International Journal on Software Tools for Technology Transfer (2020) 22:583–599
https://doi.org/10.1007/s10009-020-00576-x

STTT

Special Issue: SPIN 2019

Swarm model checking on the GPU

Richard DeFrancisco[1] · Shenghsun Cho[1] · Michael Ferdman[1] · Scott A. Smolka[1]

Published online: 16 June 2020

Abstract
We present Grapple, a new and powerful framework for explicit-state model checking on GPUs. Grapple is based on swarm verification (SV), a model-checking technique wherein a collection or swarm of small, memory- and time-bounded verification tests (VTs) are run in parallel to perform state-space exploration. SV achieves high state-space coverage via diversification of the search strategies used by constituent VTs. Grapple represents a swarm implementation for the GPU. In particular, it runs a parallel swarm of internally parallel VTs, which are implemented in a manner that specifically targets the GPU architecture and the SIMD parallelism its computing cores offer. Grapple also makes effective use of the GPU shared memory, eliminating costly inter-block communication overhead. We conducted a comprehensive performance analysis of Grapple focused on various design parameters, including the size of the visited-state queue structure, implementation of guard statements, and nondeterministic exploration order. Tests are run with multiple hardware configurations, including on the Amazon cloud. Our results show that Grapple performs favorably compared to the SPIN swarm and a prior non-swarm GPU implementation. Although a recently debuted FPGA swarm is faster, the deployment process to the FPGA is much more complex than Grapple's.

Keywords GPU · Model checking · Swarm verification · Grapple

1 Introduction

Modern computing exists in a space that is increasingly parallel, distributed, and heterogeneous. High-performance co-processors such as GPUs (Graphics Processing Units) are utilized in many super-computing applications due to their high computational throughput, energy efficiency, and low cost [20]. GPGPU (General-Purpose Computing on a GPU) is achieved through the use of GPU programming languages such as the Open Computing Language (OpenCL) [33] and the Compute Unified Device Architecture (CUDA) [1].

In 2014, we adapted the multicore SPIN model checking (MC) algorithm of [22] to the GPU [7]. While our approach achieved speedups up to 7.26× over traditional SPIN, and 1.26× over multicore SPIN, it was severely limited by the memory footprint of the GPU, and by an explicit limit on the state-vector size set by the hash function [34].

✉ Richard DeFrancisco
rdefrancisco@cs.stonybrook.edu

1 Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-2424, USA

The introduction of Swarm Verification (SV) in [25] represented an entirely new approach to parallel MC. In SV, a large number of MC instances are executed in parallel, each with a restricted memory footprint and a different search path. Each instance is called a verification test (VT), because it does not seek to cover the full state space as a model checker would. Through the use of diversification techniques, VTs are largely independent of one other in terms of the portions of the model's state space they cover. By executing a sufficiently large number of VTs, one is therefore statistically guaranteed to achieve nearly complete, if not complete coverage of the entire state space.

In this paper, we present Grapple, bringing the lightweight yet powerful nature of SV to the massively parallel GPU architecture. While other swarm implementations run internally sequential VTs in parallel, Grapple VTs are internally parallel and evolved from our previous GPU-based MC design [7]. Each VT runs on a single block of the GPU, with a bitstate hash table in shared memory, compacting per-state storage by a factor of 64 compared to the cuckoo tables used in [7]. These tables use the hash function of [27], eliminating the hard 64-bit state vector limit of our previous model checker design.

Grapple VTs run in parallel on all available GPU streaming multiprocessors (SMs), and make efficient use of the GPU scheduler to quickly replace jobs the instant an SM becomes available. As VTs are independent of each other and each one is tightly bound to a single chip on hardware, there is no need for inter-block communication or additional synchronization primitives.

To assess Grapple's performance, we used a benchmark specifically designed for SV-based model checkers [12,25]: a model that can randomly generate more than 4 billion states. Exploration progress in the benchmark is captured by the visitation of 100 randomly distributed states, or *waypoints*, with 100 waypoints approaching complete state-space exploration. Our experiments, which we ran on multiple hardware configurations, including the Amazon cloud [2], evaluate the impact of variations in queue size, guard-statement implementation, and nondeterministic exploration order.

We also compared Grapple's performance with the FPGA swarm implementation of [12], the CPU swarm of [25], and the original (non-swarm) GPU implementation of [7]. Grapple easily outperforms the GPU implementation and the CPU swarm, and reaches all waypoints in a number of VTs comparable to that required by the FPGA implementation. While it cannot compete in raw speed with the hardware-level FPGA implementation, it offers much easier deployment, with VTs that complete in under a second.

We additionally evaluated Grapple using multiple configurations of the Dining Philosophers problem, a small model with a known state-space size and deadlock violation. Results are also included for the BEEM [4] benchmark models considered in [7].

In summary, our main contributions are as follows. (i) We introduce Grapple, a GPU-based swarm verification model checker with internally parallel verification tasks. (ii) We analyze structural elements of VTs (e.g., search strategy, queue size, guard logic, number of threads per VT) to determine how they impact the rate of exploration. (iii) We compare Grapple's performance to previous SV implementations on the CPU [25] and FPGA [12], as well as to our non-swarm GPU-based model checker [7].

The rest of the paper is organized as follows. Section 2 provides background on GPU hardware, the CUDA programming model, the SPIN model checker, and swarm verification. Section 3 presents our Grapple model checker. Section 4 presents our various experimental results. Section 5 considers related work. Section 6 interprets our findings and offers directions for future work.

This paper is an extended version of [13], with additional experiments and analysis, as well as the introduction of a new search strategy: *Parallel Deep Search* (PDS), a search method specifically designed for swarm environments. Our new array of experiments serves to: 1) illustrate the frequency of specific waypoint occurrence within a Grapple swarm; 2) showcase additional Dining Philosopher results, including process-PDS, and add two additional BEEM models: Anderson and Peterson; 3) add structure-oriented results, including two-phase swarms, depth-limited PDS, and scatter PDS; and 4) present Grapple swarms composed of VTs with multiple diversification techniques, including on the cloud. We believe that these additions represent at least 30% in new material compared to the conference version [13].

2 Background

To motivate our design decisions for Grapple, we first explain the intricacies of GPU hardware and the associated CUDA programming model, and provide an overview of the SPIN model checker [32], on which Grapple is based. Further information on the GPU hardware and CUDA software is available in the CUDA C Programming Guide [9].

2.1 GPU hardware model

The GPU is a high-performance co-processor designed to efficiently render 3D graphics in real time. GPUs are well-suited for linear algebra, matrix arithmetic, and other computations frequently used in graphical applications. As illustrated in Fig. 1, the GPU architecture consists of a scalable array of N multithreaded streaming multiprocessors (SMs), each of which is made up of M stream processor (SP) cores. Each core is equipped with a fully pipelined integer-arithmetic logic unit (ALU) and a floating-point unit (FPU) that execute one integer or floating-point instruction per clock cycle. Each SM controls a *warp* of 32 threads, executing the same instructions in lock-step for all threads.

The GPU features a number of memory types, differing in access speed, capacity, and read/write availability. Global memory is large (order of gigabytes), available device-wide, but relatively slow. Constant memory is a cached, read-only memory intended for storing constant values that are not updated during execution. Finally, each SM has a shared memory region ($16 - 48$ KB). In practice, accessing shared memory can be up to 100 times faster than using global memory for the same transaction.

Devices connect to the host machine using the the PCIe bus. Communication between the host and device are extremely costly compared to on-board memory accesses, including those that use global memory.

2.2 CUDA programming model

CUDA is the proprietary NVIDIA programming model for general-purpose computing on their GPU architecture. While

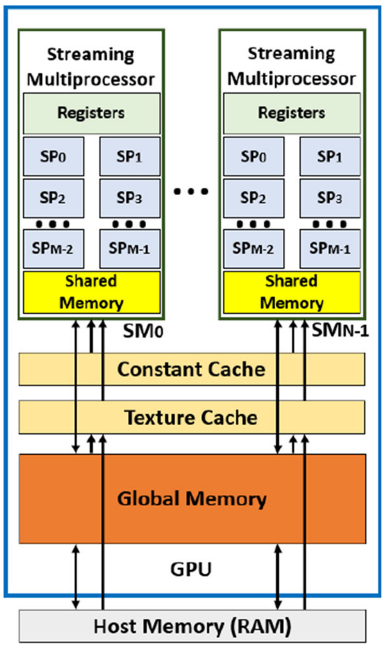

Fig. 1 GPU hardware model. SP = Stream processor

the alternative model, OpenCL, is universally compatible with all GPU architectures, the high-performance of CUDA has led to wide adoption. We decided to write Grapple in CUDA for this reason, but an OpenCL implementation would be very similar.

The CUDA parallel computing model uses tens of thousands of lightweight threads assembled into one- to three-dimensional thread blocks. A thread executes a function called a kernel, which contains the computations to be run in parallel. Each thread uses different parameters. Threads located in the same thread block can work together in several ways. They can insert a synchronization point into the kernel, which requires all threads in the block to reach that point before execution can continue. They can also share data during execution. In contrast, threads located in different thread blocks cannot communicate in such ways and essentially operate independently.

Shared-memory transactions are typically parallel to some number n distinct banks, but if two or more address requests fall in the same bank, the collision causes a serialization of the access. It is therefore important to understand addressing patterns when utilizing shared memory. Register management is also critically important. Use of registers is partitioned among all threads and, as such, using a large number of registers within a CUDA kernel will limit the number of threads that can run concurrently. Double and long long variables, use of shared memory, and unoptimized block/warp geometry all lead to increased register use. If available registers are exhausted, the contents will spill over into local memory—a special type of device memory with the same high-latency and low-bandwidth as global memory.

The SIMD nature of warps on SPs has a great impact on code structure for the GPU. As warps act in lock-step, any branching logic encountered by a warp must have all branches explored by all threads. The data created during the additional branch exploration is simply discarded. This phenomenon is referred to as branch divergence and is warp-local; other warps continue to perform independently of the divergent warp. This can lead to scheduling conflicts where non-branching warps must wait for the divergent warps to complete. It is also generally a performance loss within a warp, especially for cases where one or more branches is long but uncommonly taken.

Finally, kernels can be launched in parallel on a single device, as long as that device has the capacity to do so. Streams are command sequences that execute in order internally, but can be concurrent with each other. The number of concurrent streams is device dependent, and additional streams will queue until the device has availability. Streams are unnecessary to run parallel commands on multiple devices, and are not needed for pipelining data transfers with kernel execution. Two commands from multiple streams cannot run concurrently if the host specifies memory manipulation or kernel launches on stream 0 (default) between them. Synchronization, where necessary, can be invoked within a stream or across streams with provided CUDA sync statements.

2.3 SPIN model checker

SPIN [32] is a widely used model checker designed to verify multi-threaded software. SPIN has an ever-growing list of features and options, including optimization techniques, property specification types, and hardware support. State spaces can be pruned using partial order reduction, speed can be increased by changing search strategies or disabling certain checks, and memory footprint can be reduced through bitstate hashing. SPIN can handle safety and liveness properties, any LTL specification, Büchi automata, never claims, and invariant assertions. Multicore support was added in 2007 [21], improved in 2012 [22], and extended to liveness properties in 2015 [17].

A central feature of the 2012 algorithm is the structure holding the frontier of newly discovered states. In order to assign these states to N worker threads, SPIN uses two sets of $N \times N$ queues. By splitting each frontier queue into an $N \times N$ structure all threads can communicate without the need for mutex locks. Of these two queue sets, one (output) fills with a new frontier as the other (input) empties the current frontier. When the input queue is empty, all threads synchronize and the two swap labels. This process continues until both the

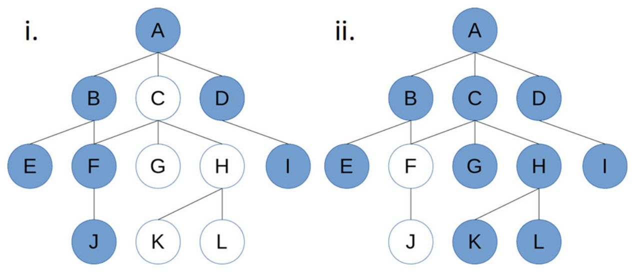

Fig. 2 Pruning states via hash collision. **i** Hash collision $\{B, C\}$ on trace $ABEFJDI$. **ii** Hash collision $\{E, F\}$ on trace $ABECGHKLDI$

input and output are empty or a violation is found. We adopt this structure for Grapple.

2.4 Swarm verification

Recently, support for large-scale parallel model checking on CPU-based systems was added to SPIN in the form of swarm verification (SV) [23–25]. SV is a technique wherein a large number of small verification tasks (VTs) are run in parallel on many independent processors, including multiple CPUs, multicore CPUs, and distributed systems [23]. The term verification test is used in place of model checker or verifier because these tests are not guaranteed to complete. Instead, each test is given a set amount of time and memory to explore whatever portion of the state space it can. VTs can be as small as a number of KBs.

Each VT is independent, and the state space it covers is differentiated through the use of various diversification techniques. These techniques include reversing search direction or search order, randomizing nondeterministic choice order of transitions, and other perturbations of the original search algorithm. VTs do not share resources nor need to live on the same physical machine. Given enough parallel hardware, all VTs can run concurrently. When these resources have more limited availability, VTs will be scheduled like any other batch of independent programs.

The most potent diversification technique is the use of statistically independent hash functions. With up to 10^8 suitable unique 32-bit hash polynomials, in addition to other search diversification methods, the potential number of distinct concurrent searches is easily in the billions [23]. Hash functions reduce the state space graph via collisions; as each hash table is much smaller than the total number of states, collisions are frequent. If we treat each collision as valid (consider them the same state, even if that is not the case), the state space will be quickly, and naturally pruned.

Figure 2 depicts state-space pruning via collision. In both searches, a left-favoring Depth-First Search strategy is used, but their hash tables use different hash polynomials to store states. In the left graph, nodes B and C have the same hashed value, so C appears to be the same state and will not be expanded. In the right graph, E and F have the same value, preventing the expansion of F. Pruning nearly guarantees that an individual VT will not reach the entire state space, but this is not a problem. With a sufficient number of diverse VTs, the swarm as a whole will achieve full coverage.

3 Swarm verification via the Grapple model checker

The Grapple model checker brings the power of GPU computing to the model-checking problem via swarm verification. For simplicity of presentation, we discuss Grapple's design in terms of a Waypoints (WPs) benchmark specifically designed for SV-based model checkers [12,25]. The WP benchmark involves a model that can randomly generate more than 4 billion states. Said model is comprised of 8 processes each in control of 4 bits. At successor generation, the current process will nondeterministically set one of its bits to 1. Exploration progress in the benchmark is captured by the visitation of 100 randomly distributed states, or waypoints, with 100 waypoints suggesting a nearly complete state-space exploration. This style of presentation does not in any way imply that Grapple is limited to this one benchmark; it is still a general-purpose model checker. Indeed, in Sect. 4, we present results for additional models, taken from the BEEM database [4].

Grapple is currently restricted to the verification of safety and reachability properties. This is also the case for [12], whereas the swarm-based model checker of [25] has the full expressiveness of SPIN (LTL, never claims, process invariants, and Büchi automata).

Although traditionally each VT is a small, sequential version of SPIN, this is not the case for Grapple VTs, which run on the GPU. As discussed in Sect. 2.1, the GPU has a SIMD/SIMT programming model: a single instruction or set of instructions is given to a group of threads operating on different data. Warps of 32 threads execute in lock-step, and all branches in logic must be fully explored by the entire warp. Mimicking SPIN by running a completely sequential VT on an entire warp would waste massive amounts of resources. Instead, we use a modified version of the 2014 GPU MC algorithm [7] to run a single, internally parallel VT per warp. VTs execute independently in parallel outside of the warp, but internally (i.e., within a given VT), all data structures are shared among the threads and there is a single state space to explore.

While the queue structure and general search algorithm remain the same as the 2014 MC, as Fig. 3 illustrates, there are a number of alterations made to the GPU VT to take advantage of the new swarm environment. First and foremost, the hash table is a bitstate implementation moved to shared memory. This hash table is only shared among threads

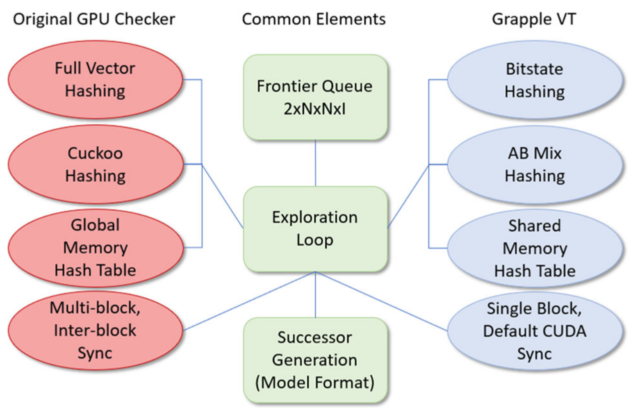

Fig. 3 Original GPU model checker versus Grapple verification task

within a VT and not between VTs. Factors typically considered weaknesses of a shared-memory approach are the hash table's locality to an SM and its small size (48 KB maximum). With each SIMD-parallel VT limited to a single warp, all threads within the VT are guaranteed to be on the same SP within the same SM, and therefore all have access to this structure.

The 48 KB limit is not an issue for VTs utilizing bitstate hashing, as such a table can hold nearly 400,000 entries. This is on the low end of the scale for a VT compared to those in other SV implementations [23,25], but VTs of this size were shown to work well in a recent FPGA implementation [12].

Also as in the FPGA implementation, cuckoo hashing [34] has been replaced with an AB mix function based on the Bob Jenkins Linear Feedback Shift Register (LFSR) [27]. For this purpose, two random integers, A and B, are generated on the host machine for each VT and included as parameters in the VT's kernel launch. This change in hash function is motivated by the desire to better align with the FPGA implementation, as well as the elimination of the multiple-function schema used in the cuckoo algorithm.

For Grapple, the primary requirements for a hashing algorithm are speed and simplicity. Unlike in most other systems, there is no need for the function to minimize collisions, as collisions help diversify VT searches. The random integers are reused on the GPU in some search strategies as quick random-digit generators, as on-device random generation tends to be convoluted and this method is more efficient and suffices for our purposes.

Since each VT is relegated to a single warp, the fast-barrier synchronization [42] used in the previous GPU MC implementation has also been removed. Instead, the on-board CUDA __syncthreads() function is used at the required synchronization points.

Grapple, like the FPGA swarm [12], runs multiple VTs within a single program, with additional copies of that program launched by script if necessary. In contrast, the SPIN swarm [25] is coordinated by a script that simply launches every VT as an independent thread. A Grapple program running on the GPU initiates multiple VTs, each a CUDA kernel, and utilizes streams to run these kernels in parallel whenever possible.

The number of VTs that a core program can launch is dependent upon the hardware of the device(s) available, the memory footprint of each VT, and how initialization and memory transfers are handled. In the current design, all variables and structures are initialized, transferred to the GPU before kernel launch, transferred back to the host after kernel completion, and then freed in a single batch. Theoretically, more VTs could be launched within a program and additional efficiency squeezed out if the transfers were pipelined with some VT execution, but the current arrangement also has benefits.

Since the primary diversification techniques in Grapple are alterations in hash polynomial, search structure, and nondeterminism order, most of the host-level set-up is common across VTs. Overall, these common elements reduce the cost of this process to be nearly negligible when compared to time spent on the device. In this case, pipelining would increase overall complexity of the core code with minimal benefit. On the theme of common initialization, structures are placed in constant memory whenever possible so all VTs gain fast read-only access.

Figure 4 illustrates the control flow of Grapple. Upon start-up, a swarm script launches a CUDA program on all available hardware devices (GPUs). When there is only a single device, these K programs must sequentialize, with one program launching after the execution of the previous program and its sort instance (the Linux sort utility is used to count WPs) has terminated. Internally, each CUDA program initializes a number of VTs, in this case 250, sharing common data wherever possible to minimize overhead. Examples of this include setting up the initial state and sending WP identifiers to GPU constant memory. This initialization/pre-launch procedure runs on the CPU (host).

Each VT is assigned to a single stream, and as many of them as possible will be launched in parallel to the N streaming multiprocessors (SMs) available on the device. The number of VTs maintained by a given GPU program, in this case 250, is a function of the global memory footprint of the VTs' data structures. While the hash tables are assigned to the 48KB of on-chip shared memory, frontier queues and other support structures must still hold the full-length global state vectors and combine to reach the upper limits of the GPU global memory. Despite sitting on global memory, these structures are still access-limited to a single VT, maintaining VT independence.

Once launched, a VT executes its complete search until its frontier queues are empty, and there are no more states to be explored. This exhaustion process is driven by the limited size

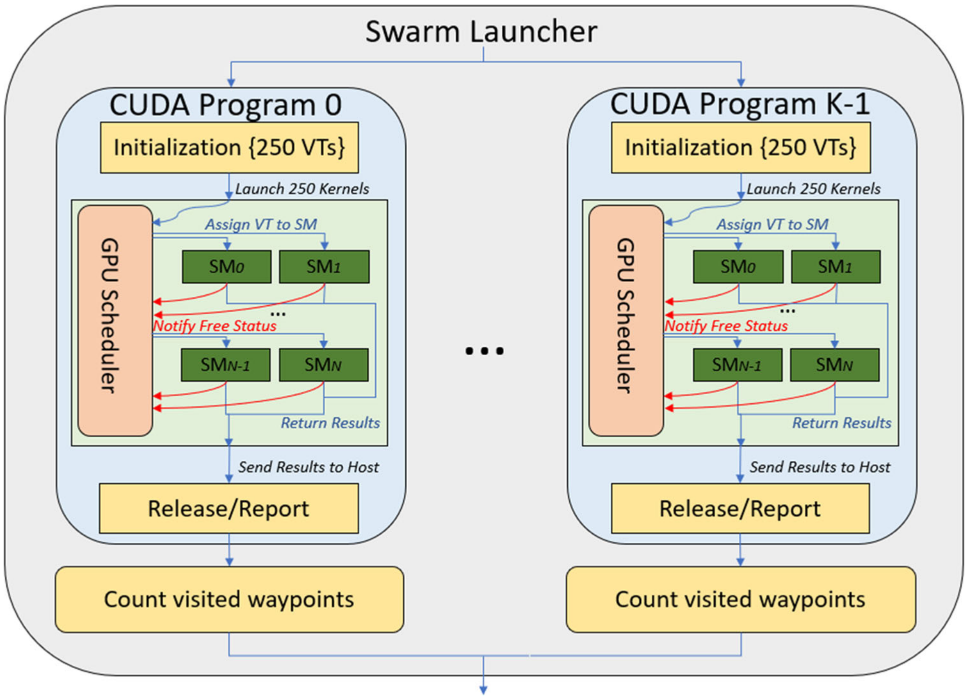

Fig. 4 Control flow for Grapple with 250*K VTs

of the hash table, and the collision-based pruning mentioned described back in Sect. 2.4.

To achieve maximal utilization of SMs and therefore maximal parallelism at the SM-level, VTs are assigned to SMs using pipelining: as soon as a VT completes its execution on a SM, the GPU scheduler replaces it with a new VT, until all VTs within the CUDA program have been executed on some SM. At this point, the host collects the discovered WPs from all 250 VTs and appends this information to a single output file. All data structures on both the GPU and CPU are released, and the program terminates. The output file is read by a sort utility, and current progress reported by the swarm script. The next GPU program is launched, and the process continues until all GPU programs in the swarm are exhausted.

Note that for a single GPU system, a swarm of size 50,000 VTs requires 200 sequentially launched CUDA programs. One of the benefits of Grapple, and SV in general, is that if additional GPUs are available, even on different machines in different locations, these 200 CUDA programs can run in parallel with each other without additional modification. These other GPUs may be heterogeneous, with more memory or more SMs allowing for more VTs per program or more concurrent execution of VTs, respectively.

Due to the abridged nature of VT searches, minute changes in control flow can have a major impact on the set of visited states for each VT. As hash collisions are resolved by dropping the new entry, even differences in the order of constituent operations change the results. To better understand a VT's behavior, we offer in Algorithm 1 a comprehensive breakdown of a VT's main control loop. Furthermore, in Sect. 4, we conduct a series of tests that illuminate the effects of making even minor changes to the code.

Algorithm 1 includes somewhat technical implementation details, but we feel they are necessary for a complete understanding of Grapple. Here state is the current state, table is a hash table of 8-bit integers, and a and b are random values used for hashing. Grapple uses a single bit of information to represent each visited state, but the minimal addressable space is a byte, so we need to store and retrieve states with bit manipulation. Hashed state vectors are divided into selection, an integer location in table, and sel, a bit selection within an 8-bit integer. When the current contents of the table are queried, visited_state is returned as an 8-bit integer and sel determines the desired bit.

The nondeterministic choice (NDC) has a variety of different implementation options. Traditionally, all nondeterministic options would be accessed in order as in standard BFS (parallel BFS in this case) behavior. With minor modification, all nondeterministic options can be visited in random order. To minimize the amount of branching logic, all NDC order possibilities are enumerated in constant memory, and the selection of order is completely random for each step in the loop.

The NDC loop can be removed through the use of a new search strategy: Parallel Deep Search (PDS). Each Grapple VT with PDS randomly selects one option per nondeterministic choice, discarding all other paths. PDS is a heuristic designed to reach deeper states than parallel BFS on the full, nondeterministic model. Since discarded branches cannot be

Algorithm 1 State-Space Exploration Loop executed by each VT thread

```
Each thread $i$ of a VT's $N$ parallel threads:

while output queues $[i][0] \cdots [i][N-1]$ are non-empty do
  for input queues $[0][i] \cdots [N-1][i]$ do
    while input queue $[j][i]$ is non-empty do
      for all processes in the model do
        for all nondeterministic choices NDC
        within a process do
          successor = successor_generation(process,
            NDC, state);
          selection = (mix($a$, $b$, state));
          hashed_value = (selection/8) % table_size;
          sel = selection%8;
          visited_state = table[hashed_value];
          table[hashed_value] |= (1«sel);
          if (visited_state &(1 «sel)) == 0 then
            Report state back to CPU for check
              against 100 WPs
            Pick random thread $i' \in N$ to
              output to
            if $[i][i']$ has slots then
              Insert the new state into queue $[i][i']$
            end if//implicit else drop the state
          end if
        end for//close for (NDC)
      end for//close for (process)
    end while
  end for
  __syncthreads();
  Check output queues for emptiness
end while
```

explored, there is nowhere for the search to go but deeper. While each VT with PDS cannot explore the discarded paths, the combined swarm still achieves highly effective state-space coverage. PDS can only be used in SV environments like Grapple.

As described in Sect. 2.3, Grapple VTs use a set of NxN queue structures to allow lock-free communication between threads. Each thread has a set of N input queues and N output queues, with I slots in each queue. We call an NxNxI set of queues a queue structure. In Sect. 4, we consider a queue structure in Grapple to be the same as a queue in SPIN and FPGA VTs. For this to hold, I will often be as small as four or five slots.

In Grapple, the input and output queue structures are sets of pointers to a single array in GPU global memory. To avoid illegal memory access, a VT must first check that there are slots available when attempting to insert a new state. In Algorithm 1, this check happens after a state is marked visited. If there are no queue slots available, the state is dropped and its successors potentially lost. If instead the queue check happens before the state is marked visited, the same state (or a state with the same hash value) can be visited later. This second style of check is used in FPGA and Grapple VTs.

The logic employed with this check also plays a factor in Grapple's performance. If the check prevents writing outside the bounds of the underlying array structure, hence referred to as the old guard, it will still allow threads to write to unintended targets. A stricter boundary check, the new guard, enforces the local limitation of I. Both guards will result in state drops when the queue is full, but with the new guard keeping the newer state in the resulting collision, and the old guard favoring the opposite. In practice (see Sect. 4), VTs with the old guard have better performance.

All discussion of dropped states to this point has been about random drops or partial-match drops (hash collisions). It is also possible to do complete explicit-state drops for specific state-vector matches. The default behavior of the FPGA swarm is to consider WPs to be violations. When one of these states is encountered, it is reported and dropped without generating successors. While for other models, this behavior may lead to unreachable portions of the state space, this is not the case for the WP model. Our Grapple tests include variants with and without this WP dropping.

4 Experimental results

In this section, we present experimental results for Grapple. The first set of experiments use the WP benchmark to test variants of the Grapple VT design, and allow us to compare performance with the SPIN [25] and FPGA [12] swarms, as well as with our non-swarm GPU implementation [7]. All of these tests use the same 100 WPs, selected from a random distribution over the 32-bit integer space.

The GPU used in these experiments is an Nvidia Geforce 660 Ti GPU with 2 GB GPU global memory, and 7 SMs. This is an older, inexpensive GPU model, but one that still allows us to demonstrate Grapple's performance benefits. SPIN experiments run Swarm 3.2 with SPIN 6.4.7, using an Intel dual-socket server that has two Xeon E5-2670v3 CPUs (24 cores total) running at 2.3GHz and Hyper-Threading enabled (48 hardware threads total), with 128 GB of RAM. FPGA experiments are done with cycle-accurate SystemC simulations using Xilinx Vivado HLS 2017.4, targeting a Xilinx Virtex-7 XC7V690T FFG1761-3 FPGA.

The test environments for the SPIN and FPGA experiments are the same as in [12]. Additionally, we include experiments using the Dining Philosopher's problem in order to demonstrate Grapple's ability to discover a known deadlock violation. Grapple results for the other BEEM models considered in [7] are also included. These results prompted an additional set of tests focused on model structure. Finally, we show Grapple's potential in a high-performance environment by running WP benchmark tests on Amazon's EC2 GPU cloud platform [2].

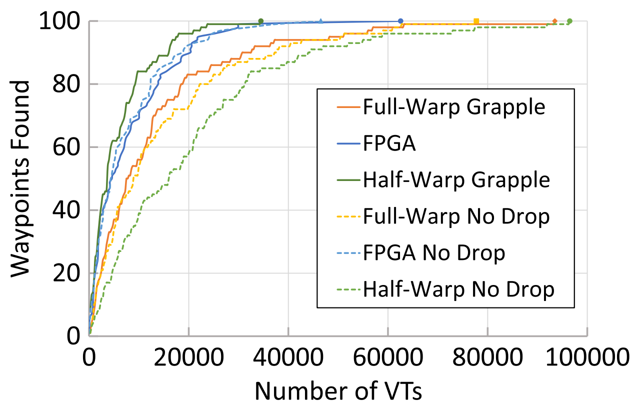

Fig. 5 Grapple VT versus FPGA VT

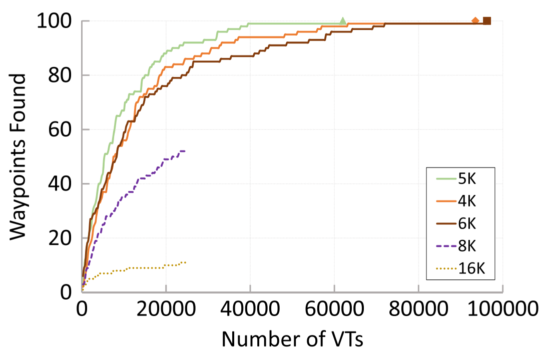

Fig. 6 Impact of frontier size on Grapple search

4.1 WP benchmark

FPGA experiments use internally sequential VTs with 48KB of storage each. The FPGA runs in batches of 44 concurrent VTs, starting a new batch when the previous one finishes. Unlike the general-purpose VT designs of the GPU and CPU swarms, which can be applied to any Promela model, the FPGA swarm is currently limited (hardwired) to the 32-bit random number generator. Fortunately, there are still some variants of this WP benchmark to test against.

Figure 5 shows combined results of two FPGA swarm variants and three Grapple variants running the WP benchmark. In the standard configuration, WPs are recorded upon discovery, considered a violation, and the state is dropped. Non-WP states first check the queue, and are marked visited and propagate if there are slots available or drop and remain unvisited if the queue is full. This allows the state (or a colliding state) to potentially be visited later by the same VT. In later Grapple tests, we refer to this control flow as “FPGA-style”, as it matches the behavior of VTs in [12].

Half-warp (16 threads per VT) Grapple leads the FPGA in number of WPs from the very beginning (in terms of number of VTs), and reaches the 100^{th} WP in 34,500 VTs, over 28,000 fewer VTs than its FPGA counterpart. The full-warp (32 threads per VT) Grapple implementation, however, is outpaced by the FPGA. The FPGA completes the WP benchmark in 30,947 fewer VTs. While these three versions share the same control flow and queue structure size (4,096 entries), the half-warp Grapple implementation has much better performance when using the WP/VT metric. In terms of raw speed, however, the half-warp version is slower, with VTs lasting 650ms compared to the full-warp’s average of 451 ms. Both Grapple versions cannot match the hardware-level speed of the FPGA implementation, but Grapple offers fast VTs with a much easier deployment process than the FPGA swarm.

There is an alternate control flow wherein the 100 WPs are reported but otherwise treated like any other state. In this case, all 100 are discovered by the FPGA in 46,515 VTs or roughly 74.4% the number of VTs as the previous version. Full-warp Grapple also sees improvement, completing in 77,750 VTs. This is not significant enough to catch up with the FPGA or half-warp Grapple. The no-drop version of half-warp Grapple is the one instance where performance dips, becoming the slowest group throughout the search. This may indicate that whatever gave the FPGA-style half-warp version of Grapple such a large performance advantage is something unique to the interaction between the problem topography and its search strategy, instead of a property of half-warp Grapple more generally.

On FPGA hardware, the swarms from Fig. 5 complete in an extremely fast 12.5 s for the original and 9.3 s for no-drop, with individual VTs lasting only ~0.2 ms. The speed of the FPGA implementation is due to it being akin to creating hardware specifically designed to perform the model checking task. Since the“hardwar” is specifically tailored to the implementation, an absolute minimum number of clock cycles are used. These swarms, however, were run on a cycle-accurate FPGA simulator, where one second of simulated time takes approximately one hour of wall-clock time. The simulation allows for more useful data collection without harming FPGA performance, and is cheaper and faster than deploying to a physical FPGA.

While speed can vary quite a bit between VT configurations, the execution time of each CUDA program with the same configuration is relatively stable. Throughout testing, running an initial batch of three programs (750 VTs) would give a good linear approximation for a batch of hundreds of programs. For our test machine, a batch of 100,000 VTs (400 CUDA programs) would complete in 3~5 hours, depending on VT configuration. This can be sped up considerably by using more advanced hardware, as will be shown in Sect. 4.4.

Figure 6 shows the impact of the queue structure size on Grapple’s performance. This test was inspired by the WP/VT difference between earlier half-warp vs full-warp tests. For the same size queue structure (N×N×I), a Grapple half-warp VT has more slots per thread (a smaller N value means a

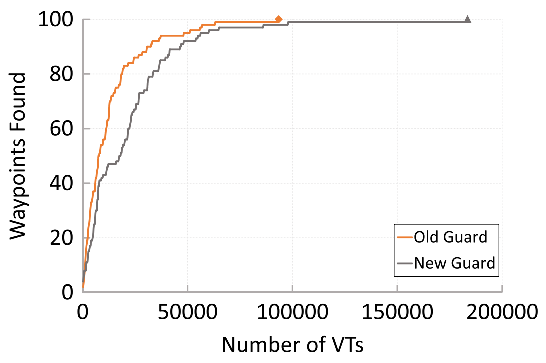

Fig. 7 Impact of guard logic change on Grapple search

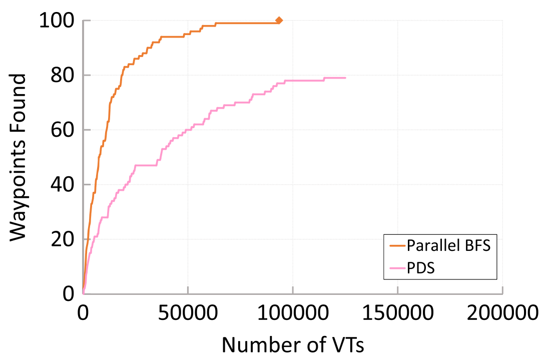

Fig. 8 Grapple VT with and without PDS

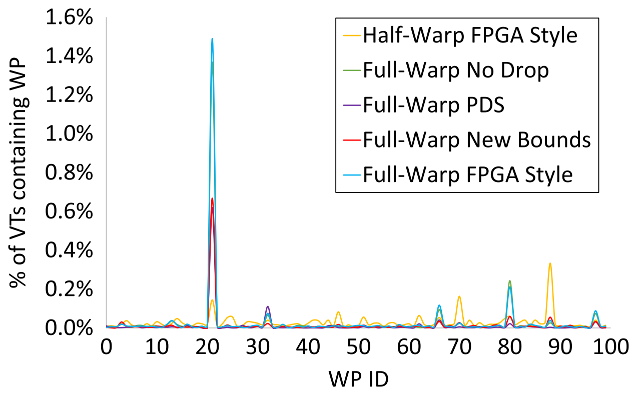

Fig. 9 Percentage of Grapple VTs that find each WP

larger I value). Since the number of slots can impact state-drops (see Sect. 3), we ran a series of tests expanding the queue structure size (and thus the I value) for full-warp Grapple. When $I = 16$ or $I = 8$ (16,384 or 8192 total queue structure size), by 25,000 VTs we determined that these versions would not outperform the $I = 4$ control and terminated the swarms. $I = 6$ performs just slightly worse than the control. Grapple achieved peak performance with $I = 5$ (5120 queue structure size), reaching 100 WPs in 62,000 VTs. This is better than the 93,500 VTs of the control, but still worse than the 34,500 of half-warp Grapple. Since half-warp Grapple uses a queue structure of 4096 elements ($I = 16$ with $N = 16$), but outperforms all full-warp versions in WP/VT, the difference in performance requires further study. It is likely due to a low-level bottleneck, such as register access patterns or to differences in exploration order arising from the fewer random thread options.

We also tested the impact of altering the guard logic for full-warp Grapple's queue structure, as explained in Sect. 3. Both versions use a queue structure with 4096 entries, and otherwise identical control flow. Figure 7 shows the old guard logic maintaining a WP lead throughout the lifetime of the swarm, reaching the 100^{th} WP in 93,500 VTs. The new guard logic takes an additional 90,000 VTs to find all 100 WPs, with $\sim$47% of the search spent looking for the final WP.

In Sect. 3, we introduced the new search strategy *Parallel Deep Search* (PDS), which we designed to reach deeper states than parallel BFS on nondeterministic models. Figure 8 compares Grapple's performance (in terms of number of VTs versus number of waypoints found) using PDS and using Grapple's default behavior (parallel BFS with random ND order). The VTs employing PDS have worse WP/VT performance, reaching only 79 WPs in 125,000 VTs. Since PDS's performance is greatly impacted by state-space topography, it is possible that this particular model may not be suited for exploration via PDS.

Another option is to run PDS on a portion of Grapple VTs, instead of the full swarm. Our data indicates that certain search strategies reach particular WPs more often than others. These ideas and others for PDS are explored later in this section.

Figure 9 shows how often particular WPs were discovered for several search strategies. The number of occurrences are presented as a percent of the total number of Grapple VTs run with that configuration. While most WPs appear in less than 0.2% of VT searches, some are found much more frequently, leading to peaks in the figure. While WPs are randomly distributed across the 32-bit integer space, each VT begins its search at the same initial state of 0. It should come as no surprise that even with diversification techniques, some states are accessed more often than others.

The more interesting case is when a search strategy has a significantly higher rate of discovery for a particular WP than its peers. "Full-Warp FPGA style" and "Full-Warp No Drop" both have greater occurrences of WP 21 (177,865,216) and WP 80 (1,161,888,038). "Half-Warp FPGA Style" more frequently discovers WP 70 (4250701303) and WP 88 (3,304, 030,197). Finally, "Full-Warp PDS" finds WP 34 (3,392,086, 205) more often than other configurations. This chart is useful for constructing Grapple swarms out of multiple VT variations, as it is frequently the case that a large portion of the search time is spent attempting to find one final WP.

The WP percentage statistics are applied to the frontier-size variations given in Fig. 10. The peaks in this figure

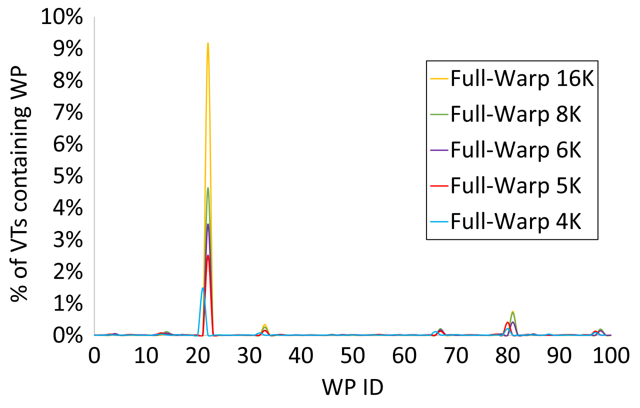

Fig. 10 Percentage of Grapple VTs that find each WP for different queue structure sizes

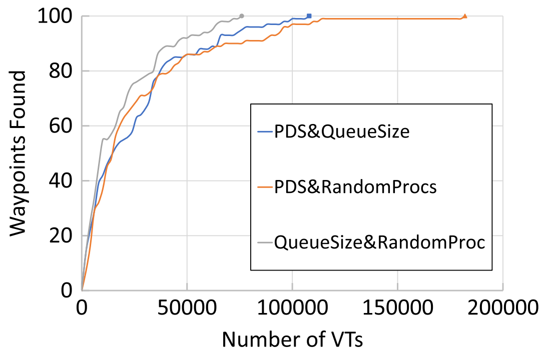

Fig. 11 Three Grapple swarms with multiple diversification techniques

(particularly WP 21 and 80) are common across all five variants. This is not surprising as these tests all use the same search strategy, just with different queue sizes. A point of interest in this figure is that WP 21 has an occurrence rate of 9.176% for the 16K variant, compared to a maximum of ∼1.5% in Fig. 9. It should be noted that the 16K and 8K variants both use data from truncated swarms (25,000 VTs each), but this rate is still very high.

These WP statistics motivate additional experiments on mixed-strategy swarms. Figure 11 presents results for three such swarms, each with a slightly different configuration. All three use eight search strategies, including: half- and full-warp "FPGA-style", and half- and full-warp "no-drop". The remaining four strategy-slots feature either full-warp 5K and full-warp 6K frontier-size variations, half- and full-warp PDS, or half- and full-warp *process randomization*.

The best-performing configuration so far has included the frontier variants and process randomization, but not PDS. Breaking down results by technique, the half-warp FPGA-style and full-warp 5k/6k frontier versions find the greatest number of WPs. Interestingly, the "no-drop" variants visit fewer WPs. This would seem to contradict our earlier experiments, but that is not the case. In those same experiments, the "no-drop" WP count would normally start behind that of their FPGA-style counterparts, but would surpass them later on. It was often the case, as might be expected, that the total search time would be dominated by the last several WPs. The "no-drop" versions tended to experience the most success in this dominating period.

Since in the mixed environment the performance of a single technique is less important than the spread of the search, we analyzed combinations of techniques aimed at eliminating redundancies. This analysis led to the top-performing frontier/process randomization combination. Analysis of that configuration suggests that not including the "no-drop" technique would be favorable in the mixed environment. Since each technique does not reach the final dominating period on its own, the benefit of the "no-drop" variant never achieves fruition. It would be more prudent to replace those strategy-slots with either additional copies of the top-performers, or to introduce additional techniques with further perturbations in the exploration order.

Unlike in Grapple and the FPGA tests, where the hash table size is always 48 KB per VT, SPIN swarm experiments run on a variety of different hash table sizes. While a 48 KB hash table would be ideal for comparison purposes, a SPIN swarm requires hash tables to be multiples of 32. With the table size set to 32 KB, SPIN ran for over a week without discovering all 100 WPs, after which we terminated the search. The next step up, with 64 KB-hash-table VTs, managed to find 90 WPs in 263,220 s (just over three days). As in [12], the optimal configuration for the SPIN swarm seems to be a 256 MB table per VT. This version uncovers all 100 WPs in 10,890 s, $\sim 3.4\times$ as long as half-warp Grapple or $\sim 1.8\times$ as long as full-warp Grapple.

The optimal setting for SPIN VTs requires over $5000\times$ the amount of memory per VT as Grapple and the FPGA. A larger memory footprint for each VT lets a VT cover a greater portion of the state-space, but at the cost of a longer execution time per VT. The SPIN results suggest that either the overhead for creating many small SPIN VTs hinders their effectiveness, or that SPIN's implementation of diversification techniques favor larger VTs. While SPIN could run more concurrent VTs if more machines were available, thereby improving performance, this is also true for the Grapple and FPGA versions.

Non-swarm GPU tests were difficult for the WP model. Our original implementation in [7] required four full explicit-state cuckoo hash tables to contain every possible state vector. Although the WP benchmark uses randomly generated 32-bit states, the states are still wrapped in a 64-bit unsigned long long integer. Following the original MC design, the total hash storage alone would be 128 GB, much larger than the 2 GB of global memory on this GPU. Converting this checker to bitstate hashing allows us to cut the hash storage to a morereasonable 500 MB. This, however, does not account for the other support structures that still use full 64-bit state vectors.

The simplest solution is to run a version that is 250× the size of a single Grapple VT, since we know that 250 Grapple VTs can be allocated in one CUDA program without exhausting memory. A table this size can hold just over 98 million states, a fraction of the state-space generated by the WP benchmark. Our non-swarm checker explores this space in 352 s, reaching 10 WPs. As a standalone program, the GPU MC clearly cannot compete with the full state-space exploration of a Grapple swarm.

4.2 BEEM models

Table 1 contains results for Dining Philosophers, where each philosopher picks up the left stick, then the right, releases the left and then the right. There is a violating state (deadlock) when all philosophers pick up their respective left stick concurrently. The minimum number of VTs tested is 7, since less than 7 would take the same amount of time to run on this GPU. For versions with more processes, we use sets of 451 VTs (an arbitrary large number that fits within the GPU memory footprint), but for DP10 and DP11, we determined that more precision would be better than just saying $\times \leq 451$.

The number of VTs needed to fully explore the state space increases dramatically when the number of processes is increased to 12. This is as expected, as DP11 has 177,146 states to fit into 392,800 slots per VT (∼45% occupancy), while DP12 has 531,440 states to fit into the same number of slots (∼135% occupancy). Beyond 12, we prematurely terminate the search due to the low rate of new state discovery.

The final column of Table 1 shows the percentage of the state space covered in the first 451 VTs. Due to search overlap, the number of unique states visited grows logarithmically with the number of VTs. The effect is more pronounced in a deterministic model like Dining Philosophers, since the only source of diversification in Grapple for such models is the difference in hash polynomial among VTs.

An important factor in model checking is the discovery and recording of paths to a violation. To maintain such debugging information, a Grapple VT requires one small modification. When a newly generated state s' is added to a counting table to be sent back to the host, the parent state s is added to a second table at the same address. If a violation is found for a state in the counting table, the parent is taken from the second table and used to recover its address in the counting table. This process repeats for the parent of s and its ancestors until reaching the initial state.

In Table 2, we introduce a diversification technique for deterministic models: process randomization. This works similarly to nondeterministic choice (NDC) randomization, with a random process order selected from a table in constant memory. In fact, we utilize the exact same table as for NDCs, combining multiple selections to cover larger numbers of processes. The first clear difference between Tables 1 and 2 is that the latter has a much lower percentage of VTs

Table 1 Dining philosophers model in Grapple

Number of processes	% of VTs finding violation	Average VT execution time	State space size	# of VTs to explore	% of state space covered by first 451 VTs
10	67.72	195 ms	59,048	100% in 7	100
11	46.65	366 ms	177,146	100% in 14	100
12	25.55	677 ms	531,440	100% in 3157	99.99
13	13.75	832 ms	1,594,322	99.21% in 24,805	98.65
14	11.18	882 ms	4,782,968	97.76% in 13,530	92.72
15	11.35	902 ms	14,348,906	93.19% in 50,061	76.56

Table 2 Dining philosophers model with random process order

Number of processes	% of VTs finding violation	Average VT execution time	State space size	# of VTs to explore	% of state space covered by first 451 VTs
10	4.60	180 ms	59,048	100% in 42	100
11	5.11	297 ms	177,146	100% in 91	100
12	12.89	715 ms	531,440	100% in 5863	99.99
13	1.17	810 ms	1,594,322	99.13% in 13,530	97.77
14	0.92	928 ms	4,782,968	98.47% in 13,530	96.84
15	1.56	966 ms	14,348,906	92.52% in 13,530	78.8

Table 3 Dining philosophers model with process-PDS

Number of processes	% of VTs finding violation	Average VT execution time	State space size	# of VTs to explore	% of state space covered by first 451 VTs
10	54.86	556 ms	59,048	100% in 6765	99.97
11	39.00	895 ms	177,146	99.99% in 13,530	99.93
12	20.20	1623 ms	531,440	99.93% in 13,530	99.44
13	11.75	2034 ms	1,594,322	98.71% in 13,530	95.16
14	5.73	2276 ms	4,782,968	96.27% in 13,530	82.77
15	4.99	2399 ms	14,348,906	88.77% in 13,530	64.62

discovering violations. While this may seem detrimental, as finding violations is a goal in model checking, it is actually a positive indicator of a spread-out state space.

Ideally, a swarm without overlap would visit all states, including violating states, exactly once. Looking at the last two columns of both tables, the models with randomized process order tend to have higher coverage in the first 451 VTs, and maintain this higher coverage with fewer VTs (not shown). While the state-space spread is still not ideal, with only 92.52% coverage for DP15 using 13,530 VTs, it takes only about one fourth the number of VTs to be half a percentage point behind the non-random version. Minimization of state-space overlap is the key issue when selecting diversification techniques for a swarm.

Once process randomization was in place, it was not difficult to modify the search strategy to behave similarly to PDS. Table 3 contains the results for this modification. Like process randomization, this process-PDS has a lower percentage of VTs discovering a violation, but the reduction is not as dramatic. It does not, however, have the same coverage benefits of process randomization, exhibiting less coverage than the default case. The average VT execution time is also much higher than in either of the previous cases.

This phenomenon is easy to account for: the process-PDS version of Grapple is reaching deeper states as a DFS would, but unlike DFS it does not explore the "non-first" paths. This can be useful for finding deep violations, but it is not particularly suited for complete exploration. It is likely that a depth-limited version of process-PDS could reach deep states without incurring a penalty in execution time. We plan to pursue this direction as part of future work. For now, process-PDS can be used in conjunction with other, more exhaustive VT configurations.

As Grapple builds upon the foundations of [7], the models used in that work can be used as is with Grapple. For models with state spaces smaller than the hash table size, full coverage is achieved by the first batch of VTs. These include Anderson 2 and 4, and Peterson 1–3. When the state space grows even slightly larger than the table capacity, coverage growth slows dramatically. Anderson 3, with a state space of approximately 52.5 million states, reaches just over 3.4 million (6.52%) in the first 451 VTs. Even after nearly 30,000 VTs, only $\sim$ 21% of the state space is covered. The Peterson 4 model, at an even smaller $\sim$ 1.1 million states, achieves approximately 59.55% coverage in the first 451 VTs, and $\sim$ 75.85% after 6765 VTs.

The important factor here is that the full state space of Peterson 4 is less than 3× the size of a VT hash table. While one might suspect an issue with Grapple's scalablity, that is not the problem here. The first 451 VTs of Anderson 3 reach more unique states than the full space of Peterson 4. Furthermore, Grapple had no issue reaching the over 4 billion states in the WP benchmark. We believe there is a fundamental structural difference between the WP benchmark and the BEEM models, and explore this conjecture in the following subsection.

4.3 Structure-oriented results

Our results from Sect. 4.2 on the BEEM models suggest a fundamental structural difference in the WP model compared to the BEEM models, resulting in very different Grapple performance. Another contributing factor may be the extremely small hash tables used in Grapple VTs, compared to e.g., the larger structures favored by SPIN's swarm.

One structural distinction is that the WP model has a high degree of connectivity. That is, not only does each state have a large number of outward edges (children), the minimum edge distance between any two states in the model is relatively small, and the number of paths between any two states is very high.

Models that are either generally linear or have "bottleneck structures" may be less suited for our initial set of swarm strategies. We consider a state space to be generally linear if the average number of edges per state is close to two (one inward and one outward edge). A state space is said to have a bottleneck structure if there is any single state or small group of states, other than the initial state, that the model checker must pass through to reach a large percentage of said state

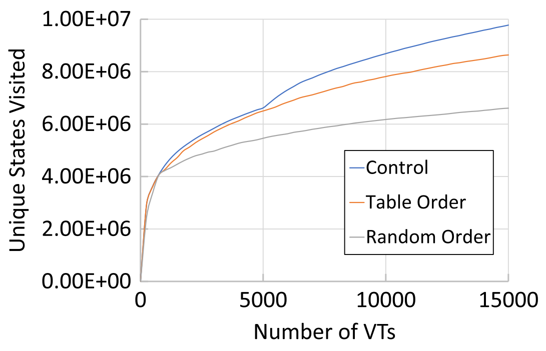

Fig. 12 Two-phase Grapple search for Anderson 3

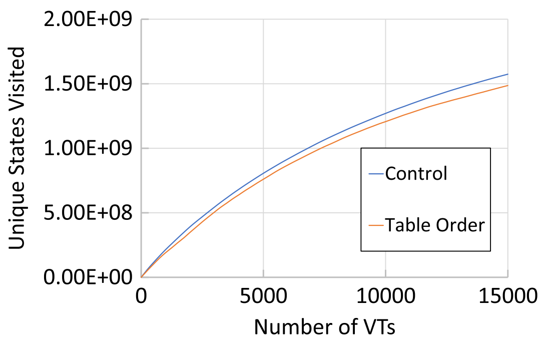

Fig. 13 Two-phase Grapple search for WP Model

space. Models with either of these issues is considered to have "low connectivity".

We designed Grapple tests to benchmark performance on models with low connectivity. Since connectivity, as we have defined it, can impact a VT's performance in terms of reachability of portions of the state space, VTs for models with low connectivity may fill their hash tables before passing through bottlenecks or reaching states further from the root.

One possible solution we have explored is to change a VT's starting state. In a two-phase swarm, a number of VTs first explore the state space as normal and send their complete hash table contents to the host. The contents of these hash tables (representing the set of states visited by "phase one" VTs) are then defined as start states for "phase two" VTs, which otherwise run as normal. As these new initial states were reachable from the root during phase one, any states reached during phase two must also be reachable from the root.

During testing, we use two different selection criteria for new initial states: hash table order and random selection. Hash table order refers to the order of states in a master hash table kept on the host, which combines the tables from the phase-one VTs. The master table is only used for this initial state selection and unique state counting, with no other impact on the normal control flow of VTs. The hash function is simply the full state vector modulo the table size; so the order is numerical, as long as the table is larger than the largest vector.

Initial results given in Figs. 12 and 13 show fastest exploration with the default initial state, and worst-case performance with random selection, with hash table order in between. The random selection was not included for the WP problem, as random selection had worse performance than table order in the Anderson 3 problem results. While other selection criteria may offer better results, we have ceased exploring this direction for the time being.

Another option for this structure-focused exploration is a modified version of PDS with a set depth limit. The depth limitation prevents the time per search from ballooning, but does not necessarily decrease overall state-space coverage. In fact, our results show that the implementation of a depth limit improves coverage of Grapple swarms with PDS enabled, as opposed to PDS without such limits. Truncated results for various models are shown in Tables 4, 5, 6 and 7 and Fig. 14. The best-performing depth limit varies among models and configurations, so a great deal of fine-tuning is possible.

The depth limit improved the performance of PDS, but not to the point where it became competitive with, much less outperforming, parallel-BFS. With this in mind, we combined the two previous approaches to great success. In what we call scatter PDS, a given VT will search up to a specified depth limit using PDS or process-PDS, and then switch to parallel-BFS to complete the search before returning to the host. Unlike the two-phase swarm, both phases are self-contained within a VT.

The term "scatter" is meant to signify the search first exploring deep into the state space and then spreading out at the reached depth. In the worst-case scenario, swarms with scatter PDS are comparable in performance to the control (parallel-BFS), with certain depth selections for a given model configuration visibly outperforming said control.

The results displayed in Figs. 15, 16, 17 and 18 compare control cases with the best depth limit for each model. In all cases, we used process-PDS for the initial portion of the search, to simplify the experimental process. We collected data for each model with multiple depth limits. Since the performance roughly fell between the two selections already graphed, such data would only decrease graph legibility. While performance gain is not large, small gains in efficiency can have a significant impact at scale.

Figure 16 appears to be linear, but this is not quite the case. All of the scatter graphs include data up to 3,000 VTs for comparative purposes, and the searches are all logarithmic. For the WP model, 3000 VTs covers only the fast-growth portion of the logarithmic graph, before the search slows. While the size of the state space largely dictates the shape of these graphs, connectivity may also play a role. Given models with state spaces of equal size, the graph of the low connectivity model should begin to curve earlier in the exploration pro-

Table 4 Anderson 3 Model with depth-limited process-PDS

Number of VTs	DL: 25	DL: 50	DL: 100	DL: 150	DL: 200	DL: 400
21	8486	75,422	317,162	919,529	1,232,234	1,219,168
42	8486	76,056	343,691	1,286,814	1,618,478	1,575,375
63	8486	76,373	356,905	1,475,900	1,860,234	1,842,690

Table 5 WP model with depth-limited PDS

Number of VTs	DL: 10	DL: 20	DL: 25	DL: 50	DL: 100	DL: 200
21	1,568,791	3,069,066	3,312,712	3,274,978	3,294,512	3,259,563
42	2,962,603	6,148,748	6,504,026	6,645,742	6,693,707	6,619,897
63	4,167,890	9,027,792	9,599,406	10,112,060	9,925,403	9,987,968

Table 6 WP model with depth-limited process-PDS

Number of VTs	DL: 25	DL: 35	DL: 40	DL: 50	DL: 55	DL: 60	DL: 75	DL: 100	No limit
21	5,754,810	5,752,133	6,110,764	6,112,610	5,757,537	5,758,393	5,754,157	5,760,098	5,381,120
42	11,422,827	11,414,222	12,119,298	11,773,368	11,421,705	12,129,924	12,125,604	11,783,131	10,681,600
63	17,028,839	17,722,132	17,717,586	17,373,677	17,726,645	17,726,195	17,721,424	17,035,476	15,905,315

Table 7 Peterson 3 model with depth-limited PDS

Number of VTs	DL: 25	DL: 30	DL: 40	DL: 45	DL: 50	DL: 100
21	65,931	114,363	148,540	152,231	151,705	153,346
42	68,312	118,708	156,753	158,228	159,550	159,119
63	69,129	121,161	159,828	160,725	161,665	161,612

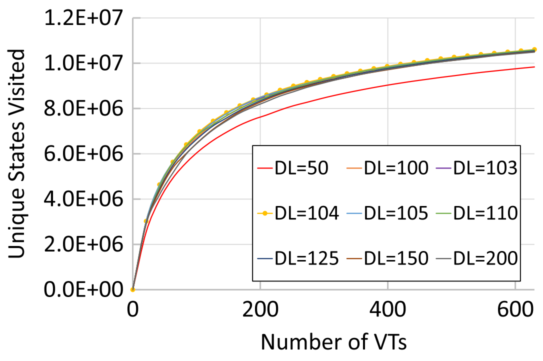

Fig. 14 DP 15 model with depth-limited process-PDS

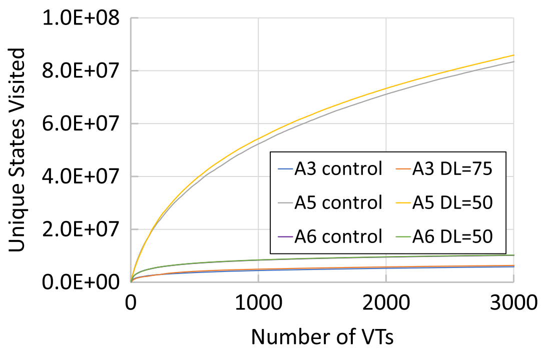

Fig. 15 Scatter Grapple search for Anderson models

cess. While the number of yet unreached states is the same, it is easier to reach any state in a high connectivity model, extending the fast-growth phase of the logarithmic graph.

4.4 Large-scale results on the cloud

For our large-scale experiments, we used two Amazon EC2 nodes [2], one with four and one with eight Tesla V100 devices. Each device features 16GB of global memory and 80 SMs. All devices for each configuration run concurrently and their reported WPs are collected by a script on the host. As in the previous tests, each VT is independent and features data structures private to said VT. There is no inter-GPU communication other than WP counting by the script. Each CUDA program runs 2 000 VTs between reports to the host.

As in Fig. 19, the 4-GPU node reaches all 100 WPs in 72,000 VTs (18,000 per GPU). The 8-GPU node reaches all 100 in 80,000 VTs (10,000 per GPU). Even with state-recording overhead, they complete in 42 min and 21 min, respectively. This is faster than our previous results with such recording disabled. Turning off state-recording results in a

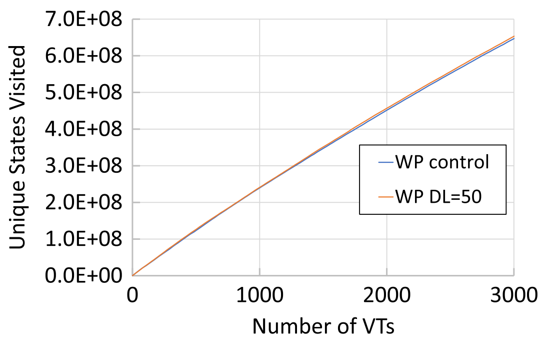

Fig. 16 Scatter Grapple search for WP model

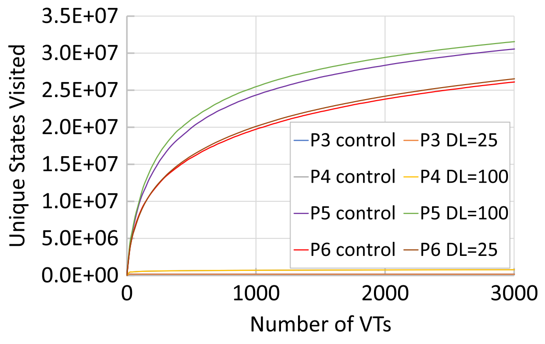

Fig. 17 Scatter Grapple search for Peterson models

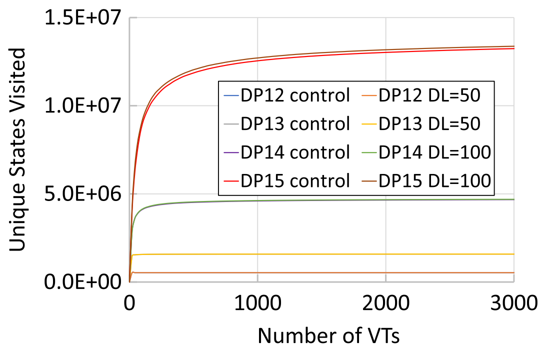

Fig. 18 Scatter Grapple search for DP models

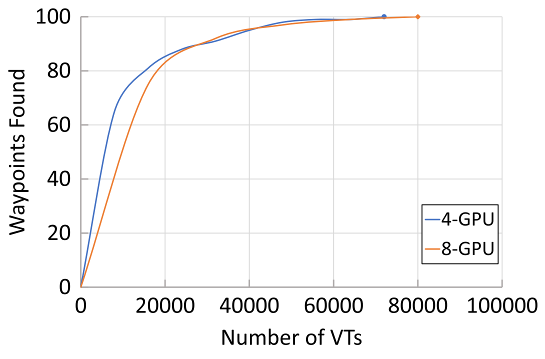

Fig. 19 Grapple with 16 threads/VT on amazon EC2

reduction of average VT time from 1.02250 s to 203.51 ms. This is a significant reduction of 80.1%.

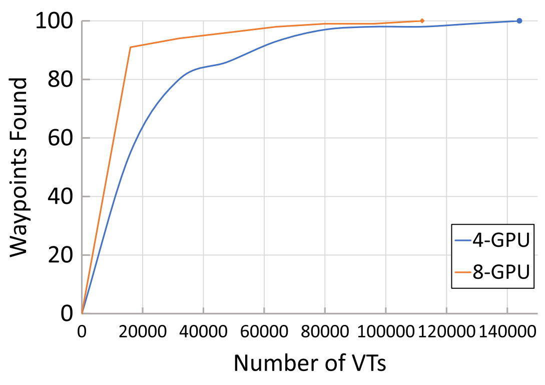

Fig. 20 Grapple with diverse swarm on Amazon EC2

Figure 20 shows a variant of the previous test, but with diverse swarms. The configurations are derived from the best-performing options in our previous diversity test: altering queue size and adding process randomization. Both the 4-GPU and 8-GPU variations required more VTs than in their non-diverse counterparts. With that said, the diverse 8-GPU configuration reached 91 WPs in its first batch of 16,000 VTs. The original configuration (16 threads/VT with no-drop behavior) reached only 75 WPs in its first batch. This suggests that swarm diversity can help hasten exploration, but our configuration needs more tuning to achieve the best results.

5 Related work

In [21], SPIN was extended to support dual-core processors, using nested DFS to check safety and liveness properties. This work was extended to multicore systems for safety properties in [22] and liveness properties in [17]. Despite the earlier debut of a distributed model checker [6], the dual-core version of SPIN was the first parallel MC to reach wide adoption.

Other work sought to avoid the naturally sequential depth-first post-order found in dual-core SPIN's nested DFS algorithm by leveraging the parallelism in breadth-first reachability analysis on both distributed [36] and multi-core systems [5]. This was mainly accomplished using two algorithms: One Way Catch Them Young (OWCTY) and Maximal Accepting Predecessors (MAP). Both algorithms perform parallel reachability analysis, but differ in the way they detect cycles in the state-space graph.

Early GPU-based MC efforts focused on a priori graph exploration, as opposed to generating new states on-the-fly [3,14,19,26,30]. The first on-the-fly GPU-based approach used the GPU to generate new states with enabled transitions, and the CPU for duplicate detection [15]. This is not unlike waypoint counting in Grapple, but their system makes less efficient use of the GPU hardware and is not based on SV.

GPUexplore [39] was introduced in 2014 along with our own GPU-based model checker [7]. While we tried to redesign SPIN to take advantage of the GPU architecture, GPUexplore worked on Labeled Transition Systems (LTSs) and followed a symbolic approach. Grapple uses VTs based on our 2014 design, so it is still very different than GPUexplore. A GPU-based on-the-fly reachability checking system for LTSs that achieved $50--100\times$ performance over sequential search was presented in [41].

In [37], GPUs were used for strong and branching bisimilarity checking. A GPU-based method for liveness checking for finite-state concurrent system appeared in [38]. Three partial-order reduction algorithms were implemented on the GPU in [31], bringing GPUexplore closer to parity with existing CPU-based checkers. A second version of GPUexplore was released that same year, with improvements made to lock-less hashing and thread synchronization [40]. Unlike [38], this version does not include support for liveness properties.

Scalability tests for GPUexplore were carried out in [8], achieving 5.5 million states/second on a 61.9 million state model. Additionally, they used GPUexplore to pit the 2015 Maxwell Architecture Nvidia Titan X GPU against the 2016 Pascal Titan X GPU, averaging a $1.73\times$ improvement on the new device. A more in-depth comparison between cuckoo hashing and the GPUexplore table was carried out in [10], concluding that cuckoo hashing is $3\times$ faster for random data and up to $9\times$ faster for non-random data.

A GPU-based parameter-synthesis tool for stochastic systems was presented in [11]. Utilizing a single GPU, it achieves up to $31\times$ the performance of sequential approaches. A multi-core version of the LTSMIN model checker [29] outperformed the 2005 multi-core SPIN and the 2008 multi-core DiVinE model checkers. In [16], a new multi-core DFS algorithm called CNDFS with better performance than the OWCTY algorithm was presented. This technique uses a swarm approach with state coloring to perform cycle detection concurrently with state-space exploration. LTSMIN saw further improvements in 2015, including support for new modeling languages [28].

In [18,35], an FPGA was used to accelerate the exploration of a relatively small 10,000-state model, achieving a $50\times$ speed-up compared to its software equivalent. The FPGA swarm of [12], to which this work is compared, achieved a $900\times$ improvement over a SPIN swarm for a model of a much more substantial size (4B+ states). While this scale of improvement is unlikely for a single GPU device, the process of deployment to the FPGA is much more complex compared to the GPU. Additionally, their FPGA swarm was designed specifically for the 32-bit WP model, while Grapple can handle arbitrary Promela models.

6 Conclusions

We have presented Grapple, a new framework for highly efficient, explicit-state model checking on the GPU. Grapple is based on swarm verification (SV), and its features include: a parallel swarm of internally parallel verification tasks (VTs); GPU-optimized implementations of hash functions and bitstate representation of visited states; optimal use of GPU shared memory, thereby eliminating inter-block communication/synchronization overhead; and a new search strategy called Parallel Deep Search, designed to reach deeper states in a VT's exploration graph. Our experimental results show that Grapple outperforms multicore SV [23] and GPU non-SV [7] approaches, and that it uses a number of VTs similar to that required by an FPGA swarm [12].

Future work includes adding support for larger state vectors, allowing us to test Grapple with larger-scale model instances from the BEEM database [4]. We will also expand upon the promising results of "scatter" PDS, and investigate other model-structure-focused techniques for Grapple. Finally, we will increase the scope of our comparative tests to other non-SV parallel model checkers such as GPUexplore [40].

References

1. About CUDA | NVIDIA developer zone. https://developer.nvidia.com/about-cuda
2. Amazon EC2 P3 instances. https://aws.amazon.com/ec2/instance-types/p3/
3. Alcantara, D.A.F.: Efficient hash tables on the GPU, 2011. Copyright-Copyright ProQuest, UMI Dissertations Publishing 2011; Last updated - 2014-01-23; First page - n/a; M3: Ph.D
4. BEEM: BEnchmarks for Explicit Model Checkers-ParaDiSe. http://paradise.fi.muni.cz/beem/
5. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing fast LTL model checking algorithms for many-core GPUs. J. Parallel Distrib. Comput. **72**(9):1083–1097 (2012)
6. Barnat, J., Brim, L., Ročkai, P.: Scalable multi-core LTL model-checking. In: D. Bošnački and S. Edelkamp, editors, Proc. of SPIN 2007: the 14th international SPIN conference on Model checking software, vol. 4595 of Lecture Notes in Computer Science, pages 187–203. Springer Berlin Heidelberg, (2007)
7. Barnat, J., Brim, L., Stříbrná, J.: Distributed LTL model-checking in SPIN. In: Dwyer, M. (ed.) Proc. SPIN 2001: the 8th International SPIN Workshop on Model Checking of Software, volume 2057 of Lecture Notes in Computer Science, pp. 200–216. Springer, Berlin Heidelberg (2001)
8. Bartocci, E., DeFrancisco, R., Smolka, S.A.: Towards a GPGPU-parallel SPIN model checker. In: Proceedings of the 2014 International SPIN Symposium on Model Checking of Software, pp. 87–96. ACM, (2014)
9. CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
10. Cassee, N., Neele, T., Wijs, A.: On the scalability of the GPUexplore explicit-state model checker. In: Proceedings Third Workshop on Graphs as Models (GaM 2017), Uppsala, Sweden, (2017)

11. Cassee, N., Wijs, A.: Analysing the performance of GPU hash tables for state space exploration. Electr. Proc. Theor. Comput. Sci. EPTCS **263**, 1–15 (2017)
12. Češka, M., Pilař, P., Paoletti, N., Brim, L., Kwiatkowska, M.: PRISM-PSY: precise gpu-accelerated parameter synthesis for stochastic systems. In: Chechik, M., Raskin, J.-F.: (eds.), Tools and Algorithms for the Construction and Analysis of Systems, pp. 367–384. Springer, Berlin (2016)
13. Cho, S., Ferdman, M., Milder, P.: FPGASwarm: High throughput model checking using FPGAs. In: 28th International Conference on Field Programmable Logic and Applications (FPL). IEEE, (2018)
14. DeFrancisco, R., Cho, S., Ferdman, M., Smolka, S.A.: Swarm model checking on the GPU. In: Biondi, F., Given-Wilson, T., Legay, A. (eds.) Model Checking Software, pp. 94–113. Springer, Cham (2019)
15. Deng, Y., Wang, B.D., Mu, S.: Taming irregular EDA applications on GPUs. In: Proceedings of the ICCAD '09: the International Conference on Computer-Aided Design, ICCAD '09, pp. 539–546, New York, NY, USA, (2009). ACM
16. Edelkamp, S., Sulewski, D.: Efficient explicit-state model checking on general purpose graphics processors. In: Pol, J., Weber, M. (eds.) Proceedings SPIN'10: the 17th International SPIN Conference on Model Checking Software, vol. 6349 of Lecture Notes in Computer Science, pp. 106–123. Springer, Berlin (2010)
17. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved multi-core nested depth-first search. In: Chakraborty, S., Mukund, M. (eds.) Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, pp. 269–283. Springer, Berlin (2012)
18. Filippidis, I., Holzmann, G.J.: An improvement of the piggyback algorithm for parallel model checking. In: Proceedings of the 2014 International SPIN Symposium on Model Checking of Software, pp. 48–57. ACM, (2014)
19. Fuess, M.E., Leeser, M., Leonard, T.: An FPGA implementation of explicit-state model checking. In: Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines, FCCM '08, pp. 119–126, Washington, DC, USA, (2008). IEEE Computer Society
20. Green 500 | TOP500 supercomputer sites. https://www.top500.org/green500/
21. Harish, P., Narayanan, P.: Accelerating large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V. (eds.) Proc. HiPC'07: the 14th international conference on High performance computing, vol. 4873 of Lecture Notes in Computer Science, pp. 197–208. Springer, Berlin Heidelberg (2007)
22. Holzmann, G., Bošnački, D.: The design of a multicore extension of the SPIN model checker. IEEE Trans. Softw. Eng. **33**(10), 659–674 (2007)
23. Holzmann, G.J.: Parallelizing the SPIN model checker. In: Donaldson, A., Parker, D. (eds.) Proc. SPIN 2012: the 19th International Workshop on SPIN Model Checking Software, vol. 7385 of Lecture Notes in Computer Science, pp. 155–171. Springer, Berlin (2012)
24. Holzmann, G.J.: Cloud-based verification of concurrent software. In: Proceedings of the 2016 International Conference on Verification, Model Checking, and Abstract Interpretation. Springer, Berlin (2016)
25. Holzmann, G.J., Joshi, R., Groce, A.: Swarm verification. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, pp. 1–6, Washington, DC, USA, (2008). IEEE Computer Society
26. Holzmann, G.J., Joshi, R., Groce, A.: Swarm verification techniques. Softw. Eng. IEEE Trans. **37**(6), 845–857 (2011)
27. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating CUDA graph algorithms at maximum warp. In: Proceedings of PPoPP '11: the 16th ACM Symposium on Principles and Practice of Parallel Programming, pp. 267–276, (2011)
28. Jenkins, B.: A hash function for hash table lookup. https://burtleburtle.net/bob/hash/doobs.html
29. 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.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 692–707. Springer, Berlin (2015)
30. Laarman, A., van de Pol, J., Weber, M.: Multi-core LTSmin: marrying modularity and scalability. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, (eds.) NASA Formal Methods, vol. 6617 of Lecture Notes in Computer Science, pp. 506–511. Springer, Berlin (2011)
31. Luo, L., Wong, M., Hwu, W.: An effective GPU implementation of breadth-first search. In: Proceedings of DAC '10: the 47th Design Automation Conference, DAC '10, pp. 52–55, (2010)
32. Neele, T., Wijs, A., Bošnački, D., van de Pol, J.: Partial-order reduction for GPU model checking. In: Automated Technology for Verification and Analysis: 14th International Symposium, ATVA 2016, Chiba, Japan, Proceedings, pp. 357–374. Springer International Publishing, (Oct. 2016)
33. OpenCL technology™- intel.com. http://software.intel.com/OpenCL
34. Spin-formal verification. https://spinroot.com/
35. Tie, M.E.: Accelerating explicit state model checking on an FPGA: PHAST. Master's thesis, Northeastern University, (2012)
36. Verstoep, K., Bal, H., Barnat, J., Brim, L.: Efficient large-scale model checking. In: Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, pp. 1–12, (May 2009)
37. Wijs, A.: GPU accelerated strong and branching bisimilarity checking. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 368–383. Springer, Berlin (2015)
38. Wijs, A.: BFS-based model checking of linear-time properties with an application on GPUs. In: Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, Proceedings, Part II, pp. 472–493. Springer International Publishing, (July 2016)
39. Wijs, A., Bošnački, D.: GPUexplore: Many-core on-the-fly state space exploration using GPUs. In E. Ábráham and K. Havelund, (eds) Proceedings of TACAS 2014: the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems
40. 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: Formal Methods, pp. 694–701. Springer International Publishing, Cham (2016)
41. Wu, Z., Liu, Y., Sun, J., Shi, J., Qin, S.: GPU accelerated on-the-fly reachability checking. In: 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 100–109, (2015)
42. Xiao, S., chun Feng, W.: Inter-block GPU communication via fast barrier synchronization. In: Proceedings of IPDPS 2010: the IEEE International Symposium on Parallel Distributed Processing, pp. 1–12, (April 2010)