A Framework for Fast and Fair Evaluation of Automata Processing Hardware

Xiaodong Yu, Kaixi Hou, Hao Wang, Wu-chun Feng
Department of Computer Science, Virginia Tech
Email: {xdyu, kaixihou, hwang121, wfeng}@vt.edu

Abstract—Programming Micron’s Automata Processor (AP) requires expertise in both automata theory and the AP architecture, as programmers have to manually manipulate state transition elements (STEs) and their transitions with a low-level Automata Network Markup Language (ANML). When the required STEs of an application exceed the hardware capacity, multiple reconfigurations are needed. However, most previous AP-based designs limit the dataset size to fit into a single AP board and simply neglect the costly overhead of reconfiguration. This results in unfair performance comparisons between the AP and other processors.

To address this issue, we propose a framework for the fast and fair evaluation of AP devices. Our framework provides a hierarchical approach that automatically generates automata for large datasets through user-defined paradigms and allows the use of cascadable macros to achieve highly optimized reconfigurations. We highlight the importance of counting the configuration time in the overall AP performance, which in turn, can provide better insight into identifying essential hardware features, specifically for large-scale problem sizes. Our framework shows that the AP can achieve up to 461x overall speedup fairly compared to CPU counterparts.

I. INTRODUCTION

The Automata Processor (AP) [2], introduced by Micron for non-deterministic finite automata (NFA) simulations, can perform parallel automata processing within memory arrays by leveraging memory cells to store trigger symbols and simulate NFA state transitions. Previous AP-related designs report thousands-of-fold speedups over their CPU counterparts. However, these remarkable speedups are based on limited-size datasets within a single AP board capacity, thus treating the compilation and configuration overhead as a one-time cost and excluding it from performance evaluation. We argue that such computation-only comparisons become unfair when the dataset size exceeds a single AP board capacity, thus requiring multi-round reconfigurations. The AP software development kit (SDK) provides pre-compiled macros to reduce the reconfiguration cost. However, each macro only works for a fixed-shape automaton; any automaton structural change forces the construction to start from scratch, which is a complicated and error-prone process since it requires expertise in both automata theory and AP architecture.

The major contributions of our work include the following:

- We provide a framework for users to fully and easily explore AP device capacity and conduct fair comparisons to counterpart hardware. It includes a hierarchical approach to automatically generate AP automata and cascadable macros to ultimately minimize the reconfiguration cost.
- We evaluate our framework using real-world datasets and conduct an end-to-end performance comparison (i.e., configuration+computation) between the AP and CPU.

II. AUTOMATA PROCESSORS

The AP consists of three programmable components: state transition elements (STEs), counters, and boolean gates. STEs simulate NFA states with trigger symbols. Connections between STEs simulate NFA state transitions. Developers can define their own AP automata using ANML, a XML-based language, by describing the layout of STEs and transitions. Then, an AP toolchain can parse ANML automata and compile it to binary images. AP compilation is a time-consuming process due to the complex place-&-route calculations. To mitigate the overhead of compilation, the AP SDK supports the pre-compiling of automata to be macros to enable automata reuse. At runtime, after the AP board loads a binary image, it starts the searching against input streams by processing one character per clock cycle.

III. FRAMEWORK DESIGN

A. Approximate Pattern Matching (APM) on AP

APM finds strings that match a pattern with a limited number of errors (e.g., insertion, deletion, and substitution). The Levenshtein distance is one of the most common distance types that allow all three kinds of APM errors. The Levenshtein automata can be manually constructed in the AP recognizable format shown in Fig. 1.

Fig. 1: Optimized STE and transition layout for Levenshtein automata for the pattern “object” allowing up to two errors.
B. Paradigm-based AP Automata Construction

Manually transforming Levenshtein automata to the ANML-based format requires advanced knowledge of both automata and AP architecture as well as tedious programming effort. Hence, we introduce a hierarchical approach to automatically construct automata from paradigms.

**Paradigms and Building Blocks:** APM has three types of errors: insertion (I), deletion (D), and substitution (S). These errors, along with match (M), can be treated as the paradigms of any APM problem and represented in an AP recognizable format (Fig. 2a). For the Levenshtein automata, a building block including two STEs and four types of transitions is shown in Fig. 2b. The STE with the asterisk is the design alternative to support multiple outgoing transitions with different character sets.

**Block Matrix:** With the building block, once the length of desired pattern \(n\) and the maximum number of errors \(m\) are given, building an AP automaton can be implemented by duplicating the building blocks and organizing them into a \((m+1) \times (n-m)\) block matrix. The \(m\) rows correspond to the number of errors allowed (row 0 for the exact match), and the \((n-m)\) columns correspond to the pattern length.

**From Building Blocks to Automata:** We provide an algorithm that accepts paradigm types, pattern length, and maximum number of errors. It then automatically fabricates complex AP automata. It creates the building blocks, builds up the automaton with the block matrix determined by maximum error \(m\) and target pattern length \(n\), and finally sets all STE labels. Note that this process is independent with specific APM applications, in the sense that it can work for any types of distance using paradigm combinations shown in Fig. 2a.

C. Cascadable Macros Design

Although the pre-compiling technique can build macros to reduce the compilation time, it is restricted to same automata structure reuse. In our framework, we propose cascadable macros to support extended reuse of macros. We connect one or more macro instances to compose a larger and different AP automaton through our carefully-designed interconnection algorithm. With the cascадable macros, the reconfiguration overhead can be further significantly reduced, since only the connections between instances need to be placed-&-routed.

IV. Evaluation

We evaluate the framework with two real-world datasets: the BLAST dataset ("Bio") and the NHTSA information retrieval dataset ("IR"). The overall STEs demand of either dataset exceeds the capacity of a single AP board and require dozens of reconfiguration rounds. We compare the performance of our framework to other two AP automata construction approaches: Functional Units (FU) [4] and basic ANML APIs (ANML), as well as CPU implementation PatMaN. Fig. 3 shows our framework can achieve 2x to 461x (runtime+compiling time) speedup over CPU PatMaN and up to 33.1x and 14.8x speedup over ANML APIs and FU, respectively.

V. Related Work

Significant effort has been invested to map automata on parallel architectures including CPUs [7], [6], GPUs [8], [5]. Previous studies have revealed AP can improve performance for many applications, e.g., bioinformatics [4]. Recently, Angstadt et al. [1] propose RAPID, a high-level programming model for AP. However, it may underperform a pre-compiling strategy. Nourian et al. [3] consider fair comparisons between AP and other hardware but only leverage the classical macros.

VI. Conclusions

In this work, we provide a framework allowing users to easily conduct fair end-to-end performance comparison between AP and its counterparts, especially for large-scale problem sizes leading to high reconfiguration costs.

Acknowledgements

This work was supported in part by NSF I/UCRC IIP-1266245 via NSF CHREC and the SEECS Center via the Institute for Critical Technology and Applied Science (ICTAS).

References