RDFS Reasoning on Massively Parallel Hardware

Norman Heino[1], Jeff Z. Pan[2]

[1] Agile Knowledge Engineering and Semantic Web (AKSW)
Department of Computer Science
Leipzig University, Johannisgasse 26, 04105 Leipzig, Germany
`heino@informatik.uni.leipzig.de`
[2] Department of Computing Science, University of Aberdeen, Aberdeen AB24 3UE, UK,
`jeff.z.pan@abdn.ac.uk`

Abstract. Recent developments in hardware have shown an increase in parallelism as opposed to clock rates. In order to fully exploit these new avenues of performance improvement, computationally expensive workloads have to be expressed in a way that allows for fine-grained parallelism. In this paper, we address the problem of describing RDFS entailment in such a way. Different from previous work on parallel RDFS reasoning, we assume a shared memory architecture. We analyze the problem of duplicates that naturally occur in RDFS reasoning and develop strategies towards its mitigation, exploiting all levels of our architecture. We implement and evaluate our approach on two real-world datasets and study its performance characteristics on different levels of parallelization. We conclude that RDFS entailment lends itself well to parallelization but can benefit even more from careful optimizations that take into account intricacies of modern parallel hardware.

1 Introduction

Reasoning is an important aspect of the Semantic Web vision. It is used for making explicit previously only implicit knowledge, thus making it available to non-reasoning query engines and as a means for consistency checking during ontology engineering and applications. Applied to large amounts of data, reasoning has been computationally demanding, even if restricted to less expressive RDFS entailment. The idea of concurrent reasoning on parallel hardware has therefore been of research interest for quite some time. Previous work on parallelizing reasoning has focused on cluster-based implementations on top of *shared nothing* architectures that require significant expenses in hardware costs [7,19,21]. Since scalability of parallel RDFS reasoning has been shown by such work, looking at other parallel architectures seems a promising approach.

Over the last years the number of CPU cores available in commodity hardware has increased while the clock rates have not changed much. At the same time graphics processing units (GPUs) have been successfully used for general purpose computing tasks [11]. GPUs provide an even higher level of parallelism by reducing the complexity of a single compute unit and are thus referred to as *massively parallel* hardware. In order to exploit such high levels of parallelism provided by modern hardware, it is essential to devise algorithms in such a way that fine-grained parallelisms become possible.

In this paper, we consider the problem of parallel RDFS reasoning on massively parallel hardware. In contrast to work on cluster-based implementations, we assume a *shared memory* architecture as is found on such hardware and exposed by modern parallel programming frameworks like CUDA[3] or OpenCL[4]. Our goal is to devise and implement an approach that is agnostic of the actual hardware parallelism (i. e. number of cores) and thus able to exploit any degree of parallelism found.

We show that applying the same principles as used in cluster-based approaches in our architecture often incurs a performance impairment, due to massive amounts of duplicates generated by naïve parallel application of RDFS entailment rules. To better understand the nature and origin of those duplicates, we study the problem on two different datasets. We derive two approaches that make use of shared memory in order to prevent duplicate triples from being materialized. Our implementation is based on the OpenCL framework and can thus be used on a wide range of devices, including multicore CPUs and modern GPUs. We evaluate our system on two real-world datasets in the range of tens of millions of triples.

The remainder of this paper is structured as follows. We give an introduction to both classical and parallel RDFS reasoning and discuss related work in Section 2. We describe our approach and its implementation in Section 3 and report on experimental results in Section 4. We conclude and give an outlook on future work in Section 5.

2   Background

2.1   RDFS Entailment

The W3C recommendation on RDF semantics defines a vocabulary with special meaning the interpretation of which can give rise to new triples [5]. In addition, a set of rules are presented whose repeated application is said to yield the RDFS closure of an RDF graph. In this paper, we consider a subset of the RDFS vocabulary that has been shown to capture 'the essence' of RDFS, called the ρdf vocabulary [9]. Table 1 shows those RDFS rules that produce the closure for the ρdf vocabulary. Note that these rules contain all the rules from the RDFS vocabulary that have at least two antecedents. We ignore rules with only one antecedent since, as was already noted in [19], their (trivial) entailments can be computed easily in a single pass over the data. Since non-authorative statements that use resources from the RDF or RDFS vocabulary in the subject position could change the meaning of the vocabulary we ignore those as well. As in previous publications, all figures given in this work have been determined without using RDFS axiomatic triples. However, the system described is capable of including those axiomatic triples that make use of the ρdf vocabulary subset. In particular, the infinite number of axiomatic container membership properties are not included since ρdf does not contain such properties.

2.2   Parallelizing RDFS entailment

A more detailed inspection of the rules presented in Table 1 reveals two classes. Roughly speaking, there are (i) rules that operate solely on schema triples and (ii) rules that operate on a schema triple and an instance triple.

---

[3] `http://www.nvidia.com/object/cuda_home_new.html`
[4] `http://www.khronos.org/opencl/`

Table 1: Subset of the RDFS entailment rules with two antecedents.

(5)	p `rdfs:subPropertyOf` q	& q `rdfs:subPropertyOf` r	$\implies p$ `rdfs:subPropertyOf` r
(11)	C `rdfs:subClassOf` D	& D `rdfs:subClassOf` E	$\implies C$ `rdfs:subClassOf` E
(2)	$s\ p\ o$	& p `rdfs:domain` D	$\implies s$ `rdf:type` D
(3)	$s\ p\ o$	& p `rdfs:range` R	$\implies o$ `rdf:type` R
(7)	$s\ p\ o$	& p `rdfs:subPropertyOf` $q \implies s\ q\ o$	
(9)	s `rdf:type` B	& B `rdfs:subClassOf` C	$\implies s$ `rdf:type` C

Rules of the first kind compute the *transitive closure* of a property. In Table 1 those rules are shown in the upper section (rules (5) and (11)).

The second kind of rule is shown in the lower part of Table 1 (rules (2), (3), (7), and (9)). We refer to each of these rules as a *join rule*, since it essentially computes a database join between instance and schema triples with the join attribute being the subject of the schema triple and either the property or the object of an instance triple.

Since no rules depend on two instance triples, each can be applied to different instance triples in parallel. Such a *data parallel* task is well suited to GPU workloads due to their ability of efficiently scheduling large numbers of threads. Communication of entailment results is then only necessary between application of different rules. The application of each join rule can thus be considered an *embarrassingly parallel*[5] problem.

```
1  @base <http://example.com/> .
2  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3
4  <p> rdfs:domain <C> .
5  <C> rdfs:subClassOf <D> .
6  <A> <p> "O1", "O2", "O3" .
```

Listing 1: RDF graph that produces duplicates when rules (2) and (9) are applied to it.

Treating RDFS reasoning as such a problem, however, can lead to suboptimal performance, since RDFS entailment has an inherent tendency towards producing duplicate triples. To see that, consider the RDF graph in Listing 1. When applying rule (2) to it, each of the triples in line 6 together with the one in line 4 would entail the same triple $\langle A, \texttt{rdf:type}, C \rangle$. Applying rule (9) thereafter would entail the triple $\langle A, \texttt{rdf:type}, D \rangle$, again three times. Since, in this case, duplicates are generated by the same rule we refer to them as *local* duplicates. Another kind of duplicate can be

---

[5] An embarrassingly parallel problem refers to a problem where very little or no communication is needed.

generated by entailing triples that have already been entailed by a previous rule. Those duplicates we refer to as *global* duplicates.

The duplicate problem has been acknowledged in previous work. Urbani et al., for example, combine rules (2) and (3) and cluster instance triples by equality of subject and object, so as to not entail the same `rdf:type` statements via `rdfs:domain` or `rdfs:range`. The full extent of the problem is not addressed in their work, however, as they do not materialize triples. In Subsection 3.4 we analyze the ramifications that this incurs on performance and discuss approaches to mitigate the problem.

2.3 OpenCL Programming Model

OpenCL is a vendor-agnostic programming model and API for parallel programming. In OpenCL, parallel workloads are expressed in the form of *compute kernels* that are submitted for execution on a parallel device. Since the way in which kernels are written allows for fine-grained parallelism, such devices can range from manycore CPUs to massively parallel GPUs. Execution and submission of kernels are controlled by a host program, which usually runs on the CPU. An instance of a compute kernel (run on a compute device) is called a *work item* or simply a thread[6]. Work items are combined into *work group*s, where those items have access to low-latency shared memory and the ability to synchronize load/store operations using memory barriers. Each work item is assigned a globally (among all work items) and locally (within a work group) unique identifier which also imposes a scheduling order. Those can be used to compute memory offsets for load and store operations. Data transfer between the host system and compute units is done via global memory to which all work items have access, albeit with higher latency. OpenCL provides no means for memory allocation on the device. This means that all output buffers must be allocated on the host *before* executing a kernel. For workloads with a dynamic result size (like RDFS reasoning) this implies a problem. Section 3.3 discusses our approach to this problem.

In this paper we refer to *device* when we talk about the hardware component on which a compute kernel is executed, while we refer to controlling code as being executed on the *host*. Note that in the OpenCL programming model both host and device can refer to the same CPU.

2.4 Related work

Related work on RDFS entailment has primarily focused on distributing the reasoning workload on a cluster of compute nodes.

An embarrassingly parallel implementation of RDFS entailment over a compute cluster is presented in [21]. Their work uses an iterative fixpoint approach, applying each RDFS rule to triples until no new inferences can be made. Triples generated by different nodes are written to separate files. Thus, duplicates are not detected and consume both memory bandwidth and storage space.

An interesting approach to parallel RDFS reasoning using distributed hash tables is presented by Kaoudi et al. [7]. They give algorithms for both forward and backward chaining and an analytical cost model for querying and storage.

---

[6] In this work we use the terms *work item* and *thread* interchangeably.

Oren et al. describe a distributed RDFS reasoning system on a peer network [13]. Their entailment is asymptotically complete since only a subset of generated triples are forwarded to other nodes. Duplicate elimination is performed on a node determined by the hash value of a triple. Each node keeps a local bloom filter with all seen triples and deletes subsequent triples that hash to the same positions. This approach is incompatible with our complete reasoning implementation: bloom filters operate with a controllable false positive rate. A triple that is detected as being a duplicate might actually be a false positive and is thus erroneously deleted.

Urbani et al. describe a parallel implementation of the RDFS rule set using the MapReduce framework [19]. They introduce a topological order of the RDFS rules such that there is no need for fixpoint iteration. Our work in principle is based on this approach by using the same rule ordering (see Section 3.3). However, they do not materialize triples and thus do not encounter the duplicate detection problem.

The most recent work on single-node RDFS entailment is found in [3]. In this work, RDFS entailment rules for annotated data as described in [17] are implemented on top of PostgreSQL. Furthermore, idempotency of rules (5) and (11) is shown. Since their result can easily be extended to classical RDFS entailment (and thus ρdf), we make use of that result as well. In difference to this work, our implementation operates in main memory but has no problem dealing with datasets even bigger than those used by the authors to evaluate their approach.

Rule-based parallel reasoners for the OWL 2 EL profile have been presented for TBox [8] and ABox reasoning [14]. Both works rely on a shared queue for centralized task management. In our work, tasks are described in a much more fine-grained manner and global synchronization is only necessary during kernel passes.

Other related work deals with specialized algorithms for GPU computing. The parallel prefix sum (or scan) over a vector of elements computes for each index i the sum of elements with indexes $0 \ldots i-1$ (exclusive) or $0 \ldots i$ (inclusive). Our scan implementation is based on the work of Sengupta et al. [16], while our sorting algorithm draws inspiration from Satish et al. [15].

3   Approach

In this section we describe our approach and its implementation in detail. We explain how data is prepared and how it is stored. We then describe how each of the ρdf rules is implemented in our system. Subsection 3.4 discusses problems we encountered with regard to duplicate triples and how we address them.

3.1   Data representation

Parsing and preparing the data are performed in serial which therefore tends to dominate the process. We use the Turtle parser implementation from RDF-3X [10] with a few modifications. All string values (i. e. URIs and literal objects) are encoded to 64-bit integers. This serves three purposes:

– it guarantees that each value is of fixed length, simplifying the implementation of data structures,

– comparing integers is much faster than comparing strings,
– strings are on average longer than 8 bytes which means we essentially perform data compression, thus lowering the required memory bandwidth.

The dictionary is stored in a disk file which is mapped into memory using the `mmap` system call. The file contains buckets storing the literal identifier along with character data and the address of potential overflow buckets. In order to save space, we doubly hash dictionary entries: A hash value is computed for each literal string to be stored in the dictionary. This hash is used as an index into a second hash table that associates it with an offset into the dictionary file. Since literal identifiers are consecutive integers, the reverse index, which maps literal identifiers to string values, is just a vector of file offsets with the id being the vector index (direct addressing). For entries that are less than a virtual memory page size in length, we ensure they do not span a page boundary: If the entry is larger than the space left on the current page (4096 bytes on most systems) we place it entirely on the next page.

We use the 48 least significant bits (i. e. bits 0–47) for value encoding and the 16 most significant bits (bits 48–63) for storing associated metadata (e. g. whether the encoded value is a literal or a blank node). This is needed since RDFS rules as presented in [5] are incomplete if applied to standard RDF graphs [18]. To solve this problem we internally store a generalized RDF graph (i. e. we allow literals in subject position as well as blank nodes in predicate position). For RDF output, however, we need to be able to detect those non-RDF triples.

3.2 Data storage and indexing

Data is stored in memory in STL[7] data structures. We use one `std:vector` of terms storing a column of RDF terms which is reminiscent to what column-oriented database systems store [1]. In other words, we keep a separate vector for all subjects, properties, and objects, enabling us to quickly iterate over the attributes of adjacent triples which is a common operation when preparing data for the reasoning stage. In addition, a fourth vector holds flags for each triple; e. g.., entailed triples are stored with an *entailed flag* which allows us to easily determine those.

The index into these vectors is kept in a hash table which is indexed by triple. An amortized constant-time hash lookup is hence sufficient to determine whether a triple has already been stored or not. The cost for this lookup is about half of the cost for actually storing the triple.

The storage layer also implements the iterator concept known from STL containers. A triple iterator can be used to retrieve all stored triples. In order to produce entailed triples only, a special iterator is provided that skips over all triples whose entailed flag is not set.

3.3 Sound and complete rule implementation

As previously noted, our algorithm is based on the rule ordering in [19] with a few differences. Due to each rule being executed in a separate kernel run (pass), our approach

---

[7] Standard Template Library–part of the C++ standard library

involves an implicit *global synchronization* step that is inherent to submitting a kernel to an OpenCL device. Hence all entailments created during one pass are seen by all subsequent ones. In particular, if a schema triple is created during a rule application, it will be used as such by subsequent rules. In addition, we perform a fixpoint iteration over rules (5) and (7). This is necessary since rule (7) is able to produce arbitrary triples, including schema triples, and is itself dependent on rule (5). Since it is possible to extend the RDFS vocabulary with custom properties, the fixpoint iteration is necessary to materialize all schema triples before applying other rules. Figure 1 depicts our approach with its four passes as well as synchronization steps in between.
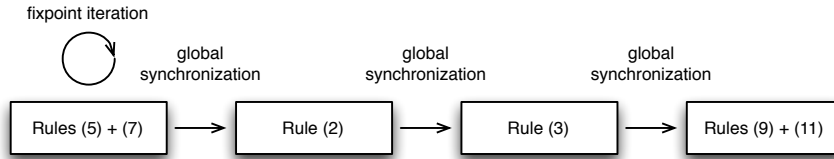

Fig. 1: Passes for computing the subset of the RDFS rules considered in this work.

Computing transitive closure-based rules Parallel algorithms for computing the transitive closure of a graph are based on boolean matrix multiplication as proposed by Warshall [20]. Due to its regular access pattern, it maps easily to the OpenCL memory model. Transitive property hierarchies on RDF graphs, however, tend to be very sparse. In YAGO2 Core, for instance, the number of vertices taking part in the `rdfs:subClassOf` relation is 365,419, while the number of triples using that property in the full closure is about 3.4 million. Representing such a graph in a quadratic matrix is wasteful since most entries will be zero. In YAGO2 Core, it is also infeasible because storing the adjacency matrix of 365,419 vertices would require almost 16 GiB, if each entry is compressed to a single bit.

A space-efficient serial algorithm for calculating the transitive closure of a graph was presented by Nuutila [12]. We tried the implementation found in the Boost Graph Library[8] but it was unable to cope with the graph size from YAGO2 Core. We thus provide our own implementation of that algorithm which is computed serially on the host. A parallel implementation of the algorithm from [12] is beyond the scope of this paper and will be reserved for future work.

Computing join rules In our parallel implementation, each thread is assigned a single instance triple based on its global identifier. Thus, a join rule needs to find a matching subject of a schema triple.

A very efficient join algorithm in database systems is known as the hash join [4]. Hash joins are typically used in cases where a large relation must be joined with a

---

[8] http://www.boost.org/libs/graph/

smaller one. A hash table is populated with the values of the smaller relation. For each value of the large relation, a simple hash lookup can determine whether there is a match. Once a matching schema subject has been found, the objects of all schema triples for that particular rule can be used to materialize new triples.

A hash-join implementation of the process works as follows. Consider rule (9) from Table 1. The object (B) of a given triple is hashed and matched against all `rdfs: subClassOf` schema triples. If a match is found all of its successors (C) become objects of new triples that need to be written to the output vector.

In the OpenCL computing model, this is, however, impossible. All buffers for storing results must be allocated on the host before kernel invocation. We therefore compute join rules in two passes. During the first pass, each thread performs the join with schema triples as described above. Instead of materializing new triples, it just writes the *number of results* it would produce and the index of the matched schema triple to the output. A prefix sum over the result numbers then yields for each thread the number of results that will be written by threads with a lower id. This value is equal to the index into the global result vector at which the thread can write its result. In a second pass each thread reads the matched schema index and materializes new triples.

The hash table with schema elements is stored in two separate vectors. One vector holds bucket information with an offset into the second vector and the number of overflow entries it contains. Since it has an entry for each calculated hash index it can contain empty values when there is no schema subject that hashes to a given value. The second vector stores the buckets with the schema subject and the number of successors packed into a single 64-bit value, followed by the successors of the schema subject. For hashing we use a modified version of Google CityHash[9] which we tailored to an input length of 8 bytes.

3.4 Avoiding Duplicates

As discussed in Subsection 2.2, simple application of RDFS rules can lead to duplicate triples being generated. To study the nature of these duplicates and where they originate from, we analyzed generated triples for each rule on two datasets. Table 2 shows the amount of new triples as well as duplicates generated by each rule for both data sets. Consider rules (2) and (3) in the DBpedia dataset. Rule (2) produces more than 20 times as many duplicates as it produces unique triples while (3) produces about nine times as many. Rules (11) and (9) combined produce more than 40,000 times as many duplicates as useful triples.

This huge amount of duplicates not only waste memory but also bandwidth when being copied from and to the device. For each duplicate triple there must be determined that it actually is a duplicate which requires a hash table lookup in our implementation. The cost can be even higher if other or no index structures are used. Our storage layer can determine whether a given triple is a duplicate in about half the time it takes to store it. Given the figures above, between five and ten times the amount of work done in the storage layer is thus wasted on detecting duplicates. To eliminate this overhead it is important to detect duplicates as early as possible by avoiding their materialization. In

---

[9] `http://code.google.com/p/cityhash/`

Table 2: Number of triples generated per rule for DBpedia (ca. 26 million triples) and YAGO2 Core (ca. 36 million triples) datasets.

	DBpedia			YAGO2 Core		
Rule	Triples	Duplicates	Ratio	Triples	Duplicates	Ratio
(5)	0	0	–	0	19	>
(7)	0	0	–	3,551,361	88,477	0.03
(2)	368,832	7,630,029	21	6,450,781	13,453,038	2.1
(3)	568,715	4,939,870	8.7	409,193	1,511,512	3.7
(11)	259	610	2	3,398,943	366,764	0.1
(9)	0	8,329,278	>	6,685,946	3,173,957	0.5
(11+9)	259	10,398,328	42,162	35,061,599	57,969,000	1.7
all	1,650,607	23,775,152	14	45,766,218	89,370,361	2.0

the following sections we devise two strategies for dealing with duplicates of different origin as discussed in Subsection 2.2. We refer to these as the G strategy for global duplicates and the L strategy for local duplicates. In Section 4 we compare both strategies with respect to their efficacy on different datasets.

Preventing global duplicates RDFS rules are verbose—the same conclusions can be derived from different rules. Detecting such duplicates can only be done by allowing each thread a global view of all the triples that are already stored. Since all rules that produce large amounts of duplicates (i. e. (2), (3), and (9)) create triples with `rdf:type` as the predicate, it is sufficient to index only those triples[10]. We use an indexing scheme and a hash table similar to the one used for the schema elements when computing the hash join. Our index is stored in two vectors: one maps the calculated hash value to a bucket address, while the other one holds the buckets. The structure of buckets can be kept simpler since, with a fixed predicate, each needs to contain only a subject and an object. Due to its size, local memory cannot be used and we need to keep the index in *global device memory*. Store operations to global memory cannot be synchronized among threads in different work groups. Thus our index is static and is not extended during computation.

Removing local duplicates Avoiding rule-local duplicates on a global level cannot be done in OpenCL, since global synchronization is not possible during kernel execution. Accordingly, we instead remove those duplicates on the device after they have been materialized but before they are copied to the host. This frees the host from having to deal with those duplicates.

Our procedure for locally removing duplicates is shown in Algorithm 1. It works by first sorting the values in local memory. Each thread then determines whether its

---

[10] Rule (11), though producing a large number of duplicates, is implemented in a serial algorithm on the host and can thus not be addressed by this strategy.

```
       Input: global thread id g
       Input: local thread id l
       Input: local buffer lbuf
       Input: flag buffer flags
       Input: global result buffer gbuf
       Result: global buffer gbuf with locally unique values
  1    flags_l ⟵ 0, gbuf_l ⟵ 0
  2    local_sort(lbuf)
  3    if l > 0 then
  4        if lbuf_l = lbuf_{l-1} then flags_l ⟵ 1
  5        else flags_l ⟵ 0
  6        f ⟵ flags_l
  7        prefix_sum(flags)
  8        if f = 0 then
  9            k ⟵ flags_l
 10            gbuf_{g-k} ⟵ lbuf_l
```

Algorithm 1. Removing duplicates within a work group using local device memory.

neighbor's value is a duplicate of its own value and if so, writes 1 into a flag buffer. A parallel prefix sum is then performed over the flags. Thereafter, the flag buffer contains for each thread the number of duplicate entries in threads with lower ids. If the flag determined by a thread in line 4 was 0 (i. e. its neighbor's value is not a duplicate of its own), it is the first in a series of duplicates. Thus it copies its value to a position in the global output buffer that is k positions lower than its own id, where k is a local displacement value obtained from the scanned flag buffer.

3.5 Improving work efficiency on GPU devices

On modern GPUs work items do not execute in isolation. Independent of the work group size they are scheduled in groups of 64 (AMD) or 32 (NVIDIA), called *wavefront* or *warp*, respectively. All threads within such a group must execute the same instructions in lock-step. That is, different code paths due to control flow statements are executed by all items, throwing away results that are not needed (predication).

A naïve implementation of our algorithm that loops over all successors of a join match to materialize triples would thus incur wasted bandwidth. To address this problem, we move the loop up to the thread level, effectively increasing the number of work items for the second pass. Each work item then materializes at most one triple. To this end each thread gets passed a local successor index identifying which successor it has to compute for a given subject.

4 Experimental results

In this section, we evaluate our implementation with two real-world data sets. The DB-pedia dataset consists of the DBpedia Ontology, Infobox Types and Infobox Properties

from DBpedia 3.7 [2], together amounting to more than 26 million triples. The second dataset is YAGO2 Core [6] in the version released on 2012-01-09, which is sized at about 36 million triples.

We perform two different experiments. First, to show scalability on different levels of hardware parallelism, we measure running times of our reasoning algorithm for each dataset with different numbers of CPU cores used. We achieve this by partitioning the CPU device into sub-devices with the respective number of cores[11]. For this experiment, we used an Opteron server with four CPUs having 8 cores each. It is exposed to OpenCL as a single device having 32 compute units (cores). For our experiment we created sub-devices with 16, 8, 4 and 2 compute units.

In order to study effectiveness of our optimizations, we performed another experiment using a GPU device with 20 compute units. We measure kernel running time as well as the time for computing the complete entailment. This includes time for setting up index structures, executing our rule implementations, detecting any duplicates and storing all entailed triples. We give results for no optimization, removing local duplicates, preventing global duplicates as well as for both kinds of optimizations combined.

In order to compare our implementation with existing work we set up the system described by Damásio et al. [3]. We used PostgreSQL 9.1.3 installed on our Ubuntu system and configured it to use 6 GiB of system memory as buffer cache. Time measurements of rule implementations were done by having PostgreSQL print timestamps before and after each experiment and subtracting the values. To set up the system, we followed the authors' blog entry[12]. We performed the largest `rdfs:subClassOf` transitive closure experiment (T2) and the largest full closure experiment (T6) using the non-annotated rule sets and the fastest implementation variant as reported in [3]. Between repetitions we emptied all tables and re-imported the data. The results of this experiment are shown in Table 3. For T6 we had to disable literal detection within the materialization kernel, which normally prevents triples with literal subjects from being materialized. Experiment T2 can be used to determine the baseline speedup that is gained by using a native C++ implementation without GPU acceleration or parallelism over the PL/pgSQL implementation used by Damásio and Ferreira [3]. We determined this baseline speedup to be about 2.6. Experiment T6 is executed about 9.5 times faster by our system. That is, our system actually performs more than three times better than what one could expect given given the baseline speedup.

Table 3: Closure computation times for experiments T2 and T6 done by Damásio and Ferreira [3] repeated on our hardware and the system described in this paper.

	Input triples	Output triples	Damásio (ms)	Our system (ms)	Speedup
T2	366,490	3,617,532	23,619.90	9,038.89	2.6×
T6	1,942,887	4,947,407	18,602.43	1,964.49	9.5×

---

[11] This OpenCL 1.2 feature is not yet available on GPUs.

[12] `http://ardfsql.blogspot.de/`

Note that experiment T6 has also been done by Urbani et al. [19] on their MapReduce implementation in more than three minutes. For this graph (~1.9 million triples) our system is much faster since the whole graph including index structures fits easily into main memory, while the overhead of the MapReduce framework dominates their experiment. This result would likely change if a significantly larger graph was used. Sufficient disk-based data structures for managing such a graph are, however, beyond the scope of this paper.

4.1 Experimental setup

Each experiment was repeated five times and the minimum of all five runs was taken. The benchmark system used was a Linux desktop running Ubuntu 12.04 with an Intel Core i7 3770 CPU and 8 GiB of system memory. The GPU used in our experiments was a midrange AMD Radeon HD 7870 with 2 GiB of on-board memory and 20 compute units. For scalability experiments, we used an Opteron server running Ubuntu Linux 10.04. It was equipped with four AMD Opteron 6128 CPUs, each having eight cores and a total 128 GiB of system memory.

All experiments were run using AMD APP SDK version 2.7 and timings were determined using AMD APP profiler version 2.5.1804. C++ code was compiled with Clang 3.1[13] on the desktop and GCC 4.7.1 on the server using `libstdc`++ in both cases.

The source code of our implementation is available on GitHub[14]. Due to their size datasets are not part of the source code repository and can be recreated as described in the next section.

4.2 Data scaling

Both datasets were scaled as follows: (1) all schema triples were separated, (2) instance triples were scaled to 1/2, 1/4th, 1/8th, and 1/16th of the original size, (3) scaled instance triples were combined with all schema triples. The resulting number of instance triples for each dataset are shown in Table 4 along with the number of entailed triples. DBpedia datasets use 3,412 schema triples, YAGO2 datasets contain 367,126 schema triples.

Table 4: Datasets used in our experiments.

Dataset	Instance triples	Entailed triples	Dataset	Instance triples	Entailed triples
DBpedia	26,471,572	1,650,607	YAGO2	35,176,410	45,766,218
DBpedia/2	13,235,786	1,266,526	YAGO2/2	17,588,205	41,801,394
DBpedia/4	6,617,893	860,982	YAGO2/4	8,794,102	35,684,268
DBpedia/8	3,308,946	545,002	YAGO2/8	4,397,051	19,045,508
DBpedia/16	1,654,473	318,037	YAGO2/16	2,198,525	10,938,726

---

[13] `http://clang.llvm.org/`
[14] `https://github.com/0xfeedface/grdfs`

4.3 Results and discussion

Results of our first experiment are depicted in Figure 2. For up to 16 compute units, the kernel running time is approximately halved when the number of cores is doubled. If all 32 compute units are used the running time can be seen to even slightly increase. Note that the Opteron CPUs used in this experiment have a feature similar to Intel's Hyper-threading where some CPU resources are shared by two cores. Thus it remains unclear whether the observed effect is due to limitations of our algorithm or congestion of shared CPU resources.
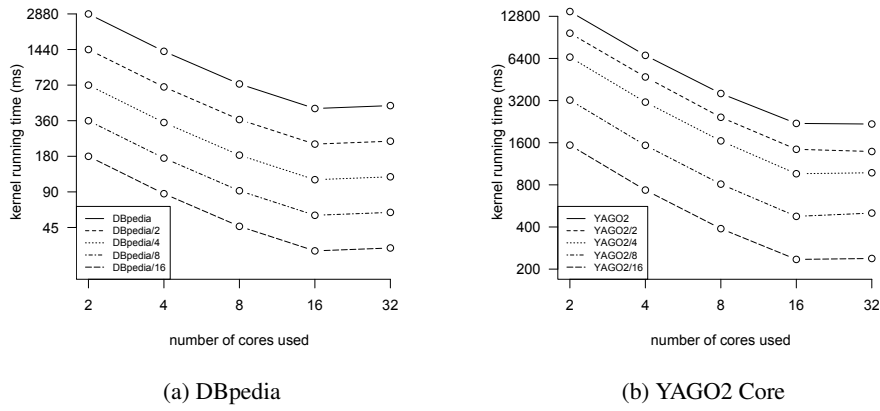

(a) DBpedia

(b) YAGO2 Core

Fig. 2: Kernel running times on different numbers of CPU cores.

Table 5 shows benchmark results on the GPU device for two datasets and different combinations of duplicate removal strategies. For DBpedia we use the full dataset, we use the YAGO2/8 dataset only since the full closure of YAGO2 does not fit into the GPU memory. Note that, in this case, the closure would have to be computed in several runs but an algorithm for dynamic data partitioning lies beyond the scope of this paper and will be addressed in future work. In Table 5 one can see that the sorting step involved in the Local strategy does increase the kernel running time to about four to seven times that of the plain kernels without duplicate removal. The result is a reduction of the number of duplicates by factor of up to 13 for YAGO2 and 2 for the DBpedia dataset. The total time needed for computing the closure is reduced by 13 % for DBpedia and 11 % for YAGO2 by this strategy.

The Global deduplication strategy appears to be less effective in terms of overall speedup. Even though the number of duplicates reduced by it for the DBpedia dataset is about seven times that of the Local strategy, closure computation is sped up by only 3.7 %. For the YAGO2 dataset, Global deduplication does not lead to a significant decrease in duplicates.

Table 5: Kernel and complete closure computing times on the GPU device with Local (L), Global (G) or both duplicate removal strategies. The speedup is shown for the complete computation over the *None* strategy.

Dataset	Strategy	Kernel time (ms)	Closure time (ms)	Duplicates	Speedup
DBpedia	None	28.444	6,884.15	23,775,152	
	L	120.915	6,083.76	12,165,520	13.2 %
	G	52.305	6,635.60	1,511,758	3.7 %
	L+G	117.400	6,557.94	1,057,470	5 %
YAGO2/8	None	25.565	21,625.19	31,552,221	
	L	187.169	19,554.09	2,399,898	10.6 %
	G	53.948	21,622.31	29,357,936	0 %
	L+G	215.947	19,807.66	1,786,753	9.2 %

One possible explanation for the reduced efficacy of the Global strategy lies in our implementation. We use a hash table that is built on the host and thus requires a significant amount of serial work. Results shown in Table 5 suggest this cost to be almost half that of removing duplicates. The Local strategy, on the other hand, is performed entirely in parallel, thus coming almost for free when compared to the Global strategy. One possible improvement that we will look at in the future is computing the hash table on the device.

Table 6: Kernel execution and total closure computation time on CPU and GPU.

Device	Kernel execution (ms)	Total (ms)
Core i7 3770 (CPU)	647.311	5509.92
Radeon HD 7870 (GPU)	114.683	5881.54

When comparing kernel execution times on GPU and CPU devices (shown in Table 6), one can see that in our system kernels execute about five times faster on the GPU than on the CPU. This is probably due to the large amount of parallelism exposed by modern GPUs. However, this does not translate into shorter closure calculation times. If computation is done on the CPU, host and device memory are essentially the same and no copying takes place. On a GPU however, data must be copied over the PCI Express bus to the device and results have to be copied back. Therefore to fully exploit GPU devices the data transfered must be kept at a minimum. At the moment we do not handle this very well since the OpenCL programming model requires buffers to be allocated in advance. If duplicates are later detected the size of buffers cannot be reduced accordingly. Instead, duplicate values are overwritten with zeros which allows easy detection on the host but does not reduce the amount of data transfered.

5 Conclusion and future work

In difference to previous related work that mainly focused on distributed reasoning via a cluster of compute notes, in this paper we tackled the problem of computing a significant subset of the RDFS closure on massively parallel hardware in a shared memory setting. We devised algorithms that can in theory exploit such high levels of parallelism and showed their scalability to at least 16 cores on CPU hardware.

In addition, we addressed two issues that make parallel RDFS reasoning non-trivial: i) we introduced a fixpoint iteration over rules (5) and (7) to support extension of the RDFS vocabulary and ii) we devised two independent strategies for dealing with the large number of duplicate triples that occur in naïve application of RDFS entailment rules. In particular, we could show our Local strategy to be highly effective in terms of speedup gain. The comparatively high cost for the Global strategy make it less effective in our system. However, if the aim lies in maximum reduction of duplicates (e. g. if the cost for duplicate detection is very high) both strategies need to be applied. One aspect of future work is thus reducing the setup cost for the Global strategy by bulding data structures on the device.

The reason for our implementation showing no performance gain on massively parallel GPU hardware over CPU hardware is due to wasted memory bandwidth when copying large amounts of data to and off the device. To overcome this future research on compressed data structures for massively parallel hardware is needed. Currently, we are not aware of any such work.

In order to cope with datasets of arbitrary size our in-memory data store would need to be replaced by a sufficient disk-based storage scheme. Such schemes specifically designed for RDF data and query workloads are already being researched. Since reasoning and query workloads could be different, designing such schemes specifically for reasoning systems might be a promising approach as well.

Massively parallel hardware is also used for cloud computing platforms that combine several devices[15]. So far, our work focused on using a single OpenCL device per reasoning workload. An obvious extension would be the use of several devices by partitioning instance triples and replicating schema triples on all devices.

Lastly, exploiting a different aspect of modern GPU hardware could be the use of annotated RDF data as described in [17]. In this work, each RDF triple is annotated with a real number $\varphi \in [0,1]$. The tradeoffs typically made in GPU chip design favor floating-point over integer arithmetic. Thus, extended RDFS rules from [17] would likely benefit from being executed on the floating-point units of modern GPUs.

Acknowledgments

---

[15] http://aws.amazon.com/about-aws/whats-new/2010/11/15/
announcing-cluster-gpu-instances-for-amazon-ec2/

References

1. D. Abadi. *Query Execution in Column-Oriented Database Systems.* PhD thesis, Massachusetts Institute of Technology, 2008.
2. C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia – A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165, Sept. 2009.
3. C. V. Damásio and F. Ferreira. Practical RDF Schema Reasoning with Annotated Semantic Web Data. In *The Semantic Web – ISWC 2011*, 2011.
4. D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1–8. ACM, 1984.
5. P. Hayes. RDF Semantics. W3C Recommendation, W3C, 2004. `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`.
6. J. Hoffart, K. Berberich, and G. Weikum. YAGO2: a Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *Artificial Intelligence Journal, Special Issue on Artificial Intelligence, Wikipedia and Semi-Structured Resources*, 2012.
7. Z. Kaoudi, I. Miliaraki, and M. Koubarakis. RDFS Reasoning and Query Answering on Top of DHTs. In *International Semantic Web Conference*, pages 499–516. Springer, 2008.
8. Y. Kazakov, M. Krötzsch, and F. Simančík. Concurrent Classification of EL Ontologies. In *International Semantic Web Conference*, pages 305–320. Springer, 2011.
9. S. Munoz, J. Pérez, and C. Gutierrez. Minimal deductive systems for RDF. In *The Semantic Web: Research and Applications*, pages 53–67. Springer, 2007.
10. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. In *Proc. of the VLDB Endowment*, pages 647–659. VLDB Endowment, 2008.
11. J. Nickolls and W. J. Dally. The GPU Computing Era. *Micro, IEEE*, 30(2):56–69, 2010.
12. E. Nuutila. An efficient transitive closure algorithm for cyclic digraphs. *Information Processing Letters*, 52(4):207–213, 1994.
13. E. Oren, S. Kotoulas, G. Anadiotis, R. Siebes, A. Ten Teije, and F. van Harmelen. Marvin: A platform for large-scale analysis of Semantic Web data. In *Proc. of the WebSci'09*, 2009.
14. Y. Ren, J. Pan, and K. Lee. Parallel ABox Reasoning of EL Ontologies. In *Proc. of the First Joint International Conference of Semantic Technology (JIST 2011)*, 2011.
15. N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *Proc. of the IEEE Intl. Symp. on Parallel & Distributed Processing*, 2009.
16. S. Sengupta, M. Harris, Y. Zhang, and J. Owens. Scan primitives for GPU computing. In *Proc. of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106. Eurographics Association, 2007.
17. U. Straccia, N. Lopes, G. Lukácsy, and A. Polleres. A General Framework for Representing and Reasoning with Annotated Semantic Web Data. In *Proc. of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10), AAAI Press*, pages 1437–1442, 2010.
18. H. J. ter Horst. 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*, 3:79–115, 2005.
19. J. Urbani, S. Kotoulas, E. Oren, and F. van Hermelen. Scalable Distributed Reasoning using MapReduce. In *The Semantic Web – ISWC 2009*, 2009.
20. S. Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11–12, 1962.
21. J. Weaver and J. Hendler. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In *The Semantic Web – ISWC 2009*, pages 682–697. Springer, 2009.