THE UNIVERSITY OF CHICAGO

EXTREME ACCELERATION AND SEAMLESS INTEGRATION OF RAW DATA PROCESSING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
YUANWEI FANG

CHICAGO, ILLINOIS
JUNE 2019

Copyright © 2019 by Yuanwei Fang

All Rights Reserved

Dedicated to my parents, for their endless love.

Essentially, all models are wrong, but some are useful.

– George E. P. Box

TABLE OF CONTENTS

LIST OF FIGURES . . . viii

LIST OF TABLES . . . x

ACKNOWLEDGMENTS . . . xi

ABSTRACT . . . xiii

1 INTRODUCTION . . . 1
- 1.1 Emerging Big Data Applications . . . 1
- 1.2 The Rise of Data Lake . . . 2
- 1.3 Radical Technology Disruptions . . . 5
- 1.4 Problem Summary . . . 8
- 1.5 Thesis Statement . . . 8
- 1.6 Contribution and Thesis Organization . . . 9

2 BACKGROUND . . . 13
- 2.1 SQL Databases . . . 13
- 2.2 NoSQL and Raw Data Processing . . . 15
- 2.3 Data Lake . . . 17
- 2.4 The Rise of Hardware Acceleration . . . 20
- 2.5 Summary . . . 21

3 RELATED WORK . . . 23
- 3.1 Raw Data Parsing . . . 23
- 3.2 Predicate Pushdown . . . 24
- 3.3 Optimizing Extract-Transform-Load . . . 24
- 3.4 Low-Latency Processing on Fresh Data . . . 26
- 3.5 Data Encodings for Fast Query Execution . . . 26
- 3.6 Database Hardware Acceleration . . . 27
- 3.7 Accelerator Hardware System Integration . . . 29
- 3.8 Hardware Customization Adoptions . . . 30
- 3.9 Summary . . . 31

4 PROBLEM AND APPROACH . . . 33
- 4.1 The Dream of Raw Data Processing . . . 33
- 4.2 Problems . . . 34
  - 4.2.1 Data Transformation Cost Bottleneck . . . 34
  - 4.2.2 Narrowly Applicable Accelerator Pitfall . . . 36
  - 4.2.3 Database Software Pitfall . . . 36
- 4.3 The ACCORDA Approach . . . 37
  - 4.3.1 Unified Transformation Accelerator . . . 39
  - 4.3.2 Accelerated Transformation Operators . . . 41
- 4.4 Summary . . . 42

5 UNIFIED TRANSFORMATION ACCELERATOR . . . 43
5.1 UTA Requirements . . . 43
5.1.1 Functional Requirements . . . 43
5.1.2 Performance Requirements . . . 44
5.1.3 Cost Requirements . . . 44
5.2 In Memory-Hierarchy Integration . . . 45
5.3 UTA Architecture: The Unstructured Data Processor . . . 47
5.3.1 Overview . . . 47
5.3.2 Key UDP Architecture Design Features . . . 48
5.3.3 UDP Lane: Fast Symbol and Branch Processing . . . 49
5.4 UTA Evaluation . . . 60
5.4.1 Methodology . . . 61
5.4.2 UDP Performance . . . 67
5.4.3 UDP Area and Power . . . 72
5.4.4 ACCORDA Micro-benchmarks . . . 75
5.5 Additional Related Work . . . 78
5.6 Summary . . . 81

6 ACCELERATED TRANSFORMATION OPERATORS . . . 82
6.1 ATO: ACCORDA's Software Architecture . . . 82
6.1.1 Encoding in the Operator Interface . . . 82
6.1.2 Uniform Worker Model . . . 85
6.2 Benefits of ATO Software Architecture . . . 87
6.2.1 Preserving Software Architecture . . . 87
6.2.2 Preserving Flexible Query Optimization . . . 89
6.3 UTA Unleashes the Power of ATO . . . 92
6.3.1 UTA as the ACCORDA Accelerator . . . 93
6.3.2 Why UTA's Memory Integration . . . 94
6.4 Summary . . . 95

7 ACCORDA EVALUATION . . . 96
7.1 Methodology . . . 96
7.1.1 System Modeling . . . 96
7.1.2 Workloads . . . 99
7.2 Benefits of In Memory-Hierarchy Integration . . . 99
7.3 Benefits of Software Programmability . . . 101
7.4 ACCORDA Raw Data Performance . . . 102
7.5 Performance Sensitivity Analysis . . . 106
7.5.1 Data Statistics Sensitivity . . . 106
7.5.2 Format Type and Complexity Sensitivity . . . 108
7.6 Summary . . . 109

8 SUMMARY AND FUTURE WORK . . . 110
8.1 Summary . . . 110
8.2 Future Work . . . 112
8.2.1 UDP Architecture Extension . . . 112
8.2.2 Domain-Specific Data Transformation Language . . . 113
8.2.3 More UTA Applications . . . 114
8.2.4 Novel Data Encoding . . . 116

REFERENCES . . . 119

LIST OF FIGURES

1.1 Massive Volume of Raw Data Powers Diverse Big Data Applications. 2
1.2 Data Lake Holds Diverse Data and Supports On-demand Analysis. 3
1.3 Data Centers Embrace Accelerated Compute and Memory Technologies. 6

2.1 Main Components of a DBMS. 14

4.1 Ideal Raw Data Analysis. 33
4.2 High Load Costs: Compressed CSV into PostgreSQL. 35
4.3 Hardware Acceleration Disrupts Database Software Architecture. 38
4.4 Approaches for Raw Data analysis. 39
4.5 Hardware Acceleration Approaches for Data Transformation. 40
4.6 Approaches for Accelerating Raw Data Processing. 42

5.1 Acceleration Functional Requirements. 44
5.2 Two Accelerator Integration Approaches. 46
5.3 In Memory-Hierarchy Integration of UTA into NoC fabric 46
5.4 UTA Local Memory is Exposed as Part of CPU Address Space. 47
5.5 UDP and UDP Lane Micro-architecture. 49
5.6 Branch Execution on Symbol: Branch Offset (BO), Branch Indirect (BI), Multi-way Dispatch (MWD). 50
5.7 Cycles spent on Branch Misprediction and Computation. 50
5.8 Relative Branch Rate for ETL kernels using BO, BI and Multi-way. 51
5.9 Code size for varied branch and dispatch approaches. 51
5.10 UDP Transition and Action Formats: Imm Action, Imm2 Action, Reg Action. All are 32-bits. 52
5.11 Huffman Decoding Tree: *00,01,10,110,111*. Solid box is symbol-size register. Other actions not shown. 54
5.12 Variable-size Symbol Approaches on kernels requiring dynamic and static variability. 55
5.13 Performance Benefit for adding stream buffer and scalar register as UDP dispatch source. 57
5.14 Addressing Models. Local: each lane has private address space; Global: each lane shares entire address space; Restricted: each lane flexibly chooses a window. . . 57
5.15 Addressing Impacts Performance. 59
5.16 Addressing Impacts More Than Performance. 60
5.17 Memory Reference Energy. 60
5.18 UDP's software stack supports a wide range of transformations. Traditional CPU and UDP computation can be integrated flexibly. 61
5.19 CSV File Parsing. 67
5.20 Huffman Encoding. 68
5.21 Huffman Decoding. 68
5.22 Pattern Matching. 69
5.23 Dictionary-RLE. 69
5.24 Histogram. 70
5.25 Snappy Compression. 70

5.26 Snappy Decompression. 71
5.27 Overall UDP Speedup vs. 8 CPU threads. 71
5.28 Overall UDP Performance/Watt vs. CPU. 72
5.29 UDP Lane Micro-architecture. 73
5.30 Regular Expression Matching. 76
5.31 LZ-77 Decompression. 77
5.32 CSV Parsing. 77
5.33 Deserialization. 78

6.1 ATO's Encoding-extended Operator Interface. 83
6.2 An Encoding-Based Query Optimization Example with Detailed Transformation. 84
6.3 Encoding Operators in a Query. 84
6.4 Many accelerated systems require different types of workers (upper). ATO has uniform accelerated workers (lower). 86
6.5 ATO Query Engine Software Architecture (added components in green). 88
6.6 Comparing Optimizer Architectures for Acceleration. 88
6.7 Comparing Execution Model. 89
6.8 ATO Rule Examples. 90
6.9 Integral and Explicit Encoding in Queries 91
6.10 Benefits of ATO Fine-grained Query Optimization 92

7.1 Modified System Components across the Stack for ACCORDA Modeling. 97
7.2 ACCORDA Performance Modeling. 98
7.3 Runtime Breakdown. 100
7.4 Off-chip Data Movement. 100
7.5 Average Waiting Time. 101
7.6 UDP Software Programmability vs. FPGA Hardware Programmability. 102
7.7 Overall System Comparison on Unloaded, On-disk, Raw Data. 103
7.8 Query Execution Time with UTA Acceleration and ATO-enabled Acceleration. . 104
7.9 Comparing ACCORDA on Raw Data with SparkSQL on Loaded Data. 105
7.10 Selectivity Impact on Execution Time. 107
7.11 Format, System, and Hardware Impact on Execution Time 108

LIST OF TABLES

5.1 Data Transformation Workloads 64
5.2 UDP Power and Area Breakdown. 74
5.3 Comparing Performance and Power Efficiency of Transformation/Encoding Algorithms. 80

7.1 Hardware Platform Statistics 97
7.2 Predicates in TPC-H Queries 106

ACKNOWLEDGMENTS

It is absolutely memorable and fruitful during my Ph.D. journey at the University of Chicago. I would like to express my most sincere gratitude to the people who have helped me through the years.

First of all, I would like to thank my Ph.D. advisor, Prof. Andrew A. Chien, for being a great mentor in research and life. His professional expertise and insights, both in industry and academia, helped me diving into cutting-edge research, his invaluable suggestions guided me through technical challenges, and his high-standard mentorship had a profound impact on my critical thinking, open mindset, and collaborative attitude. It is a great honor to work with him.

I would like to thank Prof. Aaron J. Elmore, Prof. Michael J. Franklin, and Prof. Ian T. Foster, for providing me insightful suggestions and feedback on research and serving on my defense committee. I am grateful to Aaron, Mike, and Ian, for investing significant time from their busy schedules in reading and helping improve the presentation and dissertation. Thanks to all faculty members in the UChicago System Group for an open, inspiring, and collaborative academic environment.

I feel truly fortunate to work with many excellent colleagues. Many thanks to my team members in UChicago, Qualcomm, and Google: Chen Zou, Tung T. Hoang, Aiman Fang, Fan Yang, Nan Dun, Amirali Shambayati, Dilip Vasudevan, Hajime Fujita, Chaojie Zhang, Calin Cascaval, Arun Raman, Hui Chao, Jichuan Chang, Andrew McCormick, Yixin Luo, and Biswapesh Chattopadhyay for all the research collaborations and technical discussions. I specially appreciate the infrastructure support, from both UChicago and UT Austin – especially the help from Prof. Derek Chiou, Bob Bartlett, Phil Kauffman, and Colin Hudler, that enabled my research experiments and daily work. I also want to thank Sandy Quarles and Margaret Jaffey for their kind and professional administrative help.

Beyond research activities, my Ph.D. life has been colorful and joyful with my fantastic friends: Jialei Wang, Liwen Zhang, Haopeng Liu, Wencong Qian, Weicong Chen, Zhixuan

Zhou, Huazhe Zhang, Yuan Li, Yun Li, Yuxi Chen, Yan Liu, Qinqing Zheng, Tong Hu, Riza Suminto, Stephen Fitz, Xiaoan Ding, Yi He, Junwen Yang, Chunwei Liu, Chi Li, Congxi Lu, Ke Tian, Keen Sung, Kwanghoon An, Huayang Xie, Yanning Cui, Wenfei Sun, and many more that I have certainly missed listing above.

Finally, my special thanks to my parents. They have always been supportive and encouraging me to pursue my goals in research and life. Despite being in different countries, their unconditional and greatest love has always been accompanying me.

ABSTRACT

New sources of big data such as the Internet, mobile applications, data-driven science and large-scale sensors (IoT) are driving demand for growing computing performance. Efficient analysis of data in native raw formats in real-time is increasingly important because of rapid data generation, demand for analytics, and insights for immediate responses. Traditional data processing systems can deliver high-performance on loaded data, but transforming raw data into these loaded formats is expensive. Data transformations, rather than arithmetic operations, dominate the task performance. Such transformation is a critical performance bottleneck of raw data processing. We propose the ACCelerated Operators for Raw Data Analysis (ACCORDA), a combined software and hardware approach, to accelerate data analytics on unloaded raw data. ACCORDA enables real-time decision making and fast knowledge exploration on dirty, diverse, and ad-hoc raw data, such as fresh sensor data, web crawled, and business records.

The Unified Transformation Accelerator (UTA) is ACCORDA's hardware approach. It creates flexible architecture support for data transformation in analytical workloads. Exploiting efficient hardware customization, a scratchpad memory, and MIMD parallelism, Unstructured Data Processor (UDP) is a novel hardware accelerator based on the UTA approach. UDP demonstrates the feasibility of the UTA approach. We propose the UDP's instruction set, micro-architecture, and compiler toolchain. UDP has four unique features: multi-way dispatch, variable-size symbol, flexible-source dispatch, and flexible addressing. Extensive evaluation of data transformation kernels, ranging from compression to pattern matching, shows UDP achieves 20x average speedup and 1,900x energy efficiency when compared to an 8-thread CPU. The UDP's implementation is >100x less power and area than a single CPU core.

The Accelerated Transformation Operators (ATO) is ACCORDA's software approach. ATO applies two design choices for integrating data transformation acceleration – sub-typing operator interface with encodings and uniform worker model. The encoding-extended interface

enables new accelerated operators to be included in a query plan. Runtime data formats can be transformed to meet the encoding requirements of accelerated operator implementations. In addition, query optimizer can re-order encoding operators for lazy data transformation, and fuse them to improve data locality and reduce transformation cost. Uniform worker model preserves system software architectures and provides a uniform runtime to the execution engine, empowering rule-based optimizers to drive flexible encoding-based optimization. We demonstrate that the key enablers are the UTA's low-cost, high-performance design and its in memory-hierarchy integration for efficient, low-overhead data sharing with CPUs. Together, they enable flexible software exploitation of hardware acceleration and worker thread integration.

ACCORDA achieves significant acceleration on data transformation tasks, with speedups up to 4.9x on regex matching, 2.6x on decompression, 2x on parsing, and 20x on deserialization when compared to an 8-thread CPU. We evaluate ACCORDA using end-to-end TPC-H queries on unloaded data with raw format. Hardware acceleration contributes 1.1x-6.3x improvement alone, and software elements such as query optimization for data encoding unlocked by ATO deliver an additional 1.2x-11.8x speedup. Combining UTA's acceleration and ATO's encoding optimization, ACCORDA achieves 3.3x-13.2x overall speedups on single-thread performance when compared to the baseline Spark SQL. We further show that this performance benefit is robust across format complexity of query predicates and selectivity (data statistics). Furthermore, ACCORDA robustly matches or even outperforms (by up to 11.4x) prior systems that depend on caching transformed data, while computing on raw, unloaded data.

CHAPTER 1

INTRODUCTION

This chapter introduces emerging big data applications (Section 1.1) and the data lake architecture (Section 1.2). Then, we discuss several technology disruptions in Section 1.3, followed by the problem summary and the thesis statement in Section 1.4 and 1.5, respectively. Finally, Section 1.6 describes the overall contribution of the dissertation and its organization.

1.1 Emerging Big Data Applications

With the rapid advances 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 growing driver of computing performance. Business values and knowledge insights mined through vast volume of data from diverse sources makes real-time data processing and timely complex batch analysis on raw data increasingly popular and valuable. Real-world applications include, 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). We term these data as "raw data" as they stay in their native representations as opposed to databases' loaded data. Figure 1.1 depicts the broad set of big data applications that are opened up by raw data processing. Real-time analysis on these raw data avoids expensive transformation cost up-front and reduces the latency to get answers from newly ingested data. Furthermore, advances in large-scale machine learning, deep learning, and artificial intelligence enable making accurate predictions based on historical data. Therefore, all kinds of data from various applications are kept, not only for the ones used today, but also for the ones that might be used someday in the future.

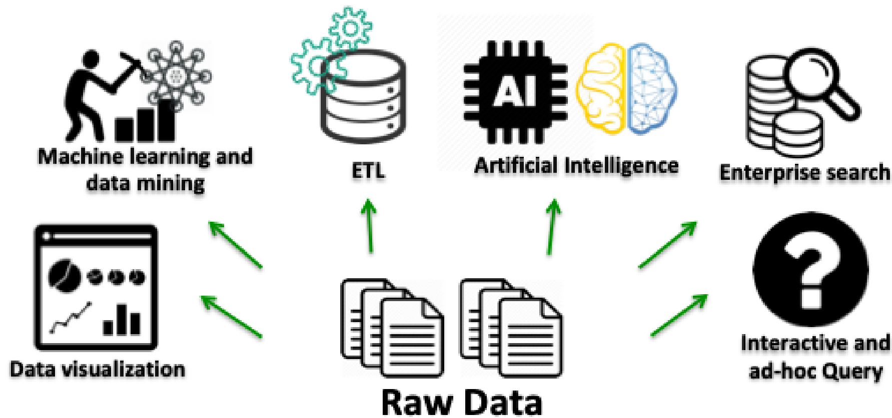


Figure 1.1: Massive Volume of Raw Data Powers Diverse Big Data Applications.

Transforming raw data formats for fast computation is expensive. Data manipulation-transformation-movement is the key barrier for high-performance analysis on raw data. Traditional databases and data warehouses pay huge up-front cost on moving and converting data from external sources to comply with internal schemas and formats. This batch process creates long latency before getting any query results. There is an increasing demand to avoid that data preparation period and allow analysis happened directly on raw data. Data should be kept for all time to allow future exploration. This philosophy fundamentally differs from traditional databases, which makes it attractive for organizations who want to pursue answers on fresh data quickly with more flexible analysis capabilities. As a result, data analysis is shifting from traditional databases to a new paradigm. We discuss more in the next section about this new paradigm – data lake.

1.2 The Rise of Data Lake

Driven by a fast increase in quantity, type, and number of sources of data, many data analytics approaches start to use the data lake architecture. In the data lake (see Figure 1.2), incoming data is pooled in large quantity – hence the name "lake" – and because of the data's varied origin (e.g., purchased from other providers, internal employee data, web scraped,

database dumps, etc.), it varies in size, encoding, age, quality, etc. Additionally, it lacks consistency or any common schema because applications change and evolve so fast that prior designed data organization and structures can't keep up with newly generated fresh data. To embrace such a fast-changing IT world, data lake doesn't enforce any fixed schema. Analytics applications pull data from the lake as needed, digesting and processing it on-demand for the current analytics task, as Figure 1.2 shown. The diversity of applications run on data lake is vast, including machine learning, data mining, artificial intelligence, ad-hoc query, and data visualization. Fast direct processing on raw data is critical for these applications to deliver valuable business and knowledge insights in real-time.

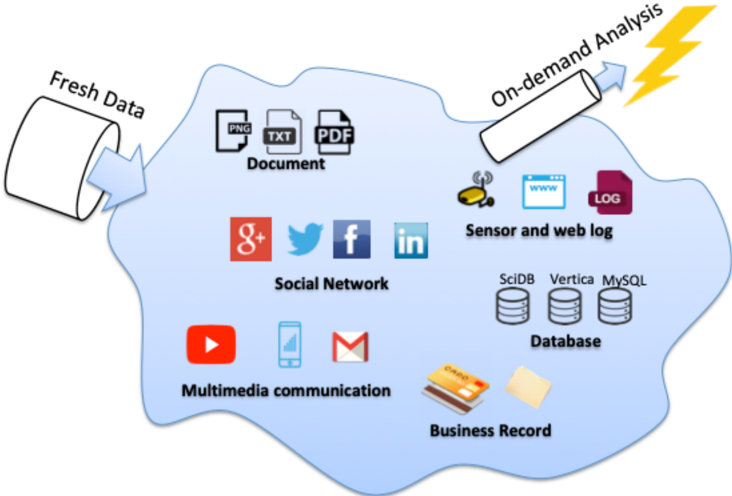


Figure 1.2: Data Lake Holds Diverse Data and Supports On-demand Analysis.

The data lake has numerous advantages over traditional data warehouses. First, raw data is only processed when needed, the data lake model avoids the processing and IO work for unused data. Second, direct access to raw data can reduce the data latency for analysis, a key property for streaming data sources or critical fresh data. Third, data lake supports exploratory analytics where schemas and attributes may be updated rapidly as the structure is learned. Fourth, data lakes can avoid information loss during loading by keeping the source information, so applications can interpret incomplete or inconsistent data in a customized fashion. Fifth, the data lake model avoids scaling limits because raw data in the lake often

exceeds the maximum capacity of database systems, and licensing limits or cost (price/loaded data licensing models). One consequence of the data lake model is that data remains in dirty and inconsistent structures instead of in-machine SQL types and a well-defined schema.

From a system point of view, the essence of the data lake is to shift ETL (Extract-Transform-Load) work from offline into the query execution. Of course, the shifted ETL work is much smaller - just the part needed for the query, but the required query work is more because there is no index support and fresh data keeps ingested all the time. For each query, we pull the relevant data from the lake, and process the raw data (essentially load, then analysis) to complete the query. Because the data lake puts raw data cleaning and conversion on the critical path of query execution, it elevates raw data processing to the level of a critical performance concern. Queries that require answers quickly, or those that require interactive and real-time analysis, perhaps on recently-arrived raw data, are sensitive to the speed of raw data processing.

Prior research on narrow performance optimization of batch ETL's parsing can accelerate the newly introduced data transformation parts that lie on the critical path of data lake analysis. For example, several software-acceleration approaches exploit SIMD instructions for parsing raw data. InstantLoad [98] exploits SIMD acceleration on finding delimiters. MISON [93] exploits speculative parsing and SIMD acceleration for JSON records. The effectiveness highly relies on representativeness of the training data and regularity of record structures. In contrast, our approach should provide robust, high-performance on parsing raw data, regardless of format complexity.

Lazy data loading is another commonly used performance optimization that can accelerate the aforementioned data transformation in data lake analysis. It avoids unnecessary ETL by only processing the data actually needed. Databases [37, 83, 82, 44] apply lazy loading, caching materialized results, and positional maps to avoid expensive data transformations. These approaches require data reuse across queries and significant main memory to achieve high-performance. When these properties do not hold, performance can become variable and

poor. Another technique for raw data processing is pushing predicates down past the loading work. For example, several systems [83, 43, 103] exploit query semantics for partial predicate pushdown. For these systems, performance is highly dependent on selectivity, and filters are limited to very simple (constant-literal predicates) and thus do not work for all external data representations (e.g., gzip). On the contrary, our approach aims at stateless raw data processing with high-performance.

In short, the rise of data lake imposes the performance challenge of raw data processing. Current approaches solve the problem partially using memory caching and SIMD acceleration, but lack performance robustness, usage flexibility, and sufficient speedups. Our approach doesn't have these limitations. It enables flexible, stateless, fast raw data processing that is robust across format complexity, data statistics, and system's internal states.

1.3 Radical Technology Disruptions

Computer architecture is facing major disruptions from advanced technologies – notably, the demise of Dennard Scaling, the slowing down of Moore's Law, and the increased importance of data movement. As our need for computing performance keeps increasing, we must continue to find ways to ensure that performance benefits are from corresponding improvements in energy efficiency and cost efficiency. Furthermore, we should consider the importance of data movement optimization in all aspects of a system, from software to hardware, and from disk to cache. These radical disruptions from technology give birth to our approach.

Dennard Scaling [58] helps to achieve improved energy efficiency in the past: every 30% reduction in transistor linear dimensions results in twice as many transistors per area and 40% faster circuits, which also comes with a corresponding reduction to supply voltage at the same rate as transistor scaling. Unfortunately, this has become extremely difficult as transistor size approaches atomic scales. It is now widely acknowledged that Dennard scaling has stopped. This means that any significant improvements in energy efficiency in the foreseeable future are likely to come from architectural techniques instead of fundamental

technology scaling. More recently, Moore's Law is also slowing down due to a number of factors spanning both economic considerations (e.g., fabrication costs in the order of billions of dollars) and fundamental physical limits (limits of CMOS scaling). The challenges are particularly exacerbated by the previously described performance need for data processing with deeper analysis over ever growing volumes of raw data. This means that continued improvements in computing performance need to come from architectural optimizations, for area and resource efficiency, but also around more hardware-software codesign and fundamental new architectural paradigms.

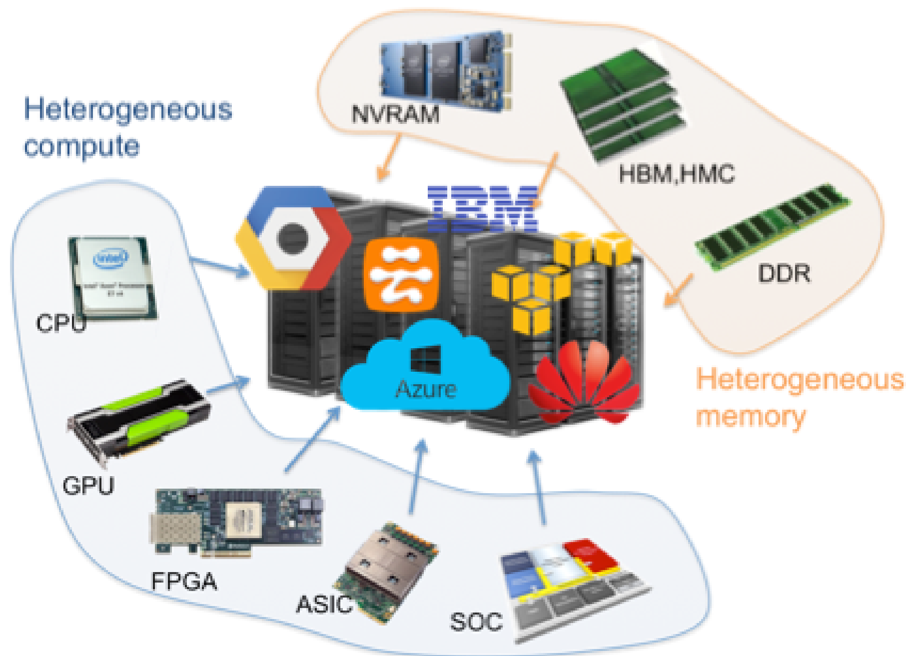


Figure 1.3: Data Centers Embrace Accelerated Compute and Memory Technologies.

Accelerators are by far the most promising approach. By tailoring the architecture to the application, we can achieve both improved power and area efficiency. The trade-off often is that we sacrifice generality and flexibility to increase efficiency for specific types of workloads. We have seen modern data centers are evolving towards heterogeneity, containing accelerated compute and memory devices as shown in Figure 1.3. However, many existing accelerator approaches provide very narrow acceleration [123, 122, 85, 81, 109, 91, 36, 70]. Continually

deploying such accelerators is likely to make the system Frankenstein-style and unmanagable. Accelerator design needs to take a holistic view, across hardware and software, to improve the generality. This includes thinking about the design in the context of supporting interfaces, compilers, and software system integration.

Accelerators in the post Moore's Law era provide customized computation support, keeping performance and energy efficiency increasing for certain workloads. That trend makes data movement (e.g., from memory, from SSD, etc.) more critical than ever, whose importance embodies in two aspects – energy and memory bandwidth scaling. With the advanced process and technology (e.g., FinFET, FDSOI), the relative energy cost of arithmetic logic is dramatically scaled down. On the other hand, technology for memory, is slowly scaled far behind logic due to constraints in cost and reliability [106, 28]. This causes the impact of data movement, as a fraction of system energy, becomes an important concern. For example, researchers [116] have shown that the energy cost of FFT floating point computation only contributes 0.8% for a system in 32nm process, and 0.01% for the one in 7nm FinFET process, respectively, with DDR3 memory. Most of the energy (>90%) consumed in these systems is spent on data movement. Not only relative energy consumption but also memory bandwidth scaling, makes data movement increasingly important. Over the past two decades, the exponential increase in microprocessor computation rate has far outstripped the slower increase in memory bandwidth. For example, McCalpin reported in an SC keynote in 2016 [28], that flops/memory access ratio has increase from from 4 (in 1990) to 100 (in 2020). This ratio continues to double every 5 years with no sign of slowing. The ITRS projects that package pin counts will scale less than 10 percent per year [106]. At the same time, per-pin bandwidth increases come with a difficult trade-off in power. A high-speed interface requires additional circuits (e.g. phase-locked loops, on-die termination) that consume static and dynamic power. As a result, a long history of declined bytes/flops in computer architecture [106] is observed. The forthcoming accelerator-based systems are likely performance-bounded by the available IO bandwidth. Optimizing data movement helps to eliminate this bottleneck.

In summary, the end of Dennard Scaling and Moore's Law advocates future accelerator-based heterogeneous systems [54]. As described above, data movement becomes increasingly important given the commonality in memory-limited execution. Our approach is in line with these hardware trends in computer architecture, and is especially effective in data movement optimization in a system.

1.4 Problem Summary

We briefly describe the thesis problem in this section. Section 4.2 explains the problem in-depth with a detailed discussion.

Fast raw data processing is critical for many analytics tasks spanning from data-driven science to mobile applications. However, existing systems deliver poor performance when analyzing dirty, diverse, and ad-hoc raw data. In particular, data transformation such as parsing, loading, extraction, and filtering on raw data is the key performance limiter in real-time raw data processing; it is orders of magnitude slower than IO subsystems (Section 4.2). Existing software and hardware solutions don't solve the problem because of either their narrow acceleration capability, or their sensitive speedups regarding to data statistics and system states. Even worse, integrating these approaches usually requires fundamental changes that disrupt the software architecture (e.g., query engine) of existing analytics systems. We aim at enabling robust, flexible, high-performance raw data processing on a wide range of analytics tasks with minimal changes to existing software architectures.

1.5 Thesis Statement

We propose ACCORDA (ACCelerated Operators for Raw Data Analysis), a method that combines a single data transformation accelerator and a software architecture that deeply integrates the acceleration, enabling query execution and optimization across data representations. Together, these deliver data analytics performance on raw data that matches or even

exceeds the performance of traditional SQL analytical systems on loaded data.

- **Single hardware accelerator:** a set of hardware resources that can be wholly repurposed for a variety of tasks.
- **Raw data:** native, unstructured, and error-prone inputs such as CSV files, logs, JSON records, HTML web pages, and nested documents.
- **Loaded data:** in contrast to raw data, loaded data has already been organized and converted into system internal formats for optimal execution efficiency.

1.6 Contribution and Thesis Organization

In this thesis, we propose the Accelerated Operators for Raw Data Analysis (ACCORDA) that contains a software and a hardware approach to accelerate raw data processing in a robust, flexible, and high-performance manner with minimal software integration disruption.

The ACCORDA's hardware approach, Unified Transformation Accelerator (UTA), is a high-performance, low-power data transformation accelerator with orders of magnitude performance and energy efficiency improvement on a range of ETL tasks over CPUs. We design the Unstructured Data Processor (UDP) architecture [66], based on the UTA approach, with unique features including multi-way dispatch, variable-size symbols, flexible-source dispatch, and an addressing architecture for efficient, flexible UDP lane-bank coupling. UDP is fully software-programmable with its instruction set documented in detail [61]. The ISA and compiler algorithm is designed based on the prior published ancestor, Unified Automata Processor (UAP) [64, 63]. We develop a software cycle-accurate simulator and an ASIC prototype to concretely evaluate UDP's performance, power, and area cost.

The Accelerated Transformation Operators (ATO) is ACCORDA's software approach. ATO adopts two design choices for integrating data transformation acceleration – sub-typing operator interface with encodings and uniform worker model. The encoding-extended interface enables new accelerated operators to be included in a query plan. Runtime data formats can

be transformed to meet the encoding requirements of accelerated operator implementations. In addition, query optimizer can re-order encoding operators for lazy data transformation, and fuse them to improve data locality and reduce transformation cost. Uniform worker model preserves system software architectures and provides a uniform runtime to the execution engine, keeping traditional query optimization structures and benefits. ATO enjoys low-overhead data sharing with CPU cores by UTA's in memory-hierarchy integration.

We use a hybrid-timed simulation methodology with the ATO software prototype built on SparkSQL for evaluation. ATO integrates UTA hardware acceleration that is implemented on FPGA to evaluate full ACCORDA system's performance on raw data processing, including micro-benchmarks and end-to-end queries. These results suggest that ACCORDA provides robust query performance on raw data that matches or even exceeds performance on conventional systems with loaded data. Specific contributions of the thesis include:

- Accelerated Operators for Raw Data Analysis (ACCORDA), a new software and hardware architecture that combines a single data transformation accelerator and a software architecture that integrates the acceleration, enabling query execution and optimization across data representations.

- The Unified Transformation Accelerator (UTA), a principled hardware approach to design a tiny, low-power, high-performance, and software-programmable accelerator for ACCORDA with flexible support across a wide range of data transformation tasks.

- Hardware architecture of the ACCORDA accelerator, called the Unstructured Data Processor (UDP), that realizes the UTA approach, including the key features of multi-way dispatch, variable-size symbols, flexible-source dispatch, and an addressing architecture for efficient, flexible UDP lane-bank coupling. Quantitative evaluation for each that documents its effectiveness.

- Accelerated Transformation Operators (ATO), the ACCORDA's software architecture that integrates data transformation acceleration seamlessly by sub-typing operator interfaces with encoding and provides a uniform runtime by accelerating all worker

threads. Thus, ACCORDA enables flexible exploitation of encoding-based query transformation and optimization that tailor data encodings for accelerated operators' implementation during runtime.

- Performance evaluation of the UDP architecture on diverse workloads showing speedups from 11x 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 mm2 at 864 milliwatts, making UDP viable for CPU offload, or even incorporation in a memory/flash controller or network-interface card.
- Performance evaluation of the full ACCORDA system, which demonstrates its robust and high-performance raw data processing, delivering an overall 3.3x-13.2x speedup on a range of raw data analysis when compared to a CPU and traditional database software on a representative set of TPC-H queries. When compared to systems with loaded data, ACCORDA approaches raw data analysis speeds within 2x.
- ACCORDA performance analysis shows hardware acceleration contributes 1.1x-6.3x improvement alone, and software elements such as query optimization for data encoding unlocked by ATO deliver an additional 1.2x-11.8x speedup.

The remainder of the thesis is organized as follows. Chapter 2 gives a brief background on recent advances on big data systems, the data lake model, and industrial adoption of hardware customization. In Chapter 3, we discuss the related research literature. In Chapter 4, we present the problem and our proposed solution (ACCORDA). Chapter 5 describes the ACCORDA's hardware architecture with a detailed performance and cost evaluation. In

Chapter 6, we describe the ACCORDA's software architecture and evaluate its effectiveness. In Chapter 7, we study the ACCORDA system's performance. Finally, we summarize the thesis results and outline multiple future research directions in Chapter 8.

CHAPTER 2

BACKGROUND

In this chapter, we describe basic background of traditional SQL databases, NoSQL and raw data processing, data lake, and hardware acceleration of data processing. The recent shift in big data from data warehouses to data lakes further motivates the importance of fast and efficient NoSQL raw data processing. Finally, we discuss the increasing hardware acceleration adoption in modern data centers.

2.1 SQL Databases

SQL databases use Structured Query Language (SQL) for defining and manipulating data. On one hand, it is extremely powerful because SQL is one of the most widely-used options for data analysis, making it convenient and especially great for complex queries. On the other hand, SQL databases are restrictive. SQL requires predefined schema to determine the structure of data before querying it. In addition, all data must follow the same structure. This requirement may lead to significant up-front preparation. Any changes in the structure would be both difficult and disruptive to the whole data warehouse. Data model is restricted as tables for efficient relational query optimization (e.g., join order, projection pushdown, etc). The structured data model with aggressive optimizations enables dramatics performance improvement on query processing.

A typical DBMS has five main components, as illustrated in Figure 2.1. The client communication manager maintains the connection established by the client and the database server via ODBC or JDBC connectivity protocol. Its responsibility is to establish and remember the connection state for the caller (e.g., a client or a middleware server), to respond to SQL commands from the caller, and to return both data and control messages (result, codes, errors, etc.) as appropriate. Upon receiving the SQL command from the client, the DBMS process manager must decide whether the system should begin processing the query

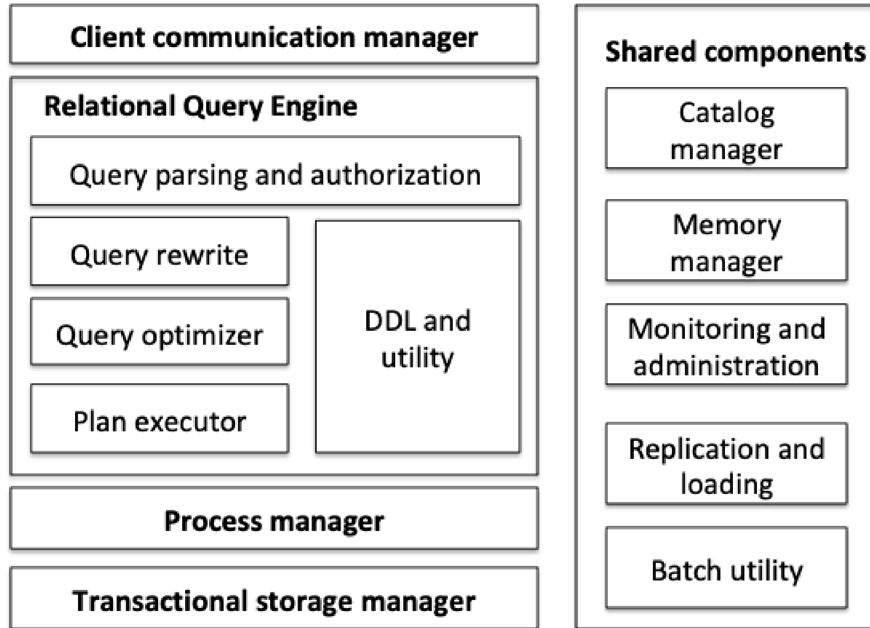


Figure 2.1: Main Components of a DBMS.

immediately, or defer execution until a time when enough system resources are available to devote to this query. At this stage, a thread of computation is assigned to the received SQL command. Once admitted and allocated as a thread of control, the query can begin to execute through the relational query processor. This set of modules checks that the user is authorized to run the query, and compiles the SQL query text into an internal query plan. Once compiled, the resulting query plan is handled via the plan executor. The plan executor consists of operators for executing any query. Typical operators implement relational query processing tasks including joins, selection, projection, aggregation, sorting and so on, as well as calls to request data records from the storage system. The storage system includes algorithms and data structures for organizing and accessing data on storage media (e.g., disk, SSD, etc), including basic structures like tables and indexes. It also includes a buffer management module that decides when and what data to transfer to memory buffers. These data fetching tasks are controlled by the transactional storage manager to preserve ACID properties. There are also components shared across queries such as catalog, monitoring, and data replication services. In this thesis, we focus on optimizing the relational query processor

for fast and efficient raw data processing without modifying other components in a DBMS.

The parallelism and scalability in a SQL database is mainly achieved by horizontal partitioning. The key idea is to distribute the rows of a relational table across the nodes of the cluster so they can be processed in parallel. Most parallel SQL database systems offer a variety of partitioning strategies, including hash, range, and round-robin partitioning. The system automatically manages and optimizes the various alternative partitioning strategies for the tables involved in the query.

There are two major types of SQL databases differed by use case: On-Line Transaction Processing (OLTP) and On-Line Analytical Processing (OLAP). Both types of systems have highly optimized internal data storage representations for performance and size efficiency. On-line transaction processing applications are high throughput and insert or update-intensive with concurrency and recoverability. As a result, the data is usually organized as rows and partitioned across machines. Direct multi-dimensional analysis using the row-based format is not efficient because of unnecessary IO and cache efficiency. Therefore, the OLAP systems are designed to address this problem using columnar format, targeting at read-only and high-performance data analysis workloads. A batch data importing phase is needed before analysis to transform external data (e.g., row-oriented format) into the internal columnar format with specific encodings and compression. This loading process creates long latency to obtain answers. Raw data in the data lake is rarely organized and optimized for efficient analysis as the ones in OLAP. In this thesis, we concentrate on accelerating direct analysis on external raw data without batch loading but approaching the efficiency of an OLAP system.

2.2 NoSQL and Raw Data Processing

NoSQL systems have dynamic schema for unstructured data. Data is stored in many ways: it can be column-oriented (e.g. Dremel[96]), document-oriented (e.g. MongoDB [55]), graph-based (e.g. Neo4j [119]) or organized as a key-value store (e.g. Cassandra [87]). Data models are not limited to tables , but also contain graph, array, unstructured document, etc. The

most popular NoSQL execution engine is Map-Reduce [57]. MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key-value pair to generate a set of intermediate key-value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.

The increasing adoption of NoSQL Map-Reduce systems in the industry comes with the rise of ETL workloads and "read-once" data sets. In short, there are five major use cases for MR systems [111]: 1) read logs of information from several different sources; 2) parse and clean the log data; 3) perform complex transformations; 4) decide what attribute data to store; and 5) load the information into a DBMS or other storage engine. Unlike a DBMS, MR systems do not require users to define a schema for their data. Thus, MR-style systems easily store and process semi-structured or unstructured data. The MR execution engine is more like a general-purpose ETL system that users can compose arbitrary procedures that operate on the supplied data sources, rather than a set of highly modularized relational query operators which result in easier query optimizations. Because more data from diverse sources, especially raw data from outside, can't be fully captured by the structured SQL models and relational schemas, they fall into the broad domain that uses NoSQL storage models and processing engines, rather than databases. The performance of raw data processing in NoSQL systems become increasingly critical.

Raw data processing requires in-line data parsing, cleaning, and transformation in addition to the computations required in traditional databases. The raw data exists in an external format, often serialized for portability, that must be parsed to be understood. Because only a small fraction of the raw data is used in most cases, additional effort is required to parse and then extract the desired data. This is a stark contrast to database computations on loaded data that exploit an internal, optimized format for execution performance. Furthermore, raw data is unorganized so that various elements are coupled together without a clear schema. To extract data elements out, parsing and encoding are required. Deserialization is required

to transform elements into binary formats that CPUs execute well on. To allow further read optimizations, row to column transposition is needed. We term these operations as data transformation. All of these transformation workloads have a common execution model derived from the finite automata or finite-state machines (FSM); these models underly all grammar based encodings. In many cases, these raw formats are composed of a sequence of tag/value pairs (e.g., Snappy, RecordIO, CSV, JSON, and XML). The corresponding operation is determined by the specific tag value (e.g. delimiter in CSV, tag in Snappy, or key in JSON). These formats often employ sub-word encodings (bytes, symbols, etc.) and variable-length encodings (strings, Huffman, etc.). The FSM is extended by a few short computations (code blocks), each of which is associated with a state transition arc, performing the operation for each tag. CPU software usually implements such FSMs with switch statements or conditional branches (multi-target branches). These branches are expensive, dominating the data loading time (Section 4.2.1 and 5.3). CPU's are notorious for poor performance both on sub-word processing and multi-target branches.

In summary, processing vast amount of raw data requires speed, flexible schema, and distributed storage. NoSQL systems became the preferred choice for managing big data with unstructured and unorganized representations.

2.3 Data Lake

With the increase in accessibility of Internet of Things, machine learning, artificial intelligence, and the availability of cheap storage, huge amounts of structured, semistructured, and unstructured data are generated and stored for a variety of applications. Traditional data warehouses can't confront the rich structures and types existed in these diverse data sources. As a result, current big data practices facilitate the birth of data lake. A data lake is a system or repository of data stored in its natural raw format, usually object blobs or files. It is a single store of all enterprise data including raw copies of source system data and transformed data used for tasks such as reporting, visualization, analytics and machine learning. It can

include structured data from relational databases (rows and columns), semi-structured data (CSV, XML, JSON), unstructured data (emails, documents, PDFs) and binary data (images, audio, video). There are four key features that a data lake outperforms a traditional data warehouse with respect to the rapid data source changes in the big data era.

First, data lake includes not only data that is in use today, but also data that may be used later or even data that may never be used just because it might be used someday. Data is also kept for all time to allow future exploration. In contrast, during the development of a traditional data warehouse, a considerable amount of time is spent on analyzing data sources, understanding business needs, and profiling data. This process leads to a highly structured data model designed for reporting. One of critical decisions to be made is what data to include and to not include in the data warehouse. Generally, if data isnt used to answer specific questions in a defined report, it may be excluded from the warehouse. This is usually done to simplify the data model and also to save space on expensive disk storage. The data lake approach is much more powerful than traditional data warehouse solutions. Even better, off-the-shelf servers combined with cheap storage makes scaling a data lake to terabytes and petabytes more affordable with rapid disk cost reduction in recent years.

Second, data lake allows rich data types, schemas, and ad-hoc structures in data. The data lake approach embraces non-traditional types existed in data sources such as web server logs, sensor data, social network activity, text and images. In the data lake, we keep all data regardless of source and structure. We keep it in its raw form and we only transform it when we need it. Native data types in the source data are captured in the data lake. In contrast, data warehouses largely ignore non-traditional data types and generally consist of data extracted from transactional systems and consist of quantitative attributes that describe them. In that sense, data lake is very general and flexible on data organization and representation, which makes it able to accommodate rapid changes in data sources in the future.

Third, data lake has broad use cases. We describe some of the most common use cases

in data analytics. In many organizations, the majority part of users are operational, who want their reports with key performance metrics in a spreadsheet every day. The data warehouse is sufficient for them because it is well organized, easy to use, and straight-forward to understand. Next, another part of users do more analysis on the data. They use the data warehouse but often go back to source systems for data that is excluded from the warehouse, or even sometimes bring in data from outside the organization. The data warehouse is the default source but they often go beyond its scope. Finally, the last part of users do deep analysis. They may create totally new data sources based on research. They leverage many different types of data and come up with entirely new questions to be answered. These users may use data warehouses but often ignore it as they are usually charged with going beyond its capabilities. These users may use advanced analytic tools (e.g., Spark and Hadoop) and capabilities like machine learning and predictive modeling. The data lake approach supports all of these users equally well. The data scientists can go to the data lake and work with large and varied data sets while other users make use of more structured views of the data provided for their use.

Finally, data lake supports exploratory data analysis with minimal cost. One of the main shortcomings about data warehouses is the long duration to change them. A considerable amount of time is spent up front during data warehouse structure development. Data warehouses have slow data loading process as a result of the design to make analysis efficient. Structural changes consume many developer resources and take long duration. Many business questions can't wait for the data warehouse team to adapt the changes. On the other hand, since data lake stores all data in its raw form, users can go beyond the data warehouse to explore data in novel ways and answer their questions at their pace. If the result of an exploration is shown to be useful and there is a desire to repeat it, then a more formal schema can be applied and automation can be developed to extend the results to a broader audience. If the result is not useful, it can be simply discarded. Neither data warehouse changes nor development resources are required.

The advances made from the data science community, especially from machine learning, has led to data lake become the future analysis paradigm. Insightful values can be delivered through timely analysis in the data lake making NoSQL systems and efficient raw data processing more attractive than before.

2.4 The Rise of Hardware Acceleration

Moore's Law is the rule of thumb that has come to dominate computing. It states that the number of transistors on a microprocessor chip will double every two years or so. The past 30 years of Moore's Law scaling bring dramatic improvements in transistor density, speed, and energy. Combined with micro-architecture and memory hierarchy techniques, it delivered 1000-fold microprocessor performance improvement [48]. The exponential improvement transformed the first crude home computers of the 1970s into more sophisticated machines of the 1980s and 1990s, and from there gave rise to high-speed Internet, smartphones, wearable devices, and, more recently, deep learning and artificial intelligence that are becoming prevalent today.

However, following Moore's Law has become increasingly expensive. Improving microprocessor performance usually means scaling down the elements of circuits such that electrons could move between them more quickly. Scaling, in turn, requires major improvements in photolithography, the basic technology for etching microscopic elements onto a silicon surface. When transistors are approaching the atomic dimensions, manufacturing becomes extremely difficult and complex. The revenue of chips, even sold for huge quantities, may not be able to cover the cost of upgrading fabrication facilities and manufacturing yields. The chip-making process is getting too complex, often involving hundreds of stages, which meant that taking the next step down in scale required a network of materials-suppliers and apparatus-makers to deliver the right upgrades at the right time. Therefore, this market-driven cycle could not sustain the relentless cadence of Moore's law by itself in the near future.

With the sunset of Moore's Law, hardware specializing might be the only viable choice

as the remaining tool in the toolbox of hardware changes that can further improve energy efficiency. Hardware specialization includes fixed-function accelerators (such as media codecs, cryptography engines, and floating-point engines), programmable accelerators, and even dynamically customizable logic (such as FPGAs and other dynamic structures). In general, customization increases computational performance by exploiting hardwired or customized computation units, customized wiring and interconnect for data movement, reduced instruction sequence overheads at some cost in generality, and sufficient parallelism and dedicated data representations for computations. In some cases, units hardwired to a particular data representation or computational algorithm can achieve 10x-100x greater energy efficiency than a general-purpose register organization [123, 122, 85, 80, 36, 70, 63, 66, 65, 54].

Historically, the deployment of specialized computing accelerators in data centers has been very limited due to their narrow acceleration capabilities. Although accelerators promised greater computing efficiencies, such benefits came at the cost of drastically restricting the number of workloads that could benefit from them. Moreover, software customization is difficult as well, especially for large programs. Developers of software applications had little incentive to customize for accelerators that might be available on only a fraction of the machines in the field and for which the performance advantage might soon be overtaken by advances in the traditional microprocessor. With slowing improvement in single-thread performance and the end of Moore's Law, this landscape has changed significantly, and for many applications, accelerators may be the only path toward increased performance or energy efficiency. The resulting challenge in future accelerator design is to achieve generality and energy efficiency at the same time. In this thesis, we demonstrate how to design hardware accelerators to tackle this challenge.

2.5 Summary

This chapter provides background in data processing with SQL databases and NoSQL systems. The increased popularity of the data lake further motivates the importance of NoSQL raw

data processing. Furthermore, we are facing the performance challenge of micro-processors due to the slowing down of Moore's Law. Together, these game-changing contexts set the stage for our proposed ACCORDA approach in Chapter 4.

CHAPTER 3

RELATED WORK

In this chapter, we discuss some state-of-art raw data processing performance optimization techniques in Section 3.1, 3.2 and 3.3. Later, Section 3.4 describes current practices on low-latency fresh data processing. Then, we present a few novel data encodings for fast query execution in Section 3.5. Finally, we discuss recent hardware acceleration techniques, integration approaches, and their industrial adoptions in Section 3.6, 3.7, and 3.8.

3.1 Raw Data Parsing

One thread of optimization on raw data processing is through SIMD acceleration for parsing. InstantLoad [98] exploits SIMD instructions to find CSV delimiters. Vectorization reduces the branch miss rate by 50% and achieves single-thread performance over 250MB/s on parsing on a conventional x86 CPU. Compared to InstantLoad, our approach provides 40x faster single thread performance and can easily saturate a standard memory channel. MISON [93] exploits speculative parsing with SIMD acceleration for JSON records. A list of pattern trees is built from a small set of training data to select the possible parsing pattern speculatively. The SIMD acceleration is applied for finding delimiters such as *brackets, quotes, and newline*. The best single thread performance of MISON achieves 2GB/s when parsing simple JSON formats. However, its effectiveness highly relies on the representativeness of training data for building the pattern trees with the underlying assumption that JSON records follow an overall structure or template that does not vary significantly across records. In contrast, our approach accelerates JSON parsing 10x faster than MISON robustly without the constraint on overall record structure, and can be further applied to more complicated nested structures (e.g., XML).

One of the major challenges in applying SIMD acceleration for parsing on raw data is the broadness of supported formats and the performance correlation regarding to internal

structure complexity. It is not clear that SIMD instructions can accelerate parsing on other popular formats such as protocol buffer and XML. On the other hand, our approach provides robust acceleration on a wide range of formats. The high-performance is also robust regardless of the complexity of internal structures.

3.2 Predicate Pushdown

Early filtering or predicate pushdown is a classic system optimization approach to filter input as early as possible to avoid the cost of subsequent computation. In raw data processing, early partial filtering with inexpensive predicates [83, 43, 103] are used to avoid data transformation for parsing and deserialization. However, the performance is sensitive to data statistics, predicate semantics, and cost of raw filters. The early filters only supports constant-literal predicate, basically a substring filtering. As a result, a broad set of range-based filters, predicates computed on loaded form, and predicates spanning across multiple columns can't benefit from the early filter approach. The narrow applicability of these early filter approaches limit the usage. Our approach doesn't apply any constraints to data statistics or query semantics. It achieves great applicability and generality with high-performance on raw data processing.

3.3 Optimizing Extract-Transform-Load

Recent work [37, 83, 82, 43, 44] improves analytics performance on external raw data through optimizing the Extract-Transform-Load (ETL) process. The key point is to avoid ETL through lazy data loading and caching, rather than accelerating the computation. Databases [37, 83, 82] apply lazy loading, caching materialized results, and positional maps to avoid parsing and data transformation. They store column index within a tuple to shortcut the record parsing. Transformed records are cached for future reuse with extra memory to avoid ETL computations on later queries. More recently, researchers start to explore the

effectiveness of a limited memory cache for loaded data reuse. ReCache [44] is a caching policy optimization using the lazy loading idea. It uses timing measurements of caching operations and selection operators in a query plan to account for the costs of reading, parsing, and caching data in nested and tabular formats. Combining these measurements with information about frequently accessed data fields in the workload, ReCache automatically optimizes the cached data layout. However, using a memory cache to avoid ETL has several limitations. First, there is a significant main memory overhead. A big cache is needed to deliver high cache hit rate. Metadata such as positional index also consumes significant storage spaces because each column requires multiple offsets. Second, the performance is sensitive to query history and pattern. Concurrent or contiguous queries that share few columns can reduce the caching effectiveness. Third, total column footprint is restricted. Lazy loading assumes only few out of many columns are needed; the effectiveness keeps decreasing with the increase of needed columns. Even worse, lazy loading combined with caching usually means a dramatic software architecture change is required in an analytics system. The effort on integrating these techniques into the query execution engine in existing systems is nontrivial. For example, Proteus [82] integrates lazy loading but requires engine customization for a broad set of formats. It generates a specialized query engine for each specific data format. Proteus also requires a big memory cache and positional maps to reuse the transformed raw data. Despite more supported formats, it requires global changes and essentially re-architect the entire database system.

On the other hand, our ACCORDA approach produces a modular software architecture that preserves existing analytical components and has easy integration. By applying ACCORDA, a large part of software investments can be maintained such as query optimization, existing operator implementation, storage hierarchy management, and distributed management. In summary, our approach uses hardware acceleration and can preserve existing system architectures. It achieves high-performance robustly without the main memory overhead and the assumptions about use cases.

3.4 Low-Latency Processing on Fresh Data

As the prevalence and volume of data continues to grow in real-time, low-latency processing on fresh, unloaded, raw data becomes more critical when making instant decisions. Streaming fresh data into data lake is usually the first step for rich data analysis. There are multiple streaming technologies allowing data ingestion and processing in real-time. For example, Flume [75] and Kafka [69] are the two most well-established messaging systems in use today. They both allow connections directly into analytical platforms. Follow-on data processing executes on the ingested data using a variety of tools (e.g. Spark [130]). Since performance is critical for fresh data processing, some systems fuse ingestion and processing together to reduce the latency and accelerate the analysis. Spark can ingest and process data without ever writing to disk with the Spark Streaming library [131]. This functionality is robust, but also compromises the original, unchanged data in raw form, which is the main principle of data lake architectures. Useful information in the original data might be dropped during this process and can't be brought back if users find they are useful during future analysis. Therefore, we generally restrict the ways in which data flows into the system. Ideally, data should be ingested, written to the storage in raw form for flexible analysis. Low-latency processing under such circumstances is challenging because expensive data transformations are required when querying raw data. Our approach addresses this challenge on low-latency streaming processing.

3.5 Data Encodings for Fast Query Execution

Data encoding strongly affects query performance. C-store [112, 34] demonstrates that column-oriented data layouts with rich encodings (e.g., bitmap, dictionary, and run-length-encodings) can dramatically optimize query processing performance on OLAP workloads. Recently, more advanced storage encodings [94, 92, 67, 88] have been proposed to exploit SIMD accelerations for fast query scan. Compressed data storage formats are designed

to allow direct computation without decompression [132, 35] under a limited set of query semantics. Our approach is different. We consider runtime data transformation during query execution without any constraints on query semantics.

Moreover, researchers show that compressed data encodings [60, 90] can accelerate multiple popular iterative machine learning algorithms. Training data is compressed and allows direct computation without decompressing. Our approach opens up more opportunities by allowing multiple co-existed formats and flexible switching across each other during runtime. Zukowski et al. [135] explores dynamic transposition between row and column format during query execution. The result suggests different formats benefit different query operations (e.g. column formats for scanning and row formats for aggregation). However, they only study simple data layouts. Our approach can transform diverse data formats with low cost, and thus is capable of using more powerful data representations during runtime.

Finally, many analytical operators [104, 110, 124, 81, 109] (e.g. sorting, partition, shuffling, aggregation) can get acceleration from using SIMD, GPU and FPGA. However, many methods assume a customized data encoding to leverage these devices. Our approach provides fast data transformation that could enhance these approaches, and enables broader accelerations choices with customized formats.

3.6 Database Hardware Acceleration

ASIC: With the end of Moore's Law and Dennard scaling, data centers are becoming heterogeneous with various accelerators (e.g. ASICs, FPGAs, GPUs) and advanced memory technologies to sustain performance and energy-efficiency improvement [105, 52, 80, 73]. Unsurprisingly, one popular area is high-performance hardware architecture support for data analytics. Several customized designs have been proposed to remove various bottlenecks in query processing [123, 122, 85, 81, 109, 91, 36, 70].

Researchers explore customized hardware accelerators for common operations in SQL such as join [123], partitioning [123, 122, 36], sorting [123], aggregation [123], regex search

[123, 70, 109], and index traversal [85]. However, it is challenging to deploy one accelerator (or function unit) for each algorithm because of the expense of chip design and silicon, obsolete architectures when new algorithms emerge, and sophisticated software maintenance and compiler support.

To our best knowledge, the closet work to our hardware approach is Oracle SparcM7 DAX [91]. It accelerates data transformations in database column scan operations on compressed formats to save memory bandwidth and reduce data movement. The DAX accelerator has multiple hardwired function units, one for each format, to transform formats like Huffman Coding, OZIP, and RLE. However, DAX only supports fixed formats. Our approach differs from the DAX accelerator in that we are designing a general-purpose software-programmable data transformation accelerator. Our proposed accelerator can be wholly reprogrammed and matches the performance and energy efficiency of DAX. Its software programmability enables new acceleration programs can process varied or even future formats.

GPU: Graphic Processing Units (GPUs) are software-programmable and deliver high-performance by exploiting data parallelism. There is an increasing interest in using GPUs for database acceleration on analytical processing [11, 49, 129, 45]. Ample hardware parallelism in GPUs can accelerate common database operations such as select, join, project, and aggregate. It is reported that the speedup can be as high as 40x-70x [49, 45] if data is organized in a GPU-friendly format (e.g. columnar-format for scan). To avoid PCIe bus data transfer overhead, big chunks of query execution, or the entire query, is offloaded to the GPU device. A careful memory management is required to keep the most interesting data in GPU's private memory. In this thesis, we focus on raw data processing, rather than in-machine SQL operations. Data encoding transformation, rather than computation, serves as the limiting factor for query performance on raw data. However, data transformation workloads have poor performance on GPUs. The root cause is the intensive conditional processing with multi-target branches in these workloads. These branches lead to quick thread divergence, which reduce GPUs' execution efficiency [134, 128]. Our accelerator contains high MIMD

parallelism, and its core micro-architecture targets at branch-intensive data transformation workloads in raw data processing.

FPGA: Several recent advances advocate the wide adoption of FPGAs in data centers [105, 52, 78, 109, 81, 77, 95]. Computation kernels are implemented on FPGAs as co-processors. For example, DAnA [95] is a framework for auto-generating FPGA implementations from UDFs for iterative batched machine learning training in RDBMS. Besides machine learning, researchers explore the idea of using FPGAs for regex matching [109], data partitioning [81] and histogram generation [78]. Implementing computation kernels directly on FPGAs brings certain speedups compared to CPUs. However, for interactive queries, adding ad-hoc circuit implementations for each algorithm on FPGA is impractical due to FPGA's scarce application-reserved resources [105]. Accelerating multiple operations on FPGAs in an interactive query requires runtime reconfiguration because these circuit implementations usually don't fit on a single FPGA device. The reconfiguration in interactive query is costly and introduces long latency that destroys the benefit from hardware acceleration. Another drawback is energy efficiency. A proper designed ASIC IP is around 10x more energy efficient than that on FPGAs [121]. Our approach differs from the FPGA approach fundamentally in that we achieve flexibility by carefully designing the ISA for software programmability, and the performance and energy-efficiency is achieved by customizing the hardware architecture.

3.7 Accelerator Hardware System Integration

Motivated by hardware accelerators' energy efficiency and performance, industry vendors, such as Google, Microsoft, Alibaba, Intel, Xilinx, Facebook, IBM, and Nvidia, are using various ways to integrate high-performance hardware accelerators into data center servers. Next, we classify common accelerated platforms according to their memory integration. Traditionally, the most widely used integration is to connect an accelerator (e.g. GPU) to a CPU via PCIe, with both components equipped with private memory. Many GPU and FPGA boards use this style of integration because of its extensibility and convenient

access to hardware boards. The customized Microsoft Catapult board integration is such an example [105, 52]. Moreover, vendors like IBM tend to support a PCIe connection with a coherent shared memory for easier programming. For example, IBM has been developing the Coherent Accelerator Processor Interface (CAPI) [113] for such an integration, and has used this platform in the IBM data engine for NoSQL. More recently, closer integration becomes available using the QuickPath Interconnect (QPI), and provides a coherent shared memory, such as the latest Intel Heterogeneous Architecture Research Platform (HARP) [32] that targets data centers. Memory system integration for these hardware systems are mainly for easy programming. However, data transfer still incurs off-chip data movement, which exacerbates the memory bandwidth wall problem.

On the other hand, on-chip accelerator integration minimized off-chip data movement. If they are integrated into the memory-hierarchy, data transfer between CPU hosts and accelerators are much more efficient because it eliminates the need to copy data across address space and shuttle traffic on-and-off the chip. However, this integration requires accelerators with tiny area and low power. FPGAs and GPUs are usually big chips and consume significant power, which prevents them from using in memory-hierarchy integration. Our approach uses on-chip integration and achieves low-overhead data sharing with CPU hosts.

3.8 Hardware Customization Adoptions

With the end of Moore's Law and Dennard scaling, we are seeing early adoptions in customizing data center hardware platforms for data processing workloads in major Internet companies and cloud providers.

Processor: Oracle develops Sparc M7 CPU processor with on-chip Data Analytics Accelerator (DAX) for in-memory SQL analytical workloads. DAX optimizes in-memory data scanning through memory-speed data decompression. SPARC M7 processor includes 32 cores supporting up to 256 hardware threads. Besides Oracle, Amazon Cloud Service provides A1 cloud compute instances powered by ARM processors it designed in-house. The

customized ARM instances are among the cheapest instance types AWS offers, designed for scale-out cloud workloads that can run across multiple low-power servers.

FPGA: Microsoft initiates the effort on transforming cloud computing by augmenting CPUs with an interconnected and configurable compute layer composed of programmable silicon [105, 52]. The FPGA can act as a local compute accelerator, an inline processor, or a remote accelerator for distributed computing. It sits between the datacenters top-of-rack (ToR) network switches and the servers network interface chip (NIC). As a result, all network traffic is routed through the FPGA device, which can perform line-rate computation on even high-bandwidth network flows. Similarly, Baidu uses FPGAs to accelerate deep learning workloads at scale for performance and cost reduction [102]. For cloud platform provisioning, Amazon starts to provide F1 instances, which use FPGAs to host custom hardware accelerations with diverse development environments: from low-level hardware developers to software developers who are more comfortable with C/C++ and openCL environments.

ASIC: Some companies are even more aggressive on developing customized ASICs for important data processing workloads such as deep learning. Google announces its Tensor Processing Unit [80] with a peak throughput of 92 TeraOps/second (TOPS) and a large (28 MiB) software-managed on-chip memory. Alibaba is developing its own Neural Processing Unit (AliNPU) for AI workloads. Similarly, Amazon offers a machine learning inference chip called AWS Inferentia through AWS to enable users doing inference with lower cost on ASICs rather than GPUs. Besides deep learning workloads, Google is actively looking into video transcoding and security (Titan) hardware acceleration. Accelerators are likely to be built in the future for critical workloads that require extreme performance and efficiency at scale.

3.9 Summary

This chapter presents related work from various aspects, covering software and hardware techniques on optimizing raw data processing, data encodings for fast query execution,

and data analytics hardware accelerators. These descriptions grow the understanding of contribution, uniqueness, and novelty of our approach regarding to the broad research landscape.

CHAPTER 4

PROBLEM AND APPROACH

In this chapter, we begin with the dream of raw data processing and then present the key problem with several common pitfalls. Later, we propose ACCORDA (ACCelerated Operators for Raw Data Analysis), a two-part software and hardware combined approach, to address the problem and these pitfalls. The performance and generality of the ACCORDA approach enable multiple new data lake analysis capabilities.

4.1 The Dream of Raw Data Processing

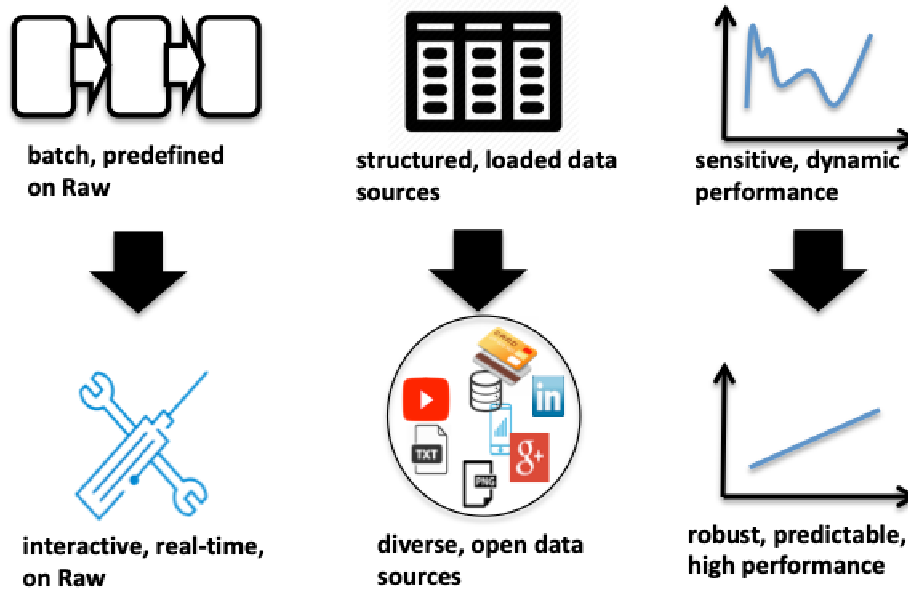


Figure 4.1: Ideal Raw Data Analysis.

The dream of raw data processing is about performance and latency to obtain answers fast on raw data without sacrificing any user flexibility in data analysis. The ever increasing need for faster answers requires interactive and real-time analysis on raw data. Users are empowered to go beyond the structure of the data warehouse to explore data in novel ways and answer their questions at their pace. The sooner the analysis is done, the faster the insights from fresh data can be delivered, and the better business value can be exploited. The

dream is embodied in three new dimensions of analytical capabilities (see Figure 4.1). The first is low-latency interactive query on raw data. Data loading delays - due to scheduling or substantial export-transform-load (ETL) - should be avoided. Second, users have great flexibility to write ad-hoc and customized queries that access any data in the lake, without regarding for it having been loaded. This external raw data is kept for all time so that users can always go back in time to any point to do analysis. It has no fixed schema, so users can employ flexible, open user-defined data types and operations. The uniform fast analytical processing opens up a full diverse space for underlying data sources, such as unstructured web logs, sensor data, social network activity, text and images, and organized columnar data. Third, systems have robust high-performance processing on raw data regardless of query history or patterns. Stateless data processing makes performance robust and irrelevant from implicit system internal states. Even better, the uniform good performance on handling data formats allows systems to exploit user-defined data types with various encodings and layouts, and create them for performance benefits. In summary, high-performance raw data processing is the key factor for the dream.

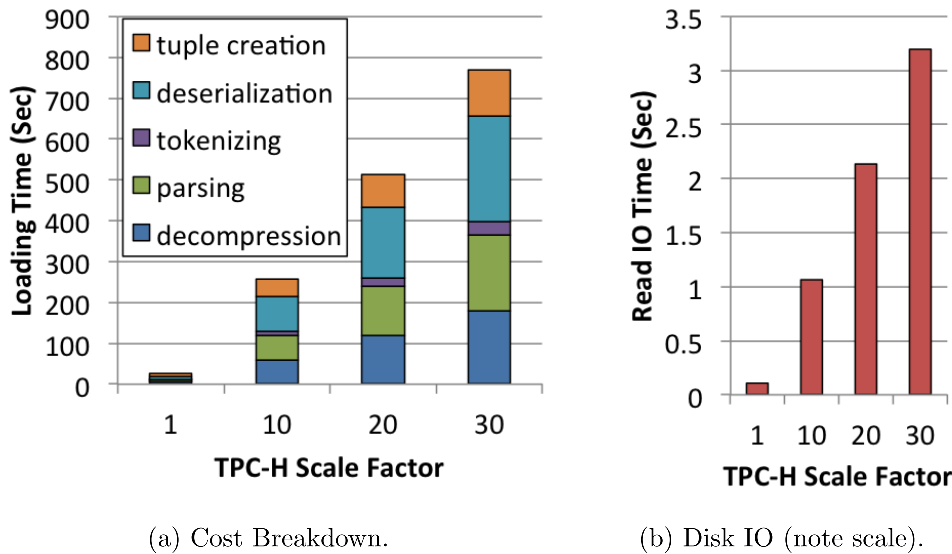
4.2 Problems

In this section, we describe the data lake performance problem that prohibits the dream of raw data processing. Then, we explain two related pitfalls about applying narrowly applicable accelerators and integrating acceleration into database software.

4.2.1 Data Transformation Cost Bottleneck

Data analytics systems deliver poor performance when analyzing dirty, diverse, and ad-hoc raw data. Transforming raw data into internal database formats is costly whether done in batch [99] or just-in-time [37]. For example, Figure 4.2a shows single-threaded costs to load all TPC-H [31] Gzip-compressed CSV files (scale factor from 1 to 30) from SSD into

the PostgreSQL relational database [27] (Intel Core-i7 CPU with 250GB SATA 3.0 SSD). This common extract-transform-load (ETL) task includes decompression, parsing record delimiters, tokenizing attribute values, and deserialization. It requires nearly 800 seconds for scale factor 30 (about 30GB uncompressed), dominating time to initial analysis [37] and in this example, the load time is 200x larger than the corresponding disk IO (>99.5% CPU time), see Figure 4.2b.



(a) Cost Breakdown.

(b) Disk IO (note scale).

Figure 4.2: High Load Costs: Compressed CSV into PostgreSQL.

Traditional CPUs suffer from poor performance on data transformation workloads. The two reasons for this are poor support for sub-word, variable-length data and unpredictable control flow. Modern CPU micro-architectures require dozens to hundreds of instructions in flight to achieve full performance, and achieving that level of instruction-level parallelism requires correct branch prediction over a dozen or more branch instructions. Section 5.3 performs an in-depth evaluation of branch impact on the performance; we further show that the fraction of CPU execution cycle-lost because of branch misprediction consumes 32% to 86% of total cycles. In summary, expensive data transformation limits both the application flexibility and the system capabilities to optimize encodings for performance. Slow data transformation is the key barrier that limits fast raw data processing.

4.2.2 Narrowly Applicable Accelerator Pitfall

Traditional CPU architectures pose fundamental limitations on performance and energy-efficiency as described earlier. Improving hardware data processing performance with accelerators is nontrivial because it requires a general architecture that supports a range of computations with high energy efficiency and low silicon cost.

One common pitfall is to deploy one accelerator for each application (task). This practice leads to an accelerator sea. However, this solution is impractical because the chip design and the silicon cost are expensive, the architecture becomes obsolete when new algorithms emerge, and the sophisticated hardware and software interface is a disaster for software maintenance and compiler support. However, most of the customized data processing accelerators fall into this category and are constrained by their narrow acceleration capability [123, 122, 85, 78, 81, 109, 91]. The runtime library management for these accelerators is complicated because some platforms may only carry a subset of these accelerators, and some lack any at all. Applications need to adapt when accelerator libraries are added, removed or modified. It becomes further complicated when these libraries themselves have cross-dependency. Narrow accelerators make future platforms extremely complex to manage and maintain. Acceleration narrowness is one of the major computer architecture problems in the post Moore's Law era. The resulted complication of accelerator deployment in both hardware and software negatively impacts the adoption of accelerators in future computing systems.

4.2.3 Database Software Pitfall

Accommodating raw data acceleration into existing analytics systems should preserve original software architecture, such as front-end, execution engine, and query optimizer; resulted changes should not affect the analytics system architecture. Low-level acceleration detail must be hidden, but a proper abstraction is required to enable explicit query optimization. Accelerated architectures should maintain query optimization flexibility and runtime

uniformity.

However, a common pitfall after introducing hardware acceleration is the disruption on existing database software architecture. Traditional hardware-accelerated databases (e.g. GPU databases) add additional layers in a system with great complexity. These modifications from acceleration disrupt database software architecture as Figure 4.3 shown. The query optimizer must consider device scheduling and data placement strategies for performance. The expensive data movement between accelerators and CPU cores limits query optimization opportunity and efficiency. A query optimizer usually splits a query plan in coarse-grained chunks and optimizes each chunk locally. Moreover, the added acceleration destroys the uniform runtime view and breaks the execution model in the query engine. For example, iterator-based execution can't be preserved after adding GPUs for acceleration. The execution model used for GPU in query processing is usually operator-at-a-time [11], which computes the entire column(s) for one operator. On the other hand, iterator-based tuple-at-a-time execution is commonly used in CPUs for efficient lazy computation and small intermediate results. Adding heavy-weight accelerators like GPUs forces the system to maintain separate execution models and add special optimizations for accelerator asynchronous processing. The resulted heterogeneous runtime is hard to manage. Accelerator integration brings significant disruptions into database software architecture.

Therefore, considering all the above facts, the desired solution requires 1) high performance with low cost in power and area, 2) acceleration generality across a wide range of applications, and 3) software integration that preserves database architectures and their properties such as flexible query optimization and uniform runtime.

4.3 The ACCORDA Approach

To address the problem and its related pitfalls, we present ACCORDA (ACCelerated Operators for Raw Data Analysis), a software and hardware architecture for data analytics systems on raw data. In Figure 4.4, we illustrate how ACCORDA fits into the landscape of analytics systems,

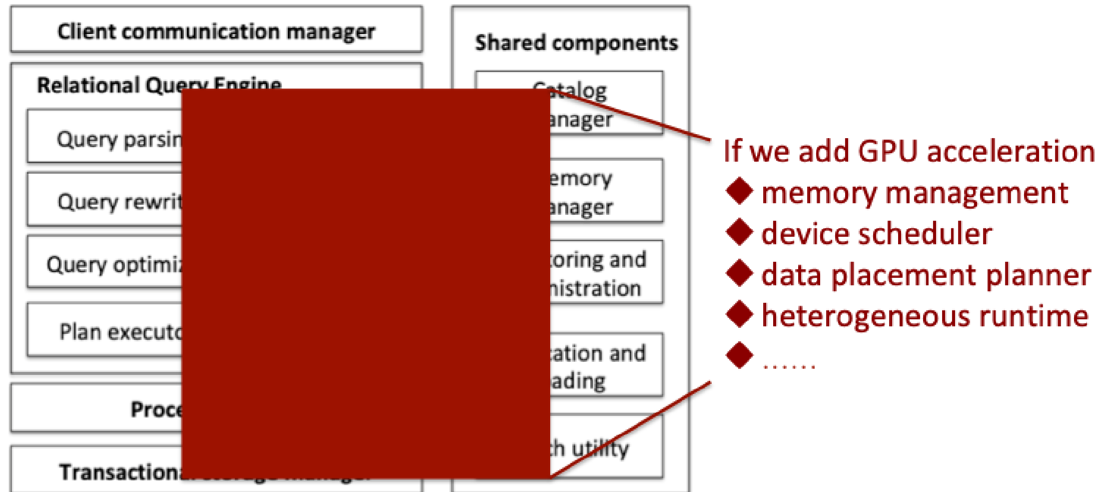


Figure 4.3: Hardware Acceleration Disrupts Database Software Architecture.

in two dimensions – batch/on-demand loading, and accelerated/non-accelerated. Conventional databases (lower left) batch load raw data, and run on CPUs. Data transformation is expensive on CPUs, consuming significant resources. Conventional accelerated database systems (upper left), use discrete GPU's or FPGA's, focusing on compute acceleration and also suffering from high cost to access acceleration. Recently, several systems have focused on raw data processing, loading raw data on-demand and using CPUs (lower right). They include RAW [37, 83, 82, 44] that loads on demand, and then caches loaded data in memory lazily to speedup query execution. Spark [43] executes on raw data but suffers from the low parsing and deserialization performance. Sparser [103] applies a narrow class of fast filters before transforming to reduce raw data processing cost.

The ACCORDA system (upper right), combines on-demand loading with seamless acceleration focused on data transformation, providing predictable high-performance on raw data processing. Key ideas include explicit representation of encoding types in a query plan, fast data transformation, in memory-hierarchy acceleration, and novel query optimizations based on encodings. In Chapter 5, we describe the Unified Transformation Accelerator (UTA) approach with a detailed discussion of the hardware acceleration on data transformation and in memory-hierarchy integration. Section 4.3.1 presents a high-level description of that. In

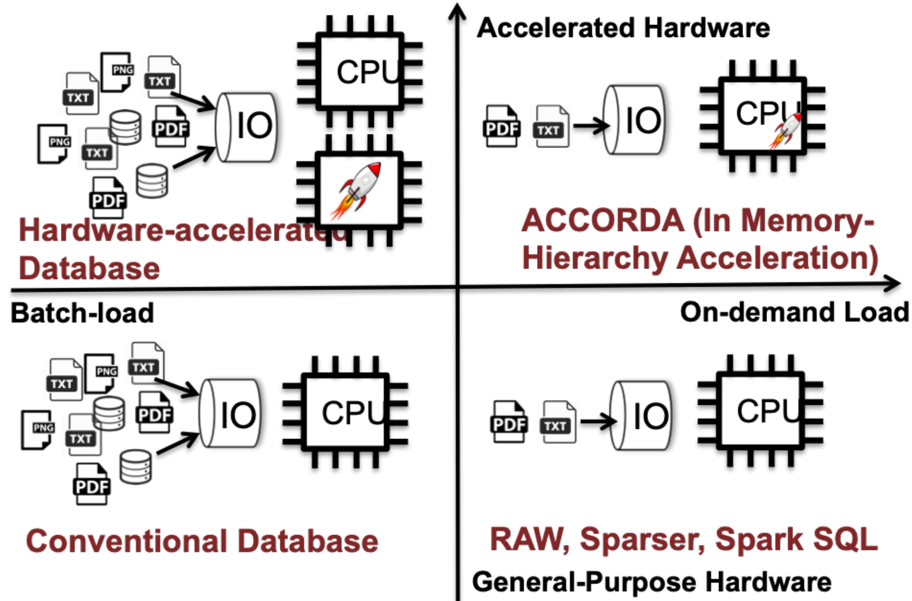


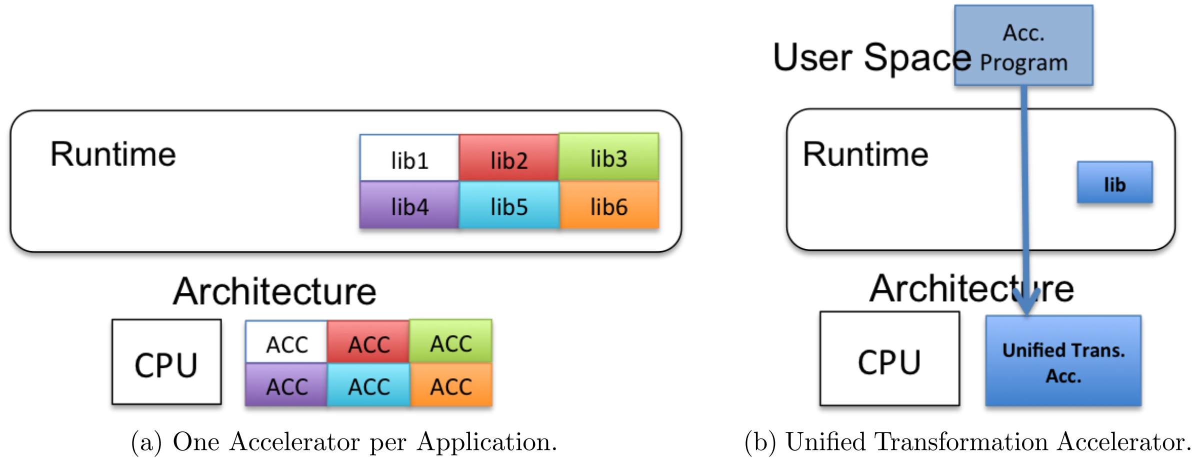
Figure 4.4: Approaches for Raw Data analysis.

Chapter 6, we describe the Accelerated Transformation Operators (ATO) which serves as the ACCORDA's software architecture that enables encoding-based optimizations and seamless accelerator integration. A brief discussion of the ATO approach can be found in Section 4.3.2.

By applying ACCORDA, the combined benefits of UTA and ATO provide a system with fast data filtering, extraction, and transformation while maintaining flexible query optimization, preserving existing software architectures, and unlocking new capabilities for data analysis that are presented in Section 4.1.

4.3.1 Unified Transformation Accelerator

The Unified Transformation Accelerator (UTA) creates new and flexible architecture support for data transformations in common analytical workloads. We carefully apply the hardware architecture customization principle [54] to make the accelerator not only efficient but also general and software-programmable. The accelerator exploits sufficient MIMD parallelism for performance (throughput), and software-controlled scratchpad memory to minimize the power consumption. We carefully customize the ISA for data transformation generality and high efficiency. Figure 4.5 contrasts a traditional SoC approach with our UTA approach. The



(a) One Accelerator per Application. (b) Unified Transformation Accelerator.

Figure 4.5: Hardware Acceleration Approaches for Data Transformation.

UTA approach (Figure 4.5b) has a simpler chip architecture and is extensible by software applications. The UTA approach targets at supporting data encoding transformation, rather than narrow computations or operator accelerations that are common in traditional accelerator design. We further push the acceleration generality by letting applications freely explore data formats to improve performance. Many researchers (Section 3.5) have shown that by using advantageous data encodings tailored to the application, performance can have orders of magnitude speedups. Therefore, great acceleration generality comes along with using beneficial data encodings flexibly. UTA accelerates the process of switching data encoding from one to another so that applications can use it freely without worrying about the cost. Accelerating data transformation for the aforementioned purpose is the key reason why UTA avoids the narrow acceleration pitfall with a simple chip architecture. In contrast, many hardwired fixed accelerators are integrated together to support broad applications (Figure 4.5a). As mentioned before, the runtime library management for these accelerators is complex. Cross-dependency further complicates the machine deployment. On the other hand, Unified Transformation Accelerator (Figure 4.5b) simplifies acceleration library management, and supports various applications by writing new programs. UTA provides fast data transformation for tasks that are critical in real-time interactive raw data processing. Detailed description of the UTA approach is presented in Chapter 5.

4.3.2 Accelerated Transformation Operators

The Accelerated Transformation Operators (ATO) is a software architecture for integrating hardware accelerations into data analytical systems. ATO integrates acceleration into query execution by introducing hardware accelerated transformation operators with a cost model or a set of optimization rules. Legacy operators can be accelerated and new operators can be created with software implementations that use accelerators. The ATO approach exposes hardware accelerators to the query optimizer. Figure 4.6 shows the ATO approach (right) and compares it to other alternatives. The software-accelerated batch ETL approach (left) applies advanced commodity hardware features (e.g. SIMD) in the batch ETL loading routines. The lazy loading approach (middle) modifies the entire query engine to avoid batch ETL process and only transforms required data. It tries to avoid parsing and deserialization by caching intermediate results. Queries can reuse the cached data to reduce loading overhead. The ATO approach (right) preserves database software architecture by extending operator interface with encodings and using a uniform worker model. Fast data transformation from UTA advocates using beneficial formats for operator implementations to speed up query processing. Data types on query edges are extended to enable encoding-specific operators. We capture raw data structures in the encoding-extended query interface to explore aggressive query optimization.

When compared to the lazy loading and memory caching approach [37, 83, 82, 43, 44], ATO's effectiveness doesn't depend on query history and system's internal state. It requires significantly less memory cost and no disruptive changes to the query engine. When compared to the raw data parsing and filtering acceleration approaches [98, 93, 103, 43], ATO leverages a general data transformation accelerator (UTA), rather than SIMD unit, to process various data formats with all kinds of filtering conditions and semantics. When compared to customized data encodings in query execution [112, 132, 35, 94, 135], ATO paves the way for dynamic encoding transformation during runtime, rather than static upfront transformation, and opens up new opportunities in query optimization. In short, the ATO approach solves the software

architecture disruption problem and enables dramatic new query optimizations across data encodings in a query plan. With accelerator integration, ATO provides low-latency time-to-first-answer with robust, predictable high-performance, and saves IO and CPU resources via on-demand data processing. The flexible software architecture integration enables UTA's deployment in data lake systems. Detailed description of the ATO approach is presented in Chapter 6.

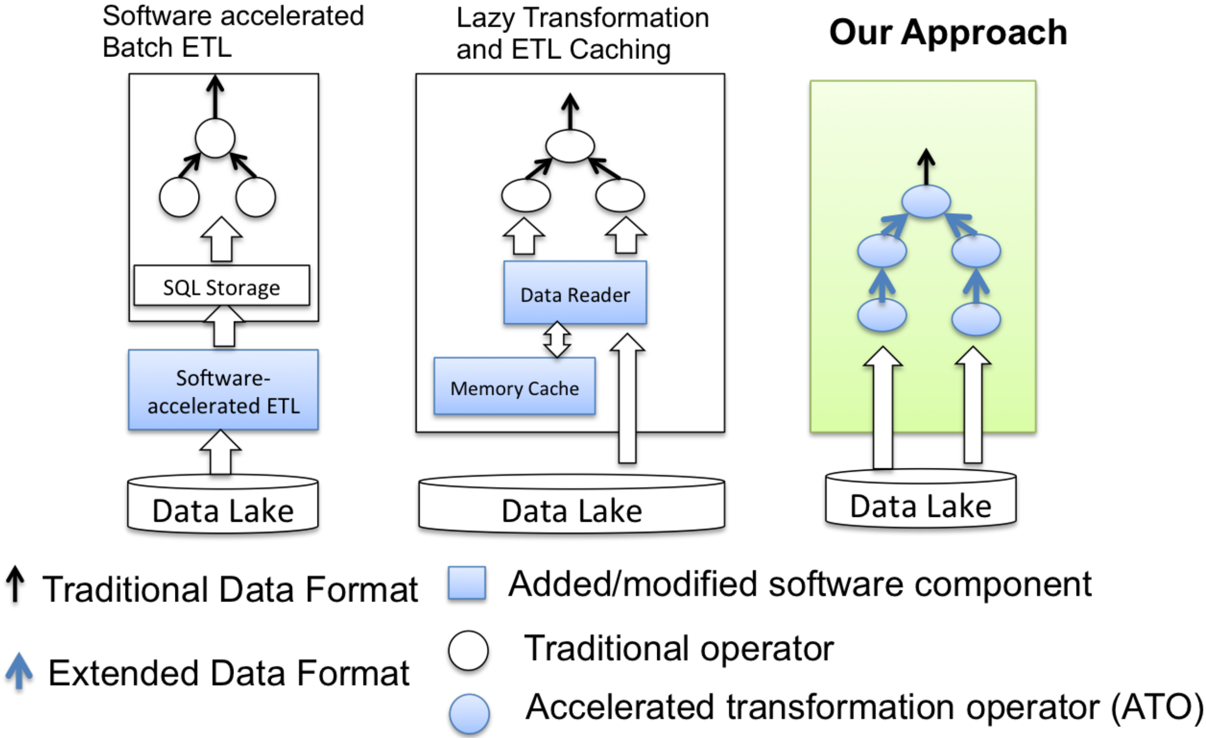


Figure 4.6: Approaches for Accelerating Raw Data Processing.

4.4 Summary

This chapter portrays the dream of raw data processing and discusses the key problem and its related pitfalls. We then briefly explain our proposed ACCORDA approach. The ACCORDA approach can be further broken down into two sub-approaches: the Unified Transformation Accelerator (UTA) and the Accelerated Transformation Operators (ATO). The details of these two approaches are discussed in the following two chapters.

CHAPTER 5

UNIFIED TRANSFORMATION ACCELERATOR

In this chapter, we propose the Unified Transformation Accelerator (UTA) approach. UTA serves as the ACCORDA hardware architecture, satisfying the performance requirements of ATO (Chapter 6) for data filtering, extraction, and transformation in analytical workloads. In Section 5.1, we first discuss the functional, performance, and cost requirements for the Unified Transformation Accelerator. Next, Section 5.2 explains the accelerator integration approach with minimal data movement overhead in a conventional hardware system. We further demonstrate the feasibility of UTA by designing and evaluating a concrete hardware instance called the Unstructured Data Processor [66]. Section 5.5 includes additional related work that is not covered in Chapter 3. Finally, Section 5.6 summarizes the chapter.

5.1 UTA Requirements

5.1.1 Functional Requirements

For the Unified Transformation Accelerator (UTA) approach, we need a deep understanding of workloads before designing a concrete architecture instance. Data recoding tasks play an important role in raw data processing with various formats. They transform data from one format to another during execution for performance. For example, source data is decompressed from the storage block, parsed and extracted for desired values, and deserialized and transposed for column scan. We identified and collected a set of typical data transformation workloads from the real-world data analytical tasks as the benchmark. The performance study includes a coarse-grained runtime profiling to understand where the time goes using a careful analysis on reading the external data. After pin-pointing the major time-consuming parts, we classify them into multiple representative computation kernels with extensive coverage as shown in Figure 5.1, ranging from (de)compression, parsing, extraction, encoding, deserialization,

transposing, and pattern matching. These isolated kernels then serve as the micro-benchmarks to allow a fine-grained architecture study.

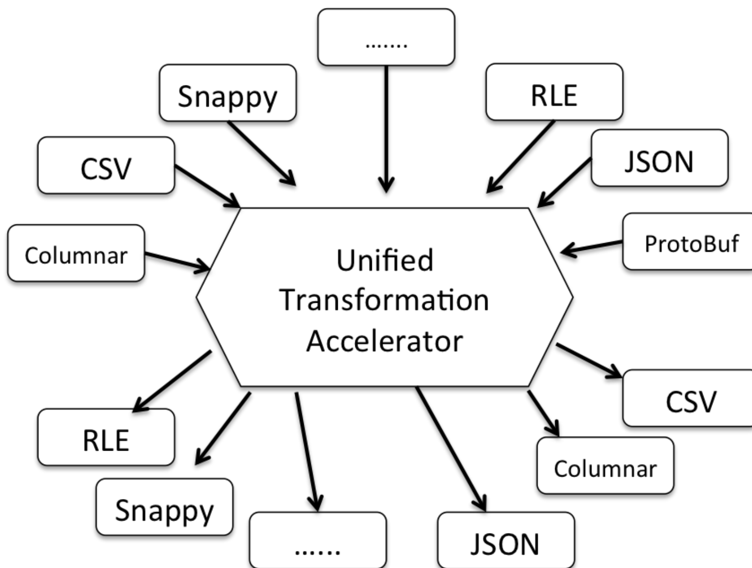


Figure 5.1: Acceleration Functional Requirements.

5.1.2 Performance Requirements

We have a strict performance requirement on UTA. Though UTA incorporates the software programmable principle, it should come with negligible loss of performance with increased generality. The ISA and micro-architecture features should be carefully designed to match the efficiency of ASICs on a diverse set of data transformation workloads. In particular, we expect a single UTA to saturate a DDR3 memory channel. Therefore, the throughput of a UTA should be around 10GB/s to 20GB/s for data transformation workloads such as decompression, encoding, parsing, and extraction. The rationale behind is that the UTA performance is bounded by the memory IO bandwidth.

5.1.3 Cost Requirements

The Unified Transformation Accelerator fulfills the functional and performance requirement by embracing software-programmable and high-performance data recoding. However, it is

critical to achieve these goals with low hardware resources and power budget. If the UTA design takes as much resources as a gigantic GPU chip, which usually takes $> 500mm^2$ silicon area and consumes 100W power, the UTA design can never be integrated into the same chip with CPU hosts. The resulted separate memory system prevents low-overhead data sharing across UTA and CPU hosts. Therefore, to have flexible software execution, UTA should sit on the same die with CPU cores and is integrated into the memory system to reduce the data movement overhead. The expected area cost of UTA should be around the area of a conventional x86 core with L1, which leads to <5% of a core with associated L1/L2/L3 caches. So it consumes 1% the area of a 4-core Xeon chip area, and in a modern system perhaps 0.13% of a commodity 32-core chip. For the power consumption, a single UTA instance should be under 1W when operating on peak performance to keep the overall power increase of the entire system as low as 1%.

5.2 In Memory-Hierarchy Integration

Traditional accelerated hardware systems use PCIe integration for accelerators. GPU is a popular accelerator for parallel computation in databases [45, 11]. The accelerator is attached to the host via a PCIe bus, as Figure 5.2 shown on the left. The massive parallelism in GPU requires significant memory bandwidth. A typical GPU-accelerated platform keeps two separate memory system: one for the CPU host and the other for the accelerator. The memory traffic isolation prevents the severe performance interferences from each other. This integration approach comes at the expensive cost of data transfer between the accelerator and the host. Data sharing is achieved by explicitly copying the data across two address spaces. Data is forced to go off the chip to DRAM for DMA execution. The resulted data movement overhead prevents fine-grained interleaved execution on the accelerator. On the other hand, we use a different accelerator integration approach as the right figure in Figure 5.2. Both CPUs and accelerators sit on the same die and the data transferring cost is minimized by in memory-hierarchy integration.

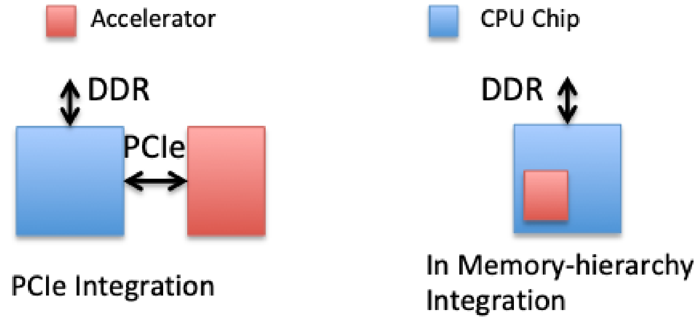


Figure 5.2: Two Accelerator Integration Approaches.

Figure 5.3 illustrates the UTA's integration into CPU's memory hierarchy. The CPU has normal access to caches, but can also access the UTA's local memory directly. The UTA local memory (blue in Figure 5.3) is mapped onto the CPU's address space as uncacheable (data won't appear in the cache memory hierarchy) as shown in Figure 5.4. When data is recoded into this UTA memory space, the library routine initiates lightweight DMA operations (like memcpy) that transfer blocks of data from the DRAM to the UTA memory with high efficiency. The DMA engine [115] acts as a traditional L2 agent to communicate with the LLC controller. The green lines in Figure 5.3 show the idea. This is very different from the memory integration in GPUs and PCIe-attached FPGA accelerators, which maintains separate address space and suffers from expensive off-chip data copy across address space.

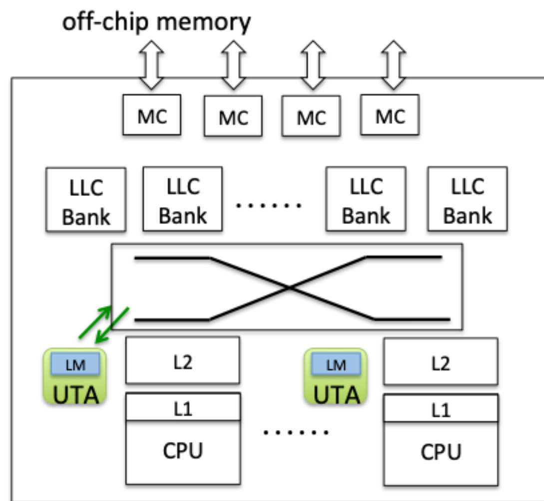


Figure 5.3: In Memory-Hierarchy Integration of UTA into NoC fabric

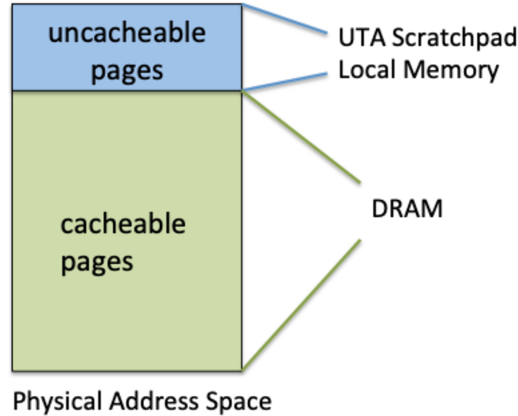


Figure 5.4: UTA Local Memory is Exposed as Part of CPU Address Space.

5.3 UTA Architecture: The Unstructured Data Processor

5.3.1 Overview

We propose an accelerator, the Unstructured Data Processor (UDP), [66] to demonstrate the potential performance and power of the UTA concept. UDP can be used to accelerate extract-transform-load and data transformation programs with the in memory-hierarchy integration (Section 5.2). UDP has flexible data sharing with CPUs. It is designed to deliver equal or higher performance at dramatically lower power, and to 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 5.1 and Section 4.2.1) producing poor performance and efficiency. UDP fulfills the UTA functional requirement via software programmability with a well-defined instruction set. UDP achieves significant performance via MIMD parallelism. The UDP has 64-parallel lanes (Figure 5.5a), 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. The cost requirement of

UTA is addressed in UDP by customizing the lane micro-architecture and specializing the memory system for low latency and low power. The specialization approach brings significant efficiency to UDP thus reducing the cost of area and power.

5.3.2 Key UDP Architecture Design Features

We outline several key architecture design features including UDP lane architecture support:

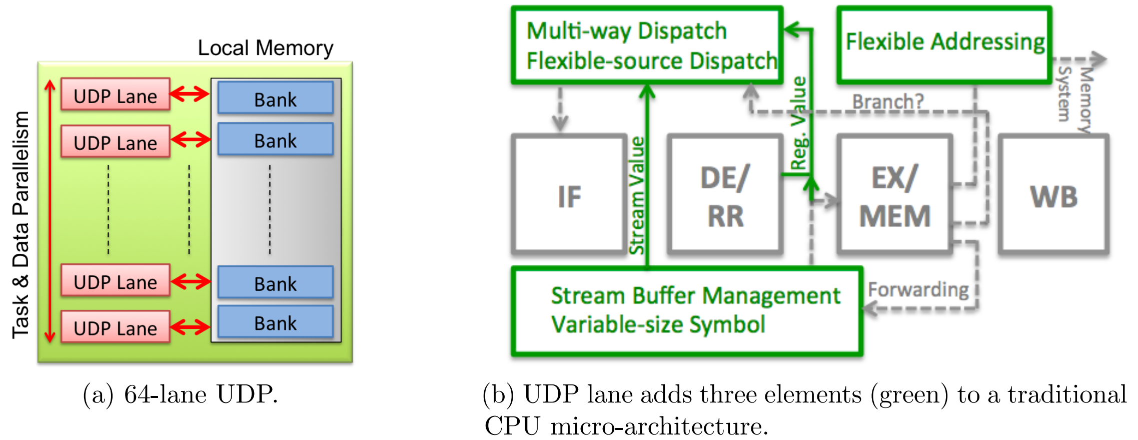
- multi-way dispatch,
- variable-size symbols, and
- dispatch from varied sources.

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 [61] 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 5.5a. 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 5.5b. Additional front-end functionality supports multi-way dispatch and flexible-source dispatch, requiring connection to the register file and stream buffer. Additional forwarding



(a) 64-lane UDP.

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

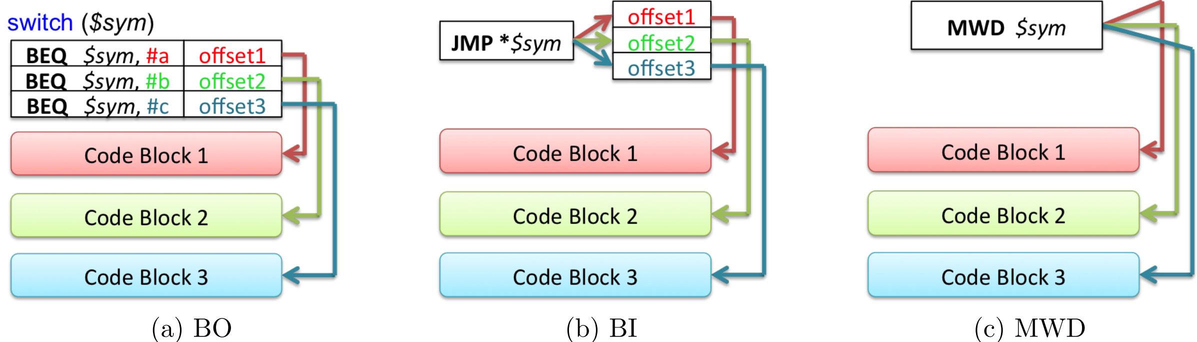
Figure 5.5: UDP and UDP Lane Micro-architecture.

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 to generate UDP programs is discussed in Section 5.4.1.

5.3.3 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.

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 5.6a), or compute an entry in a dispatch table full of targets, and then *branch-indirect* (BI) to that target (Figure 5.6b). 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



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

Figure 5.6: Branch Execution on Symbol: Branch Offset (BO), Branch Indirect (BI), Multi-way Dispatch (MWD).

the latter, the BI operation often suffers BTB (branch-target-buffer) misses, hampering performance as well.

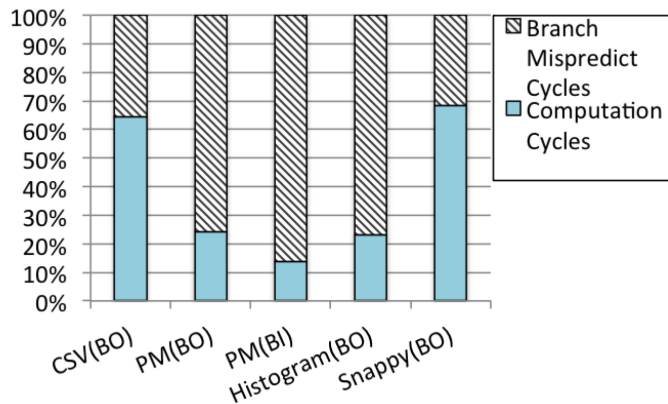


Figure 5.7: Cycles spent on Branch Misprediction and Computation.

To illustrate the problem, we studied several ETL kernels drawn from the larger set described in Table 5.1. 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 5.7). These penalties are typical of many ETL and data transformation workloads, as documented in Table 5.1.

The UDP's multi-way dispatch (see Figure 5.6c) selects efficiently from multiple targets by using the symbol (or any value) as a dynamic offset. Compared to BO, multi-way dispatch

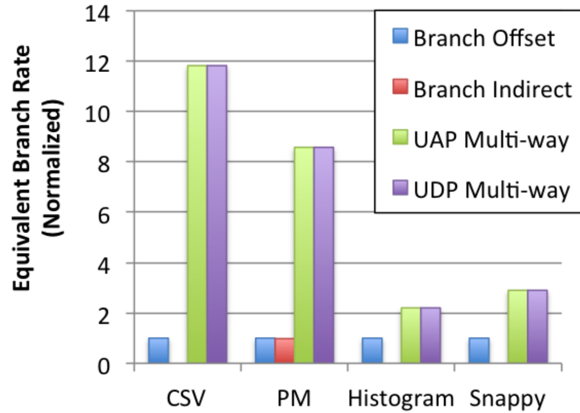


Figure 5.8: Relative Branch Rate for ETL kernels using BO, BI and Multi-way.

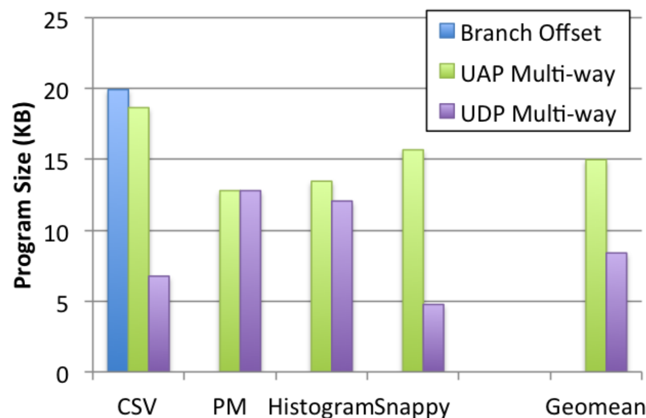


Figure 5.9: Code size for varied branch and dispatch approaches.

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 5.4.1) 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 5.10), and creates an optimized memory layout using the Efficient Coupled Linear Packing (EffCLiP) algorithm [64] that resolves the coupled code block placement constraints. A great help in this is UDP's *signature* mechanism 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.

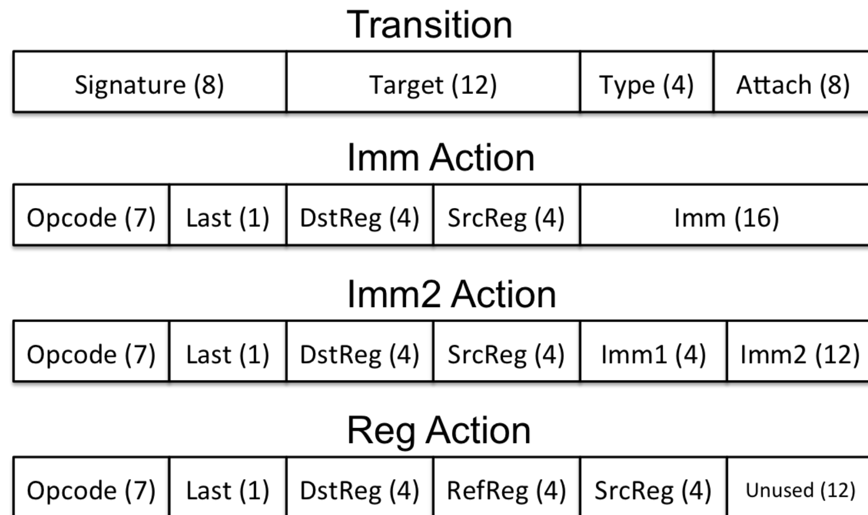


Figure 5.10: UDP Transition and Action Formats: Imm Action, Imm2 Action, Reg Action. All are 32-bits.

UDP machine encodings are summarized in Figure 5.10. 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.

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 5.8). 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.

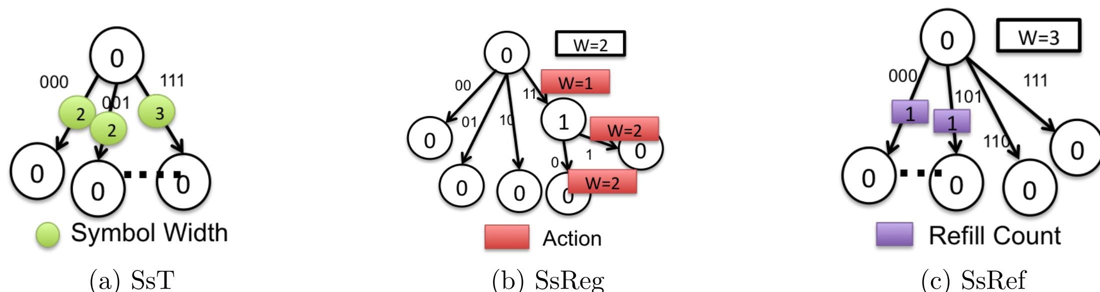
UDP's multi-way dispatch includes a significant improvement over the UAP's [63] (UDP's predecessor). Memory in accelerators is always in high-demand, and in the UDP, code size competes directly with lane parallelism, and thus performance. Both UDP and UAP [63] use *attach* to address action blocks. To 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 5.9).

Overall, UDP employs seven transitions implementing variants of multi-way dispatch: *labeled* [63], *majority* [63], *default* [63], *epsilon* [63], *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 $|\Sigma|$ *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 and *refill* that enables efficient variable-size symbol execution.

Variable-size symbols 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 5.11.



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

Figure 5.11: Huffman Decoding Tree: *00,01,10,110,111*. Solid box is symbol-size register. Other actions not shown.

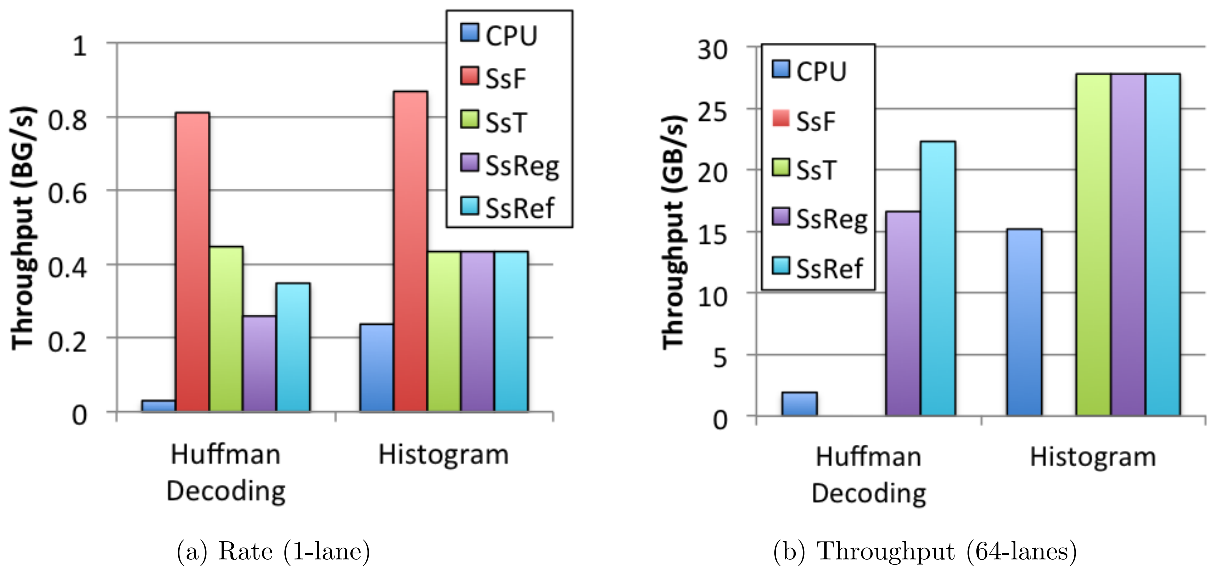
1. **Symbol-size Fixed (SsF)** hardwired dispatch width. For example, the UAP [63] has fixed 8-bit dispatch with (character symbols), achieving best performance and efficiency for regular expression matching. Applications requiring variable-size symbols (e.g. Huffman decoding) must adapt by unrolling, 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 5.11a). 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 5.4.3) 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 5.11b).

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 5.11c) 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 5.12: Variable-size Symbol Approaches on kernels requiring dynamic and static variability.

In Figure 5.12, 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 5.12a). 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 5.12b). 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).

Flexible dispatch sources is the third feature. 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 5.4.

Stream Buffer constructs streams from vector registers, extending the vector instruction set (e.g. AVX,NEON[14, 24]). 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 [63]. For simplicity, the current UDP design restricts the source to Register 0.

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 5.13). Compared to an 8-thread CPU (Section 5.4) using the rest of the ETL kernels that we have not used in the prior two architecture comparisons. Adding scalar dispatch

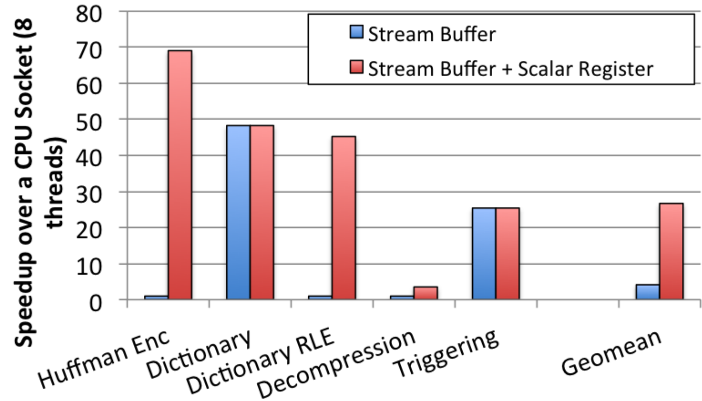


Figure 5.13: Performance Benefit for adding stream buffer and scalar register as UDP dispatch source.

enables coverage of a much broader application domain, dramatically improving the geometric mean speedup.

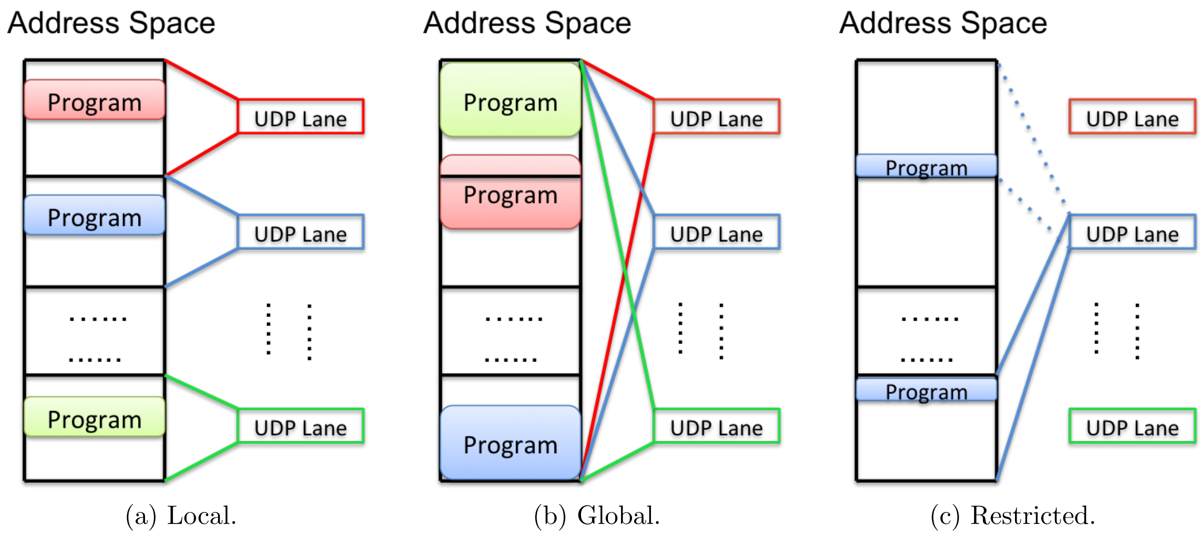


Figure 5.14: Addressing Models. Local: each lane has private address space; Global: each lane shares entire address space; Restricted: each lane flexibly chooses a window.

Flexible addressing for data-parallelism and memory utilization 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 5.14).

Each UDP lane is a 32-bit execution engine, that generates 12-bit word addresses from the *target* field for dispatch targets (Figure 5.10). 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 5.15). Figure 5.16 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" [46] 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),

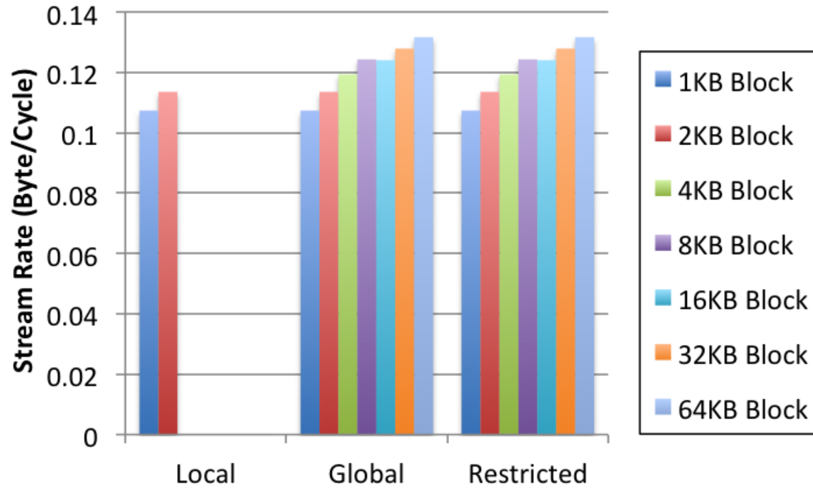


Figure 5.15: Addressing Impacts Performance.

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 5.17 displays memory reference energy for 1MB memory (64 read ports and 64 write ports) modeled using CACTI 6.5 [4]. 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.

Other key features in UDP reduces 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 or 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.

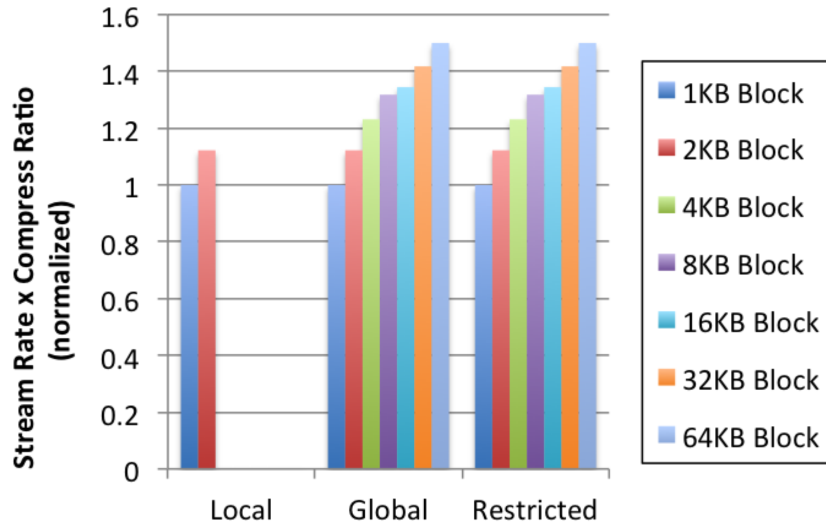


Figure 5.16: Addressing Impacts More Than Performance.

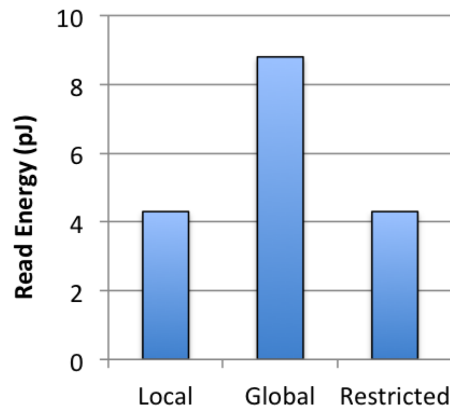


Figure 5.17: Memory Reference Energy.

5.4 UTA Evaluation

We start with the description of experiment methodology followed by the performance evaluation of a detailed UTA accelerator design (UDP) on a broad range of ETL workloads. Next, we give a resource analysis on key hardware metrics of the accelerator silicon implementation. Finally, we study the performance benefit of applying UDP in query acceleration on raw data using performance-critical ACCORDA micro-benchmarks.

5.4.1 Methodology

This section first explains the programming toolchain we created for UTA evaluation on various data transformation tasks. We then describe the selected ETL workloads, metrics, and hardware system configurations used for the experiments.

Programming the UTA Accelerator

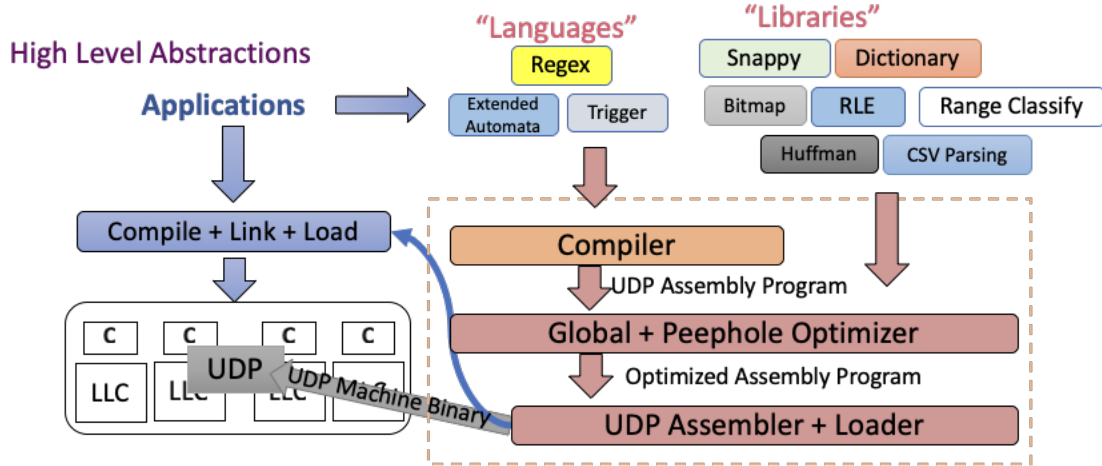


Figure 5.18: UDP's software stack supports a wide range of transformations. Traditional CPU and UDP computation can be integrated flexibly.

We use the UDP accelerator to demonstrate the programming flow of UTA-based accelerators. Briefly speaking, a number of domain-specific translators and a shared backend (see Figure 5.18) are used to create the UDP programs for application kernel evaluations in Section 5.4.2. The translators support 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.

To be specific, we implemented the entire backend compilation stack shown in Figure 5.18 in Python. The backend takes UDP programs written in an intermediate representation, called Extended-Finite Automaton (EFA). EFA is a directed graph data structure with each

transition (edge) optionally associated with one or more actions. Each action and transition (edge) represents one UDP operation respectively, which directly maps to the unique UDP assembly instruction. We documented the UDP instruction set [61] in detail. Domain-specific languages such as Regex and Triggering, are external to the backend infrastructure. These high-level languages are separately compiled into the EFA representation with their own domain-specific frontend compilers. The backend has a few build-in compilation supports for languages as Regex. However, it is an open research question on designing a high-level language to capture the execution model for UDP-like accelerators. The RAPID programming language [42] is a recent effort in this direction. Apart from natively supported languages, users are free to add new languages with their own frontend compilers (e.g. lex and yacc) to generate UDP programs in EFA format. We also built a set of data transformation libraries written in EFA format. We provide a tool called UDP program composer to connect multiple program modules together to form a larger EFA program that can handle multiple tasks sequentially or concurrently. UDP programmers can use python APIs such as *efa1.state1.connect(efa2.state2, symbol)* to link two EFAs using *symbol* as the label of the "bridge" transition between *state1* and *state2*. The composer is optional to the backend toolchain but helps to develop a decoupled modular view of UDP programs that eases the programming process for UDP users. Next, those layers discussed below are essential for compiling the EFAs down to binary UDP machine images.

The first layer in the backend is a peephole optimizer. Different from traditional optimizer that mainly targets for performance, the major responsibility of the peephole optimizer is to shrink the program size because the scratchpad memory in UDP is a scarce but critical resource for parallel performance. Conceptually, the peephole optimizer traverses the EFA graph and does pattern matching on local sub-graph parts, transforming matched structures with less transitions (edges). It reduces the transitions (edges) in an EFA via assembly instruction replacement. In the UDP assembly instruction set, a few transition-instructions are designed to represent a group of transitions. The optimizer shrinks the program size

wherever is possible. After the transformation, the UDP program is still in the EFA format but enjoys smaller size.

The assembler translates UDP assembly instructions directly into UDP machine instructions in the next layer. This translation is straight-forward for UDP actions. Each action in assembly is essentially the same as the one in machine-level instruction. Thus only a simple one-one mapping is required for the translation. However, the translation for transitions is very tricky. In the assembly, each EFA transition has a corresponding instruction, which is in the form of *(destination, source, symbol, type)*. The assembly instruction reflects the abstraction of a transition edge under a graphical view. However, the machine implements the assembly-level transition via EFA state properties. EFA transitions are logical and only exist in the assembly layer. Each EFA state (node) has a property that determines its out-going transition(s) type(s). For example, in the machine-level, if state A has the "flag" property, then all fan-out transitions of state A in the assembly level are *flag* transitions. During the compilation, the assembler assigns the state property according to the transition assembly to each state, and back propagates property information for complex transition implementations. After transitions and actions are all translated from assembly to machine instructions, the assembly phase completes.

The final loader layer determines the code layout in the UDP memory, producing the binary image. This process is effectively the same as the loader's role in an operating system. However, the Multi-way Dispatch feature (Section 5.3.3) requires the explicit control over the code placement in the UDP memory address space. Multi-way Dispatch achieves significant branching efficiency by directly computing the code block address. The UDP loader first places machine-level instructions in the linear address space and then packs located instructions into a denser smaller memory range. The packing process requires re-ordering of machine instructions with constraints on relative instruction location distance in the address space. The packing procedure is a greedy algorithm that runs in $O(N)$ time complexity. The detailed algorithm can be found in the public technical report [64].

After multiple code transformations in the backend infrastructure, a UDP program in EFA representation is converted into a binary memory image that can be directly copied into the UDP local scratchpad memory for execution.

Workloads and Metrics

Application	**Workload**	**CPU Challenge**
CSV Parsing	Crimes, NYC Taxi Trip [25], Food Inspection [8]	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 [117]	Poor locality, 1.6x L1 miss rate
Dictionary	Crimes [7]	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 [2]	15x branch mispredicts
Decompression (Snappy)	Canterbury Corpus, Berkeley Big Data	15x branch mispredicts
Signal Triggering	Keysight Scope Trace [62]	mem indirect, address, condl, 9 cycles

Table 5.1: Data Transformation Workloads

We selected a diverse set of kernels drawn from broader ETL and data transformation tasks in real analytical applications to provide sufficient coverage on the functional requirements raised in Section 5.1.1. The benchmarks are designed to exercise the architectural performance bottlenecks in conventional CPUs, and at the same time, their functions are desired for data analytical tasks.

CSV parsing involves finding delimiters, fields, and row and column structure, and copying field into the system. The CPU code is from libcsv [21]; these measurements use Crimes (128MB) [7], Trip (128MB) [25] and Food Inspection (16MB) [8] 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 [2]. 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 [117], with the CPU code as Boost C++ Regex [3]. 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 [86] and NFA [76] models.

Compression CPU code is the Snappy [10] 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 [1]. Dictionary measurements use *Arrest, District,* and *Location Description* attributes of Crime [7]. Dictionary-RLE adds a run-length encoding phase. UDP program performs encoding, using a defined dictionary.

Histogram CPU code is the GSL Histogram library [9]. Measurements use *Crimes.Latitude*, *Crimes.Longitude,* and *Taxi.Fare* with 10, 10, and 4 bins of IEEE FP values [13]. 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 [62], at 4 symbols per lookup. Trace is proprietary from Keysight oscilloscope. UDP implements exactly the same automaton.

Table 5.1 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 [26] 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.

With the above workloads, we use the following metrics to evaluate the performance and cost of the UDP.

Metric	**Description**
Rate (Megabytes/s)	Input processing speed for a single stream or 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 5.2)
TPut/power (MB/s/watt)	Power efficiency

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 5.4.3.

In Section 5.4.2, for each kernel we compare achievable rate for one UDP lane to one Xeon E5620 CPU thread [18]. 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.

5.4.2 UDP Performance

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.

CSV Parsing: As in Figure 5.19, 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.

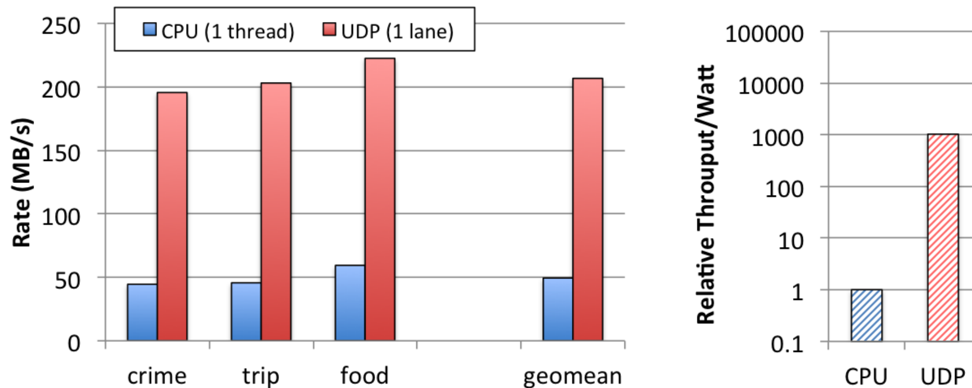


Figure 5.19: CSV File Parsing.

Huffman Encoding and Decoding: Figures 5.20 and 5.21 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.

Pattern Matching: Figure 5.22 shows that a single UDP lane surpasses a single CPU

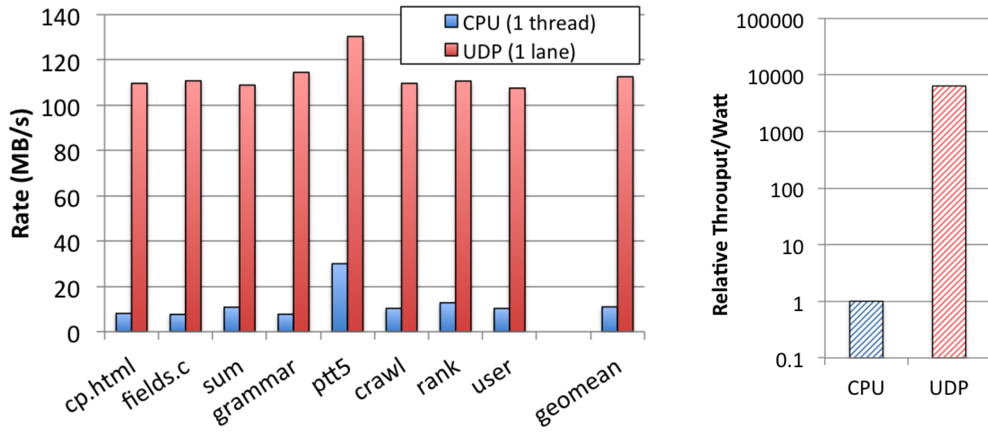


Figure 5.20: Huffman Encoding.

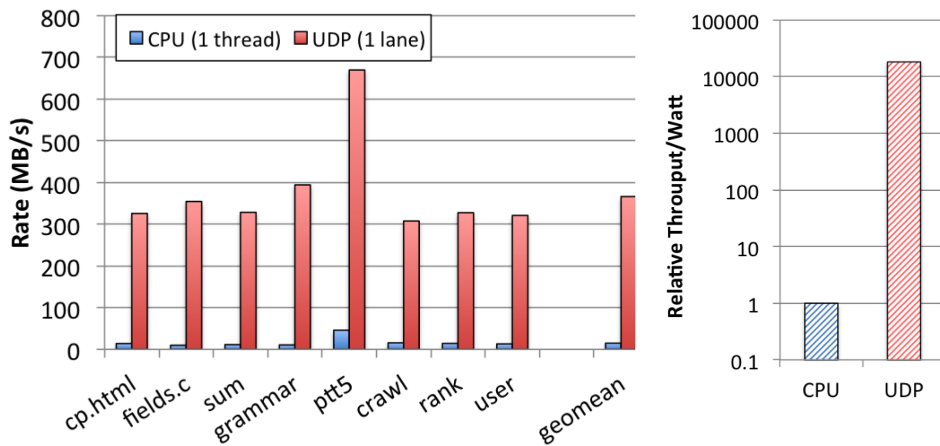


Figure 5.21: Huffman Decoding.

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.

Dictionary and Dictionary-RLE Encoding: UDP delivers a 6-fold rate benefit for both Dictionary and Dictionary-RLE (see Figure 5.23). Due to space limits, only Dictionary-RLE performance data is shown. For the full UDP, the power efficiency is more than 4,190x

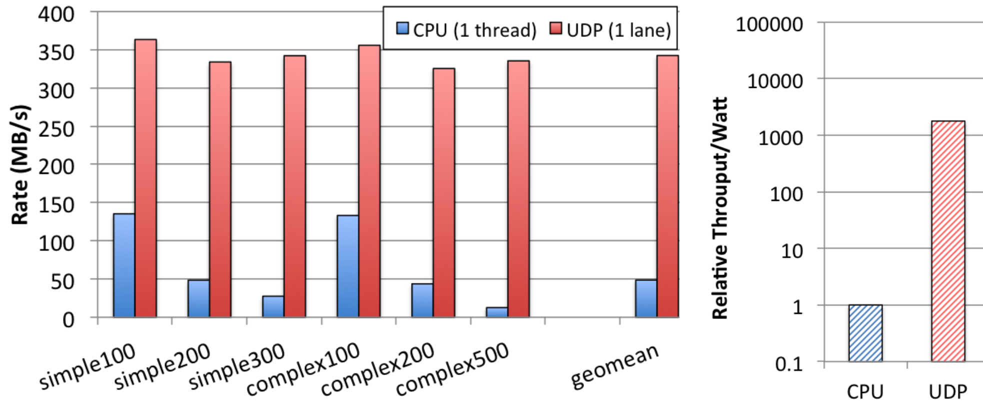


Figure 5.22: Pattern Matching.

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.

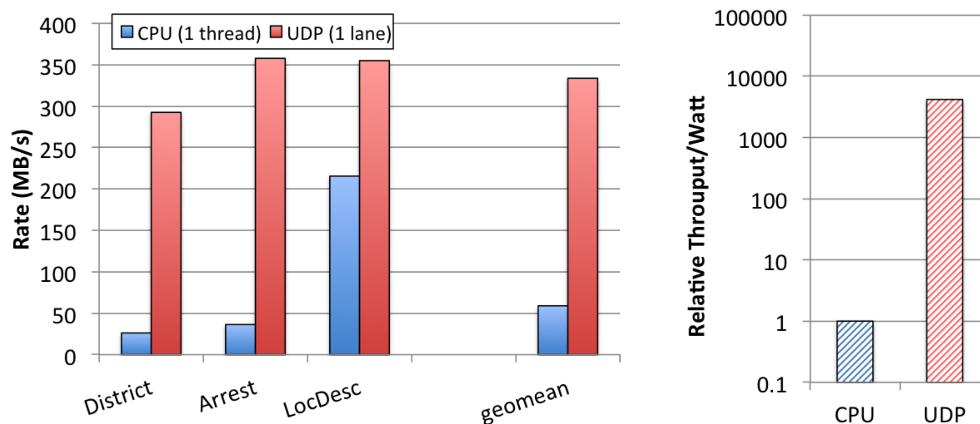


Figure 5.23: Dictionary-RLE.

Histogram: Figure 5.24 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.

Compression and Decompression: As shown in Figure 5.25, UDP Snappy compression with a single UDP lane matches a single CPU thread with performance varying from 70 MB/s to 400 MB/s (entropy). The full UDP delivers 276x better power efficiency than CPU[1].

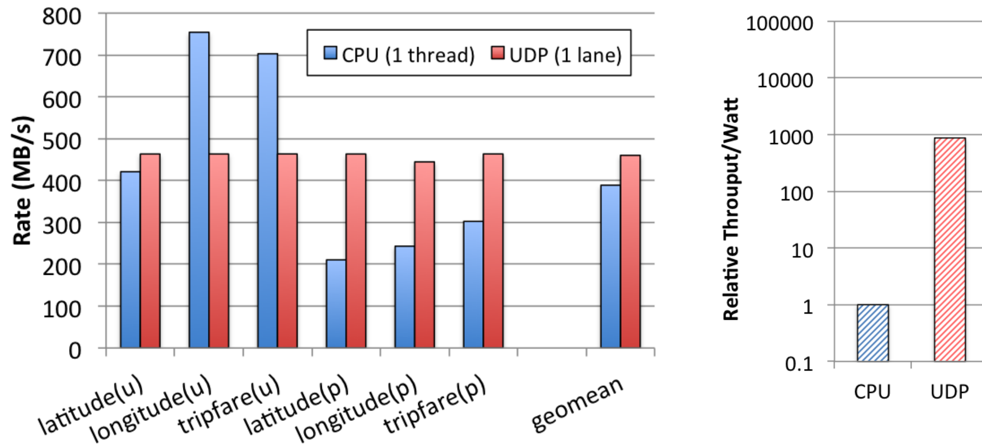


Figure 5.24: Histogram.

Figure 5.26 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.

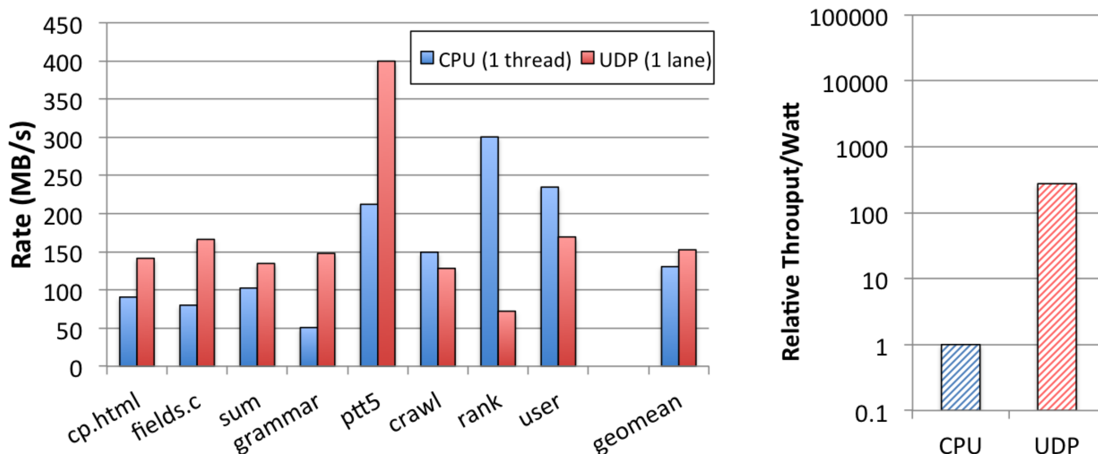


Figure 5.25: Snappy Compression.

Signal Triggering One UDP lane delivers constant 1,055 MB/s rate for all transition

1. 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.

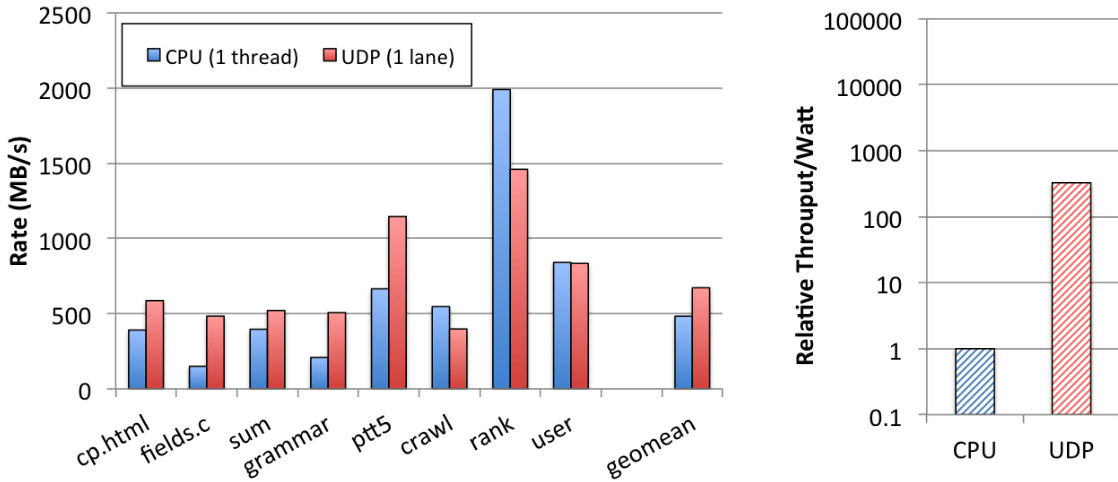


Figure 5.26: Snappy Decompression.

localization FSMs *p2-p13* [62], 4 times greater than both the CPU (275MB/s) and the FPGA implementation used in Keysight's product [20] (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.

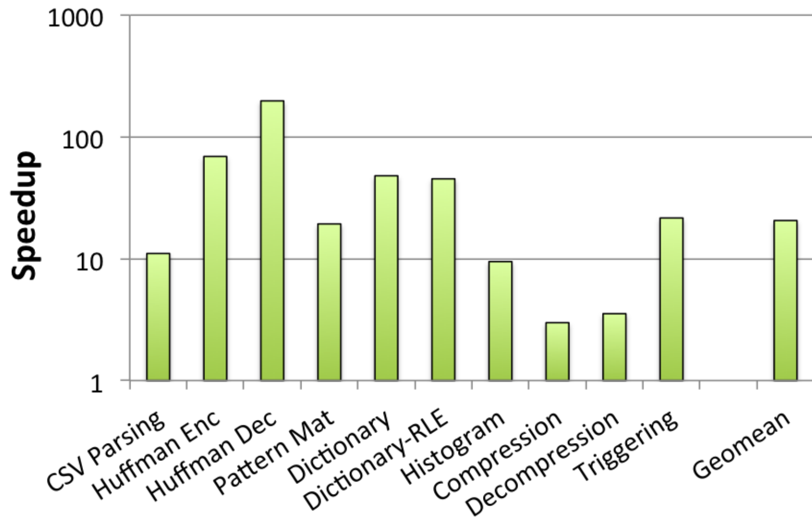


Figure 5.27: Overall UDP Speedup vs. 8 CPU threads.

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

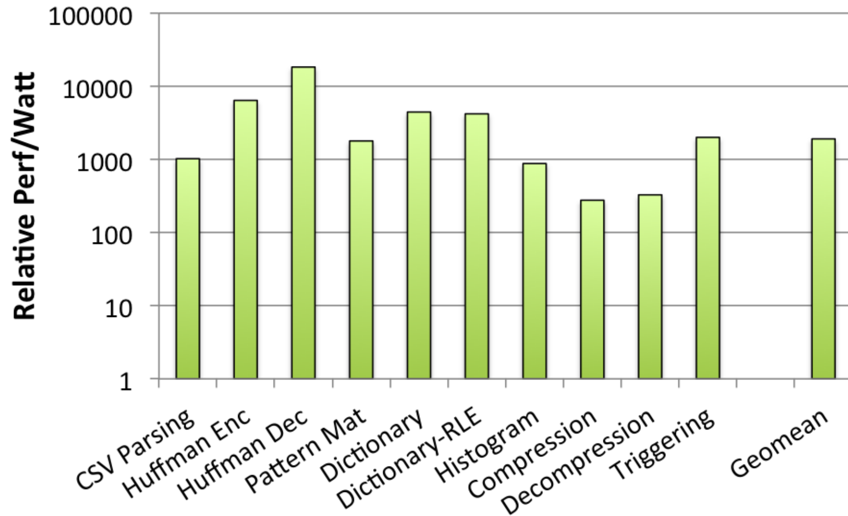


Figure 5.28: Overall UDP Performance/Watt vs. CPU.

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 5.27). Second, compare throughput/power for UDP and CPU in Figure 5.28, using UDP implementation power of 864 milliWatts from Section 5.4.3 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. Each 64-lane UDP accelerator provides >10GB/s average throughput across diverse set of data transformation workloads, which saturates a standard DDR3 memory channel easily. The resulted power consumption is less than 1% of a server X86 chip. We believe the UDP design accomplish the aforementioned performance and power requirements for UTA.

5.4.3 UDP Area and Power

We describe implementation of the UDP micro-architecture, and summarize speed, power, and area. The implementation differs from the conventional general-purpose architectures in

domain-specific customizations on data transformation workloads. This micro-architecture optimization helps the UDP design to achieve the UTA area and power requirements thus enjoys efficient in memory-hierarchy integration. Each UDP lane contains three key units: 1) Dispatch, 2) Symbol Prefetch, and 3) Action (see Figure 5.29). 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 System Verilog 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 [4]. The overall UDP system includes the UDP, a 64x2048-bit vector register file, data-layout transformation engine (DLT) [115], and a 1MB, 64-bank local memory. Silicon power and area for the UDP design is shown in Table 5.2.

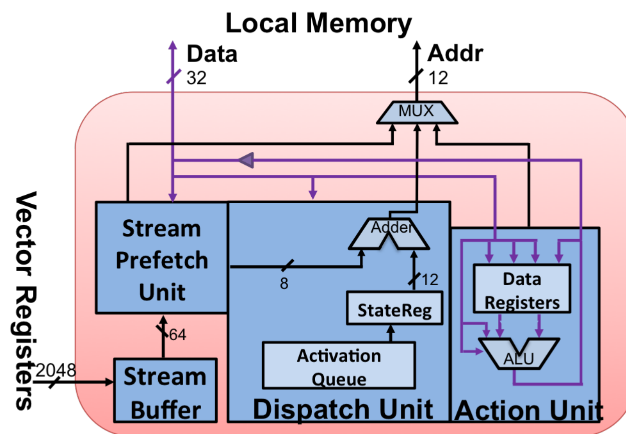


Figure 5.29: UDP Lane Micro-architecture.

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 [4]. 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

	Component	Power (mW)	Fraction	Area (mm^2)	Fraction
UDP Lane	Dispatch Unit	0.71	37.9%	0.022	40.6%
	SBPUnit	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%
	UDPLane	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 5.2: UDP Power and Area Breakdown.

Westmere EP core+L1 in a 28nm process [18]. 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.

64bit, 14nm UDP: We project the clock frequency and power consumption for a 64-bit UDP design extension under 14nm process for future research on the UDP architecture. The performance reported in Section 5.4.2 for 28nm CMOS UDP design is extrapolated for a 14nm process and 64-bit extension, which takes the previously reported speed and power from 1GHz and 864mW to 1.6Ghz and 160mW. The scaling considers the fact that the performance and power for UDP is heavily dominated by SRAM access time and energy, so

we compute it based on the CACTI [4] estimates and reflects two full process generation (TSMC 28nm to TSMC 14nm) and the shift to FinFET transistors.

Both the 32-bit, 28nm UDP and 64-bit, 14nm UDP enjoys tiny area cost and low power consumption that is less than 1% of the entire X86 Xeon chip power. As a result, these low costs enables adding UDP into the same die with host CPUs using the in memory-hierarchy integration approach. The UDP design achieves all three requirements of UTA in functionality, performance, and cost.

5.4.4 ACCORDA Micro-benchmarks

Finally, we evaluate the 32-bit UDP ASIC implementation on a set of data analytics micro-benchmarks and report its performance against other state-of-art solutions. These benchmarks serve as the key performance bottlenecks in ACCORDA full-system query execution on raw data in Chapter 7.

Regex Matching

Regular expression matching is important for data filtering as in two commonly used SQL string operators (LIKE and RLIKE). We use the TPC-H [31] Q13 regex pattern and the dataset (*comment* column in the *order* table). The ACCORDA accelerator (UDP) performance is compared to a CPU using the Intel Hyperscan library [17], a state-of-art SIMD implementation. We measure single thread performance and scale it up linearly to 8 threads assuming no interference. We also compare it to FPGA performance [109], using the best performance number based on consuming $> 50\%$ of the ALM and BRAM FPGA resources. We also compare to ASIC implementation – TitanIC RegX accelerator [120], a mature commercial regex accelerator. In Figure 5.30, the 8-thread CPU achieves 13GB/s, and the FPGA nearly doubles that (25 GB/s). The Titan accelerator performs similar to the 8-thread CPU but at much lower power. Despite much less silicon area, UDP uses its efficient multi-way dispatch and parallelism to achieve 64 GB/s, 4.9x than an 8-thread CPU.

Decompression

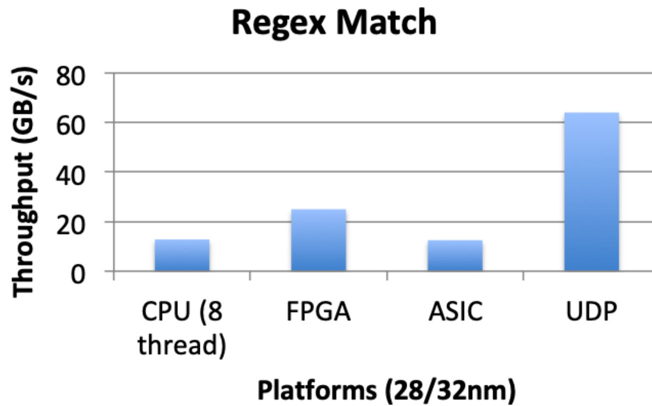


Figure 5.30: Regular Expression Matching.

LZ-77 based compression is widely used to reduce storage cost for storing raw data. Fast decompression is desired for querying raw data that is in compressed format. We use the *lineitem* compressed file in TPC-H [31] for measuring decompression performance of the ACCORDA accelerator (UDP), comparing to 8-thread CPU performance using Google Snappy [10], and an FPGA implementation derived from the Xilinx ZipAccel-D core [33]. To portray it in the best possible light, we replicate the ZipAccel-D cores to fully utilize the Virtex 7 (28nm) FPGA BRAMs (28Mb). Finally, we report the Intel 8950 ASIC [15] decompression performance. Our results (see Figure 5.31 show UDP matches best performance (13GB/s) but at a fraction of power and area cost (Section 5.4.3) required for the FPGA implementation. ASIC block give lower performance. UDP is 2.6x faster than an 8-thead CPU. The efficiency comes from avoiding branch misprediction and a parallel scratchpad memory system.

Parsing

Parsing is a critical task to extract fields from the unorganized raw data. It is one of the most time-consuming jobs on raw data processing [37]. We use the TPC-H [31] *lineitem* table in tabular format with the raw ASCII string encoding in this experiment in Figure 5.32. The parsing contains finding and validating delimiters, and extract interested fields. We compare the UDP performance with an 8-thread CPU parsing implementation using SIMD [98] and an ASIC design for parsing CSV/JSON/XML [41]. Even with a low area and power

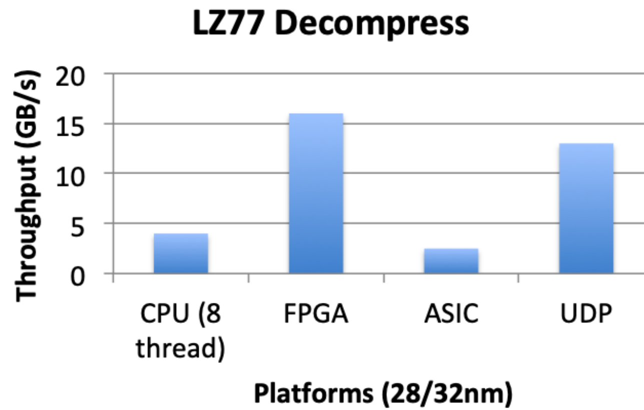


Figure 5.31: LZ-77 Decompression.

requirement, the UDP achieves 4.3GB/s throughput, 2x the performance of an 8-thread CPU (2GB/s), and 3x than a dedicated ASIC.

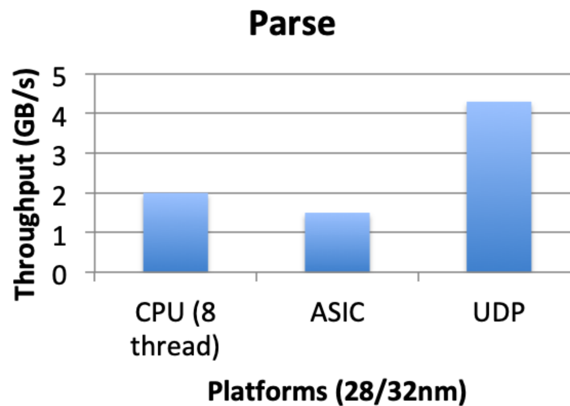


Figure 5.32: CSV Parsing.

Deserialization

Transforming data format to native machine format (e.g. binary) is important for query performance on raw data. Deserialization happens when processing data in disk-based format or network wire format. In this experiment, we use the UDP to accelerate the data transformation from ASCII format to DATE type in the *lineitem* table. It is one of the most expensive data transformation task in raw data processing [37, 83]. In Figure 5.33, we compare the Java deserialization CPU library with the UDP accelerated kernel. UDP

achieves 1.6GB/s throughput, more than 20x faster than an 8-thread CPU implementation (0.07GB/s).

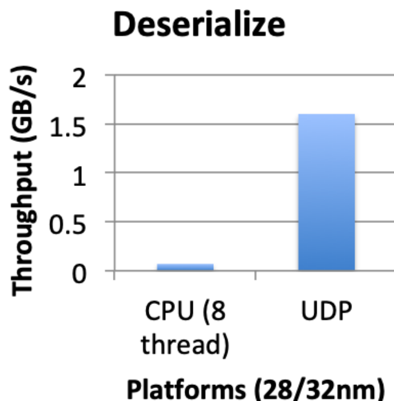


Figure 5.33: Deserialization.

5.5 Additional Related Work

In this section, we discuss the extensive related research on hardware accelerators in addition to the query accelerator contents in Chapter 3 to show UDP's superior efficiency, flexibility, and software-programmability.

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 [126, 127, 89, 79, 38]. As we have shown, the symbol and pattern oriented branch-intensive ETL workloads are particularly difficult, and our results show that UDP's multi-way dispatch (that improves on that in the UAP [63]) is an efficient solution. Many efficient encodings use variable-size symbols, and we know of some software techniques [100], 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 [19].

Table 5.3 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 [99] 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 [74] achieves 1.5 GB/s XML parsing. Compression hardware acceleration achieves 1 GB/s in PowerEN, 5.6 GB/s in Xpress [68], and 1.4GB/s DEFLATE on Intel 89xx series Chipset [15] (Table 5.3). With only 21 lanes (memory capacity limited), UDP outperforms the ASIC accelerators by 2.1-13x. The Xpress [68] 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 [63, 65, 117, 70, 6], but lack the programmability to address the costly follow-on processing (e.g. deserialization and validation) which often dominates execution time (Section 4.2.1). UDP handles these tasks and more. Its flexible programmability can address varied ETL and transformation tasks and future application-specific encodings or algorithms.

Acceleration of *stream processing* [56, 84] and *network processors* [6] can achieve high data processing rates with support for pattern-matching and network interfaces (see also NIC and "bump-in-the-wire" approaches [52]). 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 [108, 50] and aggressive prefiltering [17] to achieve 0.75-1.6 GB/s

Accelerator		Accel. Algorithm	UDP Algorithm	Accel. Perf (GB/s)	UDP Relative Perf	Accel. Power (W)	UDP Rel. Power Eff.
UAP [63]		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 [2][15]		DEFLATE	Snappy comp.	1.4	2.1	0.20W	0.50
Microsoft Xpress[3][68]		Xpress	Snappy comp.	5.6	0.54	108K ALM	- (FPGA)
Oracle M7[4][91]	DAX-RLE, -Huff, -Pack, -Ozip	RLE, Huffman, Bit-pack, Ozip	Huffman, RLE, Dictionary	11	1.4	$1.6mm^2$	0.56
IBM PowerEN[5] [74]	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 5.3: Comparing Performance and Power Efficiency of Transformation/Encoding Algorithms.

using a powerful Xeon out-of-order core. Software speculative approaches can increase stream rate, but at significant overhead [101, 133, 114]. GPU implementations [128] report throughputs 0.03 GB/s (large pattern sets) increasing to 1.6GB/s [134] (small sets). Several network processors [74, 6] employ hardwired regular expression acceleration to reach 6.25GB/s throughput. Unified Automata Processor achieves up to 5x better performance [63] by exploiting programmability to employ the best finite-automata models. UDP improves on UAP achieving much greater generality, but at a cost in performance and energy efficiency (Table 5.3). Recent work [70] achieves remarkable stream rate (32 GB/s) at high power consumption (120W).

5.6 Summary

In this chapter, we present the Unstructured Data Processor (UDP), a concrete instance of the Unified Transformation Accelerator (UTA). UDP is an architecture designed for general-purpose, high-performance data transformation and processing. Our evaluation shows UDP accelerates a diverse range of tasks that lie at the heart of ETL, query execution, stream data processing, and intrusion detection and monitoring. The design 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 design are low, making it suitable for incorporation into the CPU chip, memory and flash controller, or the storage system. The UDP design matches the domain-specific ASICs with extreme high-performance and low cost, which meets or even exceeds the UTA requirements listed in Section 5.1. The flexible functionality and high-performance enables UDP to handle various data transformation tasks in existing data analytical stacks. Low area and power cost enables low-overhead data sharing and movement with CPU hosts through in memory-hierarchy accelerator integration. These factors support UDP's flexible acceleration in any software systems without disrupting any existing software architectures or execution models. All desired architectural properties in an analytics system can be maintained such as flexible query optimization, uniform runtime management, and iterator-based execution model. We explain more detail of this in the next chapter.

2. We estimate compression power by 20W TDP[16] and exclude clock grid, IO/bus, and crypto. using relative ratio [12].

3. Altera Stratix V FPGA.

4. Scale to 28nm TSMC and estimate based on chip die size [23, 29].

5. IBM 45nm SOI [12].

CHAPTER 6

ACCELERATED TRANSFORMATION OPERATORS

In this chapter, we present the Accelerated Transformation Operators (ATO) approach. It serves as the ACCORDA's software architecture that integrates hardware acceleration, applies the design choices of encoding-extended operator interface and uniform worker model. In addition, we discuss the rationale of UTA and its integration approach with respect to ATO.

6.1 ATO: ACCORDA's Software Architecture

We discuss the ACCORDA's software architecture, Accelerated Transformation Operators (ATO), describing the key design choices of 1) sub-typing operator interfaces with data encoding and 2) uniform worker model. First, we describe the idea of data encoding types for operators, extending the traditional operator interface (in Section 6.1.1). The encoding-extended operator interface coupled with hardware acceleration for data transformation forms the key innovation in ATO. Second, we describe the uniform worker model that is used to integrate hardware acceleration into the runtime. We show how these elements of ATO preserve the structure and benefits of query engines and optimizations. They also enable adding pure encoding operations, and optimized relational operators and user-defined functions, within the same framework.

6.1.1 Encoding in the Operator Interface

ATO extends the operator interface, conceptually adding data encodings as column properties.[1] This effectively subtypes the operators with data encoding, allowing expression of a wide range specialized operators (e.g. filter for RLE-encoded INTs). These extended types (see Figure 6.1) are used for all query optimization and execution, and enable query plans to

1. Blocking and cross-column group optimizations complicate this slightly, but we defer that complexity to later discussion.

express encodings, data transformation amongst encodings, and optimize across them. Data formats can satisfy the encoding requirements of accelerated operators' implementation with runtime data transformation, and can be fused to further improve data locality and save transformation cost. New operators that implement inexpensive data transformation using acceleration (Chapter 5) provide further benefits.

Operator Interface = <Attribute1, Attribute2, …>
Attribute = <DataType, Name, Metadata>
DataType = String | Double | Date | Integer | …

ATO Interface = <Encode-Attrib1, Encode-Attrib2,…>
Encode-Attrib = <DataType, Name, Metadata, **Encode** >
Encode = Native | Dictionary | Huffman | Part-Dictionary | …

Figure 6.1: ATO's Encoding-extended Operator Interface.

A traditional operator interface is a list of *attribute* types. ATO operator interfaces are lists of <*attribute, encoding*> tuples as illustrated in Figure 6.1. ATO's current implementation supports four encodings: Native (native format), Dictionary, Huffman, and Partitioned-Dictionary (a prefix of the data items are dictionary-encoded). NATIVE allows legacy operators to be "grandfathered", and together with new explicit encoding operators makes a usable suite of operators. In addition, customized operator variants that require particular encodings for improved performance are added.

To illustrate the power of encoding-extended operator interface, we show a full query optimization example that is transformed step by step in Figure 6.2. We first introduce UTA/UDP hardware acceleration (left green arrow). The original query can then utilize the accelerated JSON reader (blue) under the traditional operator interface. However, the interface is not capable of capturing further optimizations related to data encoding. After extending the operator interface with encodings (right green arrow), a series of query plan transformations (highlighted in red) can be performed for performance optimization. In Figure 6.2, encoding for each data attribute is suffixed with "_". For example, in "string_json", *json* is the encoding and *string* is the data type. The encoding-extended interface allows lazy

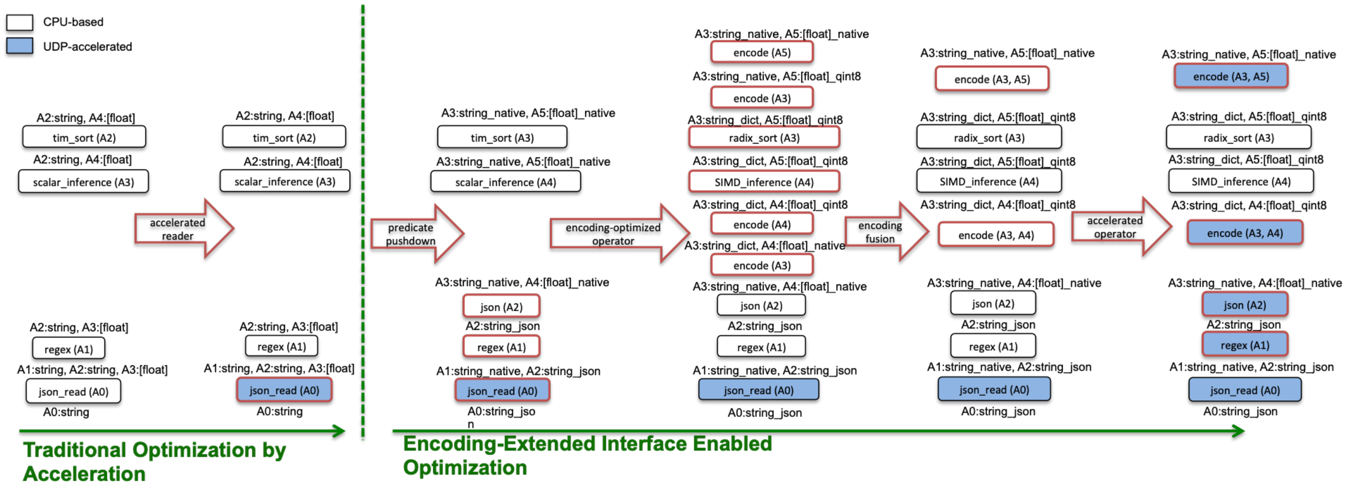


Figure 6.2: An Encoding-Based Query Optimization Example with Detailed Transformation.

parsing and deserialization by pushing predicates down. Moreover, advantageous operator implementations can be selected with specific encodings. Introduced encode operators can be fused together to improve execution efficiency and data locality. Finally, the UTA/UDP's design and its memory system integration allows operator-level hardware acceleration (blue) in a query plan.

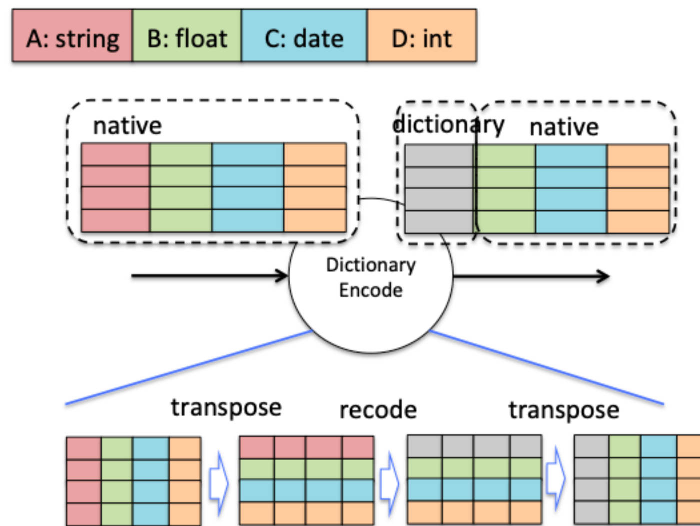


Figure 6.3: Encoding Operators in a Query.

Figure 6.3 gives a concrete example to demonstrate what happens in an encode operator. The encode operator first batches a set of records into a block and sends the block to the

UTA/UDP accelerator (Section 6.3) for a series of transformations (transpose for efficient dictionary encoding of attribute A, and then transpose back with each record streaming into downstream operators). In Section 6.3.2, we revisit this example, showing how it can be accelerated.

Of course, effective query optimization depends on sufficient data statistics and cost models collected and stored in a meta-store beforehand, or via an adaptive sampling phase before query execution. Throughout the rest of the thesis, we assume the system has sufficient information (such as a pre-built dictionary, distribution, etc) to perform the encoding optimization. In summary, the encoding-extended operator interface captures data encodings for intermediate results in a query plan allowing flexible and cascading optimizations.

6.1.2 Uniform Worker Model

In ATO, extended operator types enable accelerated and traditional operators to be treated uniformly. ATO also integrates data transformation acceleration seamlessly by using a uniform worker model, reducing access overhead and preserving data locality. This is in contrast to many hardware acceleration approaches [95, 81, 109, 49].

ATO implements a uniform worker model; all runtime worker threads are accelerated (Figure 6.4). This uniformity simplifies scheduling of work, and allows queries to run from end-to-end on a single worker without switching. That approach enhances data locality. A common approach in hardware-accelerated databases is to have two different types of workers, one type accelerated. Such an approach complicates scheduling, forces query execution to switch between workers to exploit acceleration, and reduces data locality.

As mentioned above, uniform worker model has numerous advantages over heterogeneous worker model. Traditional accelerated systems are constrained by runtime heterogeneity because of their accelerator power and area cost. For example, GPUs and FPGAs are power-hungry accelerators and occupy significant silicon area. As a result, these physical limitations prevent on-chip integration and, in turn, advocate the PCIe-based integration.

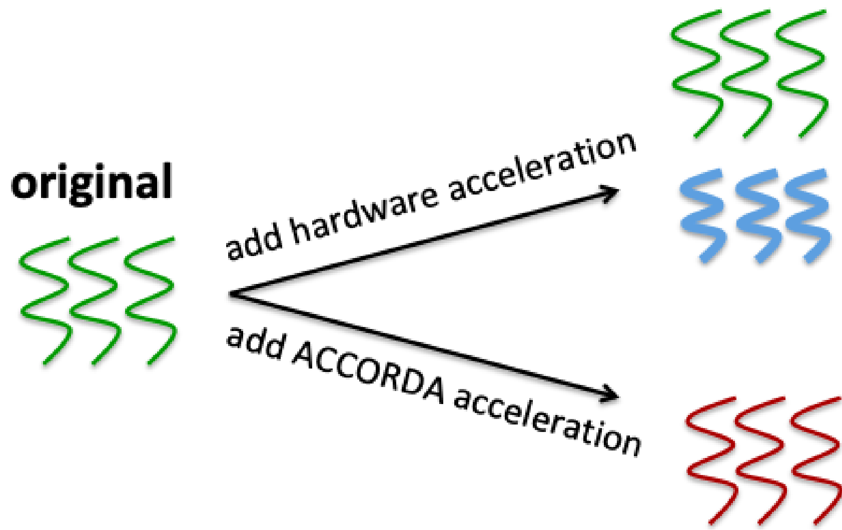


Figure 6.4: Many accelerated systems require different types of workers (upper). ATO has uniform accelerated workers (lower).

Accelerators with PCIe integration suffer from expensive data transferring between CPU cores and accelerators. Furthermore, the significant speedup from accelerators usually lead to a different execution model other than CPUs. Therefore, conventional accelerators are exposed to higher-level software system as an independent runtime to allow direct scheduling, manipulation, and optimization to mitigate the data transfer overhead and adjust for the execution model difference (e.g. row vs. column, tuple vs. block, etc.). On the other hand, because of UTA's design, on-chip integration of UTA is possible. It enables low-overhead data sharing across CPU cores and accelerators, and between accelerated and unaccelerated operators. Later, we show that a single UTA is sufficient to support time-multiplexing with as many as 16 cores, and still delivers high speedups (see evaluation in Section 7.2). More importantly, UTA's generality makes it reasonable to get integrated into a CPU chip, supporting various kinds of data transformation workloads without concerning narrow applicability or obsolete hardware architecture.

In summary, we have discussed the two design choices of ATO – encoding-extended operator interface and uniform worker model. In Section 6.2, we evaluate the benefits of these

two design choices, and show their importance for a flexible and well-preserved integration of hardware acceleration. The rationale behind pairing ATO with UTA in ACCORDA is discussed in Section 6.3.

6.2 Benefits of ATO Software Architecture

In this section, we demonstrate the benefits of ATO software architecture that are derived from encoding-extended operator interface and uniform worker model. Briefly, the key advantages are 1) acceleration integration that preserves query engine software architecture, and 2) the ability to maintain system architectural properties such as a unified runtime and flexible query optimization.

6.2.1 Preserving Software Architecture

The ACCORDA's software architecture (ATO) design preserves conventional data analytics system architecture. For example, in our implementation built on SparkSQL [43], system components such as query front-end, storage system, and distributed management are unchanged. As shown in Figure 6.5, newly added components include new rules as well as two classes of new operators (encoding-optimized compute and transformation). The new elements are depicted in green. Specifically, the optimizer layer gains new data-encoding rules for the rule-based optimizer (e.g. the Catalyst optimizer [43]). These rules transform the query plan to select favorable encodings.[2] In the execution layer, ATO encapsulates legacy operators using the extended interface, and adds the encoding and compute operators. The compute operators implement the given computation with specific input and output data formats. Each encoding operator converts among data formats (e.g. Native, Dictionary, Huffman, Partitioned-Dictionary), enabling the optimizer to exploit format-optimized compute operators. The storage layer is unchanged.

2. Our evaluation (Chapter 7) inspired addition of rules for data read, filter, hashing, regex matching and sorting to place data encoding operators in a query plan.

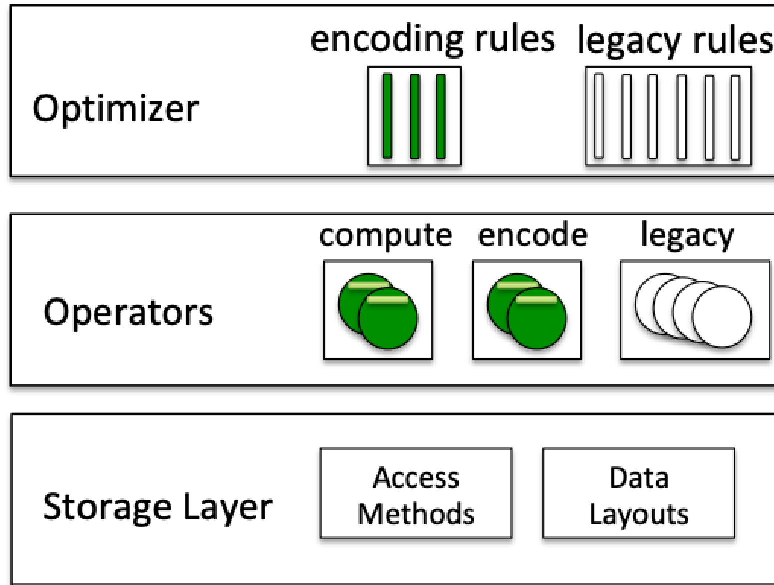


Figure 6.5: ATO Query Engine Software Architecture (added components in green).

Importantly, the ATO software architecture avoids disruptive changes to the optimizer. In Figure 6.6, we compare optimizer architectures for traditional, accelerated, and ACCORDA accelerated. On the left, the traditional non-accelerated query optimizer consists a cost model and dozens of optimization rules.

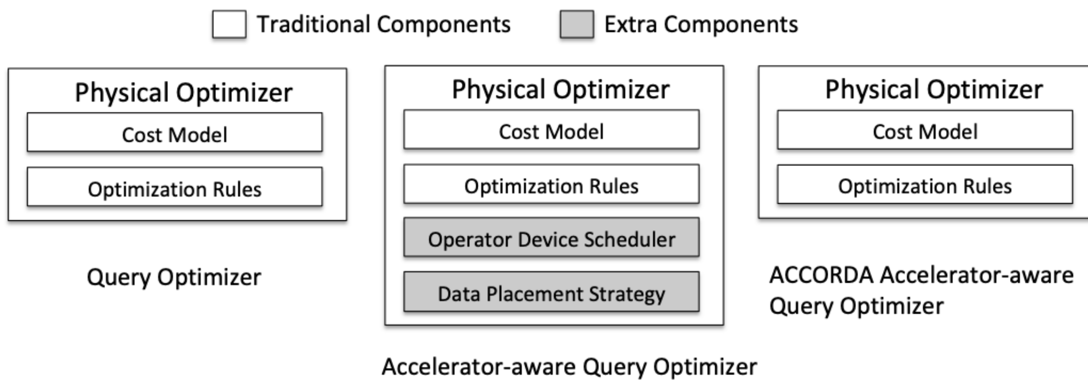


Figure 6.6: Comparing Optimizer Architectures for Acceleration.

In the middle, GPU or FPGA accelerated systems add layers to manage the cost of data transfer to and from PCIe, data transformation overheads, and data locality impacts. Device scheduling is a further complication [49]. On the right, the ACCORDA query optimizer

maintains the original optimizer architecture, adding rules instead, and using cost metrics alone to manage use of acceleration. The key to this is the ATO software architecture and in memory-hierarchy acceleration (Section 5.2), and uniform worker thread model. The query optimizer needn't manage explicit data placement and device scheduling, and all existing nice properties can be maintained.

6.2.2 Preserving Flexible Query Optimization

We discuss the importance of maintaining query optimization flexibility. With ATO software architecture, no additional layers are required to be added to manage the device heterogeneity.

Coarse-grained vs. Fine-grained

Iterator-based execution (tuple-at-a-time) has well-known virtues for avoiding unnecessary materialization – and corresponding computation and IO [71, 43]. ACCORDA's software architecture ATO, combined with in memory-hierarchy accelerator integration (Section 5.2) allows ACCORDA to preserve the tuple-at-a-time processing model (or at least a block-oriented version).

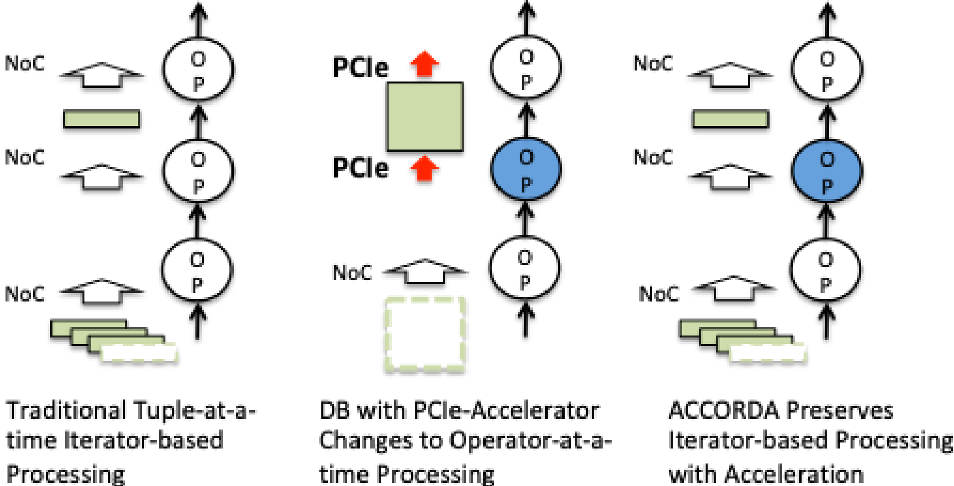


Figure 6.7: Comparing Execution Model.

In contrast, accelerated DBMS (e.g. GPU database) generally employs operator-at-a-time processing [49]. Figure 6.7 shows these differences. This is a consequence of GPU's separate

memory systems (copy-in copy-out, data movement and synchronization cost), as well as their non-uniform thread model (see Section 6.1.2). Consequences include large intermediate materialized results, extra computation and memory demand – particularly for accelerators with limited memory (e.g. GPU). So the accelerator totally disrupts the optimizer; and a common response is to use a completely new strategy that splits a query plan into coarse-grained chunks and schedules each separately. Each chunk runs on the same device using a single processing model (e.g. operator-at-a-time).

ATO enables full, fine-grained query optimization. We reuse the existing rules in Spark SQL. The optimizer treats accelerated operators as the first-class citizens as shown in Figure 6.8. Rule *a,b* are fired when the sub-tree in a query plan and the estimated statistics meet the transforming condition. Rule *c* replaces a data transformation operator with an accelerated counterpart. Rule *d* transforms a costly regex matching filter expression into a separate accelerated operator. While beneficial in ATO, these fine-grained optimizations would damage performance in a GPU or FPGA accelerated database systems.

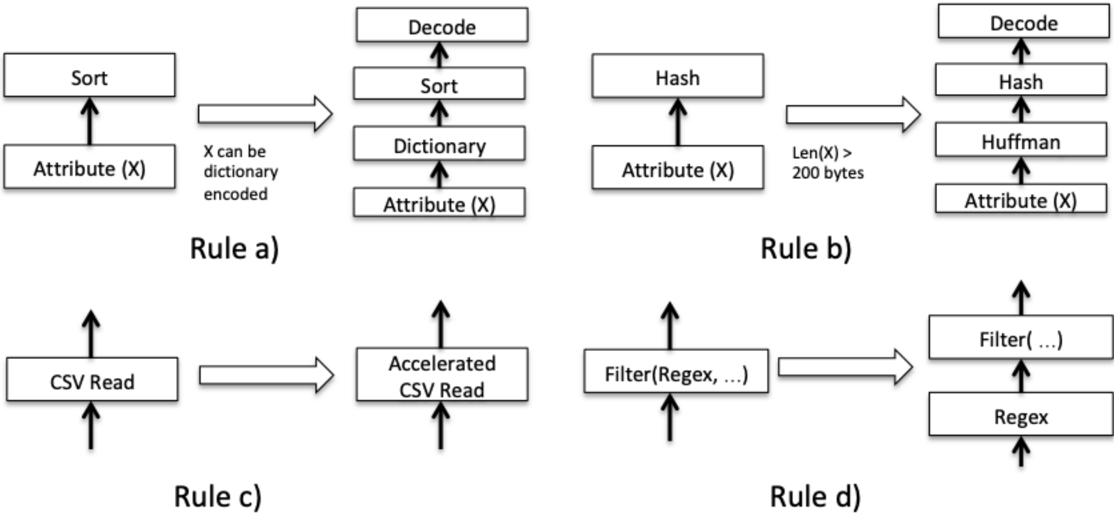


Figure 6.8: ATO Rule Examples.

In Figure 6.10a, we illustrate the benefits of fine-grained optimization for two TPC-H [31] queries using the platform, system and workload described in Section 7.1. In Q10, the ATO optimizer replaces the data source operator with an accelerated one (rule c) and combines

seven *group by* attributes together and compresses it using Huffman coding (rule b). The follow-on hash aggregation is computed based on the encoded group. Optimization with rule c alone achieves 2.6x speedup than the baseline, and an additional 30% performance improvement with rule b. Similarly in Q13, besides the accelerated data read (rule c), the expensive regex filtering expression is replaced by the hardware-accelerated regex operator (rule d). The combined effect (rule c+d) produces a 13.2x speedup compared to the baseline.

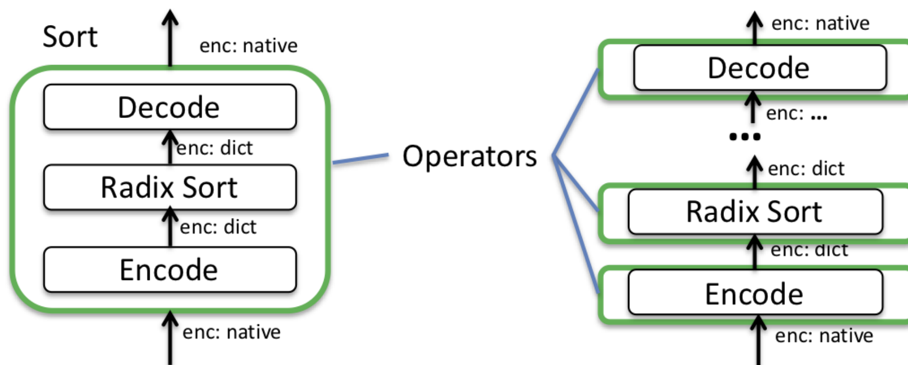
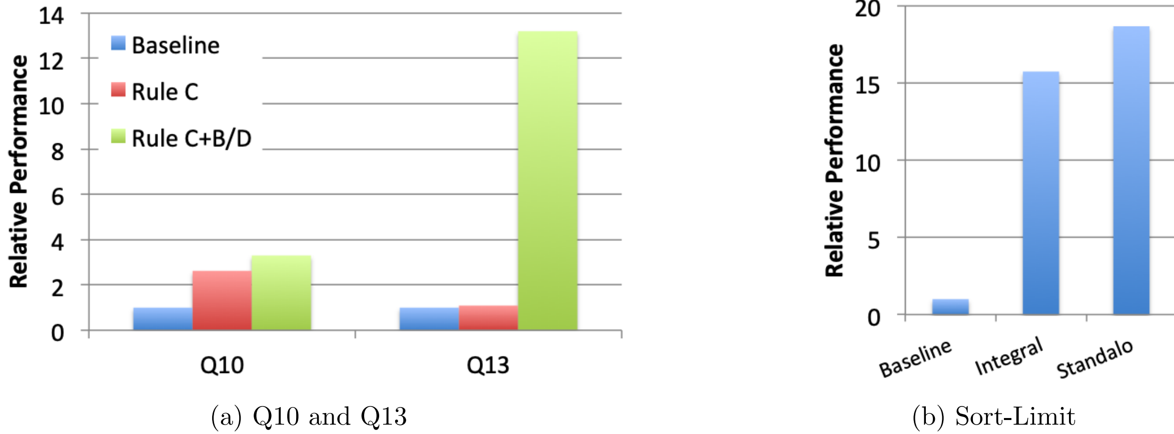


Figure 6.9: Integral and Explicit Encoding in Queries

Integral vs. Standalone Encoding

ATO's encoding-extended operator interface allows encoding operators to be decoupled (standalone) from integral implementations (encode-compute-decode). The addition of standalone encoders enables flexible and cascading query optimization across encodings. Traditional uniform encoding requires operator implementations to incorporate data transformations inside (integral) to maintain the standard encoding interface (Figure 6.9). In ATO, attributes can take on varied encodings at each phase of the execution, enabling operator implementations to avoid this overhead, for encoding operations to standalone. Thus the ATO query optimizer can use rule-based optimizations to choose the best encodings while decode operators are inserted where necessary and delayed after filtering or aggregation to reduce cost. Moreover, standalone encoding transformations can further enables fusing encoding operators to collapse multiple format transformation operations into one operator, improving accelerator recoding efficiency.



(a) Q10 and Q13

(b) Sort-Limit

Figure 6.10: Benefits of ATO Fine-grained Query Optimization

We illustrate the potential gain in Figure 6.10b with the single-thread performance of a sort-limit query on the *shipdate* attribute of the *lineitem* table on the ACCORDA system with the UTA and ATO implementations (Section 7.1). The integral implementation for sort with hardware acceleration brings a 15x speedup. But standalone decoding operators allow work to be shifted after the filter operations, producing another 20% performance improvement. Larger benefit is possible if the attributes require more complicated transformations or if selectivity is higher. Fast data transformations provided by the hardware accelerator enables the fine-grained, standalone encoding optimizations in a query plan. Each rule may bring a marginal speedup, but the cascading impact can be much larger for complex queries.

In summary, the ATO software architecture provides an elegant hardware acceleration integration into existing analytics system architectures, introducing encoding-based query optimization and uniform worker model, preserving software architecture and desired properties (e.g. query optimization flexibility, iterator execution model, etc) of existing systems.

6.3 UTA Unleashes the Power of ATO

In this section, we explain the rationale behind deploying UTA as the targeted accelerator for ATO in the ACCORDA approach. In addition, we show why UTA's in memory-hierarchy integration (Section 5.2) is desired with respect to the ATO requirements.

6.3.1 *UTA as the ACCORDA Accelerator*

Unified Transformation Accelerator (UTA) is a principled hardware approach to design a tiny, low-power, high-performance, and software-programmable accelerator with flexible support across a wide range of data transformation tasks. The UTA approach fulfills all four key requirements as an ACCORDA accelerator. First, ATO requires tight and seamless collaboration between CPU hosts and accelerators, meaning the accelerator has to be small enough to be nearby a CPU core on the same chip. Second, introducing data encodings in a query plan with ATO bases on the assumption that the accelerator provides superior speedups to significantly reduce the data recoding cost. Third, the diverse data formats and representations that ATO deals with require strong generality and flexibility from the accelerator. Finally, the vast amount of rapid innovations in data formats, encodings, and representations require accelerator's software programmability because when new formats or updates appear, only a few software programs need to be (re)written without any hardware changes.

In Chapter 5, we present a concrete design of the UTA approach, called the Unstructured Data Processor (UDP). UDP not only meets all the UTA expectations, but exceeds them by a large margin. As Section 5.3 described, UDP contains four unique architectural features: multi-way dispatch, variable-size symbol support, flexible-source dispatch (stream buffer and scalar registers), and memory addressing. These features, combined with software programmability, a fast scratchpad memory, and 64 parallel lanes enable a single UDP to provide 20x geomean speedup vs. an entire multi-core CPU across a broad variety of export-transform-load computations such as CSV parsing, Huffman encode/decode, regex pattern matching, dictionary encode/decode, run-length encoding, histogram, and snappy compression/decompression. Furthermore, UDP achieves a remarkable geomean 1,800-fold increase in performance/watt. At a tiny $3.82mm^2$ area, and 0.86 watt (Section 5.4), UDP can be incorporated on a CPU with minimal increased cost. Together, these characteristics enable the low-cost transformation of data encoding at full memory speeds in ACCORDA,

making UDP (thus UTA) a perfect candidate as the ACCORDA accelerator.

6.3.2 Why UTA's Memory Integration

With the UTA accelerator design (UDP) and the ATO software architecture, there is still one missing piece left about the UTA's integration into the memory system to meet the data sharing demand of ATO. ATO requires frequent data sharing between CPU hosts and the accelerator in fine granularity. The integration approach shouldn't destroy the superior performance delivered by the UTA. In this section, we demonstrate that the in memory-hierarchy integration approach described in Section 5.2 indeed meets this requirement.

We first consider the PCIe integration alternative for integrating the UTA accelerator into a larger hardware system. Historically, both GPU and FPGA accelerators have used PCIe integration approaches [105, 49]. One reason for this is convenient access to hardware boards. However, there are important technological reasons. First, both FPGA's and GPU's are typically large chips, and they need that large size (100's of mm2) to deliver significant acceleration performance. They are as large as a CPU. Second, GPU's in particular are high power devices, typically 50 to as much as 300 watts. They consume as much power (or even more!) than a CPU. Third, GPU's require a dedicated high bandwidth memory for high performance. This means that data sharing is done via copying, which leads to expensive data transfer between two separate memory system. During the execution of a query using ATO for accelerated operations, data is forced to go off the chip to DRAM for DMA execution. The resulted data movement overhead prevents fine-grained interleaved execution that ATO requires.

On the other hand, the in memory-hierarchy integration approach proposed in Section 5.2 overcomes the aforementioned disadvantages. The CPU has normal access to caches, but can directly address and access the accelerator's scratchpad memory directly. Interference is avoided by setting a region of the address space as uncacheable and mapping those virtual addresses to the accelerator memory. Data can be moved in and out of the scratchpad

memory via library routines that initiates lightweight DMA operations to transfer blocks of data from memory-hierarchy/DRAM to the accelerator scratchpad memory. Later in Section 7.2, we quantitatively evaluate the performance benefit that in memory-hierarchy integration brings to the ATO approach. For the example in Figure 6.3 with UTA acceleration, the record block is moved from CPU caches by the DMA engine into the accelerator's scratchpad. UTA runs a program that transposes column A and column B with balanced distribution across its parallel engines. Then, UTA performs the encoding. Finally, result is transposed again, and the results DMA'ed to the CPU cache. Avoiding off-chip traffic is the key to preserve the speedups of the accelerator in this case.

In short, UTA's in memory-hierarchy integration enables low-overhead data sharing between CPU cores and the accelerator, a key foundation of the ACCORDA software architecture.

6.4 Summary

In this chapter, we present Accelerated Transformation Operators (ATO) , the ACCORDA's software architecture that integrates UTA hardware acceleration, enabling efficient data encoding optimization and overall query optimization. We apply two design choices in ATO – sub-typing operator interface with encodings and uniform worker model. Runtime data formats can be tailored to accelerated operators' implementation dynamically, and can be collapsed together for better performance. We further demonstrate that the UTA's in memory-hierarchy integration approach is the key for these design choices whose benefits include preservation of system architectures, flexible query optimization, and uniform runtime. In the next chapter, we show a quantitative evaluation of combining the ATO and UTA approach. Together, they enable an analytics system to process raw data with high-performance robustly.

CHAPTER 7

ACCORDA EVALUATION

In this chapter, we present the evaluation of the full ACCORDA system. ACCORDA combines the aforementioned UTA (Chapter 5) and ATO (Chapter 6) approach. We begin with the discussion of experimental methodology. Then, we evaluate choices of ACCORDA accelerator integration in a hardware system, followed by a discussion of software programmability benefits. Using prototypes of UTA and ATO, we subsequently evaluate ACCORDA's overall query performance, comparing it to other raw data processing approaches. Finally, to evaluate the sensitivity of our results, we study how performance varies with data properties and format complexity.

7.1 Methodology

We discuss the performance modeling methodology of the full ACCORDA system with an emphasis on ACCORDA software (ATO) and hardware (UTA) prototype. In addition, we briefly describe our workloads on raw data processing.

7.1.1 System Modeling

The ACCORDA system contains two parts: the ATO software architecture and the UTA hardware architecture. Thus, we discuss our methodology for accurate performance modeling of ATO and UTA, respectively. Briefly speaking, we model ACCORDA system performance by combining the time on software processing, hardware acceleration and in memory-hierarchy data transfer. We study the performance of a single-thread worker, and combine the times from separate models. The baseline system runs standard software on a conventional CPU.

Software processing is based on a software prototype derived from Apache Spark SQL (version 2.2) [43] with the ATO software architecture (Chapter 6) implemented for the query engine and optimizer extensions. This includes the cost of using Java Native Interface (JNI)

to make data accessible to the accelerator. The UTA hardware acceleration is modeled using a full implementation of UDP in custom silicon that runs at 1GHz in 28nm CMOS (Chapter 5). Figure 7.1 demonstrates the software and hardware prototypes for ACCORDA performance modeling with an emphasis on the modified system components across the stack.

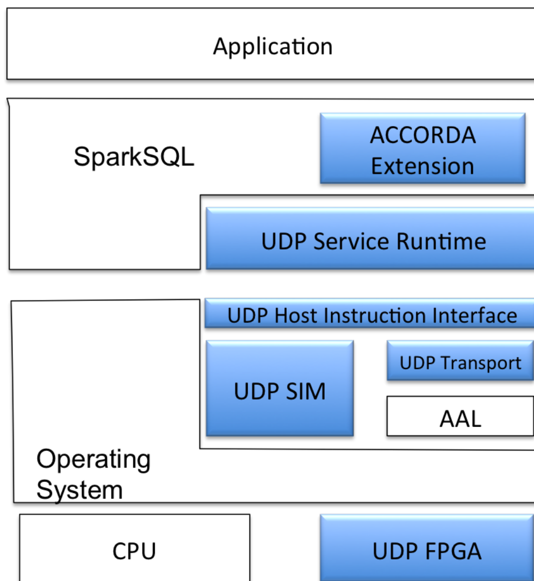


Figure 7.1: Modified System Components across the Stack for ACCORDA Modeling.

	FPGA Model	**UDP ASIC**
DMA Bandwidth	30 GB/s	30 GB/s
Clock Frequency	40 MHz	1 GHz
Resources	295K ALM, 10Mb BRAM	$8.69mm^2$

Table 7.1: Hardware Platform Statistics

To get high speed results for large-scale applications, we ran the applications and the entire ACCORDA software/hardware system on the Intel-Altera HARP FPGA platform [32] in Texas Advanced Computing Center [30]. The HARP system has two sockets connected through QPI, one is occupied by a 14-core CPU (Intel Xeon E5-2680) and the other by the FPGA (Altera Arria10). Thus each ACCORDA worker switches between the on-CPU execution (Spark tasks and acceleration stubs) and the on-accelerator execution (modeled acceleration on the FPGA). The FPGA UDP model runs at 40MHz clock frequency; some

details are listed in Table 7.1. It shares the same UDP instruction-level interface with the cycle-accurate software simulator. The FPGA software integration requires a driver library for data transportation and the Intel AAL back-end support (Figure 7.1).

We derive full system performance by taking traces from the HARP system and re-timing them (a timed hybrid-simulation modeling methodology), adjusting for the modeled UDP speed, and modeling data transfer time between CPU core and accelerator for our in memory-hierarchy integration by computing the elapsed time by dividing the transferred data size with a typical DMA bandwidth [97] (shown in Table 7.1). The rest of the software processing is modeled with wall clock time.

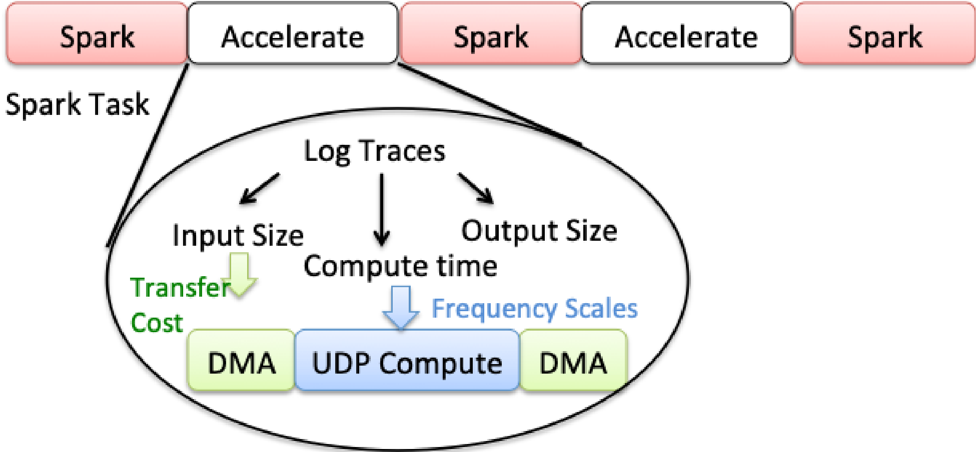


Figure 7.2: ACCORDA Performance Modeling.

The measurement infrastructure outputs event logs that capture performance. Figure 7.2 demonstrates the approach. Task information (e.g. timestamps) are recorded alone with the execution traces. The trace log is post-processed to implement the hybrid simulation model and reflect the updated task time. This hybrid performance modeling is conservative since it assumes no overlap between the CPU core and acceleration, and performance is penalized by cache pollution due to simulation and logging.

7.1.2 Workloads

We use the TPC-H dataset [31] for the evaluation. The raw data is in tabular format (TBL), which hasn't been converted into the database binary format yet via batch loading. Strictly speaking, the TPC-H dataset on TBL format is not really native raw data since the tables have been normalized. Thus it limits the potential data encoding optimizations we can explore. Nevertheless, TPC-H on TBL mimics the format property of raw data. To the best of our knowledge, it is the best publicly available dataset for raw data experiments. In particular, Q1, Q4, and Q6 are selected because of their simplicity for understanding performance benefit and representativeness in SQL. We use Q10 to represent raw data queries that require many attributes in intermediate results. Q12 and Q13 represent queries that need extensive string filtering and regular expression matching on raw data.

7.2 Benefits of In Memory-Hierarchy Integration

In memory-hierarchy accelerator integration lays the foundation for the ACCORDA's software architecture (ATO), enabling data transformation to improve performance. We compare two approaches for accelerator integration: UDP accelerator in memory hierarchy vs. on PCIe. The total performance and data movement are modeled using LogCA [40]. In this experiment, we use the TPC-H *order* table in the compressed tabular format with scaling factor 10. The query is simple: **SELECT** *date, comment* **FROM** *order* **WHERE** *comment* **RLIKE** *".*special.*requests.*"*.

Figure 7.3 shows the runtime breakdown across three phases: decompression, parsing-select, and regex filtering. Decompression and regex filtering are completely offloaded, but select requires collaborations between CPU and accelerator (UDP). The data sharing for the *select* across the PCIe bus produces 66% overhead when compared to the in memory-hierarchy one. Data movement shows a similar story (see Figure 7.4). We modify the *comment* attribute for a selectivity of 1% and 100%. The result shows that PCIe-integrated system pays 6.8x-

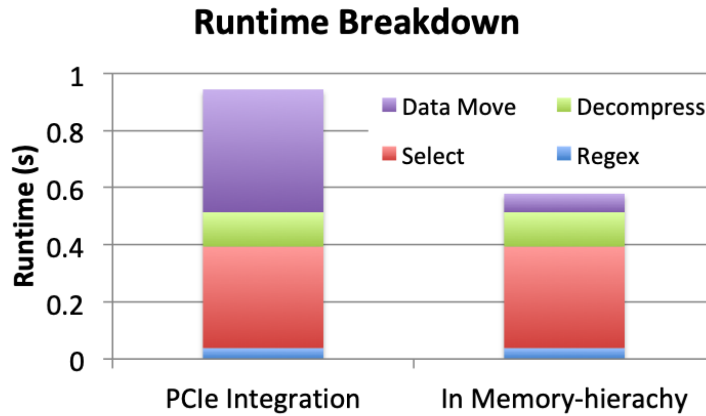


Figure 7.3: Runtime Breakdown.

14x more data movement than the memory-hierarchy integrated one. In memory-hierarchy accelerator integration preserves the iterator-based execution model, allowing ACCORDA to preserve fine-grained data sharing and interleaved execution with the accelerator.

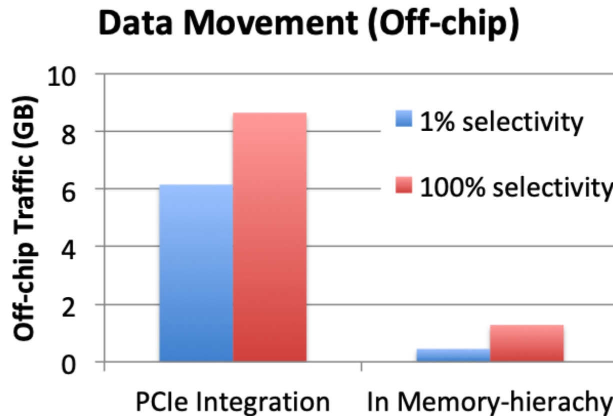


Figure 7.4: Off-chip Data Movement.

Acceleration scalability is a critical enabler of the accelerated worker uniformity. We perform the study using one UDP accelerator (64-lanes) in a single ACCORDA node. We use weak scaling on the TPC-H Q1 workload, increasing worker threads and data size together. The results (see Figure 7.5) show that even with 16 workers, the wait time fraction remains less than 20% of the entire task time, which contains data transform, filtering, hash aggregation, floating number computation, and sorting. Though 16 workers share a single accelerator, the

data transformation part in the worst case (wait completion for the rest 15 workers) still gets net speedups 10.5x versus CPU, and the entire task enjoys >3x speedup.

Figure 7.5: Average Waiting Time.

7.3 Benefits of Software Programmability

The ACCORDA accelerator (UDP) is software-programmable. On the other hand, FPGA's are a popular alternative approach to programmability. We compare these approaches on a set of data transformation and filtering tasks, assessing the performance achieved per unit silicon area (performance density). FPGA's are hardware programmable using LUTs (look up table) and interconnection configuration. LUTs trade silicon area overhead for function flexibility. In contrast, the UDP hardwires an instruction set architecture (basic primitives), and runs software written for that ISA. Thus, UDP uses silicon area efficiently and achieves a high clock rate. Figure 7.6 shows the function density for the decompression and the regular expression matching (1 and 500 patterns). For the FPGA decompression, we use the production Xilinx design with its released performance number [33]. The single-pattern regex matching uses an database-oriented design [109], and the 500-pattern regex uses a network monitoring based design [125]. The area is computed by calibrated LUT measurement [121] (actual hardware used). We exclude the BRAM cost since buffering isn't considered as a source of programmability cost. For the UDP, we present two metrics. The first is SRAM

area (memory) occupied by the acceleration program. The second adds to this the area of the UDP engine. Our results show (see Figure 7.6), performance density for decompression is 19x greater for UDP (9x with UDP engine) compared to FPGA. For regex matching, the UDP performance density ratio is even greater - 160x for the single-pattern and 90x for the 500-pattern design. As a result, software programmability provides significantly higher function density than hardware programmability.

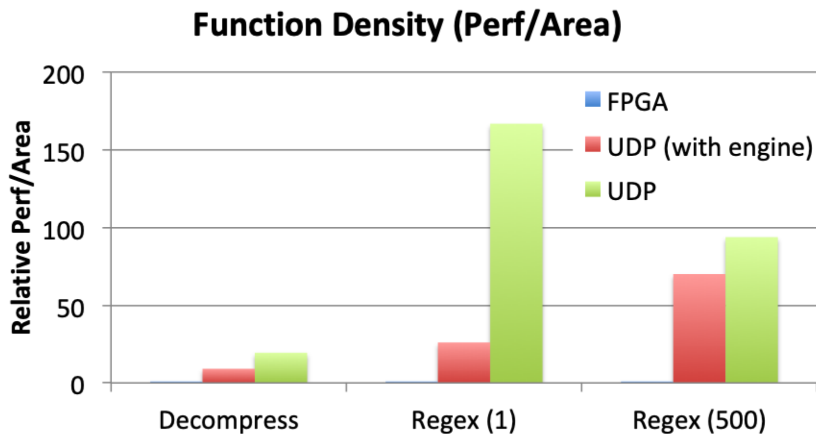


Figure 7.6: UDP Software Programmability vs. FPGA Hardware Programmability.

7.4 ACCORDA Raw Data Performance

We compare the ACCORDA system performance with other leading raw data processing approaches on a set of representative TPC-H queries [31]. The underlying source data stays on disk with its native raw TBL format. We compare against 1) SparkSQL [43] without caching any transformed data, and 2) Sparser [103] on SparkSQL , a raw filtering approach that uses inexpensive partial predicates with SIMD acceleration. Note that using SparkSQL with on-disk raw data approximates the worst case of the RAW approach [37]. In Figure 7.7, ACCORDA achieves 3.3x-13.2x, with a geometric mean of 6x speedups over the SparkSQL baseline, and is up to 6.3x faster than Sparser. Sparser can't apply raw filters in Q1, Q4, Q6 since they don't contain any literal-equal predicates. Sparser benefits Q10, Q12, and Q13

with the help of raw filters, reducing expensive parsing, deserialization, and regex filtering. ACCORDA provides robust acceleration on raw data processing in terms of parsing and deserialization. Among these queries, Q1, Q4, and Q6 are accelerated via simple replacement of data source operators. Moreover, the encoding-extended interface enables aggressive optimization in Q10, Q12, and Q13. They are further accelerated by encoding-based query optimization beyond hardware acceleration on raw data processing.

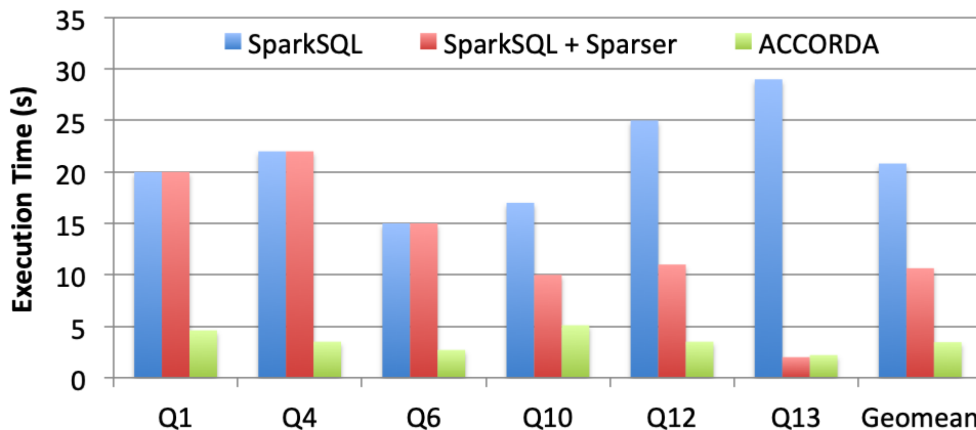


Figure 7.7: Overall System Comparison on Unloaded, On-disk, Raw Data.

Next, we explore the performance improvement from the ATO software architecture. Figure 7.8 shows the query execution time between ACCORDA (UTA) and ACCORDA (UTA+ATO). ACCORDA (UTA) accelerates raw data processing only in parsing and deserialization. Thanks to the ATO software architecture, ACCORDA (UTA+ATO) can apply aggressive encoding-based query optimization in the middle of a query plan, beyond raw data processing acceleration. Among these queries that can benefit from encoding-based optimization in Figure 7.7, Q10 is optimized with attribute group compression, Q12 uses encoding attributes for fast filtering, and Q13 leverages the accelerated regex operator. In Q10, the full optimization brings ACCORDA (UTA+ATO) another 30% performance improvement compared to ACCORDA (UTA), which is the bonus of allowing encoding-based optimization with ATO. In Q12, the time-consuming string filter on the *shipmode* attribute is transformed using dictionary encoding. ATO software architecture provides another 17% speedups on

top of simple hardware acceleration. In Q13, the expensive regex filtering is replaced by the hardware-accelerated regex operator. ACCORDA (UTA) doesn't improve much of the performance because of the expensive cost of CPU-based regex operator. With ATO, the software architecture enables a 11.8x speedup beyond ACCORDA (UTA), removing the performance bottleneck on regex matching. Note that most of the TPC-H queries are in the form of select-project-join whose relations have been properly normalized. As a result, complex data encoding optimizations are limited. We believe future raw data queries will exhibit more patterns for such optimizations to show great power of the ATO software architecture.

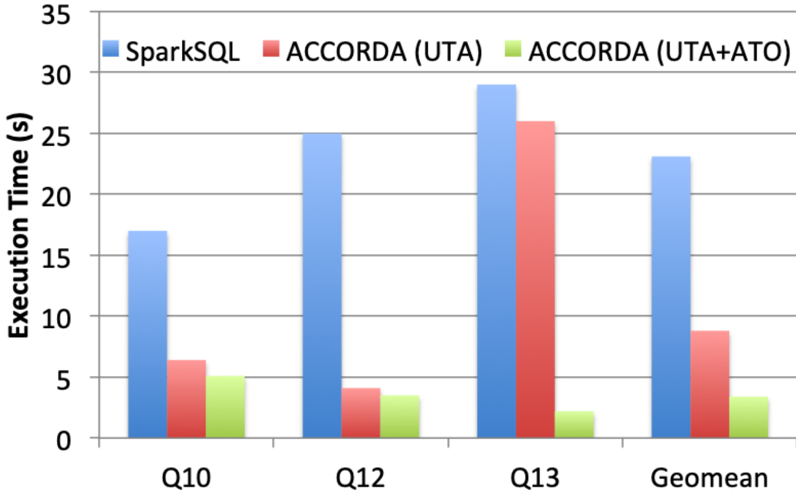


Figure 7.8: Query Execution Time with UTA Acceleration and ATO-enabled Acceleration.

Finally, Figure 7.9 compares ACCORDA that on-demand executes on-disk raw data, against SparkSQL with memory-cached loaded data. In SparkSQL (digested, in-memory data), data has been parsed, transformed, and loaded into its internal columnar-format, and cached into main memory for fast access. As a result, it approximates the best case of the RAW approach [37], which pays no raw data processing because all data accesses hit the memory buffer and only related columns are selected via projection pushdown. As we can see, the performance of ACCORDA (raw, from-disk data) is within 2x as the best case of raw data processing. The hardware acceleration and encoding-based optimization are the two key

reasons. On the other hand, ACCORDA still lacks the optimization of projection pushdown and fetching data directly from main memory. These two optimizations are specially tailored for pre-transformed data to reduce IO cost when bringing data from storage to CPU. They are not practical for data in its native raw format without any transformation paid upfront. Nevertheless, ACCORDA closes the huge performance gap on raw data processing, matching or even exceeding systems with loaded in-memory data.

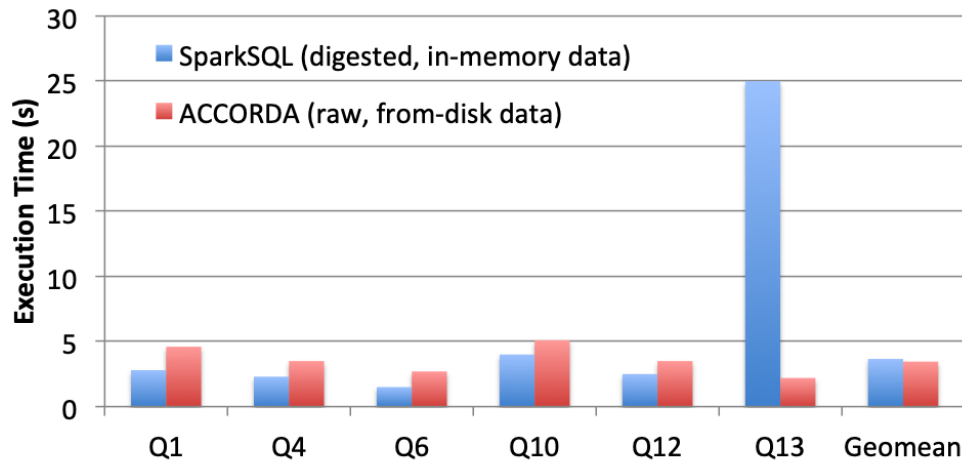


Figure 7.9: Comparing ACCORDA on Raw Data with SparkSQL on Loaded Data.

In summary, direct acceleration on data source operators essentially eliminates the expensive data transformation bottleneck on raw data, enabling ACCORDA to match the performance on systems with loaded in-memory data. Encoding optimization further opens up new performance opportunities. Hardware acceleration alone contributes 1.1x-6.3x improvement, and software elements such as data encoding optimization unlocked by ATO deliver an additional 1.2x-11.8x speedup. Combining UDP acceleration and ATO software architecture, ACCORDA delivers an overall 3.3x-13.2x speedup on raw data compared to SparkSQL with a geometric mean of 6x, up to 6.3x than Sparser, and from 0.5x to 11.4x than SparkSQL with loaded in-memory data.

7.5 Performance Sensitivity Analysis

7.5.1 Data Statistics Sensitivity

Early filtering or predicate pushdown is a classic system optimization approach to filter input as early as possible to avoid the cost of subsequent computation. In raw data processing, early partial filtering with inexpensive predicates [43, 103] are used to avoid data transformation for parsing and deserialization. However, their effectiveness depends on supported functionality, filter selectivity and the cost of predicates. In Table 7.2, we categorize all predicates in TPC-H queries according to their functionality. Since Sparser leverages SIMD acceleration that works with raw bytes, it only supports *Literal Equal* and *Regex Match* predicates. Popular predicates involving multiple columns, range comparison, or IN operator can't apply this technique. On the other hand, ACCORDA provides a general solution by using hardware acceleration on data transformation without imposing any constraints on predicate semantic.

Predicate Type	**Example**	**TPC-H Query**	**Sparser Support**	**ACCORDA Support**
Multi-Column	column1 = column2	Q2, Q4, Q12, Q17, Q20-22	N	Y
Range Compare	column < 1990-10-25	Q1, Q3-8, Q10, Q12, Q14-15, Q18-22	N	Y
IN Operator	column IN (value1, value2,...)	Q16, Q19, Q22	N	Y
Literal Equal	column = 'abcd'	Q2-3, Q5, Q7-8, Q10-12, Q16-17, Q19-21	Y	Y
Regex Match	column LIKE 'abcd%'	Q2, Q9, Q13-14, Q16, Q20	Y	Y

Table 7.2: Predicates in TPC-H Queries

In Figure 7.10, we compare ACCORDA with different raw data processing approaches. SparkSQL (on-demand) processes on-disk raw data. It approximates the worst case of the RAW approach [37]. SparkSQL with Sparser (on-demand) improves the performance by using raw filters before parsing to reduce the amount of work spent on unnecessary data transformation. SparkSQL (pre-loaded) caches loaded data in main memory and applies projection pushdown to minimize the data movement. It approximates the best case of the RAW approach with all data accesses hitting the memory buffer. Figure 7.10 shows three types of predicates – *Range Compare, Literal Equal, Regex Match.* We increase selectivity on

these predicates by either relaxing the filtering condition or modifying the data source.

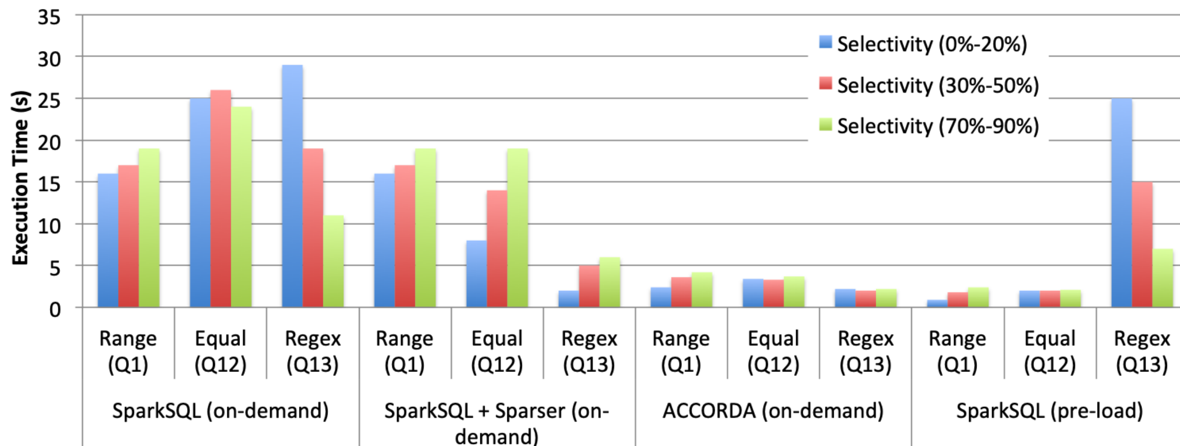
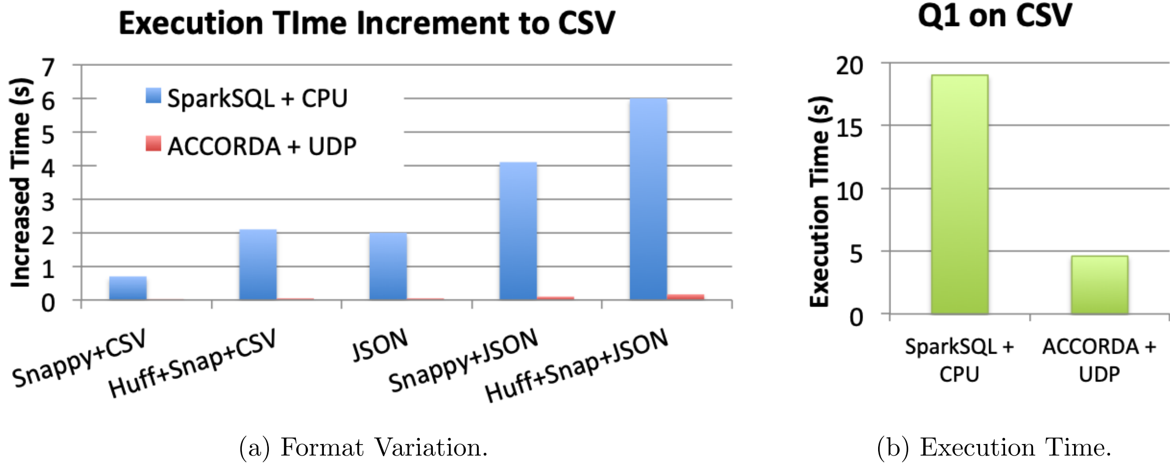


Figure 7.10: Selectivity Impact on Execution Time.

SparkSQL (on-demand) suffers from CPU's slow data transformation thus delivering poor performance in all three queries. For *Regex Match*, the computation complexity decreases with increasing selectivity because less rounds of checks are needed if not match. Sparser leverages raw filters with less data to parse, deserialize, and filter. However, Sparser doesn't support range-based predicates, which results in the same poor performance as SparkSQL (on-demand) in Q1. In Q12 and Q13, Sparser pushes substrings from the predicates before parsing, thus achieves execution time proportional to the data selectivity. Sparser's performance is sensitive to data statistics even if predicates meet the narrow functionality constraint. On the other hand, ACCORDA uses hardware acceleration to support various kinds of predicates on raw data. It reduces data loading cost significantly, approaching the best case – SparkSQL (pre-load). Thanks to the regex acceleration from UDP, ACCORDA even outperforms SparkSQL with loaded, in-memory data in Q13.

In summary, with flexible UDP acceleration across filtering, extraction, and transformation, ACCORDA provides robust high-performance on a wide range of queries on raw data regardless of data statistics, matching or even outperforming systems with loaded, in-memory data.



(a) Format Variation.

(b) Execution Time.

Figure 7.11: Format, System, and Hardware Impact on Execution Time

7.5.2 Format Type and Complexity Sensitivity

Allowing open data formats in the data lake is important. It enables data format to be optimized for storage size, human readability, execution performance, and platform portability. However, the resulting format complexity incurs processing cost, producing a dominant negative impact on raw data processing performance [93, 37, 83, 103]. The format complexity of data lakes and raw data is widely perceived to be growing with data complexity. Figure 7.11a shows TPC-H Q1 execution time on raw data with various input formats from less complex ones (e.g. CSV) to more complicated ones (e.g. Huffman+Snappy+JSON). As format complexity increases, CPU processing cost increases. The execution time of Q1 on *lineitem* in CSV format increases 32% when data is in Huffman+Snappy+JSON format. On the other hand, with UDP acceleration, the execution time only increases 3.7%. ACCORDA can deliver robust high-performance on open data formats. In Figure 7.11b, ACCORDA with UDP delivers robust 4x speedup than Spark SQL with CPU regardless of data source format complexity, eliminating expensive data read and transformation cost. The flexible nature of the software-programmable UDP accelerator provides general fast support across these data formats.

7.6 Summary

Our evaluation shows ACCORDA's speedups on a diverse range of tasks that lie at the heart of data lake analysis, with speedups up to 4.9x on regex matching, 2.6x on decompression, 2x on parsing, and 20x on deserialization when compared to an 8-thread CPU. We further demonstrate the effectiveness of in memory-hierarchy accelerator integration for flexible acceleration and query optimization. The improvements on classic select-project-join queries include 1.6x speedup and 14x less data movement. In ACCORDA, hardware acceleration alone contributes 1.1x-6.3x performance improvement, and software elements such as data encoding optimization unlocked by ATO deliver an additional 1.2x-11.8x speedup. Together, UTA (Chapter 5) and ATO (Chapter 6) enable ACCORDA to deliver robust, high-performance raw data processing with 3.3x-13.2x overall speedups on single-thread performance when compared to the baseline Spark SQL.

CHAPTER 8
SUMMARY AND FUTURE WORK

8.1 Summary

This thesis presents ACCelerated Operators for Raw Data Analysis (ACCORDA), a combined software and hardware approach for fast raw data processing with robust high-performance.

Specifically, ACCORDA applies the Unified Transformation Accelerator (UTA) approach as the key hardware design principle. Key features of the UTA accelerator include high-performance, low-cost (power and area), and software-programmability over a range of branch-intensive workloads. The UTA's generality comes not only from the architecture design, but also from the data encoding UTA accelerates so that applications can flexibly employ beneficial formats for performance. The UTA approach leverages hardware micro-architecture customization and memory system specialization to achieve these goals. We design the Unstructured Data Processor (UDP) to evaluate the full potential and feasibility of the UTA approach. UDP is a hardware architecture designed for general-purpose, high-performance data transformation and processing. It contains four unique architectural features: multi-way dispatch, variable-size symbol support, flexible-source dispatch (stream buffer and scalar registers), and memory addressing. These features, combined with software programmability, a fast scratchpad memory, and 64 parallel lanes enable significant performance benefits over a diverse set of tasks that lie at the heart of ETL, query execution, stream data processing, and intrusion detection and monitoring.

For the ACCORDA's software architecture, we propose Accelerated Transformation Operators (ATO), a software architecture approach that integrates hardware acceleration seamlessly, enabling efficient data encoding optimization and overall query optimization. The ATO approach extends operator interface types with encoding, allowing new accelerated operators to be included in query optimization. Runtime data formats can be transformed to meet the encoding requirements of accelerated operator implementations, and can be

fused to improve data locality and save transformation cost. We further demonstrate that the UTA's in memory-hierarchy accelerator integration is the key for uniform acceleration, and preserving existing system architectures and their corresponding advantages. The in memory-hierarchy integration and efficient data sharing unlock flexible software exploitation of hardware acceleration. The ATO software architecture and the UTA's memory integration empower rule-based optimizers to drive flexible data-encoding based optimization in a query plan.

Together, UTA and ATO enable ACCORDA to deliver robust high-performance raw data processing. Our evaluation shows that UDP achieves a remarkable geometric mean of 1,800-fold increase in performance/watt over a traditional x86 core. At a tiny $3.82mm^2$ area, and 0.86 watt, UDP can be incorporated on a CPU with minimal cost. Moreover, UDP delivers comparable performance of more narrowly specialized accelerators, but its real strength is its flexible programmability across them. Overall, ACCORDA speedups a diverse range of tasks including filtering, extraction, transformation, and query execution that are critical to the performance of data lake analysis. Especially, ACCORDA achieves speedups up to 4.9x on regex matching, 2.6x on decompression, 2x on parsing, and 20x on deserialization, when compared against an 8-thread CPU. The overall ACCORDA system is evaluated using end-to-end TPC-H queries on unloaded data with raw format. Hardware acceleration alone contributes 1.1x-6.3x performance improvement, and software elements such as data encoding optimization unlocked by ATO deliver an additional 1.2x-11.8x speedup. Together, ACCORDA achieves 3.3x to 13.2x speedups on single-thread performance when compared to Spark SQL. We show that this performance benefit is robust across format complexity of query predicates and selectivity (data statistics). Furthermore, ACCORDA robustly matches or even outperforms (by up to 11.4x) prior systems that depend on caching transformed data, while computing on raw, unloaded data.

8.2 Future Work

We outline a few promising research directions for future exploration based on the ACCORDA accomplishments.

8.2.1 UDP Architecture Extension

The Unstructured Data Processor design (Section 5.3.3) demonstrates the feasibility of designing a unified data transformation accelerator with high-performance and low-cost of power and area. The success of the UDP architecture encourages ongoing research on continuous refinement and enhancement of the micro-architecture to support broader applications and to further improve energy efficiency. We believe that UDP 64-bit architecture extension with more powerful function units for computation is a promising next-step for the UTA architecture research.

64-bit Architecture Extension. This is a natural enhancement to the current 32-bit architecture. As we know, modern CPU architectures are 64-bit and many tasks, such as scientific computing, require intensive use of 64-bit double floating point precision.

How to design a 64-bit UDP architecture extension to better support scientific tasks and others that require long register width? This extension provides UDP a better chance for offloading tasks that require CPU register width. With increased architecture width, UDP has faster memory comparison and copy operations, and larger arithmetic domain that better handles data encoding tasks that contain wider data types (e.g. double floating point) with less memory loads and stores. The extension leads to a better performance and less power consumption. To perform a solid study with a deep understanding of the performance and cost trade-off, a concrete circuit implementation is necessary for a systematic evaluation of the 64-bit architecture extension.

Function Unit. The UDP design provides an overall architectural framework for dispatching multiple small code blocks efficiently. The scratchpad memory system, novel

micro-architecture features (e.g. multi-way dispatch and variable-size symbol execution), and MIMD parallelism deliver UDP great performance and energy efficiency on conditional-oriented and sub-byte heavy workloads.

What are the UDP actions to be added for future data lake workloads? Is the existing UDP ISA optimal in terms of performance and cost? Currently, only simple arithmetic, logical, and memory operations are included in UDP actions. According to applications, it might be useful to extend the actions with, for example, special math functions, quantization, or even floating point operations. UDP ISA with powerful action primitives enables mixing complicated computation in data transformation tasks. Powerful actions help to improve performance by reducing the CPU post-processing as well as total instruction count. More importantly, more tasks can run on a standalone UDP without CPU core's involvement in the middle of execution. Once the accelerated kernel is interleaved with CPU cores and the accelerator, usually it means the performance speedup is destroyed by the expensive communication and synchronization across each other. Our discussion implies the UDP architecture, which is customized for branch-intensive tasks, can also be specialized for future important applications by enriching actions in the UDP ISA. The tailored architecture brings even better performance and energy efficiency. Our study on UDP encourages a systematic ISA study to push the performance and efficiency boundary of the UDP architecture.

8.2.2 Domain-Specific Data Transformation Language

The UDP's flexible programmability and dramatic performance improvements open up many opportunities for future research. However, one of the most important problems is to ease the effort of programming these data transformation accelerators. Otherwise, application developers would spend a huge amount of time on describing the intended execution on a UTA accelerator using assembly code. The cumbersome low-level programming support prevents wide adoption of UTA acceleration in a software system as well as future ACCORDA research. We need to raise the abstraction when writing UTA programs.

How should we design new domain-specific languages (DSL) and compilers that provide handy UTA programming support? Then, programmers can simply use high-level language constructs to express the meaning of UTA acceleration without resorting to the current low-level assembly code. Recently, some researchers propose new language support for automaton processing [42, 51]. The proposed syntax and constructs are designed for easy programming and debugging pattern recognition tasks on automaton-based processors [59]. However, UTA and the UDP design are significantly more general and powerful than an automaton accelerator. On UDP, code blocks can be associated with state transitions to introduce arbitrary computations with state traversal in an automaton. As a result, the proposed language semantics of RAPID [42] can't fully express the powerful execution model of UTA and UDP. In fact, our UDP's execution model is based on finite-state transducer [72]. However, existing research rarely focuses on designing a high-level general programming language for transducers that are tailored for data transformation. Such a language can facilitate the development of future UTA applications. Besides the importance, developing a DSL is much easier nowadays with recent advances in the programming language community. For example, Scala is a potential candidate as the language substrate for design and implementation of a UTA domain-specific language.

8.2.3 More UTA Applications

In the thesis, we have demonstrated that the Unified Transformation Accelerator (UTA) accelerates a broad range of raw data processing tasks by designing and evaluating the Unstructured Data Processor (UDP) architecture. Current UDP applications include parsing, extraction, regular expression matching, data encoding, and deserialization. UDP not only supports one or two typical formats for these applications, but is general enough for handling various formats, encodings, and types. Our study leads to an interesting question – *what are other applications UDP can support with high efficiency?* Here, we only list two aspects of applications for inspiration. One is from a software function perspective; the other is from

platform-level hardware deployment.

Software Function. *What are the software functions that can be accelerated by a UTA accelerator?* The ideal applications exercise heavy conditional branches followed by short code blocks. The finite-state transducer [72] falls into this category, whose state transition is implemented as branches, and operations are grouped into a code block. This execution model is a restricted Turing machine that is broad enough to capture applications beyond data encoding tasks. For example, bioinformatics applications such as genome alignment and motif search, graph processing applications such as pattern tree mining, data mining applications such as associated rule mining, can all use their automaton counterpart algorithms [118, 47, 107].

Even for the tasks that we have shown before in the thesis (Section 5.4), we can study their applications more broadly. In high-energy physics, many complex storage formats are used to store the simulation results (e.g., ATLAS). In terms of parsing and extraction, we can study how to apply the UTA accelerator to read the interested data parts out of the encoded data block, and do filtering in-place without processing the entire data block. This technique is known as the predicate and projection pushdown but the difficulty resides in its complex internal encoding. Beyond scientific computing, natural language processing is also a promising area. It requires complicated feature extraction, semantic tagging, and sentence parsing. The UTA accelerator is a good candidate to process these text-based tasks, and is even better for dirty and error-prone corpus that exist in real life. Another interesting direction is to study the benefits of accelerator collaboration with UTA in an accelerator-rich architecture. Typically, UTA would re-organize data formats to serve other data processing accelerators or SIMD units for straight-forward acceleration in a data analysis pipeline.

Accelerator Deployment. *Where should a UTA accelerator locate in a hardware platform?* The UTA's deployment location determines its role in a software system. We can study the specific accelerator deployment that incorporates UDPs (one or several) at various locations in data center infrastructures or database appliances. For example, deploying a

UTA accelerator in a solid-state drive's micro-controller helps to do in-situ filtering and search within the device, exploiting massive SSD internal IO bandwidth. Only data blocks that satisfy the conditions are fetched out from the device. As a result, not only external IO bandwidth (e.g., network, DRAM, and disk) can be saved, but also less energy consumption it takes thanks to less data movement. Besides, putting the accelerator in a memory controller enables application-agnostic data compression in the main memory. All data is compressed in memory and decompressed on-the-fly when brought to the chip using the UTA accelerator embedded in the memory controller. Of course, we need the operating system support to build a working system. Data structures (e.g., page table) that record page positions need to be modified as well.

8.2.4 Novel Data Encoding

The ACCORDA's flexible query optimizations across data encodings and its strong capabilities in data transformation acceleration opens up many opportunities in novel data encoding design for future storage and query processing.

Data Storage Encodings. *How should we design storage formats that exploit ACCORDA's fast data transformation with data statistics and common use cases?* In the Big Data era, there is an increasing amount of data generated every day. The volume is massive and requires expensive disk storage. Traditionally, people use aggressive compression to shrink the data size and reduce the disk storage cost. However, with the growth of data analysis, there is an interesting trade-off space in data storage size and query performance. The data analysis patterns expose access properties such as active hot data and long-term cold data. Leveraging data statistics, hardware features, and common use cases to customize the storage encoding for data analysis shows promise in achieving both performance and size efficiency [94, 88, 35]. BitWeaving [94] explores bit-packed data encoding with SIMD acceleration to speedup search and filtering on encoded data. DataBlock [88] further improves the speed by adding small materialized aggregates (SMA) in a block-oriented data

encoding to facilitate block-level skipping, SIMD filtering, and decoding. Succinct [35] uses a compressed format with built-in suffix tree to support direct string search on compressed storage. These approaches avoid full data decoding for performance-critical operations to achieve performance. However, the full scope of customized data encoding is still largely limited by the underlying architecture supports (e.g., SIMD).

With the ACCORDA approach, both software and hardware, storage systems are free to explore more aggressive data encodings. For example, we can couple multiple encoding algorithms to achieve better compression ratio, build application-specific dictionaries for each block in bit-level, introduce computation on encoded data with minimal transformation, etc. The key idea is to leverage the powerful ACCORDA accelerator (UTA) to do the heavy-lifting data transformation that converts the storage format into an intermediate representation that is beneficial for common query operations, such as predicate and projection pushdown. Because the UTA accelerator can efficiently probe into the encoded format, parse nested structures, and navigate itself to the desired data attributes with high-performance, system designers can explore storage encodings with complex internal structures that contains, for example, tag-value pairs and various zones of compression type. In short, the ACCORDA approach enables storage format exploitation with extreme complexity in the structures.

Encodings for Fast Computation. *What are the aggressive query optimizations for open data types with encodings and new computation operators using specific data encodings?* Researchers have shown that customized data encodings benefit certain operations, such as filtering with Run-length encoding (RLE) or bit-mapping [34], and sparse matrix multiplication with Compressed Sparse Row (CSR) [60]. Applying encoding-specific operators in a query plan typically requires a holistic optimization framework. Using the ACCORDA ATO approach, system designers are free to include arbitrary query optimizations around data encodings, thanks to ATO's seamless integration of data encodings in the operator interface. Though previous approaches [60, 34, 53] utilize customized data encodings in query execution, they generally fix the format during processing because of the expensive transformation

cost. With the ACCORDA UTA approach, such concerns no longer stand. The ACCORDA accelerator provides robust high-performance data transformation. Besides, the in memory-hierarchy integration makes it easy for a fine-grained collaboration between accelerators and CPU hosts. Therefore, researchers can look beyond and propose novel encodings that are not possible before due to cost, and develop corresponding query optimization on encodings during runtime. Another possibility is to let the UTA accelerator organize and prepare the data formats for others such as deep learning or database accelerators. These accelerators usually rely on a clean and well-structured format tied to a specific acceleration to get good performance. The end of Moore's Law encourages a heterogeneous hardware architecture for general-purpose computing [54]. Such heterogeneity is likely to require specific data types or encodings for each accelerator to fully unleash its power of hardware customization. In summary, the ACCORDA approach paves the way for ongoing research on computer architecture and database encoding innovations in the post Moore's Law era.

REFERENCES

[1] Apache parquet c++ library. https://github.com/apache/parquet-cpp.

[2] Berkeley big data benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[3] Boost c++ library. http://www.boost.org/.

[4] Cacti 6.5. http://www.cs.utah.edu/~rajeev/cacti6/.

[5] Canterbury corpus. http://corpus.canterbury.ac.nz/.

[6] Cavium nitrox dpi l7 content processor family. http://www.cavium.com/processor_NITROX-DPI.html.

[7] Chicago city crime report. http://data.cityofchicago.org.

[8] Chicago city restaurant inspection. http://data.cityofchicago.org.

[9] Gnu scientific library. https://www.gnu.org/software/gsl/.

[10] Google snappy compression library. https://github.com/google/snappy.

[11] Gpu databases-what, why, and how. https://www.kinetica.com/gpu-database/.

[12] The ibm power edge of network processor. http://www.cercs.gatech.edu/iucrc10/material/franke.pdf.

[13] Ieee 754 floating-point format. http://grouper.ieee.org/groups/754/.

[14] Intel advanced vector extensions. https://software.intel.com/en-us/isa-extensions/intel-avx.

[15] Intel chipset 89xx series. http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/scaling-acceleration-capacity-brief.pdf.

[16] Intel communications chipset 8955. http://ark.intel.com/products/80372/Intel-DH8955-PCH.

[17] Intel hyperscan. https://github.com/01org/hyperscan.

[18] Intel xeon processor e5620 specification. http://ark.intel.com/products/47925/Intel-Xeon-Processor-E5620-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI.

[19] Intel64 and ia-32 architectures. http://www.intel.com/content/dam/www/public/us/en/documents/manuals.

[20] Keysight cx3300 appliance. http://www.keysight.com/en/pc-2633352/device-current-waveform-analyzers?cc=US&lc=eng.

[21] libcsv c library. https://sourceforge.net/projects/libcsv/.

[22] libhuffman c library. https://github.com/drichardson/huffman.

[23] M7: Next generation sparc. http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/migration/m7-next-gen-sparc-presentation-2326292.html.

[24] Neon - arm. https://www.arm.com/products/processors/technologies/neon.php.

[25] New york city taxi report. http://www.andresmh.com/nyctaxitrips/.

[26] Parsec 3.0. http://parsec.cs.princeton.edu/.

[27] Postgresql database. https://www.postgresql.org/.

[28] Sc16 invited talk: Memory bandwidth and system balance in hpc systems. http://sites.utexas.edu/jdm4372/2016/11/22/sc16-invited-talk-memory-bandwidth-and-system-balance-in-hpc-systems/.

[29] Sparc m7 die size (wikipedia). https://en.wikipedia.org/wiki/SPARC.

[30] Texas advanced computing center. https://www.tacc.utexas.edu/systems/fabric.

[31] Tpc-h benchmark. http://www.tpc.org/tpch/.

[32] Xeon+fpga platform for the data center. https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdfhttps://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf.

[33] Zipaccel-d gunzip/zlib/inflate data decompression core. http://www.cast-inc.com/ip-cores/data/zipaccel-d/cast-zipaccel-d-x.pdf.

[34] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, June 2006.

[35] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[36] S. R. Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, C. Balkesen, G. Giannikis, C. Roth, N. Agarwal, et al. A many-core architecture for in-memory data processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 245–258. ACM, 2017.

[37] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: efficient query execution on raw data files. In *SIGMOD*, pages 241–252. ACM, 2012.

[38] J. Albericio, J. S. Miguel, N. E. Jerger, and A. Moshovos. Wormhole: Wisely predicting multidimensional branches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 509–520. IEEE Computer Society, 2014.

[39] J. Allen, M. Boliek, E. L. Schwartz, and D. Bednash. Huffman decoder architecture for high speed operation and reduced memory, June 28 1994. US Patent 5,325,092.

[40] M. S. B. Altaf and D. A. Wood. Logca: a performance model for hardware accelerators. *IEEE Computer Architecture Letters*, 14(2):132–135, 2015.

[41] K. Angstadt, A. Subramaniyan, E. Sadredini, R. Rahimi, K. Skadron, W. Weimer, and R. Das. Aspen: A scalable in-sram architecture for pushdown automata. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 921–932. IEEE, 2018.

[42] K. Angstadt, W. Weimer, and K. Skadron. Rapid programming of pattern-recognition processors. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, Apr. 2016.

[43] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *SIGMOD*. ACM, 2015.

[44] T. Azim, M. Karpathiotakis, and A. Ailamaki. Recache: reactive caching for fast analytics over heterogeneous data. *Proceedings of the VLDB Endowment*, 11(3):324–337, 2017.

[45] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103. ACM, 2010.

[46] C. G. Bell. *What Have We Learned from the PDP-11?* Springer Netherlands, Dordrecht, 1977.

[47] C. Bo, V. Dang, E. Sadredini, and K. Skadron. Searching for potential grna off-target sites for crispr/cas9 using automata processing across different platforms. In *2018 IEEE*

International Symposium on High Performance Computer Architecture (HPCA), pages 737–748. IEEE, 2018.

[48] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[49] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[50] 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*, PACT '14, Aug. 2014.

[51] M. Casias, K. Angstadt, T. T. II, K. Skadron, and W. Weimer. Debugging support for pattern-matching languages and accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languag es and Operating Systems*, ASPLOS '19, 2019.

[52] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, H. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *MICRO'16*. ACM/IEEE, 2016.

[53] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. *ACM SIGMOD Record*, 30(2):271–282, 2001.

[54] A. A. Chien, T. Thanh-Hoang, D. Vasudevan, Y. Fang, and A. Shambayati. 10x10: A case study in highly-programmable and energy-efficient heterogeneous federated architecture. *ACM SIGARCH Computer Architecture News*, 43(3):2–9, 2015.

[55] K. Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage.* " O'Reilly Media, Inc.", 2013.

[56] W. J. Dally, U. J. Kapasi, B. Khailany, J. H. Ahn, and A. Das. Stream processors: Programmability and efficiency. *Queue*, 2(1), Mar. 2004.

[57] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[58] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[59] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE TPDS'14*, Aug. 2014.

[60] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed linear algebra for large-scale machine learning. *Proceedings of the VLDB Endowment*, 9(12):960–971, 2016.

[61] Y. Fang and A. A. Chien. Udp system interface and lane isa definition. Technical report, University of Chicago, 2017.

[62] Y. Fang, A. A. Chien, A. Lehane, and L. Barford. Performance of parallel prefix circuit transition localization of pulsed waveforms. In *2016 IEEE International Instrumentation and Measurement Technology Conference Proceedings*, May 2016.

[63] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien. Fast support for unstructured data processing: The unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, Dec. 2015.

[64] Y. Fang, A. Lehane, and A. A. Chien. Effclip: Efficient coupled-linear packing for finite automata. *University of Chicago Technical Report, TR-2015-05*, May 2015.

[65] Y. Fang, R. ur Rasool, D. Vasudevan, and A. A. Chien. Generalized pattern matching micro-engine. In *4th Workshop on Architectures and Systems for Big Data (ASBD) held with ISCA'14*, 2014.

[66] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien. Udp: a programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 55–68. ACM, 2017.

[67] Z. Feng, E. Lo, B. Kao, and W. Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 31–46. ACM, 2015.

[68] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '15, May 2015.

[69] N. Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[70] V. Gogte, A. Kolli, M. J. Cafarella, L. D. Antoni, and T. F. Wenisch. Hare: Hardware accelerator for regular expressions. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2016.

[71] G. Graefe. Volcano/spl minus/an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[72] B. B. Grathwohl, F. Henglein, U. T. Rasmussen, K. A. Søholm, and S. P. Tørholm. Kleenex: Compiling nondeterministic transducers to deterministic streaming trans-

ducers. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, Jan. 2016.

[73] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, et al. Applied machine learning at facebook: a datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[74] T. Heil, A. Krishna, N. Lindberg, F. Toussi, and S. Vanderwiel. Architecture and performance of the hardware accelerators in ibm’s poweren processor. *ACM Trans. Parallel Comput.*, 1(1), May 2014.

[75] S. Hoffman. *Apache Flume: distributed log collection for Hadoop*. Packt Publishing Ltd, 2013.

[76] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. 1969.

[77] Z. István, G. Alonso, M. Blott, and K. Vissers. A flexible hash table design for 10gbps key-value stores on fpgas. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8. IEEE, 2013.

[78] Z. Istvan, L. Woods, and G. Alonso. Histograms as a side effect of data movement for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, June 2014.

[79] D. A. Jiménez. Piecewise linear branch prediction. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 382–393. IEEE Computer Society, 2005.

[80] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a

tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[81] K. Kara, J. Giceva, and G. Alonso. Fpga-based data partitioning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 433–445. ACM, 2017.

[82] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast queries over heterogeneous data through engine customization. *Proceedings of the VLDB Endowment*, 9(12):972–983, 2016.

[83] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive query processing on raw data. *Proceedings of the VLDB Endowment*, 7(12):1119–1130, 2014.

[84] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2), Mar. 2001.

[85] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 468–479. ACM, 2013.

[86] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *SIGCOMM Comput. Commun. Rev.*, 36(4), Aug. 2006.

[87] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[88] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data blocks: hybrid oltp and olap on compressed storage using both vectorization and

compilation. In *Proceedings of the 2016 International Conference on Management of Data*, pages 311–326. ACM, 2016.

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

[90] F. Li, L. Chen, A. Kumar, J. F. Naughton, J. M. Patel, and X. Wu. When lempel-ziv-welch meets machine learning: A case study of accelerating machine learning using coding. *arXiv preprint arXiv:1702.06943*, 2017.

[91] P. Li, J. L. Shin, G. Konstadinidis, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. Masleid, C. Zheng, Y. D. Lin, et al. 4.2 a 20nm 32-core 64mb l3 cache sparc m7 processor. In *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International*, pages 1–3. IEEE, 2015.

[92] Y. Li, C. Chasseur, and J. M. Patel. A padded encoding scheme to accelerate scans by leveraging skew. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1509–1524. ACM, 2015.

[93] 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*, 10(10):1118–1129, 2017.

[94] Y. Li and J. M. Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300. ACM, 2013.

[95] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh. In-rdbms hardware acceleration of advanced analytics. *Proceedings of the VLDB Endowment*, 11(11):1317–1331, 2018.

[96] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *PVLDB*, 2010.

[97] D. Molka, D. Hackenberg, and R. Schöne. Main memory and cache performance of intel sandy bridge and amd bulldozer. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, page 4. ACM, 2014.

[98] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *Proceedings of the VLDB Endowment*, 6(14):1702–1713, 2013.

[99] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *Proceedings of the VLDB Endowment*, 6(14), Sept. 2013.

[100] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, Mar. 2014.

[101] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, Mar. 2014.

[102] J. Ouyang. Xpu: A programmable fpga accelerator for diverse workloads. In *2017 IEEE Hot Chips 29 Symposium*, 2017.

[103] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia. Filter before you parse: faster analytics on raw data with sparser. *Proceedings of the VLDB Endowment*, 11(11):1576–1589, 2018.

[104] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-

memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508. ACM, 2015.

[105] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[106] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: challenges in and avenues for cmp scaling. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 371–382. ACM, 2009.

[107] E. Sadredini, R. Rahimi, K. Wang, and K. Skadron. Frequent subtree mining on the automata processor: challenges and opportunities. In *Proceedings of the International Conference on Supercomputing*, page 4. ACM, 2017.

[108] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira. Accelerating business analytics applications. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, Feb. 2012.

[109] D. Sidler, Z. István, M. Owaida, and G. Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 403–415. ACM, 2017.

[110] D. Song and S. Chen. Exploiting simd for complex numerical predicates. In *Data Engineering Workshops (ICDEW), 2016 IEEE 32nd International Conference on*, pages 143–149. IEEE, 2016.

[111] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[112] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB*. VLDB Endowment, 2005.

[113] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[114] A. Subramaniyan and R. Das. Parallel automata processor. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 600–612. ACM, 2017.

[115] T. Thanh-Hoang, A. Shambayati, and A. A. Chien. A data layout transformation (dlt) accelerator: Architectural support for data movement optimization in accelerated-centric heterogeneous systems. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Mar. 2016.

[116] H. Tung, A. Shambayati, F. Yuanwei, H. Hoffmann, and A. Chien. Does arithmetic logic dominate data movement? a systematic comparison of energyefficiency for fft accelerators. *Proc. of the IEEE Application-specific Systems, Architectures and Processors (ASAP)*, 2015.

[117] J. Van Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2012.

[118] K. Wang, E. Sadredini, and K. Skadron. Sequential pattern mining with the micron automata processor. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 135–144. ACM, 2016.

[119] J. Webber and I. Robinson. *A programmatic introduction to neo4j*. Addison-Wesley Professional, 2018.

[120] B. Wheeler. Titan ic floats 100gbps reg-ex engine. `https://www.linleygroup.com/newsletters/newsletter_detail.php?num=5924&year=2018&tag=3`.

[121] H. Wong, V. Betz, and J. Rose. Comparing fpga vs. custom cmos and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 5–14. ACM, 2011.

[122] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 249–260. ACM, 2013.

[123] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, Mar. 2014.

[124] W. Xu, Z. Feng, and E. Lo. Fast multi-column sorting in main-memory column-stores. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1263–1278. ACM, 2016.

[125] Y.-H. Yang and V. Prasanna. High-performance and compact architecture for regular expression matching on fpga. *IEEE Trans. Comput.*, 61(7), July 2012.

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

[127] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *ACM SIGARCH Computer Architecture News*, volume 20, pages 124–134. ACM, 1992.

[128] X. Yu and M. Becchi. Gpu acceleration of regular expression matching for large datasets: Exploring the implementation space. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, May 2013.

[129] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, 2013.

[130] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[131] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438. ACM, 2013.

[132] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. *Proceedings of the VLDB Endowment*, 11(11):1522–1535, 2018.

[133] Z. Zhao and X. Shen. On-the-fly principled speculation for fsm parallelization. In *Proc. of ASPLOS'15*, pages 619–630. ACM, 2015.

[134] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong. 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, Feb. 2012.

[135] M. Zukowski, N. Nes, and P. Boncz. Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th international workshop on Data management on new hardware*, pages 47–54. ACM, 2008.