PDTgcomp: Compilation Framework for Data Transformation Kernels on GPU

Tri Nguyen
NC State University.
Raleigh, USA
tmnguye7@ncsu.edu

Michela Becchi
NC State University.
Raleigh, USA
mbecchi@ncsu.edu

***Abstract*—Data transformation tasks - such as encoding, decoding, parsing, and conversion between common data formats - are at the core of many data analytics, data processing and scientific applications. This has led to the development of custom software libraries and hardware implementations targeting popular data transformations. By accelerating specific transformations, however, these solutions suffer from lack of generality. On the other hand, a generic and programmable data processing engine might support a wide range of data transformations, but do so at the cost of reduced performance compared to custom, algorithm-specific solutions. In this work, we aim to bridge this gap between generality and performance. To this end, we provide a compilation framework that transparently converts data transformation tasks expressed using pushdown transducers into efficient GPU code.**

I. Introduction

Data transformation tasks are often a performance bottleneck for data analytics applications across a variety of domains, from scientific to business realms [1]–[3]. For example, extract-transform-load (ETL) workloads rely on data transformations such as parsing and data query, data encoding and decoding, conversion between commonly used data formats, and data analysis. Further, applications relying on sparse matrices and graphs require transformations between data layouts providing different trade-offs between storage requirements and computation performance.

Past work has proposed custom hardware and software implementations targeting specific data transformations, such as matrix conversions [4], data encoding and decoding [5], parsing [6], among others [7]. These solutions, however, lack generality and flexibility.

A more general approach is to identify a computational abstraction at the core of these tasks, and provide a hardware or software acceleration of that abstraction. For example, pushdown transducers (PDTs) can express a variety of data transformations, including the ones listed above. When a transformation is expressed through a PDT, it is realized by a PDT traversal guided by the content of the input text. In this work, we provide a code generation framework that, given a PDT specification, generates an efficient GPU traversal kernel. The result is a generic data transformation framework that provides better performance than algorithmic-specific implementations within popular libraries, while supporting a wide range of data transformations.

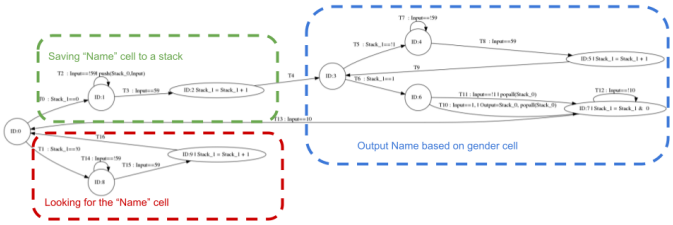

Fig. 1: PDT that extracts the names of all female individuals from a CSV file containing three columns: *name*, *gender* and *occupation*. The *popall* instruction extracts the whole stack's content.

II. Pushdown Transducers

A pushdown transducer (PDT) is a computational abstraction describing a transformation between an input and an output stream. A PDT expresses a transformation as a set of *states*, *transitions*, and *stack operations*. Stacks can be used to temporarily store input or to dynamically generate output. Transitions carry relationships between states and guard the read/write interactions involving input streams, output streams and stacks. In our considered PDT abstraction, states, which track the progress of the computation, may contain arithmetic operations that apply to the top element of one or more stacks.

Figure 1 shows a PDT extracting the names of all the female individuals from a CSV file containing three columns: *name*, *gender*, and *occupation*. The states inherently record the progress of the transformation, while the transitions check for condition on stacks and inputs, and generate the output accordingly. States 8 and 9 and their connecting transitions (highlighted by the red block) allow the PDT to look for the *name* column. States 1 and 2 and their connecting transitions (indicated by the green block) save the content of the *name* fields into a stack, while states 3-7 and their connecting transitions (indicated by the blue block) check whether the *gender* column indicates a female and, when so, they output the name of the individual from the corresponding stack. States 2, 5, 7 and 9 update a stack value storing the column counter.

III. Compiler toolchain

A. Code Generation

We implement a compiler toolchain that performs static analysis of the PDT's states and transitions and generates a GPU kernel realizing the PDT traversal. The code generation proceeds as follows. (1) We convert the list of states into an

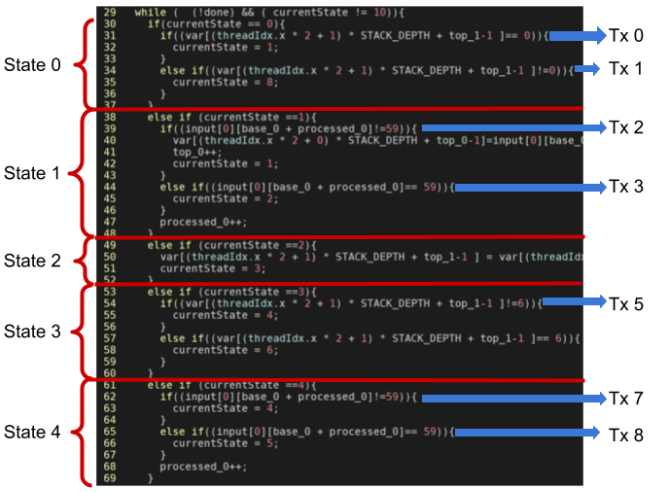

Fig. 2: Snapshot of generated traversal code for PDT in Figure 1

if-else block in which each if-condition guards the content of a state. (2) For each state: (a) we generate the instructions implementing the associated arithmetic operations (if any), (b) we convert the list of outgoing transitions into an *if-else* block, with an if-statement per transition. The if-conditions and statements are generated according to the appropriate input, output and stack read/write operation. To minimize memory access time, stacks are allocated in shared memory and all remaining context information (i.e., I/O pointers, active state indicator) is stored in the register file.

Figure 2 shows the generated code corresponding to the green and blue blocks of the PDT in Figure 1. The kernel consists of a *while* loop with an *if-else* structure, each block corresponding to a state and its outgoing transitions. The outgoing transitions information is used to construct the inner *if-else* blocks (indicated by the blue arrows), which activate different state-blocks in the next iteration of the central while loop. For example, transitions T0 and T1 outgoing from state 0 correspond to the *if-else* blocks of lines 31 and 34, which are responsible for the activation of states 1 and 8, respectively. While not shown in the pseudocode, the traversal is parallelized by breaking the input into chunks and later aggregating the output chunks through predefined kernels.

B. Optimizations

To increase performance, we apply two optimizations.

State Merging combines states that are connected by non-consuming (epsilon) transitions or transitions that are taken unconditionally. Examples include states that perform multi-step arithmetic or sequences of I/O operations. State merging reduces the overall number of states and, as a result, the size of the main *if-else* block.

Loop Unrolling. We observe that PDTs often exhibit cyclical behavior involving: (1) consuming an input symbol, (2) performing one or more arithmetic operations, and (3) outputting the operation's result. This pattern leaves little room for optimization, especially if the PDT is small (in terms of number of states). Unrolling this loop structure allows our compiler to perform more aggressive optimizations, such as reordering of read and write I/O operations.

Fig. 3: Speedup of PDTgcomp over custom CPU libraries

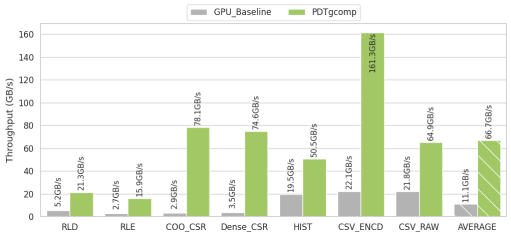

Fig. 4: Throughput of PDTgcomp and custom GPU libraries in GB/s

IV. Preliminary Results

To evaluate the performance of our framework (*PDTgcomp*), we compare the throughput of our generated code against popular CPU and GPU libraries providing custom implementations of 11 transformations across 4 classes of applications: data encoding/decoding, matrix transformation, structured data querying and histogram construction.

We conducted our experiments on a system equipped with two Intel Xeon E5-2630 processors running at 2.2GHz. The system is also equipped with an NVIDIA TITAN XP GPU, with 12GB global memory and 98KB shared memory per streaming multi-processor (SM). The GPU has 30 SMs operating at a maximum clock rate of 1.58GHz.

Figure 3 shows an average speedup of 82x over custom CPU implementations: data encodings from Apache Parquet [5], matrix transformations from Intel MKL [4], CSV querying from Pandas [8] and histogram construction from GSL [6]. The most significant speedups are reported on matrix transformations (150x), CSV querying (92x) and histogram construction (53x). A subset of our benchmarks are available on GPU with kernels provided in Nvidia's Thrust, Cub, cuSparse [9] and the RapidAI library [10]. As can be seen in Figure 4, our framework reports an average 6x speedup over custom GPU kernels. The highest throughputs are reported on CSV querying (112GB/s) and matrix transformation (76GB/s).

V. Conclusion

In summary, we proposed a compiler toolchain that generates efficient GPU code for data transformations expressed using a pushdown transducer abstraction. Our preliminary results, reported on 11 popular data transformations from 4 application classes, show that our generated kernels can outperform (in some cases significantly) custom implementations within popular CPU and GPU libraries. We were able to achieve this result without compromising on the PDT's generality, suggesting that pushdown transducers are a good computational abstraction for these classes of applications. In the future, we aim to incorporate in our toolchain more optimizations to further improve performance.

VI. ACKNOWLEDGEMENT

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 *Proc. of OSDI'04*.
[2] S. Ghemawat and et al., "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5.
[3] K. Ousterhout and et al., "Making sense of performance in data analytics frameworks," in *Proc. of NSDI'15*.
[4] "Intel mkl." [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-mkl-for-dpcpp/top.html
[5] "Apache parquet." [Online]. Available: https://parquet.apache.org/
[6] "Gnu scientific library." [Online]. Available: https://www.gnu.org/
[7] T. Sugimoto and et al, "22.2 ch audio encoding/decoding hardware system based on mpeg-4 aac," *IEEE Transactions on Broadcasting'17*.
[8] "Pandas." [Online]. Available: https://pandas.pydata.org/
[9] "Cuda toolkit." [Online]. Available: https://docs.nvidia.com/cuda/
[10] "Open gpu data science." [Online]. Available: https://rapids.ai/