GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs

Anton Wijs* and Dragan Bošnački

Eindhoven University of Technology, The Netherlands

Abstract In recent years, General Purpose Graphics Processors (GPUs) have been successfully applied in multiple application domains to drastically speed up computations. Model checking is an automatic method to formally verify the correctness of a system specification. Such specifications can be viewed as implicit descriptions of a large directed graph or state space, and for most model checking operations, this graph must be analysed. Constructing it, or on-the-fly exploring it, however, is computationally intensive, so it makes sense to try to implement this for GPUs. In this paper, we explain the limitations involved, and how to overcome these. We discuss the possible approaches involving related work, and propose an alternative, using a new hash table approach for GPUs. Experimental results with our prototype implementations show significant speed-ups compared to the established sequential counterparts.

1 Introduction

General Purpose Graphics Processing Units (GPUs) are being applied successfully in many areas of research to speed up computations. Model checking [1] is an automatic technique to verify that a given specification of a complex, safety-critical (usually embedded) system meets a particular functional property. It involves very time and memory demanding computations. Many computations rely on *on-the-fly state space exploration*. This incorporates interpreting the specification, resulting in building a graph, or state space, describing all its potential behaviour. Hence, the state space is not explicitly given, but implicitly, through the specification. The state space size is not known a priori.

GPUs have been successfully applied to perform computations for probabilistic model checking, when the state space is given a priori [2–4]. However, no attempts as of yet have been made to perform the exploration itself entirely using GPUs, due to it not naturally fitting the data parallel approach of GPUs, but in this paper, we propose a way to do so. Even though current GPUs have a limited amount of memory, we believe it is relevant to investigate the possibilities of GPU state space exploration, if only to be prepared for future hardware

* This work was sponsored by the NWO Exacte Wetenschappen, EW (NWO Physical Sciences Division) for the use of supercomputer facilities, with financial support from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organisation for Scientific Research, NWO).

E. Ábrahám and K. Havelund (Eds.): TACAS 2014, LNCS 8413, pp. 233–247, 2014.
© Springer-Verlag Berlin Heidelberg 2014

developments (for example, GPUs are already being integrated in CPUs). We also believe that the results reported in this paper can be relevant for solving other on-the-fly graph problems. In this paper, we describe several options to implement basic state space exploration, i.e. reachability analysis, for explicit-state model checking on GPUs. We focus on CUDA-enabled GPUs of NVIDIA, but the options can also be implemented using other interfaces. We experimentally compare these options, and draw conclusions. Where relevant, we use techniques from related work, but practically all related implementations are focussed on explicit graph searching, in which the explicit graph is given, as opposed to on-the-fly constructing the graph. The structure of the paper is as follows: in Section 2, the required background information is given. Then, Section 3 contains the description of several implementations using different extensions. In Section 4, experimental results are shown, and finally, Section 5 contains conclusions and discusses possible future work.

2 Background and Related Work

2.1 State Space Exploration

The first question is how a specification should be represented. Most descriptions, unfortunately, are not very suitable for our purpose, since they require the dynamic construction of a database of data terms during the exploration. GPUs are particularly unsuitable for dynamic memory allocation. We choose to use a slightly modified version of the *networks of LTSs* model [5]. In such a network, the possible behaviour of each process or component of the concurrent system design is represented by a *process* LTS, or *Labelled Transition System*. An LTS is a directed graph in which the vertices represent states of a process, and the edges represent transitions between states. Moreover, each edge has a label indicating the event that is fired by the process. Finally, an LTS has an initial state s_I. A network of LTSs is able to capture the semantics of specifications with *finite-state* processes at a level where all data has been abstracted away and only states remain. It is used in particular in the CADP verification toolbox [6]. Infinite-state processes are out of the scope here, and are considered future work.

In the remainder of this paper, we use the following notations: a network contains a vector Π of n process LTSs, with $n \in \mathbb{N}$. Given an integer $n > 0$, $1..n$ is the set of integers ranging from 1 to n. A vector $\overline{v}$ of size n contains n elements indexed by $1..n$. For $i \in 1..n$, $\overline{v}[i]$ denotes element i in $\overline{v}$, hence $\Pi[i]$ refers to the ith LTS.

Besides a finite number of LTSs, a network also contains a finite set $\mathcal{V}$ of *synchronisation rules*, describing how behaviour of different processes should synchronise. Through this mechanism, it is possible to model synchronous communication between processes. Each rule $\langle \overline{t}, \alpha \rangle$ consists of a vector $\overline{t}$ of size n, describing the process events it is applicable on, and a result α, i.e. the system event resulting from a successful synchronisation. As an example, consider the first two LTSs from the left in Figure 1, together defining a network with $n = 2$

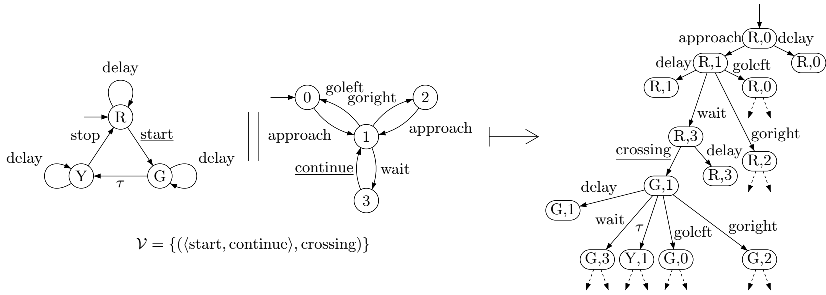

Fig. 1. Exploring the state space of a traffic light specification

of a simple traffic light system specification, where process 0 represents the behaviour of a traffic light (the states representing the colours of the light) and process 1 represents a pedestrian. We also have $\mathcal{V} = \{(\langle start, continue \rangle, crossing)\}$, meaning that there is only a single synchronisation rule, expressing that the *start* event of process 0 can only be fired if event *continue* of process 1 is fired at the same time, resulting in the event *crossing* being fired by the system as a whole.

In general, synchronisation rules are not required to involve all processes; in order to express that a rule is not applicable on a process $i \in 1..n$, we use a dummy value $\bullet$ indicating this, and define $\overline{t}[i] = \bullet$.

State space exploration now commences as follows: first, the two initial states of the processes (indicated by an incoming transition without a source state) are combined into a system state vector $\overline{s} = \langle R, 0 \rangle$. In general, given a vector $\overline{s}$, the corresponding state of $\Pi[i]$, with $i \in 1..n$, is $\overline{s}[i]$. The set of outgoing transitions (and their corresponding target states or successors of $\overline{s}$) can now be determined using two checks for each transition $\overline{s}[i] \xrightarrow{a} p_i$, with p_i a state of process i:

1. $\neg\exists\langle\overline{t}, \alpha\rangle \in \mathcal{V}.\overline{t}[i] = a \implies \overline{s} \xrightarrow{a} \overline{s}'$ with $\overline{s}'[i] = p_i \wedge \forall j \in 1..n \setminus \{i\}.\overline{s}'[j] = \overline{s}[j]$
2. $\forall\langle\overline{t}, \alpha\rangle \in \mathcal{V}.\overline{t}[i] = a \wedge (\forall j \in 1..n \setminus \{i\}.\overline{t}[j] \neq \bullet \implies \overline{s}[j] \xrightarrow{\overline{t}[j]} p_j) \implies \overline{s} \xrightarrow{\alpha} \overline{s}'$ with $\forall j \in 1..n.(\overline{t}[j] = \bullet \wedge \overline{s}'[j] = \overline{s}[j]) \vee (\overline{t}[j] \neq \bullet \wedge \overline{s}'[j] = p_j)$

The first check is applicable for all *independent* transitions, i.e. transitions on which no rule is applicable, hence they can be fired individually, and therefore directly 'lifted' to the system level. The second check involves applying synchronisation rules. In Figure 1, part of the system state space obtained by applying the defined checks on the traffic network is displayed on the right.

2.2 GPU Programming

NVIDIA GPUs can be programmed using the CUDA interface, which extends the C and FORTRAN programming languages. These GPUs contain tens of streaming multiprocessors (SM) (see Figure 2, with N the number of SMs), each containing a fixed number of streaming processors (SP), e.g. 192 for the Kepler K20 GPU,

and fast on-chip *shared memory*. Each SM employs single instruction, multiple data (SIMD) techniques, allowing for data parallelisation. A single instruction stream is performed by a fixed size group of threads called a *warp*. Threads in a warp share a program counter, hence perform instructions in lock-step. Due to this, *branch divergence* can occur within a warp, which should be avoided: for instance, consider the if-then-else construct **if (C) then A else B**. If a warp needs to execute this, and for at least one thread **C** holds, then *all* threads must step through **A**. It is therefore possible that the threads must step together through both **A** and **B**, thereby decreasing performance. The size of a warp is fixed and depends on the GPU type, usually it is 32, we refer to it as *WarpSize*. A *block* of threads is a larger group assigned to a single SM. The threads in a block can use the shared memory to communicate with each other. An SM, however, can handle many blocks in parallel. Instructions to be performed by GPU threads can be defined in a function called a *kernel*. When launching a kernel, one can specify how many thread blocks should execute it, and how many threads each block contains (usually a power of two). Each SM then schedules all the threads of its assigned blocks up to the warp level. Data parallelisation can be achieved by using the predefined keywords *BlockId* and *ThreadId*, referring to ID of the block a thread resides in, and the ID of a thread within its block, respectively. Besides that, we refer with *WarpNr* to the global ID of a warp, and with *WarpTId* to the ID of a thread within its warp. These can be computed as follows: $WarpNr = ThreadId / WarpSize$ and $WarpTId = ThreadId \,\%\, WarpSize$.

Most of the data used by a GPU application resides in *global memory* or device memory. It embodies the interface between the host (CPU) and the kernel (GPU). Depending on the GPU type, its size is typically between 1 and 6 GB. It has a high bandwidth, but also a high latency, therefore memory caches are used. The cache line of most current NVIDIA GPU L1 and L2 caches is 128 Bytes, which directly corresponds with each thread in a warp fetching a 32-bit integer. If memory accesses in a kernel can be coalesced within each warp, efficient fetching can be achieved, since then, the threads in a warp perform a single fetch together, nicely filling one cache line, instead of different fetches, which would be serialised by the GPU, thereby losing many clock-cycles. This plays an important role in the hash table implementation we propose.

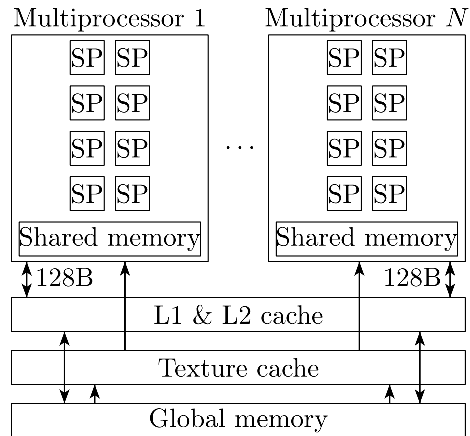

Fig. 2. Hardware model of CUDA GPUs

Finally, read-only data structures in global memory can be declared as *textures*, by which they are connected to a *texture cache*. This may be beneficial if

access to the data structure is expected to be random, since the cache may help in avoiding some global memory accesses.

2.3 Sparse Graph Search on GPUs

In general, the most suitable search strategy for parallelisation is *Breadth-First Search* (BFS), since each search level is a set of vertices that can be distributed over multiple workers. Two operations dominate in BFS: *neighbour gathering*, i.e. obtaining the list of vertices reachable from a given vertex via one edge, and *status lookup*, i.e. determining whether a vertex has already been visited before. There exist many parallelisations of BFS; here, we will focus on GPU versions. Concerning model checking, [7] describes the only other GPU on-line exploration we found, but it uses both the CPU and GPU, restricting the GPU to neighbour gathering, and it uses bitstate hashing, hence it is not guaranteed to be exhaustive. In [8], explicit state spaces are analysed.

The vast majority of GPU BFS implementations are quadratic parallelisations, e.g. [9,10]. To mitigate the dependency of memory accesses on the graph structure, each vertex is considered in each iteration, yielding a complexity of $\mathcal{O}(|V|^2 + |E|)$, with V the set of vertices and E the set of edges. In [11], entire warps are used to obtain the neighbours of a vertex.

There are only a few linear parallelisations in the literature: in [12], a hierarchical scheme is described using serial neighbour gathering and multiple queues to avoid high contention on a single queue. In [13], an approach using prefix sum is suggested, and a thorough analysis is made to determine how gatherings and lookups need to be placed in kernels for maximum performance.

All these approaches are, however, not directly suitable for on-the-fly exploration. First of all, they implement status lookups by maintaining an array, but in on-the-fly exploration, the required size of such an array is not known a priori. Second of all, they focus on using an adjacency matrix, but for on-the-fly exploration, this is not available, and the memory access patterns are likely to be very different.

Related to the first objection, the use of a hash table seems unavoidable. Not many GPU hash table implementations have been reported, but the ones in [14,15] are notable. They are both based on *Cuckoo-hashing* [16]. In Cuckoo hashing, collisions are resolved by shuffling the elements along to new locations using multiple hash functions. Whenever an element must be inserted, and hash function h_1 refers it to a location l already populated by another element, then the latter element is replaced using the next hash function for that element, i.e. if it was placed in l using hash function h_i, then function $h_{i+1} mod\ k_c$, with k_c the number of hash functions, is used. In [15], it is suggested to set $k_c = 4$.

Finally, in [14,15], a comparison is made to radix sorting, in particular of [17]. On a GPU, sorting can achieve high throughput, due to the regular access patterns, making list insertion and sorting faster than hash table insertion. Lookups, however, are slower than hash table lookups if one uses binary searches, as is done in [14,15]. An alternative is to use B-trees for storing elements, improving

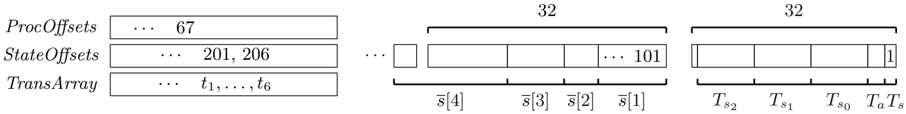

Fig. 3. Encodings of a network, a state vector and a transition entry

memory access patterns by grouping the elements in warp-segments.[1] Although we have chosen to use a hash table approach (for on-the-fly exploration, we experience that the sorting approach is overly complicated, requiring many additional steps), we will use this idea of warp-segments for our hash table.

3 GPU Parallelisation

Alg. 1 provides a high-level view of state space exploration. As in BFS, one can clearly identify the two main operations, namely *successor generation* (line 4), analogous to neighbour gathering, and *duplicate detection* (line 5), analogous to status lookup. Finally, in lines 6-7, states are added to the work sets, *Visited* being the set of visited states and *Open* being the set of states yet to be explored (usually implemented as a queue). In the next subsections, we will discuss our approach to implementing these operations.

3.1 Data Encoding

Algorithm 1. State space exploration

Require: network $\langle \Pi, \mathcal{V} \rangle$, initial state $\overline{s_I}$
 $Open, Visited \leftarrow \{\overline{s_I}\}$
2: **while** $Open \neq \emptyset$ **do**
 $\overline{s} \leftarrow Open; Open \leftarrow Open \setminus \overline{s}$
4: **for all** $\overline{s}' \in$ **constructSystemSuccs**$(\overline{s})$ **do**
 if $\overline{s}' \notin Visited$ **then**
6: $Visited \leftarrow Visited \cup \{\overline{s}'\}$
 $Open \leftarrow Open \cup \{\overline{s}'\}$

As mentioned before, memory access patterns are usually the main cause for performance loss in GPU graph traversal. The first step to minimise this effect is to choose appropriate encodings of the data. Figure 3 presents on the left how we encode a network into three 32-bit integer arrays. The first, called *ProcOffsets*, contains the start offset for each of the $\Pi[i]$ in the second array. The second array, *StateOffsets*, contains the offsets for the source states in the third array. Finally, the third array, *TransArray*, actually contains encodings of the outgoing transitions of each state. As an example, let us say we are interested in the outgoing transitions of state 5 of process LTS 8, in some given network. First, we look at position 8 in *ProcOffsets*, and find that the states of that process are listed starting from position 67. Then, we look at position 67+5 in *StateOffsets*, and we find that the outgoing transitions of state 5 are listed starting at position 201 in *TransArray*. Moreover, at position 67+6, we find the end of that list. Using these positions, we can iterate over the outgoing transitions in *TransArray*.

[1] See `http://www.moderngpu.com` (visited 18/4/2013).

One can imagine that these structures are practically going to be accessed randomly when exploring. However, since this data is never updated, we can store the arrays as textures, thereby using the texture caches to improve access.

Besides this, we must also encode the transition entries themselves. This is shown on the right of Figure 3. Each entry fills a 32-bit integer as much as possible. It contains the following information: the lowest bit (T_s) indicates whether or not the transition depends on a synchronisation rule. The next $\log_2(c_a)$ number of bits, with c_a the number of different labels in the entire network, encodes the transition label (T_a). We encode the labels, which are basically strings, by integer values, sorting the labels occurring in a network alphabetically. After that, each $\log_2(c_s)$ bits, with c_s the number of states in the process LTS owning this transition, encodes one of the target states. If there is non-determinism w.r.t. label T_a from the source state, multiple target states will be listed, possibly continuing in subsequent transition entries.

In the middle of Figure 3, the encoding of state vectors is shown. These are simply concatenations of encodings of process LTS states. Depending on the number of bits needed per LTS state, which in turn depends on the number of states in the LTSs, a fixed number of 32-bit integers is required per vector.

Finally, the synchronisation rules need to be encoded. To simplify this, we rewrite networks such that we only have rules involving a single label, e.g. $(\langle a, a\rangle, a)$. In practice, this can usually be done without changing the meaning. For the traffic light system, we could rewrite *start* and *continue* to *crossing*. It allows encoding the rules as bit sequences of size n, where for each process LTS, 1 indicates that the process should participate, and 0 that it should not participate in synchronisation. Two integer arrays then suffice, one containing these encodings, the other containing the offsets for all the labels.

3.2 Successor Generation

At the start of a search iteration, each block fetches a tile of new state vectors from the global memory. How this is done is explained at the end of Section 3.3. The tile size depends on the block size *BlockSize*.

On GPUs, one should realise fine-grained parallelism to obtain good speedups. Given the fact that each state vector consists of n states, and the outgoing transitions information needs to be fetched from physically separate parts of the memory, it is reasonable to assign n threads to each state vector to be explored. In other words, in each iteration, the tile size is at most *BlockSize*/n vectors. Assigning multiple threads per LTS for fetching, as in [11], does not lead to further speedups, since the number of transition entries to fetch is usually quite small due to the sparsity of the LTSs, as observed before by us in [4].

We group the threads into vector groups of size n to assign them to state vectors. Vector groups never cross warp boundaries, unless $n > 32$. The positive effect of this is that branch divergence can be kept to a minimum, since the threads in a vector group work on the same task. For a vector $\overline{s}$, each thread with ID i w.r.t. its vector group (the VGID) fetches the outgoing transitions of $\overline{s}[i+1]$. Each transition entry T with $T_s = 0$ can directly be processed, and

the corresponding target state vectors are stored for duplicate detection (see Section 3.3). For all transitions with $T_s = 1$, to achieve cooperation between the threads while limiting the amount of used shared memory, the threads iterate over their transitions in order of label ID (LID). To facilitate this, the entries in each segment of outgoing transitions belonging to a particular state in *TransArray* are sorted on LID before exploration starts.

Successors reached through synchronisation are constructed in iterations. In each iteration, the threads assigned to $\overline{s}$ fetch the entries with lowest LID and $T_s = 1$ from their list of outgoing transitions, and store these in a designated buffer in the shared memory. The size of this buffer can be determined before exploration as n times the maximum number of entries with the same LID and $T_s = 1$ from any process state in the network. Then, the thread with VGID 0, i.e. the vector group leader, determines the lowest LID fetched within the vector group. Figure 4 illustrates this for a vector with $n = 4$. Threads th_0 to th_3 have fetched transitions with the lowest LIDs for their respective process states that have not yet been processed in the successor generation, and thread th_0 has determined that the next lowest LID to be processed by the vector group is 1. This value is written in the *cnt* location. Since transitions in *TransArray* are sorted per state by LID, we know that all possible transitions with LID = 1 have been placed in the vector group buffer. Next, all threads that fetched entries with the lowest LID, in the example threads th_0 and th_2, start scanning the encodings of rules in $\mathcal{V}$ applicable on that LID. We say that thread i owns rule r iff there is no $j \in 1..n$ with $j < i$ and $r[j] \neq \bullet$. If a thread encounters a rule that it owns, then it checks the buffer contents to determine whether the rule is applicable. If it is, it constructs the target state vectors and stores them for duplicate detection. In the next iteration, all entries with lowest LID are removed, the corresponding threads fetch new entries, and the vector group leader determines the next lowest LID to be processed.

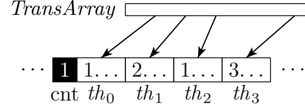

Fig. 4. Fetching transitions

3.3 Closed Set Maintenance

Local State Caching. As explained in Section 2, we choose to use a global memory hash table to store states. Research has shown that in state space exploration, due to the characteristics of most networks, there is a strong sense of locality, i.e. in each search iteration, the set of new state vectors is relatively small, and most of the already visited vectors have been visited about two iterations earlier [18, 19]. This allows effective use of block local *state caches* in shared memory. Such a cache, implemented as a linear probing hash table, can be consulted quickly, and many duplicates can already be detected, reducing the number of global memory accesses. We implemented the caches in a lockless way, apart from using a *compare-and-swap* (CAS) operation to store the first integer of a state vector.

When processing a tile, threads add successors to the cache. When finished, the block scans the cache, to check the presence of the successors in the global

hash table. Thus, caches also allow threads to cooperatively perform global duplicate detection and insertion of new vectors.

Global Hash Table. For the global hash table, we initially used the Cuckoo hash table of [15]. Cuckoo hashing has the nice property that lookups are done in constant time, namely, it requires k_c memory accesses, with k_c the number of hash functions used.

However, an important aspect of Cuckoo hashing is that elements are relocated in case collisions occur. In [15], key-value pairs are stored in 64-bit integers, hence insertions can be done atomically using CAS operations. Our state vectors, though, can encompass more than 64 bits, ruling out completely atomic insertions. After having created our own extension of the hash table of [15] that allows for larger elements, we experienced in experiments that the number of explored states far exceeded the actual number of reachable states, showing that in many cases, threads falsely conclude that a vector was not present (a *false negative*). We concluded that this is mainly due to vector relocation, involving non-atomic removal and insertion, which cannot be avoided for large vectors; once a thread starts removing a vector, it is not present anymore in the hash table until the subsequent insertion has finished, and any other thread looking for the vector will not be able to locate it during that time. It should however be noted, that although the false negatives may negatively influence the performance, they do not affect the correctness of our method.

To decrease the number of false negatives, as an alternative, we choose to implement a hash table using buckets, linear probing and bounded double hashing. It is implemented using an array, each consecutive *WarpSize* 32-bit integers forming a bucket. This plays to the strength of warps: when a block of threads is performing duplicate detection, all the threads in a warp cooperate on checking the presence of a particular $\overline{s}$. The first hash function h_1, built as specified in [15], is used to find the primary bucket. A warp can fetch a bucket with one memory access, since the bucket size directly corresponds with one cache line. Subsequently, the bucket contents can be checked in parallel by the warp. This is similar to the *walk-the-line* principle of [20], instead that here, the walk is done in parallel, so we call it *warp-the-line*. Note that each bucket can contain up to $\mathit{WarpSize}/c$ vectors, with c the number of 32-bit integers required for a vector. If the vector is not present and there is a free location, the vector is inserted. If the bucket is full, h_2 is used to jump to another bucket, and so on. This is similar to [21], instead that we do not move elements between buckets.

The pseudo-code for scanning the local cache and looking up and inserting new vectors (i.e. find-or-put) in the case that state vectors fit in a single 32-bit integer is displayed in Alg. 2. The implementation contains the more general case. The cache is declared **extern**, meaning that the size is given when launching the kernel. Once a work tile has been explored and the successors are in the cache, each thread participates in its warp to iterate over the cache contents (lines 6, 27). If a vector is new (line 8, note that empty slots are marked 'old'), insertion in the hash table will be tried up to $H \in \mathbb{N}$ times. In lines 11-13, warp-the-line is performed, each thread in a warp investigating the appropriate bucket slot. If any

Algorithm 2. Hash table find-or-put for single integer state vectors

```
    extern volatile __shared__ unsigned int cache []
 2: < process work tile and fill cache with successors >
    WarpNr ← ThreadId / WarpSize
 4: WarpTId ← ThreadId % WarpSize
    i ← WarpNr
 6: while i < |cache| do
      s̄ ← cache[i]
 8:   if isNewVector(s̄) then
        for j = 0 to H do
10:       BucketId ← h1(s̄)
          entry ← Visited[BucketId + WarpTId]
12:       if entry = s̄ then
            setOldVector(cache[i])
14:       s̄ ← cache[i]
          if isNewVector(s̄) then
16:         for l = 0 to WarpSize do
              if Visited[BucketId + l] = empty then
18:             if WarpTId = 0 then
                  old = atomicCAS(&Visited[BucketId + l], empty, s̄)
20:               if old = empty then
                    setOldVector(s̄)
22:           if ¬isNewVector(s̄) then
                break
24:       if ¬isNewVector(s̄) then
            break
26:       BucketId ← BucketId + h2(s̄)
      i ← i + BlockSize/WarpSize
```

thread sets $\overline{s}$ as old in line 13, then all threads will detect this in line 15, since $\overline{s}$ is read from shared memory. If the vector is not old, then it is attempted to insert it in the bucket (lines 15-23). This is done by the warp leader ($WarpTId = 0$, line 18), by performing a CAS. CAS takes three arguments, namely the address where the new value must be written, the expected value at the address, and the new value. It only writes the new value if the expected value is encountered, and returns the encountered value, therefore a successful write has happened if **empty** has been returned (line 20). Finally, in case of a full bucket, h_2 is used to jump to the next one (line 26).

As discussed in Section 4, we experienced good speedups and no unresolved collisions using a double hashing bound of 8, and, although still present, far fewer false negatives compared to Cuckoo hashing. Finally, it should be noted that chaining is not a suitable option on a GPU, since it requires memory allocation at runtime, and the required sizes of the chains are not known a priori.

Recall that the two important data structures are *Open* and *Visited*. Given the limited amount of global memory, and that the state space size is unknown a priori, we prefer to initially allocate as much memory as possible for *Visited*. But also the required size of *Open* is not known in advance, so how much memory should be allocated for it without potentially wasting some? We choose to combine the two in a single hash table by using the highest bit in each vector encoding to indicate whether it should still be explored or not. The drawback is that unexplored vectors are not physically close to each other in memory, but the typically large number of threads can together scan the memory relatively fast, and using one data structure drastically simplifies implementation. It has

the added benefit that load-balancing is handled by the hash functions, due to the fact that the distribution over the hash table achieves distribution over the workers. A consequence is that the search will not be strictly BFS, but this is not a requirement. At the start of an iteration, each block gathers a tile of new vectors by scanning predefined parts of the hash table, determined by the block ID. In the next section, several possible improvements on scanning are discussed.

3.4 Further Extensions

On top of the basic approach, we implemented the following extensions. First of all, instead of just one, we allow a variable number of search iterations to be performed within one kernel launch. This improves duplicate detection using the caches due to them maintaining more of the search history (shared memory data is lost once a kernel terminates). Second of all, building on the first extension, we implemented a technique we call *forwarding*. When multiple iterations are performed per launch, and a block is not in its final iteration, its threads will add the unexplored successors they generated in the current iteration to their own work tile for the next one. This reduces the need for scanning for new work.

4 Implementation and Experiments

We implemented the exploration techniques in CUDA for C.[2] The implementation was tested using 25 models from different sources; some originate from the distributions of the state-of-the-art model checking toolsets CADP [6] and mCRL2 [22], and some from the BEEM database [23]. In addition, we added two we created ourselves. Here, we discuss the results for a representative subset.

Sequential experiments have been performed using Exp.Open [5] with Generator, both part of CADP. These are highly optimised for sequential use. Those experiments were performed on a machine with an Intel Xeon E5520 2.27 GHz CPU, 1TB RAM, running Fedora 12. The GPU experiments were done on machines running CentOS Linux, with a Kepler K20 GPU, an Intel E5-2620 2.0 GHz CPU, and 64 GB RAM. The GPU has 13 SMs, 6GB global memory (realising a hash table with about 1.3 billion slots), and 48kB (12,288 integers) shared memory per block. We chose not to compare with the GPU tool of [7], since it is a CPU-GPU hybrid, and therefore does not clearly allow to study to what extent a GPU can be used by itself for exploration. Furthermore, it uses bitstate hashing, thereby not guaranteeing exhaustiveness.

We also conducted experiments with the model checker LTSmin [24] using the six CPU cores of the machines equipped with K20s. LTSmin uses the most scalable multi-core exploration techniques currently available.

Table 1 displays the characteristics of the models we consider here. The first five are models taken from and inspired by those distributed with the mCRL2 toolset (in general '.1' suffixed models indicate that we extended the existing

[2] The implementation and experimental data is available at `http://www.win.tue.nl/~awijs/GPUexplore`.

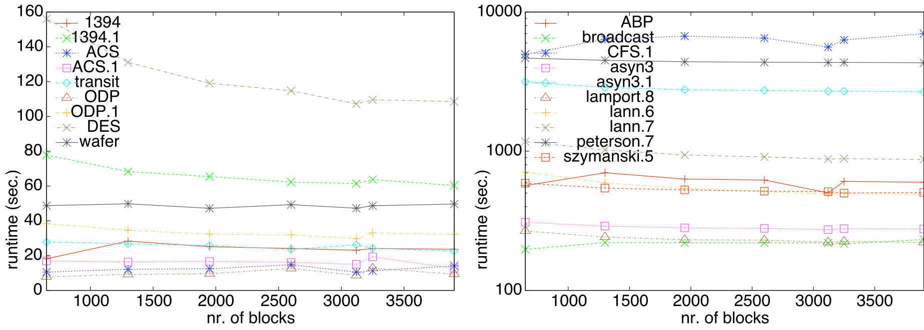

Fig. 5. Performance with varying nr. of blocks (iters=10)

models to obtain larger state spaces), the next two have been created by us, the seven after that originate from CADP, and the final five come from the BEEM database. The latter ones have first been translated manually to mCRL2, since our input, network of LTSs, uses an action-based representation of system behaviour, but BEEM models are state-based, hence this gap needs to be bridged.

Table 1. Benchmark characteristics

Model	#States	#Transitions
1394	198,692	355,338
1394.1	36,855,184	96,553,318
acs	4,764	14,760
acs.1	200,317	895,004
wafer stepper.1	4,232,299	19,028,708
ABP	235,754,220	945,684,122
broadcast	60,466,176	705,438,720
transit	3,763,192	39,925,524
CFS.1	252,101,742	1,367,483,201
asyn3	15,688,570	86,458,183
asyn3.1	190,208,728	876,008,628
ODP	91,394	641,226
ODP.1	7,699,456	31,091,554
DES	64,498,297	518,438,860
lamport.8	62,669,317	304,202,665
lann.6	144,151,629	648,779,852
lann.7	160,025,986	944,322,648
peterson.7	142,471,098	626,952,200
szymanski.5	79,518,740	922,428,824

An important question is how the exploration should be configured, i.e. how many blocks should be launched, and how many iterations should be done per kernel launch. We tested different configurations for 512 threads per block (other numbers of threads resulted in reduced performance) using double hashing with forwarding; Figure 5 shows our results launching a varying number of blocks (note the logscale of the right graph), each performing 10 iterations per kernel launch. The ideal number of blocks for the K20 seems to be 240 per SM, i.e. 3120 blocks. For GPU standards, this is small, but launching more often negatively affects performance, probably due to the heavy use of shared memory.

Figure 6 shows some of our results on varying the number of iterations per kernel launch. Here, it is less clear which value leads to the best results, either 5 or 10 seems to be the best choice. With a lower number, the more frequent hash table scanning becomes noticable, while with higher numbers, the less frequent passing along of work from SMs to each other leads to too much redundancy, i.e. re-exploration of states, causing the exploration to take more time.

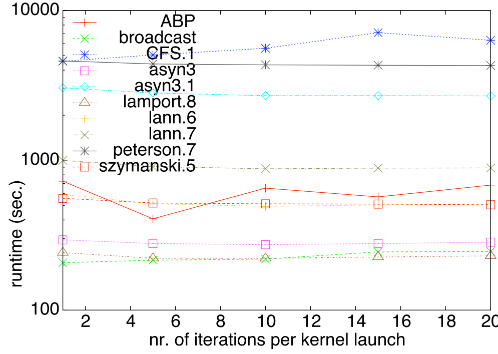

Fig. 6. Performance with varying nr. of iterations per kernel (blocks=3120)

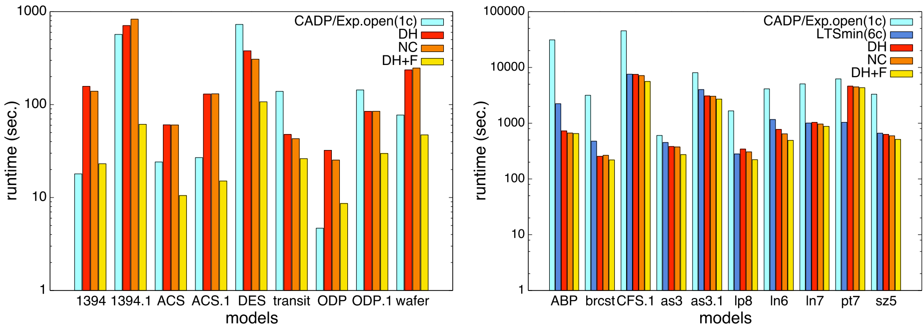

Fig. 7. Runtime results for various tools

For further experimentation, we opted for 10 iterations per launch. Figure 7 shows our runtime results (note the log scale). The GPU extension combinations used are Double Hashing (DH), DH+Forwarding (DH+F), and DH without local caches (NC). The smaller state spaces are represented in the left graph. Here, DH and NC often do not yet help to speed up exploration; the overhead involved can lead to longer runtimes compared to sequential runs. However, DH+F is more often than not faster than sequential exploration. The small differences between DH and NC, and the big ones between NC and DH+F (which is also the case in the right graph) indicate that the major contribution of the caches is forwarding, as opposed to localised duplicate detection, which was the original motivation for using them. DH+F speeds up DH on average by 42%.

It should be noted that for vectors requiring multiple integers, GPU exploration tends to perform on average 2% redundant work, i.e. some states are re-explored. In those cases, data races occur between threads writing and reading vectors, since only the first integer of a vector is written with a CAS. However, we consider these races benign, since it is important that all states are explored, not how many times, and adding additional locks hurts the performance.

The right graph in Figure 7 includes results for LTSmin using six CPU cores. This shows that, apart from some exceptions, our GPU implementation on

average has a performance similar to using about 10 cores with LTSmin, based on the fact that LTSmin demonstrates near-linear speedups when the number of cores is increased. In case of the exceptions, such as the ABP case, about two orders of magnitude speedup is achieved. This may seem disappointing, considering that GPUs have an enormous computation potential. However, on-the-fly exploration is not a straightforward task for a GPU, and a one order of magnitude speedup seems reasonable. Still, we believe these results are very promising, and merit further study. Existing multi-core exploration techniques, such as in [24], scale well with the number of cores. Unfortunately, we cannot test whether this holds for our GPU exploration, apart from varying the number of blocks; the number of SMs cannot be varied, and any number beyond 15 on a GPU is not yet available.

Concluding, our choices regarding data encoding and successor generation seem to be effective, and our findings regarding a new GPU hash table, local caches and forwarding can be useful for anyone interested in GPU graph exploration.

5 Conclusions

We presented an implementation of on-the-fly GPU state space exploration, proposed a novel GPU hash table, and experimentally compared different configurations and combinations of extensions. Compared to state-of-the-art sequential implementations, we measured speedups of one to two orders of magnitude. We think that GPUs are a viable option for state space exploration. Of course, more work needs to be done in order to really use GPUs to do model checking. For future work, we will experiment with changing the number of iterations per kernel launch during a search, support LTS networks with data, pursue checking safety properties, and experiment with partial searches [25, 26].

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
2. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel Probabilistic Model Checking on General Purpose Graphics Processors. STTT 13(1), 21–35 (2011)
3. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In: Joint HiBi/PDMC Workshop (HiBi/PDMC 2010), pp. 17–19. IEEE (2010)
4. Wijs, A.J., Bošnački, D.: Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 98–116. Springer, Heidelberg (2012)
5. Lang, F.: Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 70–88. Springer, Heidelberg (2005)
6. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)

7. Edelkamp, S., Sulewski, D.: Efficient Explicit-State Model Checking on General Purpose Graphics Processors. In: van de Pol, J., Weber, M. (eds.) SPIN 2000. LNCS, vol. 6349, pp. 106–123. Springer, Heidelberg (2010)
8. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing Fast LTL Model Checking Algorithms for Many-Core GPUs. J. Par. Distr. Comput. 72, 1083–1097 (2012)
9. Deng, Y., Wang, B., Shuai, M.: Taming Irregular EDA Applications on GPUs. In: ICCAD 2009, pp. 539–546 (2009)
10. Harish, P., Narayanan, P.J.: Accelerating Large Graph Algorithms on the GPU Using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007)
11. Hong, S., Kim, S., Oguntebi, T., Olukotun, K.: Accelerating CUDA Graph Algorithms At Maximum Warp. In: PPoPP 2011, pp. 267–276. ACM (2011)
12. Luo, L., Wong, M., Hwu, W.M.: An Effective GPU Implementation of Breadth-First Search. In: DAC 2010, pp. 52–55. IEEE Computer Society Press (2010)
13. Merrill, D., Garland, M., Grimshaw, A.: Scalable GPU Graph Traversal. In: PPoPP 2012, pp. 117–128. ACM (2012)
14. Alcantara, D., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owens, J., Amenta, N.: Real-time Parallel Hashing on the GPU. ACM Trans. Graph. 28(5), 154 (2009)
15. Alcantara, D., Volkov, V., Sengupta, S., Mitzenmacher, M., Owens, J., Amenta, N.: Building an Efficient Hash Table on the GPU. In: GPU Computing Gems Jade Edition. Morgan Kaufmann (2011)
16. Pagh, R.: Cuckoo Hashing. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 121–133. Springer, Heidelberg (2001)
17. Merrill, D., Grimshaw, A.: High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing. Parallel Processing Letters 21(2), 245–272 (2011)
18. Pelánek, R.: Properties of State Spaces and Their Applications. STTT 10(5), 443–454 (2008)
19. Mateescu, R., Wijs, A.: Hierarchical Adaptive State Space Caching Based on Level Sampling. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 215–229. Springer, Heidelberg (2009)
20. Laarman, A., van de Pol, J., Weber, M.: Boosting Multi-core Reachability Performance with Shared Hash Tables. In: FMCAD 2010, pp. 247–255 (2010)
21. Dietzfelbinger, M., Mitzenmacher, M., Rink, M.: Cuckoo Hashing with Pages. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 615–627. Springer, Heidelberg (2011)
22. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An overview of the mCRL2 Toolset and Its Recent Advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013)
23. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
24. Laarman, A., van de Pol, J., Weber, M.: Multi-Core LTSmin: Marrying Modularity and Scalability. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 506–511. Springer, Heidelberg (2011)
25. Torabi Dashti, M., Wijs, A.J.: Pruning State Spaces with Extended Beam Search. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 543–552. Springer, Heidelberg (2007)
26. Wijs, A.: What To Do Next?: Analysing and Optimising System Behaviour in Time. PhD thesis, VU University Amsterdam (2007)