GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking

Anton Wijs[1⋆], Thomas Neele[1,2], and Dragan Bošnački[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
a.j.wijs@tue.nl
[2] University of Twente, Enschede, The Netherlands

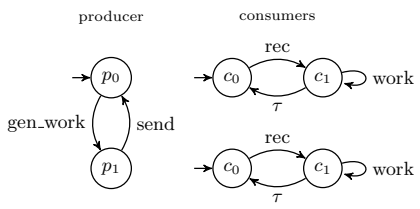
Abstract. In earlier work, we were the first to investigate the potential of using graphics processing units (GPUs) to speed up explicit-state model checking. Back then, the conclusion was clearly that this potential exists, having measured speed-ups of around 10 times, compared to state-of-the-art single-core model checking. In this paper, we present a new version of our GPU model checker, GPUexplore. Since publication of our earlier work, we have identified and implemented several approaches to improve the performance of the model checker considerably. These include enhanced lock-less hashing of the states and improved thread synchronizations. We discuss experimental results that show the impact of both the progress in hardware in the last few years and our proposed optimisations. The new version of GPUexplore running on state-of-the-art hardware can be more than 100 times faster than a sequential implementation for large models and is on average eight times faster than the previous version of the tool running on the same hardware.

1 Introduction

Explicit-state model checking [1,8] is a push-button technique to formally verify the functional correctness of hardware and software models. It is performed by systematically exploring the state space implied by the model. The main drawback of model checking is the state space explosion problem: a linear growth of the model tends to lead to an exponential growth of the state space. Although traditionally, this meant that computer memory was the practical bottleneck, these days, with large amounts of memory at our disposal, scalability of the run time is often hindering our ability to reason about models in a reasonable amount of time.

One way to improve the run time of model checking is by exploiting the computing power of modern parallel architectures. Graphics processing units (GPUs) have a lot of potential in this respect: they can run thousands of threads in parallel and can offer a speed-up of several orders of magnitude. GPUs tend to

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




```
par using
    send * rec *  _  -> trans,
    send *  _  * rec -> trans
in
    "producer.aut"
    ||
    "consumer.aut"
    ||
    "consumer.aut"
end par
```

Fig. 1. Example of LTS network with one producer and two consumers.

have much less memory than modern computer systems, but the current trend is that this amount doubles every few years. Hence, it is interesting to investigate to what extent model checking algorithms can be adapted to run on GPUs. In the last few years, GPUs have been successfully used for several model checking procedures [3,2,4,10,7,16,17].

Our tool GPUexplore [14,15] is the first to take an integrated approach: it runs a complete model checking algorithm on the GPU. Initial results were promising; speed ups of around 10 times were measured. However, at the time, two questions remained unanswered: 1) how does the approach scale over time as new hardware becomes available, and 2) are there still possibilities to further optimise GPUexplore? In this paper, we present new insights concerning both these questions. For the same benchmark set of representative models as used in our previous work [14,15], the new version of GPUexplore executed on the latest GPU hardware achieves an average speed-up of 119 times. With this amount of speed-up, we can finally claim that the leash has truly been taken off GPU model checking.

2 Using The Tool

GPUexplore[1] operates on networks of Labelled Transition Systems (LTSs) [12], which represent interacting parallel processes. 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 can be found 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. The state space of this network consists of 8 states and 24 transitions. Networks are described in the EXP format and LTSs in the AUT format (both from CADP [11]).

Besides reachability analysis, GPUexplore can also check functional properties on-the-fly. Currently, it can check for deadlocks and safety properties. Safety properties are expressed by an automaton included in the input network.

[1] Available at `http://www.win.tue.nl/~awijs/GPUMC`.

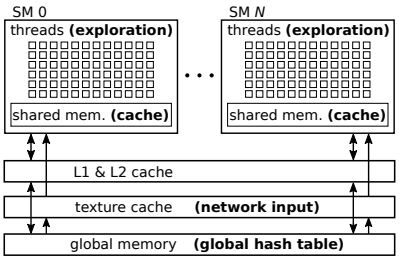


Fig. 2. Schematic overview of the GPU hardware architecture and GPUexplore

3 How GPUexplore Operates

GPU Architecture CUDA[2] is a programming interface developed by NVIDIA to enable general purpose programming on a GPU. It provides a unified view of the GPU ('device'), simplifying the process of developing for multiple devices. Code to be run on the device ('kernel') can be programmed using a subset of C++.

On the hardware level, a GPU has several *streaming multiprocessors* (SM) that contain hundreds of cores. On the programmer side, threads are grouped into *blocks*. The GPU schedules thread blocks on the SMs. One SM can run multiple blocks at the same time, but one block is assigned to a single SM. Internally, blocks are executed as one or more *warps*. A warp is a group of 32 threads that move in lock-step through the program instructions. A *half-warp* is either the first or second half of a warp.

Another important aspect of the GPU architecture is the memory hierarchy. Firstly, each block is allocated *shared memory* that is shared between its threads. The shared memory is placed on-chip, therefore it has a low latency. Secondly, there is the *global memory* that can be accessed by all the threads. It has a high bandwidth, but also a high latency. The amount of global memory is typically multiple gigabytes. There are three caches in between: the L1, L2 and the texture cache. Data in the global memory that is marked as read-only (a 'texture') may be placed in the texture cache. The global memory can be accessed by the CPU ('host'), thus it also serves as an interface between the host and the device. Figure 2 gives a schematic overview of the architecture.

The bandwidth between the SMs and the global memory is used optimally when a continuous block of 32 integers is fetched by a warp. In that case, those 32 memory transactions are performed in parallel. This is called *coalesced* access.

GPUexplore Model checking tends to require many uncoalesced memory accesses, 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

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

memory. In GPUExplore, this is mitigated by combining relevant network information as much as possible in 32-bit integers, and storing these as textures, thereby using the texture cache to speed up random accesses.

State vectors are stored in a number of 32-bit integers. Their total size depends on the number of bits needed for each process in the LTS network. In the global memory, a hash table is used to store state vectors (Fig. 2). The 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.

To each state vector with n process states, a group of n threads (a *vector group*) is assigned to construct its successors using fine-grained parallelism. Each thread collects the relevant transitions of one specific process LTS. Since access to the global memory is slow, each block uses a dedicated state cache (Fig. 2). It serves to collect newly produced state vectors, that are subsequently stored in the global hash table in batches. With the cache, block-local duplicates can be detected. The approach allows to work with vectors that require any number of integers smaller than 32 to be stored.

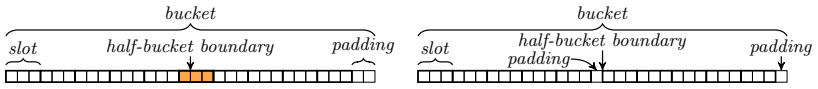
GPUExplore does not maintain a queue of states that need to be explored. Instead, it stores all states in the main hash table, and marks unexplored states. Additionally, a small amount of memory is allocated for each block to store unexplored states. We call this the *work tile*. Whenever a block has no unexplored states in its work tile, it linearly scans through its own slice of the hash table to gather new states for the work tile. Since each block only gathers work from its own slice, an unexplored state cannot be gathered more than once. Although this approach may result in an unbalanced distribution of work during the early and final stages of exploration, experimental evaluation showed that this does not impact runtime significantly.

Recently, we added support for partial-order reduction (POR), based on cluster-based POR [5,6]. This can greatly reduce the amount of memory needed.

4 Improvements

In our initial publications on GPUExplore [14,15], we noted that the mechanism to insert state vectors in the global hash table was prone to producing duplicate entries, leading to additional exploration work and an incorrect report of the number of reachable states. In GPUExplore 2.0, this problem is fixed. Also, since the initial publications, we have identified several performance bottlenecks. In this section, we both explain how the hashing mechanism now works, and discuss those improvements that had the highest impact on the performance.

Improving The Hash Table For correct, lock-less hashing, it is important that element insertion can be done atomically. CUDA provides a number of atomic operations, such as *compare-and-swap*, but only for reading and writing



(a) Layout of a bucket in the old hash table (b) Bucket layout in GPUEXPLORE 2.0

Fig. 3. Example of the layout of a bucket for a state vector length of three integers. Inconsistencies in the old hash table can occur when a slot crosses the half-bucket boundary (marked in orange). In the new situation, the padding is positioned in such a way that no element slot crosses the half-bucket boundary.

individual integers (either 32-bit or 64-bit). This is problematic for GPUEXPLORE in those cases where state vectors require more than 64 bits. This is the main cause for the inaccuracy of the hash table as reported earlier [14,15].

However, we have experimentally determined that whenever a warp is instructed to perform atomic operations on a continuous part of the memory, the GPU scheduler tends to schedule these memory requests in half-warps. For GPUEXPLORE, this means that if two warps execute atomic operations on the same bucket, the memory transactions will never be interleaved on a finer level than half-warps. Therefore, we can avoid data races by preventing state slots from spanning more than one half-bucket. The difference in layout of a bucket for a state vector length of three integers is displayed in Figure 3. As an alternative, we considered the use of spin-locks, but a solution without any form of locking is always more desirable.

Performance Optimisations Since the work scanning approach can incur significant overhead when the hash table is sparsely filled with unexplored states [15], we have implemented several optimisations in this area:

- The original version of GPUEXPLORE already implemented a technique called *work claiming*: once a work tile has been completely processed, i.e. all states in the tile have been explored, new states are copied directly from the cache to the work tile, which both reside in shared memory. This reduces the amount of scanning that needs to be performed. However, once a kernel execution terminates, the contents of the shared memory are lost. To mitigate this, GPUEXPLORE 2.0 temporarily stores the tile in global memory between kernel launches.
- We observed that the work scanning approach is especially inefficient whenever the hash table slice belonging to some block B contains no work. In that case, B would completely scan its slice at the beginning of every search iteration, thereby wasting a lot of time. GPUEXPLORE 2.0 prevents this by keeping track of the presence of unexplored states in each slice. When some block A places an unexplored state in the slice belonging to B, A sets B's work flag to **true**. Once B has gathered all the work in its slice, it sets its own flag to **false**.
- After a block has completed gathering work from some part of its hash table slice, it is not likely that this part will contain much new work once the next

work scanning is started. Therefore, in GPUexplore 2.0, blocks keep track of which part of their hash table slice they last scanned. The next scan can then continue from the position where the previous one finished.

We also performed several optimisations concerning thread synchronisation. We modified the thread hierarchy so that vector groups never span more than one warp. This allows the threads in a group to share information through warp instructions for register swapping. With these changes, we were able to remove all calls to the `__syncthreads` CUDA function from the main loop of the kernel. This greatly improved the speed of successor generation, since now, warps spend less time waiting for each other.

5 Experimental Results

To show the improvements in runtime of GPUexplore 2.0 (without applying partial-order reduction), we performed several experiments. We compared GPUexplore 2.0 with the original version of GPUexplore and with CADP [11]. The sequential experiments were executed on an Intel Xeon E5520 and 1TB of RAM. We used two different GPUs for running GPUexplore: an NVIDIA K20m (13 SMs, 5GB of global memory) and an NVIDIA Titan X (24 SMs, 12GB of global memory). For the original version of GPUexplore, we ran the kernel on 3,120 blocks of 512 threads each, and performed ten iterations, consisting of scanning for work and exploring the states in the resulting work tile, per kernel launch. For GPUexplore 2.0, we ran 6,144 blocks of 512 threads each, and performed only one iteration per kernel launch. The optimal parameters for each version of our tool differ slightly due to the changes we made in the work scanning algorithm.

As benchmarks, we used models from different origins: `odp`, `transit` and `asyn3` are models from the CADP toolkit. The `1394`, `acs` and `wafer stepper` models originate from mCRL2 [9]. The `lamport`, `lann`, `peterson` and `szymanski` models come from the BEEM database[3]. All models are encoded in the EXP format. The models with a .1-suffix are enlarged versions of the original models [15].

For each model and tool, we measured the total runtime (initialisation and state space exploration). For the GPU experiments, we took the average of five runs. The CPU experiments were run a single time. The results can be found in Table 1. The two columns under 'speed-up' indicate the speed-up of GPUexplore 2.0 over CADP and GPUexplore on the Titan X, respectively. GPUexplore 2.0 achieves an average speed-up of 70 times over CADP. It is worth noting that GPUexplore 2.0 can offer more than two orders of magnitude speed-up for larger models, when the parallel potential of the GPU is fully used.

On average, the Titan X, released in 2015, yields a speed-up of 5.5 times compared to the K20m from 2012. The speed-up gained by our optimisations is 7.8 times. This results in a combined speed-up (hardware and software improvements) of 42.6 times relative to our last publication [15]. We remark that

[3] `http://paradise.fi.muni.cz/beem`.

Table 1. Runtimes for CADP, the original GPUexplore and GPUexplore 2.0.

			CADP	GPUexplore		GPUexplore 2.0	speed-up	
model	#states	#transitions	CPU	K20m	Titan X	Titan X	seq.	orig
acs	4,764	14,760	2.25	10.51	2.26	0.33	6.9	6.9
odp	91,394	641,226	2.03	8.63	2.19	0.34	5.9	6.4
1394	198,692	355,338	2.10	23.10	3.85	0.51	4.1	7.6
acs.1	200,317	895,004	3.58	15.06	2.77	0.46	7.8	6.0
transit	3,763,192	39,925,524	37.79	26.20	4.54	1.21	31.3	3.8
wafer_stepper.1	3,772,753	19,028,708	22.25	47.25	7.33	1.42	15.7	5.2
odp.1	7,699,456	31,091,554	76.73	29.78	5.78	1.84	41.8	3.1
1394.1	10,138,812	96,553,318	66.33	61.40	8.44	1.90	34.8	4.4
asyn3	15,688,570	86,458,183	352.56	273.41	37.97	3.87	91.2	9.8
lamport8	62,669,317	304,202,665	944.80	221.80	41.30	6.91	136.7	6.0
des	64,498,297	518,438,860	468.51	107.22	25.42	18.64	25.1	1.4
szymanski5	79,518,740	922,428,824	1393.35	512.13	86.17	8.93	156.0	9.6
peterson7	142,471,098	626,952,200	3463.06	4337.41	1004.07	36.42	95.1	27.6
lann6	144,151,629	648,779,852	2377.73	492.70	94.85	12.52	189.9	7.6
lann7	160,025,986	944,322,648	3035.55	877.74	164.90	19.83	153.1	8.3
asyn3.1	190,208,728	876,008,628	4360.00	2703.61	421.87	36.61	119.1	11.5
average							69.7	7.8

the speed-up measurement for the smaller models are skewed in favour of the original version of GPUexplore, since GPUexplore 2.0 spends most of the time on initialization when the state space is small. When measuring only the time required for exploration, our optimisations result in an average speed-up of 11.5 times.

6 Conclusions

We have presented a new version of the GPU explicit-state model checker GPUexplore, which has been further optimised and supports partial-order reduction. Furthermore, we discussed the impact of both the optimisations and the recent improvements in hardware on the average runtime. For future work, we plan to add support for liveness properties. Recently, this has been investigated [13], but those findings are still to be added to the current stable version.

References

1. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
2. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing fast LTL model checking algorithms for many-core GPUs. Journal of Parallel and Distributed Computing 72(9), 1083–1097 (2012)
3. Barnat, J., Brim, L., Češka, M., Lamr, T.: CUDA Accelerated LTL Model Checking. In: 15th International Conference on Parallel and Distributed Systems. pp. 34–41. IEEE (2009)
4. Bartocci, E., Defrancisco, R., Smolka, S.A.: Towards a GPGPU-Parallel SPIN Model Checker. In: SPIN. pp. 87–96. ACM (2014)
5. Basten, T., Bošnački, D.: Enhancing Partial-Order Reduction via Process Clustering. In: 16th IEEE International Conference on Automated Software Engineering. pp. 245–253 (2001)

6. Basten, T., Bošnački, D., Geilen, M.: Cluster-Based Partial-Order Reduction. ASE 11(4), 365–402 (2004)
7. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel Probabilistic Model Checking on General Purpose Graphics Processors. STTT 13(1), 21–35 (2010)
8. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (2001)
9. 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: TACAS. LNCS, vol. 7795, pp. 199–213. Springer (2013)
10. Edelkamp, S., Sulewski, D.: Efficient Explicit-State Model Checking on General Purpose Graphics Processors. In: SPIN. LNCS, vol. 6349, pp. 106–123. Springer (2010)
11. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. STTT 15(2), 89–107 (2013)
12. Lang, F.: Refined Interfaces for Compositional Verification. In: FORTE. LNCS, vol. 4229, pp. 159–174. Springer (2006)
13. Wijs, A.: BFS-Based Model Checking of Linear-Time Properties With An Application on GPUs. In: CAV. LNCS, accepted for publication (2016)
14. Wijs, A., Bošnački, D.: GPUexplore : Many-Core On-the-Fly State Space Exploration Using GPUs. In: TACAS. LNCS, vol. 8413, pp. 233–247. Springer (2014)
15. Wijs, A., Bošnački, D.: Many-core on-the-fly model checking of safety properties using GPUs. STTT 18(2), 1–17 (2015)
16. Wu, Z., Liu, Y., Liang, Y., Sun, J.: GPU Accelerated Counterexample Generation in LTL Model Checking. In: ICFEM. LNCS, vol. 8829, pp. 413–429. Springer (2014)
17. Wu, Z., Liu, Y., Sun, J., Shi, J., Qin, S.: GPU Accelerated On-the-Fly Reachability Checking. In: 20th International Conference on Engineering of Complex Computer Systems. pp. 100–109. IEEE (2015)