



HAL
open science

Term Rewriting on GPUs

Johri Van Eerd, Jan Friso Groote, Pieter Hijma, Jan Martens, Anton Wijs

► **To cite this version:**

Johri Van Eerd, Jan Friso Groote, Pieter Hijma, Jan Martens, Anton Wijs. Term Rewriting on GPUs. 9th International Conference on Fundamentals of Software Engineering (FSEN), May 2021, Virtual, Iran. pp.175-189, 10.1007/978-3-030-89247-0_12 . hal-04074530

HAL Id: hal-04074530

<https://inria.hal.science/hal-04074530v1>

Submitted on 19 Apr 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.





Distributed under a Creative Commons Attribution 4.0 International License



This document is the original author manuscript of a paper submitted to an IFIP conference proceedings or other IFIP publication by Springer Nature. As such, there may be some differences in the official published version of the paper. Such differences, if any, are usually due to reformatting during preparation for publication or minor corrections made by the author(s) during final proofreading of the publication manuscript.

Term Rewriting on GPUs

Johri van Eerd¹, Jan Friso Groote ², Pieter Hijma^{2,3},
Jan Martens², and Anton Wijs² 

¹ Verum Software Tools BV, The Netherlands
johri.van.eerd@verum.com

² Eindhoven University of Technology, The Netherlands
{j.f.groote,j.j.m.martens,a.j.wijs}@tue.nl

³ VU Amsterdam, The Netherlands
pieter@cs.vu.nl

Abstract. We present a way to implement term rewriting on a GPU. We do this by letting the GPU repeatedly perform a massively parallel evaluation of all subterms. We find that if the term rewrite systems exhibit sufficient internal parallelism, GPU rewriting substantially outperforms the CPU. Since we expect that our implementation can be further optimized, and because in any case GPUs will become much more powerful in the future, this suggests that GPUs are an interesting platform for 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.

Keywords: Term rewriting, GPU, programming, parallel computing

1 Introduction

Graphics Processing Units (GPUs) increase in computational power much faster than the classical CPUs. GPUs are optimized 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 [3,4]). It is not without reason that modern super computers have large banks of graphical processors installed in them [10]. GPU designers realize 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 utilize the power of the GPU. For irregular problems it is the programmer's task to recognize 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 [5] and the performance of term rewriting is a long-standing and important objective [8].

We recall that a term rewriting system that enjoys the Church-Rosser property is parallel in nature, in the 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. Every processor can independently work on its own section of the system and do its evaluation there. In this paper, we investigate whether and under which conditions 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.

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 [19]. 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 [2]. 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.

Besides their use in the formal methods community, a term rewriting system is also a simple, yet universal mechanism for computation [20]. A question that follows is whether this model for computation can be used to express programs for GPUs more easily.

Current approaches for GPU programming are to design a program at a highly abstract level first and transform it in a stepwise fashion to an optimal GPU program [13]. 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 [16], an embedded array processing language for Haskell, and Futhark [12], 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, [11,22].

We designed experiments and compared GPU rewriting with CPU rewriting of the same terms. We find that our implementation manages to employ 80% of

the bandwidth of the GPU for random accesses. For rewriting, random accesses are the performance bottleneck, and therefore our implementation uses the GPU 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 subterms that can be evaluated in parallel is reduced, rewriting slows down quite dramatically. This 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.

This leads us to the following conclusion. Term rewriting 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 optimizations 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. [17]. 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 term rewriting processes is promising. It allows for abstract programming independent of the hardware details of GPUs, and it offers the potential of evaluating appropriate rewrite systems at least one order, and in the future orders of magnitude faster than a CPU.

Related to this work is the work of Nasre et al. [18] 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.

2 Preliminaries

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

Term rewrite systems A *Term Rewrite System (TRS)* is a set of rules. Each rule is a pair of terms, namely a left hand side and a right hand side. Given an arbitrary term t and a TRS R , rewriting means to replace occurrences in t of the left hand side of a rule in R by the corresponding right hand side, and then repeating the process on the result.

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 arguments or subterms. We refer to this number as the *arity* of the function symbol, and denote the arity of a function symbol f by $ar(f)$. If $ar(f) = 0$, we say f is a *constant*. Together, the sets V and F constitute the *signature* $\Sigma = (F, V)$ of a TRS. 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 $t_i \in T_\Sigma$ for $1 \leq i \leq ar(f)$, then $f(t_1, \dots, t_{ar(f)}) \in T_\Sigma$.

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

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

Definition 1 (Term rewrite system). *A TRS 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) $Var(r) \subseteq Var(l)$.*

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 . A TRS with rules not satisfying this assumption can be rewritten to one that does not contain such rules. 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).

Definition 2 (Substitution). *For a TRS R over a signature $\Sigma = (F, V)$, a substitution $\sigma : V \rightarrow T_\Sigma$ maps variables to terms. 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 $f(t_1\sigma, \dots, t_{ar(f)}\sigma)$ if $t = f(t_1, \dots, t_{ar(f)})$.*

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 $t = l\sigma$.

As an example, Listing 1.1 presents a simplified version of a merge sort rewrite system with an input tree of depth 2 consisting of empty lists (*Nil*). After the `sort` keyword, a list is given of all 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. In this example, all rewrite rules for functions on (Peano) numbers and Booleans are omitted, such as the less than (*Lt*) rule for natural numbers and the *Even* and *Odd* rules for lists, which create lists consisting of all elements at even and odd

Listing 1.1. A TRS for merge sort in a binary tree of lists

```

1  sort  List = Nil() | Cons(Nat, List) | Sort(List) | ...;
2       Tree = Leaf(List) | Node(Tree, Tree);
3       Nat = Zero() | S(Nat);
4
5  var X : Nat; Y : Nat; L : List; M : List;
6
7  eqn   Merge(Nil(), M) = M;
8       Merge(L, Nil()) = L;
9       Merge(Cons(X, L), Cons(Y, M)) = Merge2(Lt(X,Y), X, L, Y, M);
10
11      Merge2(True(), X, L, Y, M) = Cons(X, Merge(L, Cons(Y, M)));
12      Merge2(False(), X, L, Y, M) = Cons(Y, Merge(Cons(X, L), M));
13
14      Sort(L) = Sort2(Gt(Len(L), S(Zero())), L);
15      Sort2(False(), L) = L;
16      Sort2(True(), L) = Merge(Sort(Even(L)), Sort(Odd(L)));
17
18  input Node(Leaf(Sort(...)), Leaf(Sort(...)));

```

Listing 1.2. A derivation procedure, and a rewrite procedure for head symbol f

```

1  procedure derive(t, R):
2    while ¬nf(t) do
3      for i ∈ {1, ..., ar(t)} do
4        if ¬nf(subi(t)) then
5          derive(subi(t))
6        t ← rewritehs(t)(t, R)
7
8  procedure rewritef(t, R):
9    rewritten ← false
10   for (l → r) ∈ {(l, r) ∈ R | hs(l) = f} do
11     if ∃σ : V → TΣ. lσ = t then
12       t ← rσ; rewritten ← true; break
13   if ¬rewritten then nf(t) ← true
14   return t

```

positions in the given list, respectively. The 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 complete TRS is given in the extended version of the paper [21, Appendix A].

A TRS is *terminating* iff there are no infinite reductions possible. For instance, the rule $f(a) \rightarrow f(f(a))$ leads to an infinite reduction. In general, determining whether a given TRS is terminating is an undecidable problem [14].

The computation of a term in a terminating TRS 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 $r = (f(f(x)) \rightarrow a)$ and the term $t = f(f(f(a)))$. Applying r on t may result in either the normal form a or $f(a)$, depending on the chosen reduction. To make rewriting deterministic, a *rewrite strategy* is needed. We focus on the *inner-most* strategy, which gives priority to selecting redexes that do not contain other redexes. In the example, this means that the LHS of r is matched on the inner $f(f(a))$ of t , leading to $f(a)$.

Algorithmically, (inner-most) rewriting is typically performed using recursion. Such an algorithm is presented in Listing 1.2. As long as a term t is not in normal form (line 2), it is first checked whether all its subterms are in normal form (lines 3-4). For each subterm not in normal form, `derive` is called recursively (line 5), by which the inner-most rewriting strategy is achieved. If the subterms are checked sequentially from left to right, we have *left-most* inner-most rewriting. A parallel rewriter may check the subterms in parallel, since inner-most redexes do not contain other redexes. Once all subterms are in normal form, the procedure `rewritehs(t)` is called (line 6).

For each head symbol of the TRS, we have a dedicated rewrite procedure. The structure of these procedures is also given in Listing 1.2. The variable `rewritten` is used to keep track of whether a rewrite step has been performed (line 9). For each rewrite rule (l, r) with $hs(l) = f$, it is checked whether a match between l and t exists, and if so, $l \rightarrow r$ is applied on t (lines 10-12). If no rewrite rule was applicable, it is concluded that t is in normal form (line 13).

GPU basics. In this paper, 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, [6].

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 4608 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.

Threads have access to different kinds of memory. Each thread has a number of on-chip registers to store thread-local data. It allow 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.

Threads are executed using the SIMT model. This means that each thread is executed independently with its own instruction address and 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 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.

3 A GPU algorithm for term rewriting

In this section, we address how a GPU can perform inner-most term rewriting to get the terms of a given TRS 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 so-called *topology-driven* algorithm [17], 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 on a TRS, 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.

First, we discuss how TRSs are represented on a GPU. Typically, GPU data structures, such as matrices and graphs, are array-based, and we also store a TRS in a collection of arrays. Each term is associated with a unique index i , and each of 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 head symbol of their term, for instance, they will access consecutive elements of the array that stores head symbols. We introduce the following GPU data structures that reside in global memory:

Listing 1.3. The main loop of the rewrite algorithm, executed by the CPU

```

1  h_done = false;
2  while (!h_done) {
3    done ← h_done;
4    numBlocks = n / blockSize;
5    refcounts_read = refcounts;
6    nf_read = nf;
7    derive<<numBlocks, blockSize>>(nf, nf_read, hss, arg0, ...);
8    h_next_fresh ← next_fresh;
9    if (h_next_fresh > 0) {
10     n = n + h_next_fresh;
11     next_fresh ← 0;
12   }
13   h_done ← done;
14   h_garbage_collecting ← garbage_collecting;
15   if (h_garbage_collecting) {
16     collect_free_indices <<numBlocks, blockSize>>(...);
17     garbage_collecting ← false;
18   }
19 }

```

- Boolean arrays `nf` and `nf_read` keep track of which terms are in normal form, the first is used for writing and the second for reading;
- Integer variable `n` provides the current number of terms;
- Array `hss` stores the head symbols of all the terms;
- Constant `maxarity` refers to the highest arity among the function symbols in F ;
- Arrays `arg0, …, argmaxarity-1` store the indices of the subterms of each term. Index 0 is never used. If `argj[i] = 0`, for some $0 \leq j < \text{maxarity} - 1$, then the term stored at index i has arity $j - 1$, and all elements `argj[i], …, argmaxarity-1[i]` should be ignored.
- Boolean flag `done` indicates whether more rewriting iterations are needed;
- Integer arrays `refcounts`, `refcounts_read` are used to write and read the number of references to each term, respectively. When a term is not referenced, it can be deleted.

Some form of garbage collection is necessary to be able to reuse memory occupied by deleted terms. For this reason, we have the following additional data structures:

- Boolean flag `garbage_collecting` indicates whether garbage collecting is needed;
- Integer array `free_indices` stores indices that can be reused for new terms;
- Integer variables `next_free_begin`, `next_free_end` provide indices to remove elements from the front of `free_indices` and add elements at the end, respectively;
- Integer variable `next_fresh` provides a new index, greater than the largest index currently occupied by a term in the term arrays. There, a new term can be inserted.

Listing 1.3 presents the main loop of the algorithm, which is executed by the CPU. In it, two GPU kernels are repeatedly called until a fix-point has been

Listing 1.4. The derive kernel, executed by a GPU thread

```

1  derive (...) {
2    if (tid >= n) { return; }
3    refcount = refcounts_read[tid];
4    if (refcount > 0) {
5      start_rewriting = !nf_read[tid];
6      if (start_rewriting) {
7        if (all subterms are in normal form) {
8          switch (hss[tid]) {
9            case f: rewritef(...);
10           ...
11          default: nf[tid] = true;
12        }
13      }
14      done = false;
15    }
16  }
17  else {
18    garbage_collecting = true;
19  }
20 }

```

reached, indicated by **done**. To keep track of the progress, there are CPU counterparts of several variables, labeled with the ‘h.’ prefix. Copying data between CPU and GPU memory is represented by \leftarrow .

While the rewriting is not finished (line 2), the GPU **done** flag is set to **false** (line 3), 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**). After that, **refcounts** is copied to **refcounts_read**, and **nf** to **nf_read**. The reading and writing of the reference counters and normal form state is separated by the use of two arrays, to avoid newly created terms already being rewritten before they have been completely stored in memory. The **derive** kernel is then launched for the selected number of blocks (line 7). This kernel, shown in Listing 1.4, is discussed later. In the kernel, the GPU threads perform one rewrite iteration. Then, at lines 8-12, **n** is updated in case the number of terms has increased. The **next_fresh** variable is used to count the number of new terms placed at fresh indices, i.e., indices larger than **n** when **derive** was launched.

Finally, with **garbage_collecting**, it is monitored whether some indices of deleted terms need to be gathered in the **free_indices** list. This gathering is done by the **collect_free_indices** kernel: if a thread detects that the reference counter of its term is 0, it decrements the counters of the subterms and the index to the term is added to the **free_indices** list. Atomic memory accesses are used to synchronise this. Notice that **free_indices** is in device memory and no unnecessary data is transferred back and forth between host and device.

In Listing 1.4, the GPU **derive** kernel is described. When the kernel is launched for **numBlocks**·**blockSize** threads, each of those threads executes the kernel to process its term. 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 they first check 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

Listing 1.5. An example rewrite function for the rule $\text{Plus}(\text{Zero}, X) \rightarrow X$

```

1  rewritePlus(...) {
2    r_0 = arg0[tid];
3    r_hs_0 = hss[r_0];
4    if (r_hs_0 == Zero) {
5      r_1 = arg1[tid];
6      r_hs_1 = hss[r_1];
7      hss[tid] = r_hs_1;
8      copy_term_args(refscount, arg0, arg1, r_1, tid, r_hs_1);
9      atomicSub(&refscounts[r_0], 1);
10     atomicSub(&refscounts[r_1], 1);
11     nf[tid] = true;
12     return;
13   }
14   ...Check applicability of other Plus-rules
15 }
```

for rewriting is required. Rewriting is needed if the term is not in normal form (line 5) and if all its subterms are in normal form (line 7). To avoid repetitive checking of subterms in each execution of the `derive` kernel, every thread keeps track of the last subterm it checked in the previous iteration. If rewriting is required, the suitable `rewritef` function is called, depending on the head symbol of the term (lines 8-12). If no function is applicable, the term is in normal form (line 11). Finally, `done` is set to `false` to indicate that another rewrite iteration is required. Alternatively, if the reference counter is 0, the `garbage_collecting` flag is set. This causes the `collect_free_indices` kernel to be launched after the `derive` kernel (see Listing 1.3).

Given a TRS, the `rewritef` functions are automatically generated by a code generator we developed, to directly encode the rewriting in CUDA code. Listing 1.5 provides example code for the rewrite rule $\text{Plus}(\text{Zero}, X) \rightarrow X$, which expresses that adding 0 to some number X results in X . Applicability of this rule is checked by the `rewritePlus` function, which may also involve other rules for terms with head symbol `Plus`. First, to check applicability, the index of the first subterm is retrieved, and with it, the head symbol of that term (lines 2-3). If the head symbol is `Zero` (line 4), the index to the second subterm is retrieved (line 5). The rewriting procedure should ensure that the term at position `tid` is replaced by X .

When constructing terms, sharing of subterms is applied whenever possible. For instance, if a term $F(X, X)$ needs to be created, the index to X would be used twice in the new term, to make sure both subterm entries point to the same term in physical memory. When rewriting the term itself, however, as in the example, 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 6-7), and then the indices of the subterms, which is done at line 8 by the function `copy_term_args`; it copies the number of subterms relevant for a term with the given head symbol, and increments the reference counters of those subterms. Next, the reference counters of `Zero` and X are atomically decremented (since

Listing 1.6. The `get_new_index` GPU function, executed by a GPU thread

```

1 get_new_index(...) {
2   if (tid >= n) { return; }
3   n_begin = next_free_begin; n_end = next_free_end;
4   new_id = 0;
5   if (n_begin < n_end) {
6     n_begin = atomicInc(&next_free_begin);
7     if (n_begin < n_end) {
8       new_id = free_indices[n_begin];
9     }
10  }
11  if (new_id == 0) {
12    new_id = atomicInc(&next_fresh) + n;
13  }
14  return new_id;
15 }
```

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

the term `Plus(Zero, X)` is removed) (lines 9-10), and we know that the resulting term is in normal form, since `X` is in normal form (line 11).

Finally, we show how new indices are retrieved whenever a new term needs to be created. In the example of Listing 1.5, this is not needed, as the RHS of the rule has no new subterms, but for a rule such as `Plus(S(0), X) → S(X)`, with `S` representing the successor function (i.e., `S(0)` represents 1) a new term `S(X)` needs to be created, with its only subterm entry pointing to the term referenced by the second subterm entry of the LHS.

Listing 1.6 shows how we retrieve a new index. Due to garbage collection, a number of indices may be available in the first `n` entries of the input arrays which are currently used. These are stored in the `free_indices` array, from index `next_free_begin` to index `next_free_end`. If this array is not empty (line 5), `next_free_begin` is atomically incremented to claim the next index in the `free_indices` array (line 6). If this increment was not performed too late (other threads have not since claimed all available indices), the index is stored in `new_id` (lines 7-9). Otherwise, a new index must be added at the end of the current list of terms. The variable `next_fresh` is used for this purpose: `next_fresh + n` can be used as a new index, and `next_fresh` needs to be incremented for use by another thread.

4 Evaluation

In this section we provide insight into the performance of the GPU rewriter. We do this in two ways: We compare our GPU rewriter with a sequential recursive left-most inner-most rewriter for the CPU (1) and (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 [15]. We therefore include the second way of evaluating.

Table 1 shows a comparison of the used CPU and GPU. CPUs are optimized for latency: finish the program as soon as possible. In contrast, GPUs are optimized 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 a TRS, both the GPU and the CPU rewriter are generated by a Python 2.7 script. The script uses TextX [7] and Jinja 2.11⁴ to parse a TRS and generate a `rewritef` function for every rewritable head symbol f in the TRS. 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 `rewritef` functions are used, and thus the rewrite rules are equal and the CPU and GPU implementations rewrite the same number of terms.

We evaluate the GPU rewriter with a TRS for sorting one or more lists of Peano numbers with merge sort (see the example in Section 2) and with a TRS that transforms a large number of terms.⁵ These TRSs accentuate the capabilities of the GPU and the CPU. Merge sort is a divide-and-conquer algorithm amenable to parallelism, but splitting and combining lists is highly sequential.

Figure 1 shows a merge sort performed on a single list of 50 elements. The width of a red box (too small for Fig. 1a, see the zoomed in version in Fig. 1b) represents the time of a GPU rewrite step (the `derive` statement on line 7 in Listing 1.3) whereas the height represents how many terms are rewritten in parallel in this rewrite step. The figure shows that there are long tails of a low degree of parallelism before and after a brief peak of parallelism. Given Amdahl’s law that states that speedup is severely limited with a low degree of parallelism [1], it is clear that merge sort on single lists is not parallel enough for GPUs.

The performance of the GPU of 74×10^3 terms/s versus the CPU 97×10^6 terms/s highlights a different issue, namely Gustafson’s Law [9]: To overcome the overhead of using a highly parallel machine, we need a large problem with a high degree of parallelism to highlight the capabilities of the GPU. In order to benchmark this potential, we use the merge sort TRS applied on multiple lists: The Tree merge sort is given by a binary tree with a list of numbers at every leaf. All these lists are sorted concurrently using the same merge sort as in the previous example. The parallelism is exponential w.r.t. the depth of the tree.

Table 2 shows that the GPU outperforms the CPU more than a factor of three for a binary tree of 23 levels deep of merge sorts of lists of 5 numbers,

⁴ <https://jinja.palletsprojects.com>.

⁵ See [21, Appendix A] for a detailed description of the TRSs.

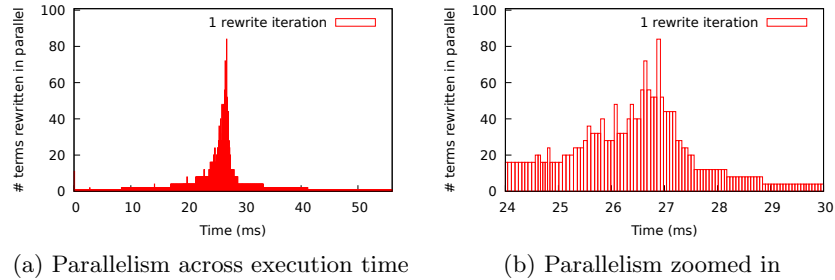


Fig. 1. Merge sort of 50 elements on the GPU.

Table 2. Performance of the rewrite systems.

Application	CPU rewritten terms/s	GPU rewritten terms/s	Speedup
Merge sort 50	97×10^6	74×10^3	0.76×10^{-3}
Tree merge sort 23 5	113×10^6	387×10^6	3.34
Transformation tree 22	265×10^6	3.12×10^9	11.7

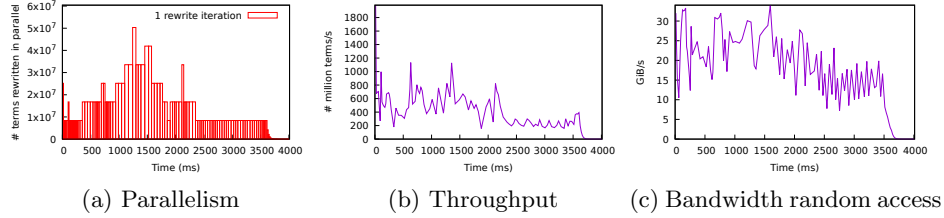
which translates to sorting approximately 8 million lists. Finally, to understand the potential of the GPU, we designed the Transformation tree benchmark that expands a binary tree to 22 levels deep (4 million leaves) where each leaf is rewritten 26 times. On this benchmark, the GPU rewriter is more than a factor 10 faster than the CPU rewriter, achieving 3.12 billion rewrites per second on average over the complete run, but sustaining around 6 billion rewrites per second for half of the execution time (the rest is setting/breaking down the tree).

To understand the performance better we focus on the more realistic Tree merge sort benchmark. Figure 2 shows several graphs for the execution with which we can analyze the performance. Figure 2a shows that this rewrite system shows a high degree of parallelism for almost all of the execution time. Figure 2b shows that the rewriter shows a high throughput with peaks up to 1 billion terms rewritten per second.

Figure 2c highlights 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 access (to subterms that can be anywhere in memory), we focus on the bandwidth for random accesses. Table 3 shows that the overall random access bandwidth of the GPU implementation reaches 18.1 GiB/s which is close to the benchmarked bandwidth. In addition, the aligned bandwidth of 95.7 GiB/s confirms that term rewriting is indeed an irregular problem and that aligned bandwidth is less of a bottleneck.

Table 3. Performance the tree merge sort in terms of bandwidth.

Application	BW random (GiB/s)	BW aligned (GiB/s)
Tree merge sort	23.5	18.1
		95.7

**Fig. 2.** Merge sort of a tree of 23 deep with lists of 5 elements.

Although we are close to the random access bandwidth of the GPU, this does not mean that we have reached the limits of term rewriting on GPUs. It does mean however, that to achieve higher performance with term rewriting on GPUs, it is necessary to introduce more regularity into the implementation, reducing the random memory accesses. It also means that other often used strategies to improve graph algorithm, like reducing branch divergence will probably not yield significant performance increase. 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 empirically switch to a GPU implementation when a high degree of parallelism is available.

Acknowledgments This work is carried out in the context of the NWO AVVA project 612.001751. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the GeForce Titan RTX used for this research.

References

1. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *SJCC*, pages 483–485, New York, NY, USA, 1967. ACM.
2. H.P. Barendregt, M.C.J.D. van Eekelen, M.J. Plasmeijer, P.H. Hartel, L.O. Hertzberger, and W.G. Vree. The Dutch parallel reduction machine project. *Future Generation Comp. Syst.*, 3(4):261–270, 1987.
3. D. Bosnacki, S. Edelkamp, D. Sulewski, and A. Wijs. GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In *PDMC-HiBi*, pages 17–19. IEEE Computer Society, 2010.
4. D. Bošnački, M.R. Odenbrett, A.J. Wijs, W.P.A. Ligtenberg, and P.A.J. Hilbers. Efficient reconstruction of biological networks via transitive reduction on general purpose graphics processors. *BMC Bioinformatics*, 13(281), 2012.

5. O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. J. Wijs, and T. A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In *TACAS*, volume 11428 of *LNCS*, pages 21–39. Springer, 2019.
6. J. Cheng and M. Grossman. *Professional CUDA C Programming*. John Wiley and Sons Ltd., 2014.
7. I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković. TextX: A Python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems*, 115:1–4, 2017.
8. F. Durán and H. Gavel. The rewrite engines competitions: A RECTrospective. In *TACAS*, volume 11429 of *LNCS*, pages 93–100. Springer, 2019.
9. J. L. Gustafson. Reevaluating Amdahl’s law. *Commun. ACM*, 31(5):532–533, 1988.
10. S. Heldens, P. Hijma, B. Van Werkhoven, J. Maassen, A. S. Z. Belloum, and R. V. Van Nieuwpoort. The landscape of exascale research: A data-driven literature analysis. *ACM Comput. Surv.*, 53(2), March 2020.
11. T. Henriksen, M. Elsmann, and C. E. Oancea. Modular acceleration: tricky cases of functional high-performance computing. In *FHPC*, pages 10–21. ACM, 2018.
12. T. Henriksen, N. G. W. Serup, M. Elsmann, F. Henglein, and C. E. Oancea. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *PLDI*, pages 556–571. ACM, 2017.
13. P. Hijma, R. V. Nieuwpoort, C. J. H. Jacobs, and H. E. Bal. Stepwise-refinement for performance: A methodology for many-core programming. *Concurr. Comput.: Pract. Exper.*, 27(17):4515–4554, December 2015.
14. G. Huet and D. Lankford. On the Uniform Halting Problem for Term Rewriting Systems. *Rapport de Recherche No 283, 1978, IRIA*, March 1978.
15. V. W. Lee, P. Hammarlund, R. Singhal, P. Dubey, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, and S. Chennupaty. Debunking the 100X GPU vs. CPU myth. In *ISCA*. ACM Press, 2010.
16. T. L. McDonell. *Optimising Purely Functional GPU Programs*. PhD thesis, University of New South Wales, Sydney, Australia, 2015.
17. R. Nasre, M. Burtscher, and K. Pingali. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *IPDPS, May 20-24*, pages 463–474. IEEE Computer Society, 2013.
18. R. Nasre, M. Burtscher, and K. Pingali. Morph algorithms on GPUs. In *PPOPP*, pages 147–156. ACM Sigplan, 2013.
19. S. L. Peyton Jones, C. D. Clack, J. Salkild, and M. Hardie. GRIP - A high-performance architecture for parallel graph reduction. In *FPCA*, volume 274 of *LNCS*, pages 98–112. Springer, 1987.
20. Terese. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.
21. Johri van Eerd, Jan Friso Groote, Pieter Hijma, Jan Martens, and Anton Wijs. Term rewriting on GPUs, 2020. [arXiv:2009.07174](https://arxiv.org/abs/2009.07174).
22. A.J. Wijs, T. Neele, and D. Bošnački. GPUexplore 2.0: Unleashing GPU explicit-state model checking. In *FM*, volume 9995 of *LNCS*, pages 694–701, 2016.