Rule-based Reasoning on Massively Parallel Hardware

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. In order to enable the semantic web as well as other time critical semantic applications, scaleable reasoning mechanisms are indispensable. To address this issue, in this paper we propose a rule-based reasoning algorithm which explores the highly parallel hardware of modern processors. In contrast to other approaches of parallel reasoning, our algorithm works with rules that can be defined depending on the application scenario and thus is able to apply different semantics. Furthermore we show how vector-based operations can be used to implement a performant match algorithm. We evaluate our approach by applying the ρdf, RDFS and pD* rule sets to different data sets and compare our results with other recent work. The evaluation shows that our approach is up to 9 times faster depending on the rule set and the used ontology and is able for example to apply the ρdf rules to an ontology with 2.2 million triples (and 1.3 million inferred triples) in less than 6 seconds.

Keywords: rule-based reasoning, GPU, parallel reasoning, Rete algorithm

1 Introduction

The use of ontologies is widely spread whether they are used in scientific applications or to enable the semantic web. One key characteristic of ontologies is the possibility to reason about the given data and to create new knowledge by inferring facts that are implicitly given by the ontology. Depending on the size and the structure of the data, the reasoning process may be very resource consuming, especially with regard to the continuously growing amount of data of the semantic web. On the other side, applications using ontologies for example in the field of ambient assisted living [1] [2] or smart spaces [3] [4] [5] may be time critical and need to process incoming data very fast.

Different approaches exist to speedup the reasoning process and to provide scalable solutions for different levels of expressivity, varying from improvements on the reasoning process itself to distributed reasoning over clusters of computational units. Especially the number of approaches using parallel structures has increased in the past few years. The ELK reasoner [6] for example takes advantage of multi-core and multi-processor systems of modern computers to perform OWL EL[3] reasoning. Other approaches rely on a distributed cluster and use the MapReduce framework to perform RDFS and pD* (also know as OWL Horst) [7] reasoning on RDF graphs with millions of triples [8] [9].

While these approaches perform very well for a predefined set of rules that define the strength of the semantics and thus the expressivity of the resulting ontology, the use of a cluster of machines for computation may not always be desirable due to the high costs. In addition the predefined set of rules that is implemented may not always be exactly what is needed for a specific application. In this paper we present an approach which uses the massively parallel hardware of a modern graphic processor unit (GPU) to apply a set of freely defined rules to an RDF-based ontology. While a single core of a GPU has not as much computation power like on a CPU, a GPU provides much more processors that are able to perform simple computation tasks in parallel. Thus it makes sense to exploit this highly parallel hardware and to break down the complex workload into fine grained tasks for parallelisation. Our approach uses an adapted version of the Rete algorithm [10] and thus implements a forward chaining rule engine, which is able for example to materialise the complete finite RDFS closure as well as to apply the pD* rules in a scaleable manner on a single machine. Also this approach is more flexible than most other reasoners, because it is not dedicated to a predefined semantic, we achieve a very high performance due to the parallelisation of the time consuming steps. In addition our approach is not required to be executed on a GPU, but also can be executed on a multicore CPU.

The main contribution of this paper will be to show how the Rete algorithm can be used to reason on ontologies in a highly parallel manner. The use of a rule-engine-based approach allows us to provide a reasoner that is not dedicated to one predefined semantic nor to a specified rule order, and thus can be used for different rule sets of various complexity and semantics. We are also going to introduce a concept for an efficient vector-based match algorithm which is one key factor for the performance of our reasoner, called AMR (act-mobile reasoner). In the next section we are starting with taking a deeper look on the related work on high performance and parallel reasoning. In section 3 we describe the Rete algorithm and show, how it can be used for parallel inferencing on parallel processors. We will also provide more details on the OpenCL[4] programming model and show how this has an impact to our approach. To evaluate our concept we use different rule sets applied to some well known ontologies of different sizes. Finally we are going to discuss our results and give a conclusion.

[3] http://www.w3.org/2007/OWL/wiki/EL

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

2 Related Work

Particularly with regard to large ontologies of hundreds of millions of triples recently the use of clusters for applying finite rules like for RDFS or OWL Horst semantics were a focus of interest in the research community. In [9] and [8] the authors presented WebPie, an inference engine based on the open-source MapReduce implementation Hadoop, which is able to compute RDFS as well as the pD* semantics on data sets containing billions of triples. The parallelisation is achieved by encoding the necessary rules as a set of Map and Reduce operations which are executed in a given order while the distribution of the workload is handled by Hadoop. Other papers propose similar approaches also based on MapReduce differing in the implemented semantics like OWL 2 EL [11] or Fuzzy pD [12]. Another approach relying on multiple computing machines is presented in [13], where a divide-conquer-swap strategy is applied. The input data is stored on a shared location and divided into smaller chunks which are processed by a grid of compute nodes. The results are exchanged between those nodes for further processing. Just like the previous mentioned approaches, the strategy relies on a predefined semantic. Another strategy for parallel reasoning is presented in [14] where the input data is also partitioned and processes running on a cluster computing the finite RDF schema closure for each partition.

Besides the use of a cluster to increase the scaleability, other approaches try to take advantage of the parallel structures of a single machine like available through modern multicore CPUs and GPUs. [15] and [16] for example propose an approach for concurrent classification (TBox) and parallel ABox reasoning for OWL 2 EL. Another interesting reasoning mechanism is presented in [17], where the ρdf vocabulary, which represents a subset of the RDFS rules, is encoded to be applied highly parallel on the graphic processor. While this approach does not only consider ABox or TBox information of an ontology and shows great performance, it still relies on a pre-defined semantic. Nevertheless, this work is most related to our work and will be used for evaluation in section 4.

As outlined by the discussed approaches, many scaleable solutions exists for reasoning on a cluster as well as on a single machine which support different semantics. Nevertheless, as far as we know, there is no scaleable solution using parallel inferencing for a general purpose reasoning process, where the semantic can be defined by the application in terms of simple rules, irrespective of whether these semantics are based on a RDFS or OWL profile or include application specific rules.

3 Parallelising Rule Execution

3.1 OpenCL programming model

OpenCL is a heterogeneous programming framework that allows to develop applications that execute across a range of device types and supports a wide range of parallelism. While the *device* refers to the typically parallel processors like a multicore CPU or a GPU, the *host* can be seen as the outer control logic that prepares the execution of logic on the device. The logic executed on the device is called kernel and thus is that part of an OpenCL program, that typically is executed in parallel. To parallelise an application, the workload needs to be partitioned into small chunks where each chunk can be computed by the same code in parallel. Each chunk is handled by a *work-item* which runs in its own thread and has a unique global identifier which can be used to identify that part of the input data, that shall be processed by a work-item. In addition, work-items are grouped into *work-groups* which have a limited size but can share local memory which can be accessed much faster than global memory. Nevertheless, local memory is very limited such that it needs to be used wisely. To achieve a high performance it is important to have a kernel with as less control structures that might be evaluated by two work-items in a different way as possible, because this would lead to idling threads during the parallel execution.

3.2 The Rete algorithm

What distinguishes the AMR reasoner from the aforementioned parallel reasoners is that we do not know which rules shall be applied to an ontology and thus can not provide dedicated methods that each implement one rule. Thus, we have to implement a generic rule engine which is able to handle ontological data. One widely used algorithm for this complex task was provided by Charles L. Forgy [10], the *Rete Match* algorithm, which is able to find all the objects matching a given pattern. The Rete algorithm builds a network of nodes where each node corresponds to a pattern occurring on the left hand side (LHS) of a rule (that part, that defines the match conditions). Thus, each pattern on the LHS corresponds to a match condition such as $(?p\ \ \text{rdfs:domain}\ \ ?c)$. For each single pattern an alpha node is created, while more than one pattern on the LHS in addition leads to at least one beta node. Thus, a beta node always has more than one pattern. For each node a list of matching objects is stored which is created by propagating the working memory and thus the input data through the network.

Considering the following rules from the RDFS semantic:

$$(?x\ \ ?p\ \ ?y) \rightarrow (?p\ \ \text{rdf:type}\ \ \text{rdf:Property}) \tag{R1}$$

$$(?x\ \ ?p\ \ ?y)\ \ (?p\ \ \text{rdfs:domain}\ \ ?c) \rightarrow (?x\ \ \text{rdf:type}\ \ ?c) \tag{R2}$$

The Rete algorithm would create one alpha node from the left hand side of the first rule R1, and one more alpha node for the second pattern of the second rule R2. Because the first pattern of R2 is equal to the pattern of R1, both rules would share one node. Because R2 consists of two patterns, in addition one beta node would be created connecting the two other alpha nodes. Further considering the following working memory from a simple university example, which contains three triples consisting of a subject, predicate and object $(s,\ p,\ o)$:

$$\text{Bob uni:publishes Paper1} \tag{WM1}$$

$$\text{Alice uni:publishes Paper2} \tag{WM2}$$

$$\text{uni:publishes rdfs:domain Researcher} \tag{WM3}$$

The Rete network resulting by parsing R1 and R2 to alpha- and beta nodes and propagating the working memory through the network can be seen in figure 1.

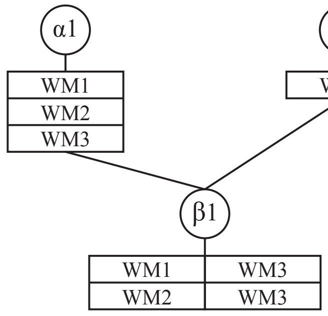


Fig. 1: Rete network for rules R1 and R2 after the working memory has been propagated

The Rete network in figure 1 has three nodes in total, while the final node of R1 is $\alpha 1$ and the final node of R2 is $\beta 1$. A final node always represents a complete rule such that the results stored by that node can be used to fire the corresponding rule. Thus, R1 would fire three times and produce two new triples (and one duplicate) while R2 would fire two times and produce also two new triples. The final working memory would be extended by the following triples:

$$\text{uni:publishes rdf:type rdf:Property} \tag{WM4}$$

$$\text{rdfs:domain rdf:type rdf:Property} \tag{WM5}$$

$$\text{Bob rdf:type Researcher} \tag{WM6}$$

$$\text{Alice rdf:type Researcher} \tag{WM7}$$

The new triples would also be propagated through the network until no new triples are derived and the rule engine has finished his job (fixpoint iteration).

3.3 Definitions

As can be seen from the previous introduction, the Rete algorithm basically performs three steps to apply rules to a working memory: an alpha-match, a beta-match and the rule firing. Before continuing, we have to define some terms that are used throughout this paper:

Definition 1. *An **alpha node** specifies a node of the Rete network that directly corresponds to one pattern of a rule. An alpha node always has exactly one pattern that consists of three (pattern-)terms.*

Definition 2. *A **beta node** specifies a node of the Rete network that has exactly two parents (p1 and p2). The parents in turn may be alpha- or beta nodes. Depending on the position of a beta node in the network, it has a depth >= 1 that is calculated by the longest distance to an alpha node.*

Definition 3. *The **pattern-width** defines the number of terms of the pattern of one node. Thus, an alpha node always has a pattern-width of 3. A beta node whose parents are both alpha nodes has a pattern-width of 6 and so on.*

Definition 4. *The **matches** of a node are referred to as a set A. Accordingly the number of matches is defined by $|A|$.*

Definition 5. ***Match-conditions** are created for each node and define a function $C(m, ...)$ with $m \in A$, that evaluates to true if a triple or a set of triples (in case of a beta node) conform to the pattern of that node.*

3.4 Rete on Parallel Hardware

To parallelise the overall reasoning process, different strategies were proposed by other approaches which essentially consists of data partitioning and rule partitioning [18]. While data partitioning means the dividing of the data into smaller units and the independent processing of each unit, the rule partitioning approach applies only a subset of the overall rules to the complete data set. Both types of parallelisation require a synchronisation of the results and especially the data partitioning approach may produce duplicates. Nevertheless, these drawbacks might be unavoidable in a cluster-based environment where each of the parallel processes runs in an independent environment.

Using the parallel structures on a single computer, a parallelisation can take place on a different level than rule or data partitioning. This is because in the case of a single computer a synchronisation of interim results of the parallel threads can be performed much more efficient by a single host application. Furthermore, to achieve a high performance for example on a GPU it is important to have lots of tasks that can be computed independently and where each task consists only of a small workload. In order to take these considerations into account, our approach parallelises the reasoning process on a deeper level. For alpha-matching, one kernel is executed where for each triple that needs to be considered a single thread (work-item) is created. This thread checks whether the corresponding triple matches one or more of the alpha node patterns and creates a list containing the matching nodes. Thus, each thread needs to iterate over all of the alpha nodes. Finally the resulting list can simply be transformed to create a match-list for each alpha node containing the matching triples. That means that the number of threads that are executed in parallel is equal to the number of triples available for processing.

Because of its complexity, the beta-match is handled in a different way. Just like the rule partitioning approach it would not be desirable to simply execute one thread for each beta node in parallel because of the low number of nodes and the high computation load. On the other side the number of match-steps, that need to be computed heavily rises with the number of matches of the parents of one beta node. Considering the example from figure 1, for $\beta 1$ only 3x1 possibilities needed to be computed. If $\alpha 1$ and $\alpha 2$ both had ten matches, the number of possible combinations would arise to 100. By taking this complexity as well as the fact, that the GPU is able to handle millions of work-items in a very efficient way, into account, it is a natural way to apply the parallelism for the beta-step during this match-algorithm. That means, that for each match of one parent of a beta node a work-item is created which iterates over all matches of the second parent of the beta node. Thus, the number i of work-items is defined by:

$$Def: \quad i = |A_{p1}|, \;\; with\; |A_{p1}| >= |A_{p2}| \tag{1}$$

and the number of iterations j each work-item has to perform by:

$$Def: \quad j = |A_{p2}|, \;\; with\; |A_{p2}| < |A_{p1}| \tag{2}$$

Finally the match-algorithm for a beta node is defined as:

$$C(m_i, m_j), \;\; m_i \in A_{p1}, \; m_j \in A_{p2} \tag{3}$$

where m_i denotes the parallelism and i corresponds to the rank of the thread. By defining $|A_{p1}| >= |A_{p2}|$ and $|A_{p2}| < |A_{p1}|$ we ensure, that the number of parallel threads is at least as high as the number of iterations each thread has to perform, which can be computed more efficient on the GPU. Furthermore the match-algorithm needs to be performed for $A_{p1} \times A_{p2}$.

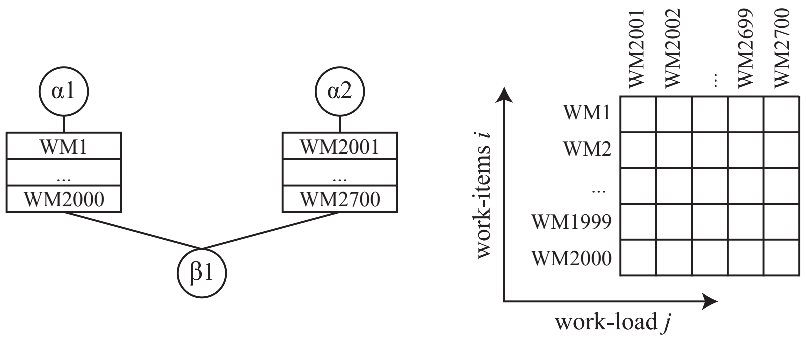


Fig. 2: Parallelise the match process

Figure 2 shows an example where $A_{\beta 1}$ shall be computed. Because $|A_{\alpha 1}| = 2000$ and $|A_{\alpha 2}| = 700$ it follows that $C(m_1, m_2)$ needs to be executed 1.4 million times. To do this, 2000 work-items are created (from each match of $\alpha 1$) where each work-item iterates over 700 matches (from $\alpha 2$) and validates $C(m_1, m_2)$. This task needs to be performed for each beta node, while the beta-match of nodes of the same depth can also run in parallel.

3.5 Applying the OpenCL Programming Model

Programming parallel hardware with OpenCL imposes some restrictions. First of all, strings can not be computed in an efficient way and thus are not suitable for a fast match-algorithm. Thus, all triple terms $s,\ p,\ and\ o$ are transformed into integer values, where each term is mapped to one value. Non literal terms are mapped only once such that there is only one index for each term, even if it is used a multiple times within the ontology. Furthermore, the memory used by a kernel needs to be allocated before kernel execution with the consequence, that each alpha- and beta-step has to be performed twice. The first time, only the number of matches is calculated (called match-count) while for the second execution the number of matches is used to allocate the required memory for all matches and thus the final results can be created. Finally, the overall reasoning process consists of an alpha-count where the resulting triples of an alpha match are calculated, an alpha-match, where the final results of the alpha-step are created and an beta-count and beta-match for each depth (see definition 2 in section 3.3). The last step is to fire the rules using the working memories of the final nodes and to create new triples. Eventually the algorithm iterates over the complete process until no new triples are derived (fixpoint). In addition the beta-count processes as well as the beta-match processes can be executed at the same time for all beta nodes of one depth (non blocking operations). This allows an out-of-order execution where the GPU may define the order of given commands to optimise the throughput and is the reason, why we first start all beta-count and beta-match operations and read the result back in a following loop.

3.6 Vector-based Matching

The most computation intensive task during the reasoning process is the beta-matching. Thus it is essential to speed this task up and make the computation as simple as possible. To achieve this, our approach uses a vector-based operation to check, if a combination of triples matches the pattern of a beta node. Vector-based operations can be computed by a GPU in a very efficient way which is why it is desirable to use them. To do this, first of all we use so called unrolled kernels for rules with up to four patterns (which is for example the max pattern-width occurring in the OWL Horst rule set). This means, that instead of using loops iterating over a defined number of items (where in this case the number of items depends on the number of patterns that a beta node has) we just write the command to execute as often as it is needed. The kernels on the other hand are executed by using a kernel implementation depending on the characteristics of the beta node. This also allows us to exactly know the pattern-width of each parent of a beta node during kernel implementation which is the basis for the vector-based match algorithm.

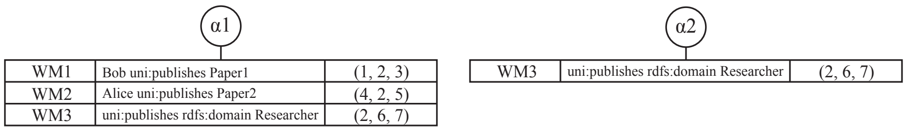
$$(?x\ \ ?p\ \ ?y)\ \ (?p\ \ \text{rdfs:domain}\ \ ?c) \rightarrow (?x\ \ \text{rdf:type}\ \ ?c) \tag{R2}$$

Recapturing rule R2, where the final beta node has two alpha nodes as parents, an unrolled kernel can be executed which assumes a pattern-width of three for both parents. Furthermore it can be seen, that to verify a match only the second term of the first pattern and the first term of the second pattern needs to be checked, such that the value for $?p$ in the first pattern is equal to the value of $?p$ of the second pattern. To do this, in each work-item a vector v_1 is created such that those elements from m_1, that need to be compared to elements of m_2, are placed at the location in v_1 where their corresponding element of m_2 is located. In addition the elements in v_1 need to be negated such that a later performed addition can result in a null vector if the elements of both vectors are equal except of their sign. Besides v_1, another vector u is created which has the same number of elements than v_1 and is filled with elements equal to 1 at those positions, where the second pattern holds an element that needs to be considered to verify a match. All other elements of u are defined as 0. Finally the loop which runs over all matches of A_{p2} only has to create a vector v_2 which holds all elements of m_2. This vector is used to verify for a match as follows:

$$(v_2 * u) + v_1 \tag{4}$$

This operation is performed in a component based manner (meaning that a concurrent, component based multiplication as well as a component based addition is performed) and results in a null-vector, if a match was found. Otherwise, at least one element of the resulting vector is unequal to 0. Furthermore only a simple operation and a minimum of data transfer is necessary within the inner loop of a kernel, which allows a very efficient execution even for a large $|A_{p2}|$.

To continue the example illustrated in figure 1 the working memory of $\alpha1$ and $\alpha2$ looks like depicted in figure 3. To improve the readability, not only the



WM1	Bob uni:publishes Paper1	(1, 2, 3)
WM2	Alice uni:publishes Paper2	(4, 2, 5)
WM3	uni:publishes rdfs:domain Researcher	(2, 6, 7)

WM3	uni:publishes rdfs:domain Researcher	(2, 6, 7)

Fig. 3: Working memory of $\alpha1$ and $\alpha2$

working memory reference (row 1) is given, but also the data (row 2) as well as the internal representation of that data (row 3). The internal representation, like described before, is a mapping of each subject, predicate and object to an integer value, which is used for computation. Based on the aforementioned description, three parallel threads would be executed to calculated $A_{\beta1}$. Because every item in the working memory of $\alpha1$ matches the pattern ($?x$ $?p$ $?y$) and every item in the working memory of $\alpha2$ matches the pattern ($?p$ rdfs:domain $?c$) (see rule R2) the only calculation necessary to see, if a combination of an $\alpha1$ and an $\alpha2$ match also is a match of the beta node $\beta1$, is to compare the corresponding $?p$ values. To do this, in each thread a constant vector v_1 is created using the corresponding match of $\alpha1$. Looking at the first thread, v_1 holds the negated

?p-value of the WM1 element, which is positioned at the first vector component, because the corresponding ?p-element of the $\alpha 2$ match is also positioned at the first element. This results in $v_1 = (-2, 0, 0)$. Because only the first component of the vectors need to be considered for R2 (only the ?p elements have to be compared), the vector u is created as $u = (1, 0, 0)$. Now, for each element of $A_{\alpha 2}$ a vector v_2 is created which simply holds the numeric representation of the corresponding triple such that v_2 results in $v_2 = (2, 6, 7)$. The final (component based) calculation is as follows:

$$\begin{pmatrix}2\\6\\7\end{pmatrix} * \begin{pmatrix}1\\0\\0\end{pmatrix} + \begin{pmatrix}-2\\0\\0\end{pmatrix} = \begin{pmatrix}0\\0\\0\end{pmatrix} \tag{5}$$

Using the same calculation to match WM3 against WM3 (WM3 is a match of $\alpha 1$ as well as of $\alpha 2$) would not result in a null vector and thus would not be a match:

$$\begin{pmatrix}2\\6\\7\end{pmatrix} * \begin{pmatrix}1\\0\\0\end{pmatrix} + \begin{pmatrix}-6\\0\\0\end{pmatrix} \neq \begin{pmatrix}0\\0\\0\end{pmatrix} \tag{6}$$

In addition to the unrolled and vector based kernels, we also implemented kernels that can be used with a pattern-width larger than 4 which allows to write even more complex rules with an arbitrary size.

4 Evaluation

4.1 ρdf, RDFS and OWL Horst

Because our approach is able to handle a given set of rules independent of the semantic that those rules belong to, we used three different rule sets with a varying complexity, which often were implemented by other reasoners in a manual way. The ρdf vocabulary [19] is a simplified version of the RDFS semantic which consists of all rules of RDFS with at least two rule body-terms. This semantic imposes a more efficient reasoning, while the results of the missing rules are supposed to be created on the fly by a reasoner, if resources are queried. These rules were also implemented by the MapReduce-based approach presented in [9] as well as the by the GPU based approach proposed in [17]. Due to space restrictions, we also refer to [17] for the ρdf rules.

The second rule set consists of the complete RDFS rules like defined by the W3C[5]. This rule set consists of 13 rules with one or two antecedents and is used in several other publications for evaluation purpose [14] [8]. The last rule set formally known as pD* was proposed by Herman J. ter Horst which incorporates RDFS and D entailment and extends these semantics with some basic support for OWL [7]. It provides a complete set of entailment rules and has become a promising ontology language for the semantic web because of its

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

expressiveness on the one side and its relatively low computation complexity on the other side. For a complete overview of the 22 rules (some of them can be combined as they share the same antecedents resulting in 16 rules) we refer due to space restrictions to [8].

4.2 Test Environment

To achieve comparable results we choose ontologies with various sizes that were already used to evaluate other approaches. Thus, we are using the Vicodi[6] ontology which is an ontology of European history used for semantical indexing of historical documents [20]. The TBox is of a moderate size while the ABox contains a large number of instances. In total, the ontology consists of 146,280 triples and thus is compared to our second ontology, known from the Lehigh University benchmark (LUBM)[7], a small sized ontology. LUBM is a benchmark ontology and defines an TBox for a university scenario. A generator allows to generate university data sets while the number of universities that are created can be defined as an input. Thus, we created 3 LUBM data sets with 268,794 triples which contains two universities (LUBM2), a LUBM5 ontology with 727,265 triples and a LUBM10 ontology with 1,480,366 triples. To use another large ontology we used the DBPedia[8] 3.7 which is a lightweight ontology containing structured data extracted from Wikipedia. For this data set we used a similar setup like described by [17] containing the DBpedia Ontology, Infobox Types and Infobox Properties. We also limited the size of the data set in a similar way by scaling the instance triples by $1/8^{th}$, $1/16^{th}$, and $1/32^{nd}$ of the original size.

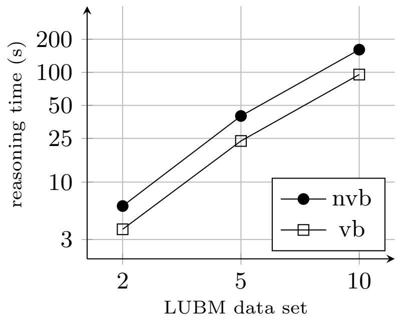
Our implementation is Java based and integrates with Jena[9] applications. Thus, we use the Jena framework to parse the ontologies and create our own data structures for the reasoning process by reading an ontology graph from Jena. To be able to use OpenCL from our Java application, we use jocl[10] as Java bindings. The tests are performed on a work station with a 2.0 GHz Intel Xeon processor with 6 cores and an AMD 7970 gaming graphic card with 3GB of memory running an Ubuntu 12.04. In order to compare our results to other approaches, we performed the same tests with the ρdf rule set with the GPU based reasoner proposed in [17]. In the following we refer to that reasoner as grdfs reasoner. Other parallel reasoners also implementing the complete RDFS or OWL Horst rules on a single computer considering the ABox as well as the TBox were not available. We also run our experiment using the non-vector-based kernel and compare the results with the vector-based version. For each experiment the total time except data transformation (i.e. loading the Jena-graph to AMR and file parsing for grdfs) were measured, which also includes the non parallel rule firing. A dedicated kernel execution time on the GPU is not given due to the

[6] http://www.vicodi.org/about.htm
[7] http://swat.cse.lehigh.edu/projects/lubm/
[8] http://dbpedia.org/About
[9] http://jena.apache.org/
[10] http://www.jocl.org/

execution of multiple kernels which are launched asynchronously such that no precise information are available. Furthermore each test was performed ten times while the average is presented.

4.3 Results and Comparison

The first experiment shows the impact of using the vector-based operations during the beta-match. Therefore we used all three LUBM data sets and executed it using the pD* rule set with the naive implementation of the kernel and with the vector-based kernel. Notice that both kernels are unrolled and the only difference is, that the second kernel uses vector operations instead of calculating each element in a single step. We choose the pD* vocabulary because it is the most computation intensive one of the three used rule sets.



LUBM	triples	entailed	nvb	vb	speedup
2	268,794	38,584	6.05 s	3.72 s	1.63
5	727,265	102,618	40.01 s	23.69 s	1.69
10	1,480,366	207,677	160.80 s	95.39 s	1.69

Fig. 4: Comparison of non-vector-based operations (nvb) and vector-based (vb) operations

The results in figure 4 show that the vector-based kernel provides a constantly better performance. The calculation using the vector-based method is round about 1.7 times faster than using the naive implementation which simply applies the comparison for two matches of the corresponding parent-nodes using single operations. On a different hardware (MacBook Pro) even a speedup of more than 4 could be measured using the vector-based operation.

The next test shall show the scaleability of our approach. Therefore we use the Vicodi ontology as well as the LUBM2 ontology and run our algorithm on the CPU of our test environment. The CPU has less cores than the GPU and is not that fast, but OpenCL allows us to use only a defined number of cores of the CPU. This way it is possible to show the speedup which is achieved by increasing the number of used cores. Because the used CPU has 6 cores where each core can run 2 threads through hyper-threading (resulting in 12 virtual cores), we applied the pD* vocabulary to the input data using 1 to 12 cores.

As can be seen from figure 5 the speedup nearly doubles with a doubling of the number of used processors for both datasets until 6 cores are used. The

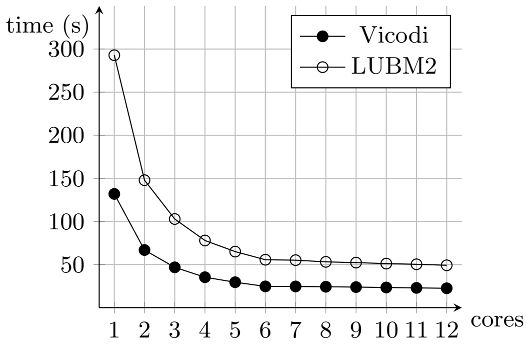


Fig. 5: Scaleability test using the Vicodi and the LUBM2 ontology on the CPU

other 6 cores still contribute to a better performance, but the impact is not that reasonable anymore. We assume that this is because two (virtual) cores always share some resources on one processor and thus do not provide such a speedup as is would be achieved if each core had physically exclusive resources.

Finally we want to compare our results to other approaches. For this we performed a set of tests with the GPU based grdfs reasoner proposed in [17] as well as with AMR. Both reasoners used the same hardware like described before. Because the grdfs reasoner implements the ρdf semantic, the tests were only performed using the corresponding rules. While we can not prove that our implementation works correct, we still get exactly the same results using the general purpose rule engine provided by the Jena framework for ontologies with a limited size which that rule engine is able to handle. The largest ontology we were able to test with Jena using the ρdf vocabulary was the LUBM2 ontology with 268,794 triples (and 146 entailed triples), which took about 27 minutes, while our approach entails exactly the same triples in about 270 ms. We also tested for example the Vicodi ontology using the Jena framework and the RDFS vocabulary which took about 12 minutes and inferred 127,886 triples to a total of 274,233. Our approach again provides exactly the same results using the GPU in less than a second. The final results of our tests including parallel and serial work (the complete reasoning process excluding parsing) are listed in table 1.

	triples	AMR	grdfs	entailed AMR	entailed grdfs	speedup
LUBM2	268,794	276 ms	1,383 ms	146	22	5.02
LUBM5	727,265	447 ms	3,153 ms	146	22	7.05
LUBM10	1,480,366	676 ms	6,207 ms	146	22	9.22
DBPedia 1/32	1,087,364	3,061 ms	7,554 ms	1,087,364	1,085,309	2.47
DBPedia 1/16	2,276,510	5,931 ms	14,681 ms	1,936,950	1,934,887	2.48
DBPedia 1/8	4,523,729	10,954 ms	27,739 ms	3,083,513	3,081,433	2.53

Table 1: Reasoning time for different data sets using AMR and grdfs reasoner

The experiment shows that our approach provides a speedup of a factor of up to 9.2 compared to the grdfs reasoner. While the speedup for the LUBM data sets is more significant than for the DBPedia data sets, it is still more than two times faster. The different speedup factor results from the fact that using the DBPedia data sets many new triples are inferred, such that up to 60% of the reasoning time of the AMR reasoner is needed for the serial implemented rule firing, which also includes operations like dictionary lookup to not infer duplicate triples.

Further results from other work for comparison can be used for example from [14], where the authors evaluated an approach to parallelise the RDFS closure using an Opteron blade cluster, each server in the cluster having two dual-core 2.6 GHz AMD Opteron processors. The LUBM10k/1024 data set from that paper has a slightly smaller size and a similar complexity like the LUBM10 data set used for this paper. While the approach from [14] took about 2 seconds in total to calculate the closure using a cluster of 128 cores, our approach calculates the RDFS closure on a single machine in about 3 seconds. On a MacBook Pro with a Core i7 processor, which has a less powerful GPU but due to the CPU a faster architecture for serial calculations, the same test could even be finished in less than 2.6 seconds using the AMR reasoner. Nevertheless, the approach from [14] is able to handle much larger data sets.

5 Discussion and Future Work

The results in section 4.3 show that our approach offers a good and scaleable performance using a single computer, also for data sets with millions of triples. Nevertheless, there are still restrictions regarding the size of an ontology. On the one hand for a performant execution the main memory of the host computer needs to be large enough to hold the complete ontology as well as inferred matches and data structures that are used for the rule execution. On the other hand the use of integers for the created index structures limits the number of processable triples. This limitation is even stronger regarding the matches-arrays of the single nodes which easily needs to hold a multiple of elements as triples are available. To overcome this issues, the use of 64bit datatypes as well as appropriate collection types to hold the triples and matches should be considered. In addition the use of collection oriented matching like describe in [21] could be considered, where matches are calculated and stored in a collection-oriented way instead of using single tuples. Furthermore a partitioning strategy could be implemented that allows to distribute the workload of large ontologies over multiple GPUs as well as over multiple machines. Thus, a combination of the cluster-based approach used in [14] and the low level parallelisation like described in this paper might be an interesting approach. Another optimisation might be possible by parallelising the rule firing, too, which will require thread safe data structures and a concept to detect duplicates.

Besides optimisations regarding the performance and the ability to handle larger data sets, in the future we are also going to investigate how we can extend the functionality of our rule-based system to also support operants like $greaterThan(?x, ?y)$ within a rule body. This way our system would offer much more flexibility for scenarios with application specific rules like used in different kinds of smart environments like [2] [5].

6 Conclusion

In the past most of the approaches to parallelise the reasoning process have focused on distributing the workload over multiple machines to use a large number of processors. Only a few approaches already considered the use of the parallel structures available on a single machine. All approaches have in common, that they implement a defined set of rules and can not be configured in an application specific way. In this paper we proposed a rule-based approach that is independent from a specific semantic and uses the parallel structures of modern CPUs as well as of GPUs. The high performance is achieved by parallelising the Rete algorithm and breaking the match-steps into fine grained tasks which can be computed highly parallel. We also introduced a vector-based operation to compute the beta matches, which easily doubles the performance of the algorithm running on a GPU. Finally our results show, that the approach scales well with the number of used cores and can apply a set of rules to an ontology in a very performant way. Thus the parallelisation of a generic rule-based approach to apply rules on ontological data can be very efficient, if the workload is partitioned into an adequate number of units which can be computed highly parallel.

References

1. Ausín, D., Castanedo, F., López-de Ipiña, D.: Benchmarking results of semantic reasoners applied to an ambient assisted living environment. In: Proceedings of the 10th international smart homes and health telematics conference on Impact Ananlysis of Solutions for Chronic Disease Prevention and Management. ICOST'12, Berlin, Heidelberg, Springer-Verlag (2012) 282–285
2. Agostini, A., Bettini, C., Riboni, D.: A performance evaluation of ontology-based context reasoning. In: Pervasive Computing and Communications Workshops, 2007. PerCom Workshops '07. Fifth Annual IEEE International Conference on. (2007) 3–8
3. Pantsar-Syvaniemi, S., Simula, K., Ovaska, E.: Context-awareness in smart spaces. In: Computers and Communications (ISCC), 2010 IEEE Symposium on. (2010) 1023–1028
4. Reinisch, C., Kofler, M., Kastner, W.: Thinkhome: A smart home as digital ecosystem. In: Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on. (2010) 256–261
5. Peters, M., Brink, C., Sachweh, S., Zündorf, A.: Performance considerations in ontology based ambient intelligence architectures. In: Proceedings of the 4th International Symposium on Ambient Intelligence. (2013)
6. Kazakov, Y., Krötzsch, M., Simančík, F.: ELK: a reasoner for OWL EL ontologies. System description, University of Oxford. In: Technical Report (2012)

7. 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
8. 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: research and Applications - Volume Part I. ESWC'10, Berlin, Heidelberg, Springer-Verlag (2010) 213–227
9. Urbani, J., Kotoulas, S., Oren, E., Harmelen, F.: Scalable distributed reasoning using mapreduce. In: Proceedings of the 8th International Semantic Web Conference. ISWC '09, Berlin, Heidelberg, Springer-Verlag (2009) 634–649
10. Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. In Raeth, P.G., ed.: Expert systems. IEEE Computer Society Press, Los Alamitos, CA, USA (1990) 324–341
11. Maier, F., Mutharaju, R., Hitzler, P.: Distributed reasoning with EL++ using mapreduce. Technical report, Kno.e.sis Center, Wright State University, Dayton, Ohio (2010)
12. 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) 54–66
13. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: A platform for large-scale analysis of semantic web data. In: Proceedings of the WebScience '09, Society On-Line (2009)
14. Weaver, J., Hendler, J.: Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In Bernstein, A., Karger, D., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K., eds.: The Semantic Web - ISWC 2009. Volume 5823 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 682–697
15. Kazakov, Y., Krötzsch, M., Simancík, F.: Concurrent classification of EL ontologies. In: Proceedings of the 10th international conference on The semantic web - Volume Part I. ISWC'11, Berlin, Heidelberg, Springer-Verlag (2011) 305–320
16. Ren, Y., Pan, J.Z., Lee, K.: Parallel abox reasoning of EL ontologies. In: Proceedings of the 2011 joint international conference on The Semantic Web. JIST'11, Berlin, Heidelberg, Springer-Verlag (2012) 17–32
17. Heino, N., Pan, J.: RDFS reasoning on massively parallel hardware. In: The Semantic Web ISWC 2012. Volume 7649 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 133–148
18. Soma, R., Prasanna, V.: Parallel inferencing for OWL knowledge bases. In: Parallel Processing, 2008. ICPP '08. 37th International Conference on. (2008) 75–82
19. 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 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 53–67
20. Nagypl, G., Motik, B.: A fuzzy model for representing uncertain, subjective, and vague temporal knowledge in ontologies. In Meersman, R., Tari, Z., Schmidt, D., eds.: On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE. Lecture Notes in Computer Science. Springer Berlin Heidelberg (2003) 906–923
21. Acharya, A., Tambe, M.: Collection oriented match. In: Proceedings of the second international conference on Information and knowledge management. CIKM '93 (1993) 516–526