A GPU Tree Database for Many-Core Explicit State Space Exploration

Anton Wijs(✉) and Muhammad Osama

Eindhoven University of Technology, Eindhoven, The Netherlands
{a.j.wijs,o.m.m.muhammad}@tue.nl

Abstract. Various techniques have been proposed to accelerate explicit-state model checking with GPUs, but none address the compact storage of states, or if they do, at the cost of losing completeness of the checking procedure. We investigate how to implement a tree database to store states as binary trees in GPU memory. We present fine-grained parallel algorithms to find and store trees, experiment with a number of GPU-specific configurations, and propose a novel hashing technique, called Cleary-Cuckoo hashing, which enables the use of Cleary compression on GPUs. We are the first to assess the effectiveness of using a tree database, and Cleary compression, on GPUs. Experiments show processing speeds of up to 131 million states per second.

Keywords: Explicit state space exploration, finite-state machines, GPU.

1 Introduction

Major advances in computation increasingly need to be obtained via parallel software, as Moore's Law is ending [30]. In the last decade, GPUs have been successfully applied to accelerate various computations relevant for model checking, such as probability computations for probabilistic model checking [8,25,48], counterexample construction [54], state space decomposition [52], parameter synthesis for stochastic systems [12], and SAT solving [34–38,40,43,56,57]. VoxLogicA-GPU applies model checking to analyse (medical) images [9].

In the earliest work on GPU explicit state space exploration, GPUs performed part of the computation, specifically successor generation [18,19] and property checking once the state space has been generated [5]. This was promising, but the data copying between main and GPU memory and the computations on the CPU were detrimental for performance. The first tool that performed the entire exploration on a GPU was GPUexplore [33,50,51,53]. It was later extended to support LTL model checking [49]. A similar exploration engine was later proposed in [55]. An approach that applied a GPU to explore the state space of Promela models, i.e., the models for the Spin model checker [21], was presented in [6]. This was later adapted to the swarm checker Grapple [16], which can efficiently explore very large state spaces, but at the cost of losing completeness. Finally, the model checker ParaMoC for pushdown systems was presented in [46,47].

The above techniques demonstrate the potential for GPU acceleration of state space exploration and (explicit-state) model checking, being able to accelerate those procedures *tens* to *hundreds* of times, but they all have serious practical limitations. Several limit the size of state vectors to 64 bits [6, 55] or the size of transition encodings to 64 bits [46, 47]. GPUExplore does not efficiently support models with variables [50, 53]. When adding variables, the amount of memory needed rapidly grows, due to the growing input model and inefficient state storage. Grapple requires less memory, but uses bitstate hashing. This rules out the ability to detect that all reachable states have been explored, which is crucial to prove the absence of undesired behaviour. ParaMoC verifies push-down systems, but does not support concurrency, and abstracts away data.

Contributions. We propose how to perform memory-efficient complete state space exploration on a GPU for concurrent *Finite-State Machines (FSMs) with data*. To make this possible, we are the first to investigate the storage of binary trees in GPU hash tables, propose new algorithms to find and store trees in a fine-grained parallel fashion, experiment with a number of GPU-specific configurations, and propose a novel hashing technique called *Cleary-Cuckoo hashing*, which enables the use of *Cleary compression* [13,15] on GPUs. To achieve this, we have to tackle the following challenges: 1) CPU-based algorithms are recursive, but GPUs are not suitable for recursion, and 2) accessing GPU global memory, in which the hash tables reside, is slow. This work marks an important step to pioneer practical GPU accelerated model checking, as it can be extended to checking functional properties of models with data, and paves the way to investigate the use of *Binary Decision Diagrams* [29] for symbolic model checking.

The structure of the paper is as follows. In Section 2, we discuss related work on GPU hash tables. Section 3 presents background information on GPU programming, and Section 4 contains an overview of the state space exploration engine. Section 5 addresses the challenges when designing a GPU tree table, and presents our new algorithms. Experimental results are given in Section 6, and in Section 7, conclusions and our future work plans are discussed.

2 Related Work

An overview of related work on GPU acceleration of model checking is given in Section 1. In the current section, we focus on *hash tables* [14] for the GPU. In explicit state space exploration, states are typically stored in a hash table. Such a table is often implemented as an array, where the elements represent the hash table *buckets*. A recent survey of GPU hash tables [31] identifies that when using integer data items and unordered insertions and queries, *Cuckoo hashing* [41] is (currently) the best option, compared to techniques such as *chaining* [3] or *robin hood hashing* [20], and the Cuckoo hashing of [1] is particularly effective. In Cuckoo hashing, collisions, i.e., situations where a data item e is hashed to an already occupied bucket, are resolved by evicting the encountered item e', storing e, and moving e' to another bucket. A fixed number of m hash functions is used to have multiple storage options for each item. Item look-up and storage is therefore limited to m memory accesses, but can lead to chains of evictions. In [1], it is demonstrated that with four hash functions, a hash table needs around $1.25N$ buckets to store N items.[1] Recent research [4] has demonstrated that using larger buckets, spanning multiple elements, that still fit in the GPU cache line is beneficial for performance, and increases the average load factor, i.e., how much the hash table can be filled until an item cannot be inserted, to 99%. We address this in detail in Section 3. However, in [4], an older NVIDIA GPU of the Volta architecture was used (2017), while more recent GPUs are supposedly less susceptible to optimisations exploiting the cache line. In this work, we experimentally assess this for hash table buckets.

Besides buckets, we also consider Cuckoo hashing as used in [1, 4], but we are the first to investigate the storage of *binary trees*, and the use of Cleary compression to store more data in less space. Libraries offering GPU hash tables, such as [23], do not offer these capabilities. Furthermore, we are the first to investigate the impact of using larger buckets for binary tree storage embedded in a state space exploration engine.

The model checker GPUexplore [11, 50, 53] uses multiple hash functions to store a state. State evictions are never performed, as each state is stored in a sequence of integers, making it not possible to store states atomically. This can lead to storing duplicate states, which tends to be worsened when states are evicted, making Cuckoo hashing not practical [51]. Besides compact state storage, a second benefit of using trees with each node being stored in a single integer is that it allows arbitrarily large states to be stored atomically, i.e., a state is stored the moment the root of its tree is stored.

Because we store trees, with the individual nodes referencing each other, we do not consider alternative storage approaches, such as using a list that is repeatedly sorted, even though Alcantara *et al.* identified that using *radix-sort* [32] is competitive to hashing [1].

3 GPU programming

CUDA[2] is a programming interface that enables general purpose programming for a GPU. It has been developed and continues to be maintained by NVIDIA since 2007. In this work, we use CUDA with C++. Therefore, we use CUDA terminology when we refer to thread and memory hierarchies.

The left part of Fig. 1 gives an overview of a GPU architecture. For now, ignore the bold-faced words and the pseudo-code. A GPU consists of a finite number of *streaming multiprocessors* (SM), each containing hundreds of *cores*. For instance, a Titan RTX, which we used for this work, has 72 SMs containing together 4,608 cores. A programmer can implement functions, named *kernels*, to

[1] This refers to the *single-level* version of their Cuckoo hashing [1], which we consider in this work. Their two-level version is more complex and less efficient.

[2] https://developer.nvidia.com/cuda-zone.

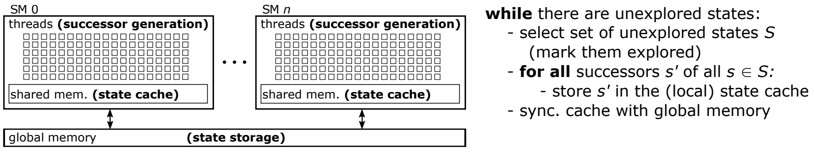
SM 0
threads **(successor generation)**
shared mem. **(state cache)**
...
SM n
threads **(successor generation)**
shared mem. **(state cache)**
global memory **(state storage)**
while there are unexplored states:
- select set of unexplored states S (mark them explored)
- **for all** successors s' of all $s \in S$:
- store s' in the (local) state cache
- sync. cache with global memory

Fig. 1: State space exploration on a GPU architecture.

be executed by a predefined number of GPU threads. Parallelism is achieved by having these threads work on different parts of the data.

When a kernel is launched, threads are grouped into *blocks*, usually of a size equal to a power of two, often 512 or 1,024. Each block is executed by one SM, but an SM can interleave the execution of many blocks. When a block is executed, the threads inside are scheduled for execution in smaller groups of 32 threads called *warps*. A warp has a single program counter, i.e., the threads in a warp run in lock-step through the program. This concept is referred to as *Single Instruction Multiple Threads* (SIMT): each thread executes the same instructions, but on different data. The threads in a warp may also follow *diverging* program paths, leading to a reduction in performance. For instance, if the threads of a warp encounter an `if C then P1 else P2` construct, and for some, but not all, `C` holds, all threads will step through the instructions of both `P1` and `P2`, but each thread only executes the relevant instructions.

GPU threads can use atomic instructions to manipulate data atomically, such as a *compare-and-swap* on 32- and 64-bit integers: ATOMICCAS(`addr`, `compare`, `val`) atomically checks whether at address `addr`, the value `compare` is stored. If so, it is updated to `val`, otherwise no update is done. The actual value read at `addr` is returned.

There are various types of memory on a GPU. The *global memory* is the largest of these, 24 GB in the case of the Titan RTX, and is used to copy data between the *host* (CPU-side) and the *device* (GPU-side). It can be accessed by all GPU threads, and has a high bandwidth, but also a high latency. Having many threads executing a kernel helps to hide this latency; the cores can rapidly switch contexts to interleave the execution of multiple threads, and whenever a thread is waiting for the result of a memory access, the core uses that time to execute another thread. Another way to improve memory access times is by ensuring that the accesses of a warp are *coalesced*: if the threads in a warp try to fetch a consecutive block of memory in size not larger than the cache line (128 bytes for a Titan RTX), then the time needed to access that block is the same as the time needed to access an individual memory address.

Other types of memory are *shared memory* and *registers*. Shared memory is fast on-chip memory with a low latency, that can be used as block-local memory; the threads of a block can share data with each other via this memory. In a Titan RTX, each block can use up to 49,152 bytes of shared memory. Register memory is the fastest, and is used to store thread-local data. It is very small, though, and allocating too much memory for thread-local variables may result in data spilling over into global memory, which can dramatically limit the performance.

Finally, the threads in a warp can communicate very rapidly with each other by means of *intra-warp instructions*. There are various instructions, such as SHUFFLE to distribute register data among the threads and BALLOT to distribute the results of evaluating a predicate. Since CUDA 9.0, threads can be partitioned into *cooperative groups*. If these groups have a size that completely divides the warp size, i.e., it is a power of two smaller than or equal to 32, then the threads in a group can use intra-warp instructions among themselves.

In Section 2, we mentioned the use of buckets in a GPU hash table. When a hash table is divided into buckets, each containing $1 < n \leq 32$ elements, that still fit in the cache line, then cooperative groups of n threads each can be created, and the threads in a group can work together for the fetching and updating of buckets. This results in more coalesced memory accesses and reduces thread divergence. However, it also means that fewer tasks can be performed in parallel, and starting with the TURING architecture (2018), which the Titan RTX is built on, NVIDIA has been working on making computations less reliant on coalesced memory accessing.

4 GPU state space exploration

SLCO. For this work, we extended the state space exploration engine of GPU-EXPLORE 2.0 [53] to support models of finite-state concurrent systems written in the *Simple Language of Communicating Objects* (SLCO), version 2.0 [44]. An SLCO model consists of a finite number of FSMs. The FSMs can communicate via globally shared variables, and each FSM can have its own local variables. Variables can be of type `Bool`, `Byte` and (32-bit) `Integer`, and there is support for arrays of these types. We refer with *(system) states* $s, s', \ldots$ to entire states of the system, and with *FSM states* $\sigma, \sigma', \ldots$ to the states of an individual FSM. A system state is essentially a vector, containing all the information that together defines a state of the system, i.e., the current states of the FMSs and the values of the variables.

An FSM transition $tr = \sigma \xrightarrow{st} \sigma'$ indicates that the FSM can change state from σ to σ' iff the associated *statement* st is *enabled*. A statement is either an *assignment*, an *expression* or a *composite*. Each can refer to the variables in the scope of the FSM. An assignment is always enabled, and assigns a value to a variable, an expression is a predicate that acts as a guard: it is enabled iff it evaluates to **true**. Finally, a composite is a finite sequence of statements $st_0; \ldots; st_n$, with st_0 being either an expression or an assignment, and $st_1, \ldots, st_n$ being assignments. A composite is enabled iff its first statement is enabled. A transition $tr = \sigma \xrightarrow{st} \sigma'$ can be *fired* if it is enabled, which results in the FSM atomically moving from state σ to state σ', and any assignments of st being executed in the specified order. When tr is fired while the system is in a state s, then after firing, the system is in state s', which is equal to s, apart from the fact that σ has been replaced by σ', and the effect of st has been taken into account. We call s' a *successor* of s.

The formal semantics of Slco defines that each transition is executed atomically, i.e., cannot be interrupted by the execution of other transitions. The FSMs execute concurrently, using an interleaving semantics. Finally, the FSMs may have non-deterministic behaviour, i.e., at any point of execution, an FSM may have several enabled transitions.

State space exploration. Given an Slco model with n FSMs, first, CUDA functions $f_1, \dots f_n$ are generated, using a new code generator, that take as input a state s, and produce as output the successors of s which can be reached by firing a transition enabled in s of the i^{th} FSM. When the state space is generated, each state s can be analysed in parallel by n threads $t_1, \dots, t_n$, where each t_i executes f_i to obtain some of the successors of s.

Fig. 1 presents how the different components of the state space exploration engine map on a GPU. We explain how the engine works insofar is needed. For more details, we refer the reader to [50, 51, 53]. Even though the type of input model has changed, as GPUexplore only supports models without data variables, the core of the engine has remained the same.

In the global memory, a large hash table (we call it $\mathcal{G}$) is maintained to store the states visited so far. At the start, the initial state of the input model is stored in $\mathcal{G}$. Each state in $\mathcal{G}$ has a Boolean flag *new*, indicating whether the state has already been explored, i.e., whether or not its successors have been constructed.

On the right in Fig. 1, the state space exploration algorithm is explained from the perspective of a *thread block*. While the block can find unexplored states in $\mathcal{G}$, it selects some of those for exploration. In fact, every block has a *work tile* residing in its shared memory, of a fixed size, which the block tries to fill with unexplored states at the start of each exploration iteration. Such an iteration is initiated on the host side by launching the exploration kernel. States are marked as explored when added by threads to their tile.

Next, every block processes its tile. For this, each thread in the block is assigned to a particular state/FSM combination. Each thread accesses its designated state in the tile, and analyses the possibilities for its designated FSM to change state, as explained before. Hence, the threads in a group can generate successors for a single state in parallel.

The generated successors are stored in a *block-local state cache*, which is a hash table in the shared memory. This avoids repeated accessing of global memory, and local duplicate detection filters out any duplicate successors generated at the block-level. Once the tile has been processed, the threads in the block together scan the cache once more, and store the new states in $\mathcal{G}$ if they are not already present. When states require no more than 32 or 64 bits in total (including the *new* flag), they can simply be stored atomically in $\mathcal{G}$ using compare-and-swap. However, sufficiently large systems have states consisting of more than 64 bits. In this paper, we therefore focus on working with these larger states, and consider storing them as binary trees.

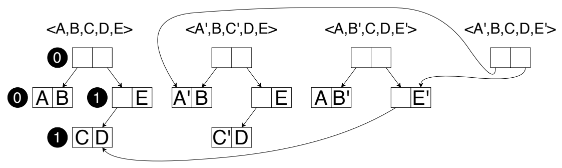

Fig. 2: An example of storing state vectors as binary trees.

5 A Compact GPU Tree Database

5.1 CPU Tree Storage

The number of data variables in a model, and their types, can have a drastic effect on the size of the states of that model. For instance, each 32-bit integer variable in a model requires 32 bits in each state. As the amount of global memory on a GPU is limited, we need to consider techniques to store states in a memory-efficient way. One technique that has proven itself for CPU-based model checkers is *tree compression* [7], in which system states are stored as binary trees. A single hash table can be used to store all tree nodes [27]. Compression is achieved by having the trees share common subtrees. Its success relies on the observation that states and their successors tend to be different in only a few data elements. In [27], it is experimentally assessed that tree compression compresses better than any other compression technique identified by the authors for explicit state space exploration. They observe that the technique works well for a multi-threaded exploration engine. Moreover, they propose an *incremental* variant that has a considerably improved runtime performance, as it reduces the number of required memory accesses to a number logarithmic in the length of the state vector.

Fig. 2 shows an example of applying tree compression to store four state vectors. The black circles should be ignored for now. Each letter represents a part of the state vector that is k bits in length. We assume that in k bits, also a pointer to a node can be stored, and that each node therefore consists of $2k$ bits. The vector <A,B,C,D,E> is stored by having a root node with a left leaf sibling <A,B>, and the right sibling being a non-leaf that has both a left leaf sibling <C,D>, and the element E. In total, storing this tree requires $8k$ bits. To store the vector <A',B,C',D,E>, we cannot reuse any of these nodes, as <A',B> and <C',D> have not been stored yet. This means that all pointers have to be updated as well, and therefore, a new root and a new non-leaf containing E are needed. Again, $8k$ bits are needed. For <A,B',C,D,E'>, we have to store a new node <A,B'> and a new root, and a new non-leaf storing E', but the latter can point to the already existing node <C,D>. Hence, only $6k$ bits are needed to store this vector. Finally, for <A',B,C,D,E'>, we only need to store a new root node, as all other nodes already exist, resulting in only needing $2k$ bits. It has been demonstrated that as more and more state vectors are stored, eventually new vectors tend to require $2k$ bits each [26, 27].

To emphasise that GPU tree compression has to be implemented vastly differently from the typical CPU approach, we first explain the latter, and the incremental approach [27]. Checking for the presence of a tree and storing it if

Algorithm 1: Tree-based Find-or-put, CPU version.

```
function FINDORPUT-CPU(node_t* $\mathcal{G}$, node_t node):
    if HAS-LEFT-SIBLING(node) and IS-UPDATED(LEFT-SIBLING(node)) then
        node.left ← FINDORPUT-CPU($\mathcal{G}$, LEFT-SIBLING(node))
    if HAS-RIGHT-SIBLING(node) and IS-UPDATED(RIGHT-SIBLING(node)) then
        node.right ← FINDORPUT-CPU($\mathcal{G}$, RIGHT-SIBLING(node))
    addr ← STORE($\mathcal{G}$, node)
    return addr
```

not yet present is typically done by means of recursion (outlined by Alg. 1). For now, ignore the red underlined text. The STORE function returns the address of the given `node` in $\mathcal{G}$, if present, otherwise it stores the node and returns its address, and the FINDORPUT-CPU function first recursively checks whether the siblings of the node are stored, and if not, stores them, after which the node itself is stored. A `node` has pointers `left` and `right` to addresses of $\mathcal{G}$, and there are functions to check for the existence of, and retrieve the siblings of a node.

In the incremental approach, when creating a successor s' of a state s, the tree for s, say $T(s)$, is used as the basis for the tree $T(s')$. When $T(s')$ is created, each node inside it is first initialised to the corresponding node in $T(s)$, and the leaves are updated for the new tree. This 'updated' status propagates up: when a non-leaf has an updated sibling, its corresponding $\mathcal{G}$ pointer must be updated when $T(s')$ is stored in $\mathcal{G}$, but for any non-updated sibling, the non-leaf can keep its $\mathcal{G}$ pointer. When incorporating the red underlined text in Alg. 1, the incremental version of the function is obtained. With this version, tree storage often results in fewer calls to STORE, i.e., fewer memory accesses.

There are two main challenges when considering GPU incremental tree storage: 1) Recursion is detrimental to performance, as call stacks are stored in global memory (and with thousands of threads, a lot of memory would be needed for call stacks), and 2) The nodes of a tree tend to be spread all over the hash table, potentially leading to many random accesses. To address these, we propose a procedure in which threads in a block store sets of trees together in parallel.

5.2 GPU Tree Generation

When states are represented by trees, the tile of each thread block cannot store entire states, but it can store the roots of trees. To speed up successor generation, and avoid repeated uncoalesced global memory accessing, the trees of those roots are retrieved and stored in the shared memory (state cache) by the thread block. Once this has been done, successor generation can commence.

Fig. 3 shows an example of the state cache evolving over time as a thread generates the successor s' =<A,B',C,D,E'> of s =<A,B,C,D,E>, with the trees as in Fig. 2. Each square represents a k-bit cache entry. In addition to two entries needed to store a node, we also use one (grey) entry to store two *cache pointers* or indices, and assume that k bits suffice to store two pointers (in practice, we use $k = 32$, which is enough, given the small size of the state cache). Hence, every pair of white squares followed by a grey square constitutes one cache slot. Initially (shown at the top of the figure), the tile has a cache pointer to the root of s, of

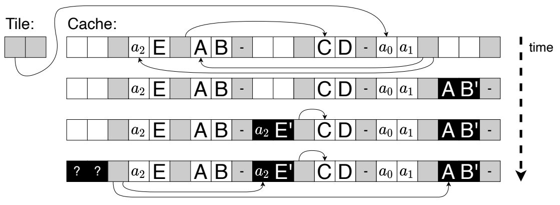

Fig. 3: Successor generation: deriving <A,B',C,D,E'> from <A,B,C,D,E>.

which we know that it contains the $\mathcal{G}$ addresses a_0 and a_1 to refer to its siblings. In turn, this root points, via its cache pointers, to the locally stored copies of its siblings. The non-leaf one contains the global address a_2. A leaf has no cache pointers, denoted by '-'. When creating s', first, the designated thread constructs the leaf <A,B'>, by executing the appropriate generated CUDA function (see Section 4), and stores it in the cache. In Fig. 3, it is coloured black, to indicate that it is marked as new. Next, the thread creates a copy of <a_2,E>, together with its cache pointers, and updates it to <a_2,E'>. Finally it creates a new root, with cache pointers pointing to the newly inserted nodes. This root still has global address gaps to be filled in (the '?' marks), since it is still unknown where the new nodes will be stored in $\mathcal{G}$.

The reason that we store global addresses in the cache is not to access the nodes they point to, but to achieve incremental tree storage: in the example, as the global address a_2 is stored in the cache, there is no need to find <C,D> in $\mathcal{G}$ when the new tree is stored; instead, we can directly construct <a_2,E'>. This contributes to limiting the number of required global memory accesses.

Note that there is no recursion. Given a model, the code generator determines the structure of all state trees, and based on this, code to fetch all the nodes of a tree and to construct new trees is generated. As we do not consider the dynamic creation and destruction of FSMs, all states have the same tree structure.

5.3 GPU Tree Storage at Block Level

Once a block has finished generating the successors of the states referred to by its tile, the state cache content must be synchronised with $\mathcal{G}$. Alg. 2 presents how this is done. The FINDORPUT-MANY function is executed by all threads in the block simultaneously. It consists of an outer `while`-loop (l.5-28), that is executed as long as there is work to be done. The code uses a cooperative group called `bg`, which is created to coincide with the size of a bucket (`bucketsize`). When no buckets are used, these groups can be interpreted as consisting of only a single thread each. At l.4, the `offset` of each thread is determined, i.e., its ID inside its group, ranging from 0 to the size of the group.

Every thread that still has *work to do* (l.5) enters the `for`-loop of l.7-27, in which the content of the state cache is scanned. The parallel scanning works as follows: every thread first considers the node at position *tid* − `offset` of the cache, with *tid* being the thread's block-local ID. This node is assigned to the thread with `bg` ID 0. If that index is still within the cache limits, all threads of

Algorithm 2: Tree-based Find-or-put-many, at thread block level.

```
1  device function FINDORPUT-MANY(node_t* $\mathcal{G}$):
2      node_t p, q; index_t addr; bool work_to_do $\leftarrow$ true; bool ready; byte ballot_result
3      auto bg $\leftarrow$ TILED-PARTITION⟨bucketsize⟩(THIS-THREAD-BLOCK())
4      byte offset $\leftarrow$ bg.THREAD-RANK()
5      while work_to_do do
6          work_to_do $\leftarrow$ false
7          for $i \leftarrow tid -$ offset; $i <$ CACHE_SIZE; $i \leftarrow i +$ BLOCK_SIZE do
8              ready $\leftarrow$ false
9              if $i +$ offset $<$ CACHE_SIZE then
10                 p $\leftarrow$ cache[$i +$ offset]
11                 if IS-NEW-LEAF(p) then  ready $\leftarrow$ true
12                 else if IS-NEW-NONLEAF(p) then
13                     if LEFT-GAP(p) then
14                         cache[$i +$ offset] $\leftarrow$ SET-LEFT-GADDR(p, cache[LEFT-CADDR(p)])
15                     if RIGHT-GAP(p) then
16                         cache[$i +$ offset] $\leftarrow$ SET-RIGHT-GADDR(p, cache[RIGHT-CADDR(p)])
17                     if $\neg$(LEFT-OR-RIGHT-GAP(p)) then  ready $\leftarrow$ true
18                     else  work_to_do $\leftarrow$ true
19             ballot_result $\leftarrow$ bg.BALLOT(ready)
20             while ballot_result do
21                 lane $\leftarrow$ FIND-FIRST-SET(ballot_result) - 1; q $\leftarrow$ bg.SHUFFLE(p, lane)
22                 addr $\leftarrow$ FINDORPUT-SINGLE(bg, $\mathcal{G}$, q)
23                 if offset $=$ lane then
24                     ready $\leftarrow$ false
25                     if addr $=$ FULL then  signal hash table full
26                     else SET-GADDR(cache[$i$], addr)
27                 ballot_result $\leftarrow$ bg.BALLOT(ready)
28         work_to_do $\leftarrow$ bg.BALLOT(work_to_do)
```

`bg` have to move along, regardless of whether they have a node to check or not. At the next iteration of the `for`-loop, the thread jumps over `BLOCK_SIZE` nodes as long as the index is within the cache limits.

The main goal of this loop is to check which nodes are ready for synchronisation with $\mathcal{G}$. Initially, this is the case for all nodes without global address gaps (see Subsection 5.2). Each thread first checks whether its own index is still within the cache limits (l.9). If so, the node `p` is retrieved from the cache at l.10. If it is a new leaf, `ready` is set to **true**, to indicate that the active thread is ready for storage (l.11). If the node is a new non-leaf (l.12), it is checked whether the node still has global address gaps. If it has a gap for the left sibling (l.13), this left sibling is inspected via the cache pointer to this sibling (retrieved with the function LEFT-CADDR (l.14)). The function SET-LEFT-GADDR checks whether the cache pointers of that sibling have been replaced by a global memory address, and if so, uses that address to fill the gap. The same is done for the right sibling at l.15-16. If, after these operations, the node `p` contains no gaps (l.17), `ready` is set to **true**. If the node still contains a gap, another loop iteration is required, hence `work_to_do` is set to **true** (l.18).

At l.19, the threads in the group perform a ballot, resulting in a bit sequence indicating for which threads `ready` is **true**. As long as this is the case for at least one thread, the `while`-loop at l.20-27 is executed. The function FIND-FIRST-SET identifies the least significant bit set to 1 in `ballot_result` (l.21), and the SHUFFLE instruction results in all threads in `bg` retrieving the node of the corresponding `bg` thread. This node is subsequently stored by `bg`, by calling

FINDORPUT-SINGLE (l.22) (explained later). Finally, the thread owning the node (l.23) resets its `ready` flag (l.24), and if the hash table is considered full, reports this globally (l.25). Otherwise, it records the global address of the stored node (l.26). After that, `ballot_result` is updated (l.27). Finally, once the `for`-loop is exited, the `bg` threads determine whether they still have more work to do (l.28).

5.4 Single Node Storage at Bucket Group Level

In this section, we address how individual nodes are stored by a cooperative group `bg`. Before we explain the algorithm for this, Alg. 3, in detail, we consider our options for hashing, and propose a novel combination of existing techniques.

In Section 2, we argued that Cuckoo hashing is very effective on a GPU. However, as it frequently moves elements, it is not suitable for a single hash table, since the non-leaves of a tree refer to the positions of other nodes. We address this by maintaining two hash tables, one for tree roots, and one for the other nodes, as done in [26]. The roots are then not referred to, and hence Cuckoo hashing can be applied on the root table.

In fact, when using two hash tables, we can be even more memory-efficient. In [26], it was shown that *Cleary tables* [13, 15] can be very effective to store state spaces. To handle collisions in Cleary tables, *order-preserving bidirectional linear probing* [2] is used, which involves moving nodes to preserve their order. This makes Cleary tables, like Cuckoo hashing, not suitable to store entire trees, but they can be used to store the *roots* of the trees. In a Cleary table for roots of size $2k$, each root r is hashed (bit scrambled) with a hash function h to a $2k$ bit sequence, from which $w < k$ bits are taken to be used as the *address* to store r in a table with exactly 2^w buckets, and at this position, the remaining $2k - w$ bits (the *remainder*) are actually stored. To enable decompression, h must be invertible; given a remainder and an address, h^{-1} can be applied to obtain r.

In a multi-threaded CPU context, this approach scales well [26], but the parallel approach of [26, 45] divides a Cleary table into *regions*, and sometimes, a region must be locked by a thread to safely reorder nodes. Unfortunately, the use of any form of locking, also fine-grained locking implemented with atomic operations, is detrimental for GPU performance. Further, the absence of coherent caches in GPUs means that expensive global memory accesses may be needed when a thread repeatedly checks the status of an acquired lock.

As an elegant alternative, we propose *Cleary-Cuckoo hashing*, which combines Cleary compression with Cuckoo hashing. We use m hash functions that are invertible (as with Cuckoo hashing) and capable of scrambling the bits of a root to a $2k$ bit sequence (as in Cleary tables). When we apply a function h_i $(0 \leq i < m)$ on a root r, we get a $2k$ bit sequence, of which we use w bits for an address d, and store at d the remainder r' consisting of $2k - w + \lceil \log_2(m) \rceil + 1$ bits. The $\lceil \log_2(m) \rceil$ bits are needed to store the ID of the used hash function (i), and the final bit is needed to indicate that the root is *new* (unexplored). It is possible to retrieve r by applying h_i^{-1} on d and r' without the hash function ID and the *new* bit. When a collision occurs, the encountered root is evicted,

Algorithm 3: Single node find-or-put, at bucket group level.

```
1  device function index_t FINDORPUT-SINGLE(tile_t bg, node_t* $\mathcal{G}$, node_t p):
2      node_t q; index_t addr
3      (q, addr) ← FOP-CUCKOO-ROOT(bg, $\mathcal{G}$, p)
4      for $i \leftarrow 0$; q $\neq$ p and $i <$ MAX_EVICT; $i \leftarrow i + 1$ do
5          (q, addr) ← FOP-CUCKOO-ROOT(bg, $\mathcal{G}$, q)
6      return ($i$ = MAX_EVICT? FULL; addr)

7  device function (node_t, index_t) FOP-CUCKOO-ROOT(tile_t bg, node_t* $\mathcal{G}$, node_t p):
8      comprnode_t cp, cq; node_t q
9      hs ← GET-HASH-START(p); byte offset ← bg.THREAD-RANK()
10     for $i \leftarrow 0$; $i <$ NUM_HASH_FUNCTIONS; $i \leftarrow i + 1$ do
11         (addr, cp) ← ADDR-COMPR-ROOT(p, $h_{(\mathtt{hs}+i) \bmod \mathtt{NUM\_HASH\_FUNCTIONS}}$)
12         (cq, pos) ← HT-FIND(bg, offset, $\mathcal{G}$, addr, cp)
13         if cq = cp then  return (p, addr + pos)
14         if cq = EMPTY then
15             hs ← $h_{(\mathtt{hs}+i) \bmod \mathtt{NUM\_HASH\_FUNCTIONS}}$
16             break
17     if $i$ = NUM_HASH_FUNCTIONS then  (cp, addr) ← ADDR-COMPR-ROOT(p, hs)
18     (cq, pos) = HT-INSERT-CUCKOO(bg, offset, $\mathcal{G}$, addr, cp)
19     if cq $\neq$ EMPTY and cq $\neq$ cp then
20         q ← GET-DECOMPR-ROOT(cq, addr)
21         return (q, addr + pos)
22     return (p, addr + pos)
```

decompressed, and stored again using the hash function next in line for that root. We refer to the application of Cleary compression to roots as *root compression*.

Alg. 3 presents one version of the FINDORPUT-SINGLE function, to which a call in Alg. 2 is redirected when a root is provided. Here, $\mathcal{G}$ is a Cleary-Cuckoo table that is only used to store roots. In FINDORPUT-SINGLE, a second function FOP-CUCKOO-ROOT (l.7-22) is called repeatedly, as long as nodes are evicted or until the pre-configured `MAX_EVICT` has been reached, which prevents infinite eviction sequences (l.4). The function FOP-CUCKOO-ROOT returns the address where the given node was found or stored, and a node, which is either the node that had to be inserted or the one that was already present.

In the FOP-CUCKOO-ROOT function, lines highlighted in purple are specific for root compression, i.e., Cleary compression of roots, while the green highlighted lines concern Cuckoo hashing, addressing node eviction. The ID of the first hash function to be used for node `p`, encoded in `p` itself, is stored in `hs` (l.9), and each thread determines its `bg` offset. Next, the thread iterates over the hash functions, starting with function `hs` (l.10-16). The $\mathcal{G}$ address and node remainder are computed at l.11. If the node is new, the remainder is marked as new. If root compression is not used, we have `p` = `cp`. Then, the function HT-FIND is called to check for the presence of the remainder in the bucket starting at `addr` (l.12). If HT-FIND returns the remainder, then it was already present (l.13), and this can be returned. Note that the returned address is (`addr` + `pos`), i.e., the offset at which the remainder can be found inside the bucket is added to `addr`. Alternatively, if `EMPTY` is returned, the node is not present and the bucket is not yet full. In this case, a bucket has been found where the node can be stored. The used hash function is stored in `hs` (l.15) and the `for`-loop is exited (l.16).

At l.17, if a suitable bucket for insertion has not been found, the initial `hs` is selected again. At l.18, the function HT-INSERT-CUCKOO is called to insert `cp`.

Algorithm 4: Single node insertion, at bucket group level.

```
1  device function (comprnode_t, index_t) HT-INSERT-CUCKOO(tile_t bg, byte offset, node_t*
     $\mathcal{G}$, index_t addr, comprnode_t cp):
2      comprnode_t cq ← $\mathcal{G}$[addr + offset]; byte ballot_result ← bg.BALLOT(cq = cp)
3      if ballot_result then return (cp, FIND-FIRST-SET(ballot_result) - 1)
4      while ballot_result ← bg.BALLOT(cq = EMPTY) do
5          if offset = FIND-FIRST-SET(ballot_result) - 1 then
6              cq ← ATOMICCAS($\mathcal{G}$[addr + offset], EMPTY, cp)
7          cq ← bg.SHUFFLE(cq, FIND-FIRST-SET(ballot_result) - 1)
8          if cq = EMPTY or cq = cp then return (cq, FIND-FIRST-SET(ballot_result) - 1)
9          cq ← $\mathcal{G}$[addr + offset]
10     byte $i$ ← GET-EVICTION-POS(cp)
11     if offset = $i$ then  cq ← ATOMICEXCH($\mathcal{G}$[addr + offset], cp)
12     cq ← bg.SHUFFLE(cq, $i$)
13     return (cq, $i$)
```

This function is presented in Alg. 4. Finally, if a value other than the original remainder `cp` or `EMPTY` is returned, another (remainder of a) node has been evicted, which is decompressed and returned at l.20-21. Otherwise, `p` is returned with its address (l.22). When Cuckoo hashing is not used, evictions do not occur, and at l.20-21, it is returned that the bucket is full.

Finally, we present HT-INSERT-CUCKOO in Alg. 4. The function HT-FIND is not presented, but it is almost equal to l.2-3 of Alg. 4. At l.2, each thread in `bg` reads its part of the bucket $\mathcal{G}$`[addr + offset]`, and checks if it contains `cp`, the remainder of `p`. If it is found anywhere in the bucket, the remainder with its position is returned (l.3). In the `while`-loop at l.4-9, it is attempted to insert `cp` in an empty position. In every iteration, an empty position is selected (l.5) and the corresponding thread tries to atomically insert `cp` (l.6). At l.7, the outcome is shared among the threads. If it is either `EMPTY` or the remainder itself, it can be returned (l.8). Otherwise, the bucket is read again (l.9). If insertion does not succeed, l.10 is reached, where a hash function is used by GET-EVICTION-POS to hash `cp` to a bucket position. The corresponding thread exchanges `cp` with the node stored at that position (l.11). After the evicted node has been shared with the other threads (l.12), it is returned together with its position (l.13).

6 Experiments

We implemented a code generator in PYTHON, using TEXTX [17] and JINJA2,[3] that accepts an SLCO model and produces CUDA C++ code to explore its state space. The code is compiled with CUDA 11.4 targeting compute capability 7.5. Experiments were conducted on a machine running LINUX MINT 20 with a 4-core INTEL CORE i7-7700 3.6 GHz, 32GB RAM, and a Titan RTX GPU.

The goal of the experiments is to assess how fast GPU next state computation with the tree database is w.r.t. 1) the various options we have for hashing, 2) state-of-the-art CPU tools, and 3) other GPU tools. For 2), we compare with multi-core Depth-First Search (DFS) of SPIN 6.5.1 [22] and (explicit-state) multi-core Breadth-First Search (BFS) of LTSMIN 3.0.2 [24,28].

[3] https://palletsprojects.com/p/jinja/.

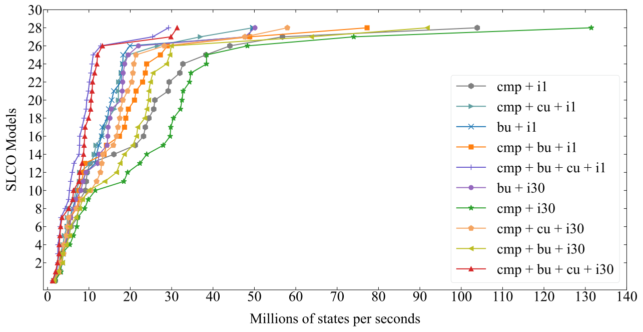

Fig. 4: Speed obtained by different GPU configurations.

In our implementation, we use 32 invertible hash functions. *Root compression* (CMP) can be turned on or off. When selected, we have a root table with 2^{32} elements, 32 bits each, and a non-root table with 2^{29} elements, 64 bits each. This enables storing 58-bit roots (two pointers to the non-root table) in $58 - 32 + \lceil \log_2(32) \rceil + 1 = 32$ bits. When using *buckets with more than one element* (CMP+BU), we have root buckets of size 8, and non-root buckets of size 16. The non-root buckets make full use of the cache line, but the root buckets do not. Making the latter larger means that too many bits for root addressing are lost for root compression to work (the remainders will be too large).

Root compression allows turning *Cuckoo hashing* on (CMP(+BU)+CU) or off (CMP(+BU)). When it is off, essentially Cleary-Cuckoo is still performed, except that evictions are not allowed, meaning that hashing fails as soon as all possible 32 buckets for a node are occupied.

In the configuration BU, neither root compression nor Cuckoo hashing is applied. We use one table with 2^{30} 64-bit elements and buckets of size 16. For reasons related to storing global addresses in the state cache, we cannot make the table larger. The 32 hash functions are used without allowing evictions.

Finally, *multiple iterations* can be run per kernel launch. Shared memory is wiped when a kernel execution terminates, but the state cache content can be reused from one iteration to the next when a kernel executes multiple iterations, by which trees already in the cache do not need to be fetched again from the tree database. We identified 30 iterations to be effective in general (i30), and experimented with a single iteration per kernel launch (i1).

With the CPU tools, we performed reachability analysis on 1- and 4-core configurations, denoted by SP-1 and SP-4 for SPIN, and LM-1 and LM-4 for LTSMIN. We only enabled state compression and basic reachability (without property checking), to favour fast exploration of large state spaces.

For benchmarks, we used models from the BEEM benchmarks [42] of concurrent systems, translated to SLCO and PROMELA (for SPIN). We scaled some of them up to have larger state spaces. Those are marked in Table 1 with '+'. Timeout is set to 3600 seconds for all benchmarks.

Table 1: Millions of states per second for various reachability tools and configurations. Pink cells: out of memory. Yellow cells: timeout. Green cell: best average. O.M.: out of memory at initialisation. SU: speedup of (CMP + i30) vs. (LM-1).

Input		CPU tools				GPUEXPLORE + SLCO Configurations								
Model	**States**	**SP-1**	**SP-4**	**LM-1**	**LM-4**	**BITS**	**CR**	**BU + i1**	**CMP + i1**	**CMP + BU + i1**	**CMP + CU + i1**	**CMP + i30**	**CMP + CU + i30**	**SU**
adding.20+	84,709,120	1.128	3.223	1.211	3.938	100	1.96	49.597	56.793	48.879	36.934	**74.026**	47.694	**61x**
adding.50+	529,767,730	🟥 0.856	🟥 O.M.	1.354	5.356	100	1.96	48.403	103.872	77.243	49.625	**131.444**	57.968	**97x**
anderson.6	18,206,917	0.623	1.362	0.516	1.309	122	1.82	14.814	16.035	13.647	11.265	**34.111**	17.649	**62x**
anderson.7	538,699,029	🟥 0.599	🟥 O.M.	🟨 0.448	🟥 1.583	141	2.75	9.309	21.192	14.244	10.426	**22.326**	10.435	**41x**
at.5	31,999,440	0.646	1.495	0.653	1.880	85	1.86	19.894	29.158	23.633	18.204	**38.457**	21.375	**59x**
at.6	160,589,600	0.454	0.869	0.695	2.387	85	1.90	17.901	38.275	27.275	19.498	**38.418**	20.359	**55x**
at.7	819,243,816	🟥 0.527	🟥 O.M.	0.666	2.372	97	1.98	12.415	**23.629**	17.381	13.194	22.329	13.378	**34x**
at.8+	3,739,953,204	🟥 0.534	🟥 O.M.	🟨 0.555	🟥 1.817	97	1.97	🟥 5.452	🟥 7.246	🟥 7.593	11.698	🟥 7.287	**11.854**	**13x**
bakery.5	7,866,401	1.400	2.570	0.410	0.904	140	2.51	11.504	7.838	7.585	6.407	**19.362**	12.782	**47x**
bakery.7	29,047,471	1.228	2.592	0.580	1.618	140	2.49	13.236	9.361	9.021	7.698	**29.783**	17.456	**51x**
bakery.8	841,696,300	0.760	1.269	0.690	2.436	140	2.40	🟥 3.745	29.410	23.957	17.116	**32.778**	18.215	**48x**
elevator2.3	7,667,712	0.554	1.099	0.463	0.985	189	3.96	4.890	3.259	3.185	2.817	**6.261**	4.827	**14x**
elevator2.4	91,226,112	0.263	0.561	0.623	1.945	213	3.97	3.025	**3.746**	2.907	3.087	3.267	2.703	**5x**
elevator2.5+	1,016,070,144	🟥 0.189	🟥 O.M.	0.473	1.630	317	5.95	🟥 1.540	**1.871**	1.545	1.520	1.839	1.491	**4x**
frogs.4	17,443,219	1.044	2.228	0.553	1.423	219	3.49	8.423	10.253	8.686	7.767	**11.549**	8.168	**21x**
frogs.5	182,772,126	0.531	1.048	0.751	2.630	251	3.84	6.766	9.573	8.214	6.898	**9.846**	6.943	**13x**
lamport.6	8,717,688	1.277	1.375	0.490	1.096	96	1.91	11.813	5.126	5.225	4.697	**27.966**	19.335	**57x**
lamport.7	38,717,846	1.001	1.822	0.672	1.979	116	1.98	18.176	23.205	18.915	16.170	**34.321**	20.641	**51x**
lamport.8	62,669,317	0.917	1.776	0.698	2.194	116	1.98	17.717	25.947	21.015	17.132	**35.387**	20.864	**50x**
loyd.2	362,880	1.278	0.758	0.255	0.497	90	1.05	**7.339**	4.204	4.220	3.723	3.243	3.930	**13x**
loyd.3	239,500,800	🟥 0.633	🟥 O.M.	0.650	2.338	114	1.96	18.268	44.073	28.970	26.556	**48.328**	28.248	**74x**
mcs.5	60,556,519	0.706	0.615	0.453	1.489	148	2.97	14.504	24.498	19.537	14.710	**29.635**	15.912	**65x**
mcs.6	332,544	1.240	0.244	0.181	0.331	156	2.75	**6.037**	3.003	3.097	2.751	3.446	3.131	**19x**
peterson.5	131,064,750	0.711	1.617	0.727	2.435	140	2.98	16.034	31.975	21.394	17.813	**32.331**	16.681	**42x**
peterson.6	174,495,861	0.852	0.756	0.720	2.451	140	2.98	15.503	32.725	22.975	17.198	**34.902**	17.030	**45x**
peterson.7	142,471,098	0.683	1.496	0.652	2.269	175	2.63	13.077	25.667	18.603	13.868	**26.183**	13.120	**37x**
phils.6	14,348,906	0.208	0.422	0.240	0.670	150	1.49	4.410	**7.458**	5.528	4.789	7.084	4.543	**30x**
phils.7	71,934,773	0.179	0.297	0.246	0.764	151	1.49	3.585	**5.702**	4.762	4.064	5.382	3.885	**22x**
phils.8	43,046,720	0.160	0.361	0.243	0.788	160	1.49	4.842	**9.151**	6.987	5.119	8.973	5.089	**37x**
szymanski.5	79,518,740	0.665	1.571	0.535	1.815	180	2.91	11.944	17.803	14.416	11.653	**18.357**	11.674	**33x**
Average		0.728	1.309	0.58	1.844	n/a		13.139	21.068	16.355	12.813	🟩 **26.621**	15.246	**40x**

Fig. 4 compares the speeds of the different GPU configurations in millions of states per second, averaged over 5 runs. For each configuration, we sorted the data to observe the overall trend. The higher the speed the better. The CMP + i30 mode (without Cuckoo hashing or larger buckets) is the fastest for the majority of models. On the other hand, it fails to complete exploration for `at.8`, the largest state space with 3.7 billion states, due to running out of memory. If Cuckoo hashing is enabled with root compression, all state spaces are successfully explored, which confirms that higher load factors can be achieved [4]. However, Cuckoo hashing negatively impacts performance, which contradicts [4]. Although it is difficult to pinpoint the cause for this, it is clear that it results from our hashing being done in addition to the exploration tasks, while in papers on GPU hash tables [1,4], hashing is analysed in isolation. With the extra variables and operations needed for exploration, hashing should be lightweight, and Cuckoo hashing introduces handling evictions. The more complex code is compiled to a less performant program, even when evictions do not occur.

Table 1 compares GPU performance with SPIN and LTSMIN. We refer to our tool as GPUEXPLORE + SLCO. From the results of Fig. 4, we selected a set of configurations demonstrating the impact of the various options. For each model, BITS and CR gives the state vector length in bits and the compression ratio, defined as (number of roots $\times$ number of leaves per tree) / (number of nodes). With the compression ratio, we measure how effective the node sharing is, compared to if we had stored each state individually without sharing. In

Table 2: Millions of states per second for various GPU tools.

Tool	anderson.6	anderson.7	lamport.8	peterson.5	peterson.6	peterson.7	szymanski.5
Grapple	2.138	14.299	n/a	10.941	9.074	8.967	n/a
GPUexplore 2.0	15.863	8.737	33.063	16.874	16.705	13.581	**26.454**
GPUexplore + Slco (cmp+i30)	**34.111**	**22.326**	**35.387**	**32.331**	**34.902**	**26.183**	18.357

addition, the speed in millions of states per second is given. Regarding out of memory, we are aware that Spin has other, slower, compression options, but we only considered the fastest, to favour the CPU speeds. Times are restricted to exploration; code generation and compilation always take a few seconds. The best GPU results are highlighted in bold. To compute the speedup (SU), the result of cmp + i30, the overall best configuration, has been divided by the Lm-1 result (the single-core configuration that completely explored all state spaces except one). All GPU experiments have been done with 512 threads per block, and 3,240 blocks (45 blocks per SM). We identified this configuration as being effective for `anderson.6`, and used it for all models.

While LTSmin tends to achieve near-linear speed-ups (compare Lm-1 and Lm-4), the speed of GPUexplore + Slco heavily depends on the model. For some models, as the state spaces of instances become larger, the speed increases, and for others, it decreases. The exact cause for this is hard to identify, and we plan to work on further optimisations. For instance, the branching factor, i.e., average number of successors of a state, plays a role here, as large branching factors favour parallel computation (many threads will become active quickly).

Our overall fastest configuration does not use larger buckets, nor Cuckoo hashing. Regarding buckets, as already noted in Section 3, starting with the Turing architecture, NVIDIA GPUs are less sensitive to uncoalesced accesses, and our results confirm that. Performing fewer tasks in parallel seems to be more harmful for performance than a larger number of uncoalesced accesses.

Finally, Table 2 compares GPUexplore + Slco with GPUexplore 2.0 and Grapple. A comparison with ParaMoc was not possible, as it targets very different types of (sequential) models. The models we selected are those available for at least two of the tools we considered. Unfortunately, Grapple does not (yet) support reading Promela models. Instead, a number of models are encoded directly into its source code, and we were limited to checking only those models. It can be observed that in the majority of cases, our tool achieves the highest speeds, which is surprising, as the trees we use tend to lead to more global memory accesses, but it is also encouraging to further pursue this direction.

7 Conclusions and Future Work

We discussed new algorithms to achieve a GPU tree database, which enables memory-efficient explicit state space exploration for FSMs with data. We proposed Cleary-Cuckoo hashing, which makes it possible to use, for the first time, Cleary compression on GPUs. Experiments show processing speeds of up to 131 million trees per second. In the last decade, new GPUs have been increasingly effective for state space exploration [10], and in the future, they are expected to be more capable of handling thread divergence, which still heavily occurs when accessing $\mathcal{G}$. Therefore, we are optimistic about further improvements. In the future, we will focus on optimisations and verifying temporal logic formulae.

Data Availability Statement. The datasets generated and analysed during the current study are available in the Zenodo repository [39].

References

1. Alcantara, D.A., Volkov, V., Sengupta, S., Mitzenmacher, M., Owens, J.D., Amenta, N.: Building an Efficient Hash Table on the GPU. In: GPU Computing Gems Jade Edition, pp. 39–53. Morgan Kaufmann Publishers Inc. (2012). https://doi.org/10.1016/B978-0-12-385963-1.00004-6
2. Amble, O., Knuth, D.: Ordered Hash Tables. The Computer Journal **17**(2), 135–142 (1974). https://doi.org/10.1093/comjnl/17.2.135
3. Ashkiani, S., Farach-Colton, M., Owens, J.: A Dynamic Hash Table for the GPU. In: IPDPS. pp. 419–429. ACM (2018). https://doi.org/10.1109/IPDPS.2018.00052
4. Awad, M., Ashkiani, S., Porumbescu, S., Farach-Colton, M., Owens, J.: Better GPU Hash Tables. Tech. Rep. 2108.07232, arXiV (2021). https://doi.org/10.48550/arXiv.2108.07232
5. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing Fast LTL Model Checking Algorithms for Many-Core GPUs. JPDC **72**(9), 1083–1097 (2012). https://doi.org/10.1016/j.jpdc.2011.10.015
6. Bartocci, E., DeFrancisco, R., Smolka, S.A.: Towards a GPGPU-parallel SPIN Model Checker. In: SPIN 2014. pp. 87–96. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2632362.2632379
7. Blom, S., Lisser, B., van de Pol, J., Weber, M.: A Database Approach to Distributed State Space Generation. Electron. Notes Theor. Comput. Sci. **198**(1), 17–32 (2008). https://doi.org/10.1016/j.entcs.2007.10.018
8. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel Probabilistic Model Checking on General Purpose Graphics Processors. STTT **13**(1), 21–35 (2011). https://doi.org/10.1007/s10009-010-0176-4
9. Bussi, L., Ciancia, V., Gadducci, F.: Towards a Spatial Model Checker on GPU. In: FORTE. LNCS, vol. 12719, pp. 188–196. Springer (2021). https://doi.org/10.1007/978-3-030-78089-0_12
10. Cassee, N., Neele, T., Wijs, A.: On the Scalability of the GPUexplore Explicit-State Model Checker. In: GaM. EPTCS, vol. 263, pp. 38–52. Open Publishing Association (2017). https://doi.org/10.4204/EPTCS.263.4
11. Cassee, N., Wijs, A.: Analysing the Performance of GPU Hash Tables for State Space Exploration. In: GaM. pp. 1–15. EPTCS, Open Publishing Association (2017). https://doi.org/10.4204/EPTCS.263.1
12. Češka, M., Pilař, P., Paoletti, N., Brim, L., Kwiatkowska, M.: PRISM-PSY: Precise GPU-Accelerated Parameter Synthesis for Stochastic Systems. In: TACAS. LNCS, vol. 9636, pp. 367–384. Springer (2016). https://doi.org/10.1007/978-3-642-54862-8
13. Cleary, J.: Compact Hash Tables Using Bidirectional Linear Probing. IEEE Trans. on Computers **c-33**(9), 828–834 (1984). https://doi.org/10.1109/TC.1984.1676499
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press (2009)

15. Darragh, J., Cleary, J., Witten, I.: Bonsai: A Compact Representation of Trees. Software - Practice and Experience **23**(3), 277–291 (1993). https://doi.org/10.1002/spe.4380230305
16. DeFrancisco, R., Cho, S., Ferdman, M., Smolka, S.A.: Swarm model checking on the GPU. Int. J. Softw. Tools Technol. Transf. **22**(5), 583–599 (2020). https://doi.org/10.1007/s10009-020-00576-x
17. Dejanović, I., Vaderna, R., Milosavljević, G., Vuković, Ž.: TextX: A Python tool for Domain-Specific Language implementation. Knowledge-Based Systems **115**, 1–4 (2017). https://doi.org/10.1016/j.knosys.2016.10.023
18. Edelkamp, S., Sulewski, D.: Efficient Explicit-State Model Checking on General Purpose Graphics Processors. In: SPIN. LNCS, vol. 6349, pp. 106–123. Springer (2010). https://doi.org/10.1007/978-3-642-16164-3_8
19. Edelkamp, S., Sulewski, D.: External memory breadth-first search with delayed duplicate detection on the GPU. In: MoChArt. LNCS, vol. 6572, pp. 12–31. Springer (2010). https://doi.org/10.1007/978-3-642-20674-0_2
20. García, I., Lefebvre, S., Hornus, S., Lasram, A.: Coherent Parallel Hashing. ACM Trans. Graph. **30**(6), 161 (2011). https://doi.org/10.1145/2070781.2024195
21. Holzmann, G.: The Model Checker Spin. IEEE Trans. Software Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
22. Holzmann, G., Bošnački, D.: The Design of a Multicore Extension of the SPIN Model Checker. IEEE Trans. on Software Engineering **33**(10), 659–674 (2007). https://doi.org/10.1109/TSE.2007.70724
23. Jünger, D., Kobus, R., Müller, A., Hundt, C., Xu, K., Liu, W., Schmidt, B.: WarpCore: A Library for Fast Hash Tables. In: HiPC. pp. 11–20. IEEE (2020). https://doi.org/10.1109/HiPC50609.2020.00015
24. Kant, G., Laarman, A., Meijer, J., Pol, J.v., Blom, S., Dijk, T.: LTSmin: High-Performance Language-Independent Model Checking. In: TACAS. LNCS, vol. 9035, pp. 692–707. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_61
25. Khan, M., Hassan, O., Khan, S.: Accelerating SpMV Multiplication in Probabilistic Model Checkers Using GPUs. In: ICTAC. LNCS, vol. 12819, pp. 86–104. Springer (2021). https://doi.org/10.1007/978-3-030-85315-0_6
26. Laarman, A.: Optimal Compression of Combinatorial State Spaces. Innov. Syst. Softw. Eng. **15**, 235–251 (2019). https://doi.org/10.1007/s11334-019-00341-7
27. Laarman, A., van de Pol, J., Weber, M.: Parallel Recursive State Compression for Free. In: SPIN. LNCS, vol. 6823, pp. 38–56. Springer (2011). https://doi.org/10.1007/978-3-642-22306-8_4
28. Laarman, A.: Scalable Multi-Core Model Checking. Ph.D. thesis, University of Twente (2014). https://doi.org/10.3990/1.9789036536561
29. Lee, C.: Representation of Switching Circuits by Binary-Decision Programs. Bell System Technical Journal **38**, 985–999 (1959). https://doi.org/10.1002/j.1538-7305.1959.tb01585.x
30. Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D., Schardl, T.B.: There's Plenty of Room at the Top: What Will Drive Computer Performance After Moore's Law? Science **368**(6495) (2020). https://doi.org/10.1126/science.aam9744
31. Lessley, B.: Data-Parallel Hashing Techniques for GPU Architectures. IEEE Trans. Parallel Distributed Syst. **31**(1), 237–250 (2019). https://doi.org/10.1109/TPDS.2019.2929768

32. Merrill, D., Grimshaw, A.: High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing. Parallel Process. Lett. **21**(2), 245–272 (2011). https://doi.org/10.1142/S0129626411000187
33. Neele, T., Wijs, A., Bošnački, D., van de Pol, J.: Partial Order Reduction for GPU Model Checking. In: ATVA. LNCS, vol. 9938, pp. 357–374. Springer (2016). https://doi.org/10.1007/978-3-319-46520-3_23
34. Osama, M.: GPU Enabled Automated Reasoning. Ph.D. thesis, Eindhoven University of Technology (2022), ISBN: 978-90-386-5445-4
35. Osama, M., Gaber, L., Hussein, A.I., Mahmoud, H.: An Efficient SAT-Based Test Generation Algorithm with GPU Accelerator. J. Electron. Test. **34**(5), 511–527 (2018). https://doi.org/10.1007/s10836-018-5747-4
36. Osama, M., Wijs, A.: Parallel SAT Simplification on GPU Architectures. In: TACAS. LNCS, vol. 11427, pp. 21–40. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_2
37. Osama, M., Wijs, A.: SIGmA: GPU Accelerated Simplification of SAT Formulas. In: IFM. LNCS, vol. 11918, pp. 514–522. Springer (2019). https://doi.org/10.1007/978-3-030-34968-4_29
38. Osama, M., Wijs, A.: GPU Acceleration of Bounded Model Checking with ParaFROST. In: CAV, Part II. LNCS, vol. 12760, pp. 447–460. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_21
39. Osama, M., Wijs, A.: Artifact for A GPU Tree Database for Many-Core Explicit State Space Exploration (2023). https://doi.org/10.5281/zenodo.7509129
40. Osama, M., Wijs, A., Biere, A.: SAT Solving with GPU Accelerated Inprocessing. In: TACAS. LNCS, vol. 12651, pp. 133–151. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_8
41. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: ESA. LNCS, vol. 2161, pp. 121–133. Springer (2001). https://doi.org/10.1007/3-540-44676-1_10
42. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: SPIN 2007. LNCS, vol. 4595, pp. 263–267 (2007). https://doi.org/10.1007/978-3-540-73370-6_17
43. Prevot, N., Soos, M., Meel, K.: Leveraging GPUs for Effective Clause Sharing in Parallel SAT Solving. In: SAT. LNCS, vol. 12831, pp. 471–487. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_32
44. de Putter, S., Wijs, A., Zhang, D.: The SLCO Framework for Verified, Model-driven Construction of Component Software. In: FACS. Lecture Notes in Computer Science, vol. 11222, pp. 288–296. Springer (2018). https://doi.org/10.1007/978-3-030-02146-7_15
45. van der Vegt, S., Laarman, A.: A Parallel Compact Hash Table. In: MEMICS. LNCS, vol. 7119, pp. 191–204. Springer (2011). https://doi.org/10.1007/978-3-642-25929-6_18
46. Wei, H., Chen, X., Ye, X., Fu, N., Huang, Y., Shi, J.: Parallel Model Checking on Pushdown Systems. In: ISPA/IUCC/BDCloud/SocialCom/SustainCom. pp. 88–95. IEEE (2018). https://doi.org/10.1109/BDCloud.2018.00026
47. Wei, H., Ye, X., Shi, J., Huang, Y.: ParaMoC: A Parallel Model Checker for Pushdown Systems. In: ICA3PP. LNCS, vol. 11945, pp. 305–312. Springer (2019). https://doi.org/10.1007/978-3-030-38961-1_26
48. Wijs, A., Bošnački, D.: Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In: SPIN. LNCS, vol. 7385, pp. 98–116. Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_9

49. Wijs, A.: BFS-Based Model Checking of Linear-Time Properties With An Application on GPUs. In: CAV, Part II. LNCS, vol. 9780, pp. 472–493. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_26
50. Wijs, A., Bošnački, D.: GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs. In: TACAS. LNCS, vol. 8413, pp. 233–247 (2014). https://doi.org/10.1007/978-3-642-54862-8_16
51. Wijs, A., Bošnački, D.: Many-Core On-The-Fly Model Checking of Safety Properties Using GPUs. STTT **18**(2), 169–185 (2016). https://doi.org/10.1007/s10009-015-0379-9
52. Wijs, A., Katoen, J.P., Bošnački, D.: Efficient GPU Algorithms for Parallel Decomposition of Graphs into Strongly Connected and Maximal End Components. Formal Methods Syst. Des. **48**(3), 274–300 (2016). https://doi.org/10.1007/s10703-016-0246-7
53. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking. In: FM. LNCS, vol. 9995, pp. 694–701. Springer (2016). https://doi.org/10.1007/978-3-319-48989-6_42
54. Wu, Z., Liu, Y., Liang, Y., Sun, J.: GPU Accelerated Counterexample Generation in LTL Model Checking. In: ICFEM. LNCS, vol. 8829, pp. 413–429. Springer (2014). https://doi.org/10.1007/978-3-319-11737-9_27
55. Wu, Z., Liu, Y., Sun, J., Shi, J., Qin, S.: GPU Accelerated On-the-Fly Reachability Checking. In: ICECCS. pp. 100–109 (2015). https://doi.org/10.1109/ICECCS.2015.21
56. Youness, H., Osama, M., Hussein, A., Moness, M., Hassan, A.M.: An Effective SAT Solver Utilizing ACO Based on Heterogenous Systems. IEEE Access **8**, 102920–102934 (2020). https://doi.org/10.1109/ACCESS.2020.2999382
57. Youness, H.A., Ibraheim, A., Moness, M., Osama, M.: An Efficient Implementation of Ant Colony Optimization on GPU for the Satisfiability Problem. In: PDP. pp. 230–235. IEEE (2015). https://doi.org/10.1109/PDP.2015.59