Entity Resolution Acceleration using Micron's Automata Processor

Chunkun Bo[1], Ke Wang[1], Jeffrey J. Fox[2], and Kevin Skadron[1]

[1]Department of Computer Science
[2]Department of Materials Science and Engineering
University of Virginia
Charlottesville, VA, 22903 USA
{*chunkun, kewang, jjf5x, skadron*}*@virginia.edu*

Abstract

Entity Resolution (ER), the process of finding identical entities across different databases, is critical to many information integration applications. As sizes of databases explode in the big-data era, it becomes computationally expensive to recognize identical entities for all possible records with variations allowed. Profiling results show that approximate matching is the primary bottleneck. Micron's Automata Processor (AP), an efficient and scalable semiconductor architecture for parallel automata processing, provides a new opportunity for hardware acceleration for ER. We propose an AP-accelerated ER solution, which accelerates the performance bottleneck of fuzzy matching for similar but potentially inexactly-matched names, and use a real-world application to illustrate its effectiveness. Results show 121x to 4200x speedups for matching one record, with better accuracy (7.6% more correct pairs and 39% less generalized merge distance cost) over the existing CPU method.

1 Introduction

Entity Resolution (ER), also known as Record Linkage, Duplication Reduction or Purging/Merging problems, refers to finding records which store the same entity within a single database or across different databases. ER is an important kernel of a large number of information integration applications. For example, the Social Networks and Archival Context (SNAC) collects records from databases all over the world to provide an integrated platform to search historical collections [1]. In such applications, the records of the same person may be stored with slight differences, because documents may come from different sources, with different naming conventions, transliteration conventions, etc. SNAC needs to find the records referring to the same entity with different representations and merge these records. The intuitive method is to compare all possible pair records and check whether a pair represents the same entity.

Determining whether two records represent the same entity is usually computationally expensive ($O(N^2)$). This is even worse in the context of big data, because one needs to compare a huge number of records. Prior work has proposed different algorithms and computation models to improve the performance [2] [3] [4] [5]. However, the performance is still unsatisfying [6].

Micron's Automata Processor is an efficient and scalable semiconductor architecture for parallel automata processing [7]. It is a hardware implementation of non-deterministic finite automata (NFA) and is capable of matching a large number of complex patterns in parallel. Therefore, we propose a hardware acceleration solution to ER using the AP. This paper focuses on solutions for string-based ER.

To illustrate how the AP can accelerate ER, we present a framework and performance evaluation of a real-world ER application. The problem is to identify and combine the records with the same identity stored in SNAC database.

In summary, we make the following contributions.

1. We propose a novel AP-based hardware acceleration framework to solve Entity Resolution.

2. We present an automata design which can process string-based ER, e.g. fuzzy name matching.

3. We compare the proposed methods with state-of-the-art technology (Apache Lucene), and it achieves both higher performance (434x speedup on average) and better accuracy (7.6% more correct pairs and 39% less GMD cost).

2 Related Work

Many methods have been proposed to solve ER. One is a domain-independent algorithm for detecting approximately duplicate database records [3]. They first compute minimum edit-distance to recognize pairs of possible duplicate records, and then a use union/find algorithm to keep track of duplicate records incrementally. Another method sorts the records and checks

whether the neighboring records are the same [2]. For approximate duplicates, some researchers define a window size and a threshold of similarity, so that they can find records similar enough to satisfy application requirements. Apache Lucene is a high-performance search engine and it uses a similar method [8]. The difference lies in that Lucene calculates the score of a document based on the query, and sorts documents instead of every individual record. However, we are not aware of any implementations of these algorithms using accelerators. Our AP-based approach implements a version of the Hamming distance based method [9].

As databases become much larger, some researchers have suggested using blocking methods to improve the performance [4]. This allows them to solve a smaller database, but one needs to have some pre-known information to divide the data and devote extra effort to deal with records comparison across different blocks.

Some novel architectures are proposed to accelerate pattern matching. Fang et. al. propose the Generalized Pattern Matching micro-engine, a heterogeneous architecture to accelerate FSM-based applications [10]. They translate regular expressions to DFA tables, store DFA tables in local memory and expose DFA parallelism by a vector instruction interfaces so that they can reduce instruction counts. Kolb et. al. propose Dedoop, a cloud-based infrastructure to speedup ER [5]. They gain 80x speedup by using 100 Amazon EC2 computation nodes. In contrast to these approaches, the AP is a memory-derived architecture which can directly implement automata efficiently and process a large number of more complex patterns in parallel in a single compute node.

3 Automata Processor

The Automata Processor is an efficient and scalable semiconductor architecture for parallel automata processing. It uses a non-Von-Neumann architecture, which can directly implement non-deterministic finite automata in hardware, to match complex regular expressions. The AP can also match other types of automata that cannot be conveniently expressed as regular expressions.

3.1 AP Functional Elements

The AP consists of three different types of functional elements that can be used to build automata: State Transition Element (STE), Counters and Boolean elements [7].

Each STE can be configured to match a set of any 8-bit symbols and activate a set of successor STEs connected to it when it finds a match. The STEs can be configured as *start*, *all-input* and *report*, so that they can read symbols from input or report when a match is found.

Counters and Boolean elements are designed to work with STEs to increase the space efficiency of automata implementations and to extend computational capabilities beyond NFAs.

3.2 Speed and Capacity

Micron's current generation D480 chip is built on 45nm-equivalent technology running at an input symbol (8-bit) rate of 133 MHz. The D480 chip has two half-cores and each half-core has 96 blocks. Each block has 256 STEs, 4 counters and 12 Boolean elements [11]. In total, one D480 chip has 49,152 STEs, 2,304 Boolean elements, and 768 counter elements. Each AP board can have up to 32 AP chips.

3.3 Programming and Reconfiguration

Automata Network Markup Language (ANML) is an XML language for describing the composition of automata networks. Micron also provides a graphical user interface tool called the AP Workbench for quick automaton design and debugging. A "macro" is a container of automata for encapsulating a given functionality, similar to a function in common programming languages. Micron's AP SDK also provides C and Python interfaces to build automata, create input streams, parse output and manage computational tasks on the AP board. Furthermore, the symbols that an STE matches can be reconfigured. The replacement time is around 0.24 millisecond for one block.

4 Real-world ER Application

The term *entity* describes the real-world object and *resolution* is used because ER is a decision process to resolve the question. Entities are described by their characteristics and *identity attributes* are the ones that distinguish them from other entities. *String-based* ER means that the *identity attributes* are strings.

In this paper, we use a real-world ER problem in SNAC to illustrate how the proposed AP-accelerated method works. When building the SNAC platform, the same person's name may not always be consistent from one record to the next because of typos, mis-spellings, different versions of abbreviation, etc. These differences lead to three major problems. 1) One may miss some correct results when querying a particular user; 2) multiple entries for one entity wastes storage space; 3) the duplicated items will also slow down the speed of searching because the same entities are compared many times. Therefore, SNAC needs to identify and combine records, which is a typical ER problem. In the following section, we refer it as the Name Matching problem, since SNAC uses people's names as identity attributes. But it is important to point out that the AP-accelerated method is not limited to Name Matching. This design can be ported to other string-based ER applications with small modifications.

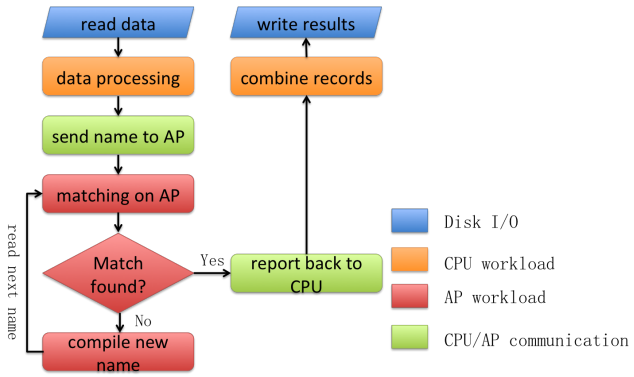

Figure 1: Workflow of Name Matching.

5 Name Matching using the AP

5.1 Workflow

The workflow of the Name Matching problem using the AP is shown in Figure 1. The CPU first extracts the names from the original document in a pre-processing step. The name-only file is then streamed into the AP. The AP stores the names in the database. If the AP finds a match of the name, it will report back to the CPU; if not, the AP continues to compare the next name. Based on the reporting STE id, the CPU can tell which name has found a match and combine these records.

5.2 Name Formats

One name is usually composed of several sub names, like family name, middle name and first name. In this paper, we only consider family name and first name, because this is sufficient to evaluate the suitabilities of the AP-accelerated solution. There are various formats of sub names and it is challenging to extract all possible format. The complexity of finding all formats is the same as solving ER, because it turns into an ER problem if we need to check if all the formats are unique. Therefore, we only consider the most common formats.

We choose a subset from the whole database randomly as a basis to extract a representative set of formats for first name and family name respectively, because they may have different formats. Middle names are less important for correct resolution, and we find that they can usually be ignored. For simplicity in this initial investigation, we simply omit any such individuals from our subset. Table 1 shows four different variants of one family name. Some family names can not be represented by these formats, e.g. "Colan Lulah Johnston Durr" has more than three parts in it. If this is the case, we treat it as a failure (not match). Refinement of these rare special cases is left for future work.

Various formats of first name are shown in Table 2. Similar to family names, we choose relatively common formats. After analyzing these formats, we find that although some of the formats are not exactly the same

Formats	Example
Abc (basic)	Aachen
Abc Bcd	Abad Santillan
Abc Bcd Cde	Abascal Yea Sousa
Abc II	Abdulhamid II

Table 1: Family name formats.

with each other, they can be processed using the same matching structure. For example, row 2, 4, 8, 10 and 13 can use the same structure, thus simplifying the automata design.

Formats	Example
Bcd (basic)	John
Bcd X.	Michael K.
Bcd Cde	Waino Waldemar
Bcd X. (Bcd Xyz)	Chuck L. (Chuck Lehman)
B. X.	P. S. P short for Paolo
B. X. (Bcd Xyz)	C. G. (Charles Greeley)
B. X. (Bcd X.)	C. G. (Charles G.)
Bcd Cde (Xyz)	Anne Jane (Gore)
Bc. Xyz (Bcde Xyz)	Jo. Hale (Joseph Hale)
Bcd, Cde	Scaisbrooke, Langhorne
Bc	Don (short for Donald)
Bcd O. X. (Bcd Opq Xyz)	Kat S.C. (Kat Sara Cabot)
Bcd (Bcd X.)	Herbert (Herbert E.)
Bcd Cde Def Efg	Beats Moss Ella Campbell

Table 2: First name formats.

5.3 Automata for Last/First name

Figure 2 shows the exact-matching automaton design to recognize family names. The first row represents the *Abc* part in Table 1; the second row represents the *Bcd* part; and the third row represents *Cde*. The first few STEs in each row store the name characters to be matched and the subsequent '+'s in are used to occupy the remaining positions, so that names with different lengths can share the same structure. This allows reconfiguring more quickly, by only updating symbols, while reusing the same structure. The '$' represents space in original input and the '#' sign represents Roman numerals. If the automaton reads a space in the input file, it will activate the next row and check the next part in the name formats. The ',' STE is configured as a reporting STE. There are 11 STEs in the first two rows, since the longest name length in the sample is 11. The last row uses 5 STEs to save hardware resources, because if it matches the first two rows and the first few characters in the last row, it is highly possible that it is a match. All the four family name formats use the same structure, because it takes less time to reconfigure the symbols matched by STEs than to compile a new structure on the AP.

This simple design will lead to some false positives, because it aims to support arbitrary string lengths; all of the STEs beyond the second STE in a given row are

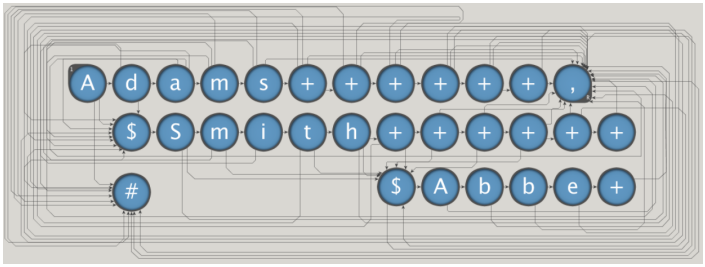

Figure 2: Automata design for family name (exact match for "*Adams Smith Abbe*")

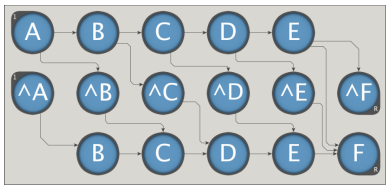

Figure 3: Structure of fuzzy macro (match sequence *ABCDEF* with Hamming distance = 0, 1)

connected to the reporting STE. For example, if the automaton in Figure 1 reads *Ada*, it reports a match; but it is not the name we want. False positives are typically acceptable, but we still need to check the first name. The chance that we get a false positive for both family name and first name is small. Another problem we need to address is the connectivity of the reporting STE, as there are more (26) edges than the AP hardware allows connected to the reporting STE. We solve this by separating the reporting STE into several STEs.

A fuzzy macro refers to the macro executing a fuzzy match. One example is shown in Figure 3. It is used to match sequence the *ABCDEF* and reports when the Hamming distance is ≤ 1. This structure is also used in [12]. Column i corresponds to the ith symbol in the sequence. The STEs in the odd rows activate on symbols in the target name and the even rows activate when there are mismatches. The Hamming distance can be extended up to k with $(2k+1)$ rows . All macros in this paper adopt this structure, with different sequence lengths. However, the macro structures are not limited to Hamming distance. E.g. we have macro designs for general edit distance.

To support different representations of the same name, such fuzzy matching is required. The fuzzy-matching (Figure 4) automata is similar to the exact match automata. Three rows are used to match the three corresponding parts in family name formats. The major difference is that we use the fuzzy macro in Figure 3 to support fuzzy match.

One problem of this design is that if the AP stores a shorter form first, it will produce false negatives. For example, if AP stores *J.* first, it will not report a match when it reads *Janet*, the full form of *J.*. This problem causes most of the inaccuracy in our ER outcome

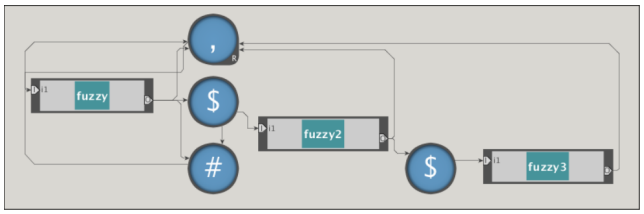

Figure 4: Automata design for family name (fuzzy macro allows Hamming distance = 0, 1)

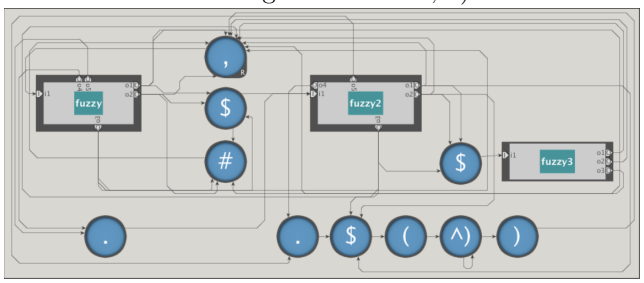

Figure 5: Automata design for first name (fuzzy macro allows Hamming distance = 0, 1)

(Section 6.3). In this paper, we consider '.' symbol as a character within a string. However, abbreviations such as *J.* often are meant to indicate *J* followed by 0 or more of any character. This is an area for future work.

The design for the first name (Figure 5) is similar to family name design. We use an extra row to support the '.' and parenthesis, which do not exist in family-name formats.

5.4 Hybrid Version

We use a hybrid version (Figure 6) in the experiments (Section 6). We match the first name and the family name using one automaton, to conserve STEs. The hypothesis is that if the family name is a match, we can compare fewer characters within the first name. Therefore, we connect the ',' STE in family name, which separates the family name and the first name, to the start STE of the first name. Furthermore, the hybrid technique allows us to use two fuzzy macros instead of three for both family name and first name. An STE pointing to itself is used to accept all the remaining characters. This helps to further reduce STEs consumed.

5.5 Cost of Porting

From the above discussion, we can see that one can do either an exact match or a fuzzy match. For a fuzzy match, we can build macros that allows different degrees of fuzziness. For example, we can extend the macro in Figure 3 to support different lengths or longer Hamming distance. We can also pre-build some macro structures to calculate similarities, like Hamming distance or edit distance. Different records can share the specific structure. Although the method is not a universal method, it can be ported in many string-based ER applications with small modifications.

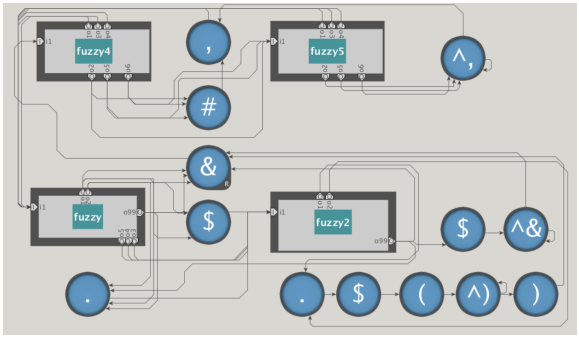

Figure 6: Automata for the whole name ('&' is the delimiter of different names.)

6 Evaluation

6.1 Experiment Setup

All the experiments are executed on a server with 16 AMD Opteron 4386 Cores (3100MHz). We use an AP simulator to derive the execution time for the AP until the real hardware is available. The hardware details are presented in Section 3.2.

In the following experiments, we use Apache Lucene on a single core as the representative CPU method. Lucene is widely used and supports many advanced query types, like proximity queries and range queries, which enables us to execute fuzzy matching on the CPU [8].

All the data is sampled from SNAC database.

6.2 Performance

To evaluate performance, we also implement the same algorithm as the proposed method, using Python's regular expression support on CPU. Instead of comparing total running time, we focus on matching time, the kernel we aim to accelerate. Total running time involves some other overhead not related to the matching process, such as index building for Lucene, and compiling for the AP. We query 300 names in total, because the average matching time stabilizes after roughly 300 names. We collect the time of the search function which only executes matching operations and calculate the average matching time for one name (Figure 7).

Matching time for Lucene and the RegEx CPU approach grow roughly linearly as the database size increases. The much slower speed for RegEx in Python has not been pinpointed; this is an area for future work. The best results of these two methods are obtained when the database size is the smallest. Detailed time is shown in Table 3 for a total number of names less than 14,000. 14,000 is the maximum number of names we can store on one single board. If the total number is smaller than this, the time stays constant and depends only on the total input length; if it is larger, we can either use many boards in parallel or replace the symbols on board. We reconfigure the symbol in STE and collect the time consumed. The matching time increases after 14,000 and stays constant until it

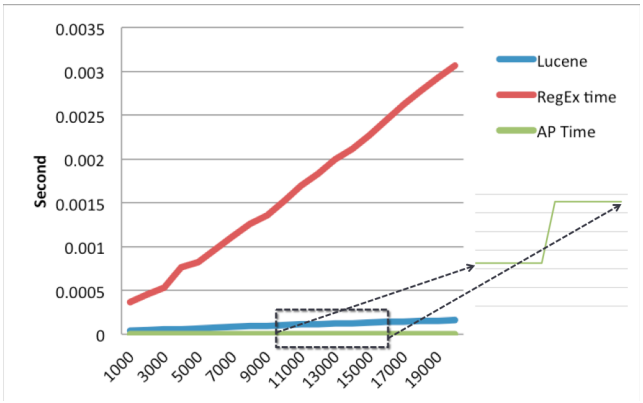

Figure 7: Average time for matching one name. (X axis represents the database size, ranging from 1,000 names to 20,000 names. Y axis represents the average matching time for one name.)

reaches 28,000 because it needs to re-stream the input after symbol reconfiguration. AP yields 121x, 4200x and 434x faster than Lucene for worst case, best case and average case.

	Best Case	Worst Case	Average
RegEx	366,670ns	2,113,330ns	1,200,530ns
Lucene	40,000ns	126,000ns	84,760ns
AP	30ns	330ns	195ns

Table 3: Detailed Running Time.

6.3 Accuracy

To evaluate ER accuracy, we use a subset of the names so that we can calculate the correct results manually. We use the manual results as a gold standard. First, we collect the compression rate (number of records after merging, over original number of records). The results are listed in Table 4. After processing, there are 366 records and 322 records left using Lucene and the AP-accelerated methods respectively. The proposed method is at least as good as Lucene; in fact, it combines 7.5% more records.

	Record Number	Compression Rate
Lucene	366	64.2%
AP	322	56.7%
Manual	268	47.0%

Table 4: Compression rate.

However, using only the compression rate cannot fully evaluate the result quality. Because it may group records together incorrectly; or it may not combine the records that should be grouped together. Therefore, we use two additional metrics to evaluate accuracy.

The first metric is the number of correct pairs. If there are more than two records in one group, every two records inside the group is counted as one pair. For example, if there are four records in one group, the correct pair count is 6. We compare the results to the gold standard calculated by hand. The results are presented in Table 5. The AP-accelerated method and

Lucene find 278 and 255 correct pairs respectively. The proposed method finds 7.6% more correct pairs than Lucene does.

	Correct Pairs #	**Pertage**
Lucene	255	81.5%
AP	279	89.1%
Manual	313	100%

Table 5: Correct pairs results.

The second metric is generalized merge distance (GMD) [13]. This method is based on the elementary operations of merging and splitting the records group. In our evaluation, we use a simple version of GMD, where the costs of merging and splitting are the same. We count the number of operations required to convert the results to the gold standard. The results are shown in Table 7. Both the AP-accelerated method and Lucene need 3 split operations. For merging, the AP-accelerated method only needs 30 operations, while Lucene needs 51. 24 of 30 are because the shorter form is stored on the AP first. Another 2 are because the names are not in the formats we currently support.

	Merge	**Split**	**Total**
Lucene	51	3	54
AP	30	3	33

Table 6: GMD results.

Our proposed method shows better results for all three metrics. This is due to the ability of the AP to better support fuzziness of matching.

In summary, the AP shows advantages on both performance and result quality. Note that the AP board we use in this paper is the first generation. Technology scaling projections suggest that, in the future, we may have larger capacity and higher frequency, which could lead to even better performance.

7 Summary and Future Work

In this paper, we propose using an accelerator-based solution to speed up ER using Micron's AP. The proposed method can make full use of the massive parallelism of the AP and search for up to 14,000 names simultaneously. We evaluate both performance and accuracy using real data sets from SNAC. On average, the AP-accelerated method works 434x faster than the CPU method, finds 7.6% more correct pairs, and needs 39% less GMD cost. In summary, the AP shows great potentials for accelerating ER.

Future work includes using the AP to process larger datasets and rare special cases, solve other string-based ER instances, and compare our proposed method with other existing methods (both software and hardware accelerations, e.g. GPU or FPGA).

8 Acknowledgements

This work was supported in part by C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by DARPA and MARCO.

References

[1] *Social Networks and Archival Context*. `http://socialarchive.iath.virginia.edu`.

[2] Hernández, M. A. et al. The merge/purge problem for large databases. In *ACM SIGMOD Record*, 1995.

[3] Monge, A. et al. An efficient domain-independent algorithm for detecting approximately duplicate database records. *Proceedings of the SIGMOD 1997 workshop on data mining and knowledge discovery*, 1997.

[4] Whang, S. E. et al. Entity resolution with iterative blocking. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009.

[5] Kolb, L. et al. Parallel entity resolution with dedoop. *Datenbank-Spektrum*, 2013.

[6] Kawai, H. et al. Bufoosh: Buffering algorithms for generic entity resolution. *Technical Report*, Stanford, 2006.

[7] Dlugosch, P. et al. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[8] *Apache Lucene*. `http://lucene.apache.org`.

[9] Wang, H. *Innovative Techniques and Applications of Entity Resolution*. IGI Global, 2014.

[10] Fang, Y. et al. Generalized pattern matching micro-engine. *Fourth Workshop on Architectures and Systems for Big Data*, 2014.

[11] Noyes, H. et al. Microns Automata Processor architecture: Reconfigurable and massively parallel automata processing. In *Fifth International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2014.

[12] Roy, I. et al. Finding motifs in biological sequences using the micron automata processor. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, 2014.

[13] Menestrina, D. et al. Evaluating entity resolution results. *Proceedings of the VLDB Endowment*, 2010.