UDP: A Programmable Accelerator for Extract-Transform-Load Workloads and More

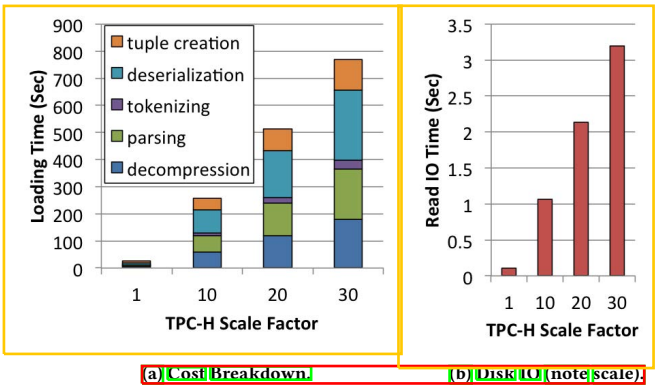
Yuanwei Fang[†] Chen Zou[†] Aaron J. Elmore[†] Andrew A. Chien[†‡]

Computer Science Department, University Of Chicago[†]
Mathematics and Computer Science Division, Argonne National Laboratory[‡]
{fywkevin,chenzou,aelmore,achien}@cs.uchicago.edu

ABSTRACT

Big data analytic applications give rise to large-scale extract-transform-load (ETL) as a fundamental step to transform new data into a native representation. ETL workloads pose significant performance challenges on conventional architectures, so we propose the design of the unstructured data processor (UDP), a software programmable accelerator that includes multi-way dispatch, variable-size symbol support, flexible-source dispatch (stream buffer and scalar registers), and memory addressing to accelerate ETL kernels both for current and novel future encoding and compression. Specifically, UDP excels at branch-intensive and symbol and pattern-oriented workloads, and can offload them from CPUs.

To evaluate UDP, we use a broad set of data processing workloads inspired by ETL, but broad enough to also apply to query execution, stream processing, and intrusion detection/monitoring. A single UDP accelerates these data processing tasks 20-fold (geometric mean, largest increase from 0.4 GB/s to 40 GB/s) and performance per watt by a geomean of 1,900-fold. UDP ASIC implementation in 28nm CMOS shows UDP logic area of $3.82mm^2$ ($8.69mm^2$ with 1MB local memory), and logic power of 0.149W (0.864W with 1MB local memory); both much smaller than a single core.



(a) Cost Breakdown. (b) Disk IO (note scale).

Figure 1: Loading compressed CSV into PostgreSQL.

ACM Reference format:
Yuanwei Fang[†] Chen Zou[†] Aaron J. Elmore[†] Andrew A. Chien[†‡]. 2017. UDP: A Programmable Accelerator for Extract-Transform-Load Workloads and More. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 14 pages.
https://doi.org/10.1145/3123939.3123983

CCS CONCEPTS

• **Information systems** ⟹ **Extraction, transformation and loading**; • **Computer systems organization** ⟹ **Parallel architectures**; • **Hardware** ⟹ **Application specific processors**; • **Theory of computation** ⟹ **Pattern matching**;

KEYWORDS

Data Encoding and Transformation, Parsing, Compression, Data Analytics, Control-flow Accelerator

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123983

1 INTRODUCTION

With the rise of the Internet, mobile applications, data driven science, and large-scale sensors, data analysis for large, messy, and diverse data (e.g. "Big Data") is an important driver of computing performance. The advent of large memory systems and scale-out aggregation to petabytes has made in-memory data analytics an increasingly important focus of computer architecture. In short, data manipulation – transformation – movement, rather than arithmetic speed, is the primary barrier to continued performance scaling.

We focus on a growing class of big data computations [14, 19, 27] that analyze diverse data for business (e-commerce, recommendation systems, targeted marketing), social networking (interest filtering, trending topics), medicine (pharmacogenomics), government (public health, event detection), and finance (stock portfolio management, high-speed trading). These applications all exploit diverse data (e.g. sensors, streaming, human-created data) that is often dirty (has errors), and in varied formats (e.g. JSON, CSV, NetCDF, compressed). Thus, ingestion and analysis can require parsing (find values), cleaning (remove and correct errors, normalize data), and validation (constraints, type checking).

In many cases, both real-time and batch analytics are required, so systems exploit highly-optimized internal formats such as columnar block compression, special encoding, and rich indexing to meet rising performance demands. These formats require costly transformations to get data into a native format [70] or just-in-time transformations to analyze in-situ [37]. For example, Figure 1a shows single-threaded costs to load all TPC-H [35] Gzip-compressed CSV files (scale factor from 1 to 30) from SSD into the PostgreSQL relational database [33] (Intel Core-i7 CPU with 250GB SATA 3.0 SSD). This common set of extract-transform-load (ETL) tasks includes decompression, parsing record delimiters, tokenizing attribute values, and deserialization (decoding specific formats and validation of domains such as dates), and consumes nearly 800 seconds for scale factor 30 (about 30GB uncompressed), dominating time to initial analysis [37]. Figure 1b shows that >99.5% wall-clock loading time is spent on CPU tasks, rather than disk IO.

Further, advances in real-time and complex batch analysis has produced diverse innovation in algorithms, data structures, representations, and frameworks both for applications and scalable data storages and analytics systems [36, 46, 50, 60, 69, 74]. These innovations often use novel, even application-specific encodings and compressions that often perform poorly on transformations on traditional CPUs.

To address these challenges, we propose a flexible, programmable engine, the unstructured data processor (UDP), to accelerate current encodings and transformations and enable new algorithms and even application-specific techniques. UDP runs programs that exploit UDP architecture features such as multi-way dispatch and dispatch from stream and scalar sources to efficiently implement branch-intensive computation. UDP includes special support for variable-size symbols, supporting very-dense bit-packed representations. 64 lanes scale up UDP performance, exploiting data parallelism available in most data transformations. Flexible addressing enables these lanes (small micro-architectures) to flexibly use memory. Together the lane efficiency and parallelism enable UDP to achieve both high performance and energy efficiency. UDP's programmable approach differs from narrow accelerators that "freeze in silicon" for particular algorithms, representations or data structures [59, 68, 82] (as shown in Table 1).

Our results using varied data transformation benchmarks reveal that in many cases, the UDP can outperform a CPU at less than 1/100th the power, providing an effective offload, that both eliminates much of the cost of data transformation, and frees the CPU for other activities.

Specific contributions of the paper include:

- Micro-architecture of the unstructured data processor (UDP), including the description of key features of multi-way dispatch, variable-size symbols, flexible-source dispatch, and an addressing architecture for efficient, flexible UDP lane-bank coupling. Quantitative comparison for each and documenting their effectiveness.
- Performance evaluation for UDP on diverse workloads showing speedups from 1.1x on CSV parsing, 69x on Huffman encoding, 197x on Huffman decoding, 19x on pattern matching, 48x on dictionary encoding, 45x on dictionary-RLE encoding, 9.5x on histogramming, 3x on compression, 3.5x on decompression, and 21x on triggering, compared to an 8-thread CPU. Geometric mean of performance gives 20-fold improvement, and 1,900-fold performance per watt over an 8-thread CPU. For many of these workloads, acceleration of branches (multi-way) dispatch is the key.
- Power and area evaluation for the UDP implementation (28nm, ASIC) that achieves 1 GHz clock, in 8.69 mm^2 at 864 milliwatts, making UDP viable for CPU offload, or even incorporation in a memory/flash controller or network-interface card.

Collectively, these results show the promise of the UDP for ETL and more general data transformation workloads.

The UDP is a part of the 10x10 project [43, 47, 48, 51, 76–78], an exploration of heterogeneous architecture that federates customized micro-engines to achieve energy efficiency and general-purpose performance. We found that half of the 10x10 micro-engines focused on data-oriented acceleration [48, 51, 76] and converged their capabilities in the Unified Automata Processor (UAP [54]) that achieved efficient automata processing (including regular expression matching). These workloads are challenging on traditional CPUs because they are indirection-intensive, branch-intensive, and operating on short, variable-size data. The UDP builds on insights from the UAP, but supports general data transformation.

The remainder of the paper is organized as follows. In Section 2, we describe challenging data transformation workloads, and compare UDP's flexible programmability to narrower accelerator designs. Next, we describe our novel architecture of the Unstructured Data Processor (UDP), including its key architectural features (Section 3). For each, we describe and evaluate several options, substantiating UDP's architectural choices. Next, we study the UDP performance on a variety of data transformation benchmarks in Section 5, using cycle-accurate simulation calibrated by a detailed ASIC design described in Section 6. We conclude with a qualitative discussion and a quantitative comparison of our results to the research literature (Section 7), and a summary (Section 8).

2 DATA TRANSFORMATION WORKLOADS AND ACCELERATORS

Big data computing workloads are diverse and typified by challenging behavior that gives poor performance in modern CPUs with high instruction-level parallelism and deep pipelines [11, 79]. We summarize a variety of these workloads below and document their challenges for CPUs in Table 2.

2.1 Workloads

Database ETL (extract, transform, and load) requires tools to integrate disparate data sources into a common data system or format, such as a column oriented database or shared analytic formats for dataflow systems. An extract phase can combine data from different source systems, which may store data in different formats (XML, JSON, flat files, binary files, etc). Transform involves a series of rules or functions that are applied to the extracted data in order to prepare it for loading into the end target. This includes both logical transformations to normalize and clean data, as well as physical transformations to prepare the data for the destination

		Compression / Decompression (DEFLATE, Snappy, Xpress, LZF, ...)	Encoding / Decoding (RLE, Huffman, Dictionary, Bit-pack, ...)	Parsing (CSV, JSON, XML, ...)	Pattern Matching (DFA, D2FA, NFA, c-NFA, ...)	Histogram (Fixed-size bin, Variable-size bin, ...)
UDP		All listed	All listed	All listed	All listed	All listed
UAP [54]		None	None	None	All listed	None
Intel Chipset 89xx [30]		DEFLATE	None	None	None	None
Microsoft Xpress (FPGA)[56]		Xpress	None	None	None	None
Oracle Sparc M7 [68]	DAX-RLE	None	RLE	None	None	None
	DAX-Huff	None	Huffman	None	None	None
	DAX-Pack	None	Bit-pack	None	None	None
	DAX-Ozip	None	OZIP	None	None	None
IBM PowerEN [61]	XML	None	None	XML	None	None
	RegX	None	None	None	DFA, D2FA	None
	Compress	DEFLATE	None	None	None	None
Cadence Xtensa [1]	Histogram TIE	None	None	None	None	Fixed-size bin
ETH Histogram (FPGA)[63]		None	None	None	None	All listed

Table 1: Coverage of Transformation/Encoding Algorithms: Accelerators and UDP.

formats. These tasks are critical for ensuring data consistency and to enable efficient data representations that optimize execution engines use. Figure 1 shows for compressed CSV files loading, ETL tasks exceed IO cost by 200x. For our experiments, ETL includes CSV parsing, format-specific encoding/decoding (e.g., dictionary-rle), de/compression, Huffman coding, and pattern matching (see Section 5).

Query Execution is critical for columnar analytic and transactional in-memory databases which often compress and encode data. Inlining decompression and query functions help to improve performance and remove the memory bandwidth bottleneck. Columnar databases encode attributes to reduce storage footprint and allow for query predicates to be pushed down directly on encoded data. Query execution often includes in-memory format conversion and value (or range) comparison for predicate based filtering. Additionally, to improve query planning (such as join ordering), databases rely on histograms to estimate data distributions. These critical classes of functions involve de/compression, Huffman decoding, histogram generation, and pattern matching.

Stream Processing includes streaming databases, streaming monitoring, sentiment analysis, real-time sensors, embedded-systems, and data capture from scientific instruments. With the growing use of real-time stream processing, acceleration for parsing, histogramming, pattern matching, and signal triggering is essential for efficient high-level analysis.

Network Intrusion Detection/Deep Packet Inspection have growing usage in networking. Current intrusion detection systems search for dangerous (malicious code) or interesting content in network packets by matching patterns. Network packets are encrypted and/or compressed to meet growing bandwidth and security goals. Important computations include decompression, parsing, pattern matching, multi-level packet inspection, and signal triggering.

2.2 Accelerating Data Transformation with Flexibility

To document the benefits of UDP's programmability, we compare its applicability to a variety of accelerators across categories and within a category (e.g., compression) in Table 1. **1)** Compression accelerators are typically hardwired, supporting only a single algorithm and format. In contrast, the UDP supports a variety at high performance and can be programmed to support new or application-specific algorithms. **2)** Other encoding accelerators support a single type, while UDP supports RLE, Huffman, dictionary, and bit-pack efficiently; and can be programmed to support more. **3)** For parsing, UDP supports formats as diverse as CSV, JSON, and XML with general-purpose primitives, whilst other accelerators focus on a single class of formats. **4)** For pattern matching, UDP supports a full diversity of extended finite-automata (FA) models, adopting key features to achieve this from the Unified Automata Processor [54], while other accelerators support one or two. The UDP's flexibility enables applications to choose the best FAs for application, achieving both small FA size and high stream rate. **5)** For histogram, other accelerators support a few widely-used models; the UDP supports these, and can be programmed to support new types. key new features of UDP include variable-size symbol, flexible-source dispatch, and flexible addressing (discussed in Section 3). For all of the algorithms, UDP achieves competitive performance and efficiency (see Table 4), while providing programmability for breadth and extensibility.

3 ARCHITECTURE OF THE UDP

3.1 Overview

We propose an accelerator, the unstructured data processor (UDP) to offload extract-transform-load and data transformation programs from CPUs (see Figure 2).[1] Thus, the UDP is designed to deliver equal or higher performance at dramatically lower power, and to

[1] UDP's low power (see Section 6) also enables different models of incorporation (as a core addition on in a flash or DRAM controller), but we focus on just one scenario here.

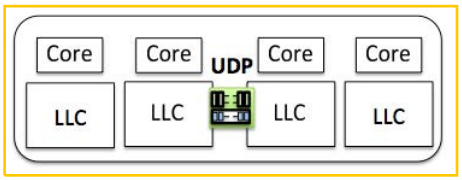


Figure 2: UDP integrated as an offload engine.

be programmable so as to support a wide and expanding variety of encoding and transformation programs.

Traditional CPUs are designed for predictable control flow, large chunks of computation, and computing on machine-standard data types. For encoding and transformation tasks, these philosophies are often invalid (see Table 2 and Figure 1a) producing poor performance and efficiency. The UDP has 64-parallel lanes (Figure 3a), each designed for efficient encoding and transcoding. Parallel lanes exploit the data parallelism often found in encoding and transformation tasks, and the lane architecture includes support for branch-intensive codes, computation on small and variable-sized application-data encodings, and programmability.

Key UDP Architecture Design Features include UDP lane architecture support for

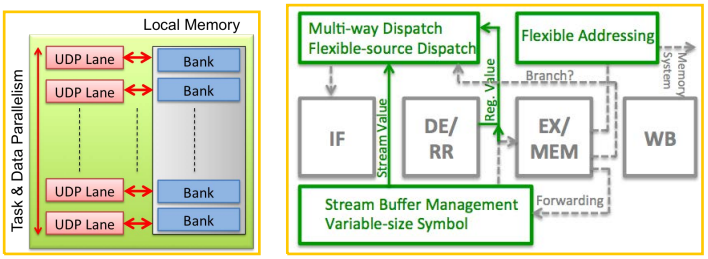
- multi-way dispatch,
- variable-size symbols, and
- dispatch from varied sources.

At the UDP and system level architecture level:

- flexible memory addressing (vary memory/lane), and
- multi-bank local memory for high bandwidth, predictable latency, and low power.

The UDP lane ISA [52] contains 7 transition types implementing the multi-way dispatch and 50 actions including arithmetic, logical, loop-comparing, configuration and memory operations to form general code blocks supporting a broad set of data transformation kernels. Each lane contains 16 general-purpose scalar data registers and a stream buffer equipped with automatic indexing management and stream prefetching logic. Register 15 stores the stream buffer index. Each lane has its own UDP program. Each unstructured data processor (UDP) includes 64 such lanes, a shared 64x2048-bit vector register file, and a multi-bank local memory as shown in Figure 3a. The local memory provides an aggregate of read/write memory bandwidth of 512GB/s with predictable latency, enabling high-speed data transformation to be overlapped with staging of data to local memory.

The UDP's novel architecture features affect the micro-architecture as shown in Figure 3b. Additional front-end functionality supports multi-way dispatch and flexible-source dispatch, requiring connection to the register file and stream buffer. Additional forwarding functionality supports both a stream buffer and its management, as well as special architecture features for variable-size symbols. Finally, the memory unit is enhanced to support UDP's window-based addressing. The UDP also employs a multi-bank local memory to provide ample bandwidth and predictable low latency and energy. The software stack used generate UDP programs is discussed in Section 4.3.



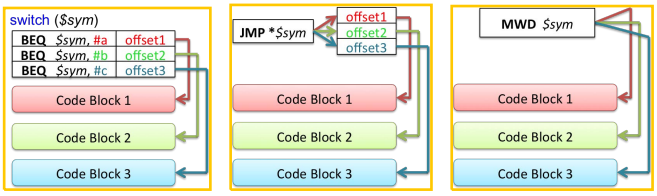
(a) 64-lane UDP. (b) UDP lane adds three elements (green) to a traditional CPU micro-architecture.

Figure 3: UDP and UDP Lane Micro-architecture.

3.2 UDP Lane: Fast Symbol and Branch Processing

The UDP's 64 lanes each accelerate symbol-oriented conditional processing. The four most important UDP lane capabilities include: 1) multi-way dispatch, 2) variable-size symbol support, 3) flexible dispatch sources for accelerated stream processing, and 4) flexible memory addressing. We consider each in turn, describing the key design alternatives and giving rationale and quantitative evidence for our choices.

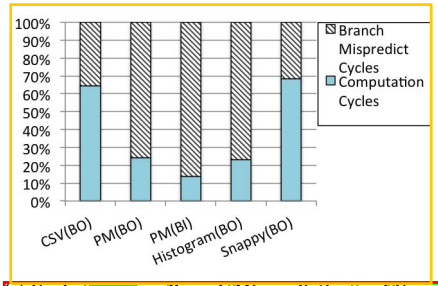
3.2.1 Multi-way Dispatch. Fast multi-way dispatch using input streaming symbols is a long-standing application challenge. Todays fastest CPU implementations either use *branch-with-offset* (BO) in a *switch()* structure that employs a sequence of compares and BO's (Figure 4a), or compute an entry in a dispatch table full of targets, and then *branch-indirect* (BI) to that target (Figure 4b). In the former approach, the static offset in each branch enables decoupled code layout, but the large number of branches and significant misprediction rates hamper performance. In the latter, the BI operation often suffers BTB (branch-target-buffer) misses, hampering performance as well.



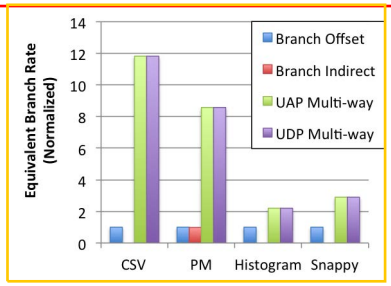
(a) BO (b) BI (c) MWD

Figure 4: Branch Execution on Symbol: Branch Offset (BO), Branch Indirect (BI), Multi-way Dispatch (MWD).

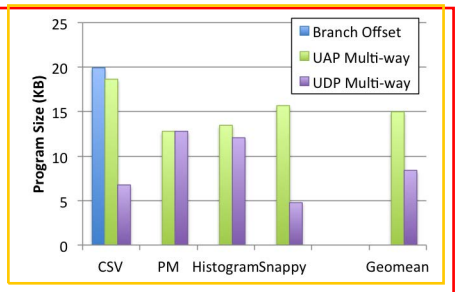
To illustrate the problem, we studied several ETL kernels drawn from the larger set described in Table 2. We measured the fraction of execution cycles consumed by branch misprediction, considering two different software approaches for these kernels on a traditional CPU — *branch-with-offset* (BO) that use static targets, and *branch-indirect* (BI) that uses a computed target. The kernels, with either approach, all suffer from branch misprediction penalties, consuming 32% to 86% of execution cycles (Figure 5a). These penalties are typical of many ETL and data transformation workloads, as documented in Table 2.



(a) Cycles spent on Branch Misprediction and Computation.



(b) Relative Branch Rate for ETL kernels using BO, BI and Multi-way.



(c) Code size for varied branch and dispatch approaches.[2]

Figure 5: Evaluating Branch-offset (BO), Branch-indirect (BI), and Multi-way Dispatch (UAP/UDP) on ETL Kernels.

The UDP's multi-way dispatch (see Figure 4c) selects efficiently from multiple targets by using the symbol (or any value) as a dynamic offset. Compared to BO, multi-way dispatch can process several branches in a single dispatch operation, and avoids explicit encoding of many offsets. Compared to BI, multi-way dispatch avoids an explicit table of branch targets, producing placement coupling challenges discussed below. As a result, multi-way has much smaller code size than both BI and BO. Also, compared to both, multi-way dispatch shuns prediction, depending on a short pipeline for good performance.

While all compilers must deal with limited range offsets, the UDP software stack (see Section 4.3) must deal with a harder problem – precise relative location constraints due to multi-way dispatch. The UDP stack converts UDP assembly to machine code (representation shown in Figure 6), and creates an optimized memory layout using the Efficient Coupled Linear Packing (EffCLiP) algorithm [55] that resolves the coupled code block placement constraints. A great help in this is UDP's *signature* mechanism (see below) that effectively allows gaps in the target range of dispatch to be filled with actual targets from other dispatches. Thus, together EffCLiP and UDP achieve dense memory utilization and a simple, fixed hash function – integer addition. This enables a high clock rate and energy efficient execution. In effect, EffCLiP achieves a "perfect hash" for a given set of code blocks. The UDP assembler back-propagates transition type information along dispatch arcs, and then generates machine binaries using machine-level transitions and actions.

UDP machine encodings are summarized in Figure 6. For transitions, *signature* is used to determine if a valid transition was found. *target* specifies the next state, and is combined with a symbol to find the target's address. *type* specifies the type of the outgoing transition and the usage of the *attach* field (either an auxiliary value of the target state's property or addressing actions). The use of *attach* varies by scenario to maximize addressing range. Three action types are used including Imm Action, Imm2 Action, and Reg Action. *opcode* specifies the action type. The actions associated with a transition are chained as a list with the end denoted by *last*. The three action types differ in the number of register operands and immediate fields, balancing performance and generality.

[2] For Histogram, results add variable-width symbols to UAP to make its code small enough to make comparison possible.

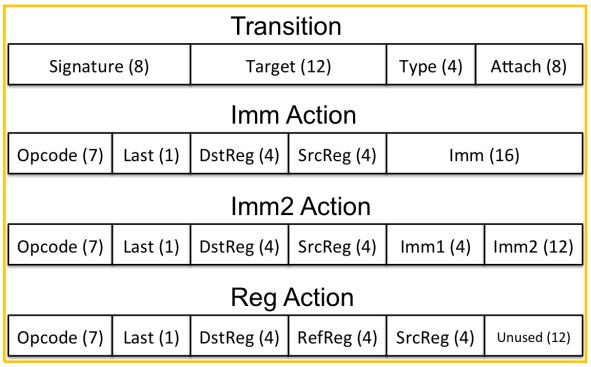


Figure 6: UDP Transition and Action Formats: Imm Action, Imm2 Action, Reg Action. All are 32-bits.

Direct comparison between multi-way dispatch and branches is difficult: one dispatch does the work of many branches. To account for this, we normalize the cycle counts of all approaches to BO, using a uniform cycle time, as the effective branch rate relative to BO. We compare this effective branch rate for several ETL applications (see Figure 5b). Our results show that UDP's multi-way dispatch achieves much higher performance. This is very challenging, as in CSV parsing, dispatch processes an arbitrary regular character or delimiter each cycle. For pattern matching, dispatch avoids all misprediction, explicitly encoding all of the character transitions, and simply selecting the right one each cycle. Overall, multi-way dispatch provides 2x to 12x speedup for these challenging benchmarks.

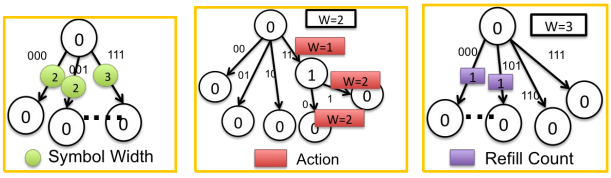
UDP's multi-way dispatch includes a significant improvement over the UAP's. Memory in accelerators is always in high-demand, and in the UDP, code size competes directly with lane parallelism, and thus performance. Both UDP and UAP use *attach* to address action blocks. To improve code reuse and program density, the UDP replaces UAP's offset addressing with two modes, direct and scaled-offset. Together, these enable both global sharing as well as private code blocks and in some ETL kernels reduce program size by more than half (see Figure 5c).

Overall, UDP employs seven transitions implementing variants of multi-way dispatch: *labeled* [54], *majority* [54], *default* [54], *epsilon* [54], *common*, *flagged*, and *refill*. They collectively achieve

generality and memory efficiency. The *labeled* transition implements a single labeled (specific symbol) transition. To reduce the number of explicitly encoded transitions, *majority* transition implements a set of outgoing transitions, representing the transitions that share the destination state from a given source state. *default* transition acts as a fallback, enabling "delta" storage for transitions that share the destination states across different source states. Each state has at most one *majority* or *default* transition with runtime overhead if *signature* check fails during multi-way dispatch. Multi-state activation is supported by *epsilon* transition. *common* transition represents 'don't care', which means no matter what symbol received, the transition is always taken. One *common* transition represents ∑. *labeled* transitions from a given source state. New transitions in UDP include *flagged* that provides control-flow driven state transfer using a UDP data register (Section 3.2.3) and *refill* that enables efficient variable-size symbol execution (Section 3.2.2).

3.2.2 *Variable-size Symbols.* Variable-length codings are essential tools for increasing information density (e.g. Huffman coding). Because these symbols can be very short, achieving high data rates for processing them requires UDP to process several symbols per dispatch (concatenating the symbols). However naive concatenation and program folding increases program size exponentially, causing layout failure and reduced parallelism.

We explore architecture support for variable symbol size that enables both high performance and good code size. We consider four designs, including the final UDP design (SsRef) that supports variable-size and sub-byte symbols efficiently. Consider the Huffman decoding tree example in Figure 7.



(a) SsT (b) SsReg (c) SsRef

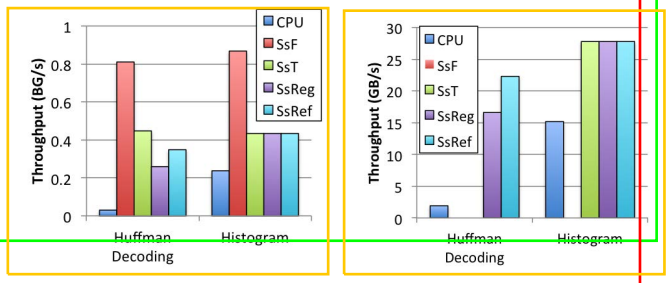
Figure 7: Huffman Decoding Tree: *00,01,10,110,111*. Solid box is symbol-size register. Other actions not shown.

(1) **Symbol-size Fixed (SsF)** hardwired dispatch width. For example, the UAP has fixed 8-bit dispatch with (character symbols), achieving best performance and efficiency for regular expression matching. Applications requiring variable-size symbols (e.g. Huffman decoding) must adapt by unrolling, causing major program size explosion.

(2) **Symbol-size per Transition (SsT)** preserves fixed dispatch width per transition, but allows each to specify its own (see Figure 7a). This enables fast execution for variable-size symbols, and the transition "puts back" excess symbol bits. Challenges include: 1) increased encoding bits (symbol size) in each transition, 2) longer hardware critical path (read transition from memory, decide symbol size, consume that number of bits).

(3) **Symbol-size Register (SsReg)** configures symbol size in a register. The UDP stream buffer prefetch unit (see Section 6) preloads the correct number of bits, taking variable size off the critical path. Avoiding specify dispatch width in each transition reduces memory overhead, but both memory and runtime are incurred by operations to change the symbol-size register (see Figure 7b).

(4) **Symbol-size Register and Refill Transitions (SsRef)** combines the benefits of *SsT* and *SsReg*. Dispatch width is stored explicitly in a symbol-size register, and UDP adds a new transition, *refill*, that refills bits that should not be consumed (see Figure 7c) based on symbol-size register via *attach* field. This hybrid approach combines fast execution with low memory overhead.



(a) Rate (1-lane) (b) Throughput (64-lanes)

Figure 8: Variable-size Symbol Approaches on kernels requiring dynamic and static variability.

In Figure 8, we compare performance for Huffman decoding (dynamic symbol-size) and Histogram (compile-time static symbol-size) for all four approaches, reporting both rate (single lane) and throughput (64-lane parallel).

UAP's 8-bit fixed symbol-size (*SsF*) requires unrolling of the Huffman decoding tree, but delivers high rate (Figure 8a). Without unrolling, *SsT*, *SsReg* and *SsRef* achieve a lower rate for both Huffman and Histogram. However, their smaller code sizes yield benefits for throughput, as code-size limits parallelism (see Figure 8b). For Huffman Decoding, UAP's code size is 508 KB. *SsT* has 5.7x smaller code size, but is limited to 4 parallelism. *SsReg* and *SsRef* enjoy full parallelism as 64 achieving higher throughput. Similar effects apply for Histogram (compile-time variable-size symbols).

3.2.3 *Flexible Dispatch Sources.* UDP can dispatch on symbols from a stream buffer or scalar data register, improving on the UAP (stream buffer only). New support for scalar register dispatch enables powerful multi-dispatch to be integrated generally into UDP programs, growing applicability to the broad range of data movement and transformation tasks described in Section 5.

Stream Buffer constructs streams from vector registers, extending the vector instruction set (e.g. AVX,NEON[3, 20]). Efficient implementation copies vector register to the UDP stream buffer, who has hardware prefetching and efficient index management support, delivering good single stream performance. Shared or private vector register coupling is supported: each lane can use private or share vector register stream.

Scalar Register enables multi-way dispatch on data (symbols) computed or drawn from arbitrary machine state, using the *flagged*

transition. This small addition expands the application space dramatically, enabling memory-based data transformation (e.g., compression), hash-based algorithms, and de/compression, Huffman encoding, CSV parsing, and Run-length encoding, than the streaming models supported by prior architectures [54]. For simplicity, the current UDP design restricts the source to Register 0.

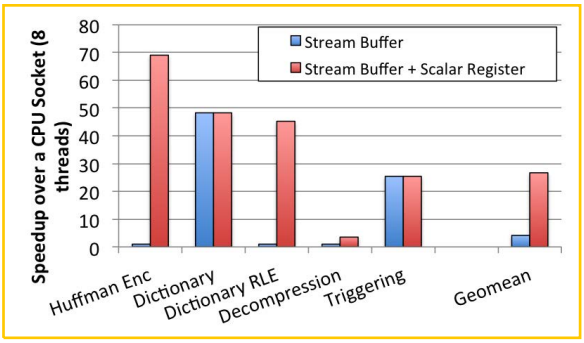


Figure 9: Performance Benefit for adding stream buffer and scalar register as UDP dispatch source.

To demonstrate the incremental performance benefit of scalar register dispatch, we present UDP's geometric mean of speedup for stream buffer only and stream buffer+scalar (see Figure 9). Compared to an 8-thread CPU (Section 5) using the rest of the ETL kernels that we have not used in the prior two architecture comparisons. Adding scalar dispatch enables coverage of a much broader application domain, dramatically improving the geometric mean speedup.

3.2.4 *Flexible Addressing for Data-Parallelism and Memory Utilization.* Flexible, programmable acceleration faces challenges in how parallelism and addressing relate to the critical resource of local memory (bandwidth, capacity, access energy). The UDP is an MIMD parallel accelerator with each lane generating memory accesses, and the 64-lanes collectively sharing a multi-bank local memory. Ideally, code generation, data-parallelism, and memory capacity are independent, but separation incurs significant memory system complexity and energy cost. We consider three scenarios, *local*, *global*, and *restricted* addressing (see Figure 10).

Each UDP lane is a 32-bit execution engine, that generates 12-bit word addresses from the *target* field for dispatch targets (Figure 6).

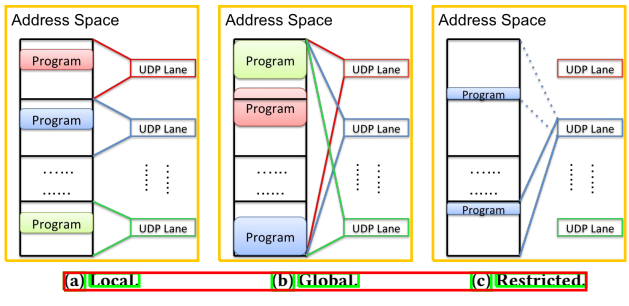


Figure 10: Addressing Models. Local: each lane has private address space; Global: each lane shares entire address space; Restricted: each lane flexibly chooses a window.

In addition, the UDP programs (actions) can generate 32-bit byte addresses.

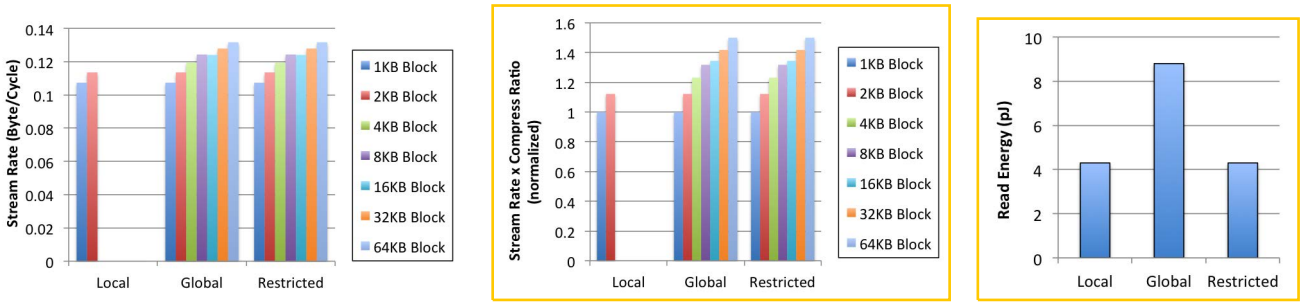
Local Addressing Each lane generates addresses confined to a single memory bank (16KB, 1/64th of the entire UDP memory). Code generation and execution for each of the 64 lanes has no dependence, and no hardware sharing of memory banks is needed. The UAP adopts this simple approach to achieve high performance. The primary drawback of local addressing is application memory flexibility, limited memory per program, and no means to vary lane parallelism. For example, if four memory banks (64KB) are needed to match natural application data size, there is no way to run with 16 lanes with 64KB memory for each lane. Snappy compression performance improves with block size, so this can be important (Figure 11a). Figure 11b shows the combined benefit for both performance (rate) and compression ratio, where the net benefit can differ as much as 50%.

Global Addressing maximizes software flexibility, "ensure there are enough address bits" [42] by allowing each UDP lane to address the entire UDP memory (18-bit word address for 1MB), increasing the *target* field, program size, and data path. This incurs both area and power overhead, but also a software problem. Code generation for each lane in a globally addressed system is complicated by UDP's absolute addressing, requiring customized loading based on lane ID, the number of active lanes, and memory partition. Alternatives based on including virtual memory or other translation incur additional energy and performance costs.

Restricted Addressing is a hybrid scheme. *Restricted addressing* adds a base register to each UDP lane. This base allows code generation similar to that with local addressing. To shift the addressable window, the UDP lane changes its base register value under software control. With compiler support, a UDP lane can access full local memory address.

Once UDP lanes can concurrently address the same memory location (global or flexible), memory consistency issues arise. UDP lane programs are all generated by a single compiler (no multiprogramming) and operate nearly synchronously, so lane interaction can be managed and minimized in software. The UDP memory consistency model is simple; it "detects and stalls" conflicting references, ensuring that both complete, but in an unspecified order. Thus, no complicated shared memory implementations are needed, and simple arbitration is used. Thus, the UDP enjoys fast local memory access and low access energy. Figure 11c displays memory reference energy for 1MB memory (64 read ports and 64 write ports) modeled using CACTI 6.5 [6]. For local and restricted addressing, 1MB memory has 64 independent banks with 1 read and 1 write port for each 16KB bank. Restricted and local addressing requires 4.3 pJ/ref while global addressing requires over double, 8.8 pJ/ref.

3.2.5 *Other Key Features.* To reduce the instruction count, UDP provides customized actions beyond basic arithmetic, logical and memory operations. *hash* action provides fast hashes of the input symbol. *loop-compare* action compares two streams, returning the matching length. *loop-copy* action copies a stream on memory block. These actions accelerate compression and a range of parsing and data transformation. The *goto* action enables reuse of code blocks, increasing code density.



(a) Addressing Impacts Performance. (b) Addressing Impacts More Than Performance. (c) Memory Reference Energy.

Figure 11: Comparing Three Addressing Modes: Local, Global, Restricted.

4 METHODOLOGY

4.1 Workloads

We selected a diverse set of kernels drawn from broader ETL and data transformations.

Application	Workload	CPU Challenge
CSV Parsing	Crimes, NYC Taxi Trip [23], Food Inspection [17]	3x branch mispredicts
Huffman Encoding	Canterbury Corpus, Berkeley Big Data	5x branch mispredicts
Huffman Decoding	Canterbury Corpus, Berkeley Big Data	5x branch mispredicts
Pattern Matching (Intrusion Detection)	IBM PowerEN dataset [80]	Poor locality, 1.6x L1 miss rate
Dictionary	Crimes [16]	Costly Hash 67% runtime
Dictionary and Run Length Encoding (RLE)	Crimes	Costly Hash 54% runtime
Histogram	Crimes, NYC Taxi Trip	5x branch mispredicts
Compression (Snappy)	Canterbury Corpus [5], Berkeley Big Data [24]	15x branch mispredicts
Decompression (Snappy)	Canterbury Corpus, Berkeley Big Data	15x branch mispredicts
Signal Triggering	Keysight Scope Trace [53]	mem indirect, address, condl, 9 cycles

Table 2: Data Transformation Workloads

CSV parsing involves finding delimiters, fields, and row and column structure, and copying field into the system. The CPU code is from libcsv [8]; these measurements use Crimes (128MB) [16], Trip (128MB) [23], and Food Inspection (16MB) [17] datasets. In Food Inspection, multiple fields contain escape quotes, including long comments and location coordinates. UDP implements the parsing finite-state machine used in libcsv. **Huffman coding** transforms a byte-stream into a dense bit-level coding, with the CPU code as an open-source library *libhuffman* [22]. Measurements use Canterbury Corpus [5] and Berkeley Big Data Benchmark [24]. Canterbury files range from 3KB to 1MB with different entropy and for BDBench we use *crawl*, *rank*, *user*; we evaluate a single HDFS block (64MB, 22MB and 64MB) respectively. For UDP, we duplicate the Canterbury data to provide 64-lane parallelism. **Pattern matching** uses regular expression patterns [80], with the CPU code as Boost C++ Regex [15]. Measurements use network-intrusion detection patterns. Boost supports only single-pattern matching, so we merge the NIDS patterns into a single combined pattern. The UDP code uses ADFA [66] and NFA [62] models. **Compression** CPU code is the Snappy [29] library, and uses the Canterbury Corpus and BDBench dataset, with the UDP library being block compatible. **Dictionary encoding** CPU code is Parquet's C++ dictionary encoder [18]. Dictionary measurements use *Arrest*, *District*, and *Location Description* attributes of Crime [16]. Dictionary-RLE adds a run-length encoding phase. UDP program performs encoding, using a defined dictionary. **Histogram** CPU code is the GSL Histogram library [28]. Measurements use *Crimes.Latitude*, *Crimes.Longitude*, and *Taxi.Fare* with 10, 10, and 4 bins of IEEE FP values [7]. On UDP, the dividers are compiled into an automata scans of 4 bits a time, with acceptance states updating the appropriate bin. Experiments are with 1) uniform-size bins and 2) percentile bins with non-uniform size based on sampling. **Signal triggering** CPU code uses a lookup table that unrolls waveform transition localization automaton described in [53], at 4 symbols per lookup. Trace is proprietary from Keysight oscilloscope. UDP implements exactly the same automaton.

Table 2 summarizes the workloads, and documents the reason for their poor performance on CPUs. First, Snappy, Huffman, CSV, and Histogram are all branch and mispredicted branch intensive as shown by ratios to the geometric mean for the PARSEC [13] benchmarks. Second, dictionary and dictionary-RLE attempt to avoid branches (hash and then load indirect), but suffer from high hashing cost. Third, pattern matching avoids branches by lookup tables but suffers from poor data locality. Finally, triggering is limited by memory indirection followed address calculation, and a conditional.

4.2 Metrics

Metric	Description
Rate (Megabytes/s)	Input processing speed for a single stream on UDP lane
Throughput (Megabytes/s)	Aggregate performance
Area (mm^2)	Silicon area in 28nm TSMC CMOS
Clock Rate (GHz)	Clock Speed of UDP implementation
Power (milliWatts)	On-chip UDP power (see Table 3)
TPut/power (MB/s/watt)	Power efficiency

4.3 UDP Software Stack

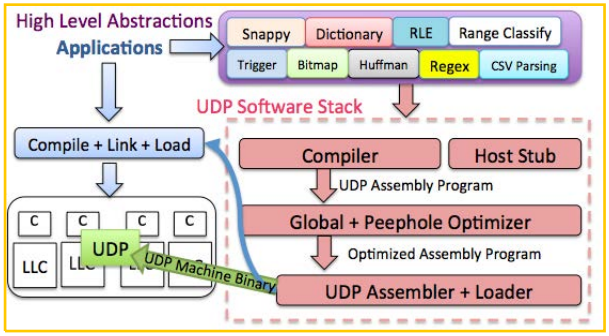


Figure 12: UDP's software stack supports a wide range of transformations. Traditional CPU and UDP computation can be integrated flexibly.

A number of domain-specific translators and a shared backend (see Figure 12) are used to create the UDP programs used for application kernel evaluations in Section 5. The translators support a high-level abstraction and translate it into a high-level assembly language. The backend does intra-block and cross-block optimization, but most importantly, it does the layout optimization to achieve high code density with multi-way dispatch. Further, it optimizes action block sharing, another critical capability for small code size. Finally, the system stubs for linking with CPU programs, enabling a flexible combination of CPU and UDP computing.

4.4 System Configuration and Comparison

We use a cycle-accurate UDP simulator written in C++ to model performance and energy, using speed (1Ghz) and power (864 milliwatts for UDP system) derived from the UDP implementation that is described Section 6.

In Section 5, for each kernel we compare achievable rate for one UDP lane to one Xeon E5620 CPU thread [10]. For throughput per watt, we compare a UDP (64 lanes + infrastructure) to E5620 CPU (TDP 80W, 4-cores, 8-threads). Because parallelized versions were not available for some benchmarks, we estimate performance by multiplying single-thread performance by 8 to create the most optimistic performance scenario for CPU speedup.

5 ARCHITECTURE EVALUATION

For each application, we compare a CPU implementation to a UDP program running on a single or up to 64 lanes (a full UDP), reporting rates and throughput per watt.

5.1 CSV Parsing

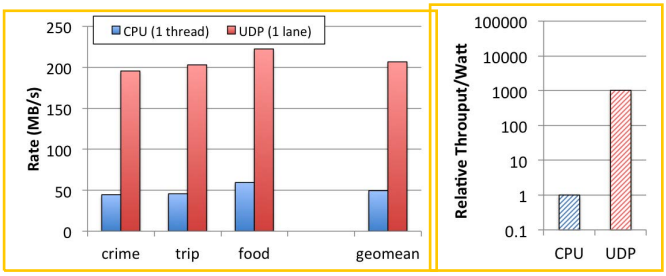


Figure 13: CSV File Parsing.

As in Figure 13, one UDP lane achieves 195–222 MB/s rate, more than 4x a single CPU thread. The full UDP achieves more than 1000-fold throughput per watt compared to CPU. UDP CSV Parsing exploits multi-way dispatch to enable fast parsing tree traversal and delimiter matching; flexible data-parallelism and memory capacity to match the output schema structure; and loop-copy action for efficient field copy.

5.2 Huffman Encoding and Decoding

Figures 14 and 15 show single-lane UDP Huffman encoding at 112 MB/s, 11x speedup and decoding at 366 MB/s, 24x speedup versus a single CPU thread. A full 64-lane UDP achieves geomean of 6,000-fold encoding and 18,300-fold decoding throughput per watt, versus the CPU. The *crawl* dataset has a large Huffman tree is 90% a 16KB local memory bank. UDP flexible addressing enables *crawl* to run by allocating two memory banks for each active lane, but this reduces lane parallelism to 32-way. Each Huffman code tree is a UDP program; one per file. We exclude tree generation time in *libhuffman*. For Huffman UDP multi-way dispatch supports symbol detection; UDP variable-size symbol support gives efficient management of Huffman symbol-size variation, both in performance and code size.

5.3 Pattern Matching

Figure 16 shows that a single UDP lane surpasses a single CPU thread by 7-fold on average, achieving 300-350MB/s across the workloads. The single lane UDP achieves 333-363 MB/s throughput on string matching dataset (simple) and 325-355 MB/s on complex regular expressions (complex). A UDP outperforms CPU by 1,780-fold on average throughput per watt. The collection of patterns are partitioned across UDP lanes, maintaining data parallelism. The UDP code exploits multi-way dispatch for complex pattern detection.

5.4 Dictionary and Dictionary-RLE Encoding

UDP delivers a 6-fold rate benefit for both Dictionary and Dictionary-RLE (see Figure 17). Due to space limits, only Dictionary-RLE performance data is shown. For the full UDP, the power efficiency is more than 4,190x on Dictionary-RLE and 4,440x on Dictionary Encoding versus CPU. The UDP code exploits multi-way dispatch to detect complex patterns and select run length. Flexible dispatch sources are used.

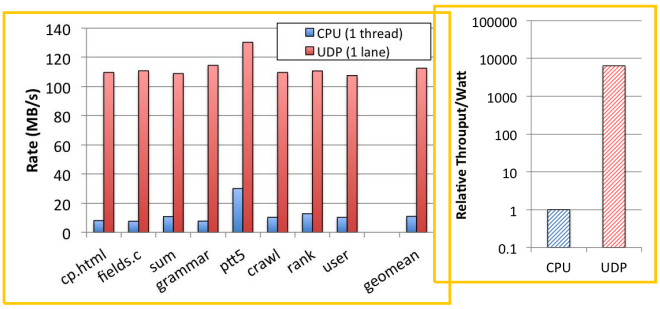


Figure 14: Huffman Encoding.

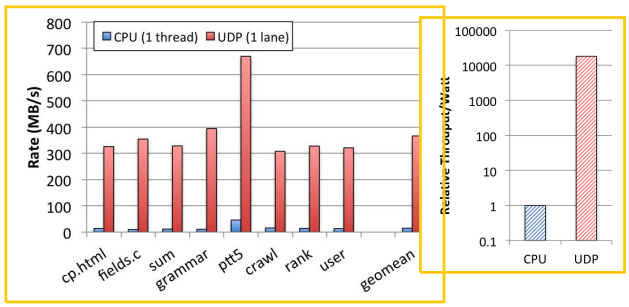


Figure 15: Huffman Decoding.

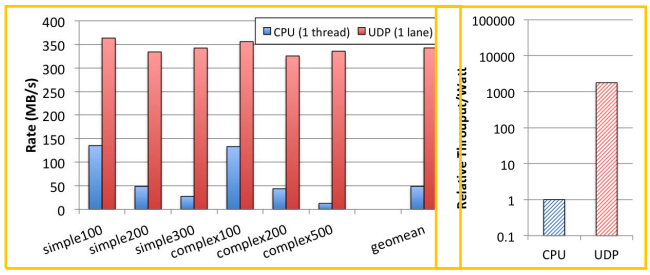


Figure 16: Pattern Matching.

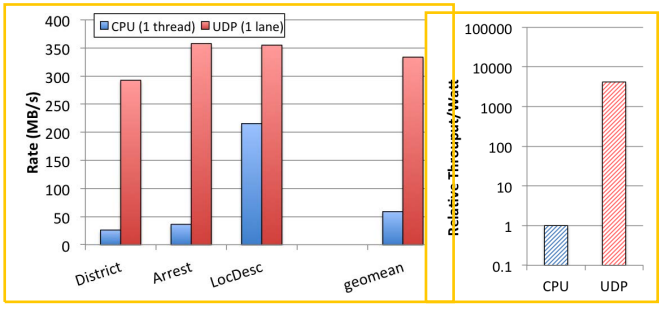


Figure 17: Dictionary-RLE.

5.5 Histogram

Figure 18 shows that one UDP achieves over 400 MB/s rate, matching one CPU thread. The full UDP is 876-fold more power efficient than CPU. The UDP code exploits multi-way dispatch extensively to classify values quickly.

5.6 Compression and Decompression

As shown in Figure 19, UDP Snappy compression with a single UDP lane matches a single CPU thread with performance varying

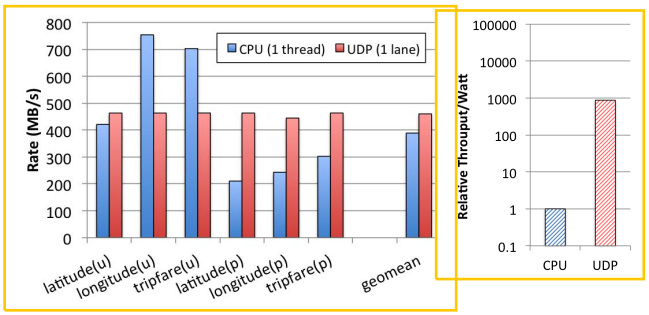


Figure 18: Histogram.

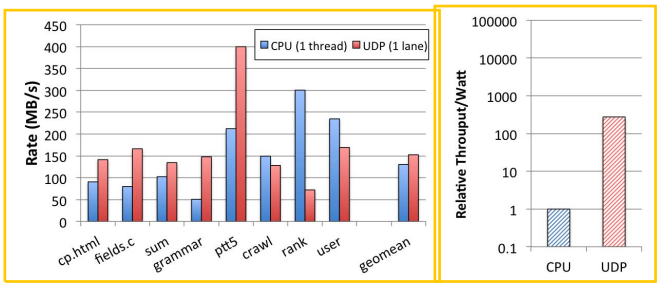


Figure 19: Snappy Compression.

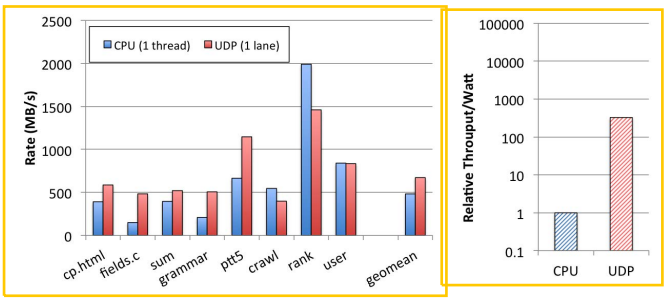


Figure 20: Snappy Decompression.

from 70 MB/s to 400 MB/s (entropy). The full UDP delivers 276x better power efficiency than CPU[3]. Figure 20 shows a similar story for decompression, parity between one UDP lane and a single CPU thread (performance 400 MB/s to 1,450 MB/s). The full UDP achieves a geomean 327x better power efficiency. The UDP Snappy implementation exploits multi-way dispatch to deal with complex pattern detection and encoding choice: flexible data-parallelism and memory addressing to match block sizes, and efficient hash, loop-compare, and loop-copy actions.

5.7 Signal Triggering

One UDP lane delivers constant 1,055 MB/s rate for all transition localization FSMs *p2-p13* [53], 4 times greater than both the CPU (275MB/s) and the FPGA implementation used in Keysight's product [31] (256MB/s). UDP can meet the needs of high-speed signal triggering for all but the highest-speed oscilloscopes. UDP code exploits multi-way dispatch for efficient FSM traversal: flexible memory addressing for large DFA spanning across multiple banks.

[3]The CPU outperforms on *rank* by guessing data is not compressible and skipping input. We did not implement this heuristic for UDP; it processes the entire input.

5.8 Overall Performance

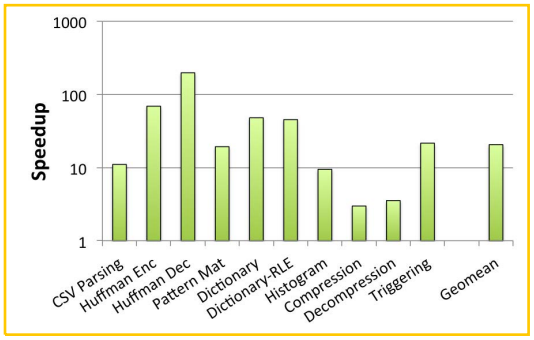


Figure 21: Overall UDP Speedup vs. 8 CPU threads.

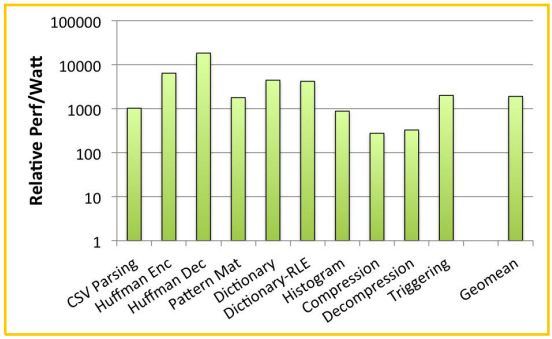


Figure 22: Overall UDP Performance/Watt vs. CPU.

The key UDP architecture features: multi-way dispatch, variable symbol size, flexible dispatch source, and flexible memory sharing accelerate the workload kernels.

Comparing a full UDP (64-lane) with 8 CPU threads shows 3 to 197-fold speedup across workloads with geometric mean speedup of 20-fold (see Figure 21). Second, compare throughput/power for UDP and CPU in Figure 22, using UDP implementation power of 864 milliWatts from Section 6 and 80 watts for the CPU. UDP's power efficiency produces an even greater advantage, ranging from a low of 276-fold to a high of 18,300-fold, with a geometric mean of 1,900-fold. This robust performance benefit and performance/power benefit documents UDP's broad utility for data transformation tasks.

6 UDP IMPLEMENTATION

We describe implementation of the UDP micro-architecture, and summarize speed, power, and area. Each UDP lane contains three key units: 1) Dispatch, 2) Symbol Prefetch, and 3) Action (see Figure 23). The Dispatch unit handles multi-way dispatch (transitions), computing the target dispatch memory address for varied transition types and sources. The Stream Prefetch unit prefetches stream data, and supports variable-size symbols. The Action unit executes the UDP actions, writing results to the UDP data registers or the local memory.

The UDP is implemented in SystemVerilog RTL and synthesized for 28-nm TSMC process with the Synopsys Design Compiler, producing timing, area, and power reports. For system modeling, we estimate local memory and vector register power and timing using CACTI 6.5 [6]. The overall UDP system includes the UDP, a 64x2048-bit vector register file, data-layout transformation engine (DLT) [76], and a 1MB, 64-bank local memory. Silicon power and area for the UDP design is shown in Table 3.

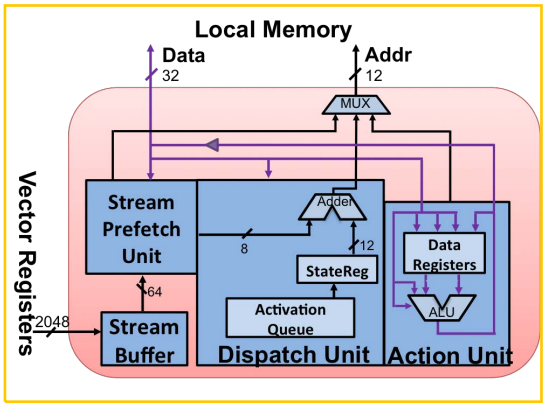


Figure 23: UDP Lane Micro-architecture.

	Component	Power (mW)	Fraction	Area (mm^2)	Fraction
UDP Lane	Dispatch Unit	0.71	37.9%	0.022	40.6%
	SBP Unit	0.24	12.8%	0.008	14.3%
	Stream Buffer	0.22	11.9%	0.002	3.7%
	Action Unit	0.68	36.1%	0.021	39.2%
	UDP Lane	1.88	100.00%	0.054	100.00%
	Component	Power (mW)	Fraction	Area (mm^2)	Fraction
UDP (64 lanes)	64 Lanes	120.56	14.0%	3.430	39.5%
	Vector Registers	8.47	1.0%	0.256	3.0%
	DLT Engine	19.29	2.2%	0.138	1.6%
Shared Area	1MB Local Memory	715.36	82.8%	4.864	56.0%
	UDP System	863.68	100.0%	8.688	100.0%
x86 Core	Core+L1	9700	n.a.	19	n.a.

Table 3: UDP Power and Area Breakdown.

Speed: The synthesized UDP lane design achieves the timing closure with a clock period of 0.97 ns, which includes 0.2 ns to access the 16KB local memory bank [6]. Thus the UDP design runs with a 1GHz clock.

Power: The 64-lane UDP system consumes 864 mW, one-tenth the power of a x86 Westmere EP core+L1 in a 28nm process [10]. Most of the power (82.8%) is consumed by local memory. The 64-lane logic only costs 120.6 mW (14%).

Area: The entire UDP is 8.69 mm^2, including 64 UDP lanes (39.5%) and infrastructure that includes 1MB of local memory organized as 64 banks (56.0%), a vector register file (3%), and a DLT engine (1.6%). The 64 UDP lanes require 3.4 mm^2 – less than one-sixth of a Westmere EP core+L1 in a 32nm process (19 mm^2), and approximately 1% of the Xeon E5620 die area. The entire UDP, including local memory, is one-half the Westmere EP Core + L1.

7 DISCUSSION AND RELATED WORK

We discuss the extensive research that accelerates data transformation, putting our work on UDP in context.

Accelerator		Accel. Algorithm	UDP Algorithm	Accel. Perf (GB/s)	UDP Relative Perf	Accel. Power (W)	UDP Rel. Power Eff.
UAP [54]		String Mat. (ADFA)	String Mat. (ADFA)	38	0.58	0.56W	0.37
		Regex Mat. (NFA)	Regex Mat. (NFA)	15	0.48	0.56W	0.32
Intel Chipset 89xx [4] [30]		DEFLATE	Snappy comp.	1.4	2.1	0.20W	0.50
Microsoft Xpress[5] [56]		Xpress	Snappy comp.	5.6	0.54	108K ALM	- (FPGA)
Oracle Sparc M7[6] [68]	DAX-RLE, -Huff, -Pack, -Ozip	RLE, Huffman, Bit-pack,Ozip	Huffman, RLE, Dictionary	11	1.4	$1.6mm^2$ (Area)	0.56 (Area Eff)
IBM PowerEN[7] [61]	XML	XML Parse	CSV Parse	1.5	2.9	1.95W	
	Compress	DEFLATE	Snappy comp.	1.0	3.0	0.30W	1.1
	Decomp.	INFLATE	Snappy decom.	1.0	13	0.30W	4.7
	RegX	String Match	String Match (ADFA)	5.0	4.4	1.95W	9.8
		Regex Match	Regex Match (NFA)	5.0	1.5	1.95W	3.3

Table 4: Comparing Performance and Power Efficiency of Transformation/Encoding Algorithms.

UDP Architecture: UDP's architecture features are a potent and novel combination for efficient data transformation. Efficient conditional control flow is a core challenge, and branch prediction has long been a focus of computer architecture [38, 64, 67, 83, 84]. As we have shown, the symbol and pattern oriented branch-intensive ETL workloads are particularly difficult, and our results show that UDP multi-way dispatch (that improves on that in the UAP [54]) is an efficient solution. Many efficient encodings use variable-size symbols, and we know of some software techniques [71], but little CPU architecture research on supporting such computations. Hardwired accelerators [39] often employ a wide lookup table and a bit shifter, but unlike the UDP's symbol-size register and *refill* transition, they are not a general, software-programmable solution. UDP's flexible addressing and flexible dispatch sources enable flexibility in data access and keep access latency and energy cost far lower than general memory systems and addressing [34].

Table 4 provides an overall performance comparison to a varied specialized data transformation accelerators, showing UDP's relative performance is at worst nearly 2x slower and up to 13x faster and relative efficiency ranges from 0.32 to 9.8-fold.

Key acceleration areas for *extract-transform-load (ETL)* include parsing, (de)compression, tokenizing, serialization, and validation. Software efforts [70] using SIMD on CPU to accelerate CSV loading and vectorize delimiter detection, achieving 0.3 GB/s single thread performance. This is competitive performance to a UDP lane, but requires much higher power. Hardware acceleration in parsing in PowerEN [61] achieves 1.5 GB/s XML parsing. Compression hardware acceleration achieves 1 GB/s in PowerEN, 5.6 GB/s in Xpress [56], and 1.4GB/s DEFLATE on Intel 89xx series Chipset [30] (Table 4). With only 21 lanes (memory capacity limited), UDP outperforms the ASIC accelerators by 2.1-13x. The Xpress [56] comparison is complicated because of its use of large dedicated FPGA. All of these specialized accelerators' implementations lack the flexible programmability of UDP. Tokenization alone can be accelerated by pattern matching accelerators [12, 54, 57, 80], but lack the programmability to address the costly follow-on processing (e.g. deserialization and validation) which often dominates execution time (Figure 1a). The UDP handles these tasks and more. Its flexible programmability can address varied ETL and transformation tasks and future application-specific encodings on algorithms.

Hardware support for *query execution*, notably relational operators, has demonstrated significant speedups for analytic workloads (Q100 [82]); these systems don't do the broad range of data transformation that is the focus of UDP. Offload systems in storage systems (e.g. Ibex [81], Netezza [2], Exadata [4]) gain performance by exploiting internal storage bandwidth. UDP's low power and programmability make it a candidate for such storage embedding. FPGA-based efforts that accelerate histogramming [63] highlight opportunities to use UDP for rich statistics at low cost to enable better query optimization.

Both PowerEN and Sparc M7 integrate accelerators for compression, transpose, scan, and crypto. On Sparc M7, a 4-core DAX unit achieves 15 GB/s scan on compressed data [68], but lacks the ability to support new formats or algorithms. In contrast, UDP's programmability supports varied tasks and application-specific formats or algorithms, while providing comparable performance (Table 4).

Acceleration of *stream processing* [49, 65] and *network processors* [12] can achieve high data processing rates with support for pattern-matching and network interfaces (see also NIC and "bump-in-the-wire" approaches [45]). The UDP complements these systems, providing programmable rich data transformation in both stream and networking contexts efficiently. UDP's performance suggests incorporation is a promising research direction.

Acceleration of *Network Intrusion Detection and Deep Packet Inspection (NID/DPI)* includes exploiting SIMD [44, 73] and aggressive

[4] We estimate compression power by 20W TDP[21] and exclude clock, grid, IO/bus, and crypto, using relative ratio [9].
[5] Altera Stratix V FPGA.
[6] Scale to 28nm TSMC and estimate based on chip die size [26, 32].
[7] IBM 45nm SOI [9].

prefiltering [25] to achieve 0.75-1.6 GB/s using a powerful Xeon out-of-order core. Software speculative approaches can increase stream rate, but at significant overhead [72, 75, 86]. GPU implementations [85] report throughputs 0.03 GB/s (large pattern sets) increasing to 1.6GB/s [87] (small sets). Several network processors [12, 61] employ hardwired regular expression acceleration to reach 6.25GB/s throughput. Unified Automata Processor achieves up to 5x better performance [54] by exploiting programmability to employ the best finite-automata models. The UDP improves on the UAP achieving much greater generality, but at a cost in performance and energy efficiency (Table 4). Recent work [57] achieves remarkable stream rate (32 GB/s) at high power consumption (120W).

	UAP Feature	UDP Feature
Transitions	stream only	control and stream-driven
Symbol	8-bit fixed	symbol size register (1-8,32 bits)
Dispatch Source	stream buffer only	stream buffer and data register
Addressing	single bank, fixed memory per lane	multi-bank addressing; match data parallelism to memory needs
Action	logic and bit-field ops	rich, flexible arithmetic and memory ops

Table 5: UAP and UDP Highlighted Differences

UDP Relationship to the UAP: The UDP builds on the architecture insights of the Unified Automata Processor (UAP), a programmable architecture that enables applications to use any extended finite automata model (both deterministic and non-deterministic), and delivers high performance and low power [54]. UDP adopts and extends UAP's multi-way dispatch mechanism with varied sources, variable-size symbols, flexible memory addressing, and a few basic instructions to dramatically broaden its scope. As a result, the UDP can accelerate parsing, compression, tokenizing, deserialization and histogramming workloads in addition to regular expression matching. These workloads share core features of control-flow intensity and prediction difficulty. The poor branch prediction makes these workloads also benefit from UDP's short execution pipeline. We highlight UDP key new features compared to in Table 5. These features enable performance across the broad range of data transformation documented in this paper.

8 SUMMARY AND FUTURE WORK

We present the Unstructured Data Processor (UDP), an architecture designed for general-purpose, high performance data transformation and processing. Our evaluation shows the UDP's significant performance benefits for a diverse range of tasks that lie at the heart of ETL, query execution, stream data processing, and intrusion detection/monitoring. The UDP delivers comparable performance of more narrowly specialized accelerators, but its real strength is its flexible programmability across them. An implementation study shows that the Si area and power costs for the UDP are low, making it suitable for CPU offload by incorporation into the CPU chip, memory/flash controller, or the storage system.

The UDP's flexible programmability and performance opens many opportunities for future research. Interesting directions include: exploration of additional new application spaces that may benefit from UDP data transformation (e.g., bioinformatics), new domain-specific languages and compilers that provide high-level programming support (e.g. [40, 58]), and specific design studies that incorporate UDP's (one or several) at various locations in data center systems or database appliances. Finally, data centers have recently begun to deploy FPGA's [41, 45], we plan to use these to enable studies with large-scale applications, exploring achievable system-level performance with the UDP. For example, one opportunity is to compare a programmable-UDP enhanced system with accelerated database appliance systems (hardwired customization).

9 ACKNOWLEDGEMENTS

This work was supported by in part the National Science Foundation under Award CNS-1405959 and the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration. It builds on the 10x10 Project funded by the Defense Advanced Research Projects Agency under award HR0011-13-2-0014. We also acknowledge support from the CERES Center for Unstoppable Computing, Keysight, and the Synopsys Academic program. The authors thank Calin Cascaval, Tung Thanh-Hoang, Xiaoan Ding and the anonymous reviewers for comments that improved the paper significantly.

REFERENCES

[1] [n.d.]. Cadence Tensilica Xtensa. https://ip.cadence.com/uploads/902/TIP_What_Why_How_Cust_Processors_WP_V3_FINAL-pdf. ([n. d.]).
[2] [n.d.]. IBM Netezza Data Warehouse Appliances. http://www-01.ibm.com/software/data/netezza/. ([n. d.]).
[3] [n.d.]. NEON – ARM. https://www.arm.com/products/processors/technologies/neon.php. ([n. d.]).
[4] [n.d.]. Oracle Exadata Storage Server. http://www.oracle.com/technetwork/index.html. ([n. d.]).
[5] 2001. Canterbury Corpus. (2001). http://corpus.canterbury.ac.nz/.
[6] 2008. CACTI 6.5. http://www.cs.utah.edu/~rajeev/cacti6/. (2008).
[7] 2008. IEEE 754 floating-point format. (2008). http://grouper.ieee.org/groups/754/
[8] 2009. libcsv C library. https://sourceforge.net/projects/libcsv/. (2009).
[9] 2010. The IBM Power Edge of Network Processor. (2010). http://www.cercs.gatech.edu/iucrc10/material/franke.pdf
[10] 2010. Intel Xeon Processor E5620 Specification. (2010). https://ark.intel.com/products/47925
[11] 2011. The ARMv8 Architecture. white paper. (2011). https://www.arm.com/files/downloads/ARMv8_white_paper_v5.pdf
[12] 2011. Cavium NITROX DPI L7 Content Processor Family. (2011). http://www.cavium.com/processor_NITROX-DPI.html
[13] 2011. PARSEC 3.0. (2011). http://parsec.cs.princeton.edu/
[14] 2012. Big Data Research and Development Initiative. https://www.whitehouse.gov/sites/default/files/microsites/ostp/big_data_press_release_final_2.pdf. (2012).
[15] 2012. Boost C++ library. http://www.boost.org/. (2012).
[16] 2012. Chicago City Crime Report. (2012). http://data.cityofchicago.org
[17] 2012. Chicago City Restaurant Inspection. (2012). http://data.cityofchicago.org
[18] 2013. Apache Parquet C++ library. https://github.com/apache/parquet-cpp. (2013).
[19] 2013. *Frontiers in Massive Data Analysis*. National Research Council Press. ISBN: 978-0-309-28778-4. DOI: 10.17226/18374.
[20] 2013. Intel Advanced Vector Extensions. (2013). https://software.intel.com/en-us/isa-extensions/intel-avx
[21] 2013. Intel communications chipset 8955. (2013). http://ark.intel.com/products/80372/Intel-DH8955-PCH
[22] 2013. libhuffman C library. https://github.com/drichardson/huffman. (2013).
[23] 2013. New York City Taxi Report. http://www.andresmh.com/nyctaxitrips/. (2013).
[24] 2014. Berkeley Big Data Benchmark. (2014). https://amplab.cs.berkeley.edu/benchmark/
[25] 2015. Intel Hyperscan. (2015). https://github.com/01org/hyperscan
[26] 2015. Sparc M7 Die Size (wikipedia). https://en.wikipedia.org/wiki/SPARC. (2015).

[27] 2016. Federal Big Data Research and Development Strategic Plan. http://www.whitehouse.gov/. (May 2016).

[28] 2016. GNU Scientific Library. https://www.gnu.org/software/gsl/. (2016).

[29] 2016. Google Snappy compression library. https://github.com/google/snappy. (2016).

[30] 2016. Intel Chipset 89xx Series. http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/scaling-acceleration-capacity-brief.pdf. (2016).

[31] 2016. Keysight CX3300 Appliance. (2016). http://www.keysight.com/en/pc-2633352/device-current-waveform-analyzers?cc=US&lc=eng

[32] 2016. M7: Next Generation SPARC. (2016). http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/migration/m7-next-gen-sparc-presentation-2326292.html

[33] 2016. PostgreSQL Database. (2016). https://www.postgresql.org/

[34] 2017. Intel64 and IA-32 Architectures. (2017). https://software.intel.com/en-us/articles/intel-sdm

[35] 2017. TPC-H Benchmark. http://www.tpc.org/tpch/. (2017).

[36] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling queries on compressed data. In *Proc. of NSDI'15*.

[37] Ioannis Alagiannis et al. 2012. NoDB: efficient query execution on raw data files. In *Proc. of SIGMOD'12*. ACM, 241–252.

[38] Jorge Albericio et al. 2014. Wormhole: Wisely predicting multidimensional branches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 509–520.

[39] James Allen et al. 1994. Huffman decoder architecture for high speed operation and reduced memory. (1994). US Patent 5,325,092.

[40] Kevin Angstadt, Westley Weimer, and Kevin Skadron. 2016. RAPID Programming of Pattern-Recognition Processors. In *Proc. of ASPLOS'16*.

[41] Jeff Barr. 2016. Developer Preview – AQS EC2 Instances (F1) with Programmable Hardware. https://aws.amazon.com/blogs/aws/developer-preview-ec2-instances-f1-with-programmable-hardware/. (nov 2016).

[42] C. Gordon Bell. 1977. *What Have We Learned from the PDP-11?* Springer Netherlands.

[43] Shekhar Borkar and Andrew A. Chien. 2011. The Future of Microprocessors. *Commun. ACM* 54, 5 (May 2011), 67–77.

[44] Robert D. Cameron et al. 2014. Bitwise Data Parallelism in Regular Expression Matching. In *Proc. of PACT '14*.

[45] Adrian Caulfield et al. 2016. A Cloud-Scale Acceleration Architecture. In *Proc. of MICRO'16*. ACM/IEEE.

[46] Fay Chang et al. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008).

[47] Andrew A. Chien, Allan Snavely, and Mark Gahagan. 2011. 10x10: A General-purpose Architectural Approach to Heterogeneity and Energy Efficiency. In *The Third Workshop on Emerging Parallel Architectures at the International Conference on Computational Science*.

[48] Andrew A. Chien, Tung Thanh-Hoang, Dilip Vasudevan, Yuanwei Fang, and Amirali Shambayati. 2015. 10x10: A Case Study in Highly-Programmable and Energy-Efficient Heterogeneous Federated Architecture. *ACM SIGARCH Computer Architecture News* 43, 3 (2015), 2–9.

[49] William J. Dally et al. 2004. Stream Processors: Programmability and Efficiency. *Queue* 2, 1 (March 2004).

[50] Ahmed Elgohary et al. 2016. Compressed linear algebra for large-scale machine learning. *Proceedings of the VLDB Endowment* 9, 12 (2016), 960–971.

[51] Yuanwei Fang et al. 2014. Generalized Pattern Matching Micro-Engine. In *4th Workshop on Architectures and Systems for Big Data (ASBD), held with ISCA'14* (2014).

[52] Yuanwei Fang and Andrew A. Chien. 2017. *UDP System Interface and Lane ISA Definition*. Technical Report. https://newtraell.cs.uchicago.edu/research/publications/techreports/TR-2017-05

[53] Yuanwei Fang, Andrew A. Chien, Andrew Lehane, and Lee Barford. 2016. Performance of parallel prefix circuit transition localization of pulsed waveforms. In *2016 IEEE International Instrumentation and Measurement Technology Conference Proceedings*.

[54] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast Support for Unstructured Data Processing: The Unified Automata Processor. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*.

[55] Yuanwei Fang, Andrew Lehane, and Andrew A. Chien. 2015. EffCLiP: Efficient Coupled-Linear Packing for Finite Automata. *University of Chicago Technical Report, UR-2015-05* (May 2015).

[56] Jeremy Fowers et al. 2015. A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs. In *Proc. of FCCM'15*.

[57] Vaibhav Gogte et al. 2016. HARE: Hardware Accelerator for Regular Expressions. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE.

[58] Bjørn Bugge Grathwohl, Fritz Henglein, Ulrik Terp Rasmussen, Kristoffer Aalund Søholm, and Sebastian Paaske Tørholm. 2016. Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*.

[59] Shay Gueron. 2012. Intel Advanced Encryption Standard (AES) New Instructions Set. https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set. (September 2012).

[60] Peter B. Hallin and David W. Ussery. 2004. CBS Genome Atlas Database: a dynamic storage for bioinformatic results and sequence data. In *Bioinformatics*, Vol. 20. Oxford Univ Press.

[61] Timothy Heil et al. 2014. Architecture and Performance of the Hardware Accelerators in IBM PowerEN Processor. *ACM Trans. Parallel Comput.* 1, 1 (May 2014).

[62] John E. Hopcroft and Jeffrey D. Ullman. 1969. *Formal Languages and Their Relation to Automata*.

[63] Zsolt Istvan et al. 2014. Histograms As a Side Effect of Data Movement for Big Data. In *Proc. of SIGMOD '14*.

[64] Daniel A. Jimenez. 2005. Piecewise linear branch prediction. In *Proc. Of ISCA'05*. IEEE Computer Society.

[65] Brucek Khailany et al. 2001. Imagine: Media Processing with Streams. *IEEE Micro* 21, 2 (March 2001).

[66] Sailesh Kumar et al. 2006. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. *Proc. of SIGCOMM'06* (Aug. 2006).

[67] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. 1997. The bi-mode branch predictor. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 4–13.

[68] Penny Li et al. 2015. 4.2 A 20nm 32-Core 64MB L3 cache SPARC M7 processor. In *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International*. IEEE, 1–3.

[69] Sergey Melnik et al. 2010. Dremel: Interactive Analysis of Web-scale Datasets. In *PVLDB'10*.

[70] Tobias Mühlbauer et al. 2013. Instant Loading for Main Memory Databases. *Proceedings of the VLDB Endowment* 6, 14 (Sept. 2013).

[71] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel Finite-state Machines. In *Proc. of ASPLOS '14*.

[72] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel Finite-state Machines. In *Proc. of ASPLOS '14*.

[73] Valentina Salapura et al. 2012. Accelerating Business Analytics Applications. In *Proc. of HPCA '12*.

[74] Mike Stonebraker et al. 2005. C-store: a column-oriented DBMS. In *Proc. of VLDB '05*. VLDB Endowment.

[75] Arun Subramaniyan and Reetuparna Das. 2017. Parallel Automata Processor. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 600–612.

[76] Tung Thanh-Hoang et al. 2016. A Data Layout Transformation (DLT) accelerator: Architectural support for data movement optimization in accelerated-centric heterogeneous systems. In *Proc. of DATE'16*. IEEE.

[77] T. Thanh-Hoang, A. Shambayati, C. Deutschbein, H. Hoffmann, and A. A. Chien. 2014. Performance and energy limits of a processor-integrated FFT accelerator. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6.

[78] T. Thanh-Hoang, A. Shambayati, H. Hoffmann, and A. A. Chien. 2015. Does arithmetic logic dominate data movement? a systematic comparison of energy-efficiency for FFT accelerators. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 66–67.

[79] Jim Turley. 2014. Introduction to Intel Architecture. white paper. (2014). https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf

[80] Jan Van Lunteren et al. 2012. Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator. In *MICRO'12*.

[81] Louis Woods, Zsolt Istvan, and Gustavo Alonso. 2014. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proc. VLDB Endow.* 7, 11 (July 2014).

[82] Lisa Wu et al. 2014. Q100: The Architecture and Design of a Database Processing Unit. In *Proc. of ASPLOS '14*.

[83] Tse-Yu Yeh and Yale N. Patt. 1991. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*. ACM, 51–61.

[84] Tse-Yu Yeh and Yale N. Patt. 1992. Alternative implementations of two-level adaptive branch prediction. In *Proc. of ISCA'92*. ACM, 124–134.

[85] Xiaodong Yu and Michela Becchi. 2013. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *Proc. of CF '13*.

[86] Zhijia Zhao and Xipeng Shen. 2015. On-the-fly principled speculation for FSM parallelization. In *Proc. of ASPLOS '15*. ACM, 619–630.

[87] Yuan Zu et al. 2012. GPU-based NFA Implementation for Memory Efficient High Speed Regular Expression Matching. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*.