Towards a GPGPU-Parallel SPIN Model Checker*

Ezio Bartocci
Vienna University of Technology

Richard DeFrancisco
Stony Brook University

Scott A. Smolka
Stony Brook University

ABSTRACT

As General-Purpose Graphics Processing Units (GPGPUs) become more powerful, they are being used increasingly often in high-performance computing applications. State space exploration, as employed in model-checking and other verification techniques, is a large, complex problem that has successfully been ported to a variety of parallel architectures. Use of the GPU for this purpose, however, has only recently begun to be studied. We show how the 2012 multicore CPU-parallel state-space exploration algorithm of the SPIN model checker can be re-engineered to take advantage of the unique parallel-processing capabilities of the GPGPU architecture, and demonstrate how to overcome the non-trivial design obstacles presented by this task. Our preliminary results demonstrate significant performance improvements over the traditional sequential model checker for state spaces of appreciable size ($>\sim$10 million unique states).

Keywords

Model Checking, SPIN, State Space Exploration, GPGPU, CUDA

1. INTRODUCTION

GPGPUs (General-Purpose Computing on Graphics Processing Units) are increasingly being used in HPC (High-Performance Computing) applications. Their highly parallel structure, improved energy-consumption/performance ratio, and low cost make them more effective than general-purpose CPUs for algorithms that process large blocks of data in parallel. The number of cores available per card is growing at a rate dramatically faster than that for CPUs and, consequently, so are the FLOPs (FLoating Point Operations per Second (FLOPS) that can be executed. Presently, the maximum number of cores in a high-performance CPU is 16 (AMD Opteron), while a desktop GPGPU (NVIDIA GeForce GTX TITAN) may feature up to 2,688 cores. Also, the amount of memory available per GPU, a bottleneck for many applications, has grown considerably over the past several years (i.e., the NVIDIA Tesla K40 now features 12GB of RAM). Furthermore, high-level GPU-based programming languages, such as the *Open Computing Language* (OpenCL) and the *Compute Unified Device Architecture* (CUDA), offer programming interfaces to transform this hardware, originally designed for graphics rendering and fast image processing, into a powerful computing device.

All of the above considerations suggest that this technology can play a substantial role in speeding up state-exploration algorithms. These computation- and memory-intensive techniques are heavily employed in model-checking software to verify that all possible system behaviors meet a particular mission- or safety-critical requirement.

In this paper, we show how Holzmann's parallel BFS algorithm [11], developed for the multicore CPU-based version of SPIN in 2012, can be re-engineered to take significant advantage of the powerful processing capabilities of a GPGPU-based computing platform. To this end, we present a preliminary CUDA-based version of this algorithm for GPUs, and extensively compare its performance against the standard SPIN distribution as well as the 2012 multicore version. We also show how the significant differences between the CPU and GPU architectures lead to correspondingly significant differences in the CUDA-based version. In particular, we tackle the following three problems.

- The BFS algorithm of [11] requires all threads involved in the parallel state-space exploration to synchronize after having generating a new frontier of states on-the-fly. In CUDA, threads are partitioned into blocks and each block is processed independently by a set of cores grouped into one of the GPU stream multiprocessors. CUDA provides a synchronization primitive, but it only works on threads within the same block. Consequently, a GPU *inter-block synchronization mechanism* is required to use more threads than the max block-size limit.

- The execution of a Promela model in SPIN requires *if-else* or *switch* statements, as well as occasional nondeterministic choice of execution paths. *Branch divergence* among threads running concurrently does not affect the speed of multicore CPU-based programs,

*Research supported in part by grants NSF CCF-092619, AFOSR FA0550-09-1-0481, and a NASA NSTRF fellowship (grant #NNX12AN15H).

since each core has its own independent control unit. This is not the case, however, for their execution on GPUs. Since many cores share a common control unit, any form of branch divergence can have serious performance implications. We show that this can be solved by using proper *predication techniques*, where conditional statements are replaced by simple multiplications with the predicates.

- The third challenge is the lack of a suitable hash table in which to store and access visited states in a highly parallel fashion. When we started this project, dynamic memory allocation (standard in CPU-based implementations) on the GPU device was not yet supported in CUDA. As such, in this preliminary version, our implementation relies on Cuckoo hashing, as described in [1, 2] and available in the CUDA Parallel Primitives library.

Our preliminary results are very promising, with speedups of up to 7.26x unique-states-per-second visited over traditional SPIN, and 1.26x that of multicore SPIN. While small problem sizes (state spaces in the low millions or less) can impose significant overhead penalties on our system, our implementation exhibits very good performance for larger problem sizes in the 10-100 million states range. Although there are ultimately limits on the problem sizes we can handle due to limits in GPU memory, this ceiling continues to rapidly increase as available GPU memory continues to double.

The rest of the paper develops along the following lines. Section 2 considers related work. Section 3 provides an overview of the GPU architecture and the CUDA programming model. Section 4 focuses on the design and implementation of our GPU-based approach to state-space exploration. Section 5 presents and compares our experimental results with those for current CPU-based and multicore-based SPIN implementations. Section 6 offers our concluding remarks and directions for future work.

2. RELATED WORK

Parallel state-space exploration and model checking have been active areas of research for over a decade. While much progress has been made, efforts to utilize the low-cost massive parallelism of the GPU to solve these problems remain in their infancy.

In 2005, the SPIN model checker was extended to support dual-core processors, using a nested depth-first search algorithm to check both safety and liveness properties [12]. This approach essentially consists of two searches, where the first expands the state space forward and the second checks backwards edges from previously visited states. A core is dedicated to each search. Scaling support of up to n cores for safety properties was included, but this version required a fair degree of tuning and load balancing that would be eliminated in the 2012 multicore implementation. Although a distributed model checker had previously been developed [6], the dual-core version of SPIN was the first widely adopted parallel model checker.

Brim's research group sought to avoid the *naturally sequential* depth-first postorder found in dual-core SPIN's nested DFS algorithm by leveraging the parallelism in breadth-first reachability analysis on both distributed [18] and multicore systems [4]. They primarily accomplished this via two algorithms: One Way Catch Them Young (OWCTY) and Maximal Accepting Predecessors (MAP). OWCTY works by removing vertices from the graph if they have either no successors or if they cannot reach an accepting vertex. This process is repeated until either no vertices remain, or the remaining vertices all fall on an accepting cycle.

The MAP algorithm is based on the idea that any accepting vertex on an accepting cycle must be its own predecessor. Instead of checking every accepting vertex, representatives are chosen based upon the linear ordering of the vertices and these representatives are checked to see if they lie on an accepting cycle. If all of the representatives fall outside of a cycle, they are removed and the process is repeated until either there are no vertices remaining or a representative falls on an accepting cycle. As both algorithms are constructed by repeatedly performing parallel reachability analysis, any improvements made on parallelizing BFS would *consequently* improve the performance of these liveness-checking algorithms. While we have yet to implement such liveness checks with our new BFS implementation, this remains a strong possibility for our future work.

The 2012 multicore version of SPIN takes off where the 2005 version left off by increasing the performance of parallel breadth-first search, while decreasing the complexity of the algorithm [11]. Holzmann's parallel BFS algorithm (see Algorithm 1) requires two lists and a set of visited states S, which is initially empty. The two lists hold the vertices in the frontier and their successors, respectively. On each iteration of the algorithm, the vertices in the frontier list are expanded and removed from the list and their successors are checked against S to see if they have been visited before. If they are not in S, they are added to S and to the successor list; this is where, in the model checker-specific version of the algorithm, safety properties are checked for violation. Once every member of the vertex list has been expanded and the list is empty, a global synchronization occurs and the successor list and frontier list *switch roles*. This synchronization is a necessary step to guarantee a global breadth-first behavior. The former successor list is now a full frontier and the process can be repeated until exploration is complete.

In addition to being simple to implement, Holzmann's algorithm is inherently parallel and features good load-balancing behavior. Race conditions are avoided by having the lists implemented as source-destination grids and S as a lock-free hash table [15, 16, 17]. Processor core w reads frontier vertices from row w of the frontier list and writes to column w of the successor list. The rows that successors are written to are randomized, and this randomization is what results in *naturally good* load-balancing. Since each core reads and writes to locations that only that core may access, no locks are required for these lists. Our system takes these various features of the 2012 multicore SPIN algorithm and applies them to the GPU architecture.

As for GPU-based state space exploration, until recently, most efforts focused on *a priori* graph exploration, as opposed to generating new states *on-the-fly* [3, 8, 10, 13, 14].

Algorithm 1 PARALLEL BFS ALGORITHM [11]

```
1:  global done = false
2:  global t = 0
3:  global S = {}                                  ▷ statespace set.
4:  global Q[0][1···N][1···N] = {}   ▷ successor set.
5:  global Q[1][1···N][1···N] = {}   ▷ successor set.
6:  global idle[1···N] = false   ▷ all elements.
7:  safety property f
8:
9:  add s_0 to Q[0][1][1] and to S   ▷ initial state.
10:
11: search(w: 1···N)   ▷ N workers.
12:   local ot = t
13:   do
14:     for each q ∈ {1···N}
15:       for each s ∈ Q[t][w][s]
16:         delete s from Q[t][w][s]
17:         for each successor s' of s
18:           if s' ∉ S
19:             add s' to S
20:             if s' violates f
21:               report error
22:             else
23:               w' = choose random 1···N
24:               add s' to Q[1 − t][w'][w]
25:             end if
26:           end if
27:         end for each
28:       end for each
29:     end for each
30:     idle[w] = true   ▷ one element
31:     if (w == i)
32:       wait until all idle[1···N] == true
33:         if (all Q[1 − t][1···N][1···N]empty)
34:           done = true
35:         else
36:           idle[1···N] = false    ▷ all elements
37:           t = 1 − t
38:         end if
39:       end wait
40:     else
41:       wait until t ≠ ot or done
42:         ot = t
43:       end wait
44:     end if
45:   while ! done
46: end search
```

While these approaches provide significant insights into graph programming on the GPU, including uncovering techniques for inter-block synchronization we would use in our own application [21], generally speaking, this is the easiest part of the searching problem. In the case of reachability analysis, for example, knowing the graph a priori means the analysis is essentially complete to begin with.

More recently, there has been work on on-the-fly GPU searches like our own. The first such implementation was a GPU-based bit-state hashing algorithm, which used the GPU to generate new states with enabled transitions, and the CPU for duplicate detection [9]. Our system both generates new states and checks for past visitation without relinquishing control from the GPU. We also use full explicit states with visitation guarantees instead of bit-states, which hold no such claim. The space-saving benefits, however, of bit-state hashing is significant. The larger problem scales enabled by turning away from explicit states in favor of bit-state hashing are very attractive. As such, we will likely include a bit-state option for our system in the future.

Perhaps the most apt comparison to our system would be to the recent work of Wijs and Bošnački [20]. While we both have full state descriptions and are working on the same problem, our methodology is very different. Wijs and Bošnački encode states as an *n-ary* vector of process LTSs, along with a vector of synchronization rules. Each entry in the rule vector is composed of a vector of relevant processes and the result of the rule. We instead use global state vectors like in SPIN, with global variables at the head of the vector followed by access-isolated local-state information for each process. Their transition labels are strings encoded as integers, and use extra bits for items such as "rule applicability" in order to fill out 32-bits. Also, the way they fetch and push states, based upon their hashing structure, does not guarantee strict BFS behavior. Our state generation involves computing all transitions for each global state vector, including those not currently enabled. The result is applied as a minimum sequence of bit masks to the original vector, with disabled transitions automatically computing as the identity mask. This technique removes branching while maintaining the integrity of the new states.

While they initially planned on using the same cuckoo hashing as us [1, 2], their state representation exceeding 64-bits broke the atomicity guarantees of checking and adding to the hash table. Since our global state vectors are 64-bit, we do not have this problem and were able to use the existing tables. Wijs and Bošnački instead created a hashing scheme combining double-hashing with buckets and linear probing, and used the hash table as a vehicle for load balancing. Our load balancing is achieved using the same above-mentioned techniques deployed in the 2012 multicore SPIN implementation. So while we solve the same problem, our two systems are intrinsically different techniques.

3. CUDA PROGRAMMING MODEL

CUDA is a general-purpose parallel-computing architecture and programming model leveraging the parallel compute engine in NVIDIA GPUs. As illustrated in Fig. 1, the GPU architecture is built around a scalable array of N multi-threaded Streaming Multiprocessors (SMs), made up of M Stream Processor (SP) cores ($M = 192$ in the current Kepler NVIDIA cards). Each core is equipped with a fully pipelined integer arithmetic logic unit (ALU) and a floating point unit (FPU) that executes one integer or floating point instruction per clock cycle. 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 the *kernel* that 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.

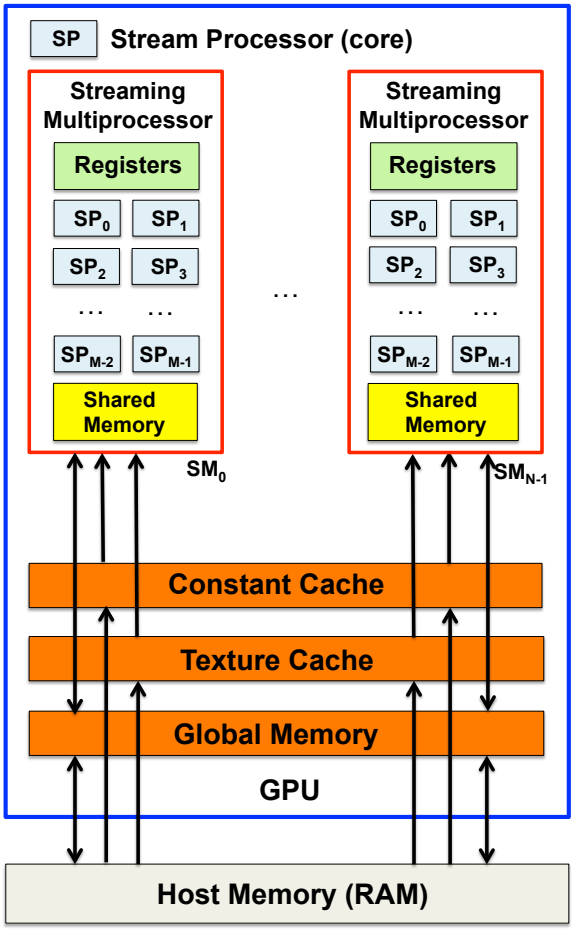

Figure 1: GPU Architecture.

They can also share data during execution. In contrast, threads located in different thread blocks cannot communicate in such ways and essentially operate independently. Although a small number of threads or blocks can be used to execute a kernel, this arrangement would not fully exploit the computing potential of the GPU.

Different types of memory are available for use in CUDA, and their judicious use is key to performance. The most general is global memory, to which all threads have read/write access. The generality of global memory makes its performance less optimized overall, so it is important that access to it be coalesced into a single memory transaction of 32, 64, or 128 bytes. Constant memory is a cached, read-only memory intended for storing constant values that are not updated during execution. All instances of a kernel may access these values regardless of location. Texture memory is another cached, read-only memory that is designed to improve access to data with spatial locality in up to three dimensions. Additional, significantly faster levels of memory are available within an SM, including 16-64KB of registers partitioned among all threads. As such, using a large number of registers within a CUDA kernel will limit the number of threads that can run concurrently. Finally, local memory is invoked when a thread runs out of available registers. CUDA library functions in the host code run on the CPU and administer such tasks as kernel execution and memory management. In addition, each SM has a shared memory region (16-48KB). This level of memory, which can be accessed nearly as quickly as the registers, facilitates communication between threads and can be used as a programmer-controllable memory cache.

4. SYSTEM DESIGN

While fundamentally our work can be seen as a GPU translation of the multicore SPIN algorithm, the particulars of the architecture, as mentioned previously, have had great influence on our overall design. Key considerations have been made to minimize memory footprint, eliminate branching where possible, be conscious of the type of memory used for each structure, and to remove the communication bottleneck with the CPU.

4.1 Non-blocking State Space Exploration on the GPU

Perhaps ”nonblocking” is a misnomer, since, by its nature, Breadth-First Search waits to visit every node at a particular depth before continuing to the next depth. That aside, concurrent systems frequently require locks to any shared data structure to avoid unexpected behavior. Additionally, in the case of the GPU, inter-block communication and synchronization is often achieved through a return over the BUS to the CPU. While we maintain global BFS behavior, the new frontier is divided among threads in a lock-free manner, and our multiple thread blocks are synchronized on the GPU device without the costly return to the host CPU.

As briefly discussed in Section 2, the 2012 multicore version of SPIN [11] uses a source-destination grid in order to have each core assign successors to other cores. Since our goal was originally to replicate multicore SPIN on the GPU, we have adopted this method. The reserved space for each thread allows for work assignment without the need for locking; each thread has exclusive access to where it writes, so there are never any race conditions. Since we maintain global BFS behavior, threads read from and write to our queues at different times, so that potential source of unexpected behavior is also removed. Random assignment of frontier nodes to new threads via these queues gives us the same naturally good load balancing that multicore SPIN enjoys.

Structurally, our algorithm is quite similar to that found in Section 2. Each thread removes a state from its queue, generates successors, checks to see if the successor has been previously visited or violates the property, and if that is not the case, assigns the successors to random new threads for the next round. A ”round” lasts until all the states in each queue have had their successors expanded and assigned to new threads. Then the process repeats with the successor states, and continues to do so until all states have been exhausted. When a round completes, synchronization occurs.

In CUDA, threads within a block can be synchronized with a *__syncthreads()* call, but traditionally the only way to synchronize multiple blocks of threads has been with a return to the host program on the CPU. Since this requires sending data over the BUS on the motherboard, this is a costly operation that is frequently a bottleneck for GPGPU programming. [21] describes a simple technique called *fast-*

barrier synchronization (see Algorithm 2), which lets multiple blocks synchronize with each other while remaining on the GPU. With N blocks, the *0th* thread of each block i sets the ith value of an input array to a common goal value. The first N threads of block 1 continuously check on their corresponding locations in the input array for that goal value, and update the ith location in an output array with the goal value when it is found. The *__syncthreads()* function is called by each thread in block 1 before updating the output array, to keep that block synchronous. The *0th* thread in each block will continue to wait until its value in the output array is the goal, at which point it will call *__syncthreads()* within its block and continue execution. Including this simple function to our code has allowed us to call the kernel once at the start of GPU execution, and continue running on the GPU until the exploration is complete.

Algorithm 2 FAST-BARRIER SYNCHRONIZATION [21]

```
1:  __device__ void syncblocks(int N, int * in, int * out)
    {
        // thread ID in a block
2:      int tid_in_blk = threadIdx.x;
3:      int nBlockNum = gridDim.x;
4:      int bid = blockIdx.x;

        //only thread 0 is used for synchronization
5:      if (tid_in_blk == 0)
        {
6:          in[bid] = N;
        }
7:      if (bid == 1)
        {
8:          if (tid_in_blk < nBlockNum)
            {
9:              while (in[tid_in_blk] != N)
                {//Do nothing here}
            }

10:         __syncthreads();

11:         if (tid_in_blk < nBlockNum)
            {
12:             out[tid_in_blk] != N;
            }
        }
13:     if (tid_in_blk == 0)
        {
14:         while (out[bid] != N)
            {//Do nothing here}
        }
15:     __syncthreads();
    }
```

4.2 State Generation

Another point which was touched upon in Section 2 is that, for reachability analysis and other interesting verification problems, an *a priori* graph search is much easier and less useful than generating new states in the graph *on-the-fly*. Ideally, entire graphs should be propagated from an initial state and a list of rules for generating new states.

The SPIN model checker creates this list of rules at compile time when pre-processing a Promela file as input. Specifically, SPIN's *pan.m* and *pan.r* files describe the rules for forward and reverse state transitions, respectively. As we are currently aiming to check safety properties only via BFS, we only concern ourselves with the *pan.m* file. This file is essentially comprised of a giant switch statement, detailing how each process type can change local and global data. In general, the size of this switch is bounded by the number of unique process types, **not** the total number of processes. As an example, with the *Dining Philosophers* problem, 5 philosophers and 15 philosophers will have the same *pan.m* file. This scaling behavior is great for the GPU, since it means these descriptions can be stored in the smaller, faster memory types for even quite large problems. Unfortunately, the fact that transition rules are listed in a switch makes this file unusable on the GPU.

While the GPU is a massively parallel computing platform, one of the drawbacks is how it handles branching behavior. CUDA, and programming for the GPGPU in general, is strongest when the same instruction, or sequence of instructions, is applied many times in parallel to different data. This is often referred to as a SIMD (Single Instruction, Multiple Data) or SIMT (Single Program, Multiple Thread) approach to parallelism. While the CUDA compiler will not reject an *if-else* or *switch* statement, there is a somewhat harsh performance penalty to be paid for including branching. Normally, if your code featured a branch where 50% of cases took the *if* path and 50% the *else* path, this would total to 100% execution time, assuming each path is the same time complexity. With CUDA, even if logically the same 50-50 split should occur, it would have each thread execute *both* paths and only keep the data from the logically correct executions. This means the same 50-50 split would take not 100% but *200%* execution time. While this may not be terrible in some cases (since a 2x penalty compared to a sequential CPU program is not terrible if you are using 200+ threads to speed up the computation), the problem really lies in what is known as *branch divergence*. Since most instructions in CUDA are performed by warps of threads, usually 32, branching imposes two problems. First, you cannot explicitly designate smaller numbers of threads to take each path in a branch by breaking it up into multiple instructions to be computed by half-warps or smaller; a warp is the smallest unit you can work with. Second, if you are scheduling multiple tasks for different warps and one has to handle branching, this can greatly impact scheduling. If all other warps are supposed to synchronize with the branching warp, they now all suffer the performance penalty of that branching warp. And while the 200% penalty may not be much, many of the switch statements in the *pan.m* files have >20 different cases- more than a **2000%** performance penalty!

Since the scaling behavior of *pan.m* is appealing, but the penalty of using it as-is would be exceedingly harsh, we required a GPU-friendly translation. Our goal became to change the different instructions described in the switch into a single, common, instruction per program, merely with different values. The result was a small template resembling the *pan.m* file, but without branching. This template also had great influence on our state format; much like SPIN,

we use global state vectors containing both global variables and process-local state information. Each of our global state vectors are encoded as single 64-bit integers and masks are selected based upon a combination of the bits describing locally relevant global variables and the current local state of the process. This is not to say that we use branching; technically speaking, all of the available bit masks are always applied, but those which are not enabled *simply compute to 0*. This is demonstrated in Algorithm 3. The comments show an if-else branching structure, but both branches are added to the state *s*. The irrelevant path adds 0, leaving no influence on the newly generated state. On top of eliminating branching from SPIN's switch statement, we use the same method to remove branching from nondeterminism. Lines 10-13 of the algorithm show an example of how nondeterministic choice is covered by our system. Currently these functions are produced manually, but since this is normally a pre-processing step in SPIN and other model checking programs, not part of the actual execution, and our encoding contains no more information than the original *pan.m* file, this is not an enormous issue. We do, however, have plans to make this encoding automatically parsed from Promela or Promela-like input in the near future.

As a simple example of our global state vectors and transition application, we offer the *Dining Philosophers* problem. The global state vector features one bit for each chopstick followed by two bits to describe each philosopher's local state, for a total of *3N* bits per philosopher. A chopstick is either available (1) or in-use (0), and a philosopher has either no sticks (00), the stick to its left (01), both sticks (11), or the stick to its right (10). Each philosopher picks up the left stick first and does not relinquish it until it has taken the right stick (and presumably eaten); the left stick is also the first stick put down by each philosopher. When trying to generate a new state, a philosopher process will take the values of the stick to its left and right and its own local state to determine enabled transitions. If there is a transition enabled by these bits, the resulting bit-mask will be applied to those sticks and local state, and they will be integrated into the global state vector. The bit-mask does not change the local states of other philosopher processes or sticks that are not locally relevant. If the transition is not enabled (the sticks necessary to make the transition are not available), then the resulting bit-mask is 0 and the state remains the same. Since that state would have already been explored, it does not get further expanded for the next round.

4.3 Hashing Techniques

Once states have been generated, they need to be checked for past visitation and stored if they are new. Since this list needs to be checked and updated by many threads at a time without an expensive locking penalty, with each thread needing to check against the entire list, we required a highly parallel (preferably GPU) data structure. Cuckoo hashing, as described in [1], later refined in [2], and available in the CUDA Parallel Primitives (CUDPP) library, seemed to suit our needs. Cuckoo hashing is a very straightforward technique in which each entry in the table gets a total of H, usually 4, unique hash values from H different hash functions. Each item will go to its first location of the table and evict the current entry, if there is one. The evicted entry uses its value and the index it was evicted from to determine which

Algorithm 3 On-the-fly state generation for GPU

```
     //i = the process id executing
     //d = nondeterministic branch
     //N = number of processes
     //s = current state (64 bits)

     //Extract the program counter for process i
1:   int bshift = i * LOCAL_STATE_NUMBITS;
2:   int pc = (s & (PC_MASK << bshift)) >> bshift;
     //Store local pc increment
3:   int pc_i= (1L << bshift);

     //Extract the local/global variables
4:   j = (s & (J_MASK << bshift));
5:   j = j >> (bshift + OFFSET_J);

6:   pos = (s & (POS_MASK << bshift));
7:   pos = pos >> (bshift + OFFSET_POS);
     .....

     //Execution of Promela code
     //NCS: if
     //:: j = 1; goto wait;
     //fi;
     //GPU code
8:   s+=(pc==NCS) * ((1-j) << (bshift + OFFSET_J));
9:   s+=(pc==NCS) * (2 * pc_i ); //goto wait
     .....
     //Handling if and nondeterminism
     //q3: if
     //:: d_step {k<N && (k==i || pos[k]<j);
                  k = k+1;} goto q3;}
     //:: d_step {pos[j-1]!=i || k==N;
                   j = j+1;} goto wait;
     //fi;
10:  k_inc = (1L << (bshift + OFFSET_K));
11:  j_inc = (1L << (bshift + OFFSET_J));
12:  s+=(pc==Q3)*(d*(k<N)*(k==i || pos_k<j)*k_inc);
13:  s+=(pc==Q3)*((1-d)*(s_j!=i || k==N)*(j_inc-2*pc_i));
14:  return s;
```

of its hash functions had been used, and then moves in to the slot corresponding to its next hash value. This eviction chain continues until either every item is properly slotted into the table, or the chain grows too large. When the chain grows excessively large an additional hash function is used for a smaller, secondary table called the stash. In practice, the hash values are tuned so the number of items in the stash are minimized and collisions there are avoided.

The hash and stash constants are combinations of a random number generator and a very large prime divisor. The constants are used alongside the global state vector to produce *H+1* unique hash values (one for the stash).

While [20] abandoned the use of cuckoo hashing in favor of their own hashing scheme, this was largely due to their state representation. Adding items to the cuckoo hash table and checking entries can only be guaranteed to be atomic for 64-bit or smaller values. Since the states used in [20] are larger than 64-bits in size, they did not have these atomic

properties. Our global state vectors are single 64-bit integers, so we can use cuckoo hashing and preserve these atomic guarantees, without the need of a major re-design.

5. RESULTS

All experiments were performed on a 2.93GHz Intel Xeon X5670 processor with 12 logical cores and 30GB shared memory. This system included an EVGA GTX 670 GPU with 2GB global memory, which was used with CUDA version 5.0 for the GPU programs. The SPIN and multicore SPIN experiments used version 6.2.3 of the model checker. Multicore experiments used 11 of the 12 logical cores available, with one core performing the remaining system tasks.

All GPU experiments used 2 blocks of 128 threads each, $H = 4$ for the hash table, and eviction chains of 30. These values were determined to be optimal in an independent series of experiments (not included in the paper). The exception to this is the number of threads, which is non-optimal for small problems but optimal for larger problems. We chose to keep this value constant to maintain the integrity of the experiments, despite the overhead disadvantage it introduces on smaller state spaces. Two of the three Promela files used came from the BEEM database, with the Promela code for Dining Philosophers provided directly by the SPIN creator Gerard Holzmann to match the specifics of our variant of the classical problem.

For all experiments, SPIN and multicore SPIN state-space optimizations are disabled. This is not to give us an unfair advantage, but rather to ensure that the total number of visited states is the same for all tools. As such, the states-per-second metric can be properly judged. The exception to this is the final three graphs, in which SPIN has its partial-order reduction algorithm enabled (for both SPIN platforms). This actually serves to demonstrate the need to disable optimizations for these tests, as both versions of SPIN have lower states/second rates for the Peterson suite with the state space reduced, but have a significant speed-up in exploration (∼1.27x).

The Anderson problem is a queue-lock mutual-exclusion algorithm found in the BEEM database, and we experimented with the Anderson 2, 3, and 4 models from said database. Anderson 2 and Anderson 3 each have three processes, but differ in how they enter the critical section. The former follows the simpler pattern of the even-numbered Anderson models, resulting in 1461 unique states, while the latter uses the more complex odd-numbered Anderson instruction, producing 52.5 million unique states. Anderson 4 is another even-patterned model, with four processes and a total of 29,300 unique states.

Figure 2 shows the results for the Anderson algorithm. As in all of these graphs, the y-axis is the number of unique states visited per second and the x-axis are the various models for the algorithm. Traditional SPIN results are in red, multicore SPIN results are in blue, and GPU-based results are in green. There are two features of this graph that stand out: the apparent lack of results for the GPU version on Anderson 2 and for multicore SPIN on Anderson 3, and the very large green bar for Anderson 3. For Anderson 2, the GPU version produces just over 1,300 unique states per second,

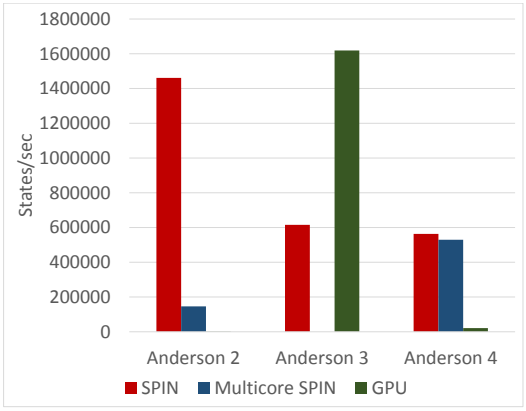

Figure 2: Anderson Algorithm Results.

compared to the massive 1.46 million unique states per second produced by SPIN. This is highly misleading, however, since Anderson 2 only has 1,461 unique states. Both our GPU system and multicore SPIN appear to perform poorly in comparison due to their respective overheads. The GPU overhead is much larger, as we kept the number of threads constant for all experiments, and we therefore incurred a significant performance penalty on smaller models.

The results for Anderson 3 are much more interesting. The GPU-based result is over 1.6 million states per second, whereas the multicore version of SPIN did not complete the search. Unlike traditional SPIN, which expands its hash table dynamically, when the hash table of multicore SPIN becomes full, the search terminates. Since the search failed to complete, its Anderson 3 data was not included. The results for the Anderson 4 model are a less pronounced version of those seen with Anderson 2, as the larger model has less overhead penalty, but it still does not perfrom well enough to trump traditional SPIN.

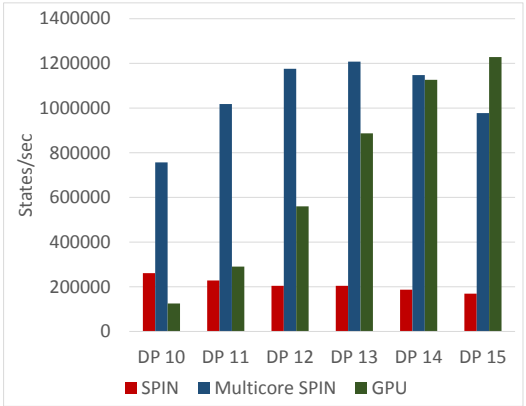

Figure 3: Dining Philosophers Algorithm Results.

The Dining Philosophers problem is a well-known mutex algorithm where N philosophers have access to N chopsticks, one to the left and the other to the right, and each philosopher needs both to eat with (access the critical section). In our specific version, the philosopher always picks up the left stick, followed by the right stick, and then places them back down in the same order. If a philosopher picks up the left stick, he cannot place it down until he has acquired the

right stick. Figure 3 shows the results for the problem for 10-15 philosopher processes. Dining Philosophers 15, which produces just over 9.3 million states, shows the GPU version coming out on top with performance decreasing until it reaches its minimum at 10 philosophers. This trend is the opposite of SPIN but for a similar reason. Just as the GPU experiences overhead regardless of the number of states visited, causing large penalties on models with fewer states, SPIN suffers overhead on larger models, since it needs to dynamically resize its hash table. As such, the GPU version is able to achieve 7.26x the number of states per second of SPIN at 15 philosophers, but only 0.48x for 10. Multicore SPIN has its peak performance at 13 processes, but maintains relatively high states-per-second rates for the other tests. At 14 processes, the performance of multicore SPIN and the GPU-based version is almost equal, with a slight edge going to the former.

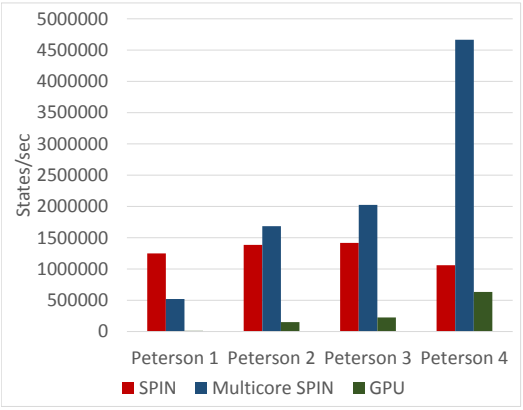

Figure 4: Peterson Algorithm Results.

Figure 4 shows the results for the Peterson problem. This is a mutual-exclusion protocol for N processes, and can be found in the BEEM model-checking database. All four variants produce a relatively small number of states, peaking at 1.1 million for Peterson 4, with the fewest at 12,498 for Peterson 1. What is noteworthy about this set of experiments is that, once again, as the number of states climb, so does the performance of both our GPU-based system and multicore SPIN.

The logical question then is why did we stop at Peterson 4, instead of continuing to Peterson 5 and beyond until the overhead penalty vanished? The answer is that the number of states produced by Peterson 5 in some preliminary tests exceeded 1 billion. This is much larger than the number of states we can support on a hash table in 2GB of global GPU memory. That may quickly change, however, as the memory footprint of GPUs continues to frequently double in size. Based on this preliminary result, and the results from the other tests showing our top performance when states are at or above roughly 10 million in number, we are confident that we will come out on top for Peterson 5 when enough GPU memory becomes available.

The final three graphs, Figures 5-7, have SPIN's partial-order reduction (p.o.r.) algorithm enabled for the traditional SPIN and Multicore platforms. This algorithm is on by default for both platforms and is disabled with the flag

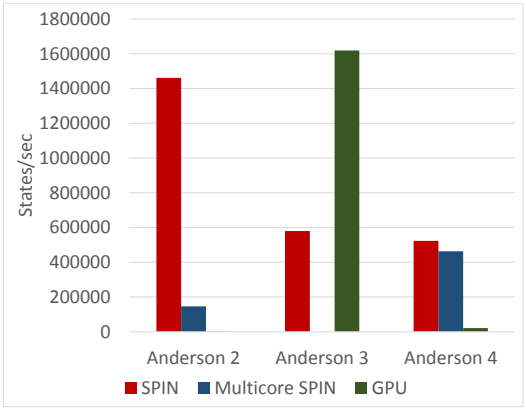

Figure 5: Anderson Results with Partial-Order Reduction for SPIN/Multicore.

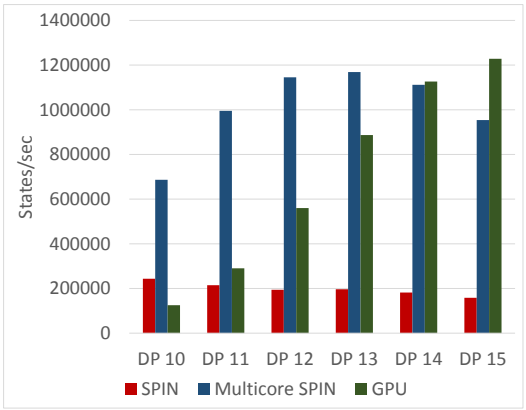

Figure 6: Dining Philosophers Results with Partial-Order Reduction for SPIN/Multicore.

-DNOREDUCE. In our testing suite, the only problem that saw a reduction in the number of states with this flag removed was Peterson. The other two saw a slight performance penalty for running the algorithm without a state-space reduction, thereby still exploring the same number of states. Looking at Figure 7, it would seem that p.o.r. negatively impacted performance. This is not true, however. Although the number of unique states visited per second decreased, the total number of states, and therefore the execution time, decreased dramatically as well.

This is the one graph in the paper that is slightly misleading, since our system does not yet have this feature and therefore explores the entire search space. The reduced state space is still a subset of the original, and the state representation is still the same, but many superfluous states are skipped in order to achieve massive speed-up in state exploration.

We chose to include these experiments because we wanted to show the power of p.o.r. as a technique to achieve speed-up; the Peterson 4 and 3 models both take over 1.27x as long to run with it disabled, and there are other problems where this can be even more significant. The multicore version of SPIN sees similar gains on problems where p.o.r. has an impact on the state space. The performance penalty when it does not work is relatively minor, and is well below the overhead for

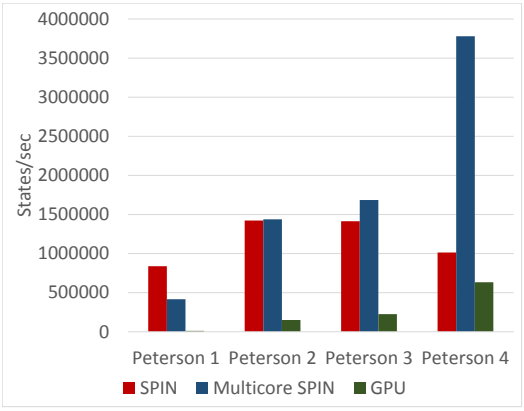

Figure 7: Peterson Results with Partial-Order Reduction for SPIN/Multicore.

running on the GPU. We plan on integrating p.o.r. with our GPU-based system to gain the available benefits.

We did not directly compare the performance of our system to GPUexplore [20]. While it was our intention to include this tool in our experiments, after running models with GPUexplore and speaking with one of the authors, we realized that any conclusions we would draw would not be apt. The reasons for this are two-fold: first, their system also requires a great deal of modeling by hand at this time, not unlike our own; second, the different state representation ends up having a significant impact on the number of states visited. The LTSs they use do not support variables, which impacts automatic model conversion, but also may alter the number of states visited. Upon running one of the provided models to completion, and running the equivalent model with SPIN, GPUexplore had not only not outperformed SPIN on the model, but produced an order of magnitude fewer unique states than the mainstay model checker. While this does not necessarily mean the results were not equivalent, it does mean that the states-per-second metric we have been measuring would have been difficult to interpret. If GPUexplore were to constantly produce less states, the metric would be unfairly biased in favor of our system. Instead, we chose not to include GPUexplore in our performance evaluation.

6. CONCLUSIONS

We set out to establish a GPGPU-parallel version of the 2012 multicore SPIN model checker [11]. While there still may be room for improvement, our initial results are quite strong. Although we incur significant overhead on smaller-sized problems due to the fact that we are using the 256 GPU threads for our experiments, we excel on problems with state spaces between approximately 10 million and 100 million states. The lower bound is due to overhead (below 10 million states, the overhead from maintaining 256 threads dominates), whereas the upper bound is due to the size of the GPU memory we had available. Since GPU memory sizes continue to double, this upper bound is ever-expanding.

Our best result was on the Dining Philosophers problem with 15 processes, where we achieved 7.26x speed-up when compared to traditional SPIN. We also managed to out-pace multicore SPIN. One may argue that this result is not very impressive considering the 256 threads used on the GPU versus the 11 logical cores used by multicore SPIN. Live, GPU-based state-space exploration, however, is a non-trivial problem, as evidenced by the considerations we needed to make for control-flow branching, memory footprint, and data transfer. See also [20], where 512-thread GPUexplore achieved results comparable to 10-core LTSmin.

Future work will focus on continuing to improve our performance via further consideration of GPU memory types and coalesced access patterns, as well as new features such as dynamic memory allocation in the new version of CUDA. Dynamic memory allocation will figure prominently in the new hashing scheme we plan to develop, as we will use this facility to ensure high utilization of the hash table.

Future work will also involve the incorporation of many features found in SPIN, including optimizations such as partial-order reduction. We are also working on generating models directly from Promela or a similar Promela-like input format. Only with this feature could we truly be considered to have the GPU-version of the Simple Promela INterpreter, but we believe we are well on our way to this goal.

7. ACKNOWLEDGMENT

We would like to thank Gerard Holzmann for passing on to us implementation details of the SPIN model checker, supplying us with an optimal Promela encoding of the Dining Philosopher problem with which we could test our implementation, and for providing general direction as we worked on implementing his algorithm on the GPGPU architecture.

8. REFERENCES

[1] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the GPU. In *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pages 154:1–154:9, New York, NY, USA, 2009. ACM.

[2] D. A. F. Alcantara. 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.

[3] J. Barnat, P. Bauch, L. Brim, and M. Češka. Designing fast LTL model checking algorithms for many-core GPUs. *Journal of Parallel and Distributed Computing*, 72(9):1083–1097, 2012. Accelerators for High-Performance Computing.

[4] J. Barnat, L. Brim, and P. Ročkai. 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*, volume 4595 of *Lecture Notes in Computer Science*, pages 187–203. Springer Berlin Heidelberg, 2007.

[5] J. Barnat, L. Brim, and P. Ročkai. DiVinE multi-core - a parallel LTL model-checker. In S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, editors, *Proc. of ATVA 2008: the 6th International Symposium on Automated Technology for Verification and Analysis, Seoul, Korea, October 20-23*, volume 5311 of *Lecture*

Notes in Computer Science, pages 234–239. Springer Berlin Heidelberg, 2008.

[6] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL model-checking in SPIN. In M. Dwyer, editor, *Proc. of SPIN 2001: the 8th International SPIN Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 200–216. Springer Berlin Heidelberg, 2001.

[7] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, New York, NY, USA, 2011. ACM.

[8] Y. Deng, B. D. Wang, and S. Mu. Taming irregular EDA applications on GPUs. In *Proceedings of the ICCAD '09: the International Conference on Computer-Aided Design*, ICCAD '09, pages 539–546, New York, NY, USA, 2009. ACM.

[9] S. Edelkamp and D. Sulewski. Efficient explicit-state model checking on general purpose graphics processors. In J. Pol and M. Weber, editors, *Proc. of SPIN'10: the 17th International SPIN Conference on Model Checking Software*, volume 6349 of *Lecture Notes in Computer Science*, pages 106–123. Springer Berlin Heidelberg, 2010.

[10] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In S. Aluru, M. Parashar, R. Badrinath, and V. Prasanna, editors, *Proc. of HiPC'07: the 14th international conference on High performance computing*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer Berlin Heidelberg, 2007.

[11] G. Holzmann. Parallelizing the SPIN model checker. In A. Donaldson and D. Parker, editors, *Proc. of SPIN 2012: the 19th International Workshop on SPIN Model Checking Software*, volume 7385 of *Lecture Notes in Computer Science*, pages 155–171. Springer Berlin Heidelberg, 2012.

[12] G. Holzmann and D. Bošnački. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, Oct 2007.

[13] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proc. of PPoPP '11: the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[14] L. Luo, M. Wong, and W. Hwu. An effective GPU implementation of breadth-first search. In *Proc. of DAC '10: the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[15] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proc. of SPAA '02: the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[16] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, may 2006.

[17] C.-H. Shann, T.-L. Huang, and C. Chen. A practical nonblocking queue algorithm using compare-and-swap. In *Proc. of ICPADS '00: the 7th International Conference on Parallel and Distributed Systems*, pages 470–475, 2000.

[18] K. Verstoep, H. Bal, J. Barnat, and L. Brim. Efficient large-scale model checking. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[19] V. Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *Proc. of CVPRW '08: the IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8, June 2008.

[20] A. Wijs and D. Bošnački. GPUexplore: Many-core on-the-fly state space exploration using GPUs. In E. Ábráham and K. Havelund, editors, *Proc. of TACAS 2014: the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 233–247. Springer Berlin Heidelberg, 2014.

[21] S. Xiao and W. chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Proc. of IPDPS 2010: the IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, April 2010.