Optimizing Similarity Computations for Ontology Matching - Experiences from GOMMA

Michael Hartung[1,2], Lars Kolb[1], Anika Groß[1,2], and Erhard Rahm[1,2]

[1] Department of Computer Science, University of Leipzig
[2] Interdisciplinary Center for Bioinformatics, University of Leipzig
{hartung,kolb,gross,rahm}@informatik.uni-leipzig.de

Abstract. An efficient computation of ontology mappings requires optimized algorithms and significant computing resources especially for large life science ontologies. We describe how we optimized n-gram matching for computing the similarity of concept names and synonyms in our match system GOMMA. Furthermore, we outline how to enable a highly parallel string matching on Graphical Processing Units (GPU). The evaluation on the OAEI LargeBio match task demonstrates the high effectiveness of the proposed optimizations and that the use of GPUs in addition to standard processors enables significant performance improvements.

Keywords: ontology matching, GPU, parallel hardware.

1 Introduction

Mappings (alignments) between ontologies are important for many life science applications and are increasingly provided in platforms such as BioPortal [13]. New mappings are typically determined semi-automatically with the help of ontology match systems such as GOMMA (Generic Ontology Matching and Mapping Management) [10] utilizing different matchers to evaluate the linguistic and structural similarity of concepts [3]. Ontology matching is challenging especially for large ontologies w.r.t. both effectiveness (achieving a high quality mapping) and efficiency, i.e., fast computation [16]. Results of the 2012 OAEI [14] LargeBio task[1] showed that some systems still have problems or are even unable to match large ontologies such as the Foundation Model of Anatomy (FMA) [5] or the Thesaurus of the National Cancer Institute (NCIT) [11].

For high efficiency, it is important to reduce the search space by avoiding the comparison of dissimilar concepts [2,9], and to utilize optimized implementations for frequently applied similarity functions such as n-gram, Jaccard, TF-IDF (e.g., by using fast set intersection [1], or pruning techniques [18]). Libraries such as SimMetrics[2] typically provide a comfortable and general interface getSim(string1, string2) for multiple similarity measures but often lack efficient implementations. For example, they either lack pre-processing steps to transform strings

[1] http://www.cs.ox.ac.uk/isg/projects/SEALS/oaei/2012/
[2] http://sourceforge.net/projects/simmetrics/

C.J.O. Baker, G. Butler, and I. Jurisica (Eds.): DILS 2013, LNBI 7970, pp. 81–89, 2013.

into representations permitting faster comparisons or they cause redundant preprocessing steps when matching a particular string multiple times.

A promising direction to speed-up processing-intensive computations such as string comparisons is the utilization of Graphical Processing Units (GPU) supporting a massively parallel processing even on low-cost graphic cards. The availability of frameworks like CUDA and OpenCL further stimulated the interest in general purpose computation on GPUs [15]. Algorithms like BLAST [17], database joins [8] or duplicate detection/link discovery systems [4,12] have already been adapted for GPU execution. Unfortunately, GPUs and their programming languages like OpenCL have several limitations. For instance, only basic data types can be used, the memory capacity of a GPU is restricted to a specific size, and data must first be transferred to the GPU. Furthermore, no dynamic memory allocation is possible, i.e., the resources required by an algorithm must be known and allocated a priori. These restrictions need to be considered in a new solution for computing string similarity in match systems such as GOMMA.

In this experience paper, we make the following contributions:

- We describe how we optimized n-gram matching for linguistic matching in GOMMA including the use of integer representations for n-grams. (Sec. 2)
- We propose a new method for n-gram matching on GPU. The technique is not limited to n-gram and can also be applied to other token-based string metrics (e.g., Jaccard). We further describe how GOMMA can exploit both CPU and GPU resources for improved efficiency. (Sec. 3)
- We evaluate our techniques on a real-world match problem, namely the FMA-NCI match task from the OAEI LargeBio track. The results show that we are able to significantly reduce the execution times on CPU as well as GPU compared to the standard solution. (Sec. 4)

2 Optimizing N-Gram Similarity Computation

GOMMA uses the n-gram string comparison to determine the similarity of names and synonyms for pairs of concepts of two input ontologies O and O'. The example in Fig. 1 shows the names/synonyms for three concepts per ontology. The match result is a mapping $M_{O,O'} = \{(c, c', sim) \mid c \in O,\ c' \in O',\ sim \in [0,1]\}$ consisting of correspondences between concepts and their match similarity. Given that a concept has a name and potentially several synonyms, there can be several n-gram similarities per pair of concepts. GOMMA thus applies an aggregation function agg, e.g., maximum or average, to aggregate multiple similarity values. Finally, GOMMA uses a threshold t to restrict the mapping to the most likely correspondences.

A naive n-gram matcher first splits the string attribute values to be compared into overlapping tokens of length n. For our example and n=3 (Trigram), the c_2 strings limbs and extremity are split into {{lim,imb}, {ext,xtr,tre,rem,emi,mit,ity}} while the single c'_2 attribute value limbs is tokenized into {{lim,imb,mbs}}. To determine the similarity between two concepts, we (1) need to compute the dice

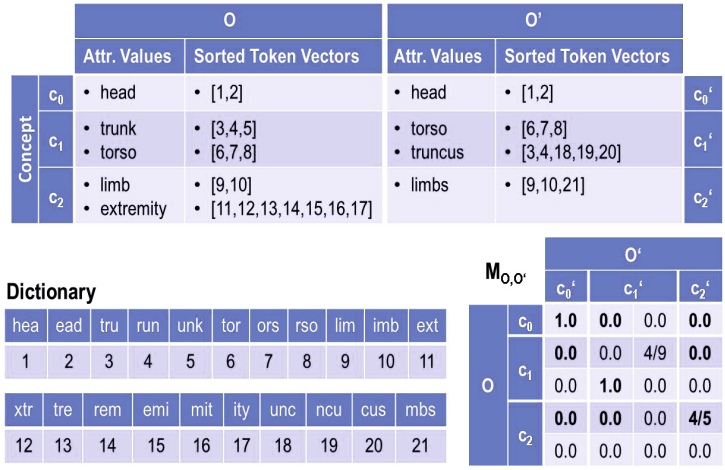

Fig. 1. Example trigram similarity computation. Attribute values (names, synonyms) are converted to sorted token vectors (upper part). Tokens are represented as integers based on a dictionary (lower left part). Individual similarities are aggregated with the max function to determine an overall similarity between two concepts (lower right part).

coefficient for each pair of token sets TS_1-TS_2 ($\mathsf{diceSim}(TS_1,TS_2)=\frac{2\cdot|TS_1 \bigcap TS_2|}{|TS_1|+|TS_2|}$) and (2) aggregate the single similarities. For instance, when determining the similarity between c_2 and c_2', we compute two single similarities $\mathsf{diceSim}(\{\mathsf{lim,imb}\}, \{\mathsf{lim,imb,mbs}\})=0.8$ and $\mathsf{diceSim}(\{\mathsf{ext,xtr,tre,rem,emi,mit,ity}\}, \{\mathsf{lim,imb,mbs}\})=0$ which are aggregated using the max function: $sim(c_2, c_2')=\mathsf{max}(0.8, 0)=0.8$.

There are several possibilities to compute the set intersection ($TS_1 \cap TS_2$) that can have a large impact on efficiency, e.g., a nested loop over both element sets or the use of hash tables. Furthermore, in case of string-valued tokens, even the check whether two tokens (strings) are equal is a quite complex operation. In general, the larger the sets and the longer the tokens are, the more time is required to compute set overlaps. Since such similarity computations frequently occur in match workflows for a large number of concepts, it turns out that an efficient implementation of the token set intersection is a key factor to speed up the matching of large ontologies. In recent years, different optimization techniques for set similarity joins (e.g., prefix, suffix, and length filtering) were proposed in the context of near duplicate detection [18]. We omit those orthogonal optimizations in favor of readability and leave their application for future work.

GOMMA's optimized n-gram matcher is described in Algorithm 1. It is based on two optimizations: the use of integer representations for tokens and a sort-merge-based computation of token set overlaps. As in the naive approach, we first split a concept's string attribute values into token sets (Line 6). We then convert all n-grams into integer values based on a global dictionary (Line 8). The dictionary is built dynamically, i.e., each time a new n-gram is observed during the tokenization, it is added to the dictionary and, from then on, represented by its integer-valued index in the dictionary. Additionally, we sort the integer

Algorithm 1. ngramSim($O, O', Attr, agg, t$)

```
1  foreach c ∈ O ∪ O' do
2      c.stvs ← ∅;   // sorted token vectors
3      S ← c.getAttrValues(Attr);
4      foreach s ∈ S do
5          stv ← [] ;  // empty token vector
6          tokens ← tokenizeNGrams(s);
7          foreach t ∈ tokens do
8              n ← getNumericTokenId(t);
9              stv.append(n);
10         c.stvs ← c.stvs ∪ {stv.sort()};
11 M ← ∅;
12 foreach c ∈ O do
13     foreach c' ∈ O' do
14         Sims ← ∅;
15         foreach stv ∈ c.stvs do
16             foreach stv' ∈ c'.stvs do
17                 s ← diceSim(stv, stv');
18                 Sims ← Sims ∪ {s};
19         sim ← agg(Sims);
20         if sim ≥ t then
21             M ← M ∪ {(c, c', sim)};
22 return M;
```

Algorithm 2. diceSim(stv_1, stv_2)

```
1  left ← 0;
2  right ← 0;
3  overlap ← 0;
4  l1 ← stv1.length();
5  l2 ← stv2.length();
6  while (left<l1) ∧ (right<l2) do
7      if stv1[left]==stv2[right] then
8          overlap++;
9          left++;
10         right++;
11     else if stv1[left]<stv2[right]
       then
12         left++;
13     else
14         right++;
15 return 2 · overlap / (l1 + l2);
```

values of each token vector in ascending order (Line 10). Thus, after this pre-processing, a concept has a set of sorted token vectors (*stvs*) representing the n-grams of their string attribute values as integers. For example, the trigrams of c_2 are represented as {[9,10], [11,12,13,14,15,16,17]}.

We then iterate over all concepts of O and O' and compare the sorted token vectors of concepts with each other. In case of multiple attribute values for a concept, the single similarities are aggregated to an overall similarity using the specified aggregation function agg. Our pre-processing allows for a very efficient overlap computation (Algorithm 2) similar to the Sort-Merge-Join used for efficient join computation in databases. Since all token sets are represented by sorted token vectors *stv*, we can do interleaved linear list scans to compute the overlap. We thus only perform $|stv_1|+|stv_2|$ comparisons in the worst case with a fast integer-based token comparison. For instance, when comparing {lim,imb} with {lim,imb,mbs}, we compare [9,10] with [9,10,21] requiring merely the comparison of the two integer pairs 9-9 and 10-10.

GOMMA also supports the parallel execution of string matching for disjoint sets of concept pairs to utilize multiple processors or cores for improved execution time. In the evaluation (Sec. 4), we will also consider this performance option.

3 GPU-Based N-Gram Similarity Computation

A GPU-based implementation needs to overcome common GPU limitations, namely (1) lack of string data type, (2) only restricted data structures such as arrays, and (3) a priori allocation of a fixed and limited amount of memory. Since our algorithm operates on integer values, the first limitation is already

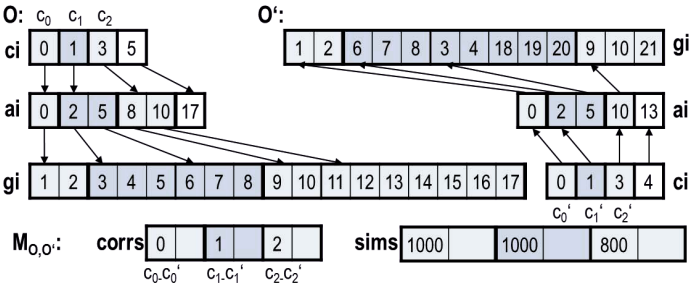

Fig. 2. GPU input and output data structures for running example and top-k=2

solved. For the second limitation we will use an index structure based on arrays. We will overcome the third limitation by partitioning large input ontologies, adapting memory-efficient data types, and determining only the best matches per concept to restrict the mapping size. We further use an execution scheme that minimizes expensive data transfers between main memory and GPU. In the following, we describe the utilized data structures and outline the n-gram similarity computation on GPUs.

Input Data Structure: In contrast to dynamically growing data structures (e.g., lists or maps) usable for CPU-based computations, GPU-based processing necessitates the preallocation of the required memory on the target device. Because the number of attributes per concept and the number of n-grams per attribute value varies, a mapping to fixed-length data structures is required. For this purpose, we adopt a multi-level index structure (illustrated in Fig. 2 for the running example) consisting of three arrays per input ontology: concept index (ci), attribute index (ai), and gram index (gi). The arrays ci and ai represent the concepts and their attributes, respectively, while gi holds the sorted token vectors of the input concepts. For each concept, there is one entry in ci pointing to its first string attribute in ai. The number of attributes for concept j thus is ci[j+1]-ci[j]. Each ai entry represents a particular string and points to the first token of its value in gi. The last (dummy) entries of ci/ai are used to mark the end of each index. Using this structure, one can easily access the tokens of an attribute of a particular concept. For instance, to access the tokens of concept $c_2 \in O$, we first read ci[2]=3 and ci[2+1]=5 to find the lower (inclusive) and the upper (exclusive) bound of its attributes in ai. Hence, the concept has two attributes represented by ai[3]=8 and ai[5-1]=10. The values at these positions can be used to access the sorted token vectors beginning at gi[8] and gi[10], respectively. To save memory and transfer costs, we use short instead of integer data types to represent the tokens (2 instead of 4 bytes per token).

Output Data Structure: The memory for storing the match result must be reserved a priori as well. To limit the result size, we utilize the observation that it is sufficient to consider only the top-k best correspondences (above threshold t) for each concept without reducing match quality. This approach marks an

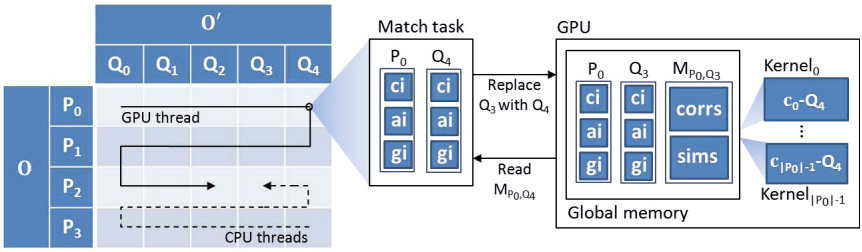

Fig. 3. Execution scheme for hybrid CPU/GPU-based n-gram similarity computation minimizing the data transfer between the host program and the GPU

upper bound of the required memory to allocate on the GPU. Our output data structure consists of two arrays corrs and sims. The former contains the ids of the (at most) top-k matches per concept, the latter contains the corresponding similarities. For our running example, we would create two arrays of length 6 to store the best two matches for each concept of O (see bottom of Fig. 2). Again, the amount of memory and data transfer can be reduced by using the short data type (instead of float) to express the similarity values. In particular, we limit their precision to three decimal places which is sufficient for match processes, e.g., the similarity value 0.8 for c_2-c'_2 is expressed by a short value of 800.

N-Gram Execution on GPU: Compared to CPUs, the architecture of GPU hardware exhibits a large number of simpler compute cores that execute the same instruction on multiple data partitions. In this study, we rely on the OpenCL framework for general purpose computation GPUs. OpenCL code is written in C as so-called compute kernels, whose submission is controlled by a host program executed on the CPU. The actual number of kernel instances running in parallel depends on the GPU's number of cores, its amount of memory, the kernel programs memory requirements, and the size of the input and output data. OpenCL assigns a global unique identifier to each kernel instance. This identifier is used to compute global memory offsets for loading and storing input data that a particular kernel is operating on.

In general, the input ontologies and the $|O|{\cdot}k$ resulting correspondences exceed the available memory of the GPU. Thus, we up-front split both input ontologies into partitions $P_i \subseteq O$ and $Q_i \subseteq O'$, analogously as in our previous work on parallel ontology matching [7]. We then iteratively ship pairs (P_i, Q_i) for comparison to the GPU. The GPU executes a kernel instance for each $c \in P_i$ that compares c with all $c' \in Q_i$ and determines its top-k correspondences above $\mathtt{t}$. The partial results are later unified by the host program. For this purpose, we utilize a job queue that supports the parallel n-gram similarity computation of different partition pairs on both the GPU as well as on the CPU. A dedicated thread takes match tasks from this queue and submits them to the GPU. In addition to this GPU thread, several CPU threads can access the job queue from the opposite end to independently perform matching on the CPU. We select jobs and ship partitions using the scheme displayed in Fig. 3. This scheme

ensures that after completion of a GPU job only a single partition needs to be transferred to the GPU. The other partition remains in the GPU's memory and is reused for the next job. For instance, when the GPU finished the P_0-Q_3 job, it starts to execute P_0-Q_4 next. In this case, only the partition Q_3 needs to be replaced by Q_4 and P_0 can be reused. Furthermore, it is beneficial to split the larger of the two input ontologies into partitions. If it even fits entirely into the device's memory, only partitions from the smaller ontology need to be replaced.

4 Evaluation

We analyzed the execution time for computing the FMA-NCIT mapping which is part of the LargeBio match task in OAEI [14]. The task consists of three sub-tasks namely, small (3,720×6,551 concepts), large (28,885×25,678 concepts), and whole (79,042×66,914 concepts). To create mappings of high quality, we applied the GOMMA match workflow with top-k=1 and n=3 (Trigram) used in OAEI 2012 (for details and quality results see [6]). The experiments were carried out on an Intel i5-2500 machine (4x3.30GHz, 8GB memory). We further used the following mid-range GPU: Asus GTX660 with 960 CUDA cores/2GB memory.

The first experiment evaluates the execution times for the three sub-tasks utilizing either one CPU thread or the GPU. For CPU-based processing, we compare the proposed SortInt n-gram matching with two alternatives using nested-loop (NLString) and hash set look-ups (HashString) for computing the token set overlap. The results displayed in Fig. 4 (left) show that SortInt-CPU significantly outperforms both standard algorithms. For the whole task, it requires ≈8 min compared to about 26 min (104 min) for HashString (NLString), i.e., it improves runtime by up to a factor of 13. This shows that our pre-processing step pays off, i.e., converting strings into integer values and sorting are non-expensive (<1 sec in all tasks) but valuable steps for an optimized overlap computation. The application of SortInt on the GTX660 GPU allows for a further significant improvement compared to the CPU implementation. The execution time for the whole task is reduced by another factor of 5 to merely 99 sec. Thus, transferring the data into the GPU pays off, i.e., the massively parallel hardware in the form of hundreds of CUDA cores substantially speeds up the computation.

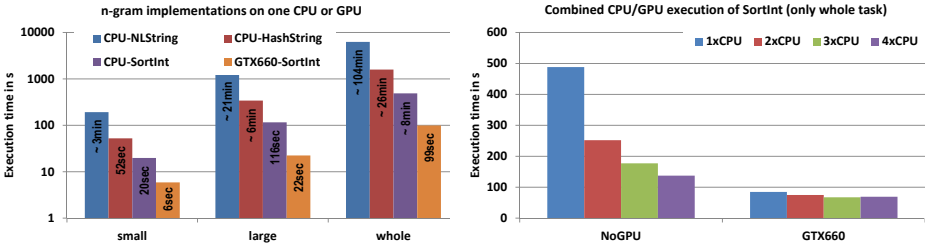

Fig. 4. Runtime of n-gram algorithms on CPU/GPU (left) and combined (right)

In a second experiment, we evaluate how application on multiple cores either without GPU (NoGPU) or in combination with GPU resources affects execution times. As shown in Fig. 4 (right), we observe that parallel CPU processing is very effective, e.g, when using four CPU threads, the execution time can be reduced to 137 sec (factor of 3.5) for the whole match task. The combined execution on CPU and GPU can further improve the execution time to about 67 sec for three and four CPU threads (factor of 2). The fourth CPU thread does not further improve the execution time due to the dedicated GPU thread for data transfer. Overall, one can see that even a moderately powered GPU can substantially reduce the execution time for string and thus for ontology matching.

5 Conclusion and Future Work

We studied how similarity functions like n-gram used for linguistic matching in GOMMA can be optimized by algorithmic tuning as well as by massively parallel processing on GPUs. The results indicate that intelligent pre-processing (e.g., integer conversion, sorting) of the input ontologies pays off substantially and speeds up ontology matching. The GPU-based execution of algorithms like n-gram matching requires some effort to overcome the GPU limitations but boosts performance even further. In the future we plan to investigate further GPU-based similarity computations and the impact of different kinds of GPU hardware.

References

1. Ding, B., König, A.C.: Fast set intersection in memory. PVLDB 4(4) (2011)
2. Ehrig, M., Staab, S.: QOM – quick ontology mapping. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 683–697. Springer, Heidelberg (2004)
3. Euzenat, J., Shvaiko, P.: Ontology matching. Springer, New York (2007)
4. Forchhammer, B., et al.: Duplicate Detection on GPUs. In: BTW (2013)
5. Foundation Model of Anatomy, http://fma.biostr.washington.edu/
6. Gross, A., Hartung, M., Kirsten, T., Rahm, E.: GOMMA Results for OAEI 2012. In: Proc. 7th Ontology Matching Workshop (2012)
7. Gross, A., Hartung, M., Kirsten, T., Rahm, E.: On matching large life science ontologies in parallel. In: Lambrix, P., Kemp, G. (eds.) DILS 2010. LNCS, vol. 6254, pp. 35–49. Springer, Heidelberg (2010)
8. He, B., et al.: Relational joins on graphics processors. In: Proc. SIGMOD (2008)
9. Hu, W., Qu, Y., Cheng, G.: Matching large ontologies: A divide-and-conquer approach. Data & Knowledge Engineering 67(1) (2008)
10. Kirsten, T., Gross, A., Hartung, M., Rahm, E.: GOMMA: A Component-based Infrastructure for managing and analyzing Life Science Ontologies and their Evolution. Journal of Biomedical Semantics 2, 6 (2011)
11. NCI Thesaurus, http://ncit.nci.nih.gov/
12. Ngomo, A.-C.N., Kolb, L., Heino, N., Hartung, M., Auer, S., Rahm, E.: When to reach for the cloud: Using parallel hardware for link discovery. In: Cimiano, P., Corcho, O., Presutti, V., Hollink, L., Rudolph, S. (eds.) ESWC 2013. LNCS, vol. 7882, pp. 275–289. Springer, Heidelberg (2013)

13. Noy, N., et al.: BioPortal: ontologies and integrated data resources at the click of a mouse. Nucleic Acids Research 37(suppl. 2) (2009)
14. Ontology Alignment Evaluation Initiative, http://oaei.ontologymatching.org/
15. Owens, J., et al.: GPU computing. Proceedings of the IEEE 96(5) (2008)
16. Rahm, E.: Towards Large Scale Schema and Ontology Matching. In: Schema Matching and Mapping. Springer (2011)
17. Vouzis, P., Sahinidis, N.: GPU-BLAST: using graphics processors to accelerate protein sequence alignment. Bioinformatics 27(2) (2011)
18. Xiao, C., Wang, W., Lin, X., Yu, J.X., Wang, G.: Efficient similarity joins for near-duplicate detection. ACM Trans. Database Syst. 36(3) (2011)