Thesis Proposal:
Extreme Acceleration and Seamless Integration of Raw Data Analysis

Yuanwei Fang

Department of Computer Science
University of Chicago
Chicago, IL, 60637

Thesis Committee

Andrew A. Chien, Chair
Ian T. Foster
Aaron J. Elmore
Michael J. Franklin
Abstract

New sources of “big data” such as the Internet, mobile applications, data-driven science and large-scale sensors (IoT) are driving the demand for growing computing performance. The efficient analysis of data in native raw formats in real-time is increasingly important due to rapid data generation, popular demands for analytics and quick insights to derive immediate responses. Traditional data processing systems can deliver high performance on loaded data, but transforming raw data into internal system formats is computationally expensive. Data transformations rather than arithmetic operations dominate the task, and such transformation is a critical barrier to increased data processing performance.

We propose a two-part software/hardware approach, namely, the unified transformation accelerator (hardware) and accelerated transformation operators (software), to accelerate data analytics on unloaded raw data and exceed the performance of structured data analytical systems. Together, they enable real-time decision making and fast knowledge exploration on messy, heterogeneous, and ad-hoc raw data.

The unified transformation accelerator (UTA) creates new and flexible architecture support for analytical workloads. Exploiting customization and local memory for efficiency and MIMD parallelism for flexibility and performance, we designed a novel hardware architecture, the Unstructured Data Processor (UDP). UDP is a programmable general-purpose data transformation accelerator, customized for data analytics. We completed the instruction set architecture design, the micro-architecture implementation, the software toolchain (compiler, linker, loader), and the simulation infrastructures. The UDP has four unique features for performance and generality: multi-way dispatch, variable-size symbol, flexible-source dispatch, and flexible addressing. Extensive evaluation of data transformation kernels, which ranges from compression to pattern matching, shows UDP achieves 20x average speedup and 1900x energy efficiency when compared with an 8-thread CPU processor. The UDP ASIC implementation is > 100x less power and area than a single CPU core.

The accelerated transformation operators (ATO) is a software framework for integrating hardware accelerations into data analytical systems in a pervasive and flexible fashion. It allows arbitrary data representations on query plan edges and arbitrary use of hardware acceleration within any operator implementations. This approach preserves full operator composition and query optimization capabilities, and requires essentially no front end and execution engine changes. We build the ACCORDA (Accelerated Operators for Raw Data Analysis) system, by extending the state-of-art distributed analytical system (SparkSQL) with the ATO approach. We will study the performance benefit of ACCORDA over a traditional analytical system (SparkSQL) using real-world end-to-end query workloads. We expect the unloaded raw data processing speed of the ACCORDA system to match or even exceed SparkSQL with loaded data. Furthermore, we describe the research plan and the full thesis outline in the proposal. The complete experiment results will be presented in the final thesis.
# Contents

1 Thesis Statement 5

2 Introduction 5
   2.1 Emerging Applications 5
   2.2 Extract Transform Load 5
   2.3 Rise of the Data Lake 6
   2.4 New Capabilities 7
   2.5 Thesis Organization 8

3 Related Work 9
   3.1 Hardware Accelerated Query Processing 9
   3.2 Optimizing ETL in Databases 10
   3.3 Data Representations for Fast Query Execution 12

4 Problem and Research Approach 12
   4.1 Problem 12
   4.2 High-Level Approach 13
   4.3 Benefits of Our Approach 14
   4.4 Research Questions 15
      4.4.1 Part 1: Unified Transformation Accelerator 15
      4.4.2 Part 2: Accelerated Transformation Operators 15
   4.5 Detailed Strategies 16
      4.5.1 Cracking The Part 1 Questions 16
      4.5.2 Cracking The Part 2 Questions 18
   4.6 Success Measure Implications 20

5 Research Plan 20
   5.1 Project Status 20
   5.2 Proposed Timeline 21
   5.3 Expected Research Contributions 22
   5.4 Full Thesis Outline 23

6 Initial Results 24
   6.1 Addressing Research Questions 24
   6.2 Data Transformation Workloads 25
   6.3 Data Transformation CPU Bottleneck 25
   6.4 The Unstructured Data Processor 26
   6.5 The ACCORDA System 26
   6.6 Evaluation 27
      6.6.1 Data Transformation Micro-benchmarks 28
      6.6.2 UDP Accelerator Metrics 29
      6.6.3 UDP Accelerator Performance 29
      6.6.4 UDP Hardware Implementation 32
      6.6.5 Preliminary ACCORDA Performance on TPCH 33
6.7 Discussion
1 Thesis Statement

We propose 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 re-purposed 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.

2 Introduction

2.1 Emerging Applications

With the rise of the Internet, mobile applications, data-driven science, and large-scale sensors, data analysis for large, messy, and diverse data (e.g. Big Data) is an essential driver of computing performance. The vast volume of data generated from various sources makes real-time in-situ data processing and timely complex batch analysis from data in native raw formats 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). Furthermore, advances in large-scale machine learning, deep learning, and artificial intelligence enable making accurate predictions based on historical data. Transforming raw data formats for fast computation is expensive. Data manipulation - transformation - movement, rather than arithmetic speed, is the primary barrier to continued performance scaling for data processing.

2.2 Extract Transform Load

For data warehousing, Extract-Transform-Load (ETL) is a process that transforms external data into internal forms such as columnar formats or bitmap encodings, as shown in Figure 1. Extract-Transform-Load (ETL) tasks include decompression, parsing record delimiters, tokenizing attribute values, and deserialization (decoding specific formats and validation of domains such as dates). Traditional data analytical systems offer high performance on loaded data after ETL. Transformed formats allow fast query execution. ETL is not a significant overhead for traditional databases [61]; frequent reuse of the internal data amortizes the ETL computation cost. Furthermore, data volume generated with the real-time analysis requirement was relatively small ten years ago. As a result, decades of database research mostly focused on performance optimization for query execution on internal loaded data; for example, deserialized binary formats for parallel SIMD computation [39, 55, 57], columnar representations for fast and efficient scans [26, 64], and compression for memory capacity and bandwidth [26, 32, 52]. These techniques depend on the assumption that massive data volume has been loaded and transformed. Little research has paid attention to the performance problem when querying external raw data generated by emerging applications.
2.3 Rise of the Data Lake

Recently, there is a trend for a new data analytics paradigm. Companies and organizations put data from rich sources (e.g., purchased from other providers, internal employee data, or database dumps) for future federated analysis in a data repository, called the data lake. A data lake is a storage repository that holds a vast amount of raw data in its native format (dirty, unstructured, ad-hoc) until it is needed. When a business question arises, the data lake can be queried for relevant data, and that smaller set of data can then be analyzed to help answer the question. Usually, the way to process the data is unknown at the point of its arrival at the data lake. Therefore, most data stay in their native complex structures and formats instead of in a transformed clean SQL-like data model. As shown in Figure 2, many applications benefit from the federated richness in the data lake such as machine learning, data mining, artificial intelligence, ad-hoc query, and data visualization.

Even if the data model can be determined beforehand, loading data from the data lake is still very expensive. Unlike databases have internal storage systems, the data lake appears to be external to the data analytical systems. Moving external data into internal database storage causes long batch ETL time and extra storage cost. The traditional usage model does not suit current needs for instant decisions and data exploration in the big data era when business value depends on timely decision-making and new data is continually generated and waiting to be analyzed. As a result, companies cannot afford the cost of loading all the data prior to real-time analysis.

Figure 2: Data Lake Enables Federated Analysis.
The need for real-time, high-performance, in-situ data processing in the data lake is increasingly needed, but traditional data systems deliver poor performance on conventional CPU architectures. This is one of the key challenges for in-memory data analytical systems [29, 72] and parallel databases such as Impala [30] and Vertica [51]. Studies [28,38,48] indicate that data format transformation for efficient computation is the critical performance bottleneck for analytical systems even with highly optimized data loading. Studies [28, 38] also show that the data transformations required in ETL when data is pulled from the data lake is 10x slower than the disk IO speed, and orders of magnitude slower than the memory speed. In short, high-performance data transformation is vital for flexible data lake analysis, but current architectures and systems deliver poor performance.

2.4 New Capabilities

This thesis focuses on the problem of high-performance query execution on external raw data (e.g. the data lake). High-performance raw data analysis will enable several new capabilities which are prohibited in practice now due to high cost. Figure 3 portrays new data analysis paradigm empowered by our proposed technology. The thesis will enable the shift from traditional, routined, and batch-oriented queries on standardized data structures with predefined usage models to future customized, interactive, and dynamic analysis on federated, application-specific data representations with unclear usage beforehand in a cost-effective way for large-scale data volume. Next, I describe two potential real-life examples that can take advantage of the new paradigm.

The first example is related to security. Imagine you want to find someone as soon as possible and predict his/her reaction towards some events according to the personality. However, you don’t know the exact information such as name, appearance and background, but with only a few vague and approximate clues. With the incoming ubiquitous Internet of Things (IoT), Artificial Intelligence (AI), and our accelerated raw data analysis, it is possible to leverage the power of rich data coupled with a federated analysis in the data lake to provide a solution to address this need. In the future, many physical devices will offer automatic services (e.g., self-driving cars) without human intervention. These devices record massive machine logs and sensor data periodically and upload them to the data lake. To quickly narrow down the candidates, we can query those newest machine logs and sensor data given a set of approximate criteria such as race, weight, and travel locations. Other data sources might suddenly become useful when you roughly figure out who the person is for modeling the person’s characteristic such as social network posts, public talks, previous working experience and so on. Since various data sources may exist in different structures and formats, and their indices and metadata might be missing, incomplete, or out-of-date, regarding newly generated data, search and transformation speed limits the amount of information and knowledge we can query. However, using our accelerated raw data analysis (assume 100x speedup), we can search 1 TB unstructured data instead of 10 GB in one second with 100 cloud servers. Moreover, we can interactively issue ad-hoc follow-up queries on demand and look into the newest possible unstructured data in realtime.

Another motivating example concerns the real-time advertisement. Imagine we are in a shopping mall. One customer walks in and logs into his/her shopping mall account, which links to a social network account (e.g., Facebook, Twitter). During the stay in the mall, data analysis can be performed using the most recent social posts, including their friends’ public posts, to predict which cool products the customer might be interested in. Combined with the customer’s real-time locations in the shopping mall (categories of the glossary region) and the historical purchase records in the database, the product recommendations can be more accurate with the cost of more computational power on heterogeneous data representations. More importantly, these analysis tasks can be fully customized and dynamic to maximize the effectiveness of data richness in the data lake, not like the traditional routinely executed regular analysis. All these background data analytical tasks must be done within a short time period before the customer walks out of the store. With
our high-performance raw data analysis capabilities, all these computations are completed instantly and can deliver advertisements in time.

In particular, the thesis demonstrates an accelerated analytical system that makes these new capabilities possible, as shown in Figure 4. The accelerated system preserves existing software architectures such as a query processor, storage hierarchy, and distributed management, but achieves a significant speed advantage over raw data processing through data transformation acceleration, which comes from clever algorithms, customized data structures or hardware accelerators. For raw data analysis, the success of the accelerated analytical system is to enable fast query execution on unloaded raw data, whose performance matches or exceeds traditional analytical systems operating on loaded data.

2.5 Thesis Organization

The remainder of the proposal is organized as follows. In Section 3, we provide a detailed survey of the research literature related to our work. Next, we present the formal problem description followed by a list of derived research questions and the approaches. Section 5 sketches a research plan to complete the full thesis. The initial results are discussed in Section 6.
3 Related Work

3.1 Hardware Accelerated Query Processing

With the end of Moore’s law and Dennard scaling, data centers will become heterogeneous in the future, equipped with various accelerators (e.g. ASICs, FPGAs, GPUs) and advanced memory technologies to sustain the performance improvement (Figure 5). Unsurprisingly, one thread of invention is high-performance hardware architecture support for data analytics. Several customized accelerator designs have been proposed to remove various computation bottlenecks in query processing [27, 41, 45, 46, 49, 54, 62, 69, 70]. Researchers explore customized hardware accelerators for common operations in SQL such as join [70], partitioning [27, 69, 70], sorting [70], aggregation [70], regex search [41, 62, 70], and index traversal [49]. However, it is challenging to deploy one accelerator (or functional unit) for each possible application/algorithm because the chip design and the silicon cost are expensive, the architecture is not scalable when new algorithms emerge, and the sophisticated hardware/software interface is a disaster for software maintenance and compiler support. To our best knowledge, the closet work is the Oracle SparcM7 DAX [54]. It accelerates data transformations for column scan operations on compressed formats to save the memory bandwidth and reduce the data movement. The DAX accelerator has multiple hardwired functional units, one for each format, to support conversions such as Huffman Coding, OZIP, and RLE. 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 re-purposed with the same piece of hardware and matches the performance and efficiency of DAX at the same time. Exploiting the software programmability, new acceleration programs can process broad or even future formats on our accelerator. Automata and regex accelerators [34, 37, 41, 65] are good candidates for pattern matching but the lack of support for analysis operations after matches constraints their applicability. Our approach not only performs well on pattern matching but also deliver high performance on various data analysis and transformation workloads. Table 1 summarizes the broad applicability of our approach (UDP) compared with several state-of-art hardwired fixed data transformation accelerators. The UDP is discussed in Section 4.2 and Section 6 in detail. Table 2
### Table 1: Coverage of Transformation/Encoding Algorithms: Accelerators and UDP.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All listed</td>
<td>None</td>
<td>None</td>
<td>All listed</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>All listed</td>
</tr>
<tr>
<td>Compression / Decompression (DEFLATE, Snappy, Xpress, LZF, ...)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding / Decoding (RLE, Huffman, Dictionary, Bit-pack, ...)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parsing (CSV, JSON, XML, ...)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pattern Matching (DFA, D2FA, NFA, c-NFA, ...)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histogram (Fixed-size bin, Variable-size bin, ...)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Programmable accelerators like GPUs deliver high performance by exploiting data parallelism. However, analytical data transformation workloads are full of multi-target branches, which leads to GPU threads diverge quickly and reduce GPU's execution efficiency. Our approach shares the common advantage of GPU on sufficient MIMD parallelism for performance but differs from the core micro-architecture design. We target at branch-intensive and highly dependent FSM-based workloads, which perform poorly on GPUs.

Recent advances [31,60] in FPGA lead to multiple FPGA acceleration techniques for database operations [44,45,46,62]. Computation kernels are implemented on FPGA as co-processors. Using FPGAs for irregular parallelism brings certain speedup compared to CPUs. However, adding each ad-hoc circuit implementation for each algorithm on FPGA is impractical due to FPGA's scarce application-reserved resources [60]. As a result, FPGA approaches have to reconfigure when there are various computation kernels if the entire circuit implementations can’t fit on a single FPGA chip. The reconfiguration is costly and introduces an extra performance penalty. Even worse, computations are interleaved (volcano-iterator model) in database query execution, which requires frequent FPGA reconfigurations. Another drawback is energy efficiency; a proper designed ASIC/ASIP IP is around 10x more energy efficient [68]. We differ from the FPGA approach fundamentally. Our approach achieves flexibility by carefully designing the ISA for software programmability while the FPGA approaches rely on hardware programmability for chip reconfiguration.

### 3.2 Optimizing ETL in Databases

Recent work [28, 29, 47, 48, 56, 58] improve analytics performance on external raw data through ETL acceleration. There are several software-accelerated ETL approaches using SIMD instructions. InstantLoad
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UAP [37]</td>
<td>String Mat. (ADFA)</td>
<td>String Mat. (ADFA)</td>
<td>38</td>
<td>0.58</td>
<td>0.56W</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td>Regex Mat. (NFA)</td>
<td>Regex Mat. (NFA)</td>
<td>15</td>
<td>0.48</td>
<td>0.56W</td>
<td>0.32</td>
</tr>
<tr>
<td>Intel Chipset 89xx [13]</td>
<td>DEFLATE</td>
<td>Snappy comp.</td>
<td>1.4</td>
<td>2.1</td>
<td>0.20W</td>
<td>0.50</td>
</tr>
<tr>
<td>Microsoft Xpress² [40]</td>
<td>Xpress</td>
<td>Snappy comp.</td>
<td>5.6</td>
<td>0.54</td>
<td>108K ALM</td>
<td>- (FPGA)</td>
</tr>
<tr>
<td>Oracle Sparc M7³ [54]</td>
<td>DAX-RLE, Huffman, Bit-pack, Ozip</td>
<td>Huffman, RLE, Dictionary</td>
<td>11</td>
<td>1.4</td>
<td>1.6mm² (Area)</td>
<td>0.56 (Area Eff)</td>
</tr>
<tr>
<td>IBM PowerEN⁴ [42]</td>
<td>XML Parse</td>
<td>CSV Parse</td>
<td>1.5</td>
<td>2.9</td>
<td>1.95W</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Compress</td>
<td>DEFLATE</td>
<td>1.0</td>
<td>3.0</td>
<td>0.30W</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>Decomp.</td>
<td>INFLATE</td>
<td>1.0</td>
<td>13</td>
<td>0.30W</td>
<td>4.7</td>
</tr>
<tr>
<td>RegX</td>
<td>String Match</td>
<td>String Match (ADFA)</td>
<td>5.0</td>
<td>4.4</td>
<td>1.95W</td>
<td>9.8</td>
</tr>
<tr>
<td></td>
<td>Regex Match</td>
<td>Regex Match (NFA)</td>
<td>5.0</td>
<td>1.5</td>
<td>1.95W</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Table 2: Comparing Performance and Power Efficiency of Transformation/Encoding Algorithms.

[58] exploits SIMD acceleration on finding the CSV delimiters. Vectorization reduces the branch miss rate by 50% and achieves the single-thread performance over 250MB/s on parsing on a conventional x86 CPU. Compared to InstantLoad, our approach provides 40x better single thread performance and can easily saturate a standard DDR3 memory channel. MISON [56] exploits speculative parsing and 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 the 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 representative training data for building the pattern trees and the underlying assumption that JSON records follow an overall structure or template that does not vary significantly across records. Our approach accelerates JSON parsing 10x better than MISON without the assumption that records follow an overall structure, and can be further applied to more complicated nested structures (e.g. XML). The achieved speedup is also independent of the JSON record structures.

On the other hand, lazy ETL approaches accelerate data loading through avoiding ETL. Databases [28, 47, 48] 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 accelerate the ETL computations for later queries. However, this approach has several limitations: 1) Main memory overhead. A big cache is needed to enjoy the high cache hit rate. Positional index consumes significant spaces because each column requires multiple offsets. 2) Performance sensitivity to query pattern. Concurrent or contiguous queries that share few columns reduces the caching effectiveness. Besides, the reuse distance also matters the caching behavior. 3) Total column footprint restriction. Lazy loading is built on the assumption that only few out of the entire columns are needed. Our approach uses the hardware acceleration instead of avoiding the work, which doesn’t suffer...
from these drawbacks. The Proteus system [47] exploits engine customization. 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. Our approach produces a modular software architecture that leverages existing analytical components and has easy integration. The proposed architecture preserves software investments such as query optimization, existing optimized operators, storage hierarchy management, and distributed/parallel management. Later systems [29, 48] further exploit query semantics for early predicate pushdown. Partial loading and early filtering help to avoid unnecessary parsing and deserialization cost in later stages. However, predicates require execution on the loaded data format for efficient CPU processing. Our approach doesn’t have this constraint and can push predicate directly into raw data with high efficiency.

3.3 Data Representations for Fast Query Execution

Database storage representation strongly affects the query performance. C-store [26, 64] 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 data storage representations [39, 55, 57] are proposed to exploit SIMD accelerations for fast query scan. Our approach is different. We consider dynamically transforming data representations during query execution for performance. It is different from the static storage layout optimizations but can leverage these beneficial formats to optimize operator execution. Moreover, researchers show that compressed data representations [35, 53] can be used to accelerate multiple popular iterative machine learning algorithms. The training data is compressed during execution before the ML algorithms start. The compressed formats are carefully designed to allow direct computation on them without decompressing. Our approach explores more opportunities by allowing multiple formats co-existed and switch across each other during runtime. Zukowski et al. [73] first explored dynamic transposition between row and column format during query execution. The results suggest column format benefits scanning and row format benefits aggregation and hashing. However, they only studied simple data layouts. Our approach can transform diverse data formats cheaply thus is capable of using more powerful data representations during runtime. Furthermore, multiple database core operators [46, 59, 62, 63, 71] (e.g. sorting, partition, shuffling, aggregation) can get accelerations from using SIMD, GPU and FPGA. However, many of them assume a customized data layout to leverage these accelerators. Our approach provides such a foundation for the assumption by enabling data transformations almost for free.

4 Problem and Research Approach

In this section, we formally describe the problem and motivate the approach. The approach leads to several successive research questions; detailed approaches for each of the questions are provided respectively. In the end, we discuss measure of success and likely implications of our study.

4.1 Problem

Data analytical systems deliver poor performance when analyzing messy, heterogeneous, and ad-hoc raw data. The traditional CPU architectures pose fundamental limitations on performance and efficiency. Improving hardware data processing performance with accelerators is hard because it requires general-purpose architecture that can support a range of computations with high energy efficiency and low silicon cost. It is impractical to deploy one accelerator for each possible application because the chip design and the silicon cost are expensive, the architecture becomes obsolete when new algorithms emerge, and the sophisticated
hardware/software interface is a disaster for software stack maintenance and compiler support. However, most of the customized data processing accelerators fall into this category and are constrained by their narrow applicability and generality [45, 46, 49, 54, 62, 69, 70]. The challenges for accelerators are also significant for software. Accommodating raw data analysis acceleration into existing systems should preserve the original software architecture, such as the front end, the execution engine, and the query optimizer; the modularity changes should not affect the entire system architecture. Low-level acceleration detail should be hidden from the system, but a proper abstraction is required to enable the explicit control of query optimization. Accelerated architectures should preserve full query composition and optimization capabilities. However, current approaches either only accelerate isolated data reader plugins [56, 58] or require significant modifications to the entire query processor to benefit from holistic acceleration optimizations [28, 47, 48].

### 4.2 High-Level Approach

To address the problem, we use a two-part approach: namely, the unified transformation accelerator (hardware) and accelerated transformation operators (software). The unified transformation accelerator (UTA) creates new and flexible architecture support for analytical workloads. We carefully apply the hardware architecture customization principle [33] 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 then carefully customize the data transformation ISA for generality and high efficiency. Figure 6 contrasts a traditional SoC approach and our approach. The UTA approach (Figure 6b) has simpler chip architecture and is extensible by software applications. In Figure 6a, many hardwired fixed accelerators are integrated with CPU to support broad applications. 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. On the other hand, a unified transformation accelerator (Figure 6b) erases the burden of complex accelerator library management, running its own programs to support various applications. Users can express algorithms using a high-level languages and write a user-space acceleration program for new applications.

The accelerated transformation operators (ATO) is a software framework for integrating hardware
accelerations into data analytical systems in a pervasive and flexible fashion. This framework integrates the acceleration pervasively in query execution by introducing hardware accelerated transformation operators with a cost model. Traditional analytical operators can be accelerated, new operators can be created, and functions can be fused and offloaded to the accelerators. The framework enables hardware accelerators in the system and exposes them to the query optimizer. Figure 7 shows the ATO approach and compares it to other alternatives. The software-accelerated batch ETL approach applies advanced commodity hardware features (e.g. SIMD) in the batch ETL loading routines. The lazy loading approach 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 implicitly reuse the cached data to minimize loading overhead. Different from the lazy loading approach, the ATO approach preserves the existing query engine architecture by only adding multiple new operators. Cheap data transformation makes it possible for beneficial formats to be used for the particular operators to speed up the query processing. Intermediate data representations on the edges of a query plan are extended to enable format-specific operators. Finally, we capture the raw data types and structures in the extended representations to avoid unnecessary computation.

**Figure 7: Approaches for Accelerating Raw Data Analysis.**

4.3 Benefits of Our Approach

The two-part approach has several advantages. The UTA approach delivers significant speedup and energy savings than a conventional CPU chip on a broad class of large-scale data transformation workloads by its high energy efficiency, closing the performance gap for raw data analysis. Generality and programmability make it possible to accelerate various tasks in analytical workloads such as compression, parsing and searching. With flexible composition of the customized ISA, future emerging data transformation algorithms and data
formats can be easily supported by new acceleration programs. Constructing acceleration programs in user space offers dramatic generality and flexibility compared with fix-functioned API calls to hardwired ASIC accelerators. Since one accelerator can do the data transformation jobs of many fixed-function accelerators, only one accelerator runtime is needed. The software library management and application portability can be greatly simplified.

The ATO approach enjoys these advantages and provides more. The operator-based clean abstraction preserves the system architecture and hides the low-level hardware techniques. The abstraction enables holistic query optimization without needing to re-architect the entire query engine. Users are abstracted away from the underlying hardware acceleration and can focus on the actual query semantics. With direct hardware acceleration on data transformations (e.g., ETL), queries on raw data are as fast as if they are executed on loaded data. The performance is stable and independent of the query pattern (e.g., selectivity, execution order, etc.). Furthermore, cheap data transformation enables other accelerated operators, whose incompatible data representation requirements prohibit their practical integration. Without the format restriction, the query optimizer can manage and schedule the accelerated operators in arbitrary data representations for better performance. Finally, capturing raw data structures enables lazy transformation in different stages of a query plan to aggressively avoid computation.

4.4 Research Questions

The two-part approach is powerful but requires exploring several unknown research questions. We divide all the derived research questions into two parts below, and discuss the detailed strategies for them in Section 4.5, respectively.

4.4.1 Part 1: Unified Transformation Accelerator

For the unified transformation accelerator (UTA) approach, we need a deeper understanding of the workloads before we can design it. What are the functional requirements for a unified transformation accelerator? In this question, we not only need to analyze the workloads to identify the potential tasks, but also to study the common properties across them to understand whether there exists some general acceleration mechanisms. Second, what are the requirements for UTA system-level integration? In this question, we will explore how the accelerator can be applied relative to the rest of the computation. The integration should give the desired flexibility and application performance. After understanding the first two questions, we can design the hardware. What is the design of a unified transformation accelerator? The answer should include the design of major hardware components and micro-architecture features. Finally, in terms of performance and power, what are the capabilities of the proposed accelerator?

4.4.2 Part 2: Accelerated Transformation Operators

The accelerated transformation operator (ATO) approach involves a set of research questions as well. First, there is a gap between hardware acceleration and data analytical system. What is the software architecture of an ATO-enabled analytical system? The software architecture and integration complexity determines how practical it will be to apply the ATO approach in an existing system. The answer should satisfy the requirements set in Section 4.1. Second, what are the capabilities of an ATO-enabled analytical system? This question involves the potential query execution benefits ATO brings. We will perform several case studies followed by a performance study in the answer.
4.5 Detailed Strategies

Next, we outline the research approaches to answer these questions. We first crack the research questions for the unified transformation accelerator approach (part 1), and then discuss the directions to explore the accelerated transformation operators approach (part 2).

4.5.1 Cracking The Part 1 Questions

• **What are the functional requirements for a unified transformation accelerator?**
  We will first collect some real-world data analytical workloads and typical use cases as the benchmarks. To study the performance, a coarse-grained runtime profiling is required to understand where the time goes. Careful analysis is required on the computations which load the external data. After identifying the major time-consuming parts, we will divide them into smaller representative computation kernels. We expect the kernels to cover rich data transformation tasks as shown in Figure 8, such as compression, parsing, encoding, deserialization, and pattern matching. These isolated kernels then serve as the micro-benchmarks to allow the fine-grained architecture study.

![Figure 8: Acceleration Functional Requirements.](image)

• **What are the requirements for UTA system-level integration?**
  On the chip level, the potential locations include a memory controller, a SoC-style accelerator, or a functional unit in a CPU core. We target at batch offloading tasks and expect that a SoC accelerator integration (Figure 9a) fits our need. For the memory system integration, the memory of UTA is mapped to a portion of the physical memory address space of the CPU system (Figure 9b). This address space should be uncacheable to ensure there is no inconsistency. Software control of the data movement engine (DMA) is required. For the software integration, we plan to use the compiler intrinsics or direct API calls to launch the accelerator.

• **What is the design of a unified transformation accelerator?**
  We will design a concrete instance, called the Unstructured Data Processor (UDP), to demonstrate the feasibility of a unified transformation accelerator. Figure 10 depicts the high-level design choice. We
will use the customized architecture approach for efficiency, MIMD parallelism for throughput, and software-controlled local memory to minimize the power consumption. To design the core processing element (PE) for each thread of the MIMD parallelism, we should first understand why conventional CPU architectures have poor performance on querying raw data. With the chosen kernels, we can measure architectural metrics such as cache hit rate, branch miss prediction, cycles, and instruction count, using performance analyzing tools such as the Intel Performance Counter Monitor [14] and Linux Perf [19]. The derived architectural insights can facilitate us to narrow down the scope of the CPU inefficiency and bottlenecks. We will then determine the core architectural features for the Unstructured Data Processor.

**What are the capabilities of the proposed accelerator?**
The capabilities include the performance and the implementation cost. We will develop a functional cycle-accurate simulator and a real hardware RTL implementation as the primary modeling tool. To study the performance, we will run the proposed kernels on the simulator, collecting architectural metrics such as cycle count, instruction count, and memory reference. This helps us to understand whether the architecture can efficiently support the workloads and improve the accelerator design. Then we will use the clock period derived from the RTL implementation to calibrate the cycle-accurate
simulator for the execution time modeling. Finally, the EDA tools (Synopsys and CACTI [4]) generate the power cost and the area cost for the UDP enabling a systematic comparison with a CPU processor on performance, power, and area.

4.5.2 Cracking The Part 2 Questions

- **What is the software architecture of an ATO-enabled analytical system?**
  
  Figure 11 shows the software architecture of the accelerated relational query processor in an ATO-enabled analytical system. The extension requires no more than a set of operators and several optimization rules. There are two types of the newly introduced operators. One gets the speedup directly from the hardware acceleration on the required data transformations inside. The other one does the data transformations for other accelerated-operators (e.g., implemented by SIMD, GPUs, FPGAs, ASICs) that require specific data formats (e.g., columnar for SIMD). Essentially, this framework preserves the traditional analytical system architecture but allows explicit query optimization and flexible operator composition across hardware accelerations. We plan to build an accelerated analytical system, named ACCORDA (Accelerated Operators for Raw Data Analysis), by extending the state-of-art distributed analytical system (SparkSQL [29]), using the ATO approach. The new operators are implemented using the data transformation library we built for deep hardware acceleration service integration. The library offers scalable services for transforming rich data representations with high performance and flexible compositions across these transformations. Internally, the library uses the volcano-style iterator execution model to increase cache locality and reduce memory footprint. Its core functions are implemented in C++ and connect to Java bindings via JNI. Block processing is used to minimize the hardware accelerator offloading overhead. To test the feasibility of our ATO approach, we will first emulate the new optimizer-aware physical operators with UDFs and manually place them in the query plan as if there is query optimization. The extended intermediate data representations on the query plan edges are wrapped inside the binary data type to be compatible with the original system. The accelerated transformation operators are responsible for transforming the representations.

- **What are the capabilities of an ATO-enabled analytical system?**
  
  The fast and cheap data transformations delivered by the UDP enable the accelerated transformation operators. The accelerated data source operators (CSV, JSON, etc.) are one form. They require
significant data transformations (parsing and deserialization) that can be accelerated directly. Other possible ATOs include, for example, ad-hoc filter (e.g., regex match and value predicates) operators executing on native raw formats, new application-specific (de)compression operators to save memory or communication costs, and novel accelerated transpose operators to shuffle row/column (group) formats, which support the SIMD/GPU/FPGA/ASIC accelerated operators such as scan, hash/range partitioning, and sort operators on unstructured raw data (usually row-based). Figure 12 demonstrates a running example for a traditional scan-aggregate query on JSON data. The new plan uses the accelerated JSON operator for fast parsing and data type conversion. Next, the operator transforms the row-based format into a customized columnar format for fast SIMD-accelerated scan operation [57]. Finally, the results are transformed back to the row-based format for a traditional sum operation. Furthermore, the raw-aware data representations enable raw structures existed within a query plan. Raw data is lazily converted to SQL data types on-the-fly alongside the query execution to minimize the expensive loading cost.

```
select sum(student.score) where university.name = “UCicago”
```

Figure 12: Accelerated Transformation Operator (ATO) Enables New Query Optimization.

We plan to investigate the performance and the scalability of the ACCORDA system. For the performance, to demonstrate the ATO potential benefit, we will perform multiple case studies with raw data, especially ones that benefit from enriching data representations during query execution. The queries can exploit lazy transformations and SIMD/GPU/FPGA accelerations for scan, sort, and aggregation, with the help of ATO. We will compare the baseline SparkSQL system running on a CPU server with the ACCORDA system on an UDP-integrated hardware platform. The UDP hardware is modeled in two ways: a cycle-accurate software simulator and a real hardware FPGA implementation on HARP [24]. We will profile each part of the computation in the end-to-end query workloads and feed the measurements into a performance model to project the benefit of deploying a real ASIC UDP implementation. To study the scalability, we plan to develop a performance model which replays the collected simulation traces from small-scale experiments and extrapolate the large-scale (thousands of nodes) performance.
4.6 Success Measure Implications

We present the quantitative measure of success for the two-part approach. For the unified transformation accelerator (UTA) approach, the proposed UDP design should achieve more than 10GB/s throughput across the diverse data transformation workloads, to saturate a standard DDR3 memory channel bandwidth. The energy efficiency should be within 30% of the most efficient ASIC design. The implementation should have an area under 5 mm² and power consumption less than 1 Watt to limit the deployment cost within 1% relative to the host CPU system. For the accelerated transformation operator (ATO) approach, the critical judgment is the performance. We use the single thread throughput (GB/s) as the metric. The ACCORDA system with external unloaded data should be within 90% of the ideal performance, which is executed on internal loaded data. The performance benefit should come from the dramatic reduction (> 10x) of the ETL computation overhead alone. Moreover, combined with the accelerated operators and rich intermediate data representations, ACCORDA should exceed the ideal query performance. The net benefit should come from novel data representations, but without the overhead of data transformations. The benefit should be robust across queries, raw data formats, and data types. Achieving these requirements will mean that we can do faster query execution with raw data than traditional high-performance analytical systems with loaded data.

The success of the UTA and the ATO shows the feasibility of the new capabilities described in Section 2. Since one UDP accelerator saturates a DDR3 channel, it is possible to deliver >100x unstructured data processing speed than CPUs. In order to search for a person, we are able to query 100x larger machine logs, sensor data, and related documents in the data lake with the ACCORDA system. The acceleration enables processing 10TB data per second using 1000 servers in a second, meeting the requirements for interactive analysis. The ACCORDA performance also satisfies the real-time product recommendations. A single machine can quickly pull and process a few gigabytes of the related social posts in less than a second, including filtering for particular post types, parsing sentence structure with special characters (e.g. #, @, facial expression), cross-join social posts to enrich features (e.g. at someone’s post), transform dirty and error-prone dialogues into clean structures for latter word embedding, topic modeling, etc. The high performance data processing capability from UDP and ACCORDA makes it possible to handle real-time computations on massive newly generated data in the data lake, which are waiting to be indexed, add meta-data or be loaded into the database.

5 Research Plan

In this section, I will discuss the current status of the project and propose a plan to complete the thesis project. In the end, I sketch an outline of the final thesis.

5.1 Project Status

This project requires a set of software and hardware infrastructures to be built. For the hardware, the Unstructured Data Processor (UDP) design and implementation should be evaluated through a cycle-accurate architecture simulator and a rigorous ASIC design. Flexible UDP program construction and compilation require on a full-blown code generation toolchain, including a front-end compiler, an assembler, a linker and a loader. To run large-scale experiments, we need a CPU-FPGA prototype to process tens of gigabytes of data fast. For the software, we need to implement the ACCORDA (Accelerated Operators for Raw Data Analysis) runtime management system and integrate it into the SparkSQL with a set of new physical operators. A modified query optimizer or several hard-coded query plans might also be needed to test the effectiveness of the ATO approach.
Up to the time of writing this proposal, we have designed the UDP architecture, including the micro-architecture implementation. The experimental platforms include a CPU-FPGA system and a CPU-simulator based system. We have constructed the complete UDP software compiler back-end tools including an assembler, an optimizer, a linker and a loader. Users specify the UDP program in the intermediate representation (IR) or other high-level languages with the front-end compiler support, and our back-end tools generate the binary code. We have shown the UDP acceleration on a diverse set of data transformation workloads ranging from parsing to pattern matching, achieving over 20x speedup and 1900x energy efficiency compared to an 8-thread CPU. We will further investigate the benefit of using the UDP acceleration in the analytical systems to speed up queries on external raw data. The description of the UTA architecture, implementation and evaluations are presented in Section 6. In terms of the proposed research questions in Section 4.4, we have addressed all the questions in part 1 (see Section 6.1). The final thesis will answer all the remaining research questions (part 2) in Section 4.4. Particularly, I’m going to investigate several representative raw data analysis cases to show the great combinational performance benefit of applying the accelerated transformation operators (ATO) with SIMD and GPU acceleration together into existing analytical systems. At the meantime, I will build the software infrastructure (ACCORDA) and integrate the hardware platform (UDPSIM, UDPFPGA) for large-scale experiments. Finally, I propose the following timeline to complete the thesis.

5.2 Proposed Timeline

- SparkSQL UDP-accelerated data source package for CSV format running on the simulator-based hardware platform. [1/31/2017]
  - CSV format Spark-side bindings for the UDP acceleration via JNI. [12/31/2017]
  - Implement UDP kernel code for CSV parsing, deserialization for various data types and predicate pushdown on raw data. [1/15/2017]
  - Integrate and debug the UDP-accelerated CSV data source operator. [1/31/2017]

- Evaluate the simulator-based UDP-accelerated system performance. [3/14/2018]
  - Port the UDP-accelerated packages to support full end-to-end query workloads. [2/14/2018]
  - Measure single node and multi-node ETL performance. [2/21/2018]
  - Revise the ETL-accelerated system implementation. [3/1/2018]
  - Measure effectiveness of the UDP-enabled predicate pushdown. [3/7/2018]
  - Revise the UDP-enabled predicate implementation. [3/14/2018]

- Build more accelerated transformation operators. [4/14/2018]
  - Implement accelerated transformation for multiple data formats during query execution. [4/1/2018]
  - Implement operators accelerated by SIMD and GPU using specific formats. [4/7/2018]
  - Measure experiment performance benefit. [4/14/2018]

- UDP-accelerated SparkSQL running on the FPGA-based platform. [5/14/2018]
  - Port the UDP acceleration on the FPGA-based platform. [5/7/2018]
  - Re-measure the above experiments for validation on the FPGA-based platform. [5/14/2018]
– Analyze the validation results and revise the system implementation on both platforms. [5/14/2018]

• Build performance model and collect traces to project performance for large-scale experiments. [5/21/2018]

• Prepare the ASPLOS 2019 paper submission. [8/1/2018]
  – Write a detail paper outline for the ACCORDA data processing system. [6/11/2018]
  – Write a complete paper. [8/1/2018]

• Documenting the thesis research contribution. [5/1/2019]
  – Write the full PhD thesis document. [4/1/2019]

• PhD thesis defense. [6/1/2019]

5.3 Expected Research Contributions

In this thesis, we expect the research contributions will include:

• Description of the Unified Transformation Accelerator (UTA) approach to enable cheap and flexible data transformations for fast analysis on raw data.

• Hardware architecture of an instance of the UTA approach, the unstructured data processor (UDP), including the description of key features of multi-way dispatch, variable-size symbols, flexible-source dispatch, and an addressing architecture for efficient, flexible UDP lane-bank coupling. Quantitative comparison for each and documenting their effectiveness.

• Performance evaluation for UDP on diverse workloads showing speedups from 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.

• Description of the Accelerated Transformation Operator (ATO) approach that enables seamless UTA integration, explicit query optimization, and flexible operator composition into existing analytical systems.

• Software architecture of an instance of the ATO approach, the ACCORDA (Accelerated Operators for Raw Data Analysis) system, including detail methods for exploiting cheap and flexible data transformations to construct and accelerate SQL operators.

• Performance evaluation of the ACCORDA system on a diverse set of end-to-end large-scale analytical workloads (＞100GB), achieving speedup from 2x to 10x on querying external raw data.
5.4 Full Thesis Outline

Finally, I sketch the full outline of my PhD thesis as the following:

1. Introduction
   1.1. Data Analytical Challenges
   1.2. Rising of the Datalake
   1.3. Ubiquitous Analysis on Raw Data
   1.4. Goals and Scope of the Dissertation
   1.5. Organization of this Thesis

2. Background
   2.1. Data Transformation Workloads
   2.2. The Sea of Accelerators
   2.3. End-to-end Query Execution on Raw Data
   2.4. Challenges and Discussion

3. Unified Transformation Accelerator
   3.1. Limitations of the CPU Architectures
   3.2. The Architecture of the Unstructured Data Processor
   3.3. UDP Lane: Fast Symbol and Branch Processing
      i. Multi-way Dispatch
      ii. Variable-size Symbol
      iii. Flexible Dispatch Sources
      iv. Flexible Addressing for Data-Parallelism and Memory Utilization.

4. Accelerated Transformation Operator
   4.1. Bottlenecks for Queries on External Native Data
   4.2. ACCORDA Software Architecture
   4.3. Accelerated Query Plan Construction

5. Methodology
   5.1. Evaluate the Unstructured Data Processor (UDP)
      i. Data Transformation Micro-benchmarks
      ii. Metrics
      iii. Configuration and Comparison
   5.2. Evaluate the ACCORDA system
      i. End-to-end Analytical Workloads
      ii. Configuration and Comparison
6. Evaluation

6.1. The UDP Hardware Architecture
   i. Performance Evaluation
   ii. Hardware Implementation

6.2. The ACCORDA System
   i. Accelerated-ETL Data Transformation
   ii. Combining Accelerated Operators
   iii. Large-scale Accelerated Query Execution

7. Discussion and Related Work

7.1. ETL Optimized System
7.2. Software Accelerated Query Processing
7.3. Hardware Accelerated Query Processing
7.4. Emerging Architectures

8. Conclusion

8.1. Conclusion
8.2. Future Work

6 Initial Results

In this section, we study the Unified Transformation Accelerator approach (UTA) and the Accelerated Transformation Operator approach (ATO) with a few promising results. Moreover, we describe the lessons we learned from designing a general-purpose data transformation accelerator (UDP) in Section 6.6, and discuss the implications (Section 6.7) of such design to the thesis project, as well as to the computer architecture research community. Finally, we briefly describe the on-going progress on building the ACCORDA system in Section 6.5 with a few preliminary results in Section 6.6.5.

6.1 Addressing Research Questions

Specifically, the research questions from part 1 in Section 4.4.1 are fully studied and answered in the following sections. Section 6.2 discusses the functional requirements of the UDP acceleration (first research question). It narrows down the applications to a few representative micro-benchmarks for later architectural evaluations. The UDP integration and design (second and third research question) are analyzed and explained in detail in Section 6.3 and Section 6.4. The quantitative evaluation of the UDP’s capabilities (fourth research question) is presented in Section 6.6. In order to answer the part 2 questions, we are working in the progress of building the ACCORDA system prototype. The system design and implementation are discussed in Section 6.5.
6.2 Data Transformation Workloads

We collected a broad set of commonly used big data analytical workloads such as extract-transform-load, searching unstructured data, gathering data statistics, in-memory machine learning, and query execution. As much as 80% of the run time can be spent on the data transformation process (detail see Section 6.3). With a closer look at these workloads, we identified a set of computation kernels. 1) Compression and decompression, which are heavily used in distributed computing to reduce IO cost. 2) Parsing, which extracts interested values against complex, messy input data. 3) Pattern matching, which searches text files using regular expressions or strings. 4) Coding and (de)serialization, which are used to reduce memory capacity overhead and optimize the performance. We then designed a small but representative set of micro-benchmarks for the UDP evaluation and collected several practical end-to-end analytical workloads for the ACCORDA evaluation. The micro-benchmark covers the question about the functional requirement for the unified transformation accelerator in Section 4.4.

6.3 Data Transformation CPU Bottleneck

We first studied the performance of complex batch ETL. Figure 13a shows single-threaded costs to load all TPC-H [23] Gzip-compressed CSV files (scale factor from 1 to 30) from SSD into the PostgreSQL relational database [22] (Intel Core-i7 CPU with 250GB SATA 3.0 SSD). This common set of extract-transform-load (ETL) consumes nearly 800 seconds for scale factor 30 (about 30GB uncompressed), dominating time to initial analysis [28]. Figure 13b shows that >99.5% wall-clock loading time is spent on CPU tasks, rather than disk IO. CPUs is nearly 250x slower than the IO speed. For real-time query execution with raw data, the transformation is the primary bottleneck as well. The ETL data transformation contains parsing and deserialization. Figure 14 shows the execution time breakdown of TPC-H Q1 to Q8 using SparkSQL with raw data. Parsing and deserialization dominate the query execution time, ranging from 50% to 90% of the total time.

Through rigorous measurements of the state-of-art data transformation micro-benchmarks, we learned that the major issue with the conventional general-purpose CPU architecture is the poor branch prediction performance. These workloads are all FSM-based computations. The CPU implementation treats FSM computation as branches associated with the code blocks. However, these workloads are full of branches that are inherently hard to predict. We discovered that the branches are conditioned directly on the symbols
from the input stream which usually has high entropy. For applications like pattern matching, branch mis-prediction wastes as much as 90% of the CPU cycles. The CPU branch predictor behaves poorly and leads to frequent pipeline flush. Moreover, the short code blocks reduce the effectiveness of the deep pipeline and the super-scalar features in the CPU. Finally, sub-byte operations are not efficiently supported in CPUs. These insights helped us design and build a unified data transformation accelerator.

6.4 The Unstructured Data Processor

We answer how to design and integrate a unified transformation accelerator, by creating a concrete instance, called the Unstructured Data Processor (UDP) [38]. The CPU host system integrates the UDP as a standalone SoC accelerator and maps the UDP’s memory address to a range of uncacheable physical address. An on-chip DMA engine supplies data for the UDP under software control. We achieved the generality and the programmability for data transformations by designing the UDP ISA. For the performance, UDP’s MIMD parallel architecture (multiple UDP lanes) harvests the plenty data block parallelism in the transformation workloads. Software-managed scratchpad local memory is used to minimize the power consumption and access latency. For the fast sequential processing speed of a single UDP lane, we further identified four key micro-architecture features. 1) Multi-way dispatch. Multi-way dispatch doesn’t use branch prediction thus incurs no branch mis-prediction overhead. It implements a multi-target branch using only one dispatch operation by directly computing the target address, which requires coupling the code layout placement during the compiling. 2) Variable-size Symbol. This feature generalizes the symbol size that the architecture can efficiently support for various sub-byte encodings. 3) Flexible dispatch source. We provide architecture support for various input sources to enable efficient streaming processing, block processing and the flexible mix of the traditional control-flow programs. 4) Flexible addressing. The UDP architecture can flexibly aggregate memory resources and adjust the MIMD parallelism to allow the processing elements (UDP lane) addressing more memory beyond dedicated one. With these features, the UDP supports FSM operations efficiently.

6.5 The ACCORDA System

We designed the ACCORDA (Accelerated Operators for Raw Data Analysis) runtime management system (Figure 15) and packaged it as an external library for applications. From the application perspective, hardware accelerations are abstracted as separate heterogeneous ACCORDA clients, each of which is in charge of some
acceleration tasks. Internally, the clients submit the tasks to the runtime manager for execution. The runtime system is responsible for efficient time-multiplexing the underlying hardware accelerators, including task scheduling, program swapping, and context switching. It provides a logical abstraction to the applications as if there are multiple dedicated hardware accelerators. We leveraged this library and integrated it into the SparkSQL’s iterator execution model, as shown in Figure 16. The accelerated transformation operators encapsulate the data transformation accelerators by using direct internal calls to the runtime clients.

6.6 Evaluation

In this section, we describe the detail of the experiments that have been conducted and present the related results.
6.6.1 Data Transformation Micro-benchmarks

To study the performance and generality of the proposed UDP accelerator, we selected a diverse set of kernels drawn from broader ETL and data transformations.

<table>
<thead>
<tr>
<th>Application</th>
<th>Workload</th>
<th>CPU Challenge</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSV Parsing</td>
<td>Crimes, NYC Taxi Trip [20], Food Inspection [8]</td>
<td>3x branch mispredicts</td>
</tr>
<tr>
<td>Huffman Encoding</td>
<td>Canterbury Corpus, Berkeley Big Data</td>
<td>5x branch mispredicts</td>
</tr>
<tr>
<td>Huffman Decoding</td>
<td>Canterbury Corpus, Berkeley Big Data</td>
<td>5x branch mispredicts</td>
</tr>
<tr>
<td>Pattern Matching (Intrusion Detection)</td>
<td>IBM PowerEN dataset [67]</td>
<td>Poor locality, 1.6x L1 miss rate</td>
</tr>
<tr>
<td>Dictionary and Run Length Encoding (RLE)</td>
<td>Crimes</td>
<td>Costly Hash 54% runtime</td>
</tr>
<tr>
<td>Histogram</td>
<td>Crimes, NYC Taxi Trip</td>
<td>5x branch mispredicts</td>
</tr>
<tr>
<td>Compression (Snappy)</td>
<td>Canterbury Corpus [6], Berkeley Big Data [2]</td>
<td>15x branch mispredicts</td>
</tr>
<tr>
<td>Decompression (Snappy)</td>
<td>Canterbury Corpus, Berkeley Big Data</td>
<td>15x branch mispredicts</td>
</tr>
<tr>
<td>Signal Triggering</td>
<td>Keysight Scope Trace [36]</td>
<td>mem indirect, address, condl, 9 cycles</td>
</tr>
</tbody>
</table>

Table 3: Data Transformation Workloads

**CSV parsing** involves finding delimiters, fields, and row and column structure, and copying field into the system. The CPU code is from libcsv [17]; these measurements use Crimes (128MB) [7], Trip (128MB) [20] 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** [18]. Measurements use Canterbury Corpus [6] 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 [67], 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 [50] and NFA [43] models. **Compression** CPU code is the Snappy [11] 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 [10]. Measurements use Crimes.Latitude, Crimes.Longitude, and Taxi.Fare with 10, 10, and 4 bins of IEEE FP values [12]. 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 [36], at 4 symbols per lookup. Trace is proprietary from Keysight oscilloscope. UDP implements exactly the same automaton.

Table 3 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 [21] 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.

6.6.2 UDP Accelerator Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rate (Megabytes/s)</td>
<td>Input processing speed for a single stream or UDP lane</td>
</tr>
<tr>
<td>Throughput (Megabytes/s)</td>
<td>Aggregate performance</td>
</tr>
<tr>
<td>Area (mm²)</td>
<td>Silicon area in 28nm TSMC CMOS</td>
</tr>
<tr>
<td>Clock Rate (GHz)</td>
<td>Clock Speed of UDP implementation</td>
</tr>
<tr>
<td>Power (milliWatts)</td>
<td>On-chip UDP power</td>
</tr>
<tr>
<td>TPut/power (MB/s/watt)</td>
<td>Power efficiency</td>
</tr>
</tbody>
</table>

To understand the capabilities of the UDP, we model the performance and the energy by using a cycle-accurate UDP simulator written in C++. We compare the achievable rate for one UDP lane to one Xeon E5620 CPU thread [15]. For the throughput per watt, we compare a UDP (64 lanes + infrastructure) to a E5620 CPU (TDP 80W, 4-cores, 8-threads). We compare the CPU baseline to a UDP program running on a single or up to 64 lanes (a full UDP), reporting rates and throughput per watt below.

6.6.3 UDP Accelerator 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 17, 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.

Snappy Compression. As shown in Figure 18, 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 CPU³. 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.
Huffman Decoding. Figures 19 show single-lane UDP Huffman decoding at 366 MB/s, 24x speedup versus a single CPU thread. A full 64-lane UDP achieves geometric mean of 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 20 shows that a single UDP lane surpasses a single CPU thread by 7-fold on average, achieving 300-350MB/s across the workloads. The single lane UDP achieves 333-363 MB/s throughput on string matching dataset (simple) and 325-355 MB/s on complex regular expressions (complex). A UDP outperforms CPU by 1,780-fold on average throughput per watt. The collection of patterns are partitioned across UDP lanes, maintaining data parallelism. The UDP code exploits multi-way dispatch for complex pattern detection.

Dictionary-RLE Encoding. UDP delivers a 6-fold rate benefit for both Dictionary and Dictionary-RLE (see Figure 21). Due to space limits, only Dictionary-RLE performance data is shown. For the full UDP,

---

5The CPU outperforms on rank by guessing data is not compressible and skipping input. UDP processes the entire input.
the power efficiency is more than 4,190x on Dictionary-RLE and 4,440x on Dictionary Encoding versus CPU. The UDP code exploits multi-way dispatch to detect complex patterns and select run length. Flexible dispatch sources are used.

**Histogram.** Figure 22 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.
Overall Performance. The key UDP architecture features: multi-way dispatch, variable symbol size, flexible dispatch source, and flexible memory sharing accelerate the workload kernels. Comparing a full UDP (64-lane) with 8 CPU threads shows 3 to 197-fold speedup across workloads with geometric mean speedup of 20-fold (see Figure 23a). Second, compare throughput/power for UDP and CPU in Figure 23b, using UDP implementation power of 864 milliWatts (Appendix 6.6.4) 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 that lie at the heart of ETL, query execution, stream data processing, and intrusion detection/monitoring.

6.6.4 UDP Hardware Implementation

We describe implementation of the UDP micro-architecture, and summarize speed, power, and area. Each UDP lane contains three key units: 1) Dispatch, 2) Symbol Prefetch, and 3) Action (see Figure 24). The Dispatch unit handles multi-way dispatch (transitions), computing the target dispatch memory address for varied transition types and sources. The Stream Prefetch unit prefetches stream data, and supports variable-size symbols. The Action unit executes the UDP actions, writing results to the UDP data registers or the local memory.

The UDP is implemented in SystemVerilog RTL and synthesized for 28-nm TSMC process with the Synopsys Design Compiler, producing timing, area, and power reports. For system modeling, we estimate local memory and vector register power and timing using CACTI 6.5 [4]. The overall UDP system includes the UDP, a 64x2048-bit vector register file, data-layout transformation engine (DLT) [66], and a 1MB, 64-bank local memory. Silicon power and area for the UDP design is shown in Table 4.

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 Westmere EP core+L1 in a 28nm process [15]. 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.

### 6.6.5 Preliminary ACCORDA Performance on TPCH

To systematically evaluate the UDP and ACCORDA performance, we built an experimental prototype stack shown in Figure 25. We extended the SparkSQL framework with our data transformation software and hardware described in Section 6.4 and 6.5. The colored blocks are the hardware/software components we have implemented.

![Figure 25: Overall Experiment Software Stack.](image)

By the time of writing this proposal, we are able to run a TPCH query (Q4) using the stack in Figure
<table>
<thead>
<tr>
<th>Component</th>
<th>Power (mW)</th>
<th>Fraction</th>
<th>Area (mm²)</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP Lane</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dispatch Unit</td>
<td>0.71</td>
<td>37.9%</td>
<td>0.022</td>
<td>40.6%</td>
</tr>
<tr>
<td>SBPUnit</td>
<td>0.24</td>
<td>12.8%</td>
<td>0.008</td>
<td>14.3%</td>
</tr>
<tr>
<td>Stream Buffer</td>
<td>0.22</td>
<td>11.9%</td>
<td>0.002</td>
<td>3.7%</td>
</tr>
<tr>
<td>Action Unit</td>
<td>0.68</td>
<td>36.1%</td>
<td>0.021</td>
<td>39.2%</td>
</tr>
<tr>
<td>UDPLane</td>
<td>1.88</td>
<td>100.00%</td>
<td>0.054</td>
<td>100.00%</td>
</tr>
<tr>
<td>UDP (64 lanes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 Lanes</td>
<td>120.56</td>
<td>14.0%</td>
<td>3.430</td>
<td>39.5%</td>
</tr>
<tr>
<td>Vector Registers</td>
<td>8.47</td>
<td>1.0%</td>
<td>0.256</td>
<td>3.0%</td>
</tr>
<tr>
<td>DLT Engine</td>
<td>19.29</td>
<td>2.2%</td>
<td>0.138</td>
<td>1.6%</td>
</tr>
<tr>
<td>Shared Area</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1MB Local Memory</td>
<td>715.36</td>
<td>82.8%</td>
<td>4.864</td>
<td>56.0%</td>
</tr>
<tr>
<td>UDP System</td>
<td>863.68</td>
<td>100.0%</td>
<td>8.688</td>
<td>100.0%</td>
</tr>
<tr>
<td>x86 Core</td>
<td>Core+L1</td>
<td>9700</td>
<td>n.a</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 4: UDP Power and Area Breakdown.

25. In the preliminary experiment, we only accelerated the raw data transformation happened at the leaves of the query plan, which transforms the tabular text format into the SparkSQL internal row format. We experimented using Apache Spark 2.2 (single executor, single thread, 1GB memory) with a single 64-lane UDP hardware accelerator, on a dual-core Intel Core-i5 (hyperthreading) @2.1GHz, 4GB DDR3, and 256GB SSD machine. We recorded the execution traces and replayed the logs offline to project the performance using the replay tool we built in Figure 26. It replaces the wall clock time of the UDP simulation with the actual UDP ASIC acceleration time using the same configurations in Section 6.6.4. With the updated task time, the replayer reconstructs and schedules the Spark DAG and models the accelerated application performance. In Figure 27, we show the single-thread performance of our ACCORDA system versus SparkSQL on Q4 using s1-scale datasets. Only stage 0 and stage 1 are accelerated by our UDP accelerator, which are mainly parsing and deserialization computations. The ACCORDA system achieves more than 2x speedup than the baseline for the entire query with only accelerating the data loading. In Figure 28, we further show the isolated data loading performance. The ACCORDA delivers 9.3GB/s throughput, more than 300x speedup than SparkSQL. The ACCORDA system indeed provides cheap and fast data transformations.

6.7 Discussion

We prototyped the UDP ASIC design using the TSMC 28nm process. We demonstrated the generality by implementing a diverse set of the data transformation algorithms on our architecture ranging from pattern matching, (de)compression, encoding, to histogram. The high-performance UDP architecture out-performs the 8-threaded CPU 20x on average (Figure 23a). We also showed the prototyped ASIC design requires less than 1% of the CPU chip area and 400x less power, matching the energy efficiency of the current state-of-art hardwired ASIC designs. The implementation study shows that the Si area and power costs for the UDP are low, making it suitable for CPU offload by incorporation into the CPU chip, memory/flash controller, or the storage system. The preliminary ACCORDA results show a dramatic speedup can be achieved via more
ATOs with aggressive query optimization.

The success of the UDP architecture has a significant impact on the system research. It provides the hardware foundation for the accelerated transformation operator. The generality and programmability allow the UDP to accelerate data transformations in various stages of the query execution. The query optimizer is free to choose any advantageous data representations without concerning the actual transformation cost. Besides, the computer architecture contribution is also notable. It opens up a series of exciting directions including exploration of new applications that benefit from the UDP data transformation (e.g., bioinformatics), new domain-specific languages and compilers that provide high-level programming support, and specific design studies that incorporate UDPs (one or several) at various locations in data center systems or database appliances.
Figure 27: TPCH Query 4 Time Breakdown.

Figure 28: TBL Transformation Performance.
References


