Innermost many-sorted term rewriting on GPUs

Science of Computer Programming 225 (2023) 102910

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Innermost many-sorted term rewriting on GPUs

Johri van Eerd[a], Jan Friso Groote[a], Pieter Hijma[b], Jan Martens[a], Muhammad Osama[a], Anton Wijs[a,*]

[a] Eindhoven University of Technology, Eindhoven, the Netherlands
[b] VU Amsterdam, the Netherlands

ARTICLE INFO

Article history:
Received 6 February 2022
Received in revised form 22 November 2022
Accepted 23 November 2022
Available online 9 December 2022

Keywords:
Term rewriting
GPU
Programming
Parallel computing

ABSTRACT

This article presents a way to implement many-sorted term rewriting on a GPU. This is done by letting the GPU repeatedly perform a massively parallel evaluation of all subterms. Innermost many-sorted term rewriting is experimentally compared with a relaxed form of innermost many-sorted term rewriting, and two different garbage collection mechanisms, to remove terms that are no longer needed, are discussed and experimentally compared. It is concluded that when the many-sorted term rewrite systems exhibit sufficient internal parallelism, GPU rewriting substantially outperforms the CPU. Both relaxed innermost many-sorted rewriting and garbage collection further improve this performance. Since the implementation can probably be even further optimised, and because in any case GPUs will become much more powerful in the future, this suggests that GPUs are an interesting platform for (many-sorted) term rewriting. As term rewriting can be viewed as a universal programming language, this also opens a route towards programming GPUs by term rewriting, especially for irregular computations.

1. Introduction

Graphics Processing Units (GPUs) have enormous computational power and performance-per-watt compared to (multi-core) CPUs [1]. GPUs are optimised for the highly parallel and regular computations that occur in graphics processing, but they become more and more interesting for general purpose computations (for instance, see [2–7]). It is not without reason that modern super computers have large banks of graphical processors installed in them [8]. GPU designers realise this and make GPUs increasingly suitable for irregular computations. For instance, they have added improved caches and atomic operations.

This raises the question to what extent the GPU can be used for more irregular computational tasks. The main limitation is that a highly parallel algorithm is needed to fully utilise the power of the GPU. For irregular problems it is the programmer's task to recognise the regularities in problems over irregular data structures such as graphs.

The evaluation of term rewriting systems (TRSs) is an irregular problem that is interesting for the formal methods community. For example, term rewriting increases the expressiveness of models in the area of model checking [9] and the performance of term rewriting is a long-standing and important objective [10]. *Many-sorted* term rewriting systems involve

[*] Corresponding author.
E-mail addresses: johrivaneerd@hotmail.com (J. van Eerd), j.f.groote@tue.nl (J.F. Groote), pieter@cs.vu.nl (P. Hijma), j.m.martens@tue.nl (J. Martens), o.m.m.muhammad@tue.nl (M. Osama), a.j.wijs@tue.nl (A. Wijs).

https://doi.org/10.1016/j.scico.2022.102910

sorts, i.e., data types, which provides a means to require that terms are well-sorted. A question that follows is whether this model for computation can be used to express programs for GPUs more easily. Besides their use in the formal methods community, a TRS is also a simple, yet universal mechanism for computation [11,12].

We recall that a term rewriting system that enjoys the Church-Rosser property is parallel in nature, in a sense that rewriting can take place at any point in the system and the order in which it takes place does not influence the outcome. This suggests a very simple model for parallel evaluation. The (sub)terms of the system can be distributed over the available processors, and each processor can independently work on its own terms and do its evaluation.

We designed experiments and compared GPU many-sorted term rewriting with CPU many-sorted term rewriting of the same terms. Furthermore, we experimented with two different mechanisms for garbage collection of deleted terms, and considered both innermost many-sorted term rewriting and a relaxed version of innermost many-sorted term rewriting. We find that our implementation manages to employ 80% of the bandwidth of the GPU for random accesses. Since random accesses are the performance bottleneck in our implementation, the GPU is used quite well. For intrinsically parallel rewrite tasks, the GPU outperforms a CPU with up to a factor 10. The experiments also show that if the number of terms that can be evaluated in parallel is reduced, rewriting slows down quite dramatically. For this reason, relaxed innermost rewriting is often faster than innermost rewriting, as the former is more flexible in evaluating terms in parallel than the latter. The reason that the possibilities for parallel evaluation affect the performance so much is due to the fact that individual GPU processors are much slower than a CPU processor and GPU cycles are spent on non-reducible terms.

Garbage collection is not only necessary to keep having sufficient memory during rewriting, it also tends to speed up the rewriting itself, as the use of garbage collection tends to lead to better memory access patterns when terms are accessed. Our garbage collection experiments show that typically, maintaining a queue of free positions for new terms is more efficient than periodically moving terms to shift out the deleted ones, but our results suggest that as the number of rewritten terms increases, the latter mechanism becomes increasingly efficient.

This leads us to the following conclusion. Term rewriting, both many-sorted and single-sorted, on a GPU certainly has potential. Although our implementation performs close to the random access peak bandwidth, this does not mean that performance cannot be improved. It does mean that future optimisations need to focus on increasing regularity in the implementation, especially in memory access patterns, for example by grouping together similar terms, or techniques such as kernel unrolling in combination with organizing terms such that subterms are close to the parent terms as proposed by Nasre et al. [13]. Furthermore, we expect that GPUs quickly become faster, in particular for applications with random accesses.

However, we also observe that when the degree of parallelism in a term is reduced, it is better to let the CPU do the work. This calls for a hybrid approach where it is dynamically decided whether a term is to be evaluated on the CPU or on the GPU depending on the number of subterms that need to be rewritten. This is future work. We also see that designing inherently parallel rewriting systems is an important skill that we must learn to master.

Although much work lies ahead of us, we conclude that using GPUs to solve (many-sorted) term rewriting processes is promising. It allows for abstract programming independent of the hardware details of GPUs, in the sense that it is not needed to explicitly indicate which data should be processed by which threads, and what type of memory should be used. Also, it offers the potential of evaluating appropriate rewrite systems at least one order, and in the future several orders of magnitude faster than a CPU.

Contributions In this article, we investigate whether and under which conditions many-sorted term rewriting systems can be evaluated effectively on GPUs. We experimented with different compilation schemes from rewrite systems to GPU code and present here one where all processors evaluate all subterms in parallel. This has as drawback that terms that cannot be evaluated still require processing time. Terms can become discarded when being evaluated, and therefore garbage collection is required. All processors are also involved in this.

The current work extends our previously published work on GPU many-sorted term rewriting [14]. Compared to that paper, we have added the following contributions:

1. The fact that our parallel term rewriting supports the use of sorts is made explicit in the definitions.
2. More aspects of the parallel term rewriting mechanism are algorithmically explained, and the examples have been expanded.
3. The data structures have been redesigned, to further optimise the code.
4. The garbage collector introduced in [14] periodically collects the indices of free term positions in a list. An alternative is to periodically sort the terms, thereby shifting the free positions out. In [6], two of the current authors investigated the latter approach in the context of SAT solving. In the current article, we use both approaches, and experimentally compare their impact on performance for term rewriting.
5. In [14], we used a rewriting strategy known as *innermost* rewriting, which has the nice property for our parallelisation that it guarantees thread-safety, i.e., the absence of data races. However, it can also be a limiting factor w.r.t. the amount of rewriting that can be done in parallel. In this article, we also consider a *relaxed* form of innermost rewriting, which can lead to more rewriting being possible in parallel (this depends on the rewrite system), while still being thread-safe.

2. Related work

An earlier approach to inherently evaluate a program in parallel was done in the eighties. The Church-Rosser property for pure functional programs sparked interest from researchers, and the availability of cheap microprocessors made it possible to assemble multiple processors to work on the evaluation of one single functional program. Jones et al. proposed GRIP, a parallel reduction machine design to execute functional programs on multiple microprocessors that communicate using an on-chip bus [15]. At the same time Barendregt et al. proposed the Dutch Parallel Reduction Machine project, that follows a largely similar architecture of many microprocessors communicating over a shared memory bus [16]. Although technically feasible, the impact of these projects was limited, as the number of available processors was too small and the communication overhead too severe to become a serious contender of sequential programming. GPUs offer a different infrastructure, with in the order of a thousand fold more processors and highly integrated on chip communication. Therefore, GPUs are a new and possibly better candidate for parallel evaluation of TRSs.

Current approaches for GPU programming are to make a program at a highly abstract level and transform it in a stepwise fashion to an optimal GPU program [17]. Other approaches are to extend languages with notation for array processing tasks that can be sparked off to the GPU. Examples in the functional programming world are Accelerate [18], an embedded array processing language for Haskell, and Futhark [19], a data parallel language which generates code for NVIDIA's Compute Unified Device Architecture (CUDA) interface. While Futhark and Accelerate make it easier to use the power of the GPU, both approaches are tailored to highly regular problems. Implementing irregular problems over more complicated data structures remains challenging and requires the programmer to translate the problem to the regular structures provided in the language as seen in, for example, [20–22].

Related to this work is the work of Nasre et al. [23] where parallel graph mutation and rewriting programs for both GPUs and CPUs are studied. In particular they study Delaunay mesh refinement (DMR) and points-to-analysis (PTA). PTA is related to term rewriting in a sense that nodes do simple rule based computations, but it is different in the sense that no new nodes are created. In DMR new nodes and edges are created but the calculations are done in a very different manner. The term rewriting in this work can be seen as a special case of graph rewriting, where symbols are seen as nodes and subterms as edges.

Finally, regarding garbage collection for GPUs, the authors of [24,25] investigated how to offload garbage collectors to an Accelerated Processing Unit (APU). A promising alternative for stream compaction [26] via parallel defragmentation has been proposed in [27]. The two garbage collectors that we consider in this article are simpler: one, originally proposed in the conference version of this article [14], has been tailored to the setting of term rewriting, and the second one, originally proposed in [6], has originally been designed for SAT solving, but can be adapted for term rewriting in a straightforward way. This allows both garbage collectors to be simple, yet effective.

3. Preliminaries

We introduce many-sorted term rewriting, what it means to apply rewrite rules, and an overview of the CUDA GPU computing model.

Many-sorted term rewriting Terms are constructed from a set of variables V and a set of function symbols F. A function symbol is applied to a predefined number of terms as arguments. We refer to this number as the *arity* of the function symbol, and denote the arity of a function symbol f by $\mathtt{arity}(f)$. If $\mathtt{arity}(f) = 0$, we say f is a *constant*.

The set of *subterms* $\mathtt{sub}^+(t)$ of a term t is inductively defined as follows:

$$\mathtt{sub}^+(t) = \begin{cases} \emptyset & \text{if } t \in V, \\ \bigcup_{1 \le i \le \mathtt{arity}(t)} \{t_i\} \cup \mathtt{sub}^+(t_i) & \text{if } t = f(t_1, \dots, t_{\mathtt{arity}(f)}). \end{cases}$$

With $\mathtt{sub}_i(t)$ $(i \ge 1)$, we refer to the i-th *direct subterm* of term t. This is defined as follows, with $\perp$ denoting that $\mathtt{sub}_i(t)$ is undefined:

$$\mathtt{sub}_i(t) = \begin{cases} \perp & \text{if } t \in V \text{ or } i > \mathtt{arity}(t), \\ t_i & \text{if } t = f(t_1, \dots, t_{\mathtt{arity}(f)}) \text{ and } i \le \mathtt{arity}(t). \end{cases}$$

The *head symbol* of a term t is defined as $\mathtt{hs}(f(t_1, \dots, t_k)) = f$. If $t \in V$, $\mathtt{hs}(t)$ is undefined. With $\mathtt{Var}(t)$, we refer to the set of variables occurring in term t. It is defined as follows:

$$\mathtt{Var}(t) = \begin{cases} \{t\} & \text{if } t \in V, \\ \bigcup_{1 \le i \le \mathtt{arity}(t)} \mathtt{Var}(t_i) & \text{if } t = f(t_1, \dots, t_{\mathtt{arity}(f)}). \end{cases}$$

In addition, there is a set of sorts S. Every variable and function symbol has a sort in S, and for function symbols, it is defined which sort each of its arguments has. With S^n, we refer to n-tuples over S, and S^* denotes the set of all finite sequences over S. The following functions are used to map terms to sorts:

$$\mathtt{st} : F \cup V \rightarrow S$$

$$\mathtt{ar} : F \rightarrow S^*$$

We require for all $f \in F$ that $\mathtt{ar}(f) \in S^{\mathtt{arity}(f)}$. The function st maps each variable in V and function symbol in F to a sort. The function ar defines the sort of each argument of a function symbol. To refer to the sort of the i-th argument of a function symbol f, with $i \in \{1, \ldots, \mathtt{arity}(f)\}$, we use the notation $\mathtt{ar}(f, i)$.

To refer to the sort of a term, we use a function sort. The sort of a term t is defined as the sort associated to its head symbol, in case $t \notin V$. Otherwise, it is the sort associated to the variable t:

$$\mathtt{sort}(t) = \begin{cases} \mathtt{st}(t) & \text{if } t \in V, \\ \mathtt{st}(\mathtt{hs}(t)) & \text{if } t \notin V. \end{cases}$$

Together, the sets V, F and S constitute a *signature* $\Sigma = (F, V, S, \mathtt{arity}, \mathtt{st}, \mathtt{ar})$.[1] The set of terms T_Σ over a signature Σ is inductively defined as the smallest set satisfying:

- If $t \in V$, then $t \in T_\Sigma$;
- If $f \in F$, and for all $1 \le i \le \mathtt{arity}(f)$, we have $t_i \in T_\Sigma$ and $\mathtt{sort}(t_i) = \mathtt{ar}(f, i)$, then $f(t_1, \ldots, t_{\mathtt{arity}(f)}) \in T_\Sigma$.

A *Many-sorted Term Rewrite System* (*MTRS*) over a signature Σ is a set of rules. Each rule is a pair of terms from T_Σ, namely a left-hand side and a right-hand side. Given an arbitrary term t and an MTRS R, rewriting means to replace occurrences in t of an instance of the left-hand side of a rule in R by the corresponding instance of its right-hand side, and then repeating the process on the result.

Definition 1 *(Many-sorted term rewrite system).* An MTRS R over a signature Σ is a set of pairs of terms, i.e., $R \subseteq T_\Sigma \times T_\Sigma$. Each pair $(l, r) \in R$ is called a *rule*, and is typically denoted by $l \rightarrow r$. Each rule $(l, r) \in R$ satisfies two properties: (1) $l \notin V$, and (2) $\mathtt{Var}(r) \subseteq \mathtt{Var}(l)$.

Given a rule $l \rightarrow r$, we refer to l as the left-hand-side (LHS) and to r as the right-hand-side (RHS).

In the remainder of this article, *many-sorted term rewriting* is simply referred to as *term rewriting*, i.e., we always consider term rewrite systems that are many-sorted.

Definition 2 *(Substitution).* For an MTRS R over a signature $\Sigma = (F, V, S, \mathtt{arity}, \mathtt{st}, \mathtt{ar})$, a *substitution* $\sigma : V \rightarrow T_\Sigma$ maps variables to terms, with for all $v \in V$, $\mathtt{sort}(v) = \mathtt{sort}(\sigma(v))$. We write $t\sigma$ for a substitution σ applied to a term $t \in T_\Sigma$, defined as $\sigma(t)$ if $t \in V$, and $f(t_1\sigma, \ldots, t_{\mathtt{arity}(f)}\sigma)$ if $t = f(t_1, \ldots, t_{\mathtt{arity}(f)})$.

Substitutions allow for a *match* between a term t and rule $l \rightarrow r$. A rule $l \rightarrow r$ is said to match t iff a substitution σ exists such that $l\sigma = t$. If such a σ exists, then we say that t *reduces* to $r\sigma$. A match $l\sigma$ of a rewrite rule $l \rightarrow r$ is also called a *redex*.

A term t is in *normal form*, denoted by *nf*(t), iff its subterms are in normal form and there is no rule $(l, r) \in R$ and substitution σ such that $t = l\sigma$.

A term can be a redex, but it can also *contain* a number of redexes. In a *reduction step*, one of these redexes is reduced. For a given MTRS R over a signature $\Sigma = (F, V, S, \mathtt{arity}, \mathtt{st}, \mathtt{ar})$, we define the one-step reduction relation $\rightarrow_R$:

Definition 3 *(One-step reduction).* Given an MTRS R over a signature $\Sigma = (F, V, S, \mathtt{arity}, \mathtt{st}, \mathtt{ar})$, the one-step reduction relation $\rightarrow_R$ is defined on T_Σ inductively as follows:

1. For all $t \in V$, t is in normal form w.r.t. $\rightarrow_R$;
2. Assume that $\rightarrow_R$ has already been defined for terms $t_1, \ldots, t_{\mathtt{arity}(f)}$, then $\rightarrow_R$ is defined for term $f(t_1, \ldots, t_{\mathtt{arity}(f)})$ as follows:
   (a) For all $i \in \{1, \ldots, \mathtt{arity}(f)\}$, if $t_i \rightarrow_R t'$, then

   $$f(t_1, \ldots, t_{\mathtt{arity}(f)}) \rightarrow_R f(t_1, \ldots, t_{i-1}, t', t_{i+1}, \ldots, t_{\mathtt{arity}(f)})$$

   (b) If there is a rule $\rho : l \rightarrow r \in R$ for which a substitution σ exists such that $f(t_1, \ldots, t_{\mathtt{arity}(f)}) = l\sigma$, then

   $$f(t_1, \ldots, t_{\mathtt{arity}(f)}) \rightarrow_R r\sigma$$

[1] This definition of signatures departs from the one used in algebraic specifications [28,29], as it includes variables, but it is in line with other papers on term rewriting (for instance, [30,31]).

If neither of the two cases above is applicable, $f(t_1, \ldots, f_{\texttt{arity}(f)})$ is in normal form w.r.t. $\rightarrow_R$.

Given an MTRS R over a signature Σ, the computation of a term $t \in T_\Sigma$ consists of repeatedly reducing one of its redexes, or t itself, i.e., this computation is formalised by $\rightarrow_R$.

As an example MTRS, Listing 1 presents a merge sort rewrite system with an input tree of depth 2 consisting of lists. After the `sort` keyword, a list is given of all sorts and function symbols. After the keyword `eqn`, rewrite rules are given in the form LHS = RHS. The set of variables is given as a list after the `var` keyword. The `input` section defines the input term. It is expected that for an input term t, it holds that $\texttt{Var}(t) = \emptyset$, i.e., that it is a *ground* term. For convenience, when we refer to an MTRS in this article, we sometimes actually refer to the combination of an MTRS and an input term on which this MTRS can be applied.

In the `sort` section, natural numbers, Booleans, lists and trees are given by the inductively defined sorts `Nat`, `Bool`, `List`, and `Tree`, respectively. For the natural numbers, we define the constant function `Zero()`, and the function `S` on natural numbers defines the successor of a number. With these two functions, Peano numbers can be constructed. The function `Len` on lists returns the length of a list. The rewrite rules for the latter function are given at lines 11–12. The Booleans are given by the two constant functions `True()` and `False()`, and the two functions on natural numbers, `Lt` and `Gt`, evaluate like the less than and greater than function, respectively. These two functions are defined by means of rewrite rules at lines 35–38 and 30–33, respectively. Note that these rules indeed define the less than and greater than functions.

A list is either empty (`Nil()`), or a natural number followed by a list (`Cons`). The functions `Merge` and `Merge2` merge two sorted lists into one sorted list (rewrite rules are given at lines 14–19). The functions `Even(L)` and `Odd(L)`, with `L` a direct subterm of type `List`, will evaluate to a `List` that contains only the elements located at even and odd positions in `L`, respectively. The `Sort` function represents the implementation of merge sort and splits the list direct subterm until it consists of at most one term, using the `Sort2` function (see lines 21–23). Finally, a tree consists of leafs that each contain a list (`Leaf`) and nodes with two subtrees (`Node`).

We have used a tree data structure to support input terms consisting of multiple lists. The MTRS rewrites all the `Sort` terms in the leaves of a given tree of lists in parallel. Other potential for parallel rewriting is implicit and can be seen, for instance, in the `Sort2` rule. The two arguments of `Merge` in the RHS of `Sort2` can be evaluated in parallel. Note that `Nil()`, `Zero()` and `S(Nat)` are in normal form, but other terms may not be.

The input term is given at lines 40–41. Note that it defines a tree of depth 2: the term is a `Node`, with each of its subtrees being a `Node` as well, and in turn, each subtree of those two subtrees is a `Leaf`.

An MTRS is *terminating* iff there are no infinite reductions possible. For instance, the rule $f(a) \rightarrow f(f(a))$, with $\texttt{sort}(a) = \texttt{sort}(f)$, leads to an infinite reduction. Determining whether a given MTRS is terminating is an undecidable problem [32].

The computation of a term in a terminating MTRS is the repeated application of rewrite rules until the term is in normal form. Such a computation is also called a *derivation*. Note that the result of a derivation may be non-deterministically produced. Consider, for example, the rewrite rule $\rho : f(f(x)) \rightarrow a$ and the term $t = f(f(f(a)))$, with $\texttt{sort}(x) = \texttt{sort}(f)$. Applying ρ on t may result in either the normal form a or $f(a)$, depending on the chosen reduction. To remove non-determinism arising between nested redexes, a *rewrite strategy* is needed. First, we focus on the *innermost* strategy, which gives priority to selecting redexes that do not contain other redexes. In the example, this means that the LHS of ρ is matched on the inner $f(f(a))$ of t, leading to $f(a)$. In Section 4.4, we relax this, and consider what we call *relaxed innermost* rewriting.

Algorithmically, (innermost) rewriting is typically performed using recursion. Such an algorithm is presented in Listing 2. With `ptr to` t, we refer to a pointer to the term t. As long as the term t is not in normal form (line 2), it is first checked whether all its direct subterms are in normal form (lines 3–4). For each direct subterm not in normal form, `derive` is called recursively (line 5), by which the innermost rewriting strategy is achieved. The `derive` function is given a pointer to a term, such that any rewriting of a term t is seen by all terms that have t as a direct subterm. If the direct subterms are checked sequentially from left to right, we have *leftmost* innermost rewriting. A parallel rewriter may check the direct subterms in parallel, since innermost redexes do not contain other redexes. Once all direct subterms are in normal form, the procedure $\texttt{rewrite}_{\texttt{hs}(t)}$ is called (line 6).

For each head symbol of the MTRS, we have a dedicated rewrite procedure. The structure of these procedures is also given in Listing 2. The variable `rewritten` is used to keep track of whether a rewrite step has been performed (line 9). For each rewrite rule $l \rightarrow r$ with $\texttt{hs}(l) = f$, it is checked whether a substitution σ exists such that $l\sigma = t$, and if so, $l \rightarrow r$ is applied on t (lines 10–12). If no rule was applicable, it is concluded that t is in normal form (line 13).

Non-determinism can also arise in rewriting due to multiple rules matching on the same term. For instance, if we have the rewrite rules $\rho_1 : f(x) \rightarrow a$ and $\rho_2 : f(x) \rightarrow b$, then applying this MTRS on a term $t = f(a)$, with $\texttt{sort}(x) = \texttt{sort}(a)$, may result in a or b. Note in Listing 2 that at line 10, it is not specified that the rules have to be considered in a particular order. In our work though, we enforce an order, defined by our implementation of the rewrite functions for the various function symbols. By doing so, we rule out this form of non-determinism.

Besides the two properties for each rule $(l, r) \in R$ stated in Definition 1, we assume that each variable $v \in V$ occurs at most once in l, i.e., l is *linear*, meaning that $l \rightarrow r$ is *left-linear* [11]. When innermost (and relaxed innermost) rewriting is used, an MTRS with non-left-linear rules can be rewritten to one that only contains left-linear rules [30,31]. To do so, an

```
sort Nat = struct Zero() | S(Nat) | Len(List);
     Bool = struct True() | False() | Lt(Nat, Nat) | Gt(Nat, Nat);
     List = struct Nil() | Cons(Nat, List) | Merge(List, List) |
               Merge2(Bool, Nat, List, Nat, List) | Even(List) |
               Odd(List) | Sort(List) | Sort2(Bool, List);
     Tree = struct Leaf(List) | Node(Tree, Tree);

var X, Y : Nat; B : Bool; L, M : List;

eqn
  Len(Nil()) = Zero();
  Len(Cons(X, L)) = S(Len(L));

  Merge(Nil(), M) = M;
  Merge(L, Nil()) = L;
  Merge(Cons(X, L), Cons(Y, M)) = Merge2(Lt(X, Y), X, L, Y, M);

  Merge2(True(), X, L, Y, M) = Cons(X, Merge(L, Cons(Y, M)));
  Merge2(False(), X, L, Y, M) = Cons(Y, Merge(Cons(X, L), M));

  Sort(L) = Sort2(Gt(Len(L), S(Zero())), L);
  Sort2(False(), L) = L;
  Sort2(True(), L) = Merge(Sort(Even(L)), Sort(Odd(L)));

  Even(Nil()) = Nil();
  Even(Cons(X, L)) = Cons(X, Odd(L));
  Odd(Nil()) = Nil();
  Odd(Cons(X, L)) = Even(L);

  Gt(Zero(), Zero()) = False();
  Gt(Zero(), S(Y)) = False();
  Gt(S(X), Zero()) = True();
  Gt(S(X), S(Y)) = Gt(X, Y);

  Lt(Zero(), Zero()) = False();
  Lt(Zero(), S(Y)) = True();
  Lt(S(X), Zero()) = False();
  Lt(S(X), S(Y)) = Lt(X, Y);

input Node(Node(Leaf(Sort(Cons(S(Zero()), ...)),
   Leaf(Sort(...))), Node(...));
```

Listing 1: An MTRS for merge sort on multiple lists.

```
procedure derive(ptr to $t, R$):
   while $\neg nf(t)$ do
      for $i \in \{1, \ldots, \text{arity}(t)\}$ do
         if $\neg nf(\text{sub}_i(t))$ then
            derive(ptr to $\text{sub}_i(t), R$)
      $t \leftarrow$ rewrite$_{\text{hs}(t)}$ $(t, R)$

procedure rewrite$_f$ $(t, R)$:
   rewritten $\leftarrow$ false
   for $(l \rightarrow r) \in \{(l, r) \in R \mid \text{hs}(l) = f\}$ do
      if $\exists\sigma : V \rightarrow T_\Sigma . l\sigma = t$ then
         $t' \leftarrow r\sigma$, rewritten $\leftarrow$ true, break
   if $\neg$rewritten then $nf(t) \leftarrow$ true
   return $t'$
```

Listing 2: A derivation procedure for term t, and a rewrite procedure for head symbol f.

equality function needs to be introduced in the MTRS, to syntactically compare terms. By incorporating equality checks in the rules, the multiple occurrences of a variable can be removed. For instance, a rule `f(X, X) = a()` can be simulated by the following rules, where `f'` and `g` are new function symbols, and `f'` is essentially a copy of `f`:

```
         f(X, Y) = g(X, Y, eq(X, Y))
  g(X, Y, True()) = a()
 g(X, Y, False()) = f'(X, Y)
```

Although not essential, the left-linearity property is convenient when developing a term rewriter, as it simplifies finding substitutions. For an MTRS with non-left-linear rules, the implicit equality conditions must be taken into account when checking for substitutions. The above rewriting of an MTRS with non-left-linear rules to an MTRS with only left-linear rules means that these conditions are made explicit, which makes it straightforward how checking the applicability of those rules should be implemented.

GPU basics In this article, we focus on NVIDIA GPU architectures and CUDA. However, our algorithms can be straightforwardly applied to any GPU architecture with a high degree of hardware multithreading and the SIMT (Single Instruction Multiple Threads) model.

CUDA is NVIDIA's interface to program GPUs. It extends the C++ programming language. CUDA includes special declarations to explicitly place variables in either the main or the GPU memory, predefined keywords to refer to the IDs of individual threads and blocks of threads, synchronisation statements, a run time API for memory management, and statements to define and launch GPU functions, known as *kernels*. In this section we give a brief overview of CUDA. More details can be found in, for instance, [33].

GPU architecture A GPU contains a set of streaming multiprocessors (SMs), each containing a set of streaming processors (SPs). For our experiments, we used the NVIDIA Turing Titan RTX. It has 72 SMs with 64 SPs each, i.e., in total 4,608 SPs.

A CUDA program consists of a *host* program running on the CPU and a collection of CUDA kernels. Kernels describe the parallel parts of the program and are launched from the host to be executed many times in parallel by different threads on the GPU.

It is required to specify the number of threads on a kernel launch and all threads execute the same kernel. Conceptually, each thread is executed by an SP. In general, GPU threads are grouped in blocks of a predefined size, usually a power of two. A block of threads is assigned to a multiprocessor.

CUDA memory model Threads have access to different kinds of memory. Each thread has a number of on-chip registers to store thread-local data. It allows fast access. All the threads have access to the *global memory* which is large (on the Titan RTX it is 24 GB), but slow, since it is off-chip. The host has read and write access to the global memory, which allows this memory to be used to provide the input for, and read the output of, a kernel execution. Furthermore, we use *unified memory* [33] to store *unified* variables that need to be regularly accessed by both the CPU and the GPU. Unified memory creates a pool of managed memory that is shared between the host and the device. This pool is accessible to both sides using the same addresses.

CUDA execution model Threads are executed using the SIMT model. This means that each thread is executed independently with its own local state (stored in its registers), but execution is organised in groups of 32 threads, called *warps*. The threads in a warp execute instructions in lock-step, i.e., they share a program counter. If the (global and unified) memory accesses of threads in a warp can be grouped together physically, i.e., if the accesses are coalesced, then the data can be obtained using a single fetch, which greatly improves the bandwidth compared to fetching physically separate data.

4. A GPU algorithm for term rewriting

In this section, we address how a GPU can perform innermost term rewriting to get the terms of a given MTRS in normal form. Due to the different strengths and weaknesses of GPUs compared to CPUs, this poses two main challenges:

1. On a GPU, many threads (in the order of thousands) should be able to contribute to the computation;
2. GPUs are not very suitable for recursive algorithms. It is strongly advised to avoid recursion because each thread maintains its own stack requiring a large amount of stack space that needs to be allocated in slow global memory.

We decided to develop a *topology-driven* algorithm [13], as opposed to a data-driven one. Unlike for CPUs, topology-driven algorithms are often developed for GPUs, in particular for irregular programs with complex data structures such as trees and graphs. In a topology-driven GPU algorithm, each GPU thread is assigned a particular data element, such as a graph node, and all threads repeatedly apply the same operator on their respective element. This is done until a fix-point has been reached, i.e., no thread can transform its element anymore using the operator. In many iterations of the computation, it is expected that the majority of threads will not be able to apply the operator, but on a GPU this is counterbalanced by the fact that many threads are running, making it relatively fast to check all elements in each iteration. In contrast, in a data-driven algorithm, typically used for CPUs, the elements that need processing are repeatedly collected in a queue before the operator is applied on them. Although this avoids checking all elements repeatedly, on a GPU, having thousands of threads together maintaining such a queue is typically a major source for memory contention.

In our algorithm, each thread is assigned a term, or more specifically a location where a term may be stored. As derivations are applied according to an MTRS, new terms may be created and some terms may be deleted. The algorithm needs to account for the number of terms dynamically changing between iterations.

In the upcoming sections, we explain the data structures and arrays that are required to implement MTRSs on GPUs. Then, we explain in detail the main procedure of parallel term rewriting.

4.1. Data structures and arrays

First, we discuss how MTRSs are represented on a GPU. Typically, GPU data structures, such as matrices and graphs, are array-based, and we also store an MTRS in a collection of arrays. Each term is associated with a unique index i, and its attributes can be retrieved by accessing the i-th element of one of the arrays. This encourages coalesced memory access for improved bandwidth: when all threads need to retrieve the normal form status of their term, for instance, they will access consecutive elements of the array that stores the normal form flags. We introduce the following GPU data structures that reside in global memory:

- An array `term` of objects of type `Term` for storing the following per term `term`[i]:
  - `term`[i]`.index` points to the current location i. This information is used when terms are moved using the Stream Compaction Garbage Collector, see Section 5.
  - `term`[i]`.hsymbol` stores the head symbol.
  - The fields $\texttt{term}[i].\texttt{arg}_1, \ldots, \texttt{term}[i].\texttt{arg}_{\texttt{maxarity}}$ store the indices of the direct subterms of each term. The value 0 is never used as index. If $\texttt{term}[i].\texttt{arg}_j = 0$, for some $1 \leq j \leq \texttt{maxarity}$, then `term`[$i$] has arity $j - 1$, and all fields $\texttt{term}[i].\texttt{arg}_j, \ldots, \texttt{term}[i].\texttt{arg}_{\texttt{maxarity}}$ should be ignored. The number of fields needed for an MTRS depends on the maximum arity occurring in the MTRS. A code generator can produce the appropriate number of fields, given an MTRS.
  - `term`[i]`.subterm_index` has a value between 1 and `maxarity`, and is used to optimise the checking of the normal form status of direct subterms (see Section 4.2).
- An array `flags` of objects of type `Flags`. For each term t with index i, a `Flags` object is stored at `flags`[i]. Each `Flags` object consists of four bit fields. The fields `flags`[i]`.nfRead` and `flags`[i]`.nf` are needed to read the current normal form status of t, and update its status, respectively. We have two normal form flags instead of one, to avoid read/write conflicts. The other two bits are used in one of our approaches to garbage collection (see Section 5).
- Integer arrays `refcounts`, `refcountsRead` are used to write and read the number of references to each term, respectively. For a term with index i, its counters are stored at `refcounts`[i] and `refcountsRead`[i]. When a term is not referenced, it can be deleted. As with the normal form flags, we have two fields for reference counting per term, to avoid read/write conflicts.
- The constant `maxarity` refers to the highest arity among the function symbols in F.
- The integer variable `n` provides the current number of terms. It is stored in unified memory.
- The Boolean flag `done` indicates whether more rewriting iterations are needed. It is also stored in unified memory.

To avoid running out of memory, some form of garbage collection is necessary to be able to reuse memory occupied by terms that are no longer needed in the term rewriting process. How the garbage collectors work is explained in Section 5. We have the following data structures for garbage collection:

- The Boolean flag `garbageCollecting` indicates whether garbage collecting is needed. It is stored in unified memory.
- The integer array `keyIndices` is used to store indices. Its function depends on which garbage collector is used (see Section 5).
- The integer variable `nextFree` points to the next free index, greater than the largest index at which term information is stored in the term arrays. There, a new term can be inserted. It is stored in unified memory.

4.2. The main procedure for term rewriting

Listing 3 presents the main procedure of the algorithm, which is executed by the CPU. In it, two GPU kernels are repeatedly called (line 8 and either line 12 or 14, depending on which garbage collector is used) until a fix-point has been reached, indicated by `done`. Copying data between arrays or data structure fields is indicated by ←.

Initially, at line 2, `nextFree` is set to the current number of terms `n`. The `nextFree` variable is used to find a new index each time a term is created. While rewriting is not finished (line 17), the `done` flag is set to `true` (line 4), after which the number of thread blocks is determined. As the number of threads should be equal to the current number of terms, `n` is divided by the preset number of threads per block (`blockSize`), rounded up. After that, the `nf` fields are copied to the fields `nfRead`, and `refcounts` is copied to `refcountsRead`. Reading and writing of the reference counters and normal form states is performed on separate data fields, to avoid the situation that newly created terms are already being rewritten before they have been completely stored in memory. The `derive` kernel is launched for the selected number of blocks (line 8). This kernel, shown in Listing 4, is discussed later. In that kernel, the GPU threads perform one rewrite iteration and set `done` to `false` if and only if another iteration is needed. At line 9 in Listing 3, `n` is updated in case the number of terms has increased. Finally, if garbage collection is needed (line 10), one of the collectors is called at line 12 or 14, depending on the user-defined flag `streamCompacting`.

```
procedure termRewriting() {
   done = false; nextFree = n;
   do {
      done = true;
      nBlocks = n / blockSize;
      flags.nfRead ← flags.nf;
      refcountsRead ← refcounts;
      derive<<nBlocks, blockSize>>(term, flags, &done, ...);
      n = nextFree;
      if (garbageCollecting) {
         if (streamCompacting)
            streamCompactCollector(...);
         else
            queueCollector(...);
         garbageCollecting ← false;
      }
   } while (!done)
}
```

Listing 3: The main rewrite procedure, executed by the CPU.

```
1   Kernel derive(term, flags, ptr to done, ...) {
2      if (tid >= n) return;
3      refcount = refcountsRead[tid];
4      if (refcount > 0) {
5         startRewriting = !flags[tid].nfRead;
6         if (startRewriting) {
7            subtindex = term[tid].subterm_index;
8            for (i = subtindex; i <= maxarity; i++) {
9               argvalue = term[tid].arg$_i$;
10              if (argvalue == 0) break;
11              if (!flags[argvalue].nfRead) {
12                 startRewriting = false;
13                 if (subtindex != i)
14                    term[tid].subterm_index = i;
15                 break;
16              }
17           }
18        }
19        if (!startRewriting) return;
20        switch (term[tid].hsymbol) {
21           case $f$: rewrite$_f$(...);
22           ...
23           default: flags[tid].nf = true;
24        }
25        done = false;
26     }
27     else garbageCollecting = true;
28  }
```

Listing 4: The derive kernel, executed by a GPU thread.

4.3. The GPU functions for innermost rewriting

In Listing 4, the GPU derive kernel is described. When the kernel is launched for nBlocks × blockSize threads, each of those threads executes the kernel to process the current terms residing in memory. The global ID of each thread is tid. Some threads may not actually have a term to look at (if n is not divisible by blockSize), therefore each thread first checks whether there is a corresponding term (line 2). If so, the value of the reference counter for the term is read (line 3), and if it is non-zero, a check for rewriting is required. Rewriting is needed if the term is not in normal form (line 5). For innermost rewriting, first, it must be checked that all direct subterms are in normal form. To avoid repetitive checking of direct subterms in each execution of the derive kernel, every thread keeps track of the last direct subterm it checked in the previous iteration. At line 7, this information, stored in subterm_index, is retrieved. Starting from this direct subterm, all subsequent direct subterms are checked in the **for**-loop at lines 8–17. Once the index 0 is encountered, this checking can be stopped (line 10). If a direct subterm is encountered that is not in normal form, rewriting cannot commence, and subterm_index is updated, if appropriate.

If rewriting is required, the suitable rewrite$_f$ function is called, depending on the head symbol of the term (lines 20–24). If no function is applicable, the term is in normal form (line 23). Finally, done is set to false to indicate that another rewrite iteration is required. Alternatively, if the reference counter is 0, the garbageCollecting flag is set (line

```
1   rewritePlus(term, flags, refcounts, ptr to done, ...) {
2      r_0 = term[tid].arg1;
3      r_hs_0 = term[r_0].hsymbol;
4      if (r_hs_0 == S) {          // check whether r1 matches
5         r_1 = term[tid].arg2;
6         term[tid].hsymbol = S;       // make the term an S term
7         r_0_0 = term[r_0].arg1;
8         r_n_0 = getNewIndex(...);    // create a new direct subterm
9         term[tid].arg1 = r_n_0;
10        term[tid].arg2 = 0;
11        term[r_n_0].hsymbol = Plus;
12        term[r_n_0].arg1 = r_0_0;
13        term[r_n_0].arg2 = r_1;
14        term[r_n_0].arg3 = 0;
15        flags[r_n_0].nf = false;
16        refcounts[r_n_0] = 1;
17        atomicAdd(&refcounts[r_0_0], 1);
18        atomicSub(&refcounts[r_0], 1);
19        done = false;
20        return;
21     }
22     if (r_hs_0 == Zero) {       // check whether r2 matches
23        r_1 = term[tid].arg2;
24        r_hs_1 = term[r_1].hsymbol;
25        term[tid].hsymbol = r_hs_1;     // copy the arg1 term
26        copyTermArgs(refcounts, term, r_1, tid, r_hs_1);
27        atomicSub(&refcounts[r_0], 1);
28        atomicSub(&refcounts[r_1], 1);
29        flags[tid].nf = true;
30        return;
31     }
32     flags[tid].nf = true;
33  }
```

Listing 5: An example rewrite function for `Plus` terms with the rules `r1: Plus(S(X),Y) = S(Plus(X,Y))` and `r2: Plus(Zero,X) = X`, executed by a GPU thread.

27). This causes the `queueCollector` or `streamCompactCollector` procedure to be executed after the `derive` kernel (see Listing 3).

Given an MTRS, the $\texttt{rewrite}_f$ functions are automatically generated by a code generator we developed, to directly encode the rewriting in CUDA code. We formalise this as follows. Given an MTRS R over a signature $\Sigma = (F, V, S, \texttt{arity}, \texttt{st}, \texttt{ar})$, we create a partition π in which the rules of R are grouped together based on the head symbol of their LHS:

$$\pi = \Big(\bigcup_{f \in F} \{\{(l, r) \in R \mid \texttt{hs}(l) = f\}\}\Big) \setminus \emptyset$$

For each set of rules P in π, we refer with P_f to the head symbol of the LHSs of all the rules in P. A function $\texttt{rewrite}_{P_f}$ is created for each set of rules P in π. In each $\texttt{rewrite}_{P_f}$, checking for matches of the rules in P, and applying substitutions whenever possible, is done in some arbitrary order, fixed by the code generator. How the function checks for matches and applies substitution can best be shown by means of an example.

Listing 5 provides example code for terms with head symbol `Plus` and rewrite rules `Plus(S(X),Y) = S(Plus(X,Y))` and `Plus(Zero,X) = X`, with `S` representing the successor function (i.e., `S(0)` represents 1). The first rule expresses that adding up two numbers `S(X)` and `Y` is equivalent to the successor of the sum of `X` and `Y`. The second rule expresses that adding 0 to some number `X` results in `X`. Applicability of these rules is checked by the $\texttt{rewrite}_{\texttt{Plus}}$ function. First, to check applicability, the index of the first direct subterm is retrieved, and with it, the head symbol of that term (lines 2–3). If the head symbol is `S` (line 4), we know that `r1` is applicable, and the index to the second direct subterm can be retrieved (line 5).

Applying a substitution σ on a term t for a rule $l \to r$ is done by first updating the head symbol of t to the head symbol of the RHS of $l \to r$, i.e., $\texttt{hs}(t) \leftarrow \texttt{hs}(r)$, and then for each direct subterm reference to $\texttt{sub}_i(t)$ $(1 \leq i \leq \texttt{maxarity})$, update the reference such that it either refers to $\texttt{sub}_i(r\sigma)$, or it is set to 0, in case $i > \texttt{arity}(r\sigma)$. When constructing terms, sharing of subterms is applied whenever possible. For instance, if a term `F(Z, Z)` needs to be created, the index to `Z` would be used twice in the new term, to make sure both direct subterm entries point to the same term in physical memory. More formally, when a substitution σ exists for a rule $l \to r$ and term t such that $l\sigma = t$, we construct the set of terms $\mathbb{T}(l)$ using the definition $\mathbb{T}(t') = \bigcup_{1 \leq i \leq \texttt{maxarity}} \{\texttt{sub}_i(t')\} \cup \mathbb{T}(\texttt{sub}_i(t'))$, map each variable $v \in \mathbb{T}(l) \cap V$ to $\sigma(v)$, identify the references to all terms in $\mathbb{T}(l)$ w.r.t. t, and construct $r\sigma$ using those references wherever possible.

Once a term t with index `tid` has been rewritten, the flag `flags[tid].nf` is set to `true` iff $\{(l, r) \in R \mid \mathtt{hs}(l) = \mathtt{hs}(t)\} = \emptyset$, i.e., there are no rules in R with the head symbol of their LHS being equal to the head symbol of t, which implies that t cannot be rewritten anymore. In all other cases, rewriting may still be possible, and we cannot conclude that t is in normal form. In those cases `done` is set to `false`, to ensure that another `derive` iteration will be performed (see Listing 3).

In Listing 5, the head symbol of the term being rewritten is updated at line 6, and the index of the direct subterm referred to by `X` is retrieved at line 7. A fresh index is obtained at line 8 to create the new term `Plus(X, Y)`. This index is set as argument 1 of the term being rewritten (line 9) and argument 2 is set to 0 (line 10). The new term is constructed at lines 11–15. The reference counters are updated at lines 16–18. The new term is referenced once, `X` gets one more reference due to the new term, and `Y` also gets one reference, but loses one due to the fact that the rewritten term no longer directly references `Y`. Finally, `Plus(X, Y)` loses a reference. The `done` flag is set to `false`, as the resulting term may not be in normal form.

Alternatively, if the head symbol is `Zero` (line 22), the rule `r2` is applicable, and the rewriting procedure should ensure that the term at position `tid` is replaced by `X`. In this case, when rewriting the term itself, we have to copy the attributes of `X` to the location `tid` of the various arrays, to ensure that all terms referencing term `tid` are correctly updated.

This copying of terms is done by first copying the head symbol (lines 24–25), and then the indices of the direct subterms, which is done at line 26 by the function `copyTermArgs`; it copies the number of direct subterms relevant for a term with the given head symbol, and increments the reference counters of those direct subterms. Next, the reference counters of `Zero` and `X` are atomically decremented (since the term `Plus(Zero, X)` is removed) (27–28), and we know that the resulting term is in normal form, since `X` is in normal form (line 29).

Finally, if no substitutions can be found for any of the rules w.r.t. a term t with index `tid`, it can be concluded that t is in normal form, i.e., `flags[tid].nf` can be set to `true`. In the example, this is done at line 32.

In Section 5, we explain how new indices are retrieved whenever a new term needs to be created, as is the case in the example of Listing 5 at line 8.

4.4. *Relaxed innermost rewriting*

With innermost rewriting, it is ensured that whenever a thread inspects the direct subterms of its own term, no other threads are simultaneously rewriting those direct subterms, since those terms are in normal form. For parallel term rewriting, this offers the nice property that no data races occur when accessing term information. However, it also has a drawback, as it sequentialises the rewriting of terms and their direct subterms.

Outermost rewriting does not seem to be a good alternative, as it requires more complicated thread synchronisation, in our setting in which each (sub)term is assigned to a separate thread. For instance, consider the rules $a \to b$, $b \to c$, $f(c) \to d$ and $g(d) \to e$, and the input term $g(f(a))$. In order for the thread t_1 assigned to a to identify that it can rewrite its term, it must somehow discover that both the thread t_2 assigned to $f(a)$ and the thread t_3 assigned to $g(f(a))$ cannot yet rewrite their terms. Of course, t_2 and t_3 can check this, store that information somewhere, and update it in each iteration of the rewriting procedure, similar to how we store the normal form status of all terms. However, this either requires each thread to inspect the status of every ‘parent’ term, which can involve many checks, or the propagation of this information from terms to their direct subterms, which must be done in sequence.

To encourage more rewriting in parallel while still ensuring data race freedom in the elegant way of innermost rewriting, we implemented a *relaxed* form of innermost rewriting. In this rewriting strategy, the normal form status of direct subterms is checked at the individual rule level, and insofar needed for each particular rule. In general, direct subterms referenced by only a variable do not need to be checked w.r.t. their normal form status. For instance, for the rule `Plus(S(X), Y) = S(Plus(X, Y))`, the normal form status of the term referenced by `Y` is not relevant. When applying the rule, this term is not inspected, only its reference is used to construct the new term. On the other hand, the first direct subterm of `Plus(S(X), Y)` must be inspected, to determine that it has head symbol `S`, so it must first be checked whether this term is in normal form. Rules where the RHS consists of a single variable, such as `Plus(Zero, X) = X`, offer an important exception: since applying such a rule involves copying the information of the term referenced by `X`, it must be checked if that term is in normal form, even though at first it may seem that only the reference `X` is used.

In general, given a rule $l \to r$ and a term t for which there exists a substitution σ such that $l\sigma = t$, the normal form status of the following direct subterms of t must be checked when relaxed innermost rewriting is performed: $\{\mathtt{sub}_i(t) \mid 1 \leq i \leq \mathtt{arity}(t) \wedge (\mathtt{sub}_i(l) \notin V \vee \mathtt{sub}_i(l) = r)\}$.

In relaxed innermost rewriting mode, a `derive` kernel is used similar to the one in Listing 4, but without lines 6–18, which implements the normal form checking. Instead, normal form checking is now done in the rewrite functions. Listing 6 presents how this is done for the `Plus` terms and the rules as presented in Listing 5. For the rule `r1`, only the normal form status of the first direct subterm is relevant. After the value of `subterm_index` has been retrieved (line 3), it is checked whether the first direct subterm still needs to be inspected (line 6), and if so, the check is done. If it is not in normal form, rewriting cannot commence (line 7). In this case, `start_rewriting` is set to `false`, and the variable `encountered_nnf` is set to `true`. The latter variable is used to keep track of whether at any point during execution of the function, a direct subterm is encountered that is not in normal form. For each individual rule, this is done using the variable `start_rewriting` (note that this variable is reset before considering the next rule at line 19), but with `encountered_nnf`, we do this for all the rules checked by the function together. If at the end of the function, no direct

```
1  rewrite_Plus(term, flags, refcounts, ptr to done, ...) {
2     r_0 = term[tid].arg_1;
3     subtindex = term[tid].subterm_index; subt_new = subtindex;
4     start_rewriting = true;
5     encountered_nnf = false;
6     if (start_rewriting && 0 >= subt_new) {
7        if (!flags[r_0].nfRead) {
8           start_rewriting = false;
9           encountered_nnf = true;
10       }
11       else if (subt_new == 0) subt_new++;
12    }
13    if (subt_new != subtindex) term[tid].subterm_index = subt_new;
14    if (start_rewriting) {
15       ...            // check applicability of r1
16       return;        // (see lines 4-21 of Listing 5)
17    }
18    r_1 = term[tid].arg_2;
19    start_rewriting = true;
20    if (start_rewriting && 0 >= subt_new) {
21       if (!flags[r_0].nfRead) {
22          start_rewriting = false;
23          encountered_nnf = true;
24       }
25       else if (subt_new == 0) subt_new++;
26    }
27    if (start_rewriting && 1 >= subt_new) {
28       if (!flags[r_1].nfRead) {
29          start_rewriting = false;
30          encountered_nnf = true;
31       }
32       else if (subt_new == 1) subt_new++;
33    }
34    if (subt_new != subtindex) term[tid].subterm_index = subt_new;
35    if (start_rewriting) {
36       ...         // check applicability of r2
37       return;     // (see lines 22-31 of Listing 5)
38    }
39    if (!encountered_nnf) {
40       flags[tid].nf = true;
41    }
42    else {
43       done = false;
44    }
45 }
```

Listing 6: Normal form checking inside a rewrite function for `Plus` terms, for relaxed innermost rewriting, executed by a GPU thread.

subterm was ever encountered that was not in normal form, we can safely conclude that the current term is in normal form, since no substitution could be found for any of the rules, even though the relevant direct subterms are all in normal form (lines 39–41). If, on the other hand, direct subterms have been encountered that were not in normal form, `done` must be set to `false`, to ensure that another `derive` iteration will be started (line 43).

If, at line 7, the direct subterm turns out to be in normal form, the field `subterm_index` can be updated (line 11). Once all relevant direct subterms have been checked, a new value of the locally stored `subterm_index` is stored in the `term` object (line 13). If, at that stage, rewriting can commence, the applicability of the rule is checked, and the rule is applied, in the usual way, as presented in Listing 5 (line 15). For the second rewrite rule, the normal form status of both direct subterms is relevant, hence both are checked (lines 20–33). The normal form status of the first direct subterm is checked at lines 20–26, and the normal form status of the second direct subterm is checked at lines 27–33.

5. Garbage collection for term rewriting

As is demonstrated in Section 4.2, when rewriting a term t, one or more new terms may need to be created, possibly using the subterms of t as building blocks. Each new subterm needs to be stored in memory at a free location. Likewise, as some subterms may no longer be needed, removing those from memory will make term rewriting more memory-efficient. However, subterms that are no longer referred to by t cannot be trivially freed. With the $\text{rewrite}_{\texttt{Plus}}$ example in Listing 5, we explained how reference counters are maintained. A term t becomes shared when the number of references to it becomes larger than one, which can happen when t is used multiple times for the construction of new terms. When a term

```
 1  procedure queueCollector(...) {
 2     nBlocks = n / blockSize;
 3     collectFreeIndices <<nBlocks, blockSize>>(keyIndices, ...);
 4  }
 5
 6  Kernel collectFreeIndices(keyIndices, ...) {
 7     if (tid >= n) return;
 8     if (refcountsRead[tid] == 0) {
 9        for (i = 1; i <= maxarity; i++) {
10           argvalue = get_arg_index(i, term[tid]);
11           if (argvalue == 0) break;
12           atomicSub(&refcounts[argvalue], 1);
13        }
14        next_free_index = atomicAdd(&next_key_index, 1);
15        keyIndices[next_free_index] = tid;
16        term[tid].subterm_index = 0;
17        ref_counts[tid] = -2;      // mark the term as removed
18     }
19  }
```

Listing 7: The Queue garbage collector.

```
 1  getNewIndex(...) {
 2     if (tid >= n) return;
 3     n_begin = next_key_index;
 4     new_id = 0;
 5     if (n_begin > 0) {
 6        n_begin = atomicSub(&next_key_index, 1);
 7        if (n_begin > 0)
 8           new_id = keyIndices[n_begin];
 9        else next_key_index = 0;
10     }
11     if (new_id == 0)
12        new_id = atomicAdd(&nextFree, 1);
13     ref_counts[new_id] = 0;
14     return new_id;
15  }
```

Listing 8: The get_new_index GPU function, executed by a GPU thread.

t loses references to direct subterms, the number of references to those direct subterms decreases. Memory locations that contain terms that are no longer referenced by any other term can be reused via garbage collection.

In [14], we introduced a garbage collection mechanism for this, and in the current article, we propose a second approach. These correspond to the `queueCollector` and `streamCompactCollector` as referenced in Listing 3. In this section, we explain how those garbage collectors work, and how a new index can be retrieved to store a new term, which is done by the `getNewIndex` procedure (called, for instance, in Listing 5).

Queue collector This collector maintains a list of indices that can be reused to store new terms. It monitors whether some indices of deleted terms need to be gathered in the `keyIndices` list. The collector is described in Listing 7. After calculating the required number of blocks, the `collectFree` kernel is called (line 3). In this kernel, if a thread detects that the reference counter of its term is 0 (line 8), then this term is set for removal, and the reference counters of its direct subterms should be decremented. This is done in the **for**-loop at lines 9–13. Next, a position in the `keyIndices` array must be retrieved to store the index of the term, such that it can be reused in the future. The `next_key_index` counter points to the head of this list, which is used as a stack. It is atomically incremented (14), by which a position in `keyIndices` is retrieved for this thread. At this position, the index is stored (line 15). Finally, the `subterm_index` of this term is reset, and the reference counter is set to the special value -2, which prevents additional reference count decrements of the term's direct subterms in subsequent iterations of the garbage collector.

Listing 8 shows how we retrieve a new index when the `queueCollector` is used. The `queueCollector` stores the indices of positions in the `term` array that can be reused in the `keyIndices` array. If this array is not empty (line 5), `next_key_index` is atomically decremented to claim the next index in the `keyIndices` array (line 6). Again, note that this array is used as a stack. If this decrement was not performed too late (other threads might have claimed all available indices in the meantime), the index is stored in `new_id` (lines 7–8). Otherwise, it must be ensured that `next_key_index` is 0 instead of some negative value (line 9). If a new index was not retrieved, a new index must be obtained from the end of the current list of terms, which can be done via the variable `nextFree` (line 12). Finally, for the newly obtained index, the reference count must be updated again from -2 to 0.

Note that there can be data races when `next_key_index` is accessed, but these are all benign: the accesses at line 6 are atomic, and therefore decrement the value of `next_key_index` in a thread-safe way, but the accesses at lines 3 and

```
1   procedure streamCompactCollector(...) {
2      compactIf(term, refcounts, n);
3      getIndices(term, keyIndices, n);
4      updateTerms(term, flags, keyIndices, n);
5   }
6
7   Kernel getIndices(term, keyIndices, n) {
8      if (tid >= n) return;
9      oldIndex = term[tid].index;
10     keyIndices[oldIndex] = tid;
11     backup(flags[oldIndex]);
12  }
13
14  Kernel updateTerms(term, flags, keyIndices, n) {
15     if (tid >= n) return;
16     oldindex = term[tid].index;
17     term[tid].index = tid;
18     term[tid].arg_1 = keyIndices[term[tid].arg_1];
19     term[tid].arg_2 = keyIndices[term[tid].arg_2];
20     ...
21     restore(flags[tid], flags[oldIndex]);
22  }
```

Listing 9: The Stream Compaction garbage collector.

9 are not atomic. When the current value of `next_key_index` is read at line 3, it is only relevant whether the value is bigger than 0 or not (line 5), since in either case, the obtained value is no longer used. If the value is bigger than 0, lines 5–10 are executed, and `next_key_index` is again accessed at line 6 to obtain a value in a thread-safe way. If the value is not bigger than 0, `n_begin` is no longer used. The write access at line 9 is only performed if a value smaller than or equal to 0 was read at line 6. If multiple threads perform this write access simultaneously, they still all write the same value (0), making any data races benign. Each write performed at line 6 by a thread t_1 that results in a negative value will eventually be followed by a write at line 9 by t_1, resetting the value to 0 again.

If a thread t_1 reads a value bigger than 0 at line 3, and another thread t_2 writes a value smaller than or equal to 0 at either line 6 or line 9 after t_1 has executed line 5 but before it has executed line 6, t_1 will read this new value at line 6, and subsequently not use it anymore.

Stream compaction collector An alternative to maintaining a list of indices where new terms can be stored, is to always ensure that the terms still in use are stored consecutively in the `term` array. Based on this idea, we have designed a second garbage collector, presented in Listing 9. It takes advantage of stream compaction technology [26]. The collector aims to compactly store the non-deleted terms by shifting out free locations while preserving the original order. The entire garbage collection procedure is kept free of atomic operations.

At line 2, the `compactIf` function deploys the `DeviceSelect::If` standard function of the `CUB` library to compact both the `term` and `refcounts` arrays, keeping their values in consecutive positions via stream compaction. At line 3, the `getIndices` kernel (lines 7–12) is launched to store the current indices of the compacted terms in the `keyIndices` array. Once this kernel has been executed, each thread can get the new index of a term with the old index `i` by inspecting the value at `keyIndices[i]`. Thus, the new index of a term can be retrieved from position `term[tid].index` in the `keyIndices` array. At line 11, the `nf` and `nfRead` flags are saved to their auxiliary bits at the old term position. This information can later be retrieved to update the terms and their corresponding normal form flags. This is done via a separate call to the `updateTerms` kernel at line 4. This kernel is described at lines 14–22. In this kernel, the indices stored in each term `term[tid]` are updated using the `keyIndices` array (lines 16–21). At line 21, the flags of each term are restored.

Since all terms are compacted (i.e., moved to fill in the gaps left by deleted terms), the `nextFree` pointer can always be used when an index for a new term is needed. When a new term needs to be stored, we atomically increment the `nextFree` pointer in a similar way to line 12 in Listing 8. Hence, when the `streamCompactCollector` is used, the `getNewIndex` procedure only entails this plus the resetting of the reference counter (line 13 in Listing 8).

6. Evaluation

In this section, we provide insight into the performance of the GPU rewriter. Initially, we do this in two ways:

1. We compare our GPU rewriter with innermost rewriting and garbage collection using the `queueCollector` with a sequential recursive leftmost innermost rewriter for the CPU.
2. We analyze to what extent we make good use of the GPU resources.

Because CPUs and GPUs differ widely in architecture, it is often subject of debate whether a comparison is fair [34]. We therefore include the second way of evaluating.

Table 1
Comparison of the CPU and GPU.

Type	Year	Name	Mem (GB)	BW aligned (GiB/s)	BW random (GiB/s)
CPU	2017	Intel Core i5-7600	32	25.7	0.607
GPU	2018	NVIDIA Titan RTX	24	555	22.8

```
sort Nat = struct Zero() | S(Nat);
     Tree = struct Leaf(Nat) | Node(Tree,Tree) | Multl(Nat, Tree) |
              Visit(Tree);

var X : Nat; O, P : Tree;

eqn
  Multl(Zero() , O) = O;
  Multl(S(X), O) = Node(Multl(X, O), Multl(X, O));

  Visit(Node(O, P)) = Node(Visit(O), Visit(P));
  Visit(Leaf(X)) = Leaf(X);

input Visit(Multl(S(...(Zero())...)), Leaf(Zero()));
```

Listing 10: An MTRS for Tree construction and exploration.

Table 1 shows a comparison of the used CPU and GPU. CPUs are optimised for latency: finish the program as soon as possible. In contrast, GPUs are optimised for throughput: process as many elements per time unit as possible. For GPUs, parallelism is explicit, one instruction is issued for multiple threads, and the architecture is specifically designed to hide memory latency times by scheduling new warps immediately after a memory access. The differences between architectures are highlighted by the last two columns that show that the bandwidth of the GPU for aligned access is vastly superior to that of the CPU. Even the bandwidth for random accesses on the GPU almost reaches the bandwidth for aligned accesses on the CPU.

We measure the performance of the CPU and GPU rewriter in *rewritten terms per second*. Given an MTRS, both the GPU and the CPU rewriter are generated by a `Python 2.7` script. The script uses `TextX` [35] and `Jinja 2.11`[2] to parse an MTRS and generate a $\texttt{rewrite}_f$ function for every rewritable head symbol f in the MTRS. The code generated for the GPU rewriter is CUDA C++ with CUDA platform 10.1. For the CPU rewriter the code generated is in C++. The same $\texttt{rewrite}_f$ functions are used, and thus the rewrite rules are equal and the CPU and GPU implementations rewrite the same number of terms.[3]

We evaluate the GPU rewriter using three different MTRSs. The first MTRS, shown in Listing 10, represents the construction and exploration of binary trees. The `Multl` function is used to construct a tree with the depth specified in the input term as the first parameter of the `Multl` function. At each leaf, the number 0 is stored, but this value is not really important. With the `Visit` function, starting at the root of the tree, all the nodes are visited until all leafs have been reached. This is an interesting MTRS, as for the GPU term rewriter, in which terms are rewritten in parallel, it expresses the parallel exploration of trees. The potential for parallelism grows exponentially w.r.t. the depth of the tree. In case we apply this MTRS on a term that expresses that a binary tree of depth N needs to be constructed and explored, we refer to that case as *Tree exploration N*.

The second MTRS we use has been constructed to showcase massive parallelism. Listing 11 presents this MTRS. The `Tree` type consists of nodes and leaves in the form of symbols `A` through `Z` and `End`. A tree of depth N with `A`'s in the leaves is generated by rewriting the term `Expand(N)`. The symbol `A` is rewritten to a `B` and this is rewritten again until `End` is reached. This is done for all leaves, and all leaves can be processed simultaneously, rewriting all `A`'s to `B`'s in one single massively parallel rewrite step, subsequently rewriting all `B`'s in one massively parallel rewrite step, until `End` is reached. The two rules `Expand` and `Expand2` are technically the same function but are used to prevent all the leaves from being represented by a single term due to sharing. If a single term would represent all the leaves, there would be no parallelism left. We refer to a term on which this MTRS is applicable with the name *Transformation tree N*, with N the depth of the tree.

Finally, Listing 12 presents our third MTRS, which can be used to check that binary trees with each node having a Peano number associated with it are binary search trees (BSTs). Such a binary tree is a BST iff for each node, its associated value n implies that every value in its left subtree is smaller than or equal to n, and every value in its right subtree is greater than or equal to n. `Depth` defines the depth of the input tree. The term `InputTree` is used to unfold the input tree, using its associated rules. The BST verification is done via the rules for the `BST` term. The usual rules for operations such as `And`, `Geq` and `Leq` have been omitted. As with the other cases involving trees, we scale this case w.r.t. the depth of the tree,

[2] https://jinja.palletsprojects.com.

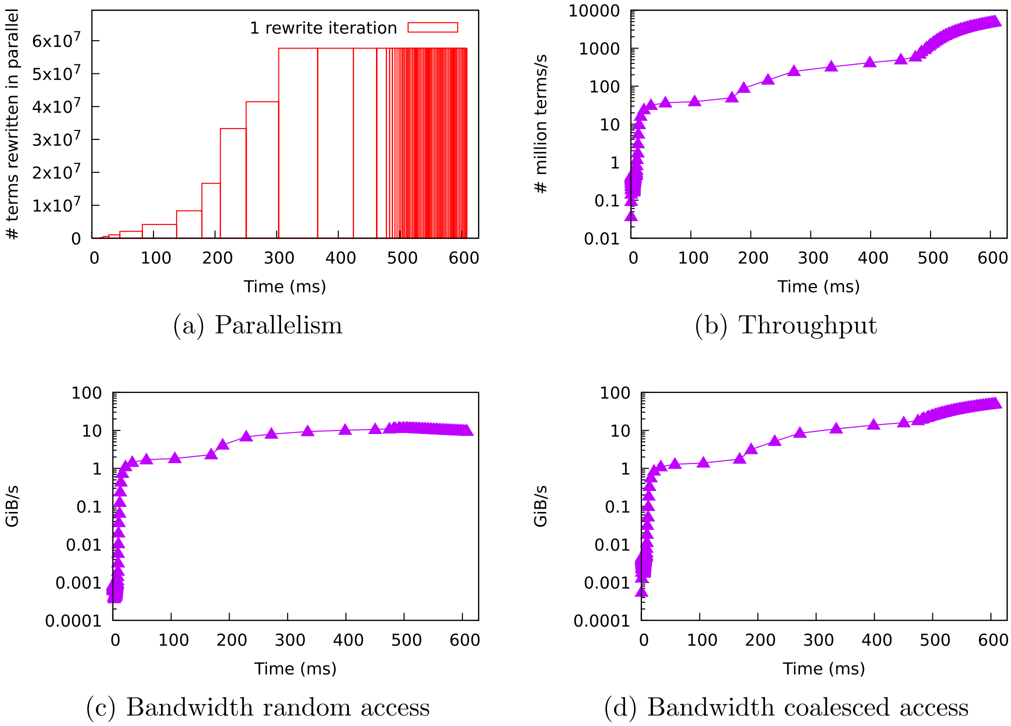
[3] The code of the generator, plus benchmark TRSs, can be found at https://www.win.tue.nl/~awijs/gpu-rewriter.

```
sort Nat = struct Zero() | Suc(Nat);
     Tree = struct A() | B() | ... | Z() | End() |
     Node(Tree, Tree) | Expand(Nat) | Expand2(Nat);

var X : Nat; O, P : Tree;

eqn
  Expand(Zero()) = A();
  Expand(S(X)) = Node(Expand(X), Expand2(X));

  Expand2(Zero()) = A();
  Expand2(S(X)) = Node(Expand(X), Expand2(X));

  A() = B();
  B() = C();
  C() = D();
  D() = E();
  ...
  Z() = End();

input Expand(Suc(Suc(Suc(...Suc(Zero())...))));
```

Listing 11: An MTRS for Transformation trees.

```
sort Nat = struct Zero() | S(Nat) | ...;
     Bool = struct True() | False() | BST(NatTree, Nat, Nat) |
     Ternary_And(...) | BST_Rel(...) | ...;
     NatTree = struct Leaf(Nat) | Tree(Nat, NatTree, NatTree) | ...;

var Levels, X, Y, Z : Nat; B, B2 : Bool; T_l, T_r : NatTree;

eqn
  Geq(Min(), Y) = False();
  Geq(Max(), Y) = True();
  ...

  Leq(Min(), Y) = True();
  Leq(Max(), Y) = False();
  ...

  BST(Leaf(X), Y, Z) = BST_Rel(X, Y, Z);
  BST(Tree(X, T_l, T_r), Y, Z) = Ternary_And(BST_Rel(X, Y, Z),
                              BST(T_l, Y, X), BST(T_r, X, Z));

  BST_Rel(X, Y, Z) = And(Leq(Y, X), Geq(Z, X));

  Ternary_And(False(), B, B2) = False();
  Ternary_And(B, False(), B2) = False();
  Ternary_And(B, B2, False()) = False();

  Ternary_And(True(), B, B2) = And(B, B2);
  ...

  Plus(Zero(), Y) = Y;
  Plus(S(X), Y) = S(Plus(X, Y));

  Mult(Zero(), Y) = Zero();
  Mult(S(X), Y) = Plus(Y,Mult(X, Y));

  InputTree(Zero(), X, Y, Z) = Tree(X, Leaf(Y), Leaf(Z));
  InputTree(S(Levels), X, Y, Z) = Tree(X, InputTree(Levels, X, Y, Z),
                               InputTree(Levels, X, Y, Z));

  Depth() = S(Zero);

input BST(InputTree(Depth(), S(S(Zero())), S(S(Zero())),
        S(S(Zero()))), Min(), Max());
```

Listing 12: An MTRS for Binary Search Tree checking of a binary tree of depth 1.



(a) Parallelism

(b) Throughput

(c) Bandwidth random access

(d) Bandwidth coalesced access

Fig. 1. Runtime results for *Binary search tree 22*.

Table 2
Performance of three instances of different rewrite systems.

Application	CPU rewritten terms/s	GPU rewritten terms/s	Speedup
Tree exploration 23	1.25×10^8	73.81×10^8	**59**×
Transformation tree 22	3.32×10^8	85.22×10^8	**26**×
Binary search tree 22	1.04×10^8	47.76×10^8	**48**×

hence the size of the input term for this MTRS grows exponentially as its depth is increased linearly. With *Binary search tree N*, we refer to this MTRS being applied on a binary tree of depth N.

Fig. 1 shows an application of the BST MTRS applied on a tree structure of 22 levels deep (*Binary search tree 22*). The width of each box (Plot 1a) represents the time of a GPU rewrite step (the `derive` statement at line 8 in Listing 3) whereas its height represents how many terms are rewritten in parallel in this rewrite step. Plots 1a and 1b reveal high degrees of parallelism resp. throughput for the majority of the execution time. The latter reaches a peak of around 4 billion terms rewritten per second. Plots 1c and 1d highlight to what extent we use the capabilities of the GPU. Usually, the performance of a GPU is measured in GFLOPS, floating point operations per second, for compute intensive applications or GiB/s for data intensive applications. Since term rewriting is a symbolic manipulation that does not involve any arithmetic, it is data intensive. From Table 1 we have seen that the maximum bandwidth our GPU can achieve is 555 GiB/s for aligned accesses and 22.8 GiB/s for random accesses. Since term rewriting is an irregular problem with a high degree of random accesses (to direct subterms that can be anywhere in memory), we focus on the bandwidth for random accesses.

As shown by Plot 1c for *Binary search tree 22*, the overall random access bandwidth of the GPU implementation reaches 15 GiB/s, which is close to the benchmarked bandwidth. Regarding coalesced accesses, a bandwidth of 50 GiB/s is measured, see Plot 1d. This confirms that term rewriting is indeed an irregular problem and that random accesses form the main bottleneck.

Table 2 shows the GPU performance for the three previously described MTRSs, where we select one input term for each MTRS. The GPU achieves a considerable speedup compared to the CPU sequential performance for all three of them. For instance, for the generation and exploration of a binary tree of 23 levels deep (*Tree exploration 23*), 7 billion terms are rewritten per second.

Next, we experiment with several configurations of the GPU rewriter, involving the `streamCompactCollector` and relaxed innermost rewriting. We compare the following configurations:

1. The sequential (CPU) recursive leftmost innermost rewriter.
2. The GPU rewriter with innermost rewriting without garbage collection.

Table 3
Performance comparison of different rewriters on various MTRS benchmarks. The QC and SCC acronyms stand for Queue Collector and Stream Compaction Collector, respectively. The Rewritten and Rewrites keywords stand for the number of terms rewritten and the number of rewriting iterations, respectively. The time measurements are averaged over 10 runs in milliseconds. The bold measurements identify the fastest running times.

MTRS	Rewritten	CPU	GPU w/o QC		GPU w/ QC		GPU Relax w/ QC		GPU w/ SCC	
		Time	Rewrites	Time	Rewrites	Time	Rewrites	Time	Rewrites	Time
Tree exploration 21	6,291,519	58.0	108	74.7	108	21.9	109	22.9	108	77.3
Tree exploration 22	12,582,978	115.9	113	139.8	113	37.8	114	**36.5**	113	143.2
Tree exploration 23	25,165,893	232.6	118	199.0	118	67.3	119	**65.2**	118	209.2
Transformation tree 22	121,634,836	366.1	94	50.3	94	50.2	94	**49.4**	94	198.9
Transformation tree 23	243,269,653	637.4	97	91.4	97	93.0	97	**90.1**	97	252.6
Transformation tree 24	486,539,286	1417.5	100	175.9	100	178.5	100	**169.3**	100	353.2
Binary search tree 20	41,943,046	391.5	139	255.6	139	**205.4**	194	264.8	139	261.6
Binary search tree 21	83,886,087	794.1	143	345.5	143	**307.5**	204	412.4	143	369.7
Binary search tree 22	167,772,168	1583.7	161	587.9	161	618.2	210	**558.8**	161	594.1

3. The GPU rewriter with innermost rewriting and Queue garbage collecting.
4. The GPU rewriter with relaxed innermost rewriting and Queue garbage collecting.
5. The GPU rewriter with innermost rewriting and Stream Compaction garbage collecting.

We also considered involving the GPU rewriter with relaxed innermost rewriting and Stream Compaction garbage collecting in our results, but that configuration does not lead to new insights. We address this when we discuss the following results.

To investigate the scalability of these configurations, we select three different input terms for each of the three MTRSs described earlier. Table 3 assembles all our results for the previous configurations. The "Rewritten" column refers to the number of terms that have been rewritten before the term rewriting terminates. For each case, the order in which terms are rewritten differs between configurations, but the total number of rewrites performed is the same for all configurations. The "Rewrites" columns refer to the number of rewriting iterations performed, i.e., the number of iterations through the loop in Listing 3. This is only relevant for the GPU configurations, as the CPU rewriter works recursively.

Table 3 shows that the GPU rewriter with relaxed innermost rewriting outperforms the CPU by more than a factor of seven for a Transformation tree of 22 levels deep, with four million leaves. Frequently, the relaxed innermost rewriting strategy positively influences the GPU rewriter, see column 9. Regarding garbage collection, we observe that the `streamCompactCollector` (SCC) starts to pay off as the amount of garbage collection work increases, as demonstrated by the *Binary search tree* cases (compare columns 7 and 11). Unfortunately, we cannot scale up this system any further beyond 22 levels as this requires more than the 24 GB of GPU memory available on our device. We expect SCC to outperform the `queueCollector` (QC) when there are sufficient hardware resources to process larger *Binary search tree* cases.

On the other hand, the QC seems to be more effective for the other benchmarks, and for several cases, applying QC even results in better coalesced memory accesses compared to the GPU rewriter without garbage collection (see columns 7 and 5, respectively). Hence, not only does garbage collection help to reuse memory and thereby reduce the memory requirements, it also improves the performance. For example, when QC is enabled for the *Binary search tree 20* benchmark, the running time decreases by 34%, even though the garbage collection adds extra overhead to the rewrite procedure.

We have not included results for the GPU rewriter with relaxed innermost rewriting and Stream Compaction garbage collecting. That combination provides no new insights: as expected, its runtime results are always between innermost rewriting with Stream Compaction garbage collecting, and relaxed innermost rewriting with Queue garbage collecting.

If we consider the potential for GPU term rewriting in the future, then we have to conclude that to achieve even higher performance, it is necessary to introduce more regularity into the implementation, reducing the random memory accesses. Other often used strategies to improve graph algorithms, such as reducing thread divergence, will probably not yield significant performance increase, due to the random accesses being the main performance bottleneck. In addition, the results we present clearly show the different capabilities of GPUs and CPUs. An interesting direction for future work is to create a hybrid rewrite implementation that can switch to a GPU implementation whenever a high degree of parallelism is available.

CRediT authorship contribution statement

Johri van Eerd: Formal analysis, Investigation, Methodology, Software, Validation, Writing – original draft. **Jan Friso Groote:** Conceptualization, Funding acquisition, Methodology, Supervision, Writing – original draft, Writing – review & editing. **Pieter Hijma:** Formal analysis, Funding acquisition, Investigation, Methodology, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Jan Martens:** Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Muhammad Osama:** Formal analysis, Investigation, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Anton Wijs:** Formal analysis, Funding acquisition, Investigation, Methodology, Resources, Software, Supervision, Validation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is carried out in the context of the NWO AVVA project 612.001751.

References

[1] M. Khairy, A. Wassal, M. Zahran, A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity, J. Parallel Distrib. Comput. 127 (2019) 65–88.

[2] D. Bošnački, S. Edelkamp, D. Sulewski, A.J. Wijs, GPU-PRISM: An extension of PRISM for general purpose graphics processing units, in: PDMC-HiBi, IEEE Computer Society, 2010, pp. 17–19.

[3] D. Bošnački, M.R. Odenbrett, A.J. Wijs, W.P.A. Ligtenberg, P.A.J. Hilbers, Efficient reconstruction of biological networks via transitive reduction on general purpose graphics processors, BMC Bioinform. 13 (281) (2012).

[4] M. Osama, A.J. Wijs, GPU acceleration of bounded model checking with ParaFROST, in: CAV 2021, Los Angeles, July 18 - 24, 2021, Proceedings, Part II, in: LNCS, Springer International Publishing, 2021, pp. 447–460.

[5] M. Osama, L. Gaber, A.I. Hussein, H. Mahmoud, An efficient SAT-based test generation algorithm with GPU accelerator, J. Electron. Test. 34 (5) (2018) 511–527, https://doi.org/10.1007/s10836-018-5747-4.

[6] M. Osama, A.J. Wijs, A. Biere, SAT solving with GPU accelerated inprocessing, in: TACAS 2021, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I, in: LNCS, vol. 12651, Springer, 2021, pp. 133–151.

[7] M. Osama, A.J. Wijs, Parallel SAT simplification on GPU architectures, in: TACAS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I, in: LNCS, vol. 11427, Springer, 2019, pp. 21–40.

[8] S. Heldens, P. Hijma, B. Van Werkhoven, J. Maassen, A.S.Z. Belloum, R.V. Van Nieuwpoort, The landscape of exascale research: a data-driven literature analysis, ACM Comput. Surv. 53 (2) (Mar. 2020).

[9] O. Bunte, J.F. Groote, J.J.A. Keiren, M. Laveaux, T. Neele, E.P. de Vink, W. Wesselink, A.J. Wijs, T.A.C. Willemse, The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability, in: TACAS, in: LNCS, vol. 11428, Springer, 2019, pp. 21–39.

[10] F. Durán, H. Garavel, The rewrite engines competitions: a RECtrospective, in: TACAS, in: LNCS, vol. 11429, Springer, 2019, pp. 93–100.

[11] Term Terese, Rewriting Systems, Cambridge Tracts in Theoretical Computer Science, vol. 55, Cambridge University Press, 2003.

[12] J.W. Klop, Term rewriting systems, in: Handbook of Logic in Computer Science (vol. 2): Background: Computational Structures, Oxford University Press, 1993, pp. 1–116, Ch. 1.

[13] R. Nasre, M. Burtscher, K. Pingali, Data-driven versus topology-driven irregular computations on GPUs, in: IPDPS, May 20-24, IEEE Computer Society, 2013, pp. 463–474.

[14] J. Van Eerd, J.F. Groote, P. Hijma, J. Martens, A.J. Wijs, Term rewriting on GPUs, in: FSEN, in: LNCS, vol. 12818, Springer, 2021, pp. 175–189.

[15] S.L. Peyton Jones, C.D. Clack, J. Salkild, M. Hardie, GRIP – a high-performance architecture for parallel graph reduction, in: FPCA, in: LNCS, vol. 274, Springer, 1987, pp. 98–112.

[16] H. Barendregt, M. van Eekelen, M. Plasmeijer, P. Hartel, L. Hertzberger, W. Vree, The Dutch parallel reduction machine project, Future Gener. Comput. Syst. 3 (4) (1987) 261–270.

[17] P. Hijma, R.V. Van Nieuwpoort, C.J.H. Jacobs, H.E. Bal, Stepwise-refinement for performance: a methodology for many-core programming, Concurr. Comput.: Pract. Exper. 27 (17) (2015) 4515–4554.

[18] T.L. McDonell, Optimising purely functional GPU programs, Ph.D. thesis, University of New South Wales, Sydney, Australia, 2015.

[19] T. Henriksen, N.G.W. Serup, M. Elsman, F. Henglein, C.E. Oancea, Futhark: purely functional GPU-programming with nested parallelism and in-place array updates, in: PLDI, ACM, 2017, pp. 556–571.

[20] T. Henriksen, M. Elsmanand, C.E. Oancea, Modular acceleration: tricky cases of functional high-performance computing, in: FHPC, ACM, 2018, pp. 10–21.

[21] A.J. Wijs, T. Neele, D. Bošnački, GPUexplore 2.0: unleashing GPU explicit-state model checking, in: FM, in: LNCS, vol. 9995, 2016, pp. 694–701.

[22] A.J. Wijs, D. Bošnački, GPUexplore: Many-core on-the-fly state space exploration using GPUs, in: TACAS, in: LNCS, vol. 8413, Springer, 2014, pp. 233–247.

[23] R. Nasre, M. Burtscher, K. Pingali, Morph algorithms on GPUs, in: PPOPP, ACM Sigplan, 2013, pp. 147–156.

[24] M. Maas, P. Reames, J. Morlan, K. Asanovic, A.D. Joseph, J. Kubiatowicz, GPUs as an opportunity for offloading garbage collection, in: ISMM, Beijing, China, June 15-16, 2012. Proceedings, ACM, 2012, pp. 25–36.

[25] R. Nasre Abhinav, FastCollect: offloading generational garbage collection to integrated GPUs, in: CASES, Pittsburgh, Pennsylvania, USA, October 1-7, 2016. Proceedings, ACM, 2016, pp. 21:1–21:10.

[26] M. Billeter, O. Olsson, U. Assarsson, Efficient stream compaction on wide SIMD many-core architectures, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics 2009, New Orleans, Louisiana, USA, August 1-3, 2009, Eurographics Association, 2009, pp. 159–166.

[27] M. Springer, H. Masuhara, Massively parallel GPU memory compaction, in: ISMM, Phoenix, AZ, USA, June 23-23, 2019. Proceedings, ACM, 2019, pp. 14–26.

[28] J. Meseguer, Membership algebra as a logical framework for equational specification, in: WADT, in: LNCS, vol. 1376, Springer, 1997, pp. 18–61.

[29] J. Goguen, Modular algebraic specification of some basic geometrical constructions, Artif. Intell. 37 (1–3) (1988) 123–153.

[30] W. Fokkink, J. Kamperman, P. Walters, Within ARM's reach: compilation of left-linear rewrite systems via minimal rewrite systems, ACM Trans. Program. Lang. Syst. 20 (3) (1998) 679–706.

[31] J. Kamperman, Compilation of term rewriting systems, Ph.D. thesis, University of Amsterdam, 1996.

[32] G. Huet, D. Lankford, On the Uniform Halting Problem for Term Rewriting Systems, Rapport de Recherche No 283 (March 1978), IRIA, 1978.

[33] J. Cheng, M. Grossman, Professional CUDA C Programming, John Wiley and Sons Ltd., 2014.

[34] V.W. Lee, P. Hammarlund, R. Singhal, P. Dubey, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, Debunking the 100X GPU vs. CPU myth, in: ISCA, ACM Press, 2010, pp. 451–460.

[35] I. Dejanović, R. Vaderna, G. Milosavljević, Ž. Vuković, TextX: a Python tool for Domain-Specific Languages implementation, Knowl.-Based Syst. 115 (2017) 1–4.