

On the Scalability of the GPUEXPLORE Explicit-State Model Checker

Nathan Cassee

Thomas Neele

Anton Wijs*

Eindhoven University of Technology
Eindhoven, The Netherlands

N.W.Cassee@student.tue.nl {T.S.Neele, A.J.Wijs}@tue.nl

The use of graphics processors (GPUs) is a promising approach to speed up model checking to such an extent that it becomes feasible to instantly verify software systems during development. GPUEXPLORE is an explicit-state model checker that runs all its computations on the GPU. Over the years it has been extended with various techniques, and the possibilities to further improve its performance have been continuously investigated. In this paper, we discuss how the hash table of the tool works, which is at the heart of its functionality. We propose an alteration of the hash table that in isolated experiments seems promising, and analyse its effect when integrated in the tool. Furthermore, we investigate the current scalability of GPUEXPLORE, by experimenting both with input models of varying sizes and running the tool on one of the latest GPUs of NVIDIA.

1 Introduction

Model checking [2] is a technique to systematically determine whether a concurrent system adheres to desirable functional properties. There are numerous examples in which it has been successfully applied, however, the fact that it is computationally very demanding means that it is not yet a commonly used procedure in software engineering. Accelerating these computations with graphics processing units (GPUs) is one promising way to model check a system design in mere seconds or minutes as opposed to many hours.

GPUEXPLORE [28, 29, 33] is a model checker that performs all its computations on a GPU. Initially, it consisted of a state space exploration engine [28], which was extended to perform on-the-fly deadlock and safety checking [29]. Checking liveness properties has also been investigated [27], with positive results, but liveness checking has yet to be integrated in the official release of the tool. Finally, in order to reduce the memory requirements of GPUEXPLORE, partial order reduction has been successfully integrated [20].

Since the first results achieved with GPUEXPLORE [28], considerable progress has been made. For instance, the original version running on an NVIDIA K20 was able to explore the state space of the *peterson7* model in approximately 72 minutes. With many improvements to GPUEXPLORE’s algorithms, reported in [33], the GPU hardware and the CUDA compiler, this has been reduced to 16 seconds. With these levels of speed-up, it has become much more feasible to interactively check and debug large models. Furthermore, GPU developments continue and many options can still be investigated.

Performance is very important for a tool such as GPUEXPLORE. However, so far, the scalability of the tool has not yet been thoroughly investigated. For instance, currently, we have access to a NVIDIA Titan X GPU, which is equipped with 12 GB global memory, but for all the models we have been using

*We gratefully acknowledge the support of NVIDIA Corporation with the donation of the GeForce Titan X used for this research.

so far, 5 GB of global memory suffices as the models used for the run-time analysis of GPUEXPLORE do not require more than 5GB for a state space exploration. In the current paper, we report on results we have obtained when scaling up models to utilise up to 12 GB.

In addition, we also experimentally compared running GPUEXPLORE on a Titan X GPU with the Maxwell architecture, which was released on 2015, with GPUEXPLORE running on a Titan X GPU with the Pascal architecture, which was released a year later. This provides insights regarding the effect recent hardware developments have on the tool.

Finally, we analyse the scalability of a critical part of the tool, namely its *hash table*. This structure is used during model checking to keep track of the progress made through the system state space. Even a small improvement of the hash table design may result in a drastic improvement in the performance of GPUEXPLORE. Recently, we identified, by conducting isolated experiments, that there is still potential for further improvement [9]. In the current paper, we particularly investigate whether changing the size of the so-called *buckets*, i.e., data structures that act as containers in which states are stored, can have a positive effect on the running time.

The structure of the paper is as follows. In Section 2, we discuss related work. Next, an overview of the inner working of GPUEXPLORE is presented in Section 3. The hash table and its proposed alterations are discussed in Section 4. After that, we present the results we obtained through experimentation in Section 5. Finally, conclusions and pointers to future work are given in Section 6.

2 Related work

In the literature, several different designs for parallel hash tables can be found. First of all, there is the hash table for GPUs proposed by Alcantara *et al.* [1], which is based on Cuckoo hashing. Secondly, Laarman *et al.* [24] designed a hash table for multi-core shared memory systems. Their implementation was later used as a basis for the hash table underlying the LTSMIN model checker. Other lock-free hash tables for the GPU are those proposed by Moazeni & Sarrafzadeh [19], by Bordawekar [6] and by Misra & Chaudhuri [18]. Cuckoo hashing as implemented by Alcantara *et al.* is publicly available as part of the CUDPP library¹. Unfortunately, to the best of our knowledge, there are no implementations available of the other hash table designs.

Besides GPUEXPLORE [33], there are several other GPU model checking tools. Bartocci *et al.* [5] developed an extension for the SPIN model checker that performs state-space exploration on the GPU. They achieved significant speed-ups for large models.

A GPU extension to the parallel model checking tool DIVINE, called DIVINE-CUDA, was developed by Barnat *et al.* [4]. To speed-up the model checking process, they offload the cycle detection procedure to the GPU. Their tool can even benefit from the use of multiple GPUs. DIVINE-CUDA achieves a significant speed-up when model checking properties that are valid.

Edelkamp and Sulewski address the issues arising from the limited amount of global memory available on a GPU. In [12], they implement a hybrid approach in a tool called CUDEMoC, using the GPU for next state computation, while keeping the hash table in the main memory, to be accessed by multiple threads running on the Central Processing Unit (CPU). In [13], they keep part of the state space in the global memory and store the rest on disk. The record on disk can be queried through a process they call *delayed duplicate detection*. Even though disk access causes overhead, they manage to achieve a speed-up over single-threaded tools.

¹<http://cudpp.github.io/>

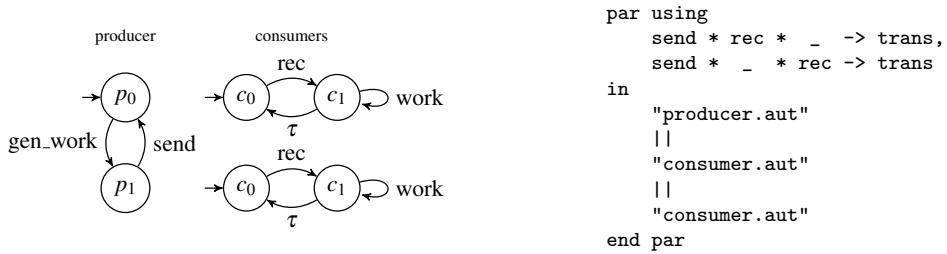


Figure 1: Example of LTS network with one producer and two consumers. On the right, the communication between the LTSs is specified using the EXP syntax [14]. Here, `producer.aut` and `consumer.aut` are files containing the specification of the producer and the consumer respectively.

Edelkamp *et al.* [7, 8] also applied GPU programming to probabilistic model checking. They solve systems of linear equations on the GPU in order to accelerate the value iteration procedure. GPUs are well suited for this purpose, and can enable a speed-up of 18 times over a traditional CPU implementation.

Wu *et al.* [35] have developed a tool called GPURC that performs the full state-space exploration process on the GPU, similar to GPUEXPLORE. Their implementation applies *dynamic parallelism*, a relatively new feature in CUDA that allows launching of new kernels from within a running kernel. Their tool shows a good speed-up compared to traditional single-threaded tools, although the added benefit of dynamic parallelism is minimal.

Finally, GPUs are also successfully applied to accelerate other computations related to model checking. For instance, Wu *et al.* [34] use the GPU to construct counter-examples, and state space decomposition and minimisation are investigated in [26, 31, 32]. For probabilistic model checking, Češka *et al.* [10] implemented GPU accelerated parameter synthesis for parameterized continuous time Markov chains.

3 GPUs and GPUEXPLORE

GPUEXPLORE [28, 29, 33] is an explicit-state model checker that practically runs entirely on a GPU (only the general progress is checked on the host side, i.e. by a thread running on the CPU). It is written in CUDA C, an extension of C offered by NVIDIA. CUDA (Compute Unified Device Architecture) provides an interface to write applications for NVIDIA’s GPUs. GPUEXPLORE takes a *network of Labelled Transition Systems* (LTSs) [17] as input, and can construct the synchronous product of those LTSs using many threads in a Breadth-First-Search-based exploration, while optionally checking on-the-fly for the presence of deadlocks and violations of safety properties. A (negation of a) safety property can be added as an automaton to the network.

An LTS is a directed graph in which the nodes represent states and the edges are transitions between the states. Each transition has an action label representing an event leading from one state to another. An example network is shown in Figure 1, where the initial states are indicated by detached incoming arrows. One producer generates work and sends it to one of two consumers. This happens by means of synchronisation of the ‘send’ and ‘rec’ actions. The other actions can be executed independently. How the process LTSs should be combined using the relevant synchronisation rules is defined on the right in Figure 1, using the syntax of the EXP.OPEN tool [17]. The state space of this network consists of 8 states and 24 transitions.

The general approach of GPUEXPLORE to perform state space exploration is discussed in this sec-

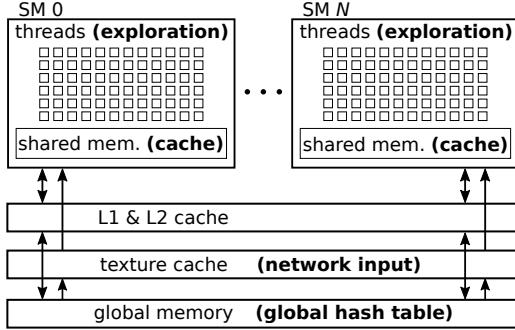


Figure 2: Schematic overview of the GPU hardware architecture and GPUEXPLORE

tion, leaving out many of the details that are not relevant for understanding the current work. The interested reader is referred to [28, 29, 33].

In a CUDA program, the host launches CUDA functions called *kernels*, that are to be executed many times in parallel by a specified number of GPU threads. Usually, all threads run the same kernel using different parts of the input data, although some GPUs allow multiple different kernels to be executed simultaneously (GPUEXPLORE does not use this feature). Each thread is executed by a streaming processor (SP). Threads are grouped in *blocks* of a predefined size. Each block is assigned to a streaming multiprocessor (SM). An SM consists of a fixed number of SPs (see Figure 2).

Each thread has a number of on-chip registers that allow fast access. The threads in a block together share memory to exchange data, which is located in the (on-chip) *shared memory* of an SM. Finally, the blocks can share data using the *global memory* of the GPU, which is relatively large, but slow, since it is off-chip. The global memory is used to exchange data between the host and the kernel. The GTX TITAN X, which we used for our experiments, has 12 GB global memory and 24 SMs, each having 128 SPs (3,072 SPs in total).

Writing well-performing GPU applications is challenging, due to the execution model of GPUs, which is *Single Instruction Multiple Threads*. Threads are partitioned in groups of 32 called *warps*. The threads in a warp run in lock-step, sharing a program counter, so they always execute the same program instruction. Hence, thread divergence, i.e. the phenomenon of threads being forced to execute different instructions (e.g., due to if-then-else constructions) or to access physically distant parts of the global memory, negatively affects performance.

Model checking tends to introduce divergences frequently, as it requires combining the behaviour of the processes in the network, and accessing and storing state vectors of the system state space in the global memory. In GPUEXPLORE, this is mitigated by combining relevant network information as much as possible in 32-bit integers, and storing these as textures, that only allow read access and use a dedicated cache to speed up random accesses.

Furthermore, in the global memory, a hash table is used to store state vectors (Figure 2). The currently used hash table has been designed to optimise accesses of entire warps: the space is partitioned into buckets consisting of 32 integers, precisely enough for one warp to fetch a bucket with one combined memory access. State vectors are hashed to buckets, and placed within a bucket in an available slot. If the bucket is full, another hash function is used to find a new bucket. Each block accesses the global hash table to collect vectors that still require exploration.

To each state vector with n process states, a *group* of n threads is assigned to construct its successors using fine-grained parallelism. Since access to the global memory is slow, each block uses a dedicated

state cache (Figure 2). It serves to store and collect newly produced state vectors, that are subsequently moved to the global hash table in batches. With the cache, block-local duplicates can be detected.

4 The GPUEXPLORE Hash Table

States discovered during the search exploration phase of GPUEXPLORE are inserted into a global memory hash table. This hash table is used to keep track of the open and closed sets maintained during the breadth first search based exploration of the state space.

Since many accesses (reads and writes) to the hash table are performed during state-space exploration, its performance is critical for our model checker. In order to allow for efficient parallel access, the hash table should be lock-free. To prevent corruption of state vectors, insertion should be an atomic operation, even when a state vector spans multiple 32 bit integers.

Given these requirements, we have considered several lock-free hash table implementations. One of them, proposed by Alcantara *et al.* [1], uses so-called *Cuckoo hashing*.

With Cuckoo hashing a key is hashed to a bucket in the hash table, and in case of a collision the key that is already in the bucket is evicted and rehashed using another hash function to a different bucket. Re-insertions continue until the last evicted key is hashed to an empty bucket, until all hash functions are exhausted or until the chain of re-insertions becomes too long [22].

The other hash table we considered is the one originally designed for GPUEXPLORE [33]; we refer to its hashing mechanism as GPUEXPLORE hashing. We experimentally compared these two hash tables, and from these comparisons we concluded that while Cuckoo hashing on average performs better, it does not meet all the demands needed by GPUEXPLORE. However, based on the performance evaluation a possible performance increase has been identified for GPUEXPLORE hashing [9]. This section discusses the proposed modification, and its implementation.

4.1 GPUEXPLORE Hashing

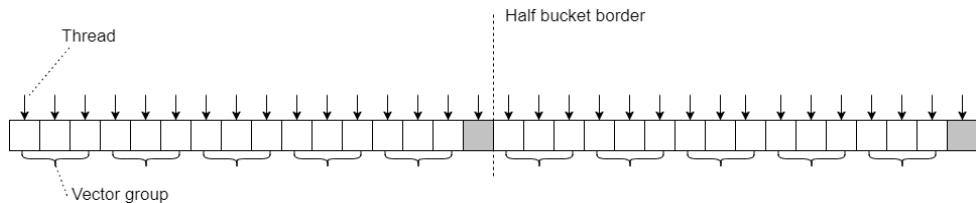


Figure 3: Division of threads in a warp over a bucket of size 32 in GPUEXPLORE 2.0

The GPUEXPLORE hash table consists of two parts: the storage used to store the discovered vectors and the set of hash constants used by the hash function to determine the position of a vector in the hash table. The memory used to store vectors is divided into buckets with a size of 32 integers. Each bucket is additionally split into two equally sized half buckets. Therefore a single bucket can store $2 * \left\lfloor \frac{16}{\text{vector_length}} \right\rfloor$ vectors. The reason for writing vectors to buckets with half-warps (a group of 16 threads) is that in many cases, atomic writes of half-warps are scheduled in an uninterrupted sequence [33]. This results in vectors consisting of multiple integers to be written without other write operations corrupting them. It should be noted that the GPUEXPLORE hash table uses *closed hashing*, i.e., the vectors themselves are stored in the hash table, as opposed to pointers to those vectors.

When inserting, a warp of 32 threads inserts a single vector into a bucket. The way threads are divided over a bucket can be observed in Figure 3, this figure visualizes a single bucket of the GPUEXPLORE hash table for a vector length of 3. Each thread in a warp is assigned to one integer of the bucket, and to one integer of the vector. This assignment is done from left to right per half bucket. For this example the first 3 threads, i.e., the first vector group, is assigned to the first slot in the bucket, and the first thread in a vector group is assigned to the first integer of the vector. By assigning every thread to a single part of the vector and of the bucket each thread has a relatively simple task, which can be executed in parallel.

The insertion algorithm first hashes the vector to insert to determine the bucket belonging to the vector. Each thread checks its place in the bucket and compares the integer on that position to the corresponding integer of the vector. After comparing, the insertion algorithm then uses CUDA warp instructions to quickly exchange data between all 32 threads in the warp to determine whether a single vector group of threads has found the vector. If the vector has been found the insertion algorithm terminates, if the vector has not been found the algorithm continues.

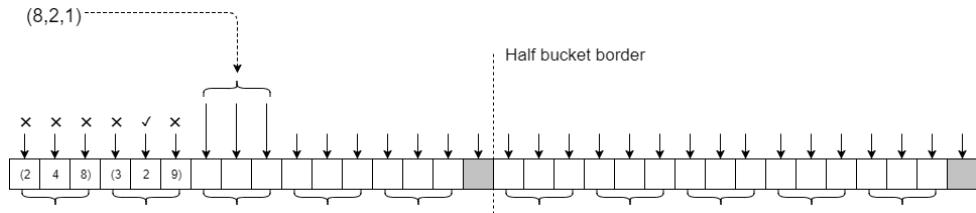


Figure 4: Example of inserting the vector $(8, 2, 1)$ into the GPUEXPLORE 2.0 hash table.

Figure 4 shows an example of the vector $(8, 2, 1)$ being inserted into the GPUEXPLORE hash table where the first two slots are already occupied. Because the first two slots are already occupied the first six threads compare their element of the vector to their element of the bucket. The icons above the arrows indicate the result of these comparisons. As can be seen there is one thread that has a match, however, because not all elements in the slot match the insertion algorithm does not report a find.

If the vector is not found in the bucket, the insertion algorithm selects the first free slot in the bucket, in this case the third slot. This selection procedure can be done efficiently using CUDA warp instructions. Next, the associated threads attempt to insert the vector $(8, 2, 1)$ into the selected slot using a compare and swap operation. If the insertion fails because another warp had claimed that slot already for another vector, the algorithm takes the next free slot in the bucket.

If a bucket has no more free slots the next set of hash constants is used, and the next bucket is probed. This is repeated until all hash constants have been used, and if no insertion succeeds into any of the buckets, the insertion algorithm reports a failure and the exploration stops.

In GPUEXPLORE 2.0 the hash table is initialized using eight hash functions. After each exploration step, blocks that have found new vectors use the insertion algorithm to insert any new vectors they found into the global hash table. The hash table is therefore a vital part of GPUEXPLORE.

Buckets with a length of 32 integers have been chosen because of the fact that warps in CUDA consist of 32 threads. This way, every integer in a bucket can be assigned to a single thread. Besides, this design choice also allows for coalesced memory access: when a warp accesses a continuous block of 32 integers, this operation can be executed in a single memory fetch operation [21]. Uncoalesced accesses, on the other hand, have to be done individually after each other. By coalescing the accesses, the available bandwidth to global memory is used efficiently.

4.2 Configurable Bucket Size

While the current implementation of the hash table makes it possible for GPUEXPLORE to achieve a considerable increase in performance over CPU-based explicit-state model checkers [33], it suffers from one disadvantage. Namely, after initially scanning the bucket, only x threads, where x is the vector length, are active at a time. The other $32 - x$ threads are inactive while they await the result of the atomic insertion of the active group. If the insertion fails but there is still a free slot in the hash table, another group of x threads becomes active to attempt an atomic insertion, while the remaining $32 - x$ threads again await the result of this operation.

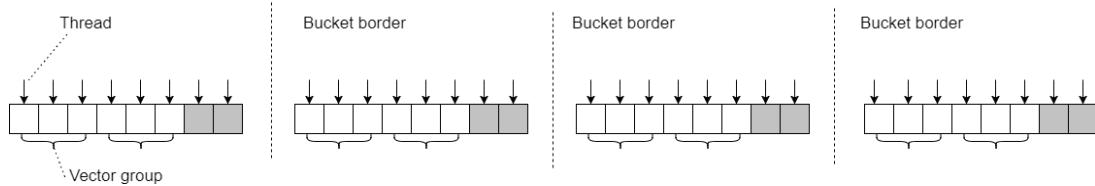


Figure 5: Division of threads in a warp over buckets of size 8

Therefore, for buckets with a lot of free slots where insertions fail the majority of threads in the warp are inactive. Furthermore, the vector size generally does not exceed four integers, which means that when attempting to atomically insert a vector, the majority of threads in a warp is inactive. Ergo, one possible improvement over the GPUEXPLORE hash table is to reduce the bucket size, so that there are fewer slots per bucket, and therefore, fewer threads are needed to insert a single vector. As a single insertion still uses one thread per integer in the bucket, in turn more vectors can be inserted in parallel.

Figure 5 shows what the division of threads over a warp looks like if buckets of size 8 instead of 32 are used. As can be observed, a warp can insert four elements in parallel, as in this diagram each group of 8 threads inserts a different vector and accesses different buckets in global memory.

The logical consequence of this improvement is that after scanning a bucket, fewer threads are inactive while the vector is being inserted into a free slot. If we suppose that the vector size for a certain model is 3, and that the new bucket size is 8, then while inserting a vector using the GPUEXPLORE hash table $32 - 3 = 29$ threads are inactive. On the other hand, if four buckets of size 8 are simultaneously accessed, only $32 - 3 \cdot 4 = 20$ threads are inactive, and four vectors are simultaneously processed, as opposed to only one.

However, while more threads can be active at the same time, smaller buckets also lead to thread divergence within a warp. First of all, of course, accessing different buckets simultaneously likely leads to uncoalesced memory accesses. Furthermore, it is also possible that in an insertion procedure, one group needs to do more work than another in the same warp. For instance, consider that the first group in the warp fails to find its vector in the designated bucket, and also cannot write it to the bucket since the latter is full. In that case, the group needs to fetch another bucket. At the same time, another group in the warp may find its vector in the designated bucket, and therefore be able to stop the insertion procedure. In such a situation, these two groups will diverge, and the second group will have to wait until the first group has finished inserting. This means that the use of smaller buckets can only be advantageous if the performance increase of the smaller buckets outweighs the performance penalty of divergence. In this paper, we address whether this is true or not in practical explorations.

The suggested performance increase has been experimentally validated by comparing an implementation of the hash table with varying bucket size to the original GPUEXPLORE 2.0 hash table, both in

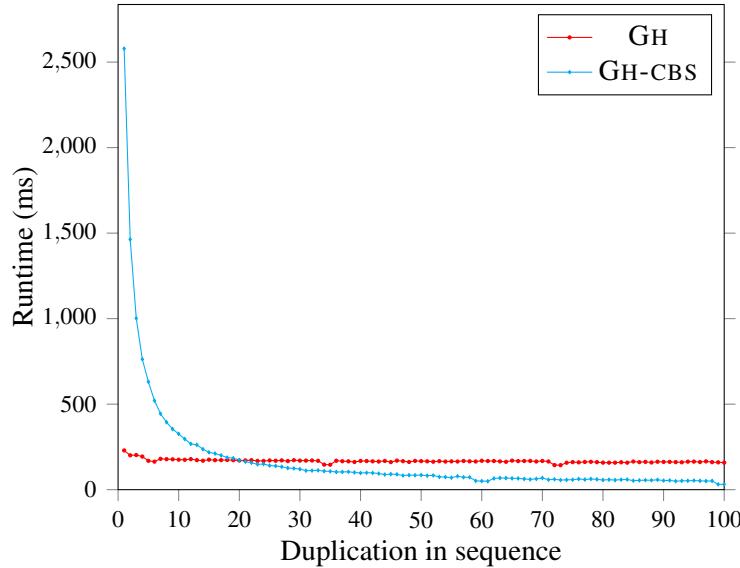


Figure 6: Results of inserting a sequence of 100,000,000 randomly generated integers into the hash tables of GPUEXPLORE (GH) and GPUEXPLORE with configurable buckets (GH-CBS). For the bucketed version a bucket size of four integers has been used.

isolation and as a part of GPUEXPLORE. The results of this comparison are presented and discussed in section 5.

5 Experiments

In this section we discuss our experiments to evaluate the scalability of GPUEXPLORE. In Section 5.1, we report on experiments to evaluate the effect of the bucket size on the runtimes of GPUEXPLORE. In the next two sections, our goal is to determine whether GPUEXPLORE scales well when varying the size of the model and the performance of the hardware, respectively. For most of the experiments, we use an NVIDIA Titan X (Maxwell) installed on a machine running LINUX MINT 17.2. The number of blocks is set to 6,144, with 512 threads per block. The hash table is allocated 5GB in global memory.

For our benchmarks, we selected a varied set of models from the CADP toolset [14], the mCRL2 toolset [11] and the BEEM database [23]. The models with a .1 suffix have been modified to obtain a larger state space.

5.1 Varying the Bucket Size

To test different bucket sizes two types of experiments have been performed. First, the hash table with varying bucket sizes has been tested in isolation, where the time taken to insert 100,000,000 elements has been measured. Second, GPUEXPLORE has been modified such that it uses a hash table with modifiable bucket size. This bucket size can be set at compile time. The performance of this version of GPUEXPLORE has been compared to the original GPUEXPLORE w.r.t. the exploration of several input models.

The input data for the performance evaluation of the hash tables in isolation is a set of sequences of

randomly generated integers, each sequence consisting of 100,000,000 vectors with a length of 1 integer. The sequences vary in how often an element occurs in the sequence. A duplication of 1 means that every unique element occurs once in the sequence, and a duplication of 100 means that each unique element in the sequence occurs 100 times. Therefore a sequence with a duplication of 100 has $\frac{100,000,000}{100}$ unique elements. Note that this experiment tries to replicate state space exploration, where many duplicate insertions are performed when the fan-in is high, i.e., many transitions in the state space lead to the same state. So this experiment replicates the performance of the hash table for models that vary in the average fan-in of the states. For each sequence, we measured the total time required to insert all elements.

The results of this comparison are depicted in Figure 6. We refer to the standard GPUEXPLORE 2.0 hash table as GH and the hash table with configurable bucket size as GH-CBS. In the experiment, GH-CBS with a bucket size of four integers has been compared to GH. GH-CBS is slower for sequences where each element only occurs a few times. However, for sequences with a higher duplication degree GH-CBS starts to outperform GH. After all, for the sequences with more duplication, less time is spent on atomically inserting elements, since most elements are already in the hash table. GH-CBS performs up to three times better than GH. We will see later why the amount of duplication is relevant for model checking.

In addition to testing the performance of the hash tables in isolation, the performance of GH-CBS has been compared with standard GPUEXPLORE hashing as a part of GPUEXPLORE 2.0. The hash table underlying GPUEXPLORE, as implemented by Wijs, Neele and Bošnački [33], has been modified to allow configuration of the bucket size at compile time. We compared the time required for state space exploration of our implementation, GPUEXPLORE with configurable bucket size, with the original implementation of GPUEXPLORE 2.0 [33].

Four bucket sizes have been compared to GPUEXPLORE 2.0, namely bucket sizes 4, 8, 16 and 32 integers. The relative performance with these bucket sizes has been recorded with the performance of GPUEXPLORE 2.0 as baseline. For each model the experiment has been run five times and the average running time of these five runs has been taken.

The result of these comparisons is illustrated in Figure 7. As can be observed the total exploration time of GPUEXPLORE with configurable bucket size is for most models larger than the runtime of GPUEXPLORE 2.0. Only *szymanski5* and *lann7* show a small performance increase for bucket size 4. For the other instances, however, the new hash table causes a slow down of up to 30%.

There are three reasons why the promising performance shown in the previous experiments is not reflected here. First, the increased complexity of the hash table negatively affects *register pressure*, i.e., the number of registers each thread requires to execute the kernel. When the register usage in a kernel increases, the compiler may temporarily place register contents in global memory. This effect is not observed when the hash table is tested in isolation, as in that case, far fewer registers per thread are required. The increase in register pressure is also the reason that GPUEXPLORE with a configurable bucket size set to 32 is slower than GPUEXPLORE with a static bucket size of 32.

Furthermore, smaller bucket sizes result in more thread divergence and more uncoalesced memory accesses when reading from the hash table. Therefore, the available memory bandwidth is used less efficiently, leading to a drop in performance. Apparently, the increased potential for parallel insertions of vectors cannot overcome this drawback.

Lastly, while exploring the state-space, GPUEXPLORE only discovers duplicates if those states have several incoming transitions. On average, the models used for the experiments have a fan-in of 4 to 6, with some exceptions that have a higher fan-in of around 8 to 11. However, from Figure 6 it can be concluded that the hash table in isolation only starts to outperform the static hash table when each element is duplicated 21 times. This partly explains the performance seen in Figure 7.

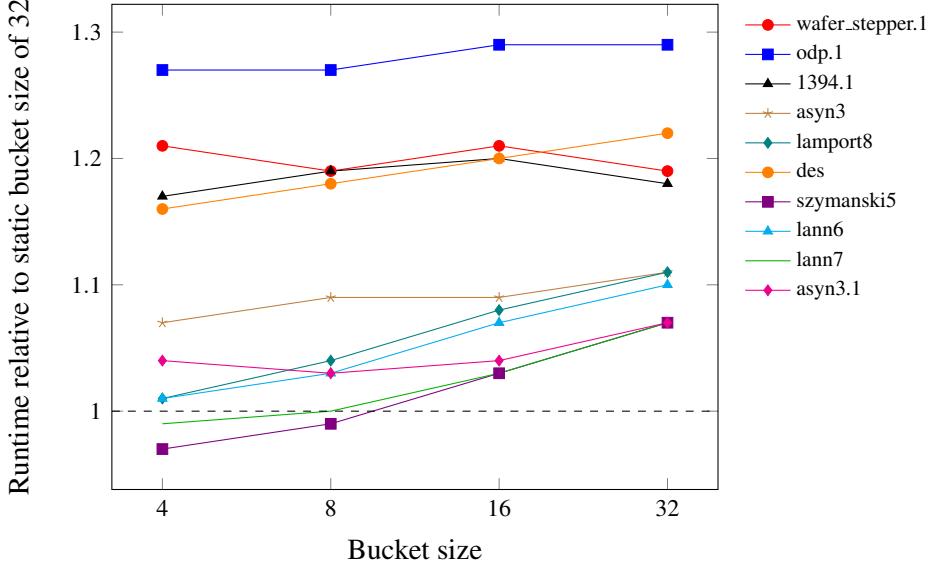


Figure 7: Relative runtime of GPUEXPLORE 2.0 with variable bucket size. The runtime GPUEXPLORE 2.0 with a fixed bucket size of 32 integers is used as a reference and is normalized to 1.

5.2 Varying the Model Size

In addition to experimentally comparing the effect of different bucket sizes, we also investigated how GPUEXPLORE 2.0 behaves when exploring state spaces of different size. We performed this experiment with two different models. The first is a version of the Gas Station model [15] where the number of pumps is fixed to two. We varied the number of customers between two and twelve. None of these instances requires a state vector longer than two 32-bit integers.

The other model is a simple implementation of a ring-structured network, where one token is continuously passed forward between the nodes. Each node has two transitions to communicate with its neighbours and a further three internal transitions. Here, we varied the number of nodes in the network.

We executed the tool five times on each instance and computed the average runtime. The results are listed in Table 1. For the smallest instances, the performance of GPUEXPLORE (measured in states/sec) is a lot worse compared to the larger instances. This has two main reasons. First of all, the relative overhead suffered from initialization and work scanning is higher. Second, the parallelism offered by the GPU cannot be fully exploited, because the size of one search layer is too small to occupy all blocks with work.

For the gas station model, peak performance is achieved for the instance with ten customers, which has 60 million states. For larger instances, the performance decreases slightly due to the increasing occupancy of the hash table. This leads to more hash collisions, therefore more time is lost on rehashing.

The results of the token ring model show another interesting scalability aspect. There is a performance drop between the instances with 10 nodes and 11 nodes. This is caused by the fact that the instance with 11 nodes is the smallest for which the state vector exceeds 32 bits in length. Longer state vectors lead to more memory accesses throughout the state-space generation algorithm.

Table 1: Performance of GPUEXPLORE for the Gas Station and the Token Ring model while varying the amount of processes.

Gas Station				Token Ring			
N	states	time (s)	states/sec	N	states	time (s)	states/sec
2	165	0.016	10,294	2	12	0.027	449
3	1,197	0.023	51,004	3	54	0.047	1,138
4	7,209	0.035	206,812	4	216	0.067	3,212
5	38,313	0.062	621,989	5	810	0.086	9,402
6	186,381	0.209	892,039	6	2,916	0.113	25,702
7	849,285	0.718	1,183,246	7	10,206	0.180	56,571
8	3,680,721	1.235	2,981,535	8	34,992	0.275	127,142
9	15,333,057	3.093	4,957,437	9	118,098	0.488	242,225
10	61,863,669	11.229	5,509,307	10	393,660	1.087	362,294
11	243,104,733	44.534	5,458,810	11	1,299,078	6.394	203,159
12	934,450,425	178.817	5,225,726	12	4,251,528	8.345	509,462
				13	13,817,466	8.138	1,697,864
				14	44,641,044	22.060	2,023,649
				15	143,489,070	68.233	2,102,934
				16	459,165,024	215.889	2,126,853

5.3 Speed-up Due to the Pascal Architecture

The Titan X we used for most of the benchmarks is based on the Maxwell architecture, and was launched in 2015. Since then, NVIDIA has released several other high-end GPUs. Most aspects have been improved: the architecture has been revised, there are more CUDA cores on the GPU and there is more global memory available. To investigate how well GPUEXPLORE scales with faster hardware, we performed several experiments with a Titan X with the Maxwell architecture (TXM) and with a Titan X with the Pascal architecture (TXP). The latter was released in 2016, and the one we used is installed in the DAS-5 cluster [3] on a node running CENTOS LINUX 7.2.

The TXM experiments were performed with 6,144 blocks, while for the TXP, GPUEXPLORE was set to use 14,336 blocks. The improvements of the hardware allows for GPUEXPLORE to launch more blocks. To evaluate the speed-ups compared to a single-core CPU approach, we also conducted experiments with the GENERATOR tool of the latest version (2017-i) of the CADP toolbox [14]. These have been performed on nodes of the DAS-5 cluster, which are equipped with an INTEL HASWELL E5-2630-v3 2.4 GHz CPU, 64 GB memory, and CENTOS LINUX 7.2.

The results are listed in Table 2. The reported runtimes are averages after having run the corresponding tool ten times. For most of the larger models, we see a speed-up of about 2 times when running GPUEXPLORE on a TXP compared to running it on a TXM. The average speed-up is 1.73. This indicates that GPUEXPLORE scales well with a higher memory bandwidth and a larger amount of CUDA cores.

Comparing GPUEXPLORE on a TXP with single-core CPU analysis, the average speed-up is 183.91, and if we only take state spaces into account consisting of at least 10 million states, the average speed-up is 280.81. Considering that with multi-core model checking, linear speed-ups can be achieved [16], this roughly amounts to using 180 and 280 CPU cores, respectively. This, in combination with the

Table 2: Performance comparison of single-core GENERATOR of CADP (GEN) and GPUEXPLORE running on a Titan X with the Maxwell architecture (TXM) and with the Pascal architecture (TXP).

model	states	runtime (seconds)			speed-ups	
		GEN	TXM	TXP	GEN-TXP	TXM-TXP
acs	4,764	4.17	0.05	0.06	71.91	0.89
odp	91,394	3.26	0.08	0.05	70.76	1.64
1394	198,692	2.81	0.20	0.15	18.95	1.36
acs.1	200,317	5.30	0.18	0.14	37.88	1.27
transit	3,763,192	34.36	0.77	0.48	70.99	1.59
wafer stepper.1	3,772,753	22.95	1.01	0.51	45.17	2.00
odp.1	7,699,456	65.50	1.34	0.66	99.54	2.03
1394.1	10,138,812	82.71	1.42	0.83	99.66	1.71
asyn3	15,688,570	358.58	3.15	1.98	181.47	1.59
lamport8	62,669,317	1048.13	5.81	3.11	336.80	1.87
des	64,498,297	477.43	12.34	6.65	71.84	1.86
szymanski5	79,518,740	1516.71	7.48	3.90	389.10	1.92
peterson7	142,471,098	3741.87	31.60	15.74	237.81	2.01
lann6	144,151,629	2751.15	10.57	5.39	510.80	1.96
lann7	160,025,986	3396.19	16.67	8.41	403.92	1.98
asyn3.1	190,208,728	4546.84	31.03	15.37	295.92	2.02
		average		183.91	1.73	

observation that frequently, speed-ups over 300 times and once even over 500 times are achieved, clearly demonstrates the effectiveness of using GPUs for explicit-state model checking.

6 Conclusion and Future Work

In this paper, we have reported on a number of scalability experiments we conducted with the GPU explicit-state model checker GPUEXPLORE. In earlier work, we identified potential to further improve its hash table [9]. However, experiments in which we varied the bucket size in GPUEXPLORE provided the insight that only for very specific input models, and only if the bucket size is set very small (4), some speed-up becomes noticeable. In the context of the entire computation of GPUEXPLORE, the additional register use per thread and the introduced uncoalesced memory accesses and thread divergence make it not worthwhile to make the bucket size configurable. This may be different for other applications, as our experiments with the hash table in isolation point out that hashing can be made more efficient in this way.

Besides this, we have also conducted experiments with models of different sizes. We scaled up a Gas Station model and a Token Ring model and obtained very encouraging results; for the second model, GPUEXPLORE can generate up to 2.1 million states per second, and for the first model, at its peak, GPUEXPLORE is able to generate about 5.5 million states per second, exploring a state space of 934.5 million states in under three minutes. We believe these are very impressive numbers that demonstrate the potential of GPU model checking.

Finally, we reported on some experiments we conducted with new GPU hardware. The Titan X

with the Pascal architecture from 2016 provides for our benchmark set of models an average speed-up of 1.73 w.r.t. the Titan X with the Maxwell architecture from 2015. We also compared the runtimes of GPUEXPLORE running on the Pascal Titan X with the CPU single-core GENERATOR tool of the CADP toolbox, and measured an average speed-up of 183.91 for the entire benchmark set of models, and of 280.81 for the models yielding a state space of at least 10 million states. Often speed-ups over 300 times have been observed, and in one case even over 500 times.

Future work For future work, we will consider various possible extensions to the tool. First of all, the ability to write explored state spaces to disk will open up the possibility to postprocess and further analyse the state spaces. This could be done directly, or after application of bisimulation reduction on the GPU [26].

Second of all, we will work on making the tool more user friendly. Currently, providing an input model is quite cumbersome, since GPUEXPLORE requires a user to express system behaviour in the low level description formalism of networks of LTSs. Specifying systems would be much more convenient if a higher-level modelling language would be supported. We will investigate which modelling languages would be most suitable for integration in the current tool.

Finally, we will also consider the application of GPUEXPLORE to conduct computations similar to model checking, such as performance analysis [30]. This requires to incorporate time into the input models, for instance by including special actions to represent the passage of time [25].

References

- [1] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens & Nina Amenta (2012): *Building an Efficient Hash Table on the GPU*. In: *GPU Computing Gems Jade Edition*, Morgan Kaufmann Publishers Inc., pp. 39–53, doi:10.1016/B978-0-12-385963-1.00004-6.
- [2] Christel Baier & Joost-Pieter Katoen (2008): *Principles of model checking*. MIT Press.
- [3] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek & Harry Wijshoff (2016): *A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term*. IEEE Computer 49(5), pp. 54–63, doi:10.1109/MC.2016.127.
- [4] Jiří Barnat, Petr Bauch, Luboš Brim & Milan Češka (2012): *Designing Fast LTL Model Checking Algorithms for Many-Core GPUs*. JPDC 72(9), pp. 1083–1097, doi:10.1016/j.jpdc.2011.10.015.
- [5] Ezio Bartocci, Richard DeFrancisco & Scott A. Smolka (2014): *Towards a GPGPU-parallel SPIN Model Checker*. In: *SPIN 2014*, ACM, New York, NY, USA, pp. 87–96, doi:10.1145/2632362.2632379.
- [6] Rajesh Bordawekar (2014): *Evaluation of Parallel Hashing Techniques*. Presentation at GTC’14 (last checked on 17 February 2017). <http://on-demand.gputechconf.com/gtc/2014/presentations/S4507-evaluation-of-parallel-hashing-techniques.pdf>.
- [7] Dragan Bošnački, Stefan Edelkamp, Damian Sulewski & Anton Wijs (2011): *Parallel Probabilistic Model Checking on General Purpose Graphics Processors*. STTT 13(1), pp. 21–35, doi:10.1007/s10009-010-0176-4.
- [8] Dragan Bošnački, Stefan Edelkamp, Damian Sulewski & Anton Wijs (2010): *GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units*. In: *PDMC*, IEEE, pp. 17–19, doi:10.1109/PDMC-HiBi.2010.11.
- [9] Nathan Cassee & Anton Wijs (2017): *Analysing the Performance of GPU Hash Tables for State Space Exploration*. In: *GaM*, EPTCS, Open Publishing Association.

- [10] Milan Češka, Petr Pilař, Nicola Paoletti, Luboš Brim & Marta Kwiatkowska (2016): *PRISM-PSY: Precise GPU-Accelerated Parameter Synthesis for Stochastic Systems*. In: TACAS, LNCS 9636, Springer, pp. 367–384, doi:10.1007/978-3-642-54862-8.
- [11] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. De Vink, Wieger Wesselink & Tim A. C. Willemse (2013): *An Overview of the mCRL2 Toolset and Its Recent Advances*. In: TACAS, LNCS 7795, Springer, pp. 199–213, doi:10.1007/978-3-642-36742-7_15.
- [12] Stefan Edelkamp & Damian Sulewski (2010): *Efficient Explicit-State Model Checking on General Purpose Graphics Processors*. In: SPIN, LNCS 6349, Springer, pp. 106–123, doi:10.1007/978-3-642-16164-3_8.
- [13] Stefan Edelkamp & Damian Sulewski (2010): *External memory breadth-first search with delayed duplicate detection on the GPU*. In: MoChArt, LNCS 6572, Springer, pp. 12–31, doi:10.1007/978-3-642-20674-0_2.
- [14] Hubert Garavel, Frédéric Lang, Radu Mateescu & Wendelin Serwe (2013): *CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes*. STTT 15(2), pp. 89–107, doi:10.1007/978-3-540-73368-3_18.
- [15] David Heimbold & David Luckham (1985): *Debugging Ada Tasking Programs*. IEEE Software 2(2), pp. 47–57, doi:10.1109/MS.1985.230351.
- [16] Alfons Laarman (2014): *Scalable Multi-Core Model Checking*. Ph.D. thesis, University of Twente, doi:10.3990/1.9789036536561.
- [17] Frédéric Lang (2006): *Refined Interfaces for Compositional Verification*. In: FORTE, LNCS 4229, Springer, pp. 159–174, doi:10.1007/11888116_13.
- [18] Prabhakar Misra & Mainak Chaudhuri (2012): *Performance Evaluation of Concurrent Lock-free Data Structures on GPUs*. In: ICPADS, pp. 53–60, doi:10.1109/ICPADS.2012.18.
- [19] Maryam Moazeni & Majid Sarrafzadeh (2012): *Lock-free Hash Table on Graphics Processors*. In: SAAHPC, pp. 133–136, doi:10.1109/SAAHPC.2012.25.
- [20] Thomas Neele, Anton Wijs, Dragan Bošnački & Jaco van de Pol (2016): *Partial Order Reduction for GPU Model Checking*. In: ATVA, LNCS 9938, Springer, pp. 357–374, doi:10.1007/978-3-319-46520-3_23.
- [21] John Nickolls, Ian Buck, Michael Garland & Kevin Skadron (2008): *Scalable Parallel Programming with CUDA*. Queue 6(2), pp. 40–53, doi:10.1145/1365490.1365500.
- [22] Rasmus Pagh & Flemming Friche Rodler (2001): *Cuckoo Hashing*. In: ESA, LNCS 2161, Springer, pp. 121–133, doi:10.1007/3-540-44676-1_10.
- [23] Radek Pelánek (2007): *BEEM: Benchmarks for Explicit Model Checkers*. In: SPIN 2007, LNCS 4595, pp. 263–267, doi:10.1007/978-3-540-73370-6_17.
- [24] Steven van der Vegt & Alfons Laarman (2011): *A Parallel Compact Hash Table*. In: MEMICS, LNCS 7119, Springer, pp. 191–204, doi:10.1007/978-3-642-25929-6_18.
- [25] Anton Wijs (2007): *Achieving Discrete Relative Timing with Untimed Process Algebra*. In: ICECCS, IEEE, pp. 35–44, doi:10.1109/ICECCS.2007.13.
- [26] Anton Wijs (2015): *GPU Accelerated Strong and Branching Bisimilarity Checking*. In: TACAS, LNCS 9035, Springer, pp. 368–383, doi:10.1007/978-3-662-46681-0_29.
- [27] Anton Wijs (2016): *BFS-Based Model Checking of Linear-Time Properties With An Application on GPUs*. In: CAV, Part II, LNCS 9780, Springer, pp. 472–493, doi:10.1007/978-3-319-41540-6_26.
- [28] Anton Wijs & Dragan Bošnački (2014): *GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs*. In: TACAS, LNCS 8413, pp. 233–247, doi:10.1007/978-3-642-54862-8_16.
- [29] Anton Wijs & Dragan Bošnački (2016): *Many-Core On-The-Fly Model Checking of Safety Properties Using GPUs*. STTT 18(2), pp. 169–185, doi:10.1007/s10009-015-0379-9.
- [30] Anton Wijs & Wan Fokkink (2005): *From χ_t to μ CRL: Combining Performance and Functional Analysis*. In: ICECCS, IEEE, pp. 184–193, doi:10.1109/ICECCS.2005.51.

- [31] Anton Wijs, Joost-Pieter Katoen & Dragan Bošnački (2014): *GPU-Based Graph Decomposition into Strongly Connected and Maximal End Components*. In: CAV, LNCS 8559, Springer, pp. 309–325, doi:10.1007/978-3-319-08867-9_20.
- [32] Anton Wijs, Joost-Pieter Katoen & Dragan Bošnački (2016): *Efficient GPU Algorithms for Parallel Decomposition of Graphs into Strongly Connected and Maximal End Components*. Formal Methods in System Design 48(3), pp. 274–300, doi:10.1007/s10703-016-0246-7.
- [33] Anton Wijs, Thomas Neele & Dragan Bošnački (2016): *GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking*. In: FM, LNCS 9995, Springer, pp. 694–701, doi:10.1007/978-3-319-48989-6_42.
- [34] Zhimin Wu, Yang Liu, Yun Liang & Jun Sun (2014): *GPU Accelerated Counterexample Generation in LTL Model Checking*. In: ICFEM, LNCS 8829, Springer, pp. 413–429, doi:10.1007/978-3-319-11737-9_27.
- [35] Zhimin Wu, Yang Liu, Jun Sun, Jianqi Shi & Shengchao Qin (2015): *GPU Accelerated On-the-Fly Reachability Checking*. In: ICECCS 2015, pp. 100–109, doi:10.1109/ICECCS.2015.21.