristics described above.

### 4.2 Mapping automata to hardware units

After defining the frequent and infrequent nodes, we create an adjacency matrix for each automaton, containing only frequent nodes. We map these adjacency matrices to Hare units, and then based on the automaton order in the result mapping, we map the remaining adjacency information to Tortoise units.

To increase utilization in Hare units, we pack different automata in the same unit as shown in Fig. 11; this is a standard technique in prior work, as it allows sharing crossbars among small automata. In Fig. 11, regions within the two black squares denote the adjacency matrices of automata 1

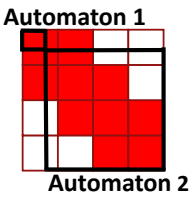



**Figure 11.** Multiple automata can share the same unit.

and 2. Automaton 2's adjacency matrix uses 10 Hare units (red filled squares), but leaves some entries unutilized; this allows automaton 1 to fit in the remaining space.

To map automata to Hare units while reducing underutilization, we use the first-fit-decreasing heuristic for bin packing. Specifically, we sort all automata in a workload by decreasing size. Then, starting from the first automaton, we assign it to a matrix with just enough blocks to store it. For each new automaton, we try to map it to an existing matrix by sharing diagonally, and if that fails, we create a new matrix with just enough blocks.

Prior work Impala [37] uses genetic algorithm for mapping. We found our approach to achieve slightly better results than reported by Impala, so for simplicity, we use our algorithm to map both our system and Impala in our evaluation.

## 5 System Integration

We detail how Hopps communicates with the host system, and how Hopps streams inputs to Hare and Tortoise units.

***Integration with host system:*** In this paper, we evaluate Hopps configurations of modest area ($0.32mm^2$ at 7nm) running at a 2GHz frequency (Hopps has short critical paths and can scale to 5GHz, like Impala, but 1-2GHz is typical in accelerators). At this scale, Hopps analyzes inputs at about 2GB/s, and can be implemented as a discrete accelerator (e.g., a PCIe card). This integration approach is similar to Impala's "peripheral device" integration option [37], and similar to some commercial products like Titan IC's RXP card [51].

Fig. 12 shows the communication between Hopps and the host system; communication is managed through DMAs. Compiling automata to Hopps generates the memory values to be stored in Hopps. For each batch, the host *(1)* streams the batch's configuration to the accelerator, and then *(2)* streams a segment of input symbols. As it processes input symbols, Hopps produces reports and sends them to the host *(3)*. These also include deferred doomed-node activations, which the host processes as a separate batch *(4)*.

Hopps requires modest I/O bandwidth (<10GB/s per direction). Inputs are processed at a maximum rate of 1 symbol/cycle, which corresponds to 2GB/s. We keep an input buffer of 16KB and an output buffer of 2KB (512 4-byte entries of metadata, like Impala), such that the communication can be managed with interrupt service routines at the host at 1MHz. To amortize reconfiguration overheads, we use input segments of 1 million symbols (1MB) in our evaluation.

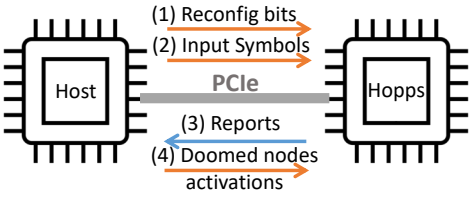



**Figure 12.** System integration.

Hopps has ~2.5MB configuration bits, corresponding to an additional bandwidth of 5GB/s.

Hopps can also be scaled up to achieve higher performance in two ways. First, increasing the number of hardware units within Hopps accommodates more patterns per batch, reducing the number of batches and improving performance; we analyze this type of scaling in Sec. 7.3. Second, multiple Hopps accelerators can be used in a data-parallel fashion, e.g., to analyze multiple packets/streams in parallel, as the Bluefield SmartNIC does [48]. Higher-performance designs would require more bandwidth, and would benefit from tighter integration, e.g., as units within an SoC, similar to RegEx accelerators in SmartNICs [28].

***Input streaming:*** Because Hare and Tortoise units run at different speeds, the symbol they process at the same cycle can diverge. Furthermore, Hare units in different automata can be delayed by different amounts due to Tortoise unit processing. We found that the majority of automata in the same workload process input symbols at near the same speed as the fastest automaton, with a small fraction of outliers deviating by ~5% of the total number of symbols at the end of the trace. Since the latency of processing each batch is determined by the long-pole automaton, faster automata can tolerate occasional stalls. Therefore, we broadcast 10 input symbol streams across the chip, corresponding to the newest symbol (position $x$ at cycle $x$), and symbols at positions $x-32$, $x-64$, $x-128$, $x-256$, ..., and $x-16384$. Hare units always process at one of these input positions, stalling to use an older stream if necessary. Tortoise units keep a buffer of 32 symbols (2% of total chip area) and can fill it with symbols from any of the streams, so that each Tortoise unit can consume symbols at their own rate. Our 16 KB global input buffer retains these symbols and delivers the skewed symbol streams, so the host streams inputs only once.

## 6 Methodology

### 6.1 Benchmarks

We evaluate our system using the AutomataZoo [44] benchmarks. Table 2 shows these workloads and their characteristics. We exclude the APPRNG (Pseudo Random Number Generation) application because it does not perform pattern matching [42], and thus has very different characteristics from others, falling beyond the scope of this work.

### 6.2 Implementation

We implemented our execution model in VASim [43], an open-source automata processing simulator.

To estimate SRAM area and energy, we used PCACTI [40] to model at 7nm all SRAM shapes used in our design and baseline designs, as shown in Table 3. We implemented the logic of the Hare and Tortoise units in RTL and synthesized them at 45nm technology using yosys [49] and the FreePDK45 open cell library [17], then scaled them down to to 7nm.

| Application | abbr. | #nodes | avg degree | #automata | max automaton size | avg automaton size |
|---|---|---|---|---|---|---|
| Brill | Brill | 115549 | 1.37 | 5946 | 40 | 19.4 |
| CRISPR_CasOT | CRP | 202000 | 1.66 | 2000 | 101 | 101.0 |
| ClamAV | CAV | 2374717 | 1 | 33171 | 22075 | 71.6 |
| EntityResolution | ER | 413352 | 1.55 | 10000 | 75 | 41.3 |
| FileCarving | FC | 2663 | 58.8 | 9 | 1183 | 295.9 |
| Hamming_N1000_l18_d3 | HM | 108000 | 1.69 | 1000 | 108 | 108.0 |
| Levenshtein_N1000_l19_d3 | LV | 109000 | 4.08 | 1000 | 109 | 109.0 |
| Protomata | Pro | 24103 | 1 | 1309 | 123 | 18.4 |
| RandomForest_20_400_200 | RF | 248000 | 1 | 8000 | 31 | 31.0 |
| SeqMatch_BIBLE_w6_p6 | SM | 51570 | 2.13 | 1719 | 30 | 30.0 |
| Snort | Snort | 202043 | 1.17 | 2486 | 4509 | 81.3 |
| YARA | YARA | 1047528 | 0.98 | 23530 | 1017 | 44.5 |

**Table 2.** Workload characteristics.

| Usage | Cell Type | Size | Area ($\mu m^2$) | Delay (ps) |
|---|---|---|---|---|
| Hopps: uncmpr. adj. | 8T | 32×32 | 20 | 33 |
| Hopps: CSR adj. | 6T | 64×8 | 7 | 20 |
| | 6T | 128×16 | 23 | 24 |
| Hopps: match set | 6T | 256×8 | 23 | 24 |
| | 6T | 64×32 | 25 | 25 |
| | 6T | 32×32 | 14 | 24 |
| Impala: adjacency | 8T | 256×256 | 943 | 109 |
| Impala: match set | 6T | 256×256 | 659 | 83 |
| SpAP: adjacency | 8T | 232×16 | 55 | 35 |
| | 8T | 24×256 | 282 | 99 |
| SpAP: match set | 6T | 256×256 | 659 | 83 |

**Table 3.** SRAM shapes used in Hopps and baselines.

| | Logic | SRAM - adj. | SRAM - match. | Total |
|---|---|---|---|---|
| Hare units | 0.014 | 0.080 | 0.066 | 0.16 |
| Tortoise units | 0.050 | 0.050 | 0.061 | 0.16 |

**Table 4.** Hopps area breakdown (unit: $mm^2$).

| | Hare | | Tortoise | |
|---|---|---|---|---|
| | Adjacency | Match | Adjacency | Match |
| Units | 4032 | 1793 | 1635 | 853 |
| Area % | 27.5 | 22.5 | 22.5 | 27.5 |

**Table 5.** Provisioning of hardware components in Hopps.

| | Logic | SRAM - adj. | SRAM - match. | Total |
|---|---|---|---|---|
| Hopps | 0.064 | 0.130 | 0.127 | 0.320 |
| Impala-based | 0.009 | 0.183 | 0.128 | 0.320 |
| SpAP-based | 0.005 | 0.251 | 0.064 | 0.320 |

**Table 6.** Area comparison for Hopps and baselines ($mm^2$).

Table 4 shows the area breakdown of Hopps. Hopps uses smaller SRAMs, and based on the SRAM delays in Table 3, we choose to target the same clock frequency for Hopps and for both baselines.

Table 5 shows the provisioning of 4 types of hardware components: the adjacency matrix and match set of Hare and Tortoise units. Hare units are laid out in rows of 21 adjacency matrices and 9 match sets; each row can be split into groups (e.g., grouping of 3 adjacency matrices and 1 match set) to achieve high utilization. Tortoise units are evenly distributed, with local connections to the Hare units. The constraints are more relaxed for the communication latency between Hare and Tortoise units, because Hopps's performance is less sensitive to the latency of Tortoise units. We sweep other ratios in Sec. 7.3.

We compare performance by measuring the latency needed for each system to process a 1 MB input trace for each workload. To calculate the latency for Hopps, we run our mapping algorithm for each workload to determine the number of batches needed to map all automata, and we obtain the latency needed for each batch using our VASim-based simulator.

### 6.3 Baselines

We compare Hopps with Impala-based and SpAP-based systems. We say these systems are *based* on Impala and SpAP because we don't evaluate them exactly as described in prior papers: instead, we implement all in the same technology (and frequency), and scale them to achieve iso-area configurations, so that performance differences do not stem from these factors (e.g., SRAM- vs DRAM-based crossbars).

***Impala-based:*** We compare our system with a canonical in-SRAM automata processor, adopting Impala's crossbar design [37]. We use 8-bit symbols for our system and both baselines, so we store $2^8$ bits per node for the match set in our baselines. We compose G4 units diagonally for automata larger than the capacity of the G4 (1024 nodes).

***SpAP-based:*** We further compare with SpAP [21], an execution model on the AP chip [10], which improves throughput by mapping infrequent nodes in separate batches and skipping cycles with no node active. We use a 10KB profiling input (1% of test trace length) for our system and SpAP. We abstract the AP chip's complex routing (node connection) pattern as a set of fixed-size SRAMs for storing the adjacency matrix. Specifically, for every 256 nodes, there are 24 nodes connected to maximum 2304 neighbors (9 blocks of 256), so we model them as nine $24 \times 256$ crossbars. The other 232 nodes have at most 16 neighbors, so we model them as a $232 \times 16$ crossbar. We optimistically assume perfect mapping like in SpAP, i.e., any 24K nodes in the workload can be mapped to fill the chip regardless of their adjacency patterns.

We target a chip area of $0.32mm^2$ for our system and both baselines, which corresponds to mapping 24K nodes on the AP chip, a capacity evaluated in the SpAP work. Table 6 lists the area breakdowns of these systems.

# 7 Evaluation

## 7.1 Performance

Fig. 13 shows the speedup of Hopps over the Impala-based and SpAP-based baselines. Performance is measured by input symbols processed per cycle, and all three systems have the same area. Each group of bars shows the speedup of the three evaluated systems normalized to Impala-based on a single workload. Workloads are sorted by ascending storage requirement along the $x$-axis; the rightmost group of bars reports the gmean speedups across workloads. Table 7 further shows the throughput (symbols/cycle) and number of batches needed for each workload on each system. SpAP-based and Hopps both take additional cycles to process the doomed nodes (Sec. 4.1); we report those additional cycles as a fractional number of additional batches ("+$x$") for better intuition.

Overall, Hopps is gmean 2.5× faster than Impala-based, and gmean 2.2× faster than SpAP-based. Hopps outperforms or matches Impala-based and SpAP-based in all evaluated workloads; in particular, Hopps excels (up to 10× speedup) in workloads with high storage requirements (YARA and ClamAV). Hopps's advantage is lower for smaller workloads, which fit on-chip even without compression (FileCarving and Protomata).

Hopps's good performance comes from both storing automata efficiently, which reduces the number of batches, and its effective execution model and partition algorithm, which allow the common case (frequent nodes) to run fast while hiding the latency of processing highly compressed infrequent nodes.

**7.1.1 Hopps is area-efficient.** Fig. 14 shows the reduction in SRAM bits for storing each workload on Hopps compared to on Impala-based. We include storage for both non-doomed and doomed nodes. We achieve high storage reduction (gmean 3.2×) across all workloads except FileCarving, a small benchmark with nodes that have many neighbors each. This limits compression opportunities in both Hare and Tortoise units. This result shows that our approach of using compressed formats and our design of hardware units are highly effective.

The reduction in batches can be lower than bits reduction, because SRAM area does not scale perfectly with capacity, and because hardware units can be underutilized when the ratio of storage needed (Hare unit vs. Tortoise unit vs. match set) vary across workloads and mismatch with what Hopps provisions. In addition, the number of equivalent batches for doomed nodes (Table 7) also depends on node activity.

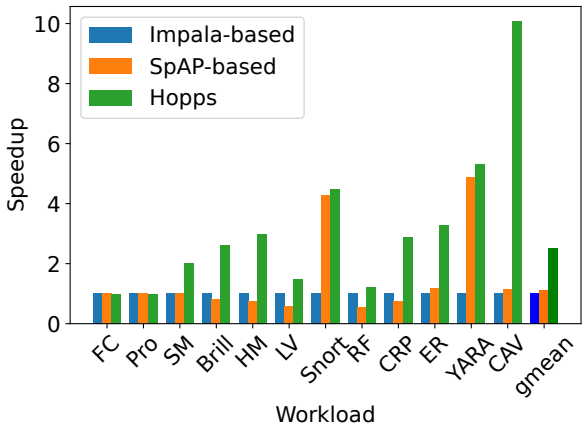



**Figure 13.** Speedup normalized to Impala-based.

| Workload | Impala-based | | SpAP-based | | Hopps | |
|---|---|---|---|---|---|---|
| | Batch | Thpt | Batch | Thpt | Batch | Thpt |
| FC | 1 | 1.0 | 1 | 1.0 | 1+0.0005 | 1.0 |
| Pro | 1 | 1.0 | 1 | 1.0 | 1+0.03 | 0.97 |
| SM | 2 | 0.5 | 2 | 0.5 | 1 | 1.0 |
| Brill | 3 | 0.33 | 3+0.61 | 0.28 | 1+0.14 | 0.87 |
| HM | 3 | 0.33 | 4+0.09 | 0.25 | 1+0.001 | 0.99 |
| LV | 3 | 0.33 | 5+0.10 | 0.2 | 2+0.002 | 0.49 |
| Snort | 5 | 0.2 | 1+0.17 | 0.86 | 1+0.1 | 0.90 |
| RF | 6 | 0.17 | 11 | 0.09 | 5+0.001 | 0.20 |
| CRP | 6 | 0.17 | 8+0.07 | 0.13 | 2+0.06 | 0.48 |
| ER | 10 | 0.1 | 8+0.56 | 0.12 | 3+0.05 | 0.33 |
| YARA | 22 | 0.045 | 4+0.52 | 0.22 | 4+0.14 | 0.24 |
| CAV | 60 | 0.017 | 3+49 | 0.02 | 4+2 | 0.17 |

**Table 7.** Number of batches and throughput (symbol/cycle) of Hopps and baselines.

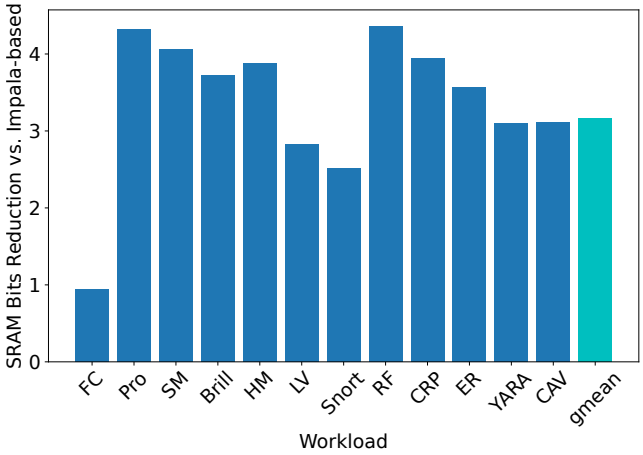



**Figure 14.** SRAM bits reduction over Impala-based.

**7.1.2 Hopps has an effective execution model and partitioning algorithm.** Fig. 15 shows the throughput of Hopps compared to an idealized version where Tortoise units have the same throughput as Hare units. For each workload, throughput is normalized to ideal (red dashed line).

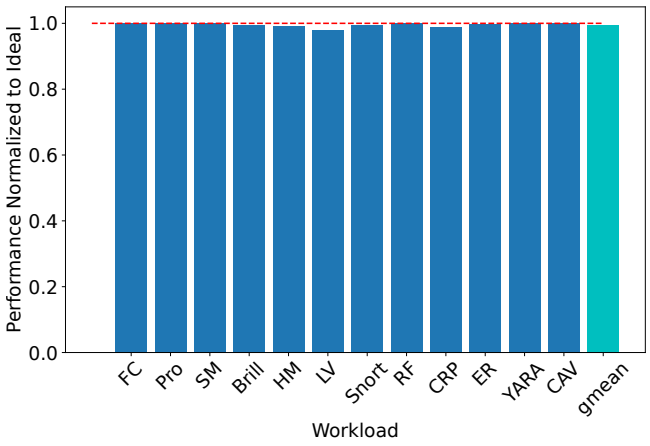



**Figure 15.** Performance of Hopps vs. an idealized version where Tortoise units are as fast as Hare units. The lower throughput of Tortoise units has a small performance impact.

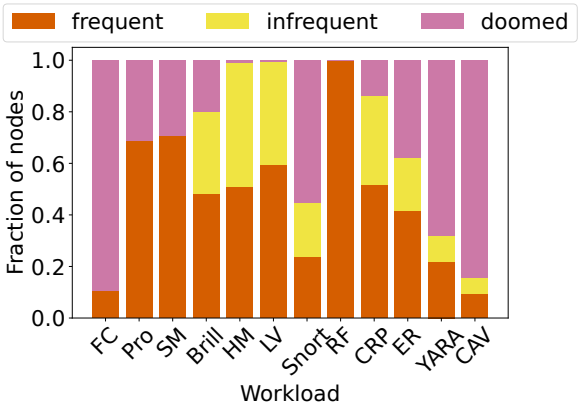



**Figure 16.** Fraction of frequent, infrequent, and doomed nodes in each workload.

Hopps is only gmean 0.5% slower than the idealized version. This shows that *(1)* our partitioning algorithm effectively identifies low-activity nodes and places them in Tortoise units, and *(2)* our execution model effectively hides the latency of Tortoise units. Even though Tortoise units take >5× latency than Hare units to process each symbol, nodes in Tortoise units are rarely activated, and infrequent-to-frequent transitions that cause Hare units to roll back are even rarer.

Hopps performs well under very different workload characterstics. Fig. 16 reports how the partitioning algorithm classifies nodes among frequent, infrequent, and doomed. Each bar shows results for one workload; the $y$-axis reports the fraction of nodes broken down by type.

Snort, YARA, and ClamAV benefit the most from doomed nodes. Most workloads have doomed nodes, which are very infrequently activated and are mapped to Hare units. Doomed nodes contribute to speedup over Impala because they can be efficiently processed in a separate pass, where symbols

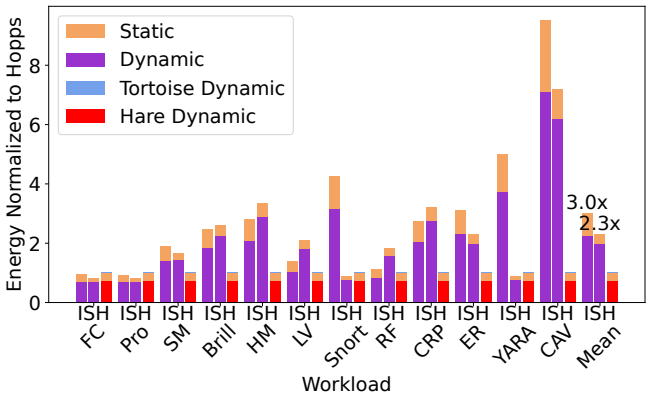



**Figure 17.** Energy breakdown for Hopps (H), SpAP-based (S), and Impala-based (I).

can be skipped if no node is activated. Although SpAP has this optimization too, we achieve >2× speedup over SpAP; Sec. 7.4's ablation study further details Hopps's advantage.

For smaller workloads FC, Pro, and SM, all or most non-doomed nodes are partitioned as frequent, because all automata can fit in one batch even if we map most nodes to Hare units.

Workload RF shows that Hopps's storage savings come not only from the compressed representation in Tortoise units, but also from the efficient design of Hare units. This is because Hopps's hybrid approach allows its Hare unit design to be less conservative. In RF, since almost all nodes are enabled in the profiling trace, all nodes are partitioned as frequent (Sec. 4.1) and mapped to Hare units. However, Hopps still outperforms both baselines, even though Hopps has less utilized area (the Tortoise units are unused). The crossbars in Impala and SpAP are sized such that typical-sized automata can be efficient, and the largest automata can be supported. However, by mapping infrequent nodes in each automaton to Tortoise units, Hopps effectively reduces the size of automata mapped to Hare unit crossbars, allowing Hare units to have smaller crossbars, which are more efficient for small automata, and can better exploit the coarse-grain sparsity in adjacency matrices. All automata in RF have 31 nodes, which efficiently fit in the 32-node crossbars of Hare units, leading to RF's good performance.

### 7.2 Energy

Fig. 17 shows the energy breakdown of Hopps, SpAP-based, and Impala-based, normalized to Hopps's total energy for each workload. Each group of 3 bars reports energy of these three systems on a particular workload. Energy consumption of each system is broken down into static or dynamic energy, and for Hopps, we further distinguish between dynamic energy consumed by Hare or Tortoise units.

For Hopps, activity (SRAM reads) in Hare units consume most of the energy, while Tortoise units have negligible

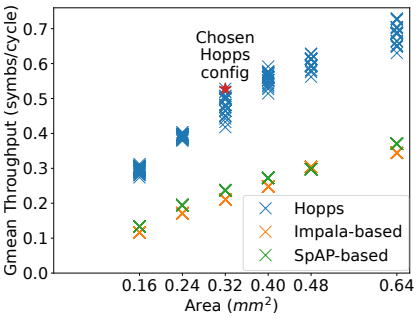



**Figure 18.** Performance of Hopps and baselines across areas and Hopps configurations.

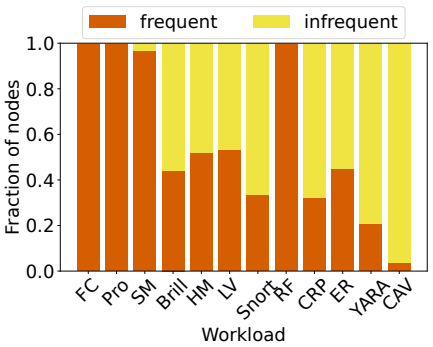



**Figure 19.** Node classification in Hopps-ND.

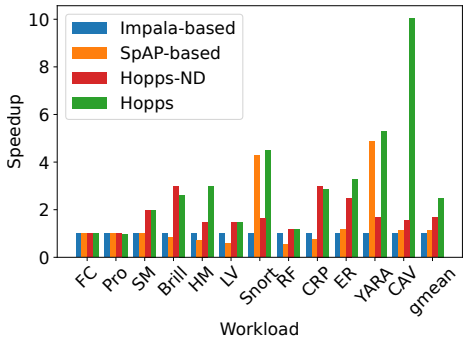



**Figure 20.** Speedup of Hopps-ND, as well as Hopps and baselines.

active energy. This demonstrates that Hopps successfully separates high-activity nodes and places them in Hare units.

Compared to Impala-based and SpAP-based, Hopps consumes 3.0× and 2.3× less energy on average. Hopps reduces both dynamic and static energy: dynamic energy is lower because Hopps effectively concentrates high-activity nodes in Hare units, so each SRAM read performs more effectual work; and static energy is lower because Hopps is faster, so each workload accrues static power over a shorter interval.

### 7.3 Sensitivity to design area

Hopps outperforms the baselines by a large margin across hardware unit provision ratios and chip areas. Fig. 18 shows the performance of Hopps and the baselines under different design areas. Each point represents one configuration for one system, with design area on the $x$-axis and gmean throughput (in symbols/cycle) across all workloads on the $y$-axis. For each design area, we also show Hopps's performance under different configurations (vertical blue points at the same $x$-coordinate). To explore configurations, we vary the percentage of area devoted to four components: the adjacency matrix and match set of Hare and Tortoise units. We sweep configurations where each component takes 20% to 30% of area in increments of 2.5%, with the condition that all four components add up to 100%. The Hopps configuration we use in other results is indicated with a star (and it is the best-performing one for its area budget). Across different configurations, each workload may need a different number of batches to map all its automata, leading to different performance. However, we see in Fig. 18 that Hopps consistently outperforms the baselines across the range.

### 7.4 Effect of doomed nodes

Hopps's partitioning algorithm combines frequent/infrequent node partitioning to use Hare and Tortoise units, and the doomed-node optimization of SpAP, which processes doomed nodes selectively in additional batches. To study the effect of these techniques in isolation, we evaluate a variant of Hopps, Hopps-ND, which does not use the doomed-node optimization. We configure Hopps-ND to use 65% of area for Tortoise units and 35% for Hare units (instead of 50%/50% as in our Hopps configuration), because Tortoise units become more important for low-activity nodes in Hopps-ND.

Fig. 19 shows the ratio of frequent/infrequent nodes in each workload for Hopps-ND. Across workloads, frequent nodes have very similar fractions as in Fig. 16; since doomed nodes in Hopps have low activity, they are classified as infrequent in Hopps-ND.

Fig. 20 shows the performance of Hopps-ND. Hopps-ND matches or outperforms Impala on all benchmarks, and matches or outperforms SpAP on all except Snort and YARA. Overall, Hopps-ND is gmean 1.7× faster than Impala and 1.5× faster than SpAP (when using the same 50%/50% configuration as Hopps, gmean speedups are 1.6× and 1.4×); speedups are more muted than Hopps (2.5× and 2.2×, respectively).

These results show that infrequent partitioning alone is an effective technique, and that many nodes can be classified as infrequent to improve performance (over 50% in most workloads, and up to 95% in CAV).

## 8 Additional Related Work

***Automata processing on GPUs:*** Several GPU automata processing implementations exploit sparsity [12, 22, 23]. Liu et al. [22] and ngAP [12] use a CSR-like representation for the adjacency matrix, and Liu et al. and AsyncAP [23] partition hot/cold nodes. However, these techniques aim to address GPU bottlenecks, which are very different from Hopps's. For example, Liu et al.'s use of CSR-style neighbor sets seeks to save memory traffic; follow-on AsyncAP observes that a transition table is more efficient. Second, Liu et al. and AsyncAP's hot/cold partitioning seek to improve thread utilization; they map hot nodes to threads one-to-one, but use shared worklists for cold nodes. Due to these different goals, GPU systems do not use a hybrid data representation like Hopps, where hot nodes are uncompressed, and cold ones compressed.

Furthermore, GPU implementations [12, 22, 23, 58] are orders of magnitude slower and less area-efficient compared to Hopps and other AP accelerators. The state-of-the-art system ngAP [12] achieves 115MB/s throughput on Snort on an NVIDIA RTX 3090 GPU. Hopps achieves 0.90 symbols/cycle; at a frequency of 2GHz, Hopps would achieve 1.8 GB/s, a >10× speedup. In addition, Hopps has an area of $0.32mm^2$, which is under 1/1000th of the area of the GPU die ($628mm^2$ at 8nm [27]). This yields a performance/area advantage beyond 10000×.

***Other general-purpose architectures:*** The CPU regex matching library Hyperscan [46] transforms and decomposes automata such that there are more string-matching and SIMD operations, which map well to CPUs. Its performance is ∼10× worse than state-of-the-art GPU solutions. CGRA dataflow accelerators have also been used for automata processing [32], reaching similar performance to GPUs while having 4× less area. However, this improvement is not enough to bridge the huge performance-per-area gap with AP accelerators.

***Crossbar-based AP accelerators:*** Automata processing accelerators use memory-based crossbars to provide high parallelism and reduce data movement. The AP chip [10] stores automata in DRAM, and adds routing hardware and simple logic to compute on automata. Cache Automaton [41] simplifies the routing, and stores automata in SRAM, achieving 3 orders of magnitude speedup over CPU, 10× over AP. Several prior systems [35–37] statically transform automata to use smaller symbols to reduce the size of the match set. This approach is orthogonal to our optimization on the adjacency matrix. UAP [11] and BVAP [47] improve performance by supporting variants of automata that are more efficient for specific workloads.

***Automata processing on FPGAs:*** FPGA-based automata processors have similar performance to the AP chip. REAPR and follow-on work [5, 29, 50] use LUTs or BRAM to store match sets, and encode adjacency matrices as routing information through 2D-mesh interconnects. FRA-FPGA [55] implements fast reconfiguration, and has uncompressed adjacency matrices of multiple sizes (36, 72, and 144) to increase utilization. hAP [45] achieves better energy efficiency by transforming and processing automaton subgraphs on von Neumann hardware. No FPGA implementation uses compressed representation for automata graphs, so our techniques should benefit these systems as well.

***Increasing parallelism in each automaton:*** Prior work proposes approaches that increase parallelism in a single automaton by eliding the dependence between consecutive state transitions. Wu et al. [57] use speculation to allow multiple threads to work on different input segments in parallel. Parallel prefix computation (scan) [3, 15, 18] is a widely used parallel algorithm that can also be used for automata processing; Mytkowicz et al. [25] build on this idea and propose a parallel algorithm that avoids dependences with low extra work. Workloads (e.g., Snort) often have many patterns (automata) to match against the input in parallel, so Hopps (and prior AP accelerators) study workloads with abundant parallelism across automata. However, these parallelization approaches are complementary and can be promising for workloads with few automata.

## 9 Conclusion

Automata processing requires high throughput, and storing automata on-chip efficiently is crucial to performance. But automata graphs are often sparse, and prior work uses inefficient uncompressed representation to store them. We have presented Hopps, an architecture that combines compressed and uncompressed data representations to improve storage efficiency and processing throughput, achieving gmean 2.5× and 2.2× speedups over prior state-of-the-art accelerators. Hopps makes storing infrequently activated nodes cheap, and processing frequently activated nodes fast. We have shown that under our execution model, offloading infrequent nodes to Tortoise units has throughput close to running all nodes in Hare units, while making both Hare and Tortoise units more storage efficient than prior work, leading to higher parallelism and throughput.

## Acknowledgments

We thank the anonymous reviewers; our shepherd, Yatish Turakhia; and Axel Feldmann, Hyun Ryong Lee, Fares Elsabbagh, Viansa Schmulbach, Courtney Golden, and Aleksandar Krastev for feedback on earlier versions of this manuscript. This work was funded in part by the National Science Foundation under grant CCF-2217099.

## References

[1] AMD. 2024. Vitis Data Analytics Library: Regular Expression Virtual Machine (regex-VM). https://docs.amd.com/r/en-US/Vitis_Libraries/data_analytics/guide_L1/internals/regex_vm.html.

[2] Michela Becchi, Charlie Wiseman, and Patrick Crowley. 2009. Evaluating regular expression matching engines on network and general purpose processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. https://doi.org/10.1145/1882486.1882495

[3] Guy E. Blelloch. 1989. Scans as primitive parallel operations. *IEEE Trans. Comput.* 38, 11 (1989). https://doi.org/10.1109/12.42122

[4] 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 *Proc. of the 24th IEEE intl. symp. on High Performance Computer Architecture*. https://doi.org/10.1109/HPCA.2018.00068

[5] Chunkun Bo, Vinh Dang, Ted Xie, Jack Wadden, Mircea Stan, and Kevin Skadron. 2019. Automata Processing in Reconfigurable Architectures: In-the-Cloud Deployment, Cross-Platform Evaluation, and Fast Symbol-Only Reconfiguration. *ACM Trans. Reconfigurable Technol. Syst.* 12, 2 (2019). https://doi.org/10.1145/3314576

[6] Pascal Caron and Djelloul Ziadi. 2000. Characterization of Glushkov automata. *Theor. Comput. Sci.* 233, 1–2 (2000). https://doi.org/10.1016/S0304-3975(97)00296-X

[7] Young H. Cho and William H. Mangione-Smith. 2005. A pattern matching coprocessor for network security. In *Proc. of the 42nd Design Automation Conf.* https://doi.org/10.1145/1065579.1065641

[8] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *Proc. of the 48th annual Intl. Symp. on Computer Architecture.* https://doi.org/10.1109/ISCA52012.2021.00053

[9] Timothy A Davis, Sivasankaran Rajamanickam, and Wissam M Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. *Acta Numerica* 25 (2016).

[10] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014). https://doi.org/10.1109/TPDS.2014.8

[11] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast support for unstructured data processing: the unified automata processor. In *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture.* https://doi.org/10.1145/2830772.2830809

[12] Tianao Ge, Tong Zhang, and Hongyuan Liu. 2024. ngAP: Non-blocking Large-scale Automata Processing on GPUs. In *Proc. of the 29th intl. conf. on Architectural Support for Programming Languages and Operating Systems.* https://doi.org/10.1145/3617232.3624848

[13] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proc. of the 52nd annual IEEE/ACM intl. symp. on Microarchitecture.* https://doi.org/10.1145/3352460.3358291

[14] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. 2023. LogNIC: A High-Level Performance Model for SmartNICs. In *Proc. of the 56th annual IEEE/ACM intl. symp. on Microarchitecture.*

[15] W. Daniel Hillis and Guy L. Steele. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (1986). https://doi.org/10.1145/7902.7903

[16] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proc. of the 28th intl. conf. on Architectural Support for Programming Languages and Operating Systems.* https://doi.org/10.1145/3582016.3582051

[17] Nangate Inc. 2008. The NanGate 45nm Open Cell Library. http://www.nangate.com/?page_id=2325.

[18] Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *J. ACM* 27, 4 (1980). https://doi.org/10.1145/322217.322232

[19] Vincent T. Lee, Justin Kotalik, Carlo C. Del Mundo, Armin Alaghi, Luis Ceze, and Mark Oskin. 2017. Similarity Search on Automata Processors. In *Proc. of the 31st IEEE Intl. Parallel and Distributed Processing Symp.* https://doi.org/10.1109/IPDPS.2017.12

[20] Tamara Lin. 2023. EmbedWay Shipping PA8921 FPGA Acceleration Card Based on Intel Agilex® 7 FPGAs F-Series Optimized for DPI and RDMA Network Acceleration. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2023-11/embedway-shipping-pa88921-fpga-acceleration-card-solution-brief.pdf.

[21] Hongyuan Liu, Mohamed Ibrahim, Onur Kayiran, Sreepathi Pai, and Adwait Jog. 2018. Architectural Support for Efficient Large-Scale Automata Processing. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture.* https://doi.org/10.1109/MICRO.2018.00078

[22] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. Why GPUs are Slow at Executing NFAs and How to Make them Faster. In *Proc. of the 25th intl. conf. on Architectural Support for Programming Languages and Operating Systems.* https://doi.org/10.1145/3373376.3378471

[23] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2023. Asynchronous Automata Processing on GPUs. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 1 (2023). https://doi.org/10.1145/3579453

[24] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proc. of the 28th intl. conf. on Architectural Support for Programming Languages and Operating Systems.* https://doi.org/10.1145/3582016.3582069

[25] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. In *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems.* https://doi.org/10.1145/2541940.2541988

[26] Quan M. Nguyen and Daniel Sanchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *Proc. of the 54th annual IEEE/ACM intl. symp. on Microarchitecture.* https://doi.org/10.1145/3466752.3480048

[27] NVIDIA. 2021. NVIDIA AMPERE GA102 GPU ARCHITECTURE. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[28] NVIDIA. 2021. NVIDIA BLUEFIELD-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf.

[29] Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. 2020. Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs. In *Proc. of the 28th IEEE Annual Intl. Symp. on Field-Programmable Custom Computing Machines.* https://doi.org/10.1109/FCCM48280.2020.00027

[30] Indranil Roy and Srinivas Aluru. 2014. Finding Motifs in Biological Sequences Using the Micron Automata Processor. In *Proc. of the 28th IEEE Intl. Parallel and Distributed Processing Symp.* https://doi.org/10.1109/IPDPS.2014.51

[31] Indranil Roy, Nagakishore Jammula, and Srinivas Aluru. 2016. Algorithmic Techniques for Solving Graph Problems on the Automata Processor. In *Proc. of the 30th IEEE Intl. Parallel and Distributed Processing Symp.* https://doi.org/10.1109/IPDPS.2016.116

[32] A. C. Rucker, S. Sundram, C. Smith, M. Vilim, R. Prabhakar, F. Kjolstad, and K. Olukotun. 2024. Revet: A Language and Compiler for Dataflow Threads. In *Proc. of the 30th IEEE intl. symp. on High Performance Computer Architecture.* https://doi.org/10.1109/HPCA57654.2024.00016

[33] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (second ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718003

[34] Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, Kevin Skadron, and Hongning Wang. 2018. A Scalable Solution for Rule-Based Part-of-Speech Tagging on Novel Hardware Accelerators. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining.* https://doi.org/10.1145/3219819.3219889

[35] Elaheh Sadredini, Reza Rahimi, Mohsen Imani, and Kevin Skadron. 2021. Sunder: Enabling Low-Overhead and Scalable Near-Data Pattern Matching Acceleration. In *Proc. of the 54th annual IEEE/ACM intl. symp. on Microarchitecture.* https://doi.org/10.1145/3466752.3480934

[36] 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 *Proc. of the 25th intl. conf. on Architectural Support for Programming Languages and Operating Systems.* https://doi.org/10.1145/3373376.3378459

[37] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. Impala: Algorithm/Architecture Co-Design for In-Memory Multi-Stride Pattern Matching. In *Proc. of the 26th IEEE intl. symp. on High Performance Computer Architecture.* https://doi.org/10.1109/HPCA47549.2020.00017

[38] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. eAP: A Scalable and Efficient In-Memory Accelerator for Automata Processing. In *Proc. of the 52nd annual IEEE/ACM intl. symp. on Microarchitecture.* https://doi.org/10.1145/3352460.3358324

[39] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. 2017. Frequent subtree mining on the automata processor: challenges and opportunities. In *Proc. of the Intl. Conf. on Supercomputing*. https://doi.org/10.1145/3079079.3079084

[40] Alireza Shafaei, Yanzhi Wang, Xue Lin, and Massoud Pedram. 2014. FinCACTI: Architectural Analysis and Modeling of Caches with Deeply-Scaled FinFET Devices. In *2014 IEEE Computer Society Annual Symposium on VLSI*. https://doi.org/10.1109/ISVLSI.2014.94

[41] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache automaton. In *Proc. of the 50th annual IEEE/ACM intl. symp. on Microarchitecture*. https://doi.org/10.1145/3123939.3123986

[42] Jack Wadden, Nathan Brunelle, Ke Wang, Mohamed El-Hadedy, Gabriel Robins, Mircea Stan, and Kevin Skadron. 2016. Generating efficient and high-quality pseudo-random behavior on Automata Processors. In *Proc. of the 34th Intl. Conf. on Computer Design*. https://doi.org/10.1109/ICCD.2016.7753349

[43] Jack Wadden and Kevin Skadron. 2016. *VASim: An Open Virtual Automata Simulator for Automata Processing Application and Architecture Research*. Technical Report CS2016-03. University of Virginia.

[44] Jack Wadden, Tommy Tracy, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Jeffrey Udall, Matthew Wallace, Mircea Stan, and Kevin Skadron. 2018. AutomataZoo: A Modern Automata Processing Benchmark Suite. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*. https://doi.org/10.1109/IISWC.2018.8573482

[45] Xuan Wang, Lei Gong, Jing Cao, Wenqi Lou, Weiya Wang, Chao Wang, and Xuehai Zhou. 2023. hAP: A Spatial-von Neumann Heterogeneous Automata Processor with Optimized Resource and IO Overhead on FPGA. In *Proc. of the 2023 ACM/SIGDA Intl. Symp. on Field-Programmable Gate Arrays*. https://doi.org/10.1145/3543622.3573190

[46] 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 *Proc. of the 16th USENIX Symp. on Networked Systems Design and Implementation*.

[47] Ziyuan Wen, Lingkun Kong, Alexis Le Glaunec, Konstantinos Mamouras, and Kaiyuan Yang. 2024. BVAP: Energy and Memory Efficient Automata Processing for Regular Expressions with Bounded Repetitions. In *Proc. of the 29th intl. conf. on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3620665.3640412

[48] Bob Wheeler. 2021. DPU-Based Hardware Acceleration: A Software Perspective. https://solutions.asbis.com/api/uploads/files/40/nvidia-doca-white-paper-final.pdf.

[49] Clifford Wolf. 2014. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.

[50] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea Stan. 2017. REAPR: Reconfigurable engine for automata processing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. https://doi.org/10.23919/FPL.2017.8056759

[51] Xilinx. 2019. RXP Accelerated Search & Analytics for Cyber Security, Big Data & AI/ML. https://www.xilinx.com/publications/solution-briefs/partner/titan_solution-brief.pdf.

[52] Yifan Yang, Joel S. Emer, and Daniel Sanchez. 2023. ISOSceles: Accelerating Sparse CNNs through Inter-Layer Pipelining. In *Proc. of the 29th IEEE intl. symp. on High Performance Computer Architecture*. https://doi.org/10.1109/HPCA56546.2023.10071080

[53] Amir Yazdanbakhsh, Ashkan Moradifirouzabadi, Zheng Li, and Mingu Kang. 2022. Sparse Attention Acceleration with Synergistic In-Memory Pruning and On-Chip Recomputation. In *Proc. of the 55th annual IEEE/ACM intl. symp. on Microarchitecture*. https://doi.org/10.1109/MICRO56248.2022.00059

[54] 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 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*. https://doi.org/10.1145/1185347.1185360

[55] Peng Zhang, Shijun Zhang, Shang Li, Jin Zhang†, Shaoxun Liu, and Youjun Bu. 2022. FRA-FPGA: Fast Reconfigurable Automata Processing on FPGAs. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. https://doi.org/10.1109/FPL57034.2022.00055

[56] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *Proc. of the 14th USENIX symp. on Operating Systems Design and Implementation*.

[57] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the "embarrassingly sequential": parallelizing finite state machine-based computations through principled speculation. In *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/2541940.2541989

[58] 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 *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. https://doi.org/10.1145/2145816.2145833