2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-9423-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/HiPC56025.2022.00038

A GPU-accelerated Data Transformation Framework Rooted in Pushdown Transducers

Tri Nguyen and Michela Becchi
NC State University
Raleigh, USA
{tmnguye7, mbecchi}@ncsu.edu

***Abstract*—With the rise of machine learning and data analytics, the ability to process large and diverse sets of data efficiently has become crucial. Research has shown that data transformation is a key performance bottleneck for applications across a variety of domains, from data analytics to scientific computing. Custom hardware accelerators and GPU implementations targeting specific data transformation tasks can alleviate the problem, but suffer from narrow applicability and lack of generality.**

To tackle this problem, we propose a GPU-accelerated data transformation engine grounded on pushdown transducers. We define an extended pushdown transducer abstraction (effPDT) that allows expressing a wide range of data transformations in a memory-efficient fashion, and is thus amenable for GPU deployment. The effPDT execution engine utilizes a data streaming model that reduces the application's memory requirements significantly, facilitating deployment on high- and low-end systems. We showcase our GPU-accelerated engine on a diverse set of transformation tasks covering data encoding/decoding, parsing and querying of structured data, and matrix transformation, and we evaluate it against publicly available CPU and GPU library implementations of the considered data transformation tasks. To understand the benefits of the effPDT abstraction, we extend our data transformation engine to also support finite state transducers (FSTs), we map the considered data transformation tasks on FSTs, and we compare the performance and resource requirements of the FST-based and the effPDT-based implementations.

***Index Terms*—Keywords - Finite state transducers, Pushdown transducers, Data transformation, GPU acceleration**

I. Introduction

In recent years, with the rise of machine learning and data analytics, the ability to process and analyze large and diverse sets of data efficiently has been crucial for performance of applications in both the business and scientific realms. Many of these applications require some form of data transformation [1]–[4]. In addition, researchers have shown that data transformation is a key performance bottleneck for in-memory data analytics systems, especially at times when the data size can scale to the order of petabytes [5]–[8].

Several classes of data transformation tasks are at the core of popular applications. For example, consider extract-transform-load (ETL) workloads. ETL applications require *extracting* information from potentially large data sources using different formats (e.g., CSV, XML, JSON), *transforming* the data (e.g., by decoding, filtering, sanitizing, encoding), and *loading* them into a destination data storage. Data transformation kernels used by these applications encompass: *parsing and data query*, *data encoding and decoding*, *transformation into a target format*, and *data analysis* (e.g., generation of statistics representations such as histograms). Further, data transformation is part of scientific applications. An important problem for applications relying on sparse matrices or graphs is the selection of an encoding format that allows for storing the data compactly without introducing a performance bottleneck for the application [9]. For example, consider sparse matrix vector multiplication (SpMV), an important kernel in many scientific applications [10]. Due to its memory access patterns, a sparse matrix layout that maximizes compression might negatively affect the performance of the computation. Furthermore, the selection of the most suitable matrix format can be affected by the hardware characteristics of the underlying system. There has been a considerable amount of work on the design of sparse matrix formats that can offer good storage, algorithmic and system requirements [11]–[15]. The ability to transform between these formats is crucial for achieving a good trade-off between storage requirements and computation performance.

Due to high branch misprediction rate, poor cache locality and irregular memory access patterns, many of these data transformation kernels exhibit poor performance on CPU. Past work has proposed accelerated CPU libraries, custom GPU implementations, and hardware accelerators for specific data transformation tasks, such as matrix transformation [16], data encoding and decoding [17], parsing [18], and other data transformation kernels from social media, audio, video and bio-signal data [19]–[22]. These solutions, however, lack generality and flexibility.

A more general approach consists of identifying the computational abstraction at the core of these tasks and providing an efficient implementation of that abstraction. This idea has motivated a large of body of work on automata processing, which has led to a number of GPU [23]–[25], FPGA [26]–[28] and custom hardware designs [29]–[34]. Since finite state automata recognize regular languages, those works address *search* applications requiring various kinds of pattern matching on textual data. Automata traversal accelerators, however, cannot be simply adapted or extended to support data transformation. By focusing on pattern search, they don't provide efficient support for dynamic output [27], [32], [35]. Nevertheless, the successes reported on automata processing suggest that accelerating a computational abstraction can be beneficial for an entire class of applications.

2640-0316/22/$31.00 ©2022 IEEE
DOI 10.1109/HiPC56025.2022.00038

In this work, we aim to provide a general and efficient means to support *data transformation kernels* used by popular applications and emerging workloads. To this end, we study the effective GPU implementation of pushdown transducers, a computational abstraction that covers a broad range of data transformation tasks. We note that standard pushdown transducers can exhibit memory requirements and complexities that make them not amenable for GPU acceleration. To address this problem, we propose effPDTs, a memory-efficient, GPU-friendly extension of pushdown transducers. Compared to previous works that explore theoretical and syntactical aspects of various categories of transducers [36]–[38], our proposed extensions arise from the goal of supporting a wide range of data transformation kernels in an efficient manner.

The final outcome of this work is a flexible GPU-accelerated data transformation framework. Our specific contributions are:

- The design of effPDT, a compact pushdown transducer model suitable for GPU acceleration;
- A data transformation engine based on effPDTs, including a set of pre- and post-processing kernels to partition input and output streams across compute units;
- The mapping of a diverse set of data transformation tasks on effPDTs and standard finite state transducers (FSTs);
- An evaluation of our GPU-accelerated data transformation framework against publicly available custom CPU and GPU libraries;
- A performance comparison between an effPDT- and a FST- based engine.

Our experiments show an average speedup of 17x over custom CPU libraries, and performance on par with, and in some cases better than, custom GPU libraries. In addition, our results confirm that, due to their memory efficiency, not only are effPDTs preferable to FSTs, but they also provide consistent performance independent of algorithmic-specific parameters settings.

Our framework will be available in open-source at the following link: *https://github.com/tringuyen0601/effPDT*

II. Background

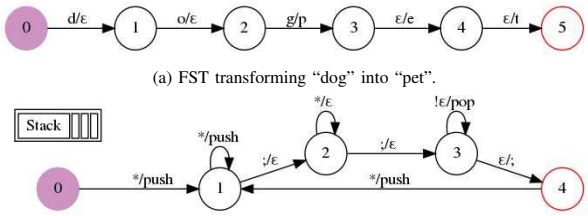
In this section, we provide some background on finite state transducers and pushdown transducers. In all cases, data transformation is performed by traversing the transducer based on the content of the input stream.

A. Finite State Transducers

Formally, a finite state transducer (FST) [39] is defined as a quintuple N = (Q, Σ, δ, s, F) such that:

- Q is a finite set of states;
- Σ is an alphabet such that $\Sigma = \Sigma_I \cup \Sigma_O$, where Σ_I is the input alphabet and Σ_O is the output alphabet;
- $\delta \subseteq Q \times (\Sigma_I \cup \{\epsilon\}) \times Q \times (\Sigma_O \cup \{\epsilon\})$ is a finite state transition relationship, ϵ being the empty string;
- $s \in Q$ is the start state;
- $F \subseteq Q$ is a set of final states.

Operationally, an FST transforms a streaming input with alphabet Σ_I into a streaming output with alphabet Σ_O based on the transition relationship δ. Specifically, an FST transition r = (q_1, σ_{O1}, q_2, σ_{O2}) belonging to δ is triggered when state q_1 is active and input symbol σ_{O1} is fed to the FST, and it causes state q_2 to be activated and symbol σ_{O2} to be generated. Symbols are written to the output stream upon traversal of a final state. For example, the FST in Figure 1(a) transforms input "dog" into output "pet".



(a) FST transforming "dog" into "pet".

(b) PDT that reads a csv file containing a set of first and last names and generates the anadromes of the first names.

Fig. 1: Examples of finite state and pushdown transducers (FST and PDT). Final states are highlighted in red.

We define *execution context* as the sequence of symbols that were processed before the current input symbol. An FST conveys its execution context through the currently activated states. In a deterministic FST, the output generated at each step depends solely on the execution context, and not on future inputs. In practice, this means that the execution will never diverge into two traversal paths.

FST have found application in speech and language processing (for example, for the representation of large dictionaries, grammars and in computational morphology) [40].

B. Pushdown Transducers

Formally, a pushdown transducer (PDT) [39] is defined as a quintuple P = (Q, Σ, δ, s, F) such that:

- Q is a finite set of states;
- Σ is an alphabet such that $\Sigma = \Sigma_I \cup \Sigma_O \cup \Sigma_S$, where Σ_I is the input alphabet, Σ_O is the output alphabet and Σ_S is the stack alphabet;
- $\delta \subseteq Q \times (\Sigma_I \cup \{\epsilon\}) \times (\Sigma_S \cup \{\epsilon\}) \times Q \times (\Sigma_O \cup \{\epsilon\}) \times (\Sigma_S \cup \{\epsilon\})$ is a finite state transition relationship;
- $s \in Q$ is the start state;
- $F \subseteq Q$ is a set of final states.

A PDT is essentially a finite state transducer with a stack. In addition to writing to an output stream, a PDT can pop symbols from a stack and push symbols onto it. A PDT transition r = (q_1, σ_{I1}, σ_{S1}, q_2, σ_{O2} σ_{S2}) is triggered when state q_1 is active, the current input symbol is σ_{I1}, and symbol σ_{S1} is at the top of the stack. Upon traversal, the transition will activate a new state q_2, generate output symbol σ_{O2}, pop symbol σ_{S1} from the stack and push symbol σ_{S2} onto it. For example, Figure 1(b) shows a PDT that reads a comma-separated values file of first and last names, extracts the first names, and outputs their anadromes. The stack allows recording the first names for later output generation.

A PDT conveys its execution context through the currently activated states and the current top of the stack. By adding memory to a FST, the stack leads to an increased expressive

power and, if an equivalent FST exists, it allows to significantly reduce the number of states. For example, a FST can count a predefined number of occurrences of a symbol in an input stream while a PDT can count an arbitrary number of occurrences of that same symbol. Our proposed effPDT model extends pushdown transducers.

III. Efficient Pushdown Transducers

Pushdown transducers provide a solid theoretical foundation for data transformation processes that involve deep contextual evaluations, including various forms of data encoding, decoding, and parsing. However, their expressive power and ability to encode data transformations in a compact way are limited by their reliance on a single stack, their use of a single input and output stream, and their lack of arithmetic support. Our goal is two-fold: on the one hand, we want to support a wide range of data transformations; on the other, we aim at a compact representation that can be efficiently deployed on GPU (and potentially other hardware accelerators). With this in mind, we introduce effPDT, a transducer model aimed to describe a wide variety of data transformations in a memory-efficient manner. To this end, we apply the following extensions to PDTs: *multiple stacks*, *multiple input and output streams*, and *arithmetic operations* associated to states.

A. Definition

Formally, a effPDT is defined as a 10-tuple TF = (Q, Σ, S, I, O, Δ, δ, γ, s, F) such that:

- Q is a finite set of states;
- Σ is an alphabet such that $\Sigma = \Sigma_I \cup \Sigma_O \cup \Sigma_S$, where Σ_I is the input alphabet, Σ_O is the output alphabet and Σ_S is the stack alphabet;
- S is a finite set of stacks;
- I is a finite set of input streams;
- O is a finite set of output streams;
- Δ is a finite set of arithmetic/logical operators;
- $\delta \subseteq Q \times I \times (\Sigma_I \cup \{\epsilon\}) \times S \times (\Sigma_S \cup \{\epsilon\}) \times Q \times O \times (\Sigma_O \cup \{\epsilon\}) \times S \times (\Sigma_S \cup \{\epsilon\})$ is a finite state transition relationship;
- $\gamma \subseteq Q \times (\Delta \cup \{\perp\}) \times \mathcal{P}(S \cup \{\epsilon\})$ is the action relationship, with $\perp$ denoting a lack of action on a state;
- $s \in Q$ is the start state;
- $F \subseteq Q$ is a set of final states.

A effPDT transition r = (q_1, i_1, σ_{I1}, s_1, σ_{S1}, q_2, o_2, σ_{O2}, s_2, σ_{S2}) is triggered when state q_1 is active, the current symbol on input stream i_1 is σ_{I1}, and symbol σ_{S1} is at the top of stack s_1. Upon traversal, the transition will activate state q_2, write symbol σ_{O2} onto output stream o_2, pop symbol σ_{S1} from stack s_1, and push symbol σ_{S2} onto stack s_2. We assume a deterministic transducer, where the execution never diverges into multiple traversal paths. Action g=(q, ψ, s_1,..,s_k) indicates that the activation of state q causes the top of stack s_1 to be assigned the result of applying operator ψ to the values on top of stacks s_2,..,s_k. Without loss of generality, we assume at most one action per state (a sequence of arithmetic/logic operations can be implemented by associating the actions to states connected by epsilon transitions).

B. Expressive Power of effPDTs

Here, we discuss the effect of the three PDT extensions listed above on expressive power. *Multiple stacks*: It has been proven that a two-stack pushdown automaton (and, by extension, a k-stack pushdown automaton) is equivalent to a Turing machine [39]. In fact, the two stacks effectively create an addressable memory, which is equivalent to a Turing machine's two-way tape. Since PDTs are a generalization of push-down automata, the multiple stacks extension provides to effPDTs the expressive power of a Turing machine. *Multiple input and output streams*: It has been proven that a multi-tape Turing machine can be simulated through a single-tape machine [41]. Therefore, the multiple input/output streams extension does not add expressive power to effPDTs. *Arithmetic operations*: Since arithmetic and boolean functions can be simulated through Turing machines, this extension does not add expressive power. In summary, while the multiple stacks extension provides Turing equivalence, the other two extensions are meant to reduce complexity and increase efficiency/performance, but do not add expressive power.

C. Practical Considerations on PDT Extensions

Here, we motivate in a pragmatic fashion the extensions to traditional transducers that we have introduced in effPDT.

Multiple stacks. The stack allows saving the input history. On each transition, a PDT can either compare the top of the stack with the current input, or use the stack as a source for dynamic output creation. However, a PDT does not offer a mechanism to compare or modify symbols at different positions of the stack. The extension of multiple stacks avoids this problem, and allows the transducer to *dynamically evaluate past symbols regardless of the order they appear*, while architecturally using the same amount of memory as a shared stack. In addition, stacks allow for *more compact transducer representations*, resulting in space efficiency. To understand why, consider the practical case where the PDT has a *finite stack* (an infinite stack is simulated by using a stack large enough for the considered problem). A PDT with a *finite stack* can support the same transformations as a FST. However, the PDT does so with significantly fewer states and transitions. To understand why, consider the FST construction process [39]. Essentially, FST construction requires: (i) enumerating the sequences of outputs corresponding to admissible input subsequences for the considered data transformation, (ii) generating a non-deterministic FST where each input/output sequence pair is mapped to a sequence of states (using ϵ transitions when the two sequences have different length), and (iii) reducing the FST by subset construction (optional optimization step). For example, consider the PDT of Figure 1(b). A finite stack of size n would support first names up to n character long. Thus, an equivalent FST could be constructed by enumerating all possible first names (i.e., all possible sequences of length n), each leading to a chain of states of length n. As a result,

using an FST rather than a PDT would cause a combinatorial growth in the number of states. In practice, the use of stacks allows for compact transducers with a number of states and transitions that is independent of the alphabet size and other data transformation specific parameters (see Section VII). On GPU, this allows for compact transducer representations that can make efficient use of the available memories.

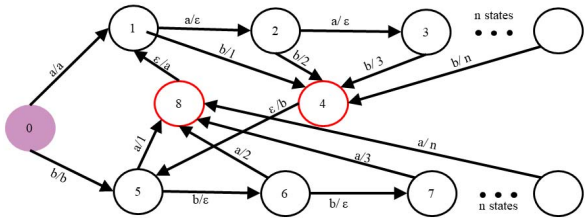
Multiple input and output streams. FST and PDT assume a single input and a single output stream. This model supports well transformations that convert a single stream of data into a single output stream. However, some data transformations can be more effectively expressed using multiple input/output streams. For example, sparse matrix layouts are often organized into separate arrays, and mapping different arrays onto separate streams enables more efficient implementations. Generally, a system with a single output stream can accommodate multiple outputs by interleaving the data corresponding to the different outputs and then adding a post-processing step to split the output stream, or by requiring multiple passes over the input. Both approaches, however, limit efficiency. Similarly, a system with a single input stream cannot handle efficiently transformations where data belonging to an input can change the processing decision of another input. For example, the compressed sparse row (CSR) format consists of three arrays (values, column indexes and row indexes); in the CSR-to-dense matrix transformation, a value from each of these arrays must be read before a dense matrix value can be produced.

Arithmetic/logic operations. effPDT associates actions (in the form of arithmetic operations on stack values) to states. In standard transducer models, states serve only as source and destination of transitions, and can indicate input acceptance (final states). Transitions read symbols from the input stream and the stack, and they write symbols to the output stream and the stack. Since transitions cannot modify values read, they can only produce a static output in the case of FST or an output including previously processed symbols in the case of PDT. Adding arithmetic operations to states avoids this limitation. Arithmetic operations on stack values can also allow for a significant reduction in the transducer size. For example, instead of enumerating all possible input values on distinct transitions, each with a different output value, a effPDT can save an input value to a stack, modify it and then output it using a single transition. Furthermore, if a stack is used to count the number of occurrences of a given symbol, the use of arithmetic operations enables supporting the same transformation using a single-element stack (a counter).

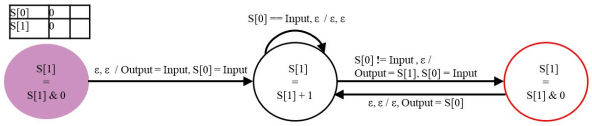
D. effPDT Construction

While providing a programming model for effPDT is outside the scope of this paper, here we describe briefly the systematic approach we take to construct effPDT.

First, we break the data transformation algorithm into a series of sequential steps, which include: initialize a stack, push an input symbol on a stack, write a symbol (either from a stack or from an input stream) into an output stream, perform arithmetic operation on top of a stack. The sequential



(a) FST implementing RLE for alphabet {a,b} and maximum run-length of n



(b) effPDT implemeting RLE (generic alphabet size and maximum run-length)

Fig. 2: FST and effPDT for the run-length encoding (RLE) scheme, accepting states are colored red.

program can contain if-statements (dependent on the value of an input or of the top of a stack) and loops expressed using goto-statements. The number of input/output streams and of stacks required and the stacks' size (which can be made configurable) depend on the algorithmic needs. For example, in the transformation in Figure 1(b) the size of the stack can be limited to the maximum length of the names to be processed. Second, we go through the sequential description and create and connect states according to the algorithmic steps and the control-flow of the program. For example: stack initialization instructions are associated to the entry state, reading input and writing output require one transition to a new state, sequences of arithmetic instructions require a chain of states connected through (non-consuming) epsilon transitions - one for each arithmetic instructions, if-statements require adding multiple outgoing transitions from a single state, goto-statements require backward transitions to a previously instantiated state.

For example, Figure 2 shows the FST and effPDT for run-length encoding (RLE), a data transformation that compresses an input text by storing runs of data (i.e., consecutive occurrences of the same symbol) as a single data value and count. For example, input *aaaabb* is transformed into output *a4b2*.

FST Construction: As explained above, building an FST requires enumerating all possible input/output sequence pairs (for RLE: $a/a1$, $aa/a2$, $aaa/a3$, etc.) and creating a chain of states for each pair. The ϵ symbol indicates that the corresponding transition either does not consume any input symbol, or does not generate any output. Transitions with the same input and output can be combined, leading to the corresponding target states to be merged.

effPDT Construction: The algorithm is broken down into the following steps: (a) initialize a counter (stack S_1), (b) read an input symbol i; (c) save i on a stack (S_0); (d) write i to the output stream, (e) read a symbol i; (f) if i is equal to the symbol recorded in S_0 increment a counter (stack S_1) and read the next symbol (goto-statement to (e)), else save i on S_0 and write the counter S_1 to the output; (g) write S_0 to the output stream; (h) read the next symbol (goto-statement to (e)). Step (a) is associated to the entry state 0. Reading an

State	Action
0	ANDI, S[1],0,S[1]
1	ADDI, S[1],1,S[1]
2	ANDI, S[1],0,S[1]

(a) State Table

Tx	Src st	Input cond	Stack cond	Dst st	Output	Stack
0	0	I[0]==*	ϵ-ϵ	1	O[0]=I[0]	S[0]=I[0]
1	1	I[0]==*	S[0]==I[0]	1	ϵ-ϵ	ϵ-ϵ
2	1	I[0]==*	S[0]==!I[0]	2	O[0]=S[1]	S[0]=I[0]
3	2	ϵ-ϵ	ϵ-ϵ	1	O[0]=S[0]	ϵ-ϵ

(b) Transition Table

Fig. 3: State and transition tables for the RLE effPDT of Figure 2(b)

input and updating an output and a stack can be done in a single transition. So, steps (b-d) can be combined in a single transition causing the creation of a new state 1. Similarly, steps (e-f) can be combined. The if-statement in step (f) leads to two outgoing transitions from state 1 conditional on i being equal to S_0. The goto-statements cause the creation of transitions to the already instantiated state 1.

IV. effPDT Engine

In this section, we describe the effPDT execution model.

A. Components

At its core, the effPDT engine comprises three data structures: *topology*, *I/O buffers* and *contextual information*. The topology is static and encoded through *state table*, *transition table* and *stacks*. Figures 3(a) and (b) show the state and transition tables for the RLE effPDT of Figure 2(b). The I/O component contains an input/output buffering system that operates in parallel with the execution engine to ensure that the effPDT engine never stalls waiting on an input or for the output to clear up. Lastly, the contextual component describes the effPDT's current state, including: currently activated state, stacks' content, and number of symbols read from the input and written to the output.

B. Execution Engine

The effPDT's execution engine consists of three stages: the *action*, *matching* and *writing* stages. The engine keeps executing as long as it has an active state and an input to process. At the beginning of execution, the transducer's entry state is active. At each execution step, the three stages operate as follows. In the *action stage*, the engine executes the arithmetic/logic instruction (if any) associated to the active state. Then, it scans the transition table to determine the transitions outgoing from that state (*pending transitions*). In the *matching stage*, the engine evaluates the pending transitions and, based on their input symbol and stack condition, it determines the transition to be taken (*matching transition*). We make the distinction between a pass-through (*) and an empty (ϵ) symbol. While conditions on both these symbols always evaluate to true, a *-symbol indicates an input/stack symbol consumption, whereas an ϵ-symbol does not. In the *writing stage*, the engine updates the selected output stream and stack (if any) based on information on the matching transition, and then sets the active state to the destination state of that transition. If no matching transition is found, the writing stage has no effect.

TABLE I: Summary of effPDT data structures

Domain	Component	Operation	Access	Location
Topology	State Table	Read Only	Global	Const
	Tx Table	Read Only	Global	Const
I/O	Input/Output streams	Read/Write	Global	Global
Context	Stacks	Read/Write	Thread	Shared
	Pending Tx	Read/Write	Thread	Local
	Matching Tx	Read/Write	Thread	Local
	Active State	Read/Write	Thread	Local
	I/O status	Read/Write	Thread	Local

C. Design Considerations

effPDT's two-table layout (represented in Figure 3) aims to keep transducer traversal cost low. To this end, the *state table* is directly indexed using the state identifiers. In the *transition table*, transitions outgoing from the same state are laid out contiguously. Besides the information shown in Figure 3, each row of the state table contains two additional fields: the index of the first outgoing transition from the corresponding state, and the number of its outgoing transitions.

The effPDT engine uses two circular buffers: one for the input and the other for the output. In both cases, one buffer interacts with the transducer while the other interacts with the disk. On the input side, the inner buffer is read one symbol at a time by the transducer while the outer buffer is periodically filled with data from the disk. On the output side, the transducer writes to the inner buffer symbol by symbol while the execution engine transfers the content of the outer buffer to disk once it is full. Periodic data transfers between inner and outer buffers overlap with transducer's accesses to the inner buffers. On both input and output, the outer buffer prevents execution from stalling waiting on the inner buffer while still allowing bulk data transfers to and from disk or between CPU and GPU memory.

V. GPU Implementation

In this section, we discuss effPDTs' GPU deployment.

A. Processing Engine

Parallelization Approach: In order to allow for parallel execution, the effPDT engine partitions the input, assigns chunks of it to worker threads for processing, and then merges the outputs of the threads. The design aims to partition and place the data efficiently while minimizing inter-thread dependencies. Context information is accessed locally by each thread and is replicated across threads to ensure independent execution environments without the need for synchronization. Besides active state, stack content and pending/matching transitions, context information includes the per-thread starting offset within the input and output streams, chunk size and counters to manage each thread's input/output coverage.

Data structures: Table I summarizes the data structures used by the effPDT engine, their access type and granularity, and their placement in the GPU's memory hierarchy.

The *state and transition tables* are read-only data structures globally accessed by all threads. Thus, they are stored in constant memory, which is cached. We recall that the use of stacks allows for reducing, in some cases significantly, the

number of states and transitions necessary to express a data transformation through a transducer. This allows topology data of effPDTs to fit in the relatively small constant memory.

Stacks are read/write data structures. To allow for fast access, they are stored in shared memory. We implement each stack as a circular buffer. We provide basic $push/pop$ commands, as well as a $popall$ command, which writes the entire stack content to the output. This command allows for more efficient support of transformations involving parsing of structured data. We recall that arithmetic operations are performed on the top of the stack (specified by the tail pointer).

The remaining thread-level *context data* include active state, pending and matching transitions, number of symbols read and written by each thread, and chunk offset and length. Pending transitions are stored as base and offset into the transition table, requiring two 32-bit variables. The other context data require one variable each. Thus, we store this information in the low latency, high bandwidth register file.

Input and output streams are stored in global memory.

B. Pre/Post-processing Kernels

Along with the transducers processing engine, the GPU implementation includes a library of kernels to partition the input stream across threads and consolidate the outputs generated by the threads into a single output stream (*pre-processing* and *post-processing* kernels, respectively).

Pre-processing kernels generate the offset and length of the input chunks assigned to the threads. Our effPDT framework supports static and dynamic partitioning. In static partitioning, each thread is assigned an equal size chunk of the input. In addition, the system allows defining specific algorithmic requirements on the chunk size. For example, the run-length decoding and bit-packing encoding schemes require the chunk size to be a multiple of 2 and 4, respectively. Furthermore, the variable length encoding/decoding schemes require chunks to overlap. Dynamic partitioning, on the other hand, allows threads to be assigned differently sized chunks. Chunk sizes can be specified by the user or determined using an additional transducer. Once the chunk sizes are determined, the pre-processing kernels perform a prefix sum to quickly generate each thread's offset within the input stream.

Post-processing consists of two steps: *post-compute* and *post-copy*. Post-compute refers to any additional processing required before merging the output chunks. Our framework includes methods to support common operations such as tail reduction and head-tail merging. In addition, it includes an interface for users to provide custom post-compute kernels. Post-copy refers to the process of eliminating any holes in the output stream. For some data transformations, the size of the output chunks can be determined statically based on the size of the input, the number of threads, and possibly other parameters. For transformations where this is not possible, buffers storing the output chunks must be over-provisioned, leading to holes between output chunks. In these cases, the post-processing kernel will perform a *fragment copy*, which copies output chunks contiguously to the final output.

TABLE II: Benchmarks summary

Application	Input dataset	CPU Baseline	GPU Baseline
Data Enc/Dec	Cantebery Corpus, Artifical Corpus [42]	Parquet [17]	Nvidia Thrust [43]
Matrix Transform	Texas A&M Sparse Matrix [44]	Intel MKL [16]	Nvidia cuSparse [43]
Histogram	RDU Accident and Crime Report [45]	GSL Histogram [18]	Nvidia CUB [43]
CSV Query	RDU Accident and Crime Report [45]	Pandas [46]	Rapids AI [47]

For example, let us consider the pre- and post-processing operations required by run-length encoding (RLE). During pre-processing, the input offsets and chunk sizes can be determined through static partitioning, and a simple *Memset* call is enough to initialize the array of chunk sizes. During post-processing, head-tail merging is required to handle cases where the last symbol of one chunk is the same as the first symbol of the next chunk. Finally, since for RLE the output size cannot be determined a priori, a fragment copy is performed to eliminate any holes within the output before transferring it back to CPU.

VI. Experimental setup

In this section, we detail the software and hardware setups used in our experiments. We evaluated our effPDT engine on 11 data transformations from four application classes: data encoding/decoding, matrix layout transformations, histogram construction and structured data query (using the CSV format).

A. Input Datasets and Baseline CPU and GPU Kernels

Table II summarizes the data transformation kernels and input datasets used in our experiments, and the custom CPU and GPU library implementations we compared against. We note that GPU libraries are available only for a subset of the data transformation kernels considered. **Data encoding/decoding kernels** include: bit-packing encoding and decoding (BPE and BPD), variable-length encoding and decoding (VLE and VLD), and run-length encoding and decoding (RLE and RLD). Custom CPU code is from the C++ Parquet library [17], custom GPU code is available only for RLE and RLD and is part of Nvidia Thrust library [43]. Inputs are from the Canterbury Corpus and Artificial Corpus Datasets [42] with file sizes ranging from 4KB to 2MB. **Matrix transformations kernels** include: the transformation from coordinate list to compressed sparse row format (COO-CSR), and the transformation from dense to compressed sparse row format (Dense-CSR). Custom CPU code is from Intel MKL Sparse Matrix library [16], custom GPU code is from Nvidia cuSparse library [43]. Input datasets are from Texas A&M Open Source sparse matrix collection [44] with sparsity ranging from 0% to 99% (g7jac160, xenon1, copter, imcol and trec). The **histogram construction kernel** uses a 4-bin and a 10-bin setup. Custom CPU code is from GNU GSL Histogram library [18], GPU code is from Nvidia CUB library [43]. Datasets are from Raleigh Sustainable Project (longitude, latitude) and Crash Location (FeetFromRoad) [45]. **CSV query kernels** are transformations that extract a subset of a CSV file based on a

TABLE III: Benchmarks: topological characteristics and resource utilization. Shared and constant memory utilization per thread-block (Sh-M and Const-M) is measured in KB. We set the block size to 128 threads.

	Topology			Architecture		
	#State	#Tx	#S	Registers	Sh-M	Const-M
BPD	15	15	4	23	3.07	1.51
BPE	17	17	3	23	2.56	1.66
VLD	14	15	6	23	4.10	1.48
VLE	6	7	2	23	2.05	0.91
RLD	3	4	2	23	2.05	0.69
RLE	3	4	2	23	2.05	0.69
COO-CSR	11	14	4	23	4.10	1.36
CSR-D	15	17	6	23	5.12	1.60
Histogram	6	10	4	23	3.58	1.1
CSV_Encd	8	15	2	23	2.56	1.3
CSV_Raw	10	17	2	23	18.9	1.4

specified user condition. We conduct our experiments on raw, unedited CSV and dictionary-encoded CSV. Custom CPU code is from Pandas [46], GPU code is from Nvidia's RAPIDS AI [47]. Datasets are Raleigh Sustainable Project (owner, status) and Crash Location (FeetFromRoad, Day_Of_Week) [45]. For each data transformation kernel, in Section VII we report the average performance across the input datasets used.

B. EffPDTs Characteristics

Topological characteristics: Table III reports the number of states, transitions and stacks of the resulting effPDTs, as well as their resource requirements (registers, shared memory and constant memory). We also encode a subset of these data transformations using FSTs: Table V reports the FST-related data for different parameters settings.

Number of streams: The effPDTs implementing the data encoding/decoding, histogram and CSV querying kernels include one input and one output stream. The COO-CSR effPDT uses three input and three output streams, while the Dense-CSR effPDT uses one input and three output streams.

Considerations on CSV query kernels: CSV query requires recording the content of one or more fields, and subsequently writing the recorded fields to the output stream if a particular condition is met. For example, consider a CSV file that contains 3 columns: *Name*, *Age*, and *Height*. If a user queries the *Name* of all people with a given *Height*, the effPDT must temporarily record the content of the *Name* field until the *Height* field is read and the condition on it is evaluated. For a dictionary-encoded-CSV file (CSV_encd), one-element stacks are enough to record the value of a field. On the other hand, CSV files with fields of arbitrary content (CSV_Raw) require deeper stacks. Thus, the effPDTs for CSV_encd and CSV_raw have a similar number of states and transitions and the same number of stacks. However, CSV_raw requires more shared memory to accommodate the stacks (Table III).

C. Pre/Post Processing

Table IV summarizes the type of pre- and post-processing required by each transformation. *Pre-processing*: For all transformations except COO-CSR and CSV query, we perform static partitioning. Since static partitioning leads to equally sized chunks, it only requires a simple CudaMemset to initialize the array storing the chunk sizes. The input offset of each thread is then determined by multiplying the chunk size by the thread identifier. For Dense-CSR, the rows of the matrix are equally distributed across threads. COO-CSR and CSV query use a custom primitive to calculate the per-thread chunk size, and prefix sum (Scan) to set each thread's input offset. *Post-processing*: VLD, RLE, matrix transformations and histogram construction require post-computation before output merging. We recall that, for transformations whose output size cannot be determined statically, a fragment copy is required before the output is transferred to the CPU. Fragment copy is implemented through a prefix sum followed by a copy-to-offset operation. All kernels except BPD, BPE and histogram construction require this extra copy.

D. System Configuration

We conducted our experiments on a system equipped with two Intel Xeon E5-2630 processors running at 2.2GHz, each with ten physical cores and a total 25MB of cache. The system is also equipped with an NVIDIA TITAN XP GPU, which has 12GB global memory, 64KB constant memory and 98KB shared memory per streaming multi-processor (SM). The GPU has 30 SMs operating at a maximum clock rate of 1.58GHz. In addition, our system has 130GB RAM and a 1TB SSD.

The system has installed Ubuntu 18.04, gcc 7.5 and CUDA toolkit 11.7. Our baseline CPU experiments are parallelized to process multiple input streams in parallel and use all available CPU cores. The effPDT- and FST-based kernels are configured to use all the available SMs.

VII. Performance Evaluation

We performed two sets of experiments. In the first set (Section VII-A), we evaluated the performance of our GPU-accelerated effPDT engine over custom CPU and GPU libraries (whenever available). In the second set (Section VII-B), we compared the performance of effPDTs and FSTs when varying algorithmic parameters that affect the FST size.

A. Overall Performance of GPU-accelerated effPDT engine

1) Comparison with Custom CPU Libraries & Impact of Data Placement on Performance: Figure 4 shows the speedup reported by our effPDT engine over the custom CPU libraries listed in Section VI-A. The first data series (*CPU effPDT*) corresponds to a parallel CPU implementation of the effPDT engine configured to use all available cores. For our GPU

TABLE IV: Pre/post-processing schemes. * indicates special requirements on chunk size.

	Pre-processing		*Post-processing*	
	Dependency	**Operation**	**Compute**	**Copy**
BPD	Input size	Memset	None	Full
BPE	Input size*	Memset	None	Full
VLD	Input size*	Memset	Tail-reduce	Frag
VLE	Input size	Memset	None	Frag
RLD	Input size*	Memset	None	Frag
RLE	Input size	Memset	Merge	Frag
COO-CSR	Row (Var)	Cust+Scan	Scan	Frag
Dense-CSR	Row (Const)	Memset	VectorAdd	Frag
Histogram	Input size	Memset	VectorAdd	Full
CSV_Encd	Row (Var)	Cust+Scan	None	Frag
CSV_Raw	Row (Var)	Cust+Scan	None	Frag

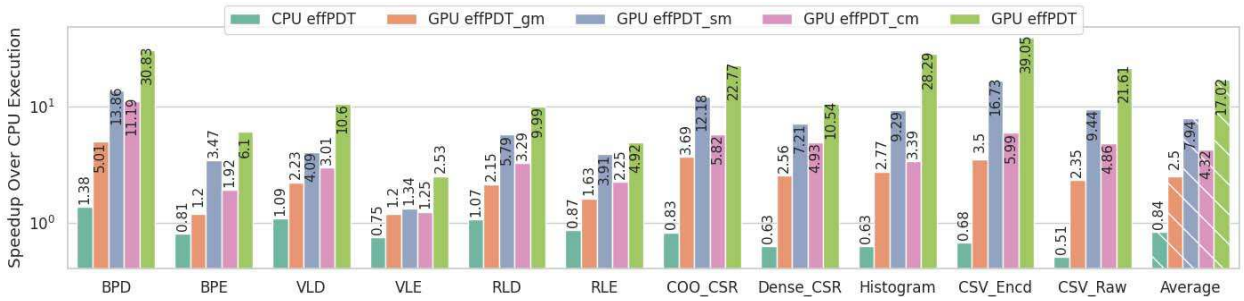


Fig. 4: Speedup of effPDT engine (CPU, GPU/global memory only, GPU/shared+global memory, GPU/constant+global memory, GPU/all memories) over custom CPU baselines in Table II. The values on the bars indicate the speedup. The last set of bars show the average speedup across the benchmarks.

implementation, we show the results reported using four memory configurations. In the *GPU effPDT_gm* configuration, all effPDTs' data structures are stored in global memory. In the *GPU effPDT_sm* and *GPU effPDT_cm* configurations, we either store the stacks in shared memory or the effPDT's topology in constant memory, respectively. Lastly, the *GPU effPDT* configuration corresponds to the implementation described in Section V-A, which utilizes both the constant and shared memories to maximize the system throughput. The numbers on the bars are the speedup values reported by each configuration. The rightmost set of bars show the average results across the benchmarks. We make the following observations.

First, our GPU engine (*GPU effPDT*) reports an average speedup of 17x over the custom CPU library implementations, which corresponds to an average 11 GB/sec throughput. The most significant speedups are reported on CSV querying (30x on average) and histogram construction (28x), followed by matrix transformation kernels (16x on average), and lastly by data encoding/decoding (10x on average).

Second, the use of all available GPU memories is key to performance. When using global memory only, our GPU engine achieves an average speedup of 2.5x over the custom CPU libraries and an average throughput of 1 GB/sec. We observe speedups ranging from 1.2x (VLE) to 5x (BPD) and throughputs ranging from 309 MB/sec (GVE) to 4.3 GB/sec (dictionary-encoded-CSV querying). Using constant and shared memory alone brings performance improvements over the global memory-only setup, with speedups of 4.3x and 8x over CPU execution. Across the board, shared memory offers better performance than constant memory.

Finally, our CPU effPDT engine performs similarly or worse than Parquet on data encoding and decoding (0.7x to 1.38x), and noticeably worse than the custom CPU libraries for matrix transformation and CSV querying. We expected to see this slowdown, since these libraries contain custom implementations of the considered algorithms while the effPDT design is generic and intended for GPU acceleration. In particular, Intel MKL and Pandas are highly efficient libraries and are the industry standard for their respective application class.

2) Comparison with Custom GPU Libraries: Figure 5 reports the throughputs (in GB/sec) achieved by the custom GPU libraries listed in Table II and by our GPU-accelerated effPDT engine on a subset of the considered data transformations. We did not find custom GPU implementations for BPD, BPE, VLD and VLE. For the custom libraries, the throughputs reported do not account for the memory transfers between CPU and GPU, which would further reduce performance. Our effPDT engine achieves performance similar to (or slightly better than) the custom GPU libraries on RLD, RLE and querying of raw CSV data, it underperforms Nvidia CUB on histogram construction, and it outperforms cuSparse and Rapids AI on matrix transformation kernels and querying of dictionary-encoded-CSV data, respectively. We note that the matrix transformation kernels provided in cuSparse performed worse than effPDTs (2.9 and 3.5GB/s compared to 8.7 and 8.1 GB/s) due to the high library setup time. Ignoring the setup step, cuSparse matrix transformations would achieve throughputs close to 10GB/s, slightly higher than effPDTs' results. On the other hand, histogram construction provided in Nvidia cub performs 1.6x faster than effPDT (19.5 GB/s compared to 11.76GB/s). This is because histogram construction is an embarrassingly parallel algorithm with no inter-thread dependencies. Overall, our engine was able to achieve an average throughput of 16GB/s across the considered data transformation workloads, higher than the 11GB/s average throughput reported by the custom GPU libraries. This result suggests that our GPU effPDT engine has the potential for accelerating a variety of data transformations without requiring custom GPU implementations. In addition, we note that best results are obtained on CSV query, suggesting that the effPDT engine can be particularly suitable for parsing structured data.

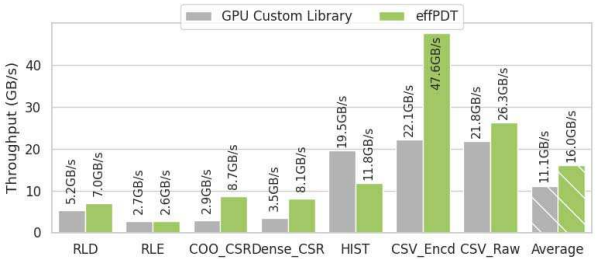
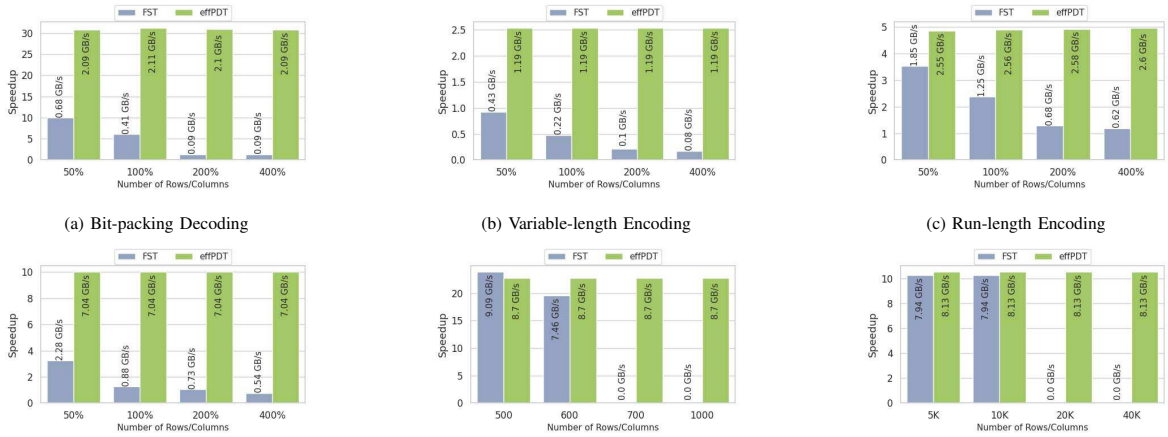


Fig. 5: Throughputs of custom GPU libraries and GPU effPDT engine



(a) Bit-packing Decoding (b) Variable-length Encoding (c) Run-length Encoding

(d) Run-length Decoding (e) COO-CSR (f) Dense-CSR

Fig. 6: Speedup of the effPDT and FST engines over Parquet and Intel MKL, with the parameter settings and FST memory requirements listed in Table V. The values on the bars indicate the throughput in Gbytes/s.

TABLE V: Characteristics of FSTs implementing the considered data transformations. For compression, $|\Sigma|$ is the alphabet size, r is the packing ratio, and l is the maximum run-length; for matrix transformations, n is the square matrix dimension. k, m and b indicate thousands, millions and billions, respectively.

	Topology			**Architecture**	
	$\|\Sigma\|$/n/r/l	**#State**	**#Tx**	**C-M(%)**	**Size**
BPD	64/4	322	385	50	27.8KB
St:$\|\Sigma\|$(r+1) + 2	128/4	642	769	100	53.7KB
Tx:$\|\Sigma\|$(r+2) + 1	256/4	1538	1793	200	122KB
	512/4	3074	3585	400	243KB
VLE	90/2	372	462	50	32KB
St≈ $\|\Sigma\|$(r+1) +1	160/2	722	882	100	62KB
Tx≈ $\|\Sigma\|$(r+2)	315/2	1497	1812	200	124KB
	630/2	3072	3702	400	250KB
RLE	8/10	89	648	50	29KB
St:l$\|\Sigma\|$+$\|\Sigma\|$+1	12/10	133	1452	100	62KB
Tx:$\|\Sigma\|$($\|\Sigma\|$l +1)	17/10	188	2907	200	122KB
	25/10	276	6275	400	256KB
RLD	7/10	393	462	50	32KB
St:$\|\Sigma\|$(l(l+1)/2+1)+1	14/10	785	924	100	64KB
Tx:$\|\Sigma\|$(l(l+3)/2+1)	28/10	1569	1848	200	124KB
	57/10	3193	3762	400	253KB
DENSE-CSR	5000	25m	50m	31m	2GB
St: n^2	10000	0.1b	0.2b	0.1b	10GB
Tx: $2n^2$	20000	0.4b	0.8b	0.6b	40GB
	40000	1.6b	3.2b	1.8b	120GB
COO-CSR	500	250k	0.1b	78m	5GB
St: n^2	600	0.3b	0.2b	0.1b	9GB
Tx: n(n(n-1) +2)	700	0.4b	0.3b	0.2b	15GB
	1000	1m	1b	0.6b	43GB

B. Comparison of effPDT- and FST-based Engines

In this section, we compare the memory requirements and performance of effPDT- and FST-based engines. To perform this set of experiments, we built a lightweight GPU execution engine supporting basic FST operations. Specifically, we reduced the effPDT engine by removing stack-related operations and simplifying the state and transition tables. We select six benchmarks (two encoding, two decoding and two matrix transformations) that can be expressed through FSTs without suffering from state explosion.

1) FST Characteristics: For all considered data transformations, the size of FST topology is dependent on algorithmic parameters: alphabet size and packing ratio between input and output for VLE and BPD, alphabet size and maximum run-length for RLE and RLD, and dimensions of input matrix for Dense-CSR and COO-CSR. Table V shows the formulas expressing size of the FST encoding these transformations (column 1), as well as the number of states, transitions and memory requirements with different parameters settings (columns 2-6). For compression/decompression, we selected parameter settings corresponding to topologies requiring 50%, 100%, 200% and 400% of the available constant memory. For matrix transformations, however, the available constant memory can only support matrices up to 1000 elements, far smaller than the ones used in this work (see Section VI-A). So, we generated smaller matrices and filled them with data from the ones of Section VI-A.

2) Memory Requirements: By comparing Tables III and V, we observe that using FSTs leads to a sizable increase in the number of states and transitions. For example, in order to support the 8-bit ASCII alphabet, a BPD FST would need about 100 times the number of states and transitions of a effPDT, leading to an increase in the required memory from 1.51KB to 122KB. Similar considerations apply for the other data transformations. Recall that, differently from FSTs, the size of effPDTs is independent of the alphabet size, maximum run-length, etc. In contrast, the size of FSTs increases with these parameters, leading to configurations that do not fit the constant memory. In case of matrix transformations, memory requirements of FST can even exceed the global memory capacity. For example, with 12GB of GPU memory, a Dense-CSR FST and a COO-CSR FST cannot support a square matrix of more than 10000 and 600 rows, respectively. These requirements limit the usability of FSTs for matrix transformation. We note that, while Unified Virtual Memory (UVM) would allow supporting FSTs exceeding the GPU memory capacity, using UVM would further decrease system performance due to page faults overhead and on-the-fly memory transfers.

3) System Throughput: Figure 6 shows the performance of the effPDT and FST GPU engines on experiments conducted using the parameter settings and FST memory requirements listed in Table V. As can be seen, the use of FSTs causes a performance slowdown (over effPDTs) across the board. With small topologies that fit in constant memory, the slowdown ranges from 0.02x (Dense-CSR) to 6x (BPD). With larger topologies exceeding the constant memory capacity, the slowdown is much more significant, ranging from 4x (RLE) to 22x (BPD). In the case of matrix transformations (Dense-CSR and COO-CSR), the FSTs for larger matrices cannot fit into GPU's memory. On average, not counting instances where it is not possible to fit the topology onto GPU memory, we see a 15.7x slowdown when using FSTs over effPDTs.

To conclude, effPDTs are more compact than FSTs, leading to lower memory requirements and more data locality. All these factors affect the throughput positively.

VIII. Related work

In the field of language theory, FSTs and PDTs have been introduced by Elgot (1965) and Evey (1963) [39] to define translation grammars. Since then, various theoretical extensions have been proposed to support different classes of application. For example, visibly pushdown transducers [36] extend PDT with a input-aware stack to support structured alphabet translation; weighted FSTs [37] assign weights to transitions and support speech-to-text applications; symbolic FSTs [38] extend FST transitions with rules over a set of variables to express applications such as image blurring, HTML decoding and malware finger printing. While previous works proposing extensions to transducers primarily explored their theoretical implications and syntactical definition, our effPDT model extends PDTs to allow space and time efficient *implementations* of a broad range of data transformations. So, our proposed extensions are grounded in practical needs.

Recent works have investigated techniques to bring transducer theory to practice. Grathwohl et al. [48] have proposed a nondeterministic FST language and compiler based on the idea of decomposing transducers into two machines: an oracle machine performing disambiguation of the input, and an action machine triggering output actions. Raghothaman et al. [49] have defined a transducer-based data query language. Zhao et al. [50], [51] have introduced methods to accelerate processing of finite state machines through speculative execution. To the best of our knowledge, there is lack of work exploring the hardware acceleration of transducers and their deployment at scale. Our work represents an effort in this direction.

IX. Conclusion and future work

This work has targeted the design of a flexible GPU-accelerated data transformation engine. To this end, we have proposed effPDTs, a computational model that extends PDTs, can express a wide range of data transformations in a space-efficient manner, and is amenable for GPU acceleration. We have showcased our engine on a set of data transformations covering data encoding and decoding, sparse matrix layout transformations, histogram construction and query of structured data. Our evaluation shows significant speedups over custom CPU implementations, and performance on par with, or better than, custom GPU implementations. In addition, we have shown the resource requirements and performance advantages of effPDTs over FSTs.

Future research directions include: (1) extending our effPDT engine to support nondeterministic behavior, enabling the acceleration of compression/decompression (and other) tasks that require back-tracking (e.g., snappy, deflate, and lz4); (2) providing a programming model and compiler for effPDTs; and (3) exploring alternative effPDT implementations.

X. Acknowledgments

This work was supported by National Science Foundation awards CNS-1812727 and CCF-1907863.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, p. 29–43, oct 2003.

[3] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 385–398.

[4] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120.

[5] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 293–307.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 15–28.

[7] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," *IEEE Micro*, vol. 36, no. 3, pp. 54–59, 2016.

[8] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle, "Darpc: Data center rpc," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–13.

[9] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 94–108.

[10] M. F. Khairoutdinov and D. A. Randall, "A cloud resolving model as a cloud parameterization in the ncar community climate system model: Preliminary results," *Geophysical Research Letters*, vol. 28, no. 18, pp. 3617–3620, 2001.

[11] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–11.

[12] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "Csx: An extended compression format for spmv on shared memory systems," *SIGPLAN Not.*, vol. 46, no. 8, p. 247–256, feb 2011.

[13] D. Langr and P. Tvrdík, "Evaluation criteria for sparse matrix storage formats," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 428–440, 2016.

[14] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. New York, NY, USA: Association for Computing Machinery, 2015, p. 339–350.

[15] B.-Y. Su and K. Keutzer, "Clspmv: A cross-platform opencl spmv framework on gpus," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 353–364.

[16] "Intel mkl." [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-mkl-for-dpcpp/top.html

[17] A. Parquet. [Online]. Available: https://parquet.apache.org/

[18] "Gnu scientific library." [Online]. Available: https://www.gnu.org/

[19] S. K. Moon and R. D. Raut, "Hardware-based application of data security system using general modified secured diamond encoding embedding approach for enhancing imperceptibility and authentication," *Multimedia Tools and Applications*, vol. 78, no. 15, pp. 22 045–22 076, Aug 2019.

[20] T. Sugimoto, Y. Nakayama, and T. Komori, "22.2 ch audio encoding/decoding hardware system based on mpeg-4 aac," *IEEE Transactions on Broadcasting*, vol. 63, no. 2, pp. 426–432, 2017.

[21] M. Safieh and J. Freudenberger, "Efficient vlsi architecture for the parallel dictionary lzw data compression algorithm," *IET Circuits, Devices & Systems*, vol. 13, no. 5, pp. 576–583, 2019.

[22] H. Wang, T. Wang, L. Liu, H. Sun, and N. Zheng, "Efficient compression-based line buffer design for image/video processing circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 10, pp. 2423–2433, 2019.

[23] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "Infant: Nfa pattern matching on gpgpu devices," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 5, p. 20–26, oct 2010.

[24] 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," ser. PPoPP '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 129–140.

[25] 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*, ser. CF '13. New York, NY, USA: Association for Computing Machinery, 2013.

[26] R. Sidhu and V. Prasanna, "Fast regular expression matching using fpgas," in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, 2001, pp. 227–238.

[27] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 50–59.

[28] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling pcre to fpga for accelerating snort ids," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ser. ANCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 127–136.

[29] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ser. ISCA '06. USA: IEEE Computer Society, 2006, p. 191–202.

[30] J. V. 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*, 2012, pp. 461–472.

[31] 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*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 533–545.

[32] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[33] E. Sadredini, R. Rahimi, V. Verma, M. Stan, and K. Skadron, "eap: A scalable and efficient in-memory accelerator for automata processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 87–99.

[34] H. Liu, M. A. Ibrahim, O. Kayiran, S. Pai, and A. Jog, "Architectural support for efficient large-scale automata processing," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018, pp. 908–920.

[35] J. Wadden, K. Angstadt, and K. Skadron, "Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 2018, pp. 749–761.

[36] E. Filiot, J.-F. Raskin, P.-A. Reynier, F. Servais, and J.-M. Talbot, "Visibly pushdown transducers," *Journal of Computer and System Sciences*, vol. 97, pp. 147–181, 2018.

[37] M. Mohri, F. Pereira, and M. Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech & Language*, 2002.

[38] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner, "Symbolic finite state transducers: Algorithms and applications," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 137–150.

[39] A. Meduna, *Automata and languages: theory and applications*. Springer, 2000.

[40] M. Mohri, "Finite-state transducers in language and speech processing," *Computational Linguistics*, vol. 23, no. 2, pp. 269–311, 1997. [Online]. Available: https://aclanthology.org/J97-2003

[41] C. H. Papadimitriou, "Complexity theory," *Addison Wesley*, 1994.

[42] "Canterbury cor." [Online]. Available: https://corpus.canterbury.ac.nz/

[43] "Cuda toolkit." [Online]. Available: https://docs.nvidia.com/cuda/

[44] T. A. University, "Suitesparse matrix collection." [Online]. Available: https://sparse.tamu.edu/

[45] "Raleigh open data." [Online]. Available: https://data.raleighnc.gov/

[46] "Pandas." [Online]. Available: https://pandas.pydata.org/

[47] "Open gpu data science." [Online]. Available: https://rapids.ai/

[48] B. B. Grathwohl, F. Henglein, U. T. Rasmussen, K. A. Søholm, and S. P. Tørholm, "Kleenex: Compiling nondeterministic transducers to deterministic streaming transducers," *SIGPLAN Not.*, vol. 51, no. 1, p. 284–297, jan 2016.

[49] R. Alur, D. Fisman, K. Mamouras, M. Raghothaman, and C. Stanford, "Streamable regular transductions," *THEORETICAL COMPUTER SCIENCE*, vol. 807, pp. 15–41, FEB 6 2020.

[50] Z. Zhao and X. Shen, "On-the-fly principled speculation for fsm parallelization," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, p. 619–630, mar 2015.

[51] J. Qiu, X. Sun, A. H. N. Sabet, and Z. Zhao, "Scalable fsm parallelization via path fusion and higher-order speculation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 887–901.