A Transducers-based Programming Framework for Efficient Data Transformation

Tri Nguyen
North Carolina State University
United States of America
tmnguye7@ncsu.edu

Michela Becchi
North Carolina State University
United States of America
mbecchi@ncsu.edu

Abstract

Many data analytics and scientific applications rely on data transformation tasks, such as encoding, decoding, parsing of structured and unstructured data, and conversions between data formats and layouts. Previous work has shown that data transformation can represent a performance bottleneck for data analytics workloads. The transducers computational abstraction can be used to express a wide range of data transformations, and recent efforts have proposed configurable engines implementing various transducer models (from finite state transducers, to pushdown transducers, to extended models). This line of research, however, is still at an early stage. Notably, expressing data transformation using transducers requires a paradigm shift, impacting programmability.

To address this problem, we propose a programming framework to map data transformation tasks onto a variety of transducer models. Our framework includes: (1) a platform agnostic programming language (xPTLang) to code transducer programs using intuitive programming constructs, and (2) a compiler that, given an xPTLang program, generates efficient transducer processing engines for CPU and GPU. Our compiler includes a set of optimizations to improve code efficiency. We demonstrate our framework on a diverse set of data transformation tasks on an Intel CPU and an Nvidia GPU.

ACM Reference Format:
Tri Nguyen and Michela Becchi. 2024. A Transducers-based Programming Framework for Efficient Data Transformation. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3656019.3676891

1 Introduction

Data transformation is one of the core processing steps in many data analytics and scientific applications. For example, Extract-Transform-Load (ETL) workloads require extracting information from a variety of formats, transforming the data, and loading them into a target format. Data transformation tasks performed by these workloads include: data encoding/decoding, data compression and serialization for communication and storage density, data analysis, and query of structured or unstructured data (using popular data formats such as CSV and JSON). In addition, scientific applications operating on matrices often require data conversion between different matrix formats [25], such as compressed sparse row (CSR), compressed sparse column (CSC), and coordinate list (COO) [15, 26, 27, 29, 41]. Performing these data transformations efficiently is crucial to application performance.

Due to rapidly increasing data sizes, data transformation has increasingly become a performance bottleneck for many data analytic applications [24, 36, 40, 46]. At the same time, the use of hardware heterogeneity to maximize performance and achieve energy efficiency has led to the need for portable implementations. To address these issues, optimized CPU and GPU libraries implementing common data transformations have been made available. These include: Parquet [37] and Cub [2] (data encoding and decoding), Pandas [8] and RapidsAI [6] (parsing), and Intel MKL[5] and cuSparse [2] (sparse matrices). While efficient, these libraries only address specific data transformations, and lack generality. As new data transformations are devised, optimized implementations tailored to different platforms are needed, leading to significant programming effort.

Broader applicability to diverse data transformation tasks and portability can be achieved by a programming framework relying on a solid computational abstraction with efficient implementations for diverse hardware platforms. Previous work on transducers processing [22, 23, 43] has shown the capability of the transducer model to express a wide range of data transformations, from natural language processing, to structured data parsing (e.g., XML and HTML), to image reversal, among others. However, the acceleration of the transducers computational model has received only limited consideration. Recent efforts [34, 35] have proposed accelerated processing engines for finite state and pushdown transducers. To this end, these works have proposed compact and efficient transducer models amenable for hardware acceleration, as well as their implementation on a variety of platforms, including CPUs, GPUs and novel accelerators [21]. Their results show that transducers-based implementations can provide performance on par with popular custom libraries running on the same hardware, and in some cases even outperform those libraries.

However, using the transducers abstraction to express data transformation tasks implies a programming paradigm shift. Programmers typically view applications in terms of sequences of algorithmic steps, often implemented through a Von Neumann language with intuitive constructs such as program variables, assignment statements, and control-flow statements. Implementing a data transformation task using a finite state or pushdown transducer requires expressing the computation through a set of states, transitions and possibly stack operations. Existing proposals either lack a high-level programming interface, or provide a basic programming interface

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PACT '24, October 14–16, 2024, Long Beach, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3676891

that, while including different constructs, inherently requires the programmer to think of the computation in terms of states and transitions [34, 35]. This paradigm shift is a major gatekeeper in the wider adoption of transducers in general data processing tasks.

In this work, we aim to address this problem by providing a programming framework for data transformation tasks rooted on the transducers abstraction. The goals of the framework are: *programmability*, *portability*, *efficiency* and *extensibility*. Specifically, the framework aims to: (1) support a wide range of transducer models (from finite state transducers, to pushdown transducers and extensions), (2) provide an intuitive and platform independent programming interface that bypasses the paradigm of states and transitions, (3) generate efficient transducer engines for CPU and GPU offering performance competitive with hand-tune transducers and custom data transformation libraries, and (4) be easily extended to support diverse hardware platforms and transducer models.

In this paper, we make the following contributions:

- xPTLang, a platform agnostic programming language that allows for expressing transducer programs as sequences of steps using common and intuitive programming constructs, such as arithmetic and control-flow statements;
- A compiler that, given an xPTLang program, constructs a transducer, optimizes it and generates efficient CPU and GPU processing engines for it;
- Four compiler optimizations to reduce the transducer's memory footprint and improve control-flow efficiency;
- An evaluation of our framework on a set of data encoding/decoding, data analytics, matrix transformation and structured data query tasks [2, 4–6, 8, 37].

Our experiments show that our implementation achieves an average speedup of 1.6× and 2× over customized library implementations for CPU and GPU, respectively, while outperforming existing transducer processing engines by 1.9× and 2.6× on average across applications and datasets. From the programmability perspective, not only does xPTLang provide a *unified programming interface* for diverse hardware platforms (i.e., CPU and GPU), but it also allows for *compact codes*, hiding from the programmer the implementation details of the transducer engine (e.g., input/output stream handling, memory management) on the targeted platforms.

2 Background and Motivation

Transducers are computational abstractions that map a streaming input into a streaming output based on a transition relationship. As such, they are a natural abstraction to express and implement data transformation applications. Two basic transducer models are *finite state tranducers* (FSTs) and *pushdown transducers* (PDTs). FSTs have a finite number of states and transitions. Transitions are associated one (or more) input and output symbols, the former denoting the symbol(s) triggering the transition, and the latter the output symbol generated when the transition is activated. PDTs extend FSTs with a stack, inherently adding state. While the theoretical PDT model includes an infinite stack, in practice stacks have a finite size, and infinite stacks can be simulated by dynamically allocating stack space as needed. Independent of the specific model, a data transformation expressed through a transducer can be implemented by processing the input stream symbol-by-symbol, and traversing the transducer's states and transitions based on the sequence of symbols read.

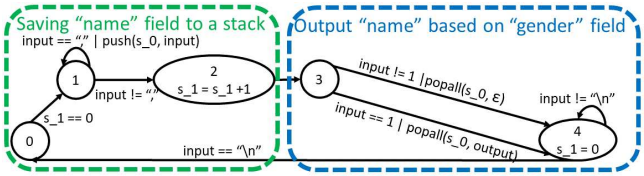


Figure 1: Pushdown transducer that extracts the names of all female individuals from a CSV file containing three columns: *name*, *gender* and *occupation*.

2.1 Formal Definition and Example

Formally, a pushdown transducer (PDT) [32] 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, Σ_O, and Σ_S are the input, output and stack alphabets, respectively;
- $\delta \subseteq Q \times (\Sigma_I \cup \{\epsilon\}) \times (\Sigma_S \cup \{\epsilon\}) \times Q \times (\Sigma_O \cup \{\epsilon\}) \times (\Sigma_S \cup \{\epsilon\})$ is a finite state transition relationship, ϵ being the empty string;
- $s \in Q$ is the start state;
- $F \subseteq Q$ is a set of final states.

A finite state transducer (FST) is similarly defined by excluding stacks from the alphabet and transitions' definition.

Operationally, besides writing to an output stream, a PDT can pop symbols from a stack and push symbols onto it. A PDT transition $r = (q_1, \sigma_{I1}, \sigma_{S1}, q_2, \sigma_{O2}\ \sigma_{S2})$ 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. Transducers can be deterministic or non-deterministic: the former have only one active state, while the latter can have multiple states active at the same time.

Figure 1 shows a PDT extracting the names of all the female individuals from a CSV file containing three columns: *name*, *gender*, and *occupation*. This transducer traverses the CSV file in row-major order using a counter (*s_1*) to iterate through each column. The states inherently record the progress of the transformation, while the transitions check for conditions on stacks and inputs, and generate the output accordingly. States 0-2 and their connecting transitions (i.e., green block) read and save the content of the *name* field into a stack (*s_0*), while states 3 and 4 and their connecting transitions (i.e., the blue block) check whether the *gender* column (column 1) indicates a female (marked as 1) and, when so, they output the name of the individual recorded in stack *s_0*.

2.2 Related Work

Over the past decade, there have been several efforts focused on extending the traditional FST and PDT models to support different classes of applications efficiently. These works range from theoretical contributions to transducer engines deployed on real-world datasets. Table 1 summarizes the characteristics of several notable transducers.

Table 1: Model features and availability of an execution engine implementation for different transducer works.

Model	*Model Features and Extensions*				*Execution Engine*
	Var.	Stacks	Arith.	Multi I/O	
openFST [7]	✓	-	-	-	-
FOMA [3]	✓	-	-	-	-
Thrax [10]	✓	-	-	-	-
SFST [43]	✓	-	✓	-	-
VPT [22]	-	✓	-	-	-
SST [12]	✓	-	-	-	-
DFST+ [35]	✓	-	✓	-	CPU
effPDT [34]	✓	✓	✓	✓	CPU/GPU
SQRE [31]	-	✓	-	-	CPU
xPTLang	✓	✓	✓	✓	CPU/GPU

- **openFST** [7], **FOMA**[3] and **Thrax**[10] are libraries for constructing finite state and weighted transducers from an existing set of rewrite rules or regular expressions.
- **Symbolic Finite State Transducers (SFSTs)** [43] extend FSTs with the concept of registers and arithmetic operations to support HTML decoding and image blurring.
- **Visibly Pushdown Transducers (VPTs)** [22] use an input-aware stack to support nested alphabet translation (for example, for XML parsing).
- **Streaming String Transducers (SSTs)** [12] utilize a set of fixed-length and variable-length registers to perform transformation of key-value pair data.
- **DFST+s** [35] extend FSTs with registers and arithmetic operations to perform data encoding/decoding.
- **StreamQREs (SQRE)** [31] provide a query language for patterned data.
- **effPDTs** [34] extend PDTs with multiple stacks, arithmetic operations and multiple I/O streams to support a wide range of data transformations.

The works on SFSTs, VPTs, and SSTs focus on the theoretical soundness of the proposed models, their decidability and expressiveness. The authors prove that the proposed transducers can support a wide range of applications: from computation intensive tasks such as image reversal, to control-flow heavy tasks such as XML parsing. In all cases, the underlying processing engine is a standard FST or PDT. It is shown that the proposed extended transducers can be converted to standard FSTs and PDTs by performing input/output enumeration. This conversion, however, can lead to state explosion when increasing the alphabet size (for example, SFSTs can get to millions of states on 6-bit alphabets).

More recent work (DFST+s and effPDTs) has focused on putting the transducers' theory to practice by using compact and hardware-friendly transducer models, and implementing high-throughput transducer processing engines on GPU and other hardware accelerators. DFST+s and effPDTs extend FSTs and PDTs, respectively, with arithmetic operations and an addressable memory. They are capable of expressing data transformations that, when using practical alphabets, cannot be feasibly supported by standard transducers on reasonable hardware.

Most of the works discussed above either require users to hand-code the transducer expressing the target data transformation (which can be complex and error-prone), or include a low-level interface that requires users to describe data transformations in terms of states and transitions, limiting programmability and adoption in practice. While lacking an efficient execution engine implementation, openFST, FOMA and Thrax can be used as frontend to our framework for data transformations that can be easily expressed through a set of rewrite rules and can be encoded using traditional models (namely, FSTs and weighted FSTs). Our work aims to provide a transducers-based programming framework with a high-level programming interface for general data transformations, while supporting the model features of the extended transducers proposed in previous work (Table 1).

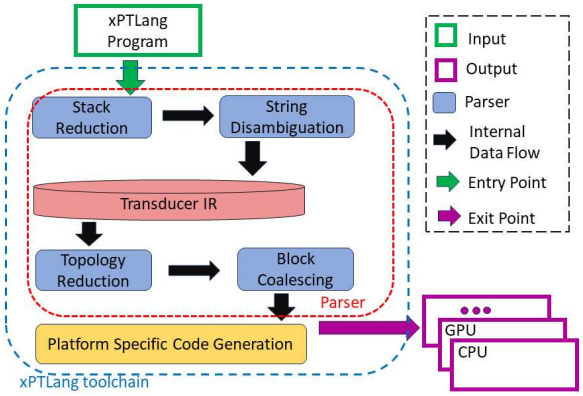


Figure 2: High-level design of the xPTLang framework.

It is worth noting that the idea of accelerating entire classes of applications by focusing on the computational abstraction at their core has been successfully explored in other contexts. For example, there has been a large body of work on the implementation and acceleration of automata processing on GPU [17, 45, 47], FPGA [14, 33, 39] and custom hardware [16, 19, 20, 28, 30, 38]. Recent work [13, 18] has proposed a high level automata description language to map pattern matching programs to Micron's Automata Processor and similar spatial architectures. Since automata are computationally equivalent to regular expressions, these works focus primarily on *search* applications that perform various kinds of pattern matching on textual data. However, they cannot be easily adapted to support data transformation because they do not provide efficient support for dynamic output generation [14, 19, 44].

3 Programming Framework's Design

Figure 2 illustrates the high-level design of the xPTLang programming toolchain. The *parser* takes an xPTLang program, constructs an internal representation of it in the form of an extended transducer (*transducer IR*), and performs various compiler optimizations (*stack reduction*, *string disambiguation*, *topology reduction*, and *block coalescing*) aimed to reduce the transducer's size and improve code efficiency. The *code generator* produces a transducer processing engine, that is, a program implementing the transducer's traversal. The current implementation supports code generation for CPU and GPU. However, the framework can be extended with additional code generators targeting other platforms.

3.1 General Design Decisions

Programming interface: First, the programming language should hide the underlying transducer abstraction from the programmer. Second, it should support the *data streaming* abstraction, which is intrinsically implemented by transducers and allows limiting memory and storage requirements. Third, different data transformation applications process the input at different granularity. For instance, many data encoding algorithms process the input *symbol-by-symbol*, and require a fixed number of read operations (typically one) on each symbol. On the other hand, text parsing/querying applications typically process the data *token-by-token* (where a token is a string of text). To improve programmability, xPTLang should support both styles of input processing.

Choice of the underlying transducer abstraction: Our goal is to support FSTs and PDTs, along with the transducer models discussed in Section 2.2. To this end, we implement an extended PDT model incorporating the features and extensions listed in Table 1, namely: memory in the form of *stacks* (we treat *variables* as single-element stacks), *arithmetic operations* associated to states, and *multiple input/output streams*. We leverage theoretical results from prior work. Specifically, the work on SFSTs [43] and well-nested VPTs [22] proves that the addition of memory and arithmetic operations retains the closure and composition properties of these transducers. A transducer is considered closed under composition if its composition operations (*concatenation*, *union*, *intersection*) produce a transducer of the same class. In practice, given an xPTLang program, this property allows us to construct a transducer incrementally, by first generating small transducers corresponding to single program statements, and then recursively composing those transducers using the three transducer composition operations based on the control flows in the program.

4 Programming Model

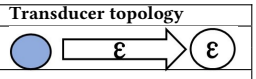
4.1 xPTLang Programming Constructs

xPTLang includes three categories of programming constructs: *actions*, *conditions* and *loops*. These constructs allow programmers to describe the computation in a sequential fashion using a set of unconditional statements, conditional statements and loops. Those statements operate on *stacks*, *input streams* and *output streams*. xPTLang provides constructs to process the input symbol-by-symbol and token-by-token.

4.1.1 Keywords for stacks, input and output streams. We use identifiers to refer to stacks, input and output streams. Keywords *s_ID*, *input_ID*, *output_ID*, where ID is a numeric value, represent stack, input and output identifiers, respectively.

4.1.2 Actions. Actions denote changes in the content of stacks, input streams or output streams. Allowed operations include: (1) arithmetic operations on the element at the top of a stack, (2) basic stack operations such as *push* and *pop*, and (3) advanced stack operations such as *flush*, *popall* and *write*. Table 2 summarizes the available actions as well as the topology of the sub-transducer generated by the compiler for each action (recall that the final transducer is built by incrementally composing smaller transducers, each expressing a portion of the program). The blue coloring is to indicate that, within the sub-transducer, arithmetic operations will be associated to states while stack and output operations will be associated to transitions.

Table 2: Actions: syntax and corresponding sub-transducers.

Category	Action	Transducer topology
Arith. op.	dst = src1 op src2	
Basic stack op.	**push** (dst, src) **pop** (src, dst)	
Adv. stack op. & Output op.	**popall** (src, dst) **flush** (src, dst) **write** (dst, src)	

Arithmetic operations have two source operands, *src1* and *src2*, and one destination operand, *dst*. *src1* can be either a stack or an input stream identifier, *src2* can be a stack identifier, an input stream identifier or a constant value, and *dst* identifies the destination stack. Allowed operators include: addition, subtraction, multiplication, division, bit-wise operations and bit manipulation operations.

Push and pop actions modify the stack identified by their first argument. The second argument indicates the provenance or the destination of the symbol pushed onto the stack or popped from it, respectively, and can be a stack, an input or an output stream identifier. In addition, the programmer can use the epsilon symbol (ϵ) as the destination argument of the *pop* operation to indicate that the symbol retrieved from the stack can be discarded. All accesses to input and output streams cause these streams to be modified.

Advanced stack operations are for enhancing productivity, but can be expressed using a combination of arithmetic and basic stack operations. *popall* and *flush* allow programmers to write the entire content of a stack into a destination (another stack or an output stream) in top-to-bottom or bottom-to-top order, respectively. The *src* operand of the *write* operation can be either a stack or a constant value. In the first case, the write operation allows for copying the element at the top of a stack to a destination (another stack or an output stream) without removing it from the source stack.

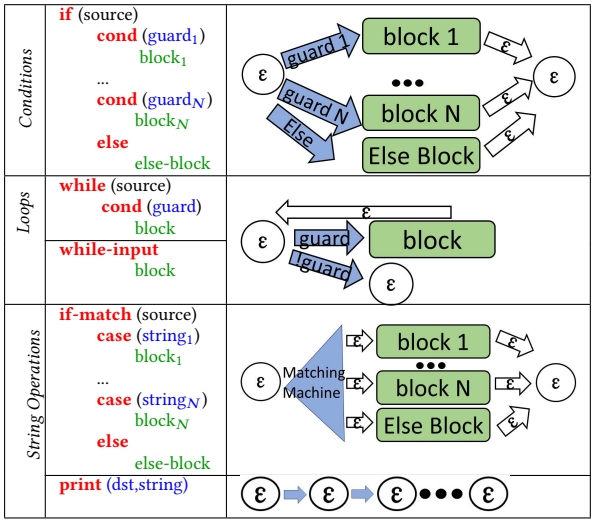
4.1.3 Conditions. Conditions denote conditional changes in the program's execution flow. Table 3 shows the basic structure of a condition statement and the corresponding sub-transducer generated by the compiler. Note that each condition contains nested block of statements ($block_i$). The compiler will generate a sub-transducer for each $block_i$. The topology in Table 3 shows how these sub-transducers are composed in case of a condition statement.

The *if* and *cond* keywords mark the beginning of a conditional block and of an execution guard, respectively. Execution guards are statements that evaluate to either true or false, and have the form ([*arithmetic comparator*][*stack/constant value*]).

Each guard determines whether a block of statements (which can include actions, conditions, loops or string operations) will be executed. The *left-hand-side* of the guard is specified by the *source* in the parent *if*.

4.1.4 Loops. A loop denotes repeated execution of a block of statements. Table 3 shows the basic structure of a loop and the corresponding sub-transducer. Notably, loops introduce backward-directed transitions. The loop body can include actions, conditions, loops and string operations, and will have an associated

Table 3: Loops, conditions, and string operations: syntax and corresponding sub-transducers. Note that $block_i$ can in turn be an action, a condition, a loop or a string operation. The green blocks to the right represent the sub-transducers associated to nested code blocks $block_i$.



sub-transducer. The *while* keyword marks the beginning of a loop structure. The guard operates as in condition statements.

4.1.5 String Operations. String operations allow for defining the input and output processing at a "token" granularity, and are suitable for transformations that require string matching, such as parsing and querying of semi-structured data. xPTLang includes two string processing primitives (listed in Table 3): *if-match* and *print*.

The *if-match* primitive allows the comparison of a *source* (either an input stream or a stack) with a set of predefined strings (denoted by the *case* keyword). The code blocks $block_i$ can include any constructs, except nested *if-match* statements on the same source. The generated sub-transducer contains a "matching machine", which performs string matching against the given set of strings, connected to the sub-transducers corresponding to the $block_i$ code blocks.

The *print* primitive writes a predefined string to an output stream. The generated sub-transducer consists of states connected by non-consuming transitions, each outputting a symbol of the string.

We note that string operations can be implemented as sequences of symbol operations, using the *condition* and *action* constructs. However, doing so will significantly increase the program length and decrease its readability. Finally, these primitives allow expressing data transformations through the enumeration of the accepted input strings and generated outputs, supported by standard FSTs.

4.2 Streaming Behavior

Following the transducer abstraction, the xPTLang language assumes read-only input and write-only output streams. Reading from an input stream consumes an input symbol, while writing to an output stream causes a symbol to be appended to the stream. In both cases, a stream pointer is implicitly incremented. As mentioned above, performing a data transformation using a finite state or a pushdown transducer requires a transducer traversal guided by the symbols in the input stream. Thus, during processing, the input stream is read symbol-by-symbol, and the computation continues as long as there are input symbols still to be processed. In the presence of multiple inputs, the processing is considered completed only when all input streams have become empty. To this end, xPTLang includes a special loop construct called *while-input* (Table 3), which iterates as long as there is a symbol to be processed in any of the input streams. xPTLang hides from the programmer the internal handling of input and output streams (i.e., handling of stream pointers, input/output buffering, and data transfers between host and device). Stream accesses (for example, through *push* and *pop* instructions) implicitly advance the corresponding stream pointer, and, when necessary, cause operations on internal buffers and host-device data transfers.

Listing 1: xPTLang program implementing RLE.

```
1  push(s_0, input_0)                          ▷ s_0: run length symbol
2  write(output_0, s_0)                        ▷ s_1: counter
3  push(s_1, 1)                                ▷ _: current symbol
4      while-input
5          push(s_2, input_0)
6          if-source(s_2)
7              cond (== s_0)
8                  s_1 = s_1 + 1
9                  pop(s_2, ϵ)
10             cond (!= s_0)
11                 pop(s_1, output_0)
12                 pop(s_0, output_0)
13                 pop(s_2, s_0)
14                 push(s_1, 1)
```

4.3 Stack Handling

We recall that arithmetic operations operate solely on the element on the top of each stack accessed, but do not push elements into the destination stack or pop elements from the source stack(s). Loop and condition guards access stacks without modifying their content. To conform with the pushdown transducers abstraction, a stack's depth is modified only by *push*, *pop*, *popall*, and *flush* operations (Table 2). Stack operations involving two stacks modify the depth of the two stacks in opposite ways. For example, *push(s_1, s_2)* causes an element to be inserted into stack *s_1* and an element to be removed from stack *s_2*. On the other hand, *write* operations involving stacks can read or write the element on top of a stack (depending on whether the stack is the *src* or *dst* argument), without adding or removing stack elements.

In terms of implementation, upon declaration, stacks are initialized to empty. The xPTLang's implementation handles internally any memory management operations required to provide a logically infinite stack. Users can define a stack-specific maximum depth (otherwise set to 128 elements by default). Additional buffers are dynamically allocated as needed when the stack's depth exceeds the pre-allocated buffer size.

4.4 Example

Listing 1 shows an xPTLang program that implements run-length encoding (RLE). This encoding scheme compresses the input data by recording every symbol in the input followed by the number

Table 4: Nesting level (NL), peers, children and body list for RLE program. Line numbers are colored to match Listing 1.

Line	NL	Peers	Children	Body List
Root	0	None	1, 2, 3, 4	S_{rs}, L_1,L_2, L_3, L_4, S_{re}
1	1	2, 3, 4	None	S_{1s}, S_{1e}
2	1	1, 3, 4	None	S_{2s}, S_{2e}
3	1	1, 2, 4	None	S_{3s}, S_{3e}
4	1	1, 2, 3	5	S_{4s}, L_5, S_{4e}
5	2	6	None	S_{5s}, S_{5e}
6	2	5	7,10	S_{6s},L_7,L_{10} S_{6e}
7	3	10	8, 9	S_{7s}, L_8, L_9, S_{7e}
8	4	9	None	S_{8s},S_{8e}
9	4	8	None	S_{9s}, S_{9e}
10	3	7	11, 12, 13, 14	S_{10s}, L_{11}, L_{12}, L_{13}, L_{14}, S_{10e}
11	4	12, 13, 14	None	S_{11s}, S_{11e}
12	4	11, 13, 14	None	S_{12s}, S_{12e}
13	4	11, 12, 14	None	S_{13s}, S_{13e}
14	4	11, 12, 13	None	S_{14s}, S_{14e}

of its consecutive occurrences (for example, *aaabbbbbc* → *a3b5c1*). Stack *s_0* stores the symbol being counted, stack *s_1* stores the run-length counter, while stack *s_2* stores the current input symbol. Lines 1-3 initialize the stacks and output the first symbol read. Lines 4 marks the main execution loop. The loop body reads the next symbol (line 5), checks if it matches the symbol being counted (lines 7-10), and increments the counter (line 8) or outputs the run-length and resets the counter (lines 11-14).

5 xPTLang Compiler

5.1 Parser

Given an xPTLang program, the xPTLang parser generates a fully-connected transducer.

5.1.1 Parser algorithm. Recall that, leveraging the transducers' composition property, the parser builds a transducer incrementally. First it constructs a sub-transducer for each statement, and then it connects sub-transducers recursively based on each statement's type and on the program's structure. This allows for nesting of complex constructs. The parser algorithm follows six steps:

Step 1: Create a root state.

Step 2: Calculate the *nesting level* of each statement. We denote a statement's nesting level as the number of conditional or loop structures a program has to go through to get to that statement. For example, a loop body's statement will have nesting level 1 plus the nesting level of the loop. The root state is assigned nesting level 0.

Step 3: For each statement, determine its child and peer statements. Statement A is a child of B if A comes after B in program order and A's nesting level is 1 below B's nesting level. Peer statements have the same nesting level.

Step 4: Traverse the list of statements. Each statement A will be assigned a *body list*, which contains the states associated to A and its children. For each statement A, create a start and an end state (S_{As} and S_{Ae}). If A is an action or print statement, connect S_{As} and S_{Ae} according to the diagrams in Tables 2 and 3. Add the states created to A's body list.

Step 5: Perform recursive composition by traversing the list of statements in reverse order. If a statement A has children, insert the children's *body lists* into A's *body list*, and connect these states according to their statement type (see Table 3). For example, if A is a condition statement with two children C and D, and C and D's *body lists* are (S_{Cs}, S_{Ce}) and (S_{Ds}, S_{De}), respectively, performing recursive composition will cause A's *body list* to become: (S_{As}, S_{Cs}, S_{Ce}, S_{Ds}, S_{De}, S_{Ae}) and four transitions to be created: $S_{As} \rightarrow S_{Cs}$, $S_{As} \rightarrow S_{Ds}$, $S_{Ce} \rightarrow S_{Ae}$, and $S_{De} \rightarrow S_{Ae}$.

Step 6: Perform recursive composition on the root state. Add transitions to connect all the children's *body lists* in series (body 1 end state to body 2 start state, body 2's end state to body 3 start state, etc). This step ensures a fully connected transducer.

5.1.2 Example. Table 4 shows the nesting level, children, peers and body list of each statement in the RLE code example (Listing 1) at the end of the execution of the parser's algorithm. For readability, we call L_X the body list of state X, and do not enumerate the children's body lists within their parent's body list. Figure 3 illustrates the recursive composition process (steps 5 and 6 of the parser's algorithm). Specifically, Figures 3a-c show the result of applying recursive composition on the statements at lines 7, 10, 6 and 4, while Figure 3d shows the fully connected transducer resulting from applying recursive composition on the root state.

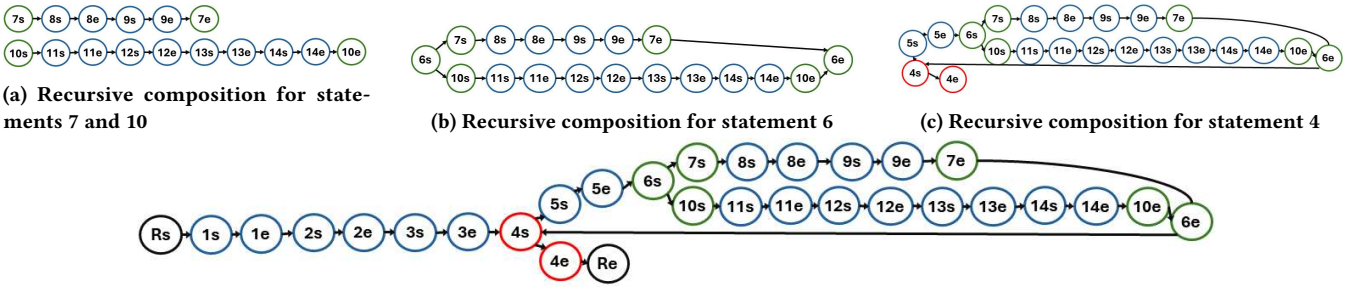
5.2 Code Generation

Given the transducer IR generated by the parser, the code generator generates the corresponding traversal engine's implementation. Two implementation approaches are possible: *memory-based* and *code-based* engine. In a memory-based engine, the transducer's topology (states, transitions and related information) is stored in memory using a predefined layout. The traversal code is transducer independent, and the code generator creates and populates the required memory data structures. In a code-based engine, the transducer topology is embedded in the traversal code, which is transducer specific. Here, we take the second approach. We focus on code-based implementations for performance considerations. By storing the transducer topology (states and transitions) in memory, memory-based engines incur additional memory accesses and require instructions to decode the topology. While our programming interface is generic, our code generator currently supports deterministic transducers (the parser raises a warning if it cannot eliminate sources of non-determinism from the transducer, for example through string disambiguation).

5.2.1 Code generation algorithm. At a high level, we map each state to a block of code that contains the state's logic (i.e., the work executed when the state is active), and we implement transitions as conditional statements that redirect the program's execution flow among these code blocks. The code generation algorithm operates in 2 steps:

Step 1: Data structure allocation: Allocate the required data structures, including: stacks, variables, and per-thread context information (i.e., active state and input/output streams' pointers). The required stacks and variables are determined in the stack reduction optimization step (Section 6.4). On GPU, variables and context information are stored in registers, while stacks are stored in shared memory (and offloaded to global memory as needed).

Step 2: Generation of traversal loop: Generate the transducer traversal loop, which iterates as long as there is an active state and an input symbol to be processed. The loop body contains an



(a) Recursive composition for statements 7 and 10

(b) Recursive composition for statement 6

(c) Recursive composition for statement 4

(d) Fully connected transducer

Figure 3: Operation of steps 5 and 6 of the parser's algorithm on the RLE code in Listing 1. For the sake of space and readability, we show only the relevant pieces of the topology.

if-block for each state. The body of the block, which is executed when the state is active, contains the the state's actions, its outgoing transitions, and the required updates to the input/output streams' and stacks' pointers. For transitions, there are two options. (1) If the outgoing transition depends on the content of an input or a stack, we generate an *if-block* with the guard being the transition's condition. The output and stack updates triggered by the transition are converted into statements inside that transition's if-block. The active state is then set to the transition's destination state. (2) If the outgoing transition is executed unconditionally, the statements implementing the output and stack updates triggered by the transition and the active state's update are added to the if-block of its source state. The example in Listing 2 shows a code snippet illustrating the salient aspects of the transducer traversal code.

Various parallelization approaches are possible: *chunk-based parallelization* (where pre- and post-processing primitives are used to break input and output in chunks processed in parallel) [34], *input-based parallelization* (where different inputs are processed in parallel) [35], and *transducer-level parallelization* (where different transducers are processed concurrently). Here, we take the first approach, but the code generator can be extended to support the other schemes. The generated code is parallelized using POSIX threads on CPU, and CUDA on GPU.

6 Compiler Optimizations

In this section, we describe four optimizations to reduce the transducer topology and improve execution efficiency.

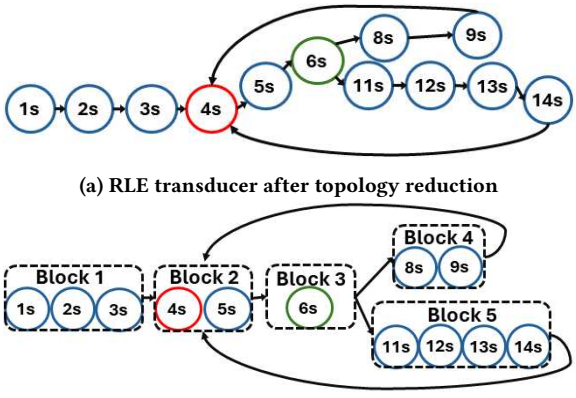
Listing 2: Transducer traversal pseudocode snippet.

```
1 void sample_transducer_kernel(....){
2     ...                                          ▷ Execution loop
3     while((state < state_no)&&(input)){
4         if(state==1){                            ▷ Unconditional tx
5             s_0[top_0++]=input[current_i++];
6             state=2;
7         }
8         else if(state==2)                        ▷ Conditional tx
9             if(stack_compare(s_0,s_2)) state=4;
10            else state=3;
11        else if (state==3){                      ▷ State arithmatic ops
12            s_1[top_1]=s_1[top_1]+1;
13            output[current_o++]=s_1[top_1++];
14 }    }    }
```

6.1 Topology Reduction

Our parser is designed to easily incorporate changes in the xPTLang language. Since each statement is logically enclosed in a pair of start and end states, additional programming constructs can be easily incorporated in xPTLang without complicating the recursive transducer composition process. However, this method generates a large number of unnecessary states and epsilon transitions, limiting code efficiency.

To address this problem, we introduce a *topology reduction* compiler pass. We call "empty" states the states that don't have an action associated to them. If an empty state s_X has an outgoing epsilon transition to state s_Y, s_X and its outgoing transition are eliminated, and its incoming transitions are connected to state s_Y. If all incoming transitions to a state s_X are epsilon transitions, state s_X and its incoming transitions are eliminated, and its outgoing transitions are replicated and connected to each state previously transitioning to s_X. Note that, in deterministic transducers, a state with an outgoing epsilon transition cannot have additional outgoing transitions. Figure 4a shows the result of applying topology reduction on the fully connected transducer in Figure 3d.



(a) RLE transducer after topology reduction

(b) RLE transducer after block coalescing

Figure 4: Topology reduction and block coalescing applied to the RLE transducer in Figure 3d.

6.2 Block Coalescing

As mentioned in Section 5.2, the xPTLang compiler converts transducer's states into blocks of code, and transitions into if-statements that redirect the program execution from one code blocks to another. In the transducer abstraction, each transition can generate or consume only one symbol, and each state can contain only one arithmetic operation. As a result, reading multiple variables and multi-step calculations require a series of states and transitions. During code generation, this inflates the number of conditional statements and creates conditional checks in unconditional execution paths. Block coalescing aims to eliminate unnecessary conditional statements by bypassing the reference transducer model and merging together states that are connected by unconditional transitions. Block coalescing is performed in three steps.

Step 1: States are partitioned in two lists: L1 and L2 . L1 contains the starting points of a block, namely: (1) the initial state, (2) states with multiple incoming transitions, and (3) states with multiple outgoing transitions and their directly connected states. L2 contains the remaining states.

Step 2: Blocks of states are generated by traversing L1. If a state in L1 has multiple outgoing transitions, a one-state block is created. If it has one outgoing transition, a new block is created and the transducer topology is traversed until another state in L1 is reached. All states along the traversed path are then removed from their respective list and added to the newly created block.

Step 3: The resulting blocks are connected by the existing transitions between the end state of a block and the start state of another.

Figure 4b shows the result of applying block coalescing on the transducer of Figure 4a. This transformation coalesces the 14-state transducer into a 5-block function.

6.3 String Disambiguation

The *if-match* construct enables matching multiple strings in parallel. The partial or full overlap among the strings can lead to a non-deterministic transducer (i.e., a transducer with multiple active states). To handle this scenario, the xPTLang compiler performs string disambiguation and constructs a transducer that performs multiple string matching in a deterministic manner. This optimization consists of four steps:

Step 1: The compiler identifies any user-introduced ambiguity in the matching conditions. These are instances where a match is fully contained within another match (for example, "apple" and "apples"). In these cases, the compiler implements a greedy policy and accepts only the shortest of the overlapping strings.

Step 2: The compiler constructs a tree-like string matching sub-transducer by merging outgoing transitions on the same symbol and the corresponding target states. This process is equivalent to subset construction for automata minimization.

Step 3: If a state of the string-matching sub-transducer from step 2 has an outgoing transition accepting a symbol c, the compiler connects that state to the sub-transducer implementing the else-block via a transition corresponding to the mismatch of c.

Step 4: Note that a leaf state in the matching sub-transducer corresponds to the matching of a string. Accordingly, each leaf state is connected to the sub-transducer implementing the block of code to be executed upon a match.

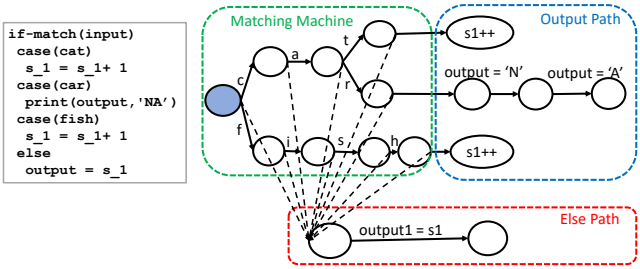


Figure 5: String disambiguation example: xPTLang code snippet and corresponding transducer.

Figure 5 illustrates string disambiguation on a code snippet performing the match of three strings: 'cat', 'car' and 'fish'. The code increments a counter if the input matches 'cat' or 'fish', writes 'NA' to the output if the input matches 'car', and outputs the value of the counter if the input does not match any of the three strings. The transducer constructed by the xPTLang compiler is shown next to the code. The 'matching machine' sub-transducer (green box) is constructed by enumerating the three strings and merging the outgoing transitions on the same character (e.g., *c* and *a*) and their target states. Each state of the 'matching machine' is then connected to the sub-transducer implementing the 'else' path (red block) via a mismatch transition. Finally, each accepting state of the 'matching machine' is connected to the sub-transducer to be executed upon a match, depicted as part of the 'output path' (blue block). The constructed transducer follows the topology shown in Table 3 and the recursive composition process described in Section 5.1.

6.4 Stack Reduction

Building on the PDT abstraction, the xPTLang language represents variables as stacks. However, implementing scalar variables using stacks in the transducer processing code is inefficient, since it increases the memory requirements and adds the overhead of managing stack operations (e.g., stack pointer updates). The *stack reduction* optimization aims to identify all the stacks whose depth never increases beyond 1 and replace them with scalar variables. We adopt a conservative approach, and use static analysis to identify stacks that can safely be reduced to variables. For each stack, we trace all *push* and *pop* actions performed on it, and verify that the sequence of stack manipulation actions does not cause the stack depth to increase beyond 1. For example, a push and a pop action to the same stack within a basic block are safe. This rule can be generalized to statements with the same nesting level. If one of the statements is a condition, we check that the opposite stack action is performed in each branch of the if-statement. When modified with a *pushall* action, a stack cannot be reduced to a variable, since the number of symbols pushed on it is generally not known at compile time. When replacing a stack with a variable, we eliminate all the associated push and pop operations, and replace them with simple assignment statements.

7 Experimental Setup

Benchmarks. We select 15 data transformation workloads from six classes of applications: (1) *data encoding/decoding*, (2) *sparse matrix*

Table 5: Benchmarks and baseline library implementations.

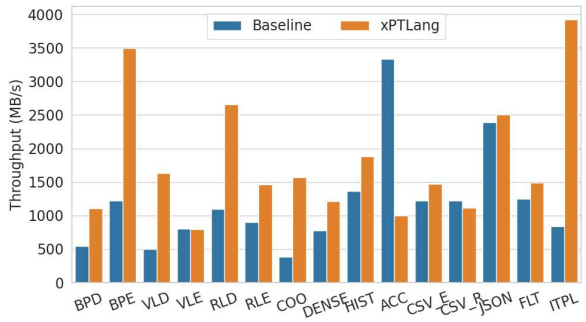
Application	Input Dataset	CPU	GPU
Data Enc/Dec	Cantebery Corpus, Artifical Corpus [1]	Parquet [37]	Cub [2]
Matrix Transform	Texas A&M Sparse Matrix [42]	Intel MKL [5]	cuSparse [2]
Statistics	RDU Accident and Crime Report [9]	GSL Hist [4]	Cub [2]
Data Query	NY City Water Consumption [11]	Pandas [8]	Rapids [6]
Prediction			
Filtering			

transformation, (3) *data statistics*, (4) *data querying*, (5) *data prediction* and (6) *data filtering*. For data encoding/decoding, we consider bit-packing (*BPE/BPD*), run-length (*RLE/RLD*) and variable-length (*VLE/VLD*) encoding/decoding. For sparse matrix transformation, we select the transformation from the COO to the CSR format (*COO*), and the transformation from dense to CSR format (*Dense*). For data statistics, we select the generation of histograms (*HIST*) and accumulations (*ACC*). For data querying, we use the JSON and CSV formats. The queries aim to extract a subset of a CSV and JSON file based on a specified user condition. We conduct our experiments on raw, unedited and dictionary-encoded data (*JSON_R CSV_R* and *CSV_E*, respectively). For data filtering, we perform range filtering (*RF*) to curate a subset of CSV and JSON data based on range data of member fields. For data prediction, we use data interpolation (*ITPL*) to fill out missing data point using linear interpolation.

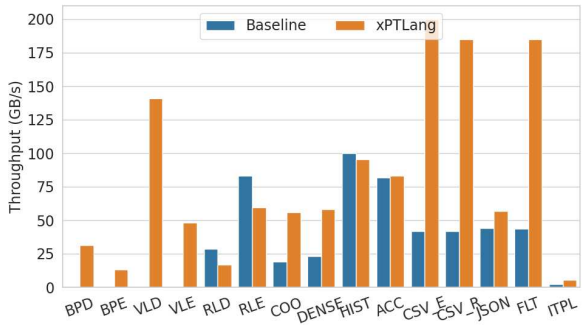
Baselines. We compare xPTLang against custom library implementations of the benchmarks above, as well as two existing *memory-based* transducer processing engines for CPU and GPU (DFST+ [35] and effPDT [34], see Section 2). The considered application-specific libraries are: Parquet[37], GNU Scientific Library [4], Pandas [8], NVIDIA cub, cuSparse, and Rapids [2]. We time the execution of the data transformation kernels in these libraries.

Input datasets. We use textual data from the Canterbury Corpus and Artificial Corpus Datasets [1], with file sizes ranging from 4KB to 2MB, sparse matrix data from the Texas A&M Open Source sparse matrix collection [42] (*g7jac160, xenon1*). For query, prediction and statistics data, we use the Raleigh Sustainable Project [9] (*longitude* and *latitude*) and the Crash Location dataset (*FeetFromRoad*) and New York city's water consumption [11]. Table 5 summarizes the datasets used in our experiments, as well as the CPU and GPU library implementations that we use as baselines. The input streams used in our experiments are constructed by replicating the content of the datasets of Table 5 until reaching a stream size of 1GB on CPU and of 10GB on GPU. Setup time and data transfer time for the baselines are not included in the timing - we measure only the execution time of the main data transformation kernels/functions.

System configuration. We run our experiments on a system equipped with two Intel Xeon processors running at 2.2GHz, each with ten physical cores and a total 25MB of cache. The system is also equipped with an NVIDIA A30 GPU, with 24GB global memory, 64KB constant memory and 48KB shared memory per streaming multi-processor (SM). The GPU has 56 SMs operating at a maximum clock rate of 1.44GHz. In addition, our system has 130GB RAM and a 1TB SSD. We use Ubuntu 18.04, gcc 7.5 and CUDA toolkit 12.1.



(a) CPU



(b) GPU

Figure 6: Throughput of xPTLang on CPU and GPU against custom library implementations (*baseline*) in Table 5.

Parallelization. We recall that the xPTLang compiler generates C++ code using POSIX threads for CPU and CUDA code for GPU. In addition, it performs chunk-based parallelization and uses the pre- and post-processing primitives from [34]. The custom CPU library implementations used as baseline are single-threaded. We parallelize CPU execution by launching multiple instances of these libraries through POSIX threads. We spawn 20 threads for all CPU implementations, and assign each thread 1GB of the input data. On GPU, the thread-block configuration is set to utilize all the available SMs.

8 Experimental Evaluation

8.1 Performance Results

CPU throughput. Figure 6a compares the throughput of the CPU code generated by the xPTLang compiler with the CPU baseline library implementations listed in Table 5. On average, xPTLang code performs 66% better than the customized baseline implementations, offering an average throughput of 1.8 GB/s vs. the 1.1 GB/s baseline throughput. In particular, xPTLang performs best on data encoding/decoding (1.8 GB/s vs. 612 MB/s), sparse matrix transformation (1.4 GB/s vs. 576 MB/s) and data prediction workloads (3.9 GB/s vs. 833 MB/s). xPTLang performs on par with baseline implementations on data query (1.6 GB/s) and data filtering (1.2GB/s) workloads, while underperforming baseline code on data statistics workloads (1.4 GB/s to 2.4 GB/s). Sparse matrix operations are irregular workloads. The streaming nature of the transducer model and xPTLang's code-based transducer processing engine allow avoiding scattered

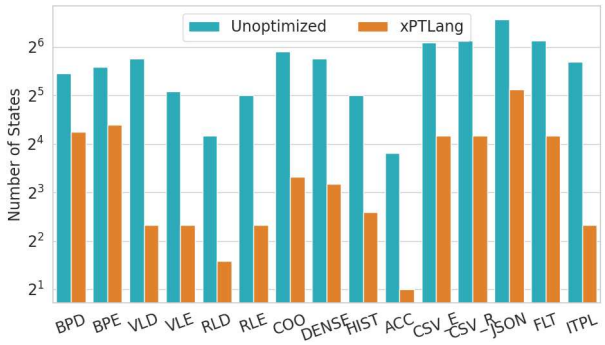


Figure 7: Number of states generated by the xPTLang framework (without and with optimizations).

memory accesses to the input matrices. The data predictions workload (ITPL) performs linear interpolation on missing values. In the Pandas library, this is implemented as a single input to single output transformation, while our xPTLang implementation uses a second input stream to quickly iterate through the dataset and identify missing values. Once a missing value is found, the main stream performs interpolation at that location. This approach allows us to overlap data accesses and interpolation as well as bypassing unnecessary data. On the other hand, the simplicity and regularity of data statistics workloads (histogram generation and data accumulation) allows for very efficient custom library implementations outperforming the xPTLang framework.

For the benchmarks using the Rapids library (CSV, CSV_RAW, JSON and FLT), xPTLang outperforms the Rapids implementations because the transducer abstraction allows streaming the input and encoding the execution context in local variables, while Rapids loads the entire dataset in memory and accesses it using irregular patterns. We observe that xPTLang and Rapids issue the same number of load instructions (about 150 millions). However, on average, xPTLang reports a 99% L1 cache hit rate while Rapids reports a 28% L1 cache hit rate. In addition, transducers allow performing parallel matching of multiple strings efficiently in a regex-like manner, while Rapids performs serial, string-by-string comparisons. We observe an average 2.5× reduction in the number of executed branch instructions when using xPTLang (1.2 and 3.1 billions branch instructions when using xPTLang and Rapids, respectively).

GPU throughput. Figure 6b compares the throughput of the GPU code generated by the xPTLang compiler with the GPU baseline library implementations listed in Table 5. We note that GPU implementations are available only for a subset of the considered workloads. On average, xPTLang performs about 2× better than baseline, offering an average throughput of 91GB/s vs. the 46 GB/s baseline throughput. In particular, xPTLang performs best on data querying/filtering (147GB/s vs. 42GB/s), sparse matrix transformation (57GB/s vs. 21GB/s) and data prediction workloads (5.6GB/s vs. 2.2GB/s). It performs on par with baseline on data statistics (90GB/s), while underperforming the baseline on data encoding (38GB/s vs. 56GB/s). Data querying and filtering workloads require grammar parsing, which involves non GPU-friendly computations such as nested evaluations and long string matching. xPTLang can leverage stacks (stored in shared memory) to encode complex grammars, and the xPTLang compiler allows efficient string matching through string disambiguation (see Section 6.3). The sparse matrix transformation, data prediction and data statistics results follow considerations made for CPU. RLE (data encoding) is also suited for GPU acceleration through custom optimizations, allowing the baseline to outperform xPTLang.

Table 6: Code size (in LOC) of xPTLang programs (*xPTL*) and generated CPU and GPU codes. Σ_{LOC} = total LOC; P = stream partitioning; M = I/O handling, data transfers and allocations, and kernel setup; K_g = core data transformation kernels generated by the xPTLang compiler; K_c = hand-tuned kernels from the libraries in Table 5 (if open-source).

	xPTL	CPU code				GPU code			
		Σ_{LOC}	P	M	K_g/K_c	Σ_{LOC}	P	M	K_g/K_c
BPD	22	123	30	64	29/14	180	87	64	29/-
BPE	22	128	30	78	20/15	182	87	75	20/-
VLD	15	147	30	73	44/20	197	87	66	44/-
VLE	25	133	30	69	34/23	183	87	62	34/-
RLD	7	119	30	69	20/20	169	87	62	20/51
RLE	14	134	30	70	34/20	184	87	63	34/52
COO	26	209	51	83	75/-	251	108	68	75/-
DENSE	28	175	30	84	61/-	217	87	69	61/-
HIST	15	140	30	71	39/87	191	87	65	39/61
ACC	6	116	30	74	12/6	162	87	63	12/24
CSV_E	33	211	30	70	111/-	262	87	64	111/-
CSV_R	33	235	51	71	113/-	290	108	69	113/-
JSON	33	432	51	74	307/-	486	108	71	307/-
FLT	5	116	30	74	12/87	162	87	63	22/16
ITPL	25	185	51	83	51/-	231	108	72	51/-

Comparison with memory-based engines. Compared to memory-based transducer processing engine implementations (DFST+ [35] and effPDT [34]), xPTLang performs on average about 80% better (1.7 GB/s vs. 900MB/sec) on CPU and 1.5× better (131GB/sec vs. 50GB/sec) on GPU. Embedding state and transition information in code has two benefits. First, it avoids frequent memory accesses to retrieve the transducer's topological information. On GPU, profiling data show an average 3.15× reduction in the number of load instructions issued (0.17 vs. 0.6 billions). Second, the optimizations discussed in Section 6 allow for compact and efficient code. Our profiling data show that xPTLang kernels take about 400 bytes, which can comfortably fit in the 32KB instruction cache on CPU and 64KB constant cache on GPU.

8.2 Programmability & Compilation

Transducer size. Figure 7 reports the number of states of the transducers generated without and with optimizations enabled (*Unoptimized* and xPTLang, respectively). We note that, thanks to the optimizations it performs, the compiler is able to automatically generate compact transducers, on average 20% smaller than hand-coded ones from previous work [34], resulting in compact programs.

Code Size. Table 6 shows the code size of the considered xPTLang programs (*xPTL*) and of the corresponding CPU and GPU implementations generated by our compiler. We break down the LOC of the generated implementations (Σ_{LOC}) into: (i) code performing stream partitioning (P), (ii) miscellaneous code handling I/O streams, memory allocations, data transfers and kernel setup (M),

and (iii) core data transformation kernels (K_g). Not all the custom libraries of Table 5 are open source; when code is available, we also report the code size of the kernels included in those libraries (K_c). K_c indicates the size of the custom data transformation kernels alone (not including stream and memory handling or setup code).

We recall that xPTLang code is platform agnostic. Moreover, the xPTLang language allows users to focus solely on encoding the data transformation task on *abstract data streams* using *simple programming constructs*. The handling of streams (including I/O operations, buffering and data partitioning), stacks, variables, memory allocations, data transfers, and platform-specific details are hidden from the programmer, and the required code is automatically generated by the compiler. This results in compact code: xPTLang can express the considered transformations in 5 to 33 LOC, while the generated implementations take from 116 to 432 LOC on CPU and from 162 to 486 LOC on GPU - a 3× to 30× increase in code size. Notably, xPTLang programs are comparable in size or smaller than the generated data transformation kernels alone (K_g). Data transformations that can be expressed using string operations (e.g., data query) enjoy significantly smaller xPTLang codes. Finally, for open-source libraries, xPTLang programs have sizes comparable to (and in several cases smaller than) custom kernels specialized for a single data transformation (K_c). Note, however, that using these custom kernels requires the additional data partitioning, data allocation and I/O handling code. To conclude, building on the transducers computational abstraction, the xPTLang framework allows platform-independent optimizations, and the generation of efficient code for multiple platforms, while hiding implementation details associated with the transducer abstraction from the programmer.

Extended PDTs vs. FSTs. Existing works focusing on theoretical aspects of transducer processing [12, 22, 43] convert extended PDTs and FSTs into standard FSTs via input/output enumeration, enabling the use of an FST traversal engine. However, especially on large alphabets, FST generation can incur state explosion. Here, we evaluate the effect of FST conversion on compilation time and processing throughout. Figure 8 shows results obtained by using the xPTLang framework to generate an FST engine and the default extended PDT engine (xPDT) for RLE. We generate FSTs for alphabet sizes ranging from 4 to 64 symbols (2- to 6- bit alphabets), leading to FST ranging from 228 to 55,000 transitions. We plot compilation time (dashed lines, right axis) and throughput (solid lines, left axis) when using xPTDs on CPU and GPU, and FSTs on GPU.

For small FSTs (up to $\tilde{5}$000 transitions), *compilation time* remains limited. The more complex FST topology as the alphabet size increases results in larger code with more divergent branches, negatively affecting the processing *throughput*. As a result, FST's performance degrades as the alphabet gets larger (from 16GB/s down to 0.4GB/s). Even with a trivial 2-bit alphabet, FST achieves only 27% of xPDT's throughput, while with a 6-bit alphabet FST run on GPU performs worse than xPDT run on CPU. On the other hand, xPDTs allow constant compilation time (limited to less than a minute) and throughput (59GB/s and 1.4GB/s on GPU and CPU, respectively). The use of stacks and arithmetic operations limits the size of the xPDT and makes its topology independent of the alphabet size. This results in compact codes and reduced branch divergence.

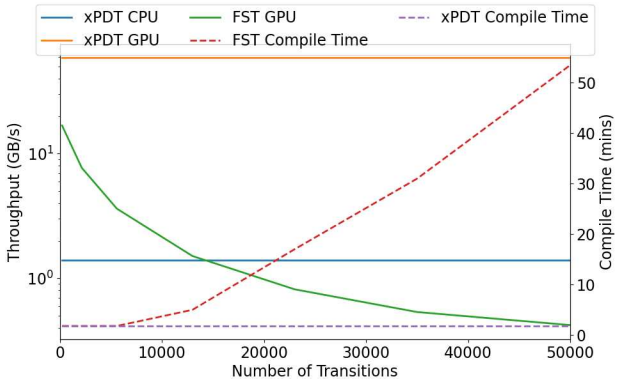


Figure 8: Throughtput (GB/s) and compilation time (sec) of FSTs and xPDTs when using xPTLang framework.

9 Conclusion

We have proposed a portable programming framework for data transformation tasks based on the transducers abstraction. Our framework includes a *programming language* (xPTLang) to express transducer programs in a serial fashion using intuitive programming constructs, and a *compiler* that transforms xPTLang programs into transducer processing engines for CPU and GPU. Our framework includes a set of optimizations that operate on the transducer's topology and aim to improve code efficiency. Our experiments, performed on a diverse set of 15 data transformation workloads, show performance and programmability advantages over custom library implementations and recently proposed transducer-based processing engines. The performance advantages over custom implementations vary across applications. Future research directions include extending our framework to support other platforms and covering nondeterministic behavior, thus enabling data transformations requiring back-tracking (e.g., snappy, deflate, and lz4).

Acknowledgments

This work was supported by National Science Foundation award CCF-1907863.

References

[1] 2023. Canterbury Cor. https://corpus.canterbury.ac.nz/.
[2] 2023. CUDA Toolkit. https://docs.nvidia.com/cuda/.
[3] 2023. Foma. https://fomafst.github.io/.
[4] 2023. GNU Scientific Library. https://www.gnu.org/.
[5] 2023. Intel MKL. https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-mkl-for-dpcpp/top.html.
[6] 2023. Open GPU Data Science. https://rapids.ai/.
[7] 2023. openFST. https://www.openfst.org/twiki/bin/view/FST/WebHome.
[8] 2023. Pandas. https://pandas.pydata.org. https://pandas.pydata.org/
[9] 2023. Raleigh Open Data. https://data.raleighnc.gov/.
[10] 2023. thrax. https://www.openfst.org/twiki/bin/view/GRM/ThraxQuickTour.
[11] 2023. US Government Data. https://data.gov/.
[12] Rajeev Alur and Jyotirmoy V. Deshmukh. 2011. Nondeterministic Streaming String Transducers. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II* (Zurich, Switzerland) *(ICALP'11)*. Springer-Verlag, Berlin, Heidelberg, 1–20.
[13] Kevin Angstadt, Westley Weimer, and Kevin Skadron. 2016. RAPID Programming of Pattern-Recognition Processors *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 593–605. https://doi.org/10.1145/2872362.2872393
[14] Michela Becchi and Patrick Crowley. 2008. Efficient Regular Expression Evaluation: Theory to Practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (San Jose, California) *(ANCS '08)*. Association for Computing Machinery, New York, NY, USA, 50–59.

https://doi.org/10.1145/1477942.1477950

[15] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11. https://doi.org/10.1145/1654059.1654078

[16] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. 2006. A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, USA, 191–202. https://doi.org/10.1109/ISCA.2006.7

[17] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. INFAnt: NFA Pattern Matching on GPGPU Devices. *SIGCOMM Comput. Commun. Rev.* 40, 5 (oct 2010), 20–26. https://doi.org/10.1145/1880153.1880157

[18] Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. 2019. Debugging Support for Pattern-Matching Languages and Accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1073–1086. https://doi.org/10.1145/3297858.3304066

[19] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098. https://doi.org/10.1109/TPDS.2014.8

[20] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast Support for Unstructured Data Processing: The Unified Automata Processor. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 533–545. https://doi.org/10.1145/2830772.2830809

[21] Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. 2017. UDP: a programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, Hillery C. Hunter, Jaime Moreno, Joel S. Emer, and Daniel Sánchez (Eds.). ACM, 55–68.

[22] Emmanuel Filiot, Jean-François Raskin, Pierre-Alain Reynier, Frédéric Servais, and Jean-Marc Talbot. 2018. Visibly pushdown transducers. *J. Comput. System Sci.* 97 (2018), 147–181. https://doi.org/10.1016/j.jcss.2018.05.002

[23] Bjørn Bugge Grathwohl, Fritz Henglein, Ulrik Terp Rasmussen, Kristoffer Aalund Søholm, and Sebastian Paaske Tørholm. 2016. Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers. *SIGPLAN Not.* 51, 1 (jan 2016), 284–297.

[24] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2016. Profiling a Warehouse-Scale Computer. *IEEE Micro* 36, 3 (2016), 54–59. https://doi.org/10.1109/MM.2016.38

[25] Marat F. Khairoutdinov and David A. Randall. 2001. A cloud resolving model as a cloud parameterization in the NCAR Community Climate System Model: Preliminary results. *Geophysical Research Letters* 28, 18 (2001), 3617–3620. https://doi.org/10.1029/2001GL013552 arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2001GL013552

[26] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. CSX: An Extended Compression Format for Spmv on Shared Memory Systems. *SIGPLAN Not.* 46, 8 (feb 2011), 247–256. https://doi.org/10.1145/2038037.1941587

[27] Daniel Langr and Pavel Tvrdík. 2016. Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Transactions on Parallel and Distributed Systems* 27, 2 (2016), 428–440. https://doi.org/10.1109/TPDS.2015.2401575

[28] Hongyuan Liu, Mohamed Assem Ibrahim, Onur Kayiran, Sreepathi Pai, and Adwait Jog. 2018. 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, 908–920. https://doi.org/10.1109/MICRO.2018.00078

[29] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA). Association for Computing Machinery, New York, NY, USA, 339–350. https://doi.org/10.1145/2751205.2751209

[30] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. 2012. Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 461–472. https://doi.org/10.1109/MICRO.2012.49

[31] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. 2017. StreamQRE: Modular Specification and Efficient Evaluation of Quantitative Queries over Streaming Data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 693–708. https://doi.org/10.1145/3062341.3062369

[32] Alexander Meduna. 2000. *Automata and languages: theory and applications*. Springer.

[33] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. 2007. Compiling PCRE to FPGA for Accelerating SNORT IDS. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems* (Orlando, Florida, USA) *(ANCS '07)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/1323548.1323571

[34] Tri Nguyen and Michela Becchi. 2022. A GPU-accelerated Data Transformation Framework Rooted in Pushdown Transducers. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 215–225. https://doi.org/10.1109/HiPC56025.2022.00038

[35] Marziyeh Nourian, Tri Nguyen, Andrew A. Chien, and Michela Becchi. 2022. Data Transformation Acceleration using Deterministic Finite-State Transducers. In *2022 IEEE International Conference on Big Data (Big Data)*. 141–150. https://doi.org/10.1109/BigData55660.2022.10020756

[36] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 293–307.

[37] Apache Parquet. [n. d.]. https://parquet.apache.org/.

[38] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. 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, 87–99. https://doi.org/10.1145/3352460.3358324

[39] R. Sidhu and V.K. Prasanna. 2001. Fast Regular Expression Matching Using FPGAs. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. 227–238.

[40] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. 2014. DaRPC: Data Center RPC. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2670994

[41] Bor-Yiing Su and Kurt Keutzer. 2012. ClSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing* (San Servolo Island, Venice, Italy) *(ICS '12)*. Association for Computing Machinery, New York, NY, USA, 353–364. https://doi.org/10.1145/2304576.2304624

[42] Texas A&M University. [n. d.]. SuiteSparse Matrix Collection. https://sparse.tamu.edu/

[43] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjorner. 2012. Symbolic Finite State Transducers: Algorithms and Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 137–150. https://doi.org/10.1145/2103656.2103674

[44] Jack Wadden, Kevin Angstadt, and Kevin Skadron. 2018. 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, 749–761. https://doi.org/10.1109/HPCA.2018.00069

[45] Xiaodong Yu and Michela Becchi. 2013. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *Proceedings of the ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '13)*. Association for Computing Machinery, New York, NY, USA, Article 18, 10 pages. https://doi.org/10.1145/2482767.2482791

[46] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28.

[47] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. GPU-Based NFA Implementation for Memory Efficient High Speed Regular Expression Matching *(PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/2145816.2145833