Scaling Parallel Rule-based Reasoning

Martin Peters[1], Christopher Brink[1], Sabine Sachweh[1], and Albert Zündorf[2]

[1] University of Applied Sciences Dortmund, Germany,
Department of Computer Science
{`martin.peters` ‖ `christopher.brink` ‖ `sabine.sachweh`}`@fh-dortmund.de`

[2] University of Kassel, Germany, Software Engineering Research Group,
Department of Computer Science and Electrical Engineering
`zuendorf@cs.uni-kassel.de`

Abstract. Using semantic technologies the materialization of implicit given facts that can be derived from a dataset is an important task performed by a reasoner. With respect to the answering time for queries and the growing amount of available data, scaleable solutions that are able to process large datasets are needed. In previous work we described a rule-based reasoner implementation that uses massively parallel hardware to derive new facts based on a given set of rules. This implementation was limited by the size of processable input data as well as on the number of used parallel hardware devices. In this paper we introduce further concepts for a workload partitioning and distribution to overcome this limitations. Based on the introduced concepts, additional levels of parallelization can be proposed that benefit from the use of multiple parallel devices. Furthermore, we introduce a concept to reduce the amount of invalid triple derivations like duplicates. We evaluate our concepts by applying different rulesets to the real-world DBPedia dataset as well as to the synthetic Lehigh University benchmark ontology (LUBM) with up to 1.1 billion triples. The evaluation shows that our implementation scales in a linear way and outperforms current state of the art reasoner with respect to the throughput achieved on a single computing node.

Keywords: #eswc2014Peters, scaleable reasoning, rule-based reasoning, GPU, parallel, RETE algorithm

1 Introduction

In order to enable the semantic web and other semantic applications, the derivation of new facts based on a given dataset is one key feature that is provided by the use of reasoners. Query answering and the provision of a complete set of information often is a performance critical task. This gets even more important with respect to the growing amount of available information, that often needs to be processed. Thus, fast and scaleable reasoning is essential for the success of many semantic applications. In [1] we described first results of a rule-based and highly parallel reasoner running on massively parallel hardware like GPUs. Unlike many other parallel reasoner implementations (e.g. [2], [3], [4]), our approach is based on the RETE algorithm [5] and does not rely on a cluster-based approach like MapReduce implementations. The use of RETE allows us to easily load different rulesets and apply them to input data. Thus, our forward chaining reasoner is not dependent on a specific ruleset like RDFS or pD* [6] and can be used for inference based on any application specific semantics that can be expressed using rules.

In this paper we introduce new concepts of workload partitioning as well as new levels of parallelization. Both aspects allow us to perform a scaleable and efficient reasoning using parallel hardware even on large ontologies that do not fit into the on-board memory of a GPU. In particular, we introduce a workload partitioning for each of the different steps of the RETE algorithm. This workload partitioning on the one hand allows us to introduce a further parallelization on the host side (that part of the application, that does not run on massively parallel hardware), and on the other hand easily allows to distribute the workload over multiple GPUs and thus to scale the hardware in a horizontal way.

In the next section we start with an introduction to the parallel implementation of the RETE algorithm for semantic reasoning before we introduce the workload partitioning schemes. Based on the partitioning schemes new levels of parallelization are proposed (new levels because they can be applied in addition to the already introduced parallel matching algorithm described in [1]). Furthermore, a strategy for reducing the derivation of invalid triples like duplicates is presented in section 3. Section 4 will evaluate our approach and show different aspects of scaleability, effectiveness of parallelization and performance of the reasoner. For this purpose we reason about datasets with up to 1.1 billion triples using different rulesets. Finally we discuss our findings with respect to related work and conclude the paper.

2 Using RETE for a Rule-based Reasoner Implementation

The RETE algorithm is a pattern matching algorithm and was introduced by Charles L. Forgy [5]. The algorithm is based on a network of nodes, which are derived by the given set of rules. The network consists of alpha and beta nodes, where an alpha node has no parents and represents exactly one rule-term. Thus, for each unique rule-term of a given ruleset an alpha node is created. A beta node in turn always has two parents which may be alpha or beta nodes. Thus, a beta node always represents at least two rule patterns and links two or more single rule-terms of one rule. Assuming the rules R1 and R2 from the pD* rules (also known as OWL-Horst) the resulting network is shown in figure 1.

$$(?v \text{ owl:hasValue } ?w)\ (?v \text{ owl:onProperty } ?p)\ (?u\ ?p\ ?w) \rightarrow (?u \text{ rdf:type } ?v) \qquad \text{(R1)}$$

$$(?v \text{ owl:hasValue } ?w)\ (?v \text{ owl:onProperty } ?p)\ (?u \text{ rdf:type } ?v) \rightarrow (?u\ ?p\ ?w) \qquad \text{(R2)}$$

Finally, the node $\beta 2$ represents the complete rule R1 and the node $\beta 3$ the rule R2. To apply the ruleset to a set of input triples (each consisting of a subject, predicate and object (s, p, o)), basically three steps are necessary. The first one is the alpha-matching and means to match every input triple against every alpha node. Because $\alpha 1$ is completely unbound (all of the three rule-term elements are

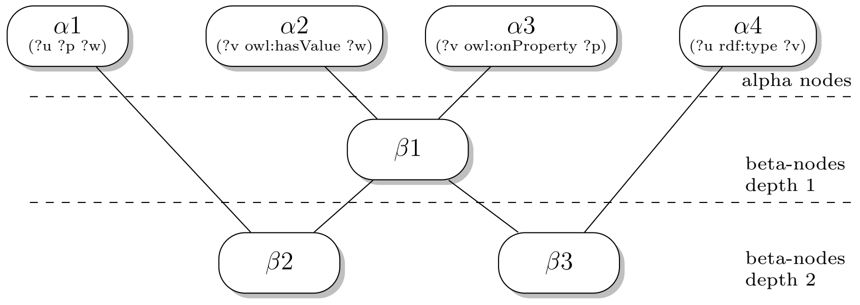

Fig. 1. RETE network for rules R1 and R2

variables) every triple will match. To match the condition of $\alpha 2$, the predicate of a triple needs to be *owl:hasValue*. Other alpha nodes are treated accordingly. For each node a list of matching triples (working memory) is created. Basically this list consists of references to the corresponding triples. The working memories of the alpha-nodes are the starting point for the second step of the RETE algorithm, the beta-matching.

During beta-matching, each match of the first parent node is combined with each match of the second parent node to see, if both matches together satisfy the conditions of the beta node. For example for $\beta 1$ the matches of $\alpha 2$ and $\alpha 3$ need to share the subject ($?v$) to be a match of $\beta 1$. After the matches of the beta nodes of the depth 1 (see figure 1) are computed, the matches of the next level of beta-nodes can be computed, too. Once the matches of all beta nodes are determined, the working memories of the final nodes of a rule can be used to fire the rules and derive new facts, which is the third step of the RETE algorithm. The final node of a rule is that node, that represents the complete rule body like mentioned before. Thus, the working memory of $\beta 2$ is used to fire R1 and the working memory of $\beta 3$ is used to fire R2. The new derived triples then need to be propagated through the network until no new triples are derived.

2.1 Parallelizing the RETE algorithm

Addressing massively parallel hardware like GPUs the main challenge is to partition the workload in a way that it can be computed by millions of threads in parallel. Looking at the RETE algorithm we have to consider the three different steps of alpha-matching, beta-matching and rule-firing that need to be parallelized. The concepts for a parallel alpha- and beta-matching were already introduced in [1]. The main idea for an efficient alpha-matching is to match every triple against all of the alpha nodes and not the other way round. This means that the number of threads that are submitted to the parallel hardware is equal to the number of input triples. The resulting list then is transformed to the working memories of the individual alpha-nodes.

The beta-matching is based on a similar concept. To compute the matches of one beta-node, the amount of threads is created on a parallel hardware that corresponds to the number of matches of one of the parent nodes. Each of the created threads holds a reference to exactly one match of the corresponding parent node and iterates through all of the matches of the second parent node. Assuming that $\alpha 2$ in figure 1 has 500 matches and $\alpha 3$ has 300 matches, a total of 500 threads is created where each thread iterates through all of the 300 matches of $\alpha 3$. For more details regarding the alpha- and beta-matching as well as an efficient matching implementation on the GPU we refer to [1].

Rule firing in [1] was performed in a serial way, which means that a single thread iterated through all matches of the final node of a rule and created the resulting triples. However, this easily can be performed in parallel, too, by submitting a thread for each match of a final node to the parallel hardware and derive the triples. Note that a thread on massively parallel hardware is much more lightweight than for example in a Java application and thus the overhead is accordingly small. The resulting triples finally need to be checked against duplicates and can be propagated through the RETE network, too.

2.2 Introducing Workload Partitioning

Like the evaluation in [1] showed, the introduced concept for a parallel RETE implementation running on massively parallel hardware performed very well regarding the performance. Nevertheless, the concept was limited by the size of the input data that could be processed. To address this issue for the alpha-matching, the workload can easily be partitioned into smaller chunks that can be processed independently and thus can be sized to an adequate size with respect to the target device. Note that in parallel programming the *device* always refers to the parallel hardware like a GPU. Figure 2 illustrates the partitioning in preparation of an alpha step for n input triples that need to be matched against p alpha nodes.

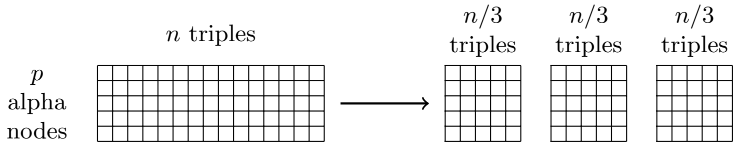

Fig. 2. Workload partitioning for alpha-matching

For beta-matching and rule-firing this partitioning is not applicable because the working memories of the nodes in the RETE network, that are used for both steps, consist of references to triples of the internal triple store. To perform the matching or rule-firing, these references need to be resolved because the corresponding triples are needed for further processing (like the matching). Thus, the complete set of available triples needs to be loaded to the main memory of the GPU. Accordingly the maximum size of the processable input data depends on the memory size of the device. To overcome this issue, we introduce a *triple-match* with the following definition:

Definition 1. *A triple-match $m = (s, p, o, r)$ is a quadruple with s=subject, p=predicate, o=object of a triple and r=triple reference (unique number, that is used for identification in the internal triple store).*

According to this definition, a triple-match not only holds the reference r to the corresponding triple, but also the triple itself. By using this data structure for computations on parallel devices instead of the pure working memory of a node we eliminate the need to transfer the complete set of available triples to the device. However, working memories that are needed for example for a beta-match processing need to be transferred to a list of triple-matches before execution. Because the triple-match holds the triple itself as well as the reference, the resulting data of an alpha- or beta-matching can still consists only of the triple references.

Another benefit on using triple-matches during a beta-match is that it allows us to perform a similar workload partitioning like for the alpha-matching. Because during beta-matching all matches of one parent node (n_{parent_1} matches) are matched against all matches of the other parent node (m_{parent_2} matches), n and m (see figure 3) both might become very large. Thus, not only a partition in one dimension is desirable, but also in two dimensions.

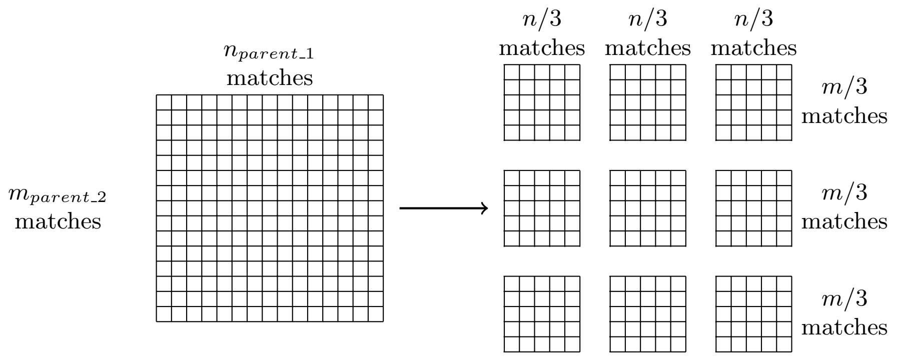

Fig. 3. Workload partitioning for beta-matching

As illustrated in figure 3 the complete workload can be divided into smaller chunks, where the size of the chunks can be chosen with respect to the used device considering for example the amount of available memory. This allows to separately submit each chunk for processing to the device and reading back the results.

For rule-firing the use of triple-matches also allows to partition the workload. Therefore, the matches of a final node can be split up into chunks of an adequate size, transferred into triple-matches and being processed on the device. The resulting triples that are transferred back from the device to the host finally can be submitted to the triple-store of the reasoner where they need to be checked against duplicates.

2.3 Workload Distribution

The introduced workload partitioning not only allows to process large datasets in small chunks, it also enables us to introduce new levels of parallelization and finally to distribute the workload over multiple devices. Considering the example from figure 1, it can be seen that the beta-matching of all beta nodes of one depth can be performed independently and thus simultaneously. That is because a beta-matching in a depth of d always relies on the results of nodes with a depth $< d$. Based on this understanding the first level of additional parallelization can be introduced by computing beta-matches of all beta-nodes of one depth in parallel. A further level of parallelization is possible through the workload partitioning, where each single partition of one beta-node can be processed in parallel, too.

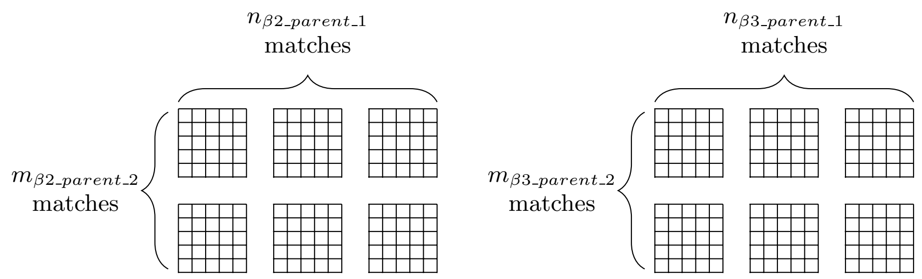
$n_{\beta2\_parent\_1}$ matches
$m_{\beta2\_parent\_2}$ matches
$n_{\beta3\_parent\_1}$ matches
$m_{\beta3\_parent\_2}$ matches

Fig. 4. Further levels of parallelization for beta-matching

Figure 4 illustrates the two levels of parallelization by showing the partitioned workload for $\beta2$ and $\beta3$ from the example RETE-network. Besides the fact that both nodes can compute their matches independently of one another, the created chunks can be computed in parallel, too. Finally this leads to the following number of possible parallel computations of one depth:

$$\sum_{i=1}^{B} \frac{n_i}{chunkwidth_i} * \frac{m_i}{chunkheight_i} \tag{1}$$

Note that B denotes to the number of beta-nodes in one depth and *chunkwidth* and *chunkheight* are constants defining the size of a chunk, which can be chosen with respect to the target device and size of n_i and m_i. We further assume that n_i always divides by *chunkwidth* and m_i by *chunkheight*.

Applying this additional parallelization allows to easily use multiple massively parallel devices like GPUs. While one GPU can only perform one task (for example process one chunk) at a time, the total workload can be distributed over multiple devices. This way the reasoner can be parallelized on the host side (the host defines the execution environment like a java application from where a parallel device is accessed) as well as on the device side. This concept can also be applied to the rule-firing, where each rule can be processed independently of one another and thus can be processed in parallel, too. A further level of parallelization is achieved by the workload partitioning, which has been mentioned before.

3 Reduction of Invalid Triples

As already stated out in [1], rule-firing can be the most time consuming task during the reasoning process, depending on the used rules and dataset. One reason for this is the huge amount of duplicates as well as triples holding a literal as a subject, that get inferred during rule-firing. Such triples need to be identified and rejected by the triple store of the reasoner before the new triples are stored. While the check of a literal-subject can for example be performed by a direct lookup using an array holding a boolean value for every unique triple element (s, p, o), the identification of duplicates is often performed using a HashMap. Nevertheless, both methods of triple validation have the drawback that they are performed after the triples have been created. Thus, the triples first need to be derived before they can be validated.

The issue of a high rate of invalid triples is particularly noticeable computing the complete RDFS closure, where most of the triples are derived by the following rules:

$$(?x\ ?p\ ?y) \rightarrow (?p \text{ rdf:type rdf:Property}) \tag{R3}$$

$$(?x\ ?p\ ?y) \rightarrow (?x \text{ rdf:type rdfs:Resource}) \tag{R4}$$

$$(?x\ ?p\ ?y) \rightarrow (?y \text{ rdf:type rdfs:Resource}) \tag{R5}$$

What can be seen on looking at these rules is, that all existing triples will match the condition of the rules. That means that the number of output triples that finally need to be validated is equal to the number of input triples for the rule-firing of these rules. However, many of the derived triples will be duplicates because for R4 for example, all triples which share one subject will produce the same output triple. The same applies to R3 and R5 except that the output triple only depends on the predicate or object of the input triples.

With the proposed concept of triple-matches, which are also used for rule-firing on parallel hardware, we can use these findings to introduce a simple reduction of duplicates by an evaluation during the triple-match creation. When preparing the triple-matches for rule-firing of a specific rule, the rule header is also known and can be evaluated to the condition that there is only one variable term like $?x$ in the header. On the other side it is known, which triple element (subject, predicate or object) of the input triples will be placed to the variable of the rule header. For R4 for example the subject of the input triple would be placed to the subject of the resulting triple, too. Thus, a HashMap can be created which stores all occurrences of subjects during triple-match creation. Based on this HashMap a check can reveal if the subject already exists in the Map and thus the triple-match would result in a duplicate. If so, the triple-match can be rejected.

This does not only reduce the amount of triples that need to be validated before they are stored to the triple store, it also reduces the amount of triple-matches that need to be created and processed on the device. A similar concept can be applied to rules where the subject of the rule-header is a variable. In this case, the element that would be placed to the subject of the resulting triple can be checked if it is a literal or not. Only in the later case, the triple-match needs to be created. Nevertheless, these concepts provide a cheap way in terms of computation time to reduce the amount of invalid triples. They do not completely avoid the derivation of invalid triples and thus a final check before storing is still necessary.

4 Evaluation

Our evaluation has three goals: First of all we want to show the impact of the introduced concepts to avoid invalid triple derivations. Secondly we want to analyze the effect of the new levels of parallelization as well as workload distribution by using multiple GPUs. Finally we want to test the scaleability of our approach for datasets with up to one billion triples.

4.1 Implementation

For evaluation purpose of the proposed concepts we extended our implementation of the reasoner presented in [1]. The reasoner is written in Java and uses OpenCL[3] to perform highly parallel processing tasks on heterogenous devices like multicore CPUs or GPUs. The jocl-Library[4] is used for OpenCL Java bindings. The internal triple store is implemented as a singleton and manages the parsed triples as well as derived triples. Lists and HashMaps are used to store the data and to allow a fast lookup, for example to check against duplicates. The new levels of parallelization that were introduced in section 2 are implemented using multithreading in Java. Each thread that is responsible to compute one chunk, for example during beta-matching, prepares all needed data like the triple-matches and submits a task to a synchronous queue. For every available GPU in the execution environment of the reasoner a worker-thread is created, that polls for new tasks on the queue and executes it on the corresponding parallel device. This way it can be guaranteed that each processing task has exclusive rights to the device during execution. We also optimize the idle time of the devices by ensuring that each task that is submitted to the queue has already prepared all needed data. By using the concept of worker-threads that are responsible to access the available devices, the application dynamically adapts to the number of existing parallel devices and thus fully exploits the hardware.

[3] OpenCL: open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/

[4] http://www.jocl.org/

4.2 Test Environment and Datasets

To evaluate the proposed concepts, we use three different rulesets with varying complexity that are often implemented by other reasoners, too. The ρdf [7] ruleset is a simplified version of the RDFS vocabulary and consists of all RDFS rules with at least two rule terms. This ruleset is often used for a time efficient reasoning, because the results of the omitted rules could be provided by the reasoner on the fly if required. The second ruleset is the complete RDFS ruleset like it is defined by the W3C[5]. Finally we use the pD* [6] ruleset (also know as OWL-Horst) which incorporates RDFS and D entailment and has some basic support for OWL. For the complete set of pD* rules we refer to [2].

The used datasets are the DBPedia Ontology 3.9 [8], including the mapping-based types and mapping-based properties, as well as the Lehigh University benchmark (LUBM) ontology [9]. The DBPedia ontology is a lightweight ontology containing extracted information from Wikipedia and thus is a real world dataset. The complete datasets consists of more than 41 million triples. Nevertheless, we scaled this dataset to different sizes by using only every n'th instance triple to get $1/2^{th}$, $1/4^{th}$, $1/8^{th}$, $1/16^{th}$, and $1/32^{nd}$ of the dataset. The LUBM ontology is designed for benchmarks and is a de-facto standard for performance evaluations and comparison for RDF reasoner. A generator can be used to create datasets representing a university scenario, where the number of generated universities is used to size the resulting dataset. Thus, it can be used to create artificial datasets of an arbitrary size. The LUBM datasets used for evaluation are annotated with the corresponding number of universities that are included, such that for example LUBM250 refers to 250 universities. Table 5 gives an detailed overview of the used datasets.

Dataset	Scale	Triples
DBPedia	1/32	1,322,055
	1/16	2,627,952
	1/8	5,238,518
	1/4	10,453,153
	1/2	20,807,047
	full	41,447,376

Dataset	Scale	Triples
LUBM	125	17,607,267
	250	35,150,241
	500	72,090,481
	1000	144,121,737
	2000	289,967,483
	4000	581,452,623
	8000	1,164,702,737

Fig. 5. Used datasets

We perform our tests on two different machines. The first one that is used for all tests except the scaleability test is equipped with two mid range AMD 7970 gaming GPUs, each having 3 GB of on-board memory, a 2.0 GHz Intel Xeon processor with 6 cores and 64 GB of system memory. For the scaleability test

[5] http://www.w3.org/TR/rdf-mt/#RDFSRules

more system memory is needed to process the large LUBM datasets, which is why we use a cloud server with a total of 192 GB of memory, two Tesla M2090 GPUs each having 6GB of on-board memory and a 2.4 GHz Intel Xeon processor with 12 cores. Every test is executed five times and the average time, excluding dictionary encoding, is given.

4.3 Invalid triples

First of all we want to analyze the impact of the proposed concepts for reducing invalid triples during triple-match creation. Therefore, we use two different datasets with a similar size (LUBM250 $\approx$ 35M triple, DBPedia $\approx$ 41M triple) and apply the RDFS ruleset. We chose the DBPedia as well as the LUBM dataset because LUBM has a very high number of instance triples (ABox) while the TBox is proportionally small. The DBPedia dataset in turn also has a larger TBox and should provide more reliable results for a real world scenario. We compare the use of a non-parallel implementation of rule-firing with a parallel implementation using the GPU as well as a parallel implementation using the proposed concepts for reducing invalid triple derivations.

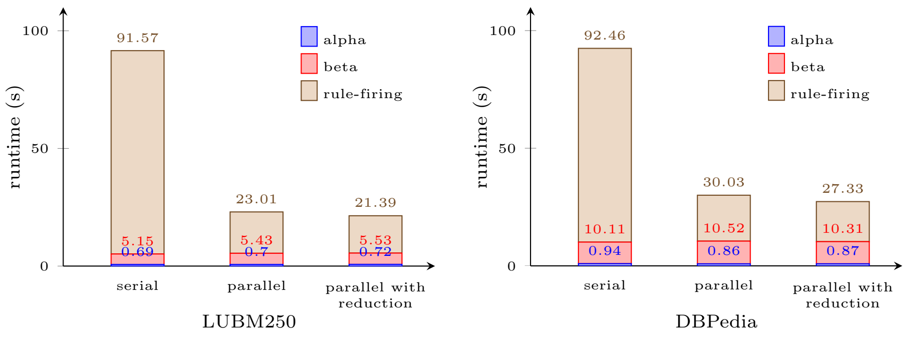

Fig. 6. Detailed reasoning time for LUBM250 and DBPedia using serial rule-firing, parallel rule-firing and parallel rule-firing with reduction of invalid triples

Figure 6 illustrates the processing time for the different phases of the RETE algorithm for each of the different rule-firing strategies. First of all it can be noted that the parallel implementation of rule-firing is about four times faster than the serial one. This speedup is achieved only by building the resulting triples including their hash-code on the GPU. The triples still need to be added to the internal triple-store where they are validated against duplicates before they get stored. By applying the concept of invalid triple reduction the triples that were submitted to be stored could be reduced for the LUBM dataset from about 227M to 130M which corresponds to a reduction of about 43%. For the DBPedia dataset a reduction of 35% could be achieved (202M triple creations instead of 312M). Nevertheless, the speedup that is accomplished for rule-firing is only 12.75% for DBPedia and 9.76% for LUBM. This is on the one hand because the application of the reduction strategy introduces an overhead during triple-match creation, too. On the other hand the deduplication based on a hash-lookup, where the hash is already computed together with the triple on the GPU, is very effective.

4.4 Parallelization

Furthermore we want to evaluate the impact of the new introduced levels of parallelization as well as the impact on using multiple GPUs to distribute the workload. Therefore, we use the RDFS as well as the pD* ruleset. We chose these two because they differ in complexity and thus can benefit in different ways from the introduced concepts.

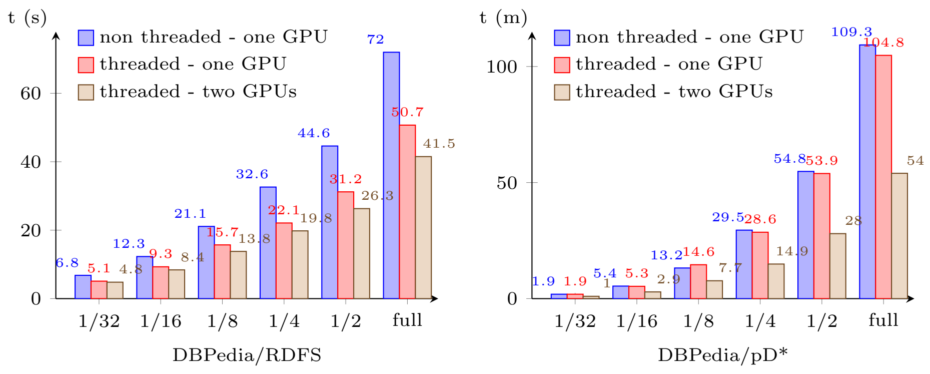

Fig. 7. Using RDFS (left) and pD* (right) on the DBPedia datasets

Because the computation of RDFS does not rely too much on work that needs to be performed on the GPU (all submitted tasks are of an adequate size), figure 7 shows that RDFS benefits primarily from the new introduced level of parallelization. This kind of parallelization allows the reasoner to perform much more work on the host at the same time while the time that a process is waiting to be executed on the GPU is relatively moderate. Thus, the speedup achieved by using a second GPU is of moderate size, too. On the other side the pD* ruleset relies much more on a high number of matches that need to be computed and thus executed on the GPU. Accordingly, the use of a second GPU drastically speeds up the execution time such that a doubling of the number of GPUs nearly results in a half of the processing time. It also can be expected that additional GPUs would further speedup the execution time as the use of additional hardware does not introduce an overhead. Both results show, that the workload partitioning and the concepts of further parallelization build on the partitioning are very efficient.

4.5 Scaleability

Finally we want to examine the execution time for the full materialization for datasets with a growing number of triples. For this tests we use the ρdf ruleset as well as the RDFS ruleset because both are widely used for performance and scaleability tests on LUBM datasets [10] [4] [11] [12] [13] and thus offer a good comparability. Figure 8 shows the results for both rulesets applied to the LUBM datasets from 17.6M triples to more than 1.1 billion triples.

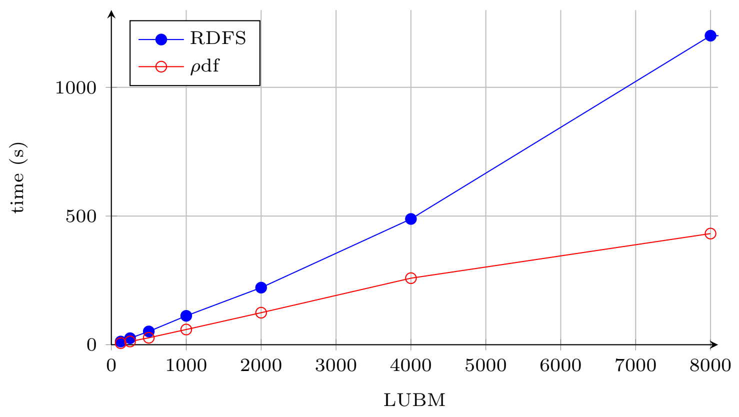

Fig. 8. Complete materialization time for LUBM datasets with up to 1.1 billion triples

As can be seen our implementation has a good scaleability since the execution time grows almost in a linear way with respect to the number of input triples for both rulesets. For the LUBM8000 dataset using the RDFS ruleset a further overhead is caused by the fact, that the size of the working memories for some beta-nodes exceeds 2 billion matches and we have to swap the matches to the hard-disk. By appending all matches to a file and accessing them in a partitioned way using a random access, we overcome the issue of the max array size in Java.

We further observe that the computation of ρdf is much faster than RDFS, which among other things can be explained by the number of inferred triples (for example on LUBM4000 143.7M inferred triples for ρdf and 238.2M for RDFS) and the findings of figure 6 that show, that the rule-firing is still the most computation intensive task for RDFS. This also illustrates the bottleneck of our implementation for these two rulesets. While the pD* ruleset could benefit from additional GPUs, the less computation intensive rulesets are thwarted by the internal triple-store which only allows to write new triples for a single process at a time. Nevertheless, we reach a throughput of up to 2.7M triples/sec. for ρdf and 1.4M triples/sec. for RDFS, which to our knowledge is the highest throughput for a single machine ever published.

5 Related Work and Discussion

While many reasoner, whether they use one computing node or a cluster of nodes, programmatically implement a specific set of rules, our implementation is based on the RETE algorithm and thus independent of a specific ruleset. In [13] an inference system that is able to support user-defined rules is proposed. The implementation is based on an Oracle database and is evaluated against LUBM datasets, too. Nevertheless, to apply RDFS to a LUBM1000 dataset a processing time of 6:34 hours is reported, while our approach performs the same computation in 269 seconds. Another single-node implementation, but with the limitation of a predefined set of rules, is shown in [12]. The DynamiTE reasoner is capable to perform stream reasoning and thus focuses on incrementally maintaining large datasets. While our approach is not able to process RDF streams, the pure materialization of ρdf in [12] achieves a throughput of 227 triples/sec. which is about 12 times slower than our results. In [14] an approach for reasoning on massively parallel hardware is presented that, in contrast to our approach which implements a generic rule engine, implements only the ρdf ruleset based on methods that are executed in a given order to produce the results. Another limitation of the reasoner in [14] is that only datasets, that fit into the device memory of a single GPU (multiple GPUs are not supported) can be processed. Both approaches were already used in [1] for a performance comparison. Further implementations of reasoners that allow a scaleable processing of large datasets often rely on the MapReduce framework. WebPIE [2][15] for example uses the MapReduce implementation Hadoop and is able to perform RDFS as well as pD* reasoning. WebPIE was evaluated against LUBM datasets with up to 100 billion triples and reached a maximum throughput of 2.1M triples/sec [15] on 64 computing nodes for ρdf. The same ruleset was applied by our implementation to the LUBM dataset with a throughput of 2.7M triples/sec. on a single machine. Nevertheless, our implementation is limited regarding the size of the dataset by the availability of main memory on the used computing node and is not able to handle such large datasets. Other MapReduce implementations differ for example in the implemented semantics [3][16].

In [4] an embarrassingly parallel algorithm is introduced, that computes the complete RDFS closure. The evaluation is performed on a cluster with up to 128 parallel processes. The largest dataset used is a LUBM10k/4 (LUBM10000 where only every fourth instance triple was used), which is comparable to a LUBM 2500 dataset. While [4] report a computation time of 291.46 seconds without a global deduplication, our approach performs the complete materialization on a similar dataset using only a single node in 270 seconds and infers only unique triples.

The evaluation showed that our implementation reaches a high throughput and offers a good scaleability for different rulesets with a limited complexity (RDFS and ρdf) on different datasets. Nevertheless, especially for ρdf and RDFS the rule-firing is still a bottleneck and consumes up to 70% of the processing time. The introduced concept to reduce the amount of invalid triple derivation only showed a small increase in speed. In [14] further concepts for deduplication (and thus for reducing invalid triples) are introduced. While the *global strategy* relies on the order of rules and thus is not applicable for our approach, the *local strategy* performs a reduction of duplicates on the GPU. To do so, the inferred triples are sorted such that a comparison with the neighbour-triple can reveal if the triple is already derived during the current processing step. Finally, only the non duplicate triples are transferred back to the host and added to the triple-store. However, to sort and rearrange the triples on the GPU was much slower for our implementation than to derive the triples and validate them using a fast hash-lookup. Rather than focusing on a reduction schema on the GPU we think that a parallel, non blocking triple-store that can be accessed by multiple threads at the same time would be much more efficient. This is particularly the case when using the new levels of parallelization which also allow the use of multiple GPUs.

A further limitation of our approach exists in the need to hold all triples for a fast access during triple-match creation in the main memory. While our implementation may be improvable with respect to memory consumption, the available main memory does limit the size of processable datasets. To overcome this issue, on the one side a distributed approach using multiple nodes equipped with massively parallel hardware might be interesting. On the other side a stream reasoning approach could also be implemented based on the proposed concepts of fast matching.

6 Conclusion

In this paper we introduced new concepts for a further parallelization and workload distribution on top of the results presented in [1], where a rule-based reasoner using massively parallel hardware is presented. The additional host-side parallelization is achieved by taking advantage of the fact, that matches of nodes of one depth in the RETE network can be computed independently. We further introduced the concept of triple-matches, which allow to perform a partitioning of the workload that needs to be computed for a single node. By using triple-matches we also overcome the issue that the maximum size of datasets that can be processed is limited by the onboard-memory of a GPU. Aside from that the partitioning allows a simple way for workload distribution over multiple parallel devices and thus for an additional parallelization. Furthermore, a strategy to reduce the amount of invalid triple derivations was introduces that is able to reduce the number of derived triples by more than 40%.

Future work will focus on different aspects. On the one side we are going to investigate how our approach can benefit from a cluster-based approach to distribute the workload not only to multiple devices, but to multiple computing nodes. This will include the need to reduce the memory usage on a single node. On the other side a faster and possibly parallel triple-store should be part of further developments to achieve a faster rule-firing.

To conclude the paper, we have shown how to scale a rule-based reasoner based on different concepts of workload partitioning and distribution. The proposed concepts where evaluated for datasets with up to 1.1 billion triples and we achieved a throughput of up to 2.7M triples/sec., which is significantly higher than provided by other state of the art reasoners for a single computing node. Thus, our system not only provides a dynamic and flexible way to apply application specific rules to a set of input data, but also a scaleable and fast way with respect to the current state of the art.

References

1. Peters, M., Brink, C., Sachweh, S., Zündorf, A.: Rule-based reasoning on massively parallel hardware. In: 9th International Workshop on Scalable Semantic Web Knowledge Base Systems. (2013) 33–49
2. Urbani, J., Kotoulas, S., Maassen, J., van Harmelen, F., Bal, H.: OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In: Proceedings of the 7th international conference on The Semantic Web. ESWC'10 (2010) 213–227
3. Liu, C., Qi, G., Wang, H., Yu, Y.: Reasoning with large scale ontologies in fuzzy pD* using MapReduce. Computational Intelligence Magazine, IEEE **7**(2) (2012)
4. Weaver, J., Hendler, J.: Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In: Proceedings of the ISWC' 09. (2009)
5. Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. In Raeth, P.G., ed.: Expert systems. (1990) 324–341
6. ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. Web Semantics: Science, Services and Agents on the World Wide Web **3**(2-3) (October 2005) 79–115
7. Muoz, S., Prez, J., Gutierrez, C.: Minimal deductive systems for RDF. In Franconi, E., Kifer, M., May, W., eds.: The Semantic Web: Research and Applications. Volume 4519. (2007) 53–67
8. DBPedia: [Online]. available at http://wiki.dbpedia.org/
9. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for OWL knowledge base systems. Web Semant. (October 2005) 158–182
10. Soma, R., Prasanna, V.: Parallel inferencing for OWL knowledge bases. In: Parallel Processing, 2008. ICPP '08. 37th International Conference on. (2008) 75–82
11. Urbani, J., Kotoulas, S., Oren, E., Harmelen, F.: Scalable distributed reasoning using MapReduce. In: Proceedings of the ISWC' 09. ISWC '09 (2009) 634–649
12. Urbani, J., Margara, A., Jacobs, C., van Harmelen, F., Bal, H.: DynamiTE: Parallel materialization of dynamic RDF data. In: Proceedings of the ISWC' 13. (2013)
13. Wu, Z., Eadon, G., Das, S., Chong, E.I., Kolovski, V., Annamalai, M., Srinivasan, J.: Implementing an inference engine for RDFS/OWL constructs and user-defined rules in Oracle. In: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on. (2008) 1239–1248
14. Heino, N., Pan, J.: RDFS reasoning on massively parallel hardware. In: The Semantic Web ISWC 2012. Volume 7649 of Lecture Notes in Computer Science. (2012) 133–148
15. Urbani, J., Kotoulas, S., Massen, J., van Harmelen, F., Bal, H.: WebPIE: A web-scale parallel inference engine using MapReduce. Web Semantics: Science, Services and Agents on the World Wide Web (0) (2012)
16. Maier, F., Mutharaju, R., Hitzler, P.: Distributed reasoning with EL++ using MapReduce. Technical report, Kno.e.sis Center, Wright State University, Dayton, Ohio (2010)