

BitstreamZoo: A Benchmark Suite for Bitstream Workloads on Commodity CPUs and GPUs

Hongyuan Liu

Stevens Institute of Technology
hliu96@stevens.edu

Tianao Ge

Shenzhen University of
Advanced Technology
getianao@suat-sz.edu.cn

Ming Li[†]

Stevens Institute of Technology
liming950814@gmail.com

James Contini

Stevens Institute of Technology
jcontini@stevens.edu

Abstract—Bitwise-parallel workloads pack many independent data items into bitstreams, so a bitstream operation describes work over all stream positions at once. This execution style appears in application areas as varied as cryptography, text processing, and neural network inference, yet these areas have largely been studied using separate, domain-specific tools. Existing CPU and GPU benchmark suites do not represent packed bitstream computations in a common form. To address this gap, we present BitstreamZoo, an open benchmark suite that expresses 15 bit-level parallel workload families in a shared bitstream program format. We also build reference CPU and GPU backends that interpret this format directly. We characterize the programs by their operation mix, dependency structure, and memory behavior. The workloads span orders of magnitude in dependency depth and in the number of operations available at each dependency level. We also find that executing one bitstream operation at a time as the reference backends do materializes every intermediate stream and can leave substantial inter-operation parallelism unexploited. BitstreamZoo gives compiler and architecture researchers a reproducible workload set for bit-level parallel programs on CPUs and GPUs.

I. INTRODUCTION

Many throughput-oriented kernels extract parallelism from bits. They pack many records into long bitstreams, so one machine-word or vector instruction updates many records at once. A **bitstream program** formalizes this idea. In a bitsliced layout, N records of k bits become k bitstreams of length N , so stream position i across those streams belongs to record i . A text buffer of N bytes is the $k=8$ case.

Consider a regular-expression matcher scanning a 512-byte input. It processes the bytes as bitstreams (Section II) and combines them with bitwise Boolean AND, non-bitwise SHIFT, and bitstream ADD. The result is a bitstream whose bit at stream position i indicates whether a match ends at input position i . This bitstream abstraction appears across published systems and algorithms, including text and JSON parsing [24], [27], [44], bitsliced cryptography [5], [28], binarized neural networks [43], bit-parallel bioinformatics [32], database scans [26], polar coding [2], circuit simulation [7], and gate-decomposed processing-in-memory (PIM) arithmetic [20], [38]. Although these applications assign different meanings to their bits, their low-level execution often reduces to the same

compute model. However, these communities have grown up in parallel, each with separate tool chains and benchmark practices.

General-purpose benchmark suites such as Rodinia [11], Parboil [40], and SHOC [12] do not represent packed bitstream computations in a shared program format. Prior research addresses adjacent problems closely but leaves this gap open. Usuba [28] and BitGen [17] are domain-specific compilers. PIMbench [39] evaluates simulated PIM hardware. BitBench [13] targets stochastic computing, where each wire encodes a probability. Section VI compares these in detail.

To study the diversity and characteristics of bitstream programs, we propose BitstreamZoo, a benchmark suite that provides a shared **bitstream program format** for expressing workloads across application areas. BitstreamZoo includes 15 workload families and expresses each program in the same format, a simple assignment language over named bitstream variables. Its values are unbounded bitstreams and indexed bitstream arrays, and its operators include bitwise Boolean logic, cross-position shifts, bitstream addition, and reduction. We also provide simple reference backends for CPUs and GPUs and characterize the programs when executed on these backends. In this work, we answer two questions: 1) What do bit-parallel workloads have in common across workload families, and how widely do they vary? 2) What costs does running one bitstream operation at a time incur, and what opportunities exist to reduce them?

This work makes the following contributions:

- **BitstreamZoo, an open benchmark suite.** It provides bitstream programs across 15 workload families in one shared program format. The same bitstream program runs on reference CPU and GPU backends.
- **Characterization of bit-parallel workloads.** We characterize program structure (Section IV) and the execution (Section V) across the suite. We observe that parallelism is abundant but heterogeneous, with dependency depth and width that span orders of magnitude across programs. Materializing every bitstream operation creates orders-of-magnitude memory traffic and leaves the GPU underutilized. Together, these findings open a design space for compiler and architecture research on bitstream backends.

[†]Ming Li contributed to this work after being admitted to the Ph.D. program at Stevens Institute of Technology, but was ultimately unable to enroll due to visa-related circumstances.

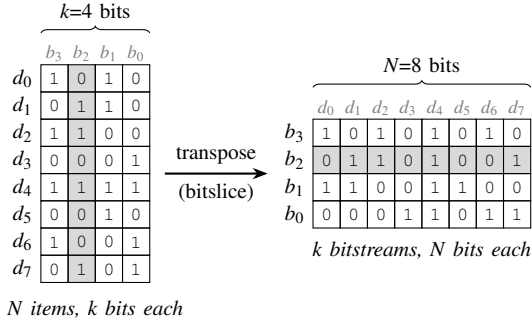


Fig. 1: **Bitslicing**. Eight 4-bit data items (left) are transposed into four bitstreams of 8 bits (right). The shaded column b_2 on the left becomes the shaded row b_2 on the right. A single word- or vector-wide bitwise Boolean operation then processes many stream positions of that bitstream in parallel.

II. BITSTREAM REPRESENTATION

Bitstreams and Bitslicing. An **unbounded bitstream** is a sequence of bits of arbitrary length. A backend stores each one as an array of machine words in memory. Input data become bitstreams by packing, determined by the application. Throughout this paper, a **stream position** is an index inside one bitstream. In a bitsliced representation, stream b_j is the bitstream holding the j th bit from many packed items. Figure 1 shows the bitslicing process that transposes N values of k bits into k bitstreams of N bits, one stream for each value-bit index. The bit at stream position i of stream b_j is the j th bit of data item i . This layout exposes bit-level parallelism: a word- or vector-wide bitwise Boolean operation processes many stream positions of one such bitstream at once.

A. The Program Format

BitstreamZoo makes this abstraction executable through one **bitstream program format**, adapted from the Parabix unbounded-bitstream model [10], [27]. A program is a simple assignment language over named bitstream variables. The stream-valued state consists of bitstreams and fixed-size arrays of bitstreams. Scalar integers supply parameters, loop bounds, shift amounts, and array indices, and they also hold `popcount` results in integer locals and integer outputs. No operator converts an integer back into a bitstream.

Operations and Semantics. Table I lists the operator set. A **bitstream operation** is one executed instance of an operator over whole input and output bitstreams. We use **bitwise** only for Boolean operators whose output bit at stream position i depends on input bits at the same stream position i . These bitwise Boolean operators are AND, OR, XOR, and NOT. A shift moves bits across stream positions, with output bit i reading input bit $i-n$ for a left shift and $i+n$ for a right shift. In the benchmark programs the offset n is a static function of parameters and loop variables. Bitstream addition propagates a carry across stream positions, and `popcount` collapses a bitstream to one integer. Thus shifts, bitstream addition, and `popcount` are bitstream operators but not “bitwise” operators.

TABLE I: The BitstreamZoo operator set.

Category	Operator	Syntax	Meaning
Bitwise Boolean	AND	$a \ \& \ b$	Pointwise AND
	OR	$a \ \ b$	Pointwise OR
	XOR	$a \ ^ \ b$	Pointwise XOR
	NOT	$\sim a$	Pointwise complement
Arithmetic	ADD	$a \ + \ b$	Carry-propagating addition
Shift	LSHIFT	$a \ \ll \ n$	Left shift by <code>int</code>
	RSHIFT	$a \ \gg \ n$	Right shift by <code>int</code>
Reduction	popcount	<code>popcount (s)</code>	Count set bits $\rightarrow$ <code>int</code>
Constants	ZERO	ZERO	All-zeros bitstream
	ONES	ONES	All-ones bitstream

```

1 // MatchStar: regex cursor extension (4 ops)
2 // Used inside the Regex family for Kleene star / plus
3 input stream M // cursor stream positions
4 input stream C // character class
5 output stream result
6
7 stream _t0 = M & C
8 stream _t1 = _t0 + C
9 stream _t2 = _t1 ^ C
10 result = _t2 | M

```

Fig. 2: The MatchStar [10] primitive in our bitstream program format, used inside the Regex family to advance cursors through character-class runs. The bitstream `+` extends cursor positions, and each remaining line is one bitwise Boolean operation.

Several workload families need bitstream addition and `popcount` directly. Bitstream addition is essential for Parabix-style cursor advancement in multi-pattern regex matching (Figure 2), and carry-propagating bit operators of the same family appear in JSON parsing for structural and escape detection (simdjson [24]). `popcount` crosses the bitstream-to-integer boundary and feeds statistical reductions such as the epistasis contingency tables of genome-wide gene-gene interaction screens [47].

Control Flow and Arrays. The format has three control-flow constructs. A `for i in lo..hi` loop iterates over a scalar induction variable, and its bounds are integer expressions over parameters and enclosing loop variables. `For` loops may nest to express multi-dimensional iteration. An `if (cond)` block uses a global decision: it executes the body across all stream positions when any bit of `cond` is set, an OR-reduction, with no per-position masking. A `while (cond)` loop uses the same OR-reduction and repeats until `cond` is all-zero.

A program’s **parameters** are named integer constants that instantiate a benchmark to one configuration, such as its input length. They are supplied at program load and stay fixed. Loop bounds, array sizes, shift amounts, and array indices are integer expressions over these parameters. Most evaluated programs use only `for` loops, so fixing the parameters lets loop unrolling yield the full operation sequence before execution.

The Regex programs are the exception, because their compiled patterns carry data-dependent control flow (Section III).

An `if` block is a scheduling decision, because its body computes zero whenever its test stream is zero. A `while` loop has no static trip count.

Bitstream arrays (`stream a[N]` with `int`-indexed access) hold one bitstream per value-bit index, recovering the bitsliced view of multi-bit values from Figure 1. An array element such as `a[j]` denotes the j th whole bitstream of the array.

B. The Benchmark Suite Overview

Terminology. We describe the suite at three levels. An **application area** is a broad category such as cryptography or text processing. A **workload family** derives from one cited algorithm or system, and Table II lists the 15 families, one row each. A **benchmark program** is an executable unit of the shared program format. A family has one or more programs, and the 15 families comprise 22 programs.

Reference Backends. BitstreamZoo provides two reference backends, one for the CPU and one for CUDA. Both execute bitstream operations in program order. Each bitwise Boolean or shift operation is one kernel or loop pass, while bitstream `ADD` and `popcount` expand to the scan and reduction sequences.

CPU Backend. The CPU backend stores each bitstream in a contiguous array of 64-bit integers. Bitwise Boolean operations are implemented with AVX-512 intrinsics and invoked through the interpreter’s operator dispatch. Bitstream addition supports two selectable modes. The ripple-carry mode uses one sequential chain of depth $O(N/W)$, where N is the bitstream length and W is the word width. The Kogge-Stone mode runs $O(\log N)$ prefix rounds with $O((N/W) \log N)$ total work.

GPU Backend. Each CUDA thread processes 64 bits of each bitstream. Bitstream variables and arrays live in device memory managed by the interpreter. The host-side interpreter walks the program’s control flow, including bounded `for` loops. Each executed bitwise Boolean or shift operation launches one operator kernel. The CUDA backend expands bitstream addition into a multi-kernel generate-propagate and an NVIDIA CUB [29] parallel-prefix scan. It expands `popcount` into a per-thread `popcount` kernel followed by a CUB reduction.

Test Infrastructure. Most programs provide up to three **input sizes**, near 50 MB, 500 MB, and 5 GB of total binary data across input and output streams. Inputs are deterministic across runs, either synthetic under a fixed RNG seed or pinned to versioned real-world datasets, and outputs are checked against the family’s reference where one is available (Section III).

III. WORKLOAD FAMILIES

BitstreamZoo includes 15 workload families across diverse application areas, comprising 22 programs.

Selection Method. The intended population is published bit-level parallel kernels: kernels that a cited system or algorithm runs over data packed by stream position. We include a family when it meets three rules: a) It has a cited reference algorithm or system. b) Its core kernel is expressible in the Table I operator set. c) It has a validation oracle, either a published vector or an independent scalar reference.

Packing Semantics. We distinguish two packing modes for what stream position i means across families. Under **instance-parallel** packing each stream position is one independent problem instance, as in the crypto kernels, Polar, Circuit Sim, Edit Distance, BNN, and GEMV, where the BNN and GEMV stream widths are the inference batch size. Under **record-parallel** packing each stream position is one record inside a larger input, as in the text, database, and Epistasis kernels. These families get their parallelism from large inputs.

Workload Families. We describe each family below, grouped by application area, and Table II summarizes each.

- **Cryptography.** AES S-box is the 113-gate NIST refinement of the Boyar-Matthews-Peralta circuit [6], [34], 117 operations after XNOR expansion. Trivium is the eSTREAM 288-bit stream cipher [15]. SIMON 32/64 is the NSA lightweight block cipher [4] with rotations as free bitsliced permutations. Montgomery Mul is a bit-serial modular-reduction kernel [31] at radix $R=2^{12}$ over the ML-KEM modulus $q=3329$ [33].
- **Text and pattern matching.** Shift-And is Bitap exact matching [3] over DNA character-class bitstreams [27]. Regex is a kernel compiled by `icgrep` [10] from AutomataZoo Snort patterns [46], 2,312 of 2,396 compiling through `icgrep`. Its three variants are distinct programs at 50, 500, and 2,312 patterns, named Regex small, Regex medium, and Regex large. A stub pattern contains constructs the `icgrep` pipeline does not support, so its output compiles to a constant ZERO stream. An active pattern compiles to input-dependent matching logic. Stubs are a compile-time property and concentrate among the smallest compiled patterns. 84% of Regex small’s patterns are stubs, so aggregate regex claims use Regex medium and Regex large, where 92% and 98% of patterns are active. Regex is also the only family with data-dependent control flow, because `icgrep` emits an `if` block for 97% of the patterns and a `while` loop for 8.5%. Mison is the structural-index kernel of the Mison JSON parser [24], [25]. BitFunnel is the row-intersection kernel of bitsliced Bloom-filter search [18] on synthetic rows.
- **Bioinformatics and statistics.** Edit Distance is Myers’ bit-parallel Levenshtein DP [32], the one kernel needing a multi-bit decrement. Epistasis is a binary 2×2 SNP contingency-table kernel [47], [48], the integer-output family, using 4 `popcount` reductions.
- **Neural network inference and linear algebra.** BNN is a **gate-decomposed** XNOR-popcount-threshold fully-connected layer following FINN’s Matrix-Vector-Threshold Unit (MVTU) [43]. Gate-decomposed means the arithmetic expands into bitwise Boolean operations, so the `popcount` reduction runs through a tree of half and full adders instead of integer instructions. Its oracle is a trained MNIST classifier with layer widths 784, 1024, 1024, and 1024. The kernel must reproduce this classifier’s per-neuron activations on the MNIST test set bit-for-bit. The full classifier costs 27.5M operations, so we characterize and benchmark a smaller network with layer widths 128, 256, 256, and 4 on synthetic inputs. This smaller configuration

TABLE II: Representativeness and validation of the 15 workload families. *Reference* is the cited source the kernel is derived from. *Oracle* is the reference each kernel is validated against. *Note* gives the scope of the modeled kernel.

Family	Reference	Oracle	Note
AES S-box	Boyar-Matthews-Peralta NIST circuit [6], [34]	All 256 bytes vs. standard S-box	Exact 113-gate circuit
Trivium	eSTREAM cipher [15]	eSTREAM all-zero test vector	Keystream generation only
SIMON 32/64	NSA cipher [4]	Scalar reference	Full 32-round encryption
Montgomery Mul	Montgomery reduction [31]	Scalar reference	Single Montgomery multiply, random operands
Shift-And	Bitap matching [3]	Scalar match	Single pattern over DNA classes
Regex	icgrep on AutomataZoo Snort [10], [46]	icgrep anchor-free semantics	Multi-pattern Snort matching
Mison	Mison JSON index [25]	Scalar index	Structural index stage only
BitFunnel	Bit-sliced Bloom search [18]	Scalar intersection	Synthetic rows, full traversal
Edit Distance	Myers bit-vector DP [32]	Scalar Levenshtein	Exact bit-parallel DP kernel
Epistasis	SNP contingency tables [47], [48]	Scalar contingency count	Single SNP-pair contingency table
BNN	FINN MVTU [43]	Trained-classifier activations on MNIST	Gate-decomposed adder tree
GEMV	Bit-serial in-cache GEMV [16]	Scalar matrix-vector	Gate-decomposed portability kernel
BitWeaving	Bitsliced column scan [26]	Scalar predicate	Single range predicate
Polar	Arikan transform [2]	Scalar XOR network	Butterfly transform only
Circuit Sim	ISCAS-85 netlists [7]	Scalar gate simulation	Exact gate-level netlists

is validated against a scalar XNOR-popcount-threshold reference. GEMV is a gate-decomposed bit-serial K -bit matrix-vector multiply, in the style of bit-serial in-cache arithmetic [16].

- **Database, coding, and circuits.** BitWeaving is a range-predicate bitsliced column scan [26] with built-in popcount, a database kernel in PIM studies [20], [38]. Polar is the Arikan butterfly XOR transform [2], the core encoder inside 5G NR polar coding [1], at $N \in \{64, 256, 1024\}$. Circuit Sim is gate-level simulation of the four representative ISCAS-85 netlists (c17, c432, c6288, c7552) [7].

IV. WORKLOAD CHARACTERIZATION

This section characterizes BitstreamZoo statically from the bitstream program. The goal is to see what the programs look like and where the bottlenecks and optimization opportunities lie. Section V then measures dynamic behavior on the reference backends. We measure dependency depth and parallelism (Table III), then model memory traffic and stream liveness in Section IV-B.

A. Dependency and Parallelism

After parameters are fixed and loops are unrolled, each benchmark becomes an operation graph whose nodes are bitstream operations and whose edges are producer-consumer dependencies between streams. We use this graph to measure parallelism between bitstream operations. This is separate from data-level parallelism, since each node in the graph already represents one bitstream operation over all stream positions (Section II).

Figure 3 illustrates the graph-level metrics. The first metric we use is **dependency depth**, the length of the longest producer-consumer chain in the graph. If a program folds one associative operator (AND, OR, XOR, ADD) over many streams in a linear chain, we re-balance that chain into a logarithmic-depth tree before measuring depth. A linear chain of OR operations, such as the stage that combines many pattern matchers, has depth equal to its operation count. OR is associative, so this fold depth

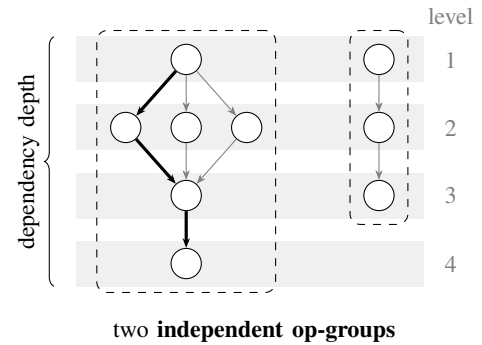


Fig. 3: Parallelism in an operation graph. Nodes are bitstream operations, edges are producer-consumer dependencies, and shaded bands are dependency levels. The bold path sets the **dependency depth**, 4 levels here. Dividing the 9 operations by this depth gives $9/4 \approx 2.3$ **operations per level**, the derived width metric. The dashed boxes are the two **independent op-groups**, the connected components of the graph and a conservative count of independent tasks. Data-level parallelism is intrinsic to every node, which already operates over the full bitstream.

reflects how the reduction was written rather than an inherent dependency. With reductions balanced, no schedule executes fewer than this many dependent levels, even with unlimited hardware parallelism. Let M be the number of bitstream operations. We define **operations per level** as M divided by dependency depth. This derived width metric shows operation-level parallelism: high values indicate many operations that can run at the same dependency level.

To see whether a program splits into independent tasks, we also count **independent op-groups**: the connected components in the operation graph, ignoring edge direction. Two operations are in the same group if a chain of stream dependencies connects them. Different groups share no producer-consumer dependencies, so each group can run as a separate task.

TABLE III: Program-structure metrics across 15 workload families (22 programs), computed on the program text at each program’s medium-size parameters. Mison’s operation count varies with input size. *Peak temporary streams (seq)* is the largest number of temporary streams simultaneously live at any program point during sequential execution. (*par*) is the same count when every same-depth operation fires in parallel. *Traffic ratio* is full-stream memory traffic under one-kernel-per-operation execution divided by an ideally fused backend’s traffic (Section IV-B). † marks the programs holding a `while` loop, whose trip count depends on the input. We count each such body once.

Program	Operations	Dependency depth	Ops per level	Peak temp streams (seq)	Peak temp streams (par)	Independent op-groups	Traffic ratio
AES S-box	117	27	4.3	28	32	1	22
Trivium	14,720	43	342.3	320	851	1	197
SIMON	2,048	65	31.5	49	80	1	11
Montgomery Mul	2,437	129	18.9	62	260	1	121
Shift-And	61	15	4.1	6	42	1	5
Regex (small)	271	11	24.6	30	156	4	82
Regex (medium)†	52,034	78	667.1	482	18,600	1,377	15,652
Regex (large)†	1.2M	1732	699.5	2,304	470,885	109,772	354,794
Mison	118	73	1.6	17	22	1	26
BitFunnel	23	7	3.3	8	13	1	6
Edit Distance	928	58	16.0	29	198	1	48
Epistasis	14	4	3.5	4	9	2	5
BNN	939,712	177	5,309.1	538	133,412	1	26
GEMV	13,312	48	277.3	15	2,560	16	59
BitWeaving	117	15	7.8	19	66	1	12
Polar (small)	192	6	32.0	59	119	6	4
Polar (medium)	1,024	8	128.0	249	501	8	6
Polar (large)	5,120	10	512.0	1,015	2,035	10	8
Circuit (c17)	12	6	2.0	4	5	1	4
Circuit (c432)	440	43	10.2	39	81	1	25
Circuit (c6288)	4,544	245	18.6	286	330	2	179
Circuit (c7552)	5,124	65	78.8	650	973	46	41

Finding 1: Parallelism is abundant but heterogeneous.

Parallelism is abundant because it stacks at three granularities. Within one operation, all bit positions are processed at once, so data-level parallelism is free. Across operations, those at the same dependency level are independent and fire together, and 15 of 22 programs exceed 10 operations per level. At the largest scale, independent op-groups run as separate tasks.

Graph shape varies sharply across programs, because it follows the algorithm. Depth spans 4 to 1,732 levels and operations per level span 1.6 to 5,309. A butterfly transform like Polar runs shallow and wide, while a multiplier like c6288 runs deep and narrow. No single schedule therefore fits the whole suite.

The op-group count signals task-level parallelism, but only loosely. 12 of 22 programs have a single op-group, yet several remain wide, BNN among them at 5,309 operations per level. GEMV (16) and Circuit c7552 (46) do break into genuine independent subgraphs. Regex large reports 109,772 op-groups, yet one component holds 90.9% of its operations. Extracting task-level parallelism therefore needs graph partitioning.

B. Modeled Memory Traffic and Stream Liveness

The dependency metrics above describe the shape of the operation graph. We consider two related views for memory cost: 1) full-stream traffic summed over the execution, and 2) the number of streams live at a single program point.

Full-stream traffic. We first model the traffic induced by our backends’ materialization policy. The model counts full-stream reads and writes per bitstream operation, independent of the concrete stream length.

In our backends, each operation’s result is materialized and later reread by its consumers. A binary bitwise Boolean operation reads two full streams and writes one. A unary bitwise Boolean operation reads one full stream and writes one. Scan and reduction operations can move more data in the reference backends. Bitstream addition is lowered to a carry-propagating scan sequence, and `popcount` is lowered to a reduction sequence. The model counts one read per input stream and one write per output stream for each bitstream operation. A `popcount` reads one full stream and writes only a single integer, negligible next to that read. The extra carry-scan and reduction passes inside the `ADD` and `popcount` expansions are omitted, so the estimate is a lower bound for programs that use them.

For comparison, we use an ideal fused lower bound. A fused region must read each distinct live-in stream at least once and write each live-out stream at least once. Some regions also need boundary or carry streams, such as carry state for bitstream addition, and we count these as n_{bc} . Thus the fused lower bound is $n_{in} + n_{bc} + n_{out}$ full-stream transfers. Here we use the whole-program fused value only as an optimistic lower bound, so a backend that fuses in segments lands strictly above it.

The resulting per-operation-to-fused traffic ratio is a program metric. For a fixed operation graph, the ratio is identical at every input size, because both the per-operation traffic and the fused lower bound scale with the stream length.

Finding 2: Materializing every bitstream operation creates orders-of-magnitude excess memory traffic. The *Traffic ratio* column of Table III varies widely across the suite. It ranges from $4\times$ for Circuit c17 to $354,794\times$ for Regex large. Outside the Regex variants, the ratio still reaches $197\times$ for Trivium and $26\times$ for BNN.

The ratio grows when many bitstream operations run over a small number of streams. Section V-A measures the performance effect of Finding 2 on the straight-line Boolean subset.

Stream liveness. This view asks how many temporary streams must be resident at once. The same materialization policy also creates large live storage even before considering fusion or scheduling.

Finding 3: Parallel execution amplifies stream liveness by orders of magnitude. Table III reports the live-temporary peak under two execution models: sequential execution, and a fully parallel schedule that fires all operations at the same dependency level together. Sequential execution already keeps hundreds of streams live on the largest programs (Regex large $2,304\times$, Polar large $1,015\times$, c7552 $650\times$). The parallel column is $248\times$ larger on BNN, $204\times$ larger on Regex large, and $171\times$ larger on GEMV, but barely changes on narrow or shallow programs (Mison $1.3\times$, Circuit c17 $1.25\times$). The amplification is largest when a program is both wide and deep, exposing many simultaneously-live streams under maximal parallelism. Reconciling the graph-level parallelism of Finding 1 with finite device memory therefore demands non-trivial scheduling.

V. DYNAMIC CHARACTERIZATION

Section II-B introduced the shared BitstreamZoo program format and the reference backends that execute it. We now use those backends to measure the cost of executing bitstream programs one operation at a time. This per-operation baseline sets up the rest of the section, which tests loop fusion, operator lowering, and existing rewrite systems as ways to reduce that cost.

Measurement Methodology. We implement both reference backends. We run the CPU backend single-threaded on one core of an AMD Ryzen Threadripper PRO 7965WX, using AVX-512 intrinsics. We run the GPU backend on an NVIDIA RTX PRO 6000 Blackwell Max-Q. It launches one CUDA kernel per bitwise Boolean or shift operation. Bitstream ADD and popcount use the scan and reduction sequences described in Section II-B. We compile the CPU backend with `-O3 -march=native` and the CUDA backend with `-O3` for the native GPU architecture. The software environment is GPU driver 580.82.07, CUDA 13.0, Ubuntu 22.04.5, and Linux kernel 5.15. We report end-to-end runtime for each backend run as the median over the number of repetitions stated in each caption.

A. Execution Overhead and Fusion

We ask where per-operation execution spends its time, and whether loop fusion across bitstream operations can reduce it. **Finding 4: Per-operation execution leaves the GPU under-utilized.** Table IV reports CPU and GPU runtime at the small input size. The *Compute* column divides the INT32 work of each program’s bitstream operations by its measured kernel time and by the GPU’s single-operation peak, its 24,064 CUDA cores at 2.28 GHz [35]. It stays between 0.002% and 0.067% on every program, so the kernels use almost none of the INT32 peak. The main cost comes from the execution model itself: each bitstream operation becomes a separate kernel, and each kernel moves whole streams through memory. This is exactly the traffic modeled by Finding 2.

Fusion. Since the bottleneck is repeated memory traffic, a backend should try to fuse operations. Fusion executes several bitstream operations in one pass and keeps intermediate streams in registers instead of writing them out and reading them back.

We use CPU code generation to isolate the cost of intermediate materialization. The experiment covers five straight-line Boolean programs: AES S-box and the four ISCAS-85 netlists (c17, c432, c6288, and c7552). Our fused C generator supports bitwise Boolean operations, but not stream shifts, bitstream +, popcount, or loop semantics. Programs using those features are therefore excluded.

For this subset, fusion is well defined because each output word depends only on input words at the same chunk index. We lower each program to a flat operation list and emit two C implementations. The per-operation implementation runs one vectorized loop per bitstream operation, materializing every intermediate stream in memory. The fused implementation places the operations in a single loop body and keeps intermediate values in registers when possible. Both implementations are compiled with `gcc -O3 -march=native -funroll-loops`. Each program runs on 32 Mibit input streams.

Over the median of 7 timed runs after 1 warmup, fusion improves runtime by $7.4\times$ on the 12-operation Circuit c17, $15.7\times$ on AES S-box, $34.1\times$ on Circuit c432, $79.0\times$ on the 4,544-operation Circuit c6288, and $18.7\times$ on the 5,124-operation Circuit c7552. The result confirms that per-operation execution pays for materializing intermediates heavily.

Multicore Scaling. Adding cores does not achieve the same effect. We enabled the CPU backend’s OpenMP mode and reran the 20 programs of Table IV at the small input size on up to 24 cores. All outputs match the single-core backend. The best speedup is $1.7\times$ on BNN, and 10 of the 20 programs run slower at 24 threads than on one core. This mode leaves shifts and ripple-carry addition serial, and parallelizing within each operation keeps the synchronization and full-stream materialization costs in place.

However, the fusion results do not directly extend to the rest of the operator set. Shifts require boundary values from neighboring chunks, bitstream addition carries information across stream positions, and popcount collapses many chunks into one scalar. These operators therefore introduce

TABLE IV: Repeated reference-backend runtime at the small input size, the median of 7 timed runs after 1 warmup. *GPU* is end-to-end device time including host-device copies; non-kernel overhead stays under 0.4% of it on every program. *Compute* is achieved SIMT INT32 ops per second as a fraction of the GPU SIMT non-Tensor INT32 peak, defined in Section V-A. *Spread* is the largest per-program (max – min)/median in percent. Relative spread stays under 20%, 11%, and 11% on CPU, GPU end-to-end, and GPU kernel. Section V gives the methodology and environment. Regex medium and Regex large are omitted because they are prohibitively slow on their single-core runs.

Program	Executed operations	CPU (1 core)	GPU end-to-end	Compute (% of peak)	Spread (%)
AES S-box	117	70 ms	9 ms	0.018	19.4
Trivium	14,720	3.68 s	758 ms	0.002	2.0
SIMON	2,048	282 ms	69 ms	0.011	4.6
Montgomery Mul	2,437	96 ms	14 ms	0.067	1.7
Shift-And	61	169 ms	34 ms	0.008	3.5
Regex (small)	271	195 ms	14 ms	0.048	3.2
Mison	110	56 ms	9 ms	0.024	4.4
BitFunnel	23	36 ms	12 ms	0.006	7.0
Edit Distance	928	118 ms	21 ms	0.028	4.2
Epistasis	14	64 ms	8 ms	0.013	14.1
BNN	939,712	17.84 s	3.05 s	0.053	5.4
GEMV	13,312	197 ms	72 ms	0.026	5.5
BitWeaving	117	88 ms	15 ms	0.017	3.9
Polar (small)	192	27 ms	7 ms	0.005	6.9
Polar (medium)	1,024	29 ms	12 ms	0.004	10.0
Polar (large)	5,120	30 ms	34 ms	0.002	10.5
Circuit (c17)	12	32 ms	7 ms	0.005	6.2
Circuit (c432)	440	48 ms	12 ms	0.019	4.1
Circuit (c6288)	4,544	197 ms	24 ms	0.066	5.5
Circuit (c7552)	5,124	83 ms	23 ms	0.016	3.3

cross-chunk communication or global reduction patterns that are absent from straight-line Boolean programs. Principled bitwise speculation [37] could target those dependences and a general fused backend may need to balance reuse against exposed parallelism. We leave such a backend to future work.

B. Scan and Reduction Operations

We next break down GPU kernel time by operation type to see where execution cost concentrates. Figure 4 reports this breakdown for all 22 programs. Bitwise Boolean operations dominate most programs, while scan and reduction operators appear in only a small subset. `ADD` is a scan (prefix sum) because the carry into each stream position depends on every lower position. `popcount` is a reduction because it folds a whole bitstream into one integer. `ADD` occurs in Mison and in the medium and large Regex variants. `popcount` occurs in Epistasis, BitFunnel, and BitWeaving.

Although these operators are less common than bitwise Boolean operations, they can still account for a noticeable fraction of execution time. Epistasis has the largest `popcount` share in the suite, even though `AND` operations still dominate its overall kernel time. In BitFunnel and BitWeaving, by contrast, `popcount` appears only as a small final reduction after a longer sequence of bitwise and shift operations.

These cases show that scan and reduction operators expose a different part of the backend design space. Their cost is not determined only by operation count. It also depends on how the backend lowers cross-position computation.

Finding 5: A non-bitwise operator’s cost depends on the backend’s lowering choice. For example, bitstream

addition (+) has three reference lowerings. The GPU backend expands each + into a five-kernel sequence that includes per-thread add, generate/propagate construction, a CUB prefix scan, and carry application. The CPU backend offers two SIMD modes for bitstream addition, ripple-carry and Kogge-Stone parallel prefix. Figure 5 measures the two CPU modes on the three programs that use the operator (Mison, Regex medium, Regex large), and the GPU cost is reported below.

On the CPU backend, Kogge-Stone is 1.6 to 1.7 $\times$ slower than ripple-carry on the small-size inputs, because the tested vector widths are not wide enough to amortize the extra work. On CUDA, each + deterministically expands to five kernel launches. Figure 4 puts the `ADD` share at 13.6%, 6.4%, and 7.6% of kernel time for Mison, Regex medium, and Regex large. A production backend would choose among ripple, prefix, inline XOR-shift expansion, and fusion-aware specialized code. The format itself stays neutral on the choice.

Popcount. Popcount is the other scan/reduction operator, reducing many chunks to one scalar. Its GPU lowering is also multi-kernel, a per-word popcount kernel followed by a CUB reduction. In one profiling run, Epistasis spends 28.2% of its kernel time on four popcount calls.

C. Rewrite Impact

We next examine whether existing rewrite systems can optimize BitstreamZoo programs. ABC [30] is Berkeley’s logic-synthesis tool for rewriting `AND`-inverter graphs to reduce gate count and depth. BitGen [17] is a compiler for multi-pattern regex matching on GPUs. It uses a `SHIFT-AND` commutation rule to reduce synchronization in interleaved GPU execution.

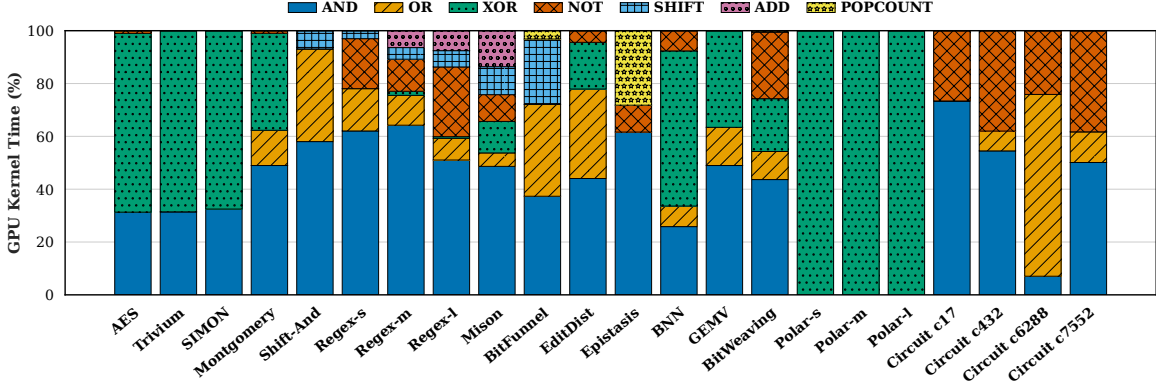


Fig. 4: GPU kernel-time breakdown by operation type from one Nsight Systems run at the medium input size, except Regex large, which uses the small input size because its medium-size working set exceeds device memory. Each bar sums to 100% of kernel time.

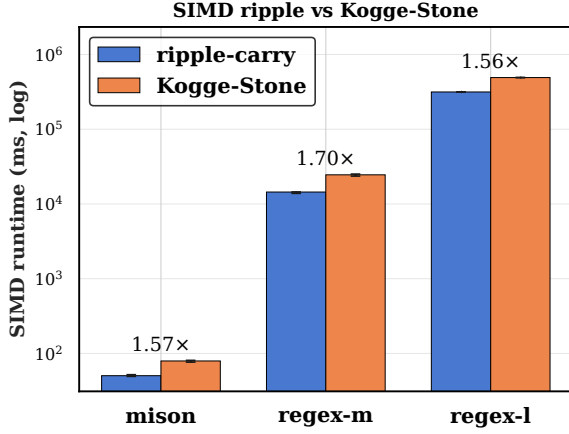


Fig. 5: SIMD ripple-carry vs Kogge-Stone parallel prefix on the three programs that use bitstream + (Mison, Regex medium, Regex large), small input size, median of five trials except Regex large (one trial). The CUDA five-kernel expansion cost is reported in the text.

These systems target different subsets of the bitstream operator space. We explore two questions: First, which programs are covered by each rewrite? Second, when a rewrite can be applied, does it reduce the dynamic cost observed by our reference backends?

Methodology. For ABC, we translate each program into a single-bit BLIF netlist using a one-bit-per-stream abstraction. Each bitstream variable becomes one BLIF wire, and each bitwise Boolean operation (AND, OR, XOR, and NOT) becomes one BLIF gate. We then run ABC’s standard `resyn2` script, which applies balancing, rewriting, and refactoring over the And-Inverter Graph (AIG).

A rewrite that is correct on the one-bit model stays correct on full streams only when the unrolled IR contains bitwise Boolean operations alone. Stream shifts, bitstream ADD, and `popcount` carry position-dependent or whole-stream information that

the one-bit abstraction discards. A rewrite that crosses such operators may therefore change program semantics. We restrict the static gate-count experiment to the 14 programs whose unrolled IR contains only AND, OR, XOR, and NOT.

After ABC optimization, we emit the optimized AIG back into an executable bitstream program, using one bitstream operation per emitted gate. We rerun the emitted program on the same inputs and check its outputs against the original on both backends. The emit-back completes and passes this equivalence check for 13 of the 14 eligible programs, which are the rows of Table VI. The rewritten BNN is too costly to rerun, so it has no verified emit-back row.

For BitGen, we evaluate applicability rather than runtime. We match BitGen’s shift-rebalancing rule,

$$(A \gg n) \& B \equiv (A \& (B \ll n)) \gg n,$$

against each program’s operation list and count the number of matches. We do not apply the rewrite in our interpreter, because the rule is designed to reduce synchronization in BitGen’s interleaved CUDA execution. That benefit is not captured by the operation count of our sequential reference backend.

Finding 6: Existing rewrites help only part of the suite. Table V reports the coverage of the BitGen rule. It matches Shift-And and the three Regex variants, which is expected because these programs contain the SHIFT-AND pattern targeted by BitGen.

ABC has broader coverage over Boolean-only programs, but its impact is uneven. It reduces the static gate count on 9 of the 14 eligible programs, including large reductions on BNN, GEMV, and Montgomery. Specifically, ABC removes 394,396 gates on BNN, 3,584 on GEMV, and 678 on Montgomery.

Table VI reports the dynamic results after emitting the optimized AIG back into bitstream form. The operation count decreases on the circuit netlists, but increases on several XOR-heavy programs. For example, Polar large increases from 5,120 to 7,796 operations, SIMON from 2,048 to 3,040, and Montgomery from 2,437 to 4,700. This is because ABC optimizes an AND-inverter representation, which does not

TABLE V: BitGen SHIFT-AND commutation opportunities, counted in the program text with no runtime change measured (§V-C). Occurrences matching the rule $(A \gg n) \& B \equiv (A \& (B \ll n)) \gg n$, reported only for the programs with a match.

Program	Shifts	Shift-then-AND occurrences	Rewritable occurrences	Rewritable fraction
Regex (small)	20	14	4	0.29
Regex (medium)	5,770	5,334	4,113	0.77
Regex (large)	83,602	77,707	73,923	0.95
Shift-And	6	6	5	0.83

TABLE VI: Executable ABC `resyn2` emit-back, small input size. The netlist is rewritten, lowered back to executable form, and rerun on the same inputs. Every row’s rewritten output matches the original on both backends. The dynamic-op columns are deterministic. CPU speedup is one timed run of the untuned backend, read as direction. BNN is omitted because emit-back is too costly to rerun.

Program	Dyn. ops before	Dyn. ops after	CPU speedup
AES S-box	117	192	0.69×
Circuit (c17)	12	8	1.30×
Circuit (c432)	440	229	1.83×
Circuit (c6288)	4,544	3,411	1.35×
Circuit (c7552)	5,124	1,955	2.40×
Edit Distance	928	830	0.31×
GEMV	13,312	16,224	0.11×
Montgomery Mul	2,437	4,700	0.08×
Polar (small)	192	291	0.44×
Polar (medium)	1,024	1,572	0.38×
Polar (large)	5,120	7,796	0.33×
SIMON	2,048	3,040	0.33×
Trivium	14,720	20,297	2.43×

have native XOR. When the optimized AIG is emitted back into bitstream operations, each XOR may expand into several gates. As a result, a smaller AIG does not necessarily become a smaller or faster bitstream program.

Similarly, at runtime, CPU time improves only on the four circuit netlists and on Trivium, or 5 of the 13 emit-back programs. The Trivium speedup comes from flattening rather than from the Boolean rewrite, because its original form interprets 285 state-copy assignments per clock and the flat emit-back removes them. These results show that existing rewrite systems are useful, but their benefits are limited by operator coverage and by the cost model they optimize. Broader optimization across BitstreamZoo requires rewrites that are aware of bitstream backend costs such as emitted operation count and memory traffic.

D. Low-Precision Gate-Decomposed Arithmetic

We conclude the dynamic characterization with a case study on GEMV. GEMV stresses gate-decomposed arithmetic, defined in Section III. The program expands each K -bit multiply into a schoolbook network of half- and full-adder gates, at a cost of $6K^2$ bitwise operations per multiply. We measure when this form is competitive on a commodity CPU core, using only standard AVX-512 instructions.

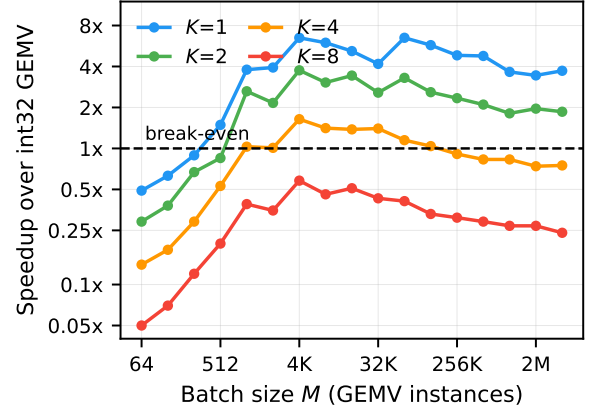


Fig. 6: Gate-decomposed GEMV speedup over a naive int32 GEMV that GCC auto-vectorizes to AVX-512, on one pinned core. The ratio is the best of at least five repetitions after warmup, transposition excluded. Each multiply costs $6K^2$ bitwise operations, so the form wins only at low precision with large batches.

Methodology. We use one 16×16 weight matrix and feed it a batch of M 16-element input vectors. Each vector is one GEMV instance, and every element is a K -bit integer. We sweep the precision K over $\{1, 2, 4, 8\}$ and the batch size M from 2^6 to 2^{22} . We time two implementations on one pinned core of the CPU platform above. Our baseline is a naive int32 triple loop over all M instances, which GCC auto-vectorizes to AVX-512 at `-O3 -march=native`. Our gate-decomposed implementation packs the M instances into bit-plane lanes, one instance per stream position. We do not count the time to transpose inputs into bit-plane form. We report the best of five repetitions after warmup, with fifteen repetitions for the microsecond-scale cells at $M \leq 65,536$. We verify the gate-decomposed output element-wise against the integer reference.

Finding 7: Larger batches favor gate-decomposed arithmetic. Figure 6 shows speedup over the baseline as the batch size M grows. At $K=1$ the gate-decomposed form breaks even between $M=256$ and $M=512$ and peaks at $6.5\times$. At $K=2$ it breaks even between $M=512$ and $M=1,024$ and peaks at $3.8\times$. At $K=4$ it stays between $0.7\times$ and $1.6\times$ for batches of 1,024 and larger, and at $K=8$ it never exceeds $0.6\times$. The shape follows the cost model, because one bitwise operation serves every lane of a vector word while the gate count grows as $6K^2$. Batches below one vector width lose at every precision, because most lanes of each word are padding. A separate sweep over matrix sizes shows gate decomposition doing better on smaller matrices and worse on larger ones. Longer dot products vectorize better in the int32 baseline, while the gate-decomposed accumulator keeps widening. Since neither implementation is hand-tuned and the runs are single-core, we only read the trends.

VI. RELATED WORK

BitstreamZoo is a workload set, assembled to understand how bitstream programs behave. We relate it to prior bench-

mark suites, bitsliced compilers, PIM systems, bitstream text-processing systems, and packed-bit software and ISA support.

Benchmark suites. Rodinia [11], Parboil [40], and SHOC [12] characterize CPU and GPU applications across many application areas. Their workloads, however, are mostly floating-point and integer programs. They do not expose the packed bitstream dataflow, cross-position carry propagation in bitstream addition, or long-lived bit-level intermediate streams that BitstreamZoo targets. ANMLZoo [45] and AutomataZoo [46] benchmark automata-processing workloads and engines. They are closest to BitstreamZoo’s regex inputs. Their focus is the automaton workload and execution engine, while BitstreamZoo measures the lowered packed-bitstream dataflow that appears after bit-parallel execution is made explicit. BitBench [13] and BitSAD [14] target stochastic and deterministic bitstream computing. Despite the similar terminology, these systems represent values over time using bit sequences. BitstreamZoo instead represents many stream positions packed into machine words and studies the resulting word-parallel dataflow.

Bitsliced compilers and PIM. Usuba [28] compiles constant-time ciphers to bitsliced SIMD C. BitstreamZoo is complementary: it provides a lower-level analyzable format, includes operators such as bitstream `+` and `popcount`, and covers non-cryptographic workload families as well. PIMbench [39] evaluates simulated processing-in-memory systems. PIM compilers such as CHOPPER [36], PIMLC [42], and PIMsynth [19] lower bitsliced code, Boolean functions, or Verilog circuits to bit-serial instructions for DRAM, SRAM, or ReRAM PIM substrates such as Ambit [38] and SIMDRAM [20]. This line of work motivates including a gate-level adder formulation in GEMV. BitstreamZoo studies packed bitstream programs running on real CPU and GPU hardware.

Bitstream text processing. Parabix [10], [27] introduced the unbounded-bitstream abstraction for text and regex processing, along with a bitstream addition primitive [8]. A related line of JSON parsers [21], [22], [24], [25], [41], [44] uses bit-parallel structural indexes in the Mison and simdjson lineage. These systems share the broad idea of using packed bits to expose text structure. Classical bit-parallel algorithms also provide important foundations. Shift-And [3] represents regex state sets as packed bits, and Myers’ edit-distance algorithm [32] uses word-level bit vectors to update many dynamic-programming states at once. BitGen [17] is closest to our regex workloads on GPUs. It is a CUDA code generation pipeline that observes poor reuse and heavy memory traffic in straightforward bitstream regex execution, then addresses the problem with fusion-based interleaved execution. BitstreamZoo collects regex among 15 workload families and measures them with one multi-backend baseline.

Packed-bit software and ISA support. Software bitslicing was popularized for cryptographic implementations [5]. Related packed-bit techniques appear in binarized neural-network XNOR-popcount datapaths [43] and bitsliced database scans [26]. These systems use the same basic hardware fact that BitstreamZoo exposes: a machine word can carry many

independent stream positions, but the resulting program often pays in intermediate streams, memory traffic, and nontrivial reductions. Specialized ISA and hardware-support proposals make this packed-bit style more explicit. IDISA [9] targets SIMD support for bitstream text processing, and the RISC-V bitslicing extension [23] targets bitsliced cryptographic computation. BitstreamZoo provides workloads and measurements that expose where such mechanisms, or backend transformations such as fusion and operator-specific lowering, may matter across the workload families in Section III.

VII. CONCLUSION

BitstreamZoo provides a shared bitstream program format for studying bitwise-parallel workloads across application areas, together with reference CPU and GPU backends. The suite shows that these workloads share a compact operation model but differ widely in dependency structure, available parallelism, and live-stream pressure. Per-operation execution exposes a common bottleneck: every intermediate stream is materialized, inflating memory traffic and leaving the GPU underutilized. Fusion recovers substantial performance on pointwise Boolean workloads, while shifts, carry-propagating addition, and reductions expose backend policy choices beyond local fusion. BitstreamZoo gives compiler and architecture researchers a reproducible workload set for bitstream programs on commodity CPUs and GPUs. The suite and its reference backends are available at <https://github.com/bigwater/bitstreamzoo>.

ACKNOWLEDGEMENTS

The authors used generative AI tools as an assistant during this work for text editing and code generation. All AI-generated content was reviewed by the authors.

REFERENCES

- [1] 3GPP, “NR; Multiplexing and Channel Coding,” 3rd Generation Partnership Project (3GPP), Technical Specification TS 38.212, 2024.
- [2] E. Arikan, “Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels,” *IEEE Transactions on Information Theory*, vol. 55, no. 7, pp. 3051–3073, 2009.
- [3] R. A. Baeza-Yates and G. H. Gonnet, “A new approach to text searching,” *Communications of the ACM*, vol. 35, no. 10, pp. 74–82, 1992.
- [4] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, “The SIMON and SPECK families of lightweight block ciphers,” *Cryptology ePrint Archive*, Paper 2013/404, 2013. [Online]. Available: <https://eprint.iacr.org/2013/404>
- [5] E. Biham, “A fast new DES implementation in software,” in *Proceedings of the 4th International Workshop on Fast Software Encryption (FSE)*, ser. Lecture Notes in Computer Science, vol. 1267. Springer, 1997, pp. 260–272.
- [6] J. Boyar, P. Matthews, and R. Peralta, “Logic minimization techniques with applications to cryptology,” *Journal of Cryptology*, vol. 26, no. 2, pp. 280–312, 2013.
- [7] F. Brglez and H. Fujiwara, “A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN,” in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 1985, pp. 663–698, special Session on Recent Algorithms for Gate-Level ATPG with Fault Simulation and Their Performance Assessment.
- [8] R. D. Cameron, E. Amiri, K. S. Herdy, D. Lin, T. C. Shermer, and F. Popowich, “Parallel scanning with bitstream addition: An XML case study,” in *Proceedings of the 17th International Conference on Parallel Processing (Euro-Par)*, ser. Lecture Notes in Computer Science, vol. 6853. Springer, 2011, pp. 2–13.

- [9] R. D. Cameron and D. Lin, "Architectural support for SWAR text processing with parallel bit streams: The inductive doubling principle," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 337–348.
- [10] R. D. Cameron, T. C. Shermer, A. Shriraman, K. S. Herdy, D. Lin, B. R. Hull, and M. Lin, "Bitwise data parallelism in regular expression matching," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014, pp. 139–150.
- [11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.
- [12] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010, pp. 63–74.
- [13] K. Daruwalla, H. Zhuo, C. Schulz, and M. H. Lipasti, "BitBench: A benchmark for bitstream computing," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2019, pp. 177–187.
- [14] K. Daruwalla, H. Zhuo, R. Shukla, and M. H. Lipasti, "BitSAD v2: Compiler optimization and analysis for bitstream computing," *ACM Transactions on Architecture and Code Optimization*, vol. 16, no. 4, pp. 43:1–43:25, 2019.
- [15] C. De Cannière and B. Preneel, "Trivium," in *New Stream Cipher Designs*, ser. Lecture Notes in Computer Science. Springer, 2008, vol. 4986, pp. 244–266.
- [16] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 383–396.
- [17] T. Ge, X. Chu, and H. Liu, "Interleaved bitstream execution for multi-pattern regex matching on gpus," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2025, pp. 385–400.
- [18] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He, "BitFunnel: Revisiting signatures for search," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2017, pp. 605–614.
- [19] D. Guo, M. Gholamrezaei, M. Hofmann, A. Venkat, Z. Zhang, and K. Skadron, "PIMsynth: A unified compiler framework for bit-serial processing-in-memory architectures," *IEEE Computer Architecture Letters*, vol. 24, no. 2, pp. 277–280, 2025.
- [20] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. a. D. Ferreira, N. Mansouri-Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, "SIMDRAM: An end-to-end framework for bit-serial SIMD computing in DRAM," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 329–345.
- [21] L. Jiang, J. Qiu, and Z. Zhao, "Scalable structural index construction for JSON analytics," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 694–707, 2021.
- [22] L. Jiang and Z. Zhao, "JSONSki: Streaming semi-structured data with bit-parallel fast-forwarding," in *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022, pp. 200–211.
- [23] P. Kiaei, T. Conroy, and P. Schaumont, "Architecture support for bitslicing," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 2, pp. 497–510, 2023.
- [24] G. Langdale and D. Lemire, "Parsing gigabytes of JSON per second," *The VLDB Journal*, vol. 28, no. 6, pp. 941–960, 2019.
- [25] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann, "Mison: A fast JSON parser for data analytics," *Proceedings of the VLDB Endowment*, vol. 10, no. 10, pp. 1118–1129, 2017.
- [26] Y. Li and J. M. Patel, "BitWeaving: Fast scans for main memory data processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013, pp. 289–300.
- [27] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. D. Cameron, "Parabix: Boosting the efficiency of text processing on commodity processors," in *Proceedings of the 2012 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2012, pp. 373–384.
- [28] D. Mercadier and P.-E. Dagand, "Usuba: High-throughput and constant-time ciphers, by construction," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019, pp. 157–173.
- [29] D. Merrill and NVIDIA Corporation, "CUB: Cooperative primitives for CUDA C++," Now part of NVIDIA CCCL: <https://github.com/NVIDIA/cccl>, 2024, original repository (archived 2024): <https://github.com/NVIDIA/cub>.
- [30] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *Proceedings of the 43rd Annual Design Automation Conference (DAC)*, 2006, pp. 532–535.
- [31] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.
- [32] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *Journal of the ACM*, vol. 46, no. 3, pp. 395–415, 1999.
- [33] National Institute of Standards and Technology, "Module-Lattice-Based Key-Encapsulation Mechanism Standard," Federal Information Processing Standards Publication FIPS 203, Aug. 2024.
- [34] NIST Circuit Minimization Team, "AES-SBOX-FWD-G113-A32-D27-AD6: Forward AES S-Box straight-line program (115-gate basis improved by 2 XORs)," <https://github.com/usnistgov/Circuits/blob/master/data/slp/aes/aes-sbox-fwd-g113-a32-d27-ad6.slp>, 2020, nIST Circuit Complexity project, <https://csrc.nist.gov/Projects/circuit-complexity>. Accessed: 2026-05-14.
- [35] NVIDIA Corporation, "NVIDIA RTX Blackwell PRO GPU Architecture," Whitepaper, version 1.0, 2025. [Online]. Available: <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/NVIDIA-RTX-Blackwell-PRO-GPU-Architecture-v1.0.pdf>
- [36] X. Peng, Y. Wang, and M.-C. Yang, "CHOPPER: A compiler infrastructure for programmable bit-serial SIMD processing using memory in DRAM," in *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 1275–1288.
- [37] J. Qiu, L. Jiang, and Z. Zhao, "Challenging sequential bitstream processing via principled bitwise speculation," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 607–621.
- [38] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. S. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 273–287.
- [39] F. A. Siddique, D. Guo, Z. Fan, M. Gholamrezaei, M. Baradaran, A. Ahmed, H. Abbot, K. Durrer, K. Nandagopal, E. Ermovick, K. Kiyawat, B. Gul, A. Mughrabi, A. Venkat, and K. Skadron, "Architectural modeling and benchmarking for digital DRAM PIM," in *Proceedings of the 2024 IEEE International Symposium on Workload Characterization (IISWC)*, 2024, pp. 247–261.
- [40] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, 2012.
- [41] S. Talluri, G. W. Di Donato, L. Danelutti, K. R. Vlaswinkel, M. Arnaboldi, A. Delamare, M. D. Santambrogio, and D. Bonetta, "GpJSON: High-performance JSON data processing on GPUs," *Proceedings of the VLDB Endowment*, vol. 18, no. 9, pp. 3216–3229, 2025.
- [42] C. Tang, C. Nie, W. Qian, and Z. He, "PIMLC: Logic compiler for bit-serial based PIM," in *Proceedings of the 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024, pp. 1–6.
- [43] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 65–74.
- [44] A. Vedadi Gargary, S. Safari Loaliyan, and Z. Zhao, "cuJSON: A highly parallel JSON parser for GPUs," in *Proceedings of the 31st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2026, pp. 85–100. [Online]. Available: <https://doi.org/10.1145/3760250.3762222>
- [45] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron, "ANMLZoo: A benchmark suite for exploring bottlenecks in automata processing

- engines and architectures,” in *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–12.
- [46] J. Wadden, T. Tracy II, E. Sadredini, L. Wu, C. Bo, J. Du, Y. Wei, J. Udall, M. Wallace, M. Stan, and K. Skadron, “AutomataZoo: A modern automata processing benchmark suite,” in *Proceedings of the 2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 13–24.
- [47] X. Wan, C. Yang, Q. Yang, H. Xue, X. Fan, N. L. S. Tang, and W. Yu, “BOOST: A fast approach to detecting gene-gene interactions in genome-wide case-control studies,” *American Journal of Human Genetics*, vol. 87, no. 3, pp. 325–340, 2010.
- [48] L. S. Yung, C. Yang, X. Wan, and W. Yu, “GBOOST: A GPU-based tool for detecting gene-gene interactions in genome-wide case control studies,” *Bioinformatics*, vol. 27, no. 9, pp. 1309–1310, 2011.