*I.J.Modern Education and Computer Science,* **2010, 1, 48-56**
Published Online November 2010 in MECS (http://www.mecs-press.org/)

# Tuning Schema Matching Systems using Parallel Genetic Algorithms on GPU

Yuting Feng
School of Computer Science & Technology, Soochow University, Suzhou, China
Email: vivian_716@sina.com

Lei Zhao, Jiwen Yang
School of Computer Science & Technology, Soochow University, Suzhou, China
Email: {zhaol, jwyang}@suda.edu.cn

*Abstract*—**Most recent schema matching systems combine multiple components, each of which employs a particular matching technique with several knobs. The multi-component nature has brought a tuning problem, that is to determine which components to execute and how to adjust the knobs (e.g., thresholds, weights, etc.) of these components for domain users. In this paper, we present an approach to automatically tune schema matching systems using genetic algorithms. We match a given schema *S* against generated matching scenarios, for which the ground truth matches are known, and find a configuration that effectively improves the performance of matching *S* against real schemas. To search the huge space of configuration candidates efficiently, we adopt genetic algorithms (GAs) during the tuning process. To promote the performance of our approach, we implement parallel genetic algorithms on graphic processing units (GPUs) based on NVIDIA's Compute Unified Device Architecture (CUDA). Experiments over four real-world domains with two main matching systems demonstrate that our approach provides more qualified matches over different domains.**

*Index Terms*—**schema matching, tuning, genetic algorithms, GPU, CUDA**

## I. Introduction

Schema matching is the task of finding semantic correspondences, i.e. matches, between disparate metadata structures. It is a key problem in numerous applications, such as e-commerce, data warehousing, web-oriented data integration, schema evolution and migration, and peer-to-peer data management [1].

Traditionally, schema matching is performed manually, and it is a labor-intensive, time-consuming, and error-prone process. Thus researchers have developed various matching techniques and prototypes, i.e. matchers, which exploit element names, data types, descriptions, schema structures, and other types of information to find matches between schemas (see Ref. [1]—[3] for recent surveys). None of these matchers outperforms all the others on all existing benchmarks (like XBenchMatch [4]). Therefore, many matching systems (e.g., Ref. [5]—[9]) combine multiple matchers, called matching components, to achieve better matching results. The multi-component nature makes matching systems extensible and customizable, however, as pointed out in Ref. [10], it also brings a tuning problem that is to determine which components to execute and how to adjust the knobs (e.g., thresholds, weights, etc.) of these components, for given matching situation.

In this paper, we consider the tuning problem as an optimization problem. We describe an approach to automatically tune schema matching systems using genetic algorithms. Given a schema $S$ and a matching system $\mathcal{M}$, we first generate a synthetic workload $\mathcal{W}$ which consists of a set of matching scenarios involving $S$, then apply $\mathcal{M}$ to scenarios in $\mathcal{W}$ and learn the best configuration using genetic algorithms. We implement parallel genetic algorithms on consumer-level GPU based on NVIDIA's Compute Unified Device Architecture (CUDA) and promote the performance of genetic algorithms.

The main interesting features of our approach are:

- Learning a configuration which effectively improves the performance of a matching system with genetic algorithms. We allow users to make a trade-off between precision and recall while evaluating the performance of a configuration candidate.
- Providing a matching scenario generator which produces various matching scenarios for the input schema. It handles both relational and XML schemas, and it can be used as a training set generator for other learning-based matchers.
- Implement parallel genetic algorithms using consumer-level GPU based on CUDA, and optimize the performance of genetic algorithms with low exact cost.
- Experiments over four real-world domains with two main matching systems demonstrate that our approach provides more qualified matches over different domains.

The rest of the paper is organized as follows. Section 2 briefly surveys related work. Section 3 focuses on the drawbacks of existing solutions and the motivation of our work. Section 4 contains an overview of our approach. Section 5 describes the implementation on GPU. The

Corresponding author: Lei Zhao.

results of experiments that show the performance of our approach are presented in section 6. Finally, section 7 gives a conclusion.

## II. RELATED WORK

Schema matching has received increasing attention since 1990s. Various matching techniques and prototypes have been developed. These techniques exploit information such as element names, data types, structures, number of sub-elements, integrity constraints, and instance data to find semantic correspondences between schemas (see Ref. [1]—[3] for recent surveys). Beyond the schema and instance data information, several types of external evidence have been exploited, like past matches [5, 6] and usage information [11] — [13].

Obviously, an effective matching solution should employ many of the techniques, each on the type of information that it can exploit. Some recent works (e.g., Ref. [5], [7]—[9]) have described a system architecture that employs multiple matchers, each of which exploits well a certain type of information mentioned above. Such multi-component matching systems are extensible and customizable for a particular application domain [14, 15]. However, there is another serious problem for domain users: *given a particular matching task, how to choose the most suitable components to execute, and how to set the multiple knobs (e.g., thresholds, weights, coefficients, etc.)*. This problem is called *the schema matching systems tuning problem*.

Lately, several researchers present their solutions to solve the tuning problem of multi-component matching systems. In Ref. [16], F. Duchateau et al. present a factory which generates matchers according to user requirements. The factory learns how to apply the similarity measures and classifiers to achieve high matching quality based on matched schemas and expert matches provided by users. The factory could build a specified matcher for a given matching situation. In Ref. [17], A. Marie et al. propose an approach to combining matchers into to ensembles. They use the boosting algorithm which is a machine learning technique to select matchers that participate in an ensemble. The combined matchers could achieve better result than random schema matchers. F. Duchateau et al. describe a method which chooses a set of matchers to execute using decision tree algorithm [18]. The training process is also based on expert correspondences provided by domain users. Y. Lee et al. present the eTuner framework which mainly concerns relational schema matching problems [10]. eTuner generates a synthetic workload by perturbing input schemas randomly, and then finds the best configuration using staged greedy search method. In Ref. [19], E. Peukert et al. optimize the schema matching process using rewrite technique, which is often used in database query optimization. They represent the matching process as a directed graph model, and choose the components with lower cost using filter-based rewrite rules.

Some of the existing solutions focus on quickly developing robust matchers for a particular matching situation (e.g., Ref. [16, 17]). And others attempt to customize existing matchers for a given matching situation (e.g., Ref. [10], [18, 19]). Our work can be seen as a part of customization techniques. We are aiming at automating the customization using learning techniques, and reducing the high total cost of schema matching process.

## III. MOTIVATION

As previously mentioned, increasing matching systems employ multiple matchers to produce better matching results. However, without tuning, a matching system tends to produce inferior performance, because it cannot exploit domain characteristics. While necessary essential, tuning is difficult for domain users, due to the wide variety of matching techniques employed and the large number of knobs involved.

Although some researchers have proposed their solutions to the tuning problem, there are still two major drawbacks. First, some solutions require domain users to provide a knowledge base consists of matched schemas and expert correspondences (e.g., Ref. [16-18]). This requirement makes the solutions involve much manual effort. E-Tuner could generate synthetic workload automatically, but it only considers matching systems that handle relational representations. In fact, most multi-component matching systems could also handle other types of data representations, such as XML schemas, ontology, web forms, etc. Second, the configuration found by existing solutions sometimes is actually local optima rather than the global optimum of the tuning problem. To find out the best configuration, a better search strategy should be applied.

To solve previously mentioned drawbacks, our approach aims at designing a workload generator for both relational and XML schemas and searching the configuration space with genetic algorithms.

As pointed in Ref. [10], the challenge of workload generator is to develop perturbing rules for particular data representation. We represent input schemas with general data model and define a set of perturbing rules for both relational and XML schemas. On the other hand, genetic algorithms (GAs) have been demonstrated efficient search methods for solving wide-range of real-world problems, since introduced in 1960s. Although cannot be proofed in mathematics, GAs are still considered as powerful tools for optimization problems [20]. A recent study has shown that GAs has certain stability and it could improve the reliability by reiteratively computation and estimate the effects of improvements [21]. GAs might take a long time to find good solutions for some difficult problems. A promising approach to overcome this problem is to parallelize GAs [22]. Recently, several researchers have implemented parallel genetic algorithms (or parallel evolutionary algorithms) on graphics processing units (GPUs), and have demonstrated GPUs have a potential for acceleration of GAs (see Ref. [23] — [26] for more details).

## IV. OVERVIEW

Fig. 1 shows the architecture of our approach. There are two main modules: a workload generator which produces a synthetic workload $\mathcal{W}$ for the input schema $S$ and a GA-based tuner which searches for the most suitable configuration of matching system $\mathcal{M}$.

The rest of this section describes the workload generator and GA-based tuner in detail.

### A. Workload Generator

Given a schema $S$ and the workload size $n$, the generator returns a set of matching scenarios as the synthetic workload $\mathcal{W}$. The data model used internal by the generator is the nested relational model which is sufficiently general for both relational and XML schemas.

*Definition 1:* A matching scenario is a triple $(S, T, \Omega)$, where $S$ is a source schema, $T$ is a target schema, and $\Omega$ is a set of semantic correspondences between $S$ and $T$.

*1) Create schema pairs* $(S,T_1),(S,T_2),\ldots,(S,T_n)$*:* the workload generator creates matching scenarios by composing basic scenarios in different ways and perturbing element names and data types of $S$ randomly. We define four basic matching scenarios: *Copying*, *Merging*, *Vertical Partition*, and *Nesting* according to the basic mapping scenarios described in Ref. [27] and real-world schema matching specifications. The generator has implemented a set of common name transformation rules. Examples include replacing a name with a common abbreviation or a synonym obtained from WordNet database[1], and dropping prefixes or suffixes. The generator has also specified transformation rules between different data types. Fig. 2 presents examples of basic matching scenarios and perturbations.

The workload generator allows a user to tune the generated matching scenarios through a set of composition parameters. The composition parameters consist of 4 repetition parameters $\mathcal{R}$ (one for each basic scenario mentioned above), 2 construction parameters $\mathcal{C}$ (specify the number of sub-elements of an element and the depth of generated schemas), along with 2 standard deviation parameters $\mathcal{D}$. Every parameter in $\mathcal{C}$ is in fact the mean value of a Gaussian distribution whose standard deviation is the value of the corresponding parameter in $\mathcal{D}$.

*2) Create semantic correspondences* $\Omega_1,\Omega_2,\ldots,\Omega_n$*:* The workload generator retraces the generation history to create $\Omega$, which is a set of correct semantic correspondences between $S$ and $T_i$. Briefly, if element $t$ of $T_i$ is created from elements $s_1, s_2, \ldots s_k$ of $S$, then we create $\{s_1 = t, s_2 = t, \ldots, s_k = t\}$ as $\Omega_i$.

The generator could exploit auxiliary information $\mathcal{I}$, such as global schemas, expert matches, and dictionaries, to build a better workload, which improves the GA-based tuning performance. Fig. 3 gives a pseudo code description of the workload generator.

### B. GA-based Tuner

As previously described, our objective is to find out a configuration of matching system $\mathcal{M}$, which optimizes the performance over a synthetic workload $\mathcal{W}$, from a potentially huge space. To address this problem, we propose a tuning approach using the genetic algorithms. Genetic algorithms are a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators (e.g. crossover operator and mutation operator) to these structures so as to preserve critical information. Genetic algorithms are often viewed as function optimizers and could always locate good solutions in reasonable amounts of time [20]. Fig. 4 shows the prototype of GA-based tuner.

*1) Initialize the population:* Most GA practitioners use bit-string representations to represent potential solutions. We could represent a potential configuration as a chromosome with multiple genes, each of which reflects a knob of the configuration. When initializing, the population is generated randomly.

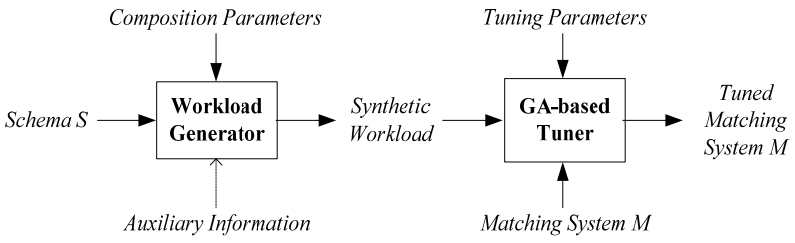



Figure 1. Architectural overview: This figure shows the main modules of our approach.

1: Cognitive Science Laboratory, Princeton University. WordNet: A Lexical Database for the English Language, http://wordnet.princeton.edu

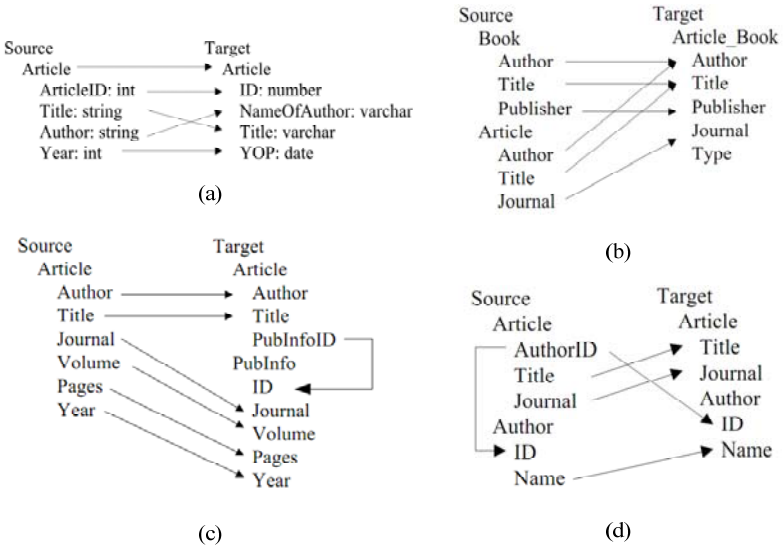



Figure 2. Examples of basic matching scenarios. (a) A copying scenario with perturbations on element names and data types. (b) A merging scenario using a column "Type" to identify records from different tables in source schema. (c) A vertical partition scenario which divides a table in source schema into two in the target. (d) A nesting scenario which creates a nested structure from a foreign key definition element in source schema.

**Input:** schema $S$, workload size $n$, auxiliary information $\mathcal{I}$ (optional)
repetition parameters $\mathcal{R}$, construction parameters $\mathcal{C}$, standard deviations $\mathcal{D}$
**Output:** synthetic workload $\mathcal{W}$
1. Create matching scenarios $(S, T_1), (S, T_2),\ldots, (S, T_n)$
   a. Let S be the source schema of the ith scenario
   b. Create the target schema $Ti$
      For each type of basic scenario $B$, let $r_B$ be the corresponding value of $B$ in $\mathcal{R}$
      For each j ∈ $[1, r_B]$
         Let $T_B$ = createBasicScenario($B$, $S$, $\mathcal{C}$, $\mathcal{D}$)
         Perturb element names and data types of $T_B$ (using $\mathcal{I}$) randomly
         $T_i$ =compose($T_i$, $T_B$)
2. Create semantic correspondences $\Omega_1, \Omega_2,\ldots, \Omega_n$
   a. Let $\Omega_i = \emptyset$
   b. For each element $t$ in $Ti$
      For each element $s$ in $S$, if $t$ is generated from $s$ then add $(s, t)$ to $\Omega_i$
3. Return $\mathcal{W} = \{(S, T_1, \Omega_1), (S, T_2, \Omega_2), \ldots, (S, T_n, \Omega_n)\}$

Figure 3. A pseudo code description of the workload generator.

**Input:** synthetic workload $\mathcal{W}$, matching system $\mathcal{M}$, fitness function $\mathcal{F}$
fitness threshold $t$, preference α, population size $n$
**Output:** the best-fit solution $h^*$
1. Initialize the population and generate $n$ possible solutions randomly
2. Evaluate each individual of the population
   For each possible solution $h$, let $\mathcal{F}(\mathcal{M}, \mathcal{W}, \alpha, h)$ be the fitness value of $h$
3. If $[\max_h \mathcal{F}(\mathcal{M}, \mathcal{W}, \alpha, h)] < t$, do
   a. Select the best-fit individuals
   b. Breed new individuals through crossover and mutation operations
   c. Evaluate the fitness of new individuals
   d. Replace the least-fit population with new individuals
4. Return the best-fit solution $h^*$

Figure 4. A pseudo code description of GA-based tuner.

*2) Evaluate potential solutions:* The fitness value of each individual in the population could be determined by the quality of its corresponding matching result. We allow domain users to specify the influence of *Precision* and *Recall* in the fitness evaluation.

*Definition 2:* The fitness function is

$$F-Measure(\alpha)=\frac{Precision\times Recall}{(1-\alpha)\times Precision+\alpha\times Recall} \quad (1)$$

α is a preference between *Precision* and *Recall*, and $0\le\alpha\le 1$. In particular, $F-Measure(\alpha)\to Precision$, when $\alpha\to 1$, and $F-Measure(\alpha)\to Recall$, when $\alpha\to 0$.

*3) Repeat generating process until termination:* Our tuner first selects the best-fit individuals and breeds new individuals through crossover and mutation operations, then evaluates the fitness of new individuals, and replaces least-fit population with new individuals. This generating process is repeated until a terminate condition (like time or generations limit, fitness threshold achieved, etc.) has been reached.

## V. IMPLEMENTATION OF PARALLEL GENETIC ALGORITHMS ON GPU

Parallel genetic algorithms (PGAs) have been considered a promising approach to make GAs faster. However, PGAs are usually applied on parallel, distributed, and networked computers which are not easy to access for many users. Recently, GPUs have become powerful tools for general purpose computing. Since GPUs have already been installed on most ordinary personal computers, the exact cost for using GPUs is quite low. Therefore, we implement PGAs on consumer-level GPU to promote the performance of tuning process.

The rest of this section presents the architecture of CUDA and our implementation of PGAs on GPU.

### *A. Architecture of CUDA*

CUDA (Computer Unified Device Architecture) is a general purpose parallel computing architecture which can be performed on any NVIDIA graphics card from GeForce 8 generation on both Linux and Windows platform. CUDA comes with a software environment that allows developers to use C as a high-level programming language. Therefore, programmers could develop parallelism programs on GPUs with relative ease [28].

CUDA is a platform for massively parallel high-performance computing on NVIDIA's powerful GPUs. At its cores are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. CUDA lets programmer partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

Fig. 5 shows the architecture of CUDA. Thread is basic unit to manipulate data in CUDA. The thread index is a 3-dimesion vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. All the threads have access to the concurrently block's shared memory. Those multi-dimensional blocks are organized into one-dimensional or two-dimensional grids, each block can be identified by one-dimensional or two-dimensional index, and all grids share one global memory.

CUDA requires all the active processors to execute the same instruction at the same time but on different data, also called SIMD for Single Instruction Multiple Data. As mentioned above, GAs need to evaluate different individuals in the population using an identical fitness

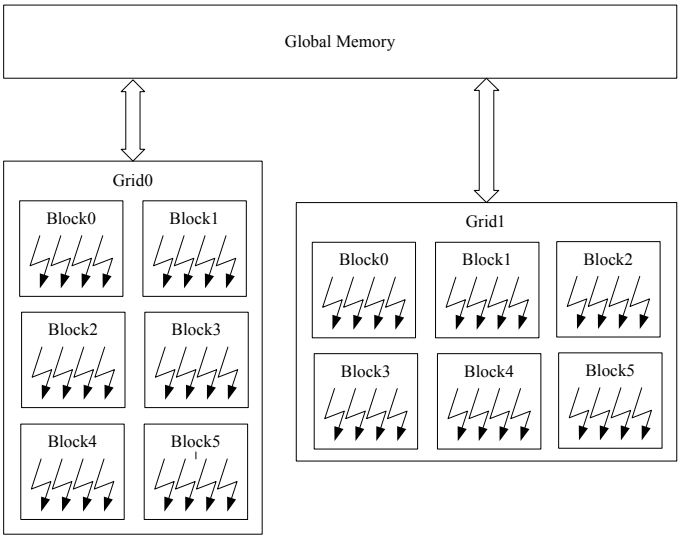



Figure 5. The architecture of CUDA.

function and are suitable to be parallelized on the architecture of CUDA.

### B. Implementation of PGAs on GPU

There are several types of PGAs: global single-population master-slave PGAs, single-population fine-grained PGAs, multiple-population coarse-grained PGAs, and hierarchical PGAs [22, 26]. In a master-slave PGA there is a single population, but the evaluation of fitness and/or genetic operators are distributed among several processors. Fine-grained PGAs are suited for massively parallel computers and consist of one population, which has a spatial structure. The interactions between individuals are limited: an individual could only compete and mate with its neighbors. However, the neighborhoods overlap permits some interactions among all individuals. Multiple-population PGAs assume that several subpopulations (demes) evolve in parallel and demes are relatively isolated. These demes exchange individuals occasionally, and the exchange is also called migration. Hierarchical PGAs combine multiple-population PGAs with master-slave PGAs or fine-grained PGAs, and have better performance than any of them alone.

According to the architecture of CUDA, we implement the multiple-population PGAs in a particular way. *Initialization:* An initial population is generated on CPU and transferred to the global memory of GPU.

*1) Partition:* The initial population is partitioned into several subpopulations. Each subpopulation is distributed to the shared memory of a thread block.

*2) Evolution:* Each thread block runs the evolution independently. The implementations of genetic operators, such as selection, crossover, and mutation have been described in Ref. [24]—[26].

*3) Migration:* The model of migration is single-direction migration. Each thread block transfers its best individuals to its *next* block. Since every thread block could be identified using thread block index, it is easy to achieve such migration. Fig. 6 shows the migration model of multiple-population PGAs.

*4) Update & Termination:* The global memory is updated with new individuals. The process would stop if the terminate condition has been reached.

## VI. EXPERIMENTS

We have evaluated our approach over two common matching systems, COMA++ [5] and SimFlood [29] with similarity metrics from SimMetrics[2], applied to four real-world XML schema matching tasks. **Publicans**, **Person**, and **University** are available in XBenchMatch benchmark [4]. **SMS** is provided by YAM as a test case [16]. Detailed features of each task are shown in Fig. 7.

To compare with our approach, we examine other two tuning methods: (1) Applying the off-the-shelf matching systems without any tuning. (2) Manual tuning, by tweaking a few knobs, examining the output of a matching system, and then adjusting the knobs again.

Our experiments are performed using an Intel Core2 Duo T5250 CPU 1024M RAM and an NVIDIA GeForce 8400M GS GPU. To compare with the implementation of

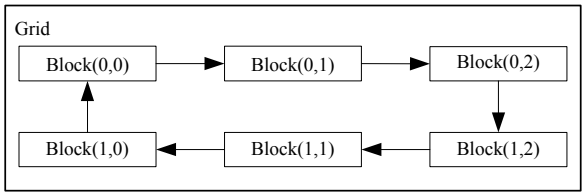



Figure 6. The migration model in our implementation of multiple-population PGAs.

PGAs on GPU, we also implement the tuning process on CPU with the help of JGAP[3], a genetic algorithms and genetic programming component provided as a java framework.

To simplify the tuning process, we only consider a few influential knobs of each matching system. This selection is done by applying the attribute selection of Weka[4] before running our approach.

We test each method 30 times to limit the impact of randomness. Each time, we generate a synthetic workload with 15--30 scenarios for each matching task. For our approach, we set the preference between precision and recall to 0.5, and the threshold of fitness value to 0.95. We set different values to the population size. Concerning about time consuming, we set the maximized number of generation to 5.

Fig. 7 shows the matching performance of COMA++ and SimFlood. It demonstrates that our method could effectively improve the performance of schema matching systems. However, the performance of a tuned system is relevant to the matching algorithms of the system. For example, the average F-Measure (0.5) value of tuned SimFlood over matching task Publicans is 0.3226 (less than half of the average F-Measure (0.5) value of tuned COMA++, 0.8205), because we do not use any dictionary in this task, and the matching algorithm is mainly driven by the initial textual match.

Fig. 8 shows the effects of population size on the run time when we apply our approach on COMA++ system for the SMS task. We divide the initial population into 2, 4, and 8 subpopulations (thread blocks). Each subpopulation consist 10 individuals (threads). The maximum number of threads per block of our device is 512, and is fair enough for larger subpopulations. The experiment shows that our approach runs faster on GPU especially when larger population involved. However, the run time of tuning process still strongly depends on the matching algorithms and the size of synthetic workload.

2: String Similarity Metrics for Information Integration, http://www.dcs.shef.ac.uk/~sam/stringmetrics.html

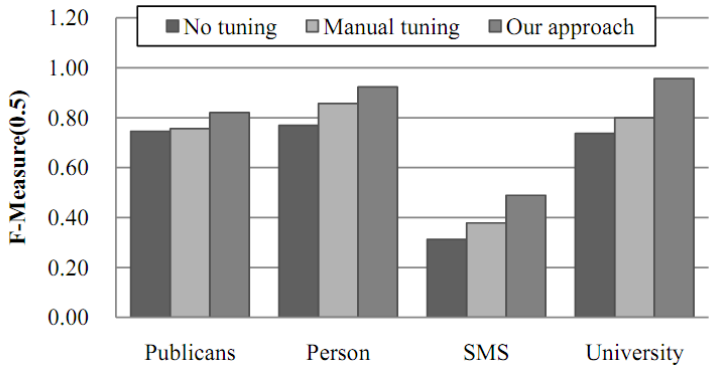
3: JGAP: Java Genetic Algorithms Package, http://jgap.sourceforge.net

4: Weka: Java Programs for Machine Learning, http://www.cs.waikato.ac.nz/~ml/weka

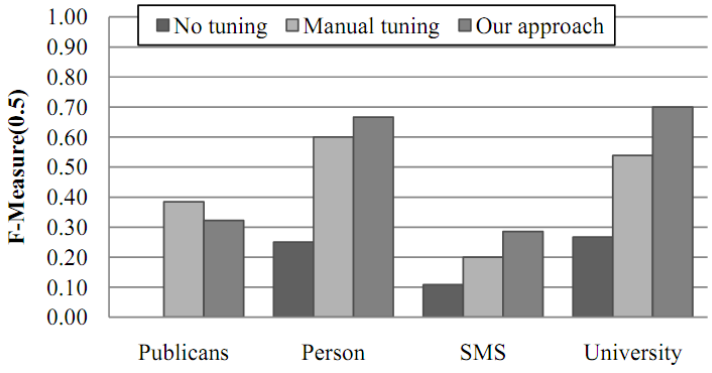
 

TABLE I. SUMMARY OF FOUR XML MATCHING TASKS

| | Publicans | Personne | SMS | University |
|---|---|---|---|---|
| Number of nodes ($S_1/S_2$) | 42/55 | 10/9 | 40/29 | 17/17 |
| Number of paths ($S_1/S_2$) | 117/81 | 10/9 | 64/44 | 17/17 |
| Number of matches | 65 | 7 | 24 | 12 |




(a)




(b)

Figure 7. Matching performance of (a) COMA++ and (b) SimFlood

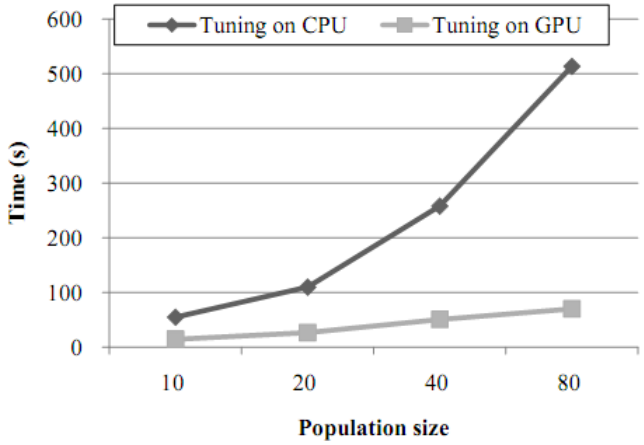



Figure 8. The effects of population size on the run time while tuning COMA++ system for the SMS task.

## VII. CONCLUSIONS

In this paper, we have presented an approach to automatically tune schema matching systems using genetic algorithms. To promote the performance of our approach, we implement parallel genetic algorithms on GPU based on CUDA. The main contributions of our work are: (1) providing a matching scenario generator for learning-based matchers, and (2) providing better performance by tuning matching systems for particular matching situation. Experiments over four real-world domains with two main matching systems demonstrate that our approach provides more qualified matches over different domains.

## ACKNOWLEDGEMENT

The paper is supported by National Natural Science Foundation of China (No. 61073061). The authors are grateful to all the people for helpful suggestions. The authors would like to thank all the anonymous reviewers for their helpful comments on earlier drafts of this paper.

## REFERENCES

[1] E. Rahm and P. A. Bernstein, "A Survey of Approaches to Automatic Schema Matching," *The VLDB Journal*, vol. 10, no. 4, 2001, pp. 334–350.

[2] P. Shvaiko and J. Euzenat, "A Survey of Schema-based Matching Approaches," *Journal on Data Semantics IV*, vol. 3730, 2005, pp. 146–171.

[3] H. H. Do, "Schema Matching and Mapping-based Data Integration," *PhD Thesis*, Department of Computer Science, University at Leipzig, Germany, 2006.

[4] F. Duchateau, Z. Bellahsène, and E. Hunt, "XBenchMatch: a Benchmark for XML Schema Matching Tools," in *VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007, pp. 1318–1321.

[5] D. Aumueller, H. H. Do, S. Massmann, and E. Rahm, "Schema and Ontology Matching with COMA++," in *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, 2005, pp. 906–908.

[6] Y. Qian, J. L. Nie, G. H. Liu, and S. H. Gao, "Corpus-based Complex Schema Matching," *Journal of Computer Research and Development (Chinese)*, vol. 43, no. Spply., 2006, pp. 200-205.

[7] P. A. Bernstein, S. Melnik, and J. E. Churchill, "Incremental Schema Matching," in *VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006, pp. 1167–1170.

[8] C. Drumm, M. Schmitt, H. H. Do, and E. Rahm, "Quickmig: Automatic Schema Matching for Data Migration Projects," in *CIKM '07: Proceedings of the 16th ACM Conference on Information and Knowledge Management*, 2007, pp. 107–116.

[9] P. Shvaiko, F. Giunchiglia, and M. Yatskevich, *Semantic Web Information Management, A Model-based Perspective*. *Springer Berlin Heidelberg*, 2010, ch. 9 Semantic Matching with S-Match, pp. 183–202.

[10] Y. Lee, M. Sayyadian, A. Doan, and A. Rosenthal, "eTuner: Tuning Schema Matching Software using Synthetic Scenarios," *The VLDB Journal*, vol. 16, no. 1, 2007, pp. 97–122.

[11] A. Nandi and P. A. Bernstein, "HAMSTER: Using Search Clicklogs for Schema and Taxonomy Matching," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, 2009, pp. 181–192.

[12] H. Elmeleegy, M. Ouzzani, and A. Elmagarmid, "Usage-based Schema Matching," in *ICDE '08: Proceedings of the 24th International Conference on Data Engineering*, 2008, pp. 20–29.

[13] G. R. Ding, G. H. Wang, and Y. H. Zhao, "Multi-schema Integration based on Usage and Clustering Approach," *Journal of Computer Research and Development (Chinese)*, vol. 47, no. 5, 2010, pp. 824-831.

[14] P. A. Bernstein, S. Melnik, M. Petropoulos, and C. Quix, "Industrial-strength Schema Matching," *SIGMOD Record*, vol. 33, no. 4, 2004, pp. 38–43.

[15] H. H. Do and E. Rahm, "Matching Large Schemas: Approaches and Evaluation," *Information Systems*, vol. 32, no. 6, 2007, pp. 857-885.

[16] F. Duchateau, R. Coletta, Z. Bellahsene, and R. J. Miller, "YAM: a Schema Matcher Factory," in *CIKM '09: Proceedings of the 18th ACM Conference on Information and Knowledge Management*, 2009, pp. 2079–2080.

[17] A. Marie and A. Gal, "Boosting Schema Matchers," in *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems:*, 2008, pp. 283–300.

[18] F. Duchateau, Z. Bellahsene, and R. Coletta, "A Flexible Approach for Planning Schema Matching Algorithms," in *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems:*, 2008, pp. 249–264.

[19] E. Peukert, H. Berthold, and E. Rahm, "Rewrite Techniques for Performance Optimization of Schema Matching Processes," in *Proceedings of EDBT 2010, 13th International Conference on Extending Database Technology*, 2010, pp. 453–464.

[20] D. Whitley, "A Genetic Algorithm Tutorial," *Statistics and Computing*, vol. 4, no. 2, 1994, pp. 65–85.

[21] Q. Yue, and S. Feng, "The Statistical Analyses for Computational Performance of the Genetic Algorithms," *Chinese Journal of Computers (Chinese)*, vol. 32, no. 12, 2009, pp. 2389–2392.

[22] E. Cantú-Paz, "A Survey of Parallel Genetic Algorithms," *Calculateurs Paralleles*, vol. 10, 1998.

[23] P. Pospichal, J. Jaros, and J. Schwarz, "Parallel Genetic Algorithm on the CUDA Architecture," in *Proceedings of the Applications of Evolutionary Computation, EvoApplicatons 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC, Part I*, 2010, pp. 442-451.

[24] C. F. Tan, A. G. Ma, and Z. C. Xing, "Research on the Parallel Implementation of Genetic Algorithm on CUDA Platform," *Computer Engineering & Science (Chinese)*, vol. 31, no. A1, 2009, pp. 68-72.

[25] T. T. Wong, and M. L. Wong, "Parallel Evolutionary Algorithms on Consumer-Level Graphics Processing Unit," *Parallel Evolutionary Computations*, 2006, pp. 133-155.

[26] S. F. Zhang, and Z. M. He, "Implementation of Parallel Genetic Algorithm Based on CUDA," in *ISICA '09: Proceedings of the 4th International Symposium on Advances in Computation and Intelligence*, 2009, pp. 24-30.

[27] B. Alexe, W. C. Tan, and Y. Velegrakis, "STBenchmark: Towards a Benchmark for Mapping Systems,"

*Proceedings of the VLDB Endowment*, vol. 1, no. 1, 2008, pp. 230–244.

[28] NVIDIA C., "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," *NVIDIA Corporation*, 2007.

[29] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching," in *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, 2002, pp. 117–128.

**Yuting Feng** received the B.S. degree in 2008 from Soochow University, Suzhou, China. She has been a M.S. candidate in the school of computer science and technology of Soochow University since 2008. Her current research interests include management information systems and information integration.

**Lei Zhao** received the Ph.D. degree in 2006 from Soochow University, Suzhou, China. He has been a faculty member of the school of computer science and technology of Soochow University since 1998. He is now Associate Professor at the Department of Network Engineering. His research interests include distributed data processing, data mining, parallel and distributed computing.

**Jiwen Yang** received the B.S. degree in 1984 from Nanjing Normal University, Nanjing, China. He has been a faculty member of the school of computer science and technology of Soochow University since 1984. He is now Professor at the Department of Information Management. His research interests include distributed data processing, management information system, parallel and distributed computing.