GPUExplore 3.0: GPU Accelerated State Space Exploration for Concurrent Systems with Data

Anton Wijs(✉) and Muhammad Osama

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

Abstract. GPUExplore 3.0 is an explicit state space exploration tool that runs entirely on a graphics processing unit (GPU), and supports models of concurrent systems with data variables. We discuss its workflow and modelling language, present several design decisions regarding work distribution and retrieval, and experimentally evaluate the impact of those decisions. Our tool achieves acceleration up to 115× and 28× compared to single- and four-core LTSmin, respectively. It currently checks for deadlocks, with verification of temporal logic formulae planned for the near future.

Keywords: Explicit state space exploration · finite-state machines · GPU

1 Introduction

Graphics processing units (GPUs) are successfully applied for a plethora of applications, ranging from fluid dynamics [3] to deep learning [21], to drastically speed up computations, and in the last decade, also have contributed to accelerating explicit-state model checking [2,5,8,23,34,35,38–41,43], term rewriting [12], symbolic model checking [24,28], and SAT solving [25–27,29,44,45]. Initially, they were used to speed up specific aspects of model checking, such as probability computations for probabilistic model checking [4,17,36], successor generation [10,11], property checking after the state space had been constructed on the CPU [1], and counter-example construction [42]. GPUExplore [39,41] was the first tool to explicitly explore state spaces entirely on a GPU, without any computations performed by the CPU. Soon, other tools followed, most notably Grapple [8], a swarm-based explorer, ParaMoc, a model checker for pushdown automata [35], and VoxLogicA-GPU [5], a spatial model checker to reason about (medical) images.

In GPUExplore 2.0, each individual process in a concurrent system is encoded as a Labelled Transition System (LTS) [20] that is stored in memory as a sparse matrix [32]. However, this does not allow efficient encodings of concurrent systems *with variables*. For example, consider a system with two 32-bit integer variables x and y, and one process in which y is assigned the value of x at

G. Caltais and C. Schilling (Eds.): SPIN 2023, LNCS 13872, pp. 188–197, 2023.
https://doi.org/10.1007/978-3-031-32157-3_11

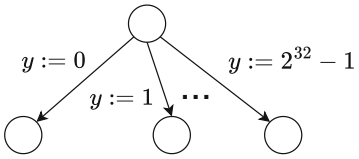

Fig. 1. Handling variables in GPUExplore 2.0.

some point. Allowing for all possible values, GPUExplore 2.0 requires that the LTS describing this process contains at least 2^{32} states, just to distinguish all possible values assigned to y (see Fig. 1). Thus, as variables are introduced, the matrices grow rapidly. Furthermore, GPU state space exploration tools are not user-friendly. Providing input is tedious, requiring manually setting up low-level descriptions of models [8,41] or using a chain of other tools [35,41].

For GPUExplore 3.0, we wanted to change that, and directly support a richer modelling language. The tool altogether avoids storing the input model in memory. To make this possible and high-performant, we developed a code generator that produces GPU code specific for verifying a given input model. Conceptually, this is similar to how Spin transforms Promela models to `pan` code [14]. GPUExplore 3.0 is the first GPU tool to apply this. Although, at a high level, its exploration mechanism has remained the same, its code base has drastically changed, the result of three years of work. The tool can check for deadlocks, and we plan to add support for Linear Temporal Logic (LTL).

In fact, this code generation extends further than is typical for CPU-based model checkers such as Spin. With the introduction of variables in input models, states grow in size. GPUExplore 3.0 is the first GPU tool *in general*, to maintain a *tree database* [18,19]. The states of input models are stored as *binary trees*, which enables effective data sharing. This requires code generation of the storage functions, as the structure and size of trees depend on the input model, and tree storage has to be performed in a non-recursive way, since recursion is detrimental to GPU performance. In addition, it is the first GPU tool to apply *Cleary compression* [6,7] to store tree roots, allowing 64-bit roots to be stored in 32-bit integers. This combination means that once a few million states have been stored, the storage of each additional state requires only 32 bits, *independent* of its size. This is completely novel for GPU hash tables in general [22].

In this paper, we present the workflow and modelling language of GPUExplore 3.0, discuss design decisions regarding work distribution and work fetching, and we experimentally evaluate the impact of those decisions.

2 Workflow and Modelling Language

Workflow. Figure 2 presents the workflow of GPUExplore 3.0. The tool accepts models written in the *Simple Language of Communicating Objects* (Slco) [31], described in more detail later. Given an input model, a code

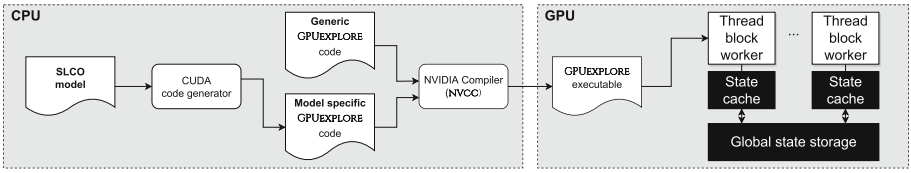

Fig. 2. The workflow of GPUExplore 3.0.

generator, implemented in Python using textX [9] and Jinja2[1], produces *model-specific* code written in NVIDIA's CUDA C++. This code entails next-state computation functions, i.e., functions that given a system state s, produce the successor system states that can be reached from s by executing a transition. Slco models consist of a finite number of Finite State Machines (FSMs) that concurrently execute transitions. In the model-specific code, one next-state computation function is produced for each FSM in the model, allowing for the successor states of a single state to be constructed in parallel, with the functions executed by different threads. This parallel construction of successors does not influence the correctness of the exploration: together, the threads end up exploring all possible execution paths of the input Slco model. In addition, the model specific code involves the handling of state trees, the structure and size of which depend on the input model.

Combined with GPUExplore's *generic* code, which implements the control flow and hash table, the code is compiled using NVIDIA's nvcc compiler. The resulting executable is suitable for CUDA-compatible GPUs with at least compute capability 7.0 and 24 GB global memory. GPUExplore launches many thread blocks of 512 threads each. Each block uses fast on-chip memory[2] to maintain a *state cache*, in which the resulting successors of next state computation are stored, before the block checks the global tree database, access to which is much slower. The database is located in global memory, which is the largest on a GPU (24 GB in a Titan RTX). GPUExplore operates in iterations. In each iteration, each block obtains states that require processing, computes successors, and stores them in the tree database if needed. This is repeated until all discovered states have been processed.

Slco. Slco models contain a finite number of FSMs and have global and FSM-local variables. The types Boolean, Integer, Byte, and arrays of those types are supported. Each FSM contains a finite number of transitions between its states, with one executable (atomic) statement associated with each transition. Statements can refer to all shared variables and those of the corresponding FSM, and are of the form $[e; x_0 := e_0; \ldots; x_n := e_n]$. The x_i's are references to variables or array elements and each e_i is an expression of the same type as x_i, and is constructed by combining references to variables and/or array elements using the typical logical and numerical operators, and e is a Boolean expression. Statement

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

[2] On a Titan RTX, used for this work, on-chip memory is 49,152 bytes in size.

```
model M {
  classes
     GlobalClass {
       variables
       Byte c := 1
       Integer x1, x2
       state machines
       S0 {
         initial Q
         states  R S
         transitions
         Q -> R {
           [c < 20; x1 := c]
         }
         R -> S {
           [x1 := x1 + c]
         }
         ...
       }
       S1 {
          ...
       }
     }

  objects globalObject:
      GlobalClass()
}
```

(a) SLCO model M

```
switch (current_state) {
  case 0:
  {
    // Allocate register memory
    // to process transition(s).
    elem_inttype buf32_0, buf32_1;
    indextype bufaddr_0, bufaddr_1;

    // Q --{ [ c < 20; x1 := c ] }--> R

    mode = STORED;
    // Fetch values of unguarded variables.
    part1 = get_vectorpart(node_index, 0);
    part2 = get_vectorpart(node_index, 1);
    get_globalObject_c(&buf32_0, part1,
        part2);
    // Statement computation.
    if (buf32_0 < 20) {
       target = 1;
       buf32_1 = buf32_0;
       mode = (mode == STORED ? TO_CACHE :
           TO_GLOBAL);
       while (mode != STORED
              && mode != GLOBAL_STORED) {
       // Store new state vector in the
           cache
       // or the global hash table.
       ...
```

(b) Generated CUDA code for M

Fig. 3. Translating SLCO models into CUDA.

$[x_0 := e_0; \ldots; x_n := e_n]$ is shorthand for $[\textbf{true}; x_0 := e_0; \ldots; x_n := e_n]$, and $[e]$ is a statement without assignments. The semantics of a transition is (informally) as follows: if e of its statement evaluates to **true**, the assignments $x_0 := e_0; \ldots x_n := e_n$ can be executed in sequence, by which the variables are updated, and the FSM atomically changes state, moving from the source state of the transition to the target state. If multiple transitions can be executed, the FSM changes state non-deterministically. Regarding concurrency, SLCO has an interleaving semantics.

Figure 3 presents an example SLCO FSM and part of the generated code. The FSM is taken from a translation of the `adding.1` model from the BEEM benchmark suite [30]. It has three process states, `Q` being the initial state. The transition statements refer to two of the three variables in the model, `c` and `x1`.

Given a system state and an FSM, a GPU thread generates successors by executing the corresponding next-state computation function. This function contains a big `switch` statement to consider the execution of transitions based on the current state of the FSM. In the example, if this FSM state, fetched from the system state and stored in the variable `current_state`, is `Q` (encoded as `0`), then the thread will retrieve the value of `c`, and store it in the variable `buf32_0`, located in thread-local *register memory*. If this value is smaller than `20`, the target FSM state is set to `1` (`R`) and the register variable `buf32_1`, associated with

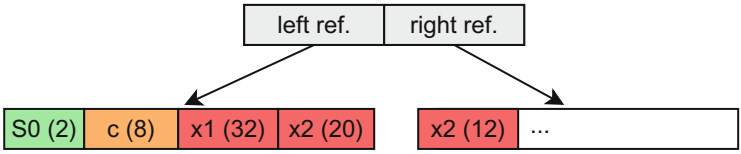

Fig. 4. State tree example.

`x1`, is assigned the value of `buf32_0`, i.e., `c`. Next, the thread will construct the new successor state by combining the original state with the new values, and store the new state in the state cache or, if it is full, the global tree database.

System states are stored as binary trees, with each tree node being a 64-bit integer. Each node can store up to 62 bits of information, with 2 bits used for bookkeeping. Figure 4 shows an example of such a tree, for the FSM given in Fig. 3. The leaf on the left stores the current state of FSM `S0`, which requires 2 bits, followed by the values of the variables. For `x2`, the value is stored in two leaves. The root consists of two references to the leaves, each requiring 29 bits to refer to a position in a hash table for non-roots with 2^{29} entries, but can be physically stored as a 32-bit integer in a separate root table with 2^{32} entries, using Cleary compression [6]. For this, invertible hash functions h_i are used. Given a node n, $h_i(n)$ provides both an address a and a remainder n' of less than 32 bits, which is stored at a. Given a remainder n' stored at a, n can be reconstructed by computing $h_i^{-1}(a, n')$. More details about the state storage can be found in [37].

3 Work Distribution and Retrieval Optimisations

Work Distribution over Thread Blocks. Each thread block has a *work tile* of a fixed size, which is filled with states that require processing at the start of each iteration. As the block produces new states, it can claim them for processing in the next iteration, but as soon as it produces more states than it can fit in its tile, the remaining work is left in the tree database for other blocks. In this way, GPUexplore does not apply *work stealing*, but rather *work sharing*.

Work Distribution Inside a Block. Inside a block, threads execute in groups of 32 threads, called *warps*. Each warp has a single program counter, hence the threads run in lock-step. This means that whenever the threads in a warp *diverge*, i.e., execute different lines of code, performance deteriorates, as the whole warp has to move over a line of code if at least one thread needs to execute it. For GPUexplore 3.0, we experimented with several options for work distribution in a block. At the top in Fig. 5, a strategy is visualised called *thread-to-FSM*. In this example, the model contains three FSMs, and their FSM states for the i-th state in the work tile are named S_0^i, S_1^i and S_2^i. The colours represent different warps. For ease of presentation, we assume that a warp has four threads. Given that

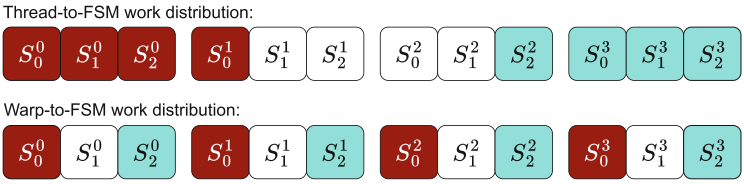

Fig. 5. Thread group tile processing strategies.

for each FSM, we have a separate next-state function, this distribution leads to the threads inside a warp diverging when they call the next-state function for their FSM. Another distribution is illustrated at the bottom of Fig. 5, called *warp-to-FSM*. Now, all threads in a warp are assigned the same FSM, resulting in those threads calling the same function using different data.

Reducing thread divergence can be taken further. Two threads that execute the same function but have different current FSM states still diverge, as they execute different `switch` cases (see Fig. 3). To minimise this, we sort the tile for each warp w.r.t. the current FSM state of its designated FSM. This results in all states with the same FSM state for the designated FSM being placed at consecutive positions in the tile, thereby stimulating that threads with consecutive IDs access states with the same current FSM state. Since the work tile is sufficiently small for the threads in a warp to store the tile in their combined register memory, sorting can be done in the register memory with *intra-warp bitonic merge sort* [15], using fast intra-warp instructions.

Multiple Iterations. Another optimisation is to execute *multiple iterations* in each exploration function call. GPUExplore calls an exploration function to execute one or more next state iterations. Shared memory is wiped once a function execution finishes. With multiple iterations, a block can reuse the trees in its state cache constructed in one iteration, for exploration in the next one.

4 Tool Evaluation

Our code generator[3] can be launched with `python slcotogpuexplore.py <input-model>.slco [options]`. It takes an Slco model as input and produces CUDA code. Several `options` can be given such as selecting a work distribution scheme or specifying the number of iterations per kernel launch. The code can be compiled with CUDA 11+ to produce an executable `gpuexplore` that can be launched with `./gpuexplore [-k <#ITERATIONS>]`.

For evaluation, we used Slco models translated from a representative subset of the Beem benchmark suite [30]. We scaled up some models, marked with '+'. For all experiments, we used CUDA 11.4, and a machine with a 4-core CPU i7-7700 operating at 3.6 GHz, 32GB RAM, and a Titan RTX GPU, running Linux Mint 20.

[3] GPUExplore is available for download here: https://bit.ly/3CUXTY8.

Table 1. Speed in millions of states per second. `tF`: thread-to-FSM, `wF`: warp-to-FSM, `wFs` (`+<n>`): `wF` + `sorting` (`+n` iterations), `SU-<to>`: Speedup of `wFs+30` vs. `<to>`, -O.M.-: out of memory.

Model	States	Spin	LTSmin		GPUexplore 3.0							SU-Spin	SU-LTSmin		SU-tF
		4-core	1-core	4-core	tF	wF	wFs	wFs+10	wFs+30	wFs+50	wFs+70	4-core	1-core	4-core	
adding.20+	84,709,120	3.22	1.40	3.94	58.02	55.65	57.18	**83.36**	77.89	67.12	59.60	**24.2x**	**55.8x**	**19.8x**	**1.2x**
adding.50+	529,767,730	-O.M.-	1.29	5.36	106.19	100.10	102.73	143.09	**148.28**	145.99	144.86	–	**114.7x**	**27.7x**	**1.4x**
anderson.6	18,206,917	1.36	0.67	1.31	9.58	13.80	16.02	31.58	31.57	**31.82**	31.71	**23.2x**	**47.2x**	**24.1x**	**3.3x**
anderson.7	538,699,029	-O.M.-	0.38	-O.M.-	7.93	15.43	20.95	**20.95**	19.78	19.75	19.68	–	**52.5x**	–	**2.5x**
at.5	31,999,440	1.50	0.61	1.88	14.05	23.79	28.73	36.74	36.54	**37.19**	36.58	**24.4x**	**60.4x**	**19.4x**	**2.6x**
at.6	160,589,600	0.87	0.66	2.39	14.39	27.76	38.34	**40.83**	40.56	40.59	40.62	**46.7x**	**61.9x**	**17x**	**2.8x**
at.7	819,243,816	-O.M.-	0.63	2.37	8.91	17.15	23.42	**23.60**	23.16	23.19	23.09	–	**36.7x**	**9.8x**	**2.6x**
bakery.5	7,866,401	2.57	0.62	0.90	7.52	7.71	7.46	11.29	19.02	**20.15**	19.98	**7.4x**	**30.9x**	**21x**	**2.7x**
bakery.7	29,047,471	2.59	0.76	1.62	8.47	9.10	9.06	20.80	29.12	30.98	**31.13**	**11.2x**	**38.5x**	**18x**	**3.7x**
bakery.8+	841,696,300	1.27	0.65	2.44	13.06	20.85	29.71	34.11	34.21	**34.31**	34.04	**27x**	**52.5x**	**14x**	**2.6x**
elevator2.3	7,667,712	1.10	0.46	0.99	3.48	3.32	3.24	5.98	6.06	**6.20**	6.10	**5.5x**	**13.1x**	**6.2x**	**1.8x**
elevator2.4+	91,226,112	0.56	0.57	1.95	2.97	3.74	**3.79**	3.22	3.28	3.33	3.34	**5.8x**	**5.8x**	**1.7x**	**1.1x**
elevator2.5+	1,016,070,144	-O.M.-	0.45	1.63	1.72	**1.88**	**1.88**	1.85	1.83	1.83	1.82	–	**4.1x**	**1.1x**	**1.1x**
frogs.4	17,443,219	2.23	0.50	1.42	7.37	10.06	9.75	11.13	**11.43**	11.32	11.26	**5.1x**	**22.9x**	**8x**	**1.5x**
frogs.5	182,772,126	1.05	0.70	2.63	6.45	9.63	9.61	10.27	**10.31**	10.23	10.18	**9.8x**	**14.6x**	**3.9x**	**1.6x**
lamport.6	8,717,688	1.38	0.49	1.10	5.07	5.20	5.09	17.94	27.35	**27.99**	27.80	**19.9x**	**55.6x**	**25x**	**5.5x**
lamport.7	38,717,846	1.82	0.62	1.98	11.00	18.13	23.04	33.50	34.47	34.45	**34.55**	**18.9x**	**55.5x**	**17.4x**	**3.1x**
lamport.8	62,669,317	1.78	0.80	2.19	10.73	18.55	25.45	34.31	34.92	35.12	**35.35**	**19.7x**	**43.9x**	**15.9x**	**3.3x**
loyd.3	239,500,800	-O.M.-	0.61	2.34	43.35	45.91	43.25	50.63	50.46	50.89	**51.04**	–	**82.3x**	**21.6x**	**1.2x**
mcs.5	60,556,519	0.62	0.42	1.49	12.07	19.44	24.26	29.98	**30.44**	30.34	30.25	**49.5x**	**72.1x**	**20.4x**	**2.5x**
peterson.5	131,064,750	1.62	0.73	2.44	11.75	21.13	28.44	**31.61**	31.28	30.76	30.70	**19.3x**	**43.1x**	**12.8x**	**2.6x**
peterson.6	174,495,861	0.76	0.68	2.45	12.05	21.04	30.47	**33.72**	33.58	33.31	33.19	**44.4x**	**49.4x**	**13.7x**	**2.8x**
peterson.7	142,471,098	1.50	0.72	2.27	10.17	20.93	22.37	**25.74**	25.44	25.21	25.21	**17x**	**35.4x**	**11.2x**	**2.5x**
phils.7	71,934,773	0.30	0.23	0.76	1.87	4.59	5.57	**5.66**	5.64	5.61	5.59	**19x**	**24.4x**	**7.4x**	**3x**
phils.8	43,046,720	0.36	0.28	0.79	2.64	9.07	8.96	**9.35**	9.27	9.21	9.17	**25.7x**	**33.5x**	**11.8x**	**3.5x**
szymanski.5	79,518,740	1.57	0.50	1.82	7.07	12.15	17.02	**19.03**	18.34	18.31	18.35	**11.7x**	**37x**	**10.1x**	**2.6x**
Average		1.43	0.63	2.00	15.30	19.85	22.99	29.63	**30.55**	30.20	29.81	**20.7x**	**44x**	**14x**	**3x**

Table 1 shows the results, comparing the impact of the presented options with four-core SPIN 6.5.1 [13] and single- and four-core LTSMIN 3.0.2 [16]. We only enabled state compression and basic reachability (without property checking) in those tools, to favour fast exploration of large state spaces. As GPUEXPLORE 3.0 does not yet have support for on-the-fly reduction methods, such as partial-order reduction [23], these have been disabled for all tools. Since LTSMIN scales near-linearly with the number of cores [33], the results indicate how many cores LTSMIN needs to be as fast as GPUEXPLORE. The best speeds are highlighted in bold. Overall, warp-to-FSM with sorting and 30 iterations is most successful.

Table 2. Millions of states per second for GPUEXPLORE 3.0 vs. version 2.0.

Tool	anderson.6	anderson.7	lamport.8	peterson.5	peterson.6	peterson.7	szymanski.5
2.0	15.863	-O.M.-	33.063	16.874	16.705	13.581	**26.454**
3.0	**34.111**	**22.326**	**35.387**	**32.331**	**34.902**	**26.183**	18.357

Finally, in Table 2, we compared GPUEXPLORE 3.0 with version 2.0 on the Titan RTX. In the comparison, we used all BEEM models for which corresponding GPUEXPLORE 2.0 models exist: `anderson.6` and `.7`, `lamport.8`, `peterson.5`,

`.6` and `.7` and `szymanski.5`. GPUExplore 2.0 ran out of memory on the `anderson.7` model while GPUExplore 3.0 was able to explore all models with an average acceleration of 1.8×. A comparison with Grapple is discussed in a recent paper [37].

References

1. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing fast LTL model checking algorithms for many-core GPUs. JPDC **72**(9), 1083–1097 (2012). https://doi.org/10.1016/j.jpdc.2011.10.015
2. Bartocci, E., DeFrancisco, R., Smolka, S.A.: Towards a gpgpu-parallel spin model checker. In: SPIN 2014, pp. 87–96. ACM, New York (2014). https://doi.org/10.1145/2632362.2632379
3. Bertolli, C., Betts, A., Mudalige, G., Giles, M., Kelly, P.: Design and performance of the OP2 library for unstructured mesh applications. In: Alexander, M., et al. (eds.) Euro-Par 2011. LNCS, vol. 7155, pp. 191–200. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29737-3_22
4. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel probabilistic model checking on general purpose graphics processors. STTT **13**(1), 21–35 (2011). https://doi.org/10.1007/s10009-010-0176-4
5. Bussi, L., Ciancia, V., Gadducci, F.: Towards a spatial model checker on GPU. In: Peters, K., Willemse, T.A.C. (eds.) FORTE 2021. LNCS, vol. 12719, pp. 188–196. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78089-0_12
6. Cleary, J.: Compact hash tables using bidirectional linear probing. IEEE Trans. Comput. **c-33**(9), 828–834 (1984). https://doi.org/10.1109/TC.1984.1676499
7. Darragh, J., Cleary, J., Witten, I.: Bonsai: a compact representation of trees. Softw. Pract. Exp. **23**(3), 277–291 (1993). https://doi.org/10.1002/spe.4380230305
8. DeFrancisco, R., Cho, S., Ferdman, M., Smolka, S.A.: Swarm model checking on the GPU. Int. J. Softw. Tools Technol. Transf. **22**(5), 583–599 (2020). https://doi.org/10.1007/s10009-020-00576-x
9. Dejanović, I., Vaderna, R., Milosavljević, G., Vuković, Ž: TextX: a python tool for domain-specific language implementation. Knowl.-Based Syst. **115**, 1–4 (2017). https://doi.org/10.1016/j.knosys.2016.10.023
10. Edelkamp, S., Sulewski, D.: Efficient explicit-state model checking on general purpose graphics processors. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 106–123. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16164-3_8
11. Edelkamp, S., Sulewski, D.: External memory breadth-first search with delayed duplicate detection on the GPU. In: van der Meyden, R., Smaus, J.-G. (eds.) MoChArt 2010. LNCS (LNAI), vol. 6572, pp. 12–31. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20674-0_2
12. van Eerd, J., Groote, J.F., Hijma, P., Martens, J., Osama, M., Wijs, A.: Innermost many-sorted term rewriting on gpus. Sci. Comput. Program. **225**, 102910 (2023). https://doi.org/10.1016/j.scico.2022.102910
13. Holzmann, G.J.: Parallelizing the spin model checker. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 155–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31759-0_12
14. Holzmann, G.: The model checker spin. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521

15. Hou, K., Liu, W., Wang, H., Feng, W.: Fast segmented sort on gpus. In: Gropp, W.D., Beckman, P., Li, Z., Cazorla, F.J. (eds.) ICS, pp. 12:1–12:10. ACM (2017). https://doi.org/10.1145/3079079.3079105
16. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
17. Khan, M.H., Hassan, O., Khan, S.: Accelerating SpMV multiplication in probabilistic model checkers using GPUs. In: Cerone, A., Ölveczky, P.C. (eds.) ICTAC 2021. LNCS, vol. 12819, pp. 86–104. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85315-0_6
18. Laarman, A.: Optimal compression of combinatorial state spaces. Innov. Syst. Softw. Eng. **15**, 235–251 (2019). https://doi.org/10.1007/s11334-019-00341-7
19. Laarman, A., van de Pol, J., Weber, M.: Parallel recursive state compression for free. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 38–56. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22306-8_4
20. Lang, F.: Refined interfaces for compositional verification. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 159–174. Springer, Heidelberg (2006). https://doi.org/10.1007/11888116_13
21. Le, Q.V., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., Ng, A.Y.: On optimization methods for deep learning. In: Getoor, L., Scheffer, T. (eds.) ICML, pp. 265–272. Omnipress (2011)
22. Lessley, B.: Data-parallel hashing techniques for GPU architectures. IEEE Trans. Parallel Distrib. Syst. **31**(1), 237–250 (2019). https://doi.org/10.1109/TPDS.2019.2929768
23. Neele, T., Wijs, A., Bošnački, D., van de Pol, J.: Partial-order reduction for GPU model checking. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 357–374. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_23
24. Osama, M.: GPU Enabled Automated Reasoning. Ph.D. thesis, Eindhoven University of Technology (2022). ISBN: 978-90-386-5445-4
25. Osama, M., Gaber, L., Hussein, A.I., Mahmoud, H.: An efficient SAT-based test generation algorithm with GPU accelerator. J. Electron. Test. **34**(5), 511–527 (2018). https://doi.org/10.1007/s10836-018-5747-4
26. Osama, M., Wijs, A.: Parallel SAT simplification on GPU architectures. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 21–40. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_2
27. Osama, M., Wijs, A.: SIGmA: GPU accelerated simplification of SAT formulas. In: Ahrendt, W., Tapia Tarifa, S.L. (eds.) IFM 2019. LNCS, vol. 11918, pp. 514–522. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34968-4_29
28. Osama, M., Wijs, A.: GPU acceleration of bounded model checking with ParaFROST. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 447–460. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_21
29. Osama, M., Wijs, A., Biere, A.: SAT solving with GPU accelerated inprocessing. In: TACAS 2021. LNCS, vol. 12651, pp. 133–151. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_8
30. 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). https://doi.org/10.1007/978-3-540-73370-6_17

31. de Putter, S., Wijs, A., Zhang, D.: The SLCO framework for verified, model-driven construction of component software. In: Bae, K., Ölveczky, P.C. (eds.) FACS 2018. LNCS, vol. 11222, pp. 288–296. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02146-7_15
32. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia (2003)
33. van der Vegt, S., Laarman, A.: A parallel compact hash table. In: Kotásek, Z., Bouda, J., Černá, I., Sekanina, L., Vojnar, T., Antoš, D. (eds.) MEMICS 2011. LNCS, vol. 7119, pp. 191–204. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-25929-6_18
34. Wei, H., Chen, X., Ye, X., Fu, N., Huang, Y., Shi, J.: Parallel model checking on pushdown systems. In: ISPA/IUCC/BDCloud/SocialCom/SustainCom, pp. 88–95. IEEE (2018). https://doi.org/10.1109/BDCloud.2018.00026
35. Wei, H., Ye, X., Shi, J., Huang, Y.: ParaMoC: A Parallel Model Checker for Pushdown Systems. In: ICA3PP. LNCS, vol. 11945, pp. 305–312. Springer (2019). https://doi.org/10.1007/978-3-030-38961-1_26
36. 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). https://doi.org/10.1007/978-3-642-31759-0_9
37. Wijs, A., Osama, M.: A GPU tree database for many-core explicit state space exploration. In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS 2023. LNCS, vol. 13993, pp. 684–703. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_35
38. Wijs, A.: BFS-based model checking of linear-time properties with an application on GPUs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 472–493. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_26
39. Wijs, A., Bošnački, D.: GPUexplore: many-core on-the-fly state space exploration using GPUs. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 233–247. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_16
40. Wijs, A., Bošnački, D.: Many-core on-the-fly model checking of safety properties using GPUs. Int. J. Softw. Tools Technol. Transf. **18**(2), 169–185 (2015). https://doi.org/10.1007/s10009-015-0379-9
41. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: unleashing GPU explicit-state model checking. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 694–701. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_42
42. Wu, Z., Liu, Y., Liang, Y., Sun, J.: GPU accelerated counterexample generation in LTL model checking. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 413–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11737-9_27
43. Wu, Z., Liu, Y., Sun, J., Shi, J., Qin, S.: GPU accelerated on-the-fly reachability checking. In: ICECCS 2015, pp. 100–109 (2015). https://doi.org/10.1109/ICECCS.2015.21
44. Youness, H., Osama, M., Hussein, A., Moness, M., Hassan, A.M.: An effective SAT solver utilizing ACO based on heterogenous systems. IEEE Access **8**, 102920–102934 (2020). https://doi.org/10.1109/ACCESS.2020.2999382
45. Youness, H.A., Ibraheim, A., Moness, M., Osama, M.: An efficient implementation of ant colony optimization on gpu for the satisfiability problem. In: PDP, pp. 230–235. IEEE (2015). https://doi.org/10.1109/PDP.2015.59