Design of a Scalable Hash Table on a GPU

Thomas Neele
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
t.s.neele@student.utwente.nl

ABSTRACT

We investigate the scalability of an existing lockless hash table when it is implemented on a GPU. This lockless hash table is an essential part of a model checker, a tool that can be used to prove the correctness of software that performs critical tasks. We show that our implementation of the hash table on a GPU is scalable and that a GPU can lookup twice as much per second as a CPU when running the same OpenCL implementation. We also propose a design for integrating our implementation with a model checker. We argue that this design will lead to better performance of the model checker.

Keywords

GPU, OpenCL, hash table, model checking, scalability

1. INTRODUCTION

The research field of formal methods not only focuses on mathematical proofs, but also on automated model checking. Static *model checkers*, like LTSmin [8] and DiVinE [3], check algorithms for possible deadlocks and other unwanted states that are defined by safety properties. To find these problems, a model checker explores the graph of all states the program can reach. A state is represented by a large vector and for each state in this graph, the safety properties are evaluated. Performing the state space exploration requires many computation steps and since the introduction of multi-core processors, large parts of the model checkers are optimized for multi-core systems.

To ensure that every state in the graph is visited exactly once, the exploration algorithm needs to store information about the visited states. The database that is used to store all those states is often implemented as a hash table. Because it will be accessed every time a new state is visited, the hash table is a critical part of a model checker and has a significant influence on its performance. To increase performance on multi-core systems, Laarman et al. [7] proposed a *lockless hash table*, which is designed with the specific requirements of model checking in mind. The hash table only supports the *find_or_put* operation, which finds a state in the database or adds it when it is not found. Because the implementation is not based on locks, it is scalable: even when many threads are storing items in the hash table, the performance of each individual thread does not decrease significantly. The hash table was later improved to a *tree database* [9]. The tree database combines the performance of a hash table with reduced memory usage.

The performance of a model checker not only depends on its implementation, but also on the hardware it is executed on. Traditionally, all general computational tasks are performed by a CPU. While *graphical processing units* (GPUs) are aimed at rendering graphical scenes, in recent years they are also being employed for more general tasks. NVIDIA's CUDA, released in 2006, and Khronos WG's OpenCL, released in 2008, provide an easy way to program GPUs. Since GPUs provide huge parallelism and memory bandwidth, they can offer a speedup of multiple magnitudes. Model checkers can benefit from the performance of GPUs, as shown by Barnat et al. [2].

We combine the scalability of Laarman's hash table with the massive parallelism of GPUs. The hash table already proved to be scalable up to 48 threads on an x86 CPU [11]; we evaluate this scalability further by implementing the hash table in OpenCL and running it on a GPU. We will show how scalable this GPU implementation is and whether the GPU can provide better performance when storing states in the hash table.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
20th *Twente Student Conference on IT* January 24th, 2014, Enschede, The Netherlands.
Copyright 2014, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

2. RESEARCH QUESTIONS

Our main research question is: "How scalable is a lockless hash table on a GPU?". In order to optimize the scalability, we have identified the following subgoals.

- How should the lockless hash table be stored in GPU memory?
- How can the workload of storing state vectors efficiently be divided between GPU threads?

We also investigate whether the performance of our implementation is better on a CPU or on a GPU. This is an important aspect when considering the application of GPUs in model checking.

3. BACKGROUND

OpenCL.

OpenCL (open computing language) is a standard developed by Khronos Working Group [6]. It aims to provide an easy way to develop parallel applications for many different devices. The code that runs on the parallel device, called a kernel, is programmed in OpenCL C, a variant of C99. Kernels and devices can be managed by the host,

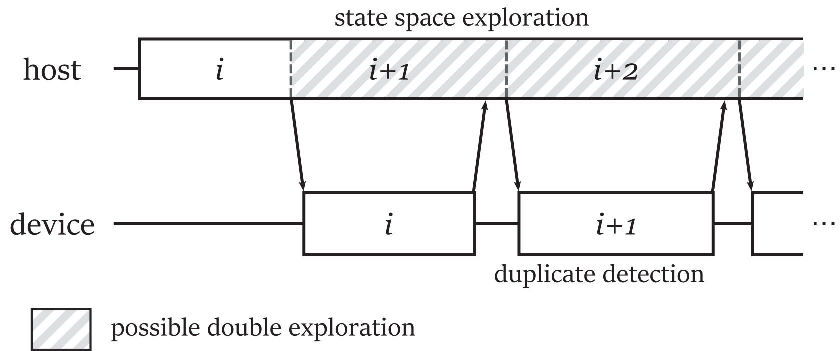

Figure 1. The host (CPU) offloads the duplicate detection to the OpenCL device (GPU). An empty horizontal timeline indicates that the device is idle.

usually a CPU, via a C or C++ API. Kernels can be executed on one or more work groups, which consist of one or more work items. The amount of work items and work groups can be set when calling the kernel. GPUs have a *single-program-multiple-data* (SPMD) architecture: all work groups execute the same code. Additionally, each work group has a *single-instruction-multiple-data* (SIMD) architecture: all work items within a work group always execute the same code path. The performance of a work group is determined by the performance of the work item that handles the worst case.

The OpenCL standard defines different address spaces that have their own characteristics. The global memory is shared across all work items, but on most devices it is relatively slow. All work items within one work group share local memory. Finally, each work item has its own memory space called private memory. When designing and implementing an algorithm in OpenCL, it is important to consider which address space will provide the best performance for a specific application.

CUDA is a similar technology developed by NVIDIA. It supports some advanced features like dynamic memory allocation. However, programs written in CUDA can only run on NVIDIA GPUs.

Lockless hash table.

The lockless hash table proposed by Laarman et al. [7] is optimized for multi-threaded use by using *compare-and-swap* operations instead of locks. Another optimization is linear probing: buckets of the hash table are aligned with cache lines of the processor. Making optimal use of the cache reduces cache synchronization between processors, thus increasing the performance.

The tree database [9] uses the same underlying hash table, but stores states in a memory efficient way. The compression technique is based on the observation that successive program states often differ for only a small part. Each state vector is saved as a balanced binary tree; subtrees that are already entered in the hash table will not take any extra space. The scalability of the uncompressed hash table and the tree database proved to be similar. Because of time constraints, our research is based on the lockless hash table.

4. RELATED WORK

Barnat et al. already showed that GPUs can provide a significant speedup when used for static model checking [2]. For the model checker DiVinE, they implemented a successor finding algorithm and property checking algorithm in CUDA. The implementation of the state database was done on the CPU, however. The algorithm was similar to the lockless hash table of Laarman et al. These integrated algorithms proved to be scalable on a variable number of CPUs.

GPUs also proved to be beneficial to the performance of probabilistic model checkers. Bošnački et al. extended the model checker PRISM with a CUDA implementation that calculates the transition probabilities [4]. Their implementation provides a speedup of 15 to 18 times when compared to a sequential implementation. The scalability of their algorithms on GPUs was not investigated.

Many kinds of data structures can benefit from the parallelism of GPUs, as proved by Misra et al. [10]. They provide an extensive analysis of the performance of their data structures. However, a scalability analysis was not performed.

An alternative to the tree database is a *reconstruction tree* [5]. In contrast to a tree database, a reconstruction tree can contain state vectors of variable width and also supports the deletion of state vectors. The *delete* operation enables additional optimizations to graph exploration algorithm. However, a reconstruction tree proved to be less scalable than a tree database or lockless hash table.

The hash table proposed by Alcantara et al. [1] also provides access times of $O(1)$. Like the lockless hash table and the tree database, it does not support vectors of variable width. The implementation of this hash table in CUDA proved to perform well. Experiments with regard to the scalability were not performed.

5. DESIGN

Memory.

The hash table can either be stored in global memory or in local memory. Storing the hash table in local memory affects the distribution of work load across work groups. Each work group stores its own part of the table, vectors destined for a certain part of the table have to be stored by the associated work group. This may lead to an uneven distribution of the work load. An uneven work load may harm efficiency, because other threads will wait for the worst case to finish, due to the SPMD architecture of a GPU. Besides that, local memory is also too small to store a large hash table, which would lead to local memory 'spilling over' into the global memory. To avoid these issues, we placed the hash table in global memory. Global memory is slower than local memory, but it can be accessed by all work items.

Work division.

We propose the following design for integrating the hash table with a model checker. To achieve good performance, an efficient division of the work between the CPU and the GPU is needed. In our design, the graph exploration algorithm runs on the CPU; performing duplicate detection is done on the GPU (figure 1). The CPU should run a *breadth first search* (BFS) algorithm to explore the state space. Whenever the CPU is done exploring a certain amount of states, a round of duplicate detection is needed. This work will be sent of to the GPU, by copying the array of state vectors to the GPU global memory. While the GPU is working on the duplicate detection, the CPU can continue to run the BFS exploration. At that time, the BFS exploration is not aware of any information about duplicate states (indicated in figure 1 by shading). Therefore, several states may be explored multiple times. We believe that this double exploration is a benign data race, because it is better to perform an exploration with outdated information than to not do any exploration at all. When the GPU is done with its duplicate detection, it will update the BFS explorer with the new information. This cycle will continue until the state space is fully explored.

In our case, duplicate detection on the GPU is implemented as a hash table. When the GPU receives an array of state vectors, it will compute the hashes of these vectors and store each hash in the database. Because this leads to unpredictable access of the memory, the memory locality is not optimal. Sorting the array of hashes and then storing them increases the memory locality, which may also improve performance. The increase of performance depends on the amount of cache a GPU has and on the amount of overhead that is caused by sorting the array.

6. IMPLEMENTATION

Our implementation is largely based on the implementation of the hash table that is part of LTSmin. The specification of OpenCL C has some limitations when compared to ANSI C, but there were no significant changes needed to the *find_or_put* algorithm. However, the OpenCL C standard does not define atomic operations on data types of 16 bits. Therefore, we have written our own implementation that depends on the standard atomic operations.

```
Data: thread_id, num_threads, database D
Input: State[ ]
1 num_vectors ← length(State) / num_threads;
2 offset ← num_vectors * thread_id;
3 forall the i: offset ≤ i < offset + num_vectors do
4 |   find_or_put(D, State[i]);
```

Algorithm 1: State lookup kernel

The kernel that executes the *find_or_put* algorithm in parallel is outlined in algorithm 1. The only argument of the kernel is an unsorted array of state vectors that are generated by the host. These vectors are equally distributed among the work items that execute the kernel. Each work item computes the offset based on its *thread id*. The work items will then start storing the elements in the database in a sequential way. We have not implemented the proposed integration with a model checker or an algorithm that sorts the hash values of the state vectors.

7. RESULTS

To evaluate the scalability of our implementation, we have created a benchmarking program. This program stores

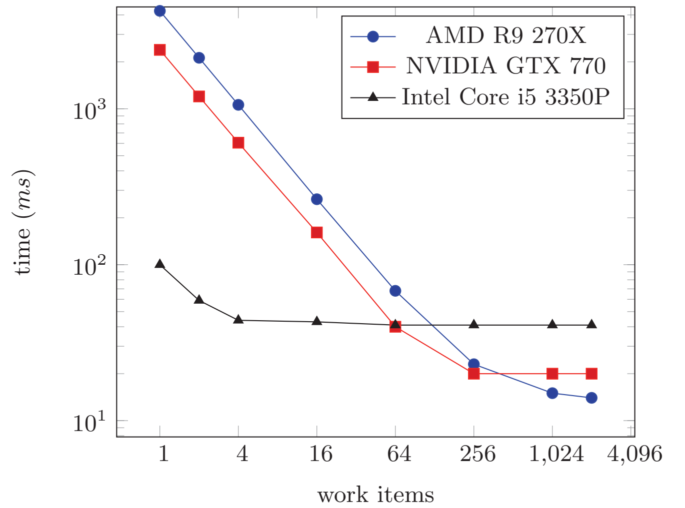

Figure 2. Runtime of executing *find_or_put* for 1M vectors

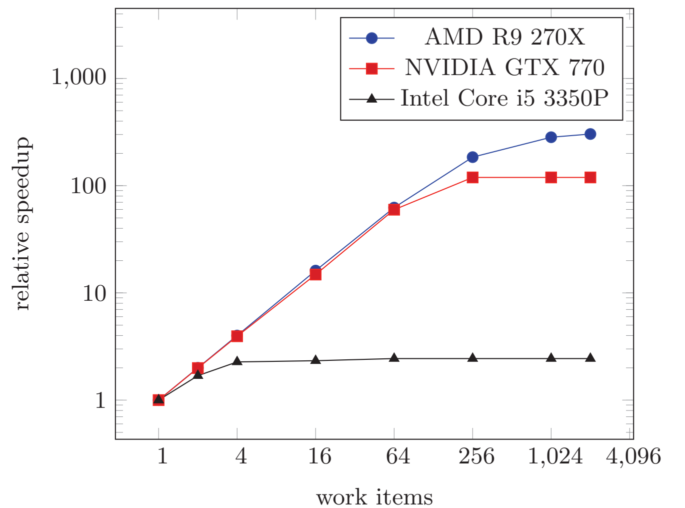

Figure 3. Relative speedup when compared to the runtime of one work item

1 million random 32-bit vectors in the database. The database has a size of 2^{25} elements and each element takes 6 bytes, resulting in a memory usage of about 192 MB. The vectors are generated by the CPU and then transferred to the GPU. The GPU threads then store these vectors in the database. We measured the time it takes for the GPU to complete the storage operations for different amounts of work items. The runtime was measured by using the profiling functions in the OpenCL API; the benchmarks were executed once. Because each work group consists of one work item, the amount of work groups is equal to the amount of work items.

To execute the benchmark, we used an AMD Radeon R9 270X and an NVIDIA GTX 770. These GPUs have 1280 and 1536 shader processors respectively. The AMD GPU was clocked at 1000 MHz and its memory was clocked at 5.6 GHz. The NVIDIA GPU was clocked at 1046 MHz and its memory was clocked at 7 GHz. In order to compare these GPUs to a CPU, we also ran the OpenCL kernel on an Intel Core i5 3350P quad-core CPU clocked at 3.1 GHz; the main memory was clocked at 1333 MHz.

8. ANALYSIS

Figure 2 shows the time it takes to store 1 million random state vectors into the hash table. The runtime is measured in milliseconds, for different amounts of work items. Figure 3 shows the same results relative to the speed of one work item.

These results show that our implementation is scalable up to a certain amount of work items. When running the benchmark on the AMD GPU, 1024 work items still provide a speedup relative to 256 work items. The NVIDIA GPU has reached its maximum performance at 256 work items. A possible explanation is that the NVIDIA GPU is fully utilized when a kernel is assigned 256 work items. A single work item of the NVIDIA GPU is faster than one of the AMD GPU (figure 2). When run on 1024 work items, the AMD GPU is faster by 33%.

While a single core of the CPU is significantly faster, the CPU does not scale beyond 4 work items. By then, all four cores are fully loaded. Both GPUs perform better when the kernel is executed on a high number of work items.

9. CONCLUSION

The results of our benchmarks show that the hash table designed by Laarman [7] can certainly benefit from the massive parallelism a GPU has to offer. Even when run on 256 work items, the relative increase in performance is 72% of the theoretical maximum (a relative speedup of 256) for AMD and 47% for NVIDIA. When the possible optimization of sorting the array of hashes is implemented, this may lead to even better use of caches in the GPU, thus increasing scalability.

Our benchmarks also show that GPUs perform better at looking up vectors than a CPU. Therefore, model checkers can benefit from this speedup offered by GPUs. We proposed a design for integrating our OpenCL implementation of the lockless hash table with a model checker. By offloading the duplicate detection to the GPU and exploiting benign data races, the CPU only has to run a graph exploration algorithm. We believe this will improve the overall performance of model checkers.

While these results are promising, the amount of memory on a GPU is severely limited. Even the most expensive GPUs have a memory size of only 12 GB, whereas a high end server may have more than 100 GB of main memory. The potential of a model checker largely depends on the amount of memory it can allocate. When more memory is available, the model checker can explore larger state graphs and check larger models.

10. FUTURE WORK

Further research into the practical applications of our implementation is needed. By implementing our design from section 5, the hash table can be integrated with a model checker. Then, it is possible to run real-world benchmarks against the hash table. An optimization that may improve the scalability of the hash table on a GPU is the sorting of hashes before inserting them in the database, as discussed in section 5.

Future work also needs to focus on the memory limitations of GPUs. A tree database [9] is much more efficient with memory, while it still provides similar scalability as a lockless hash table. Another possible solution is the application of a multi-GPU system. While this requires the hash table to be split across the devices, the total amount of memory bandwidth is increased.

11. 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. *ACM Transactions on Graphics*, 28(5):154:1–154:9, 2009. doi: 10.1145/1618452.1618500.

[2] 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, Sept. 2012. doi: 10.1016/j.jpdc.2011.10.015.

[3] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. DiVinE 3.0 – An explicit-state model checker for multithreaded C & C++ programs. In *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.

[4] D. Bošnački, S. Edelkamp, D. Sulewski, and A. Wijs. Parallel probabilistic model checking on general purpose graphics processors. *International Journal on Software Tools for Technology Transfer*, 13(1):21–35, Oct. 2010. doi: 10.1007/s10009-010-0176-4.

[5] S. Evangelista, L. Kristensen, and L. Petrucci. Multi-threaded explicit state space exploration with state reconstruction. In *Automated Technology for Verification and Analysis*, volume 8172 of *Lecture Notes in Computer Science*, pages 208–223. Springer International Publishing, 2013. doi: 10.1007/978-3-319-02444-8_16.

[6] Khronos OpenCL Working Group. OpenCL specification, 2013. URL `http://www.khronos.org/opencl/`.

[7] A. W. Laarman, J. C. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Switzerland*, pages 247–256. IEEE Computer Society, October 2010.

[8] A. W. Laarman, J. C. van de Pol, and M. Weber. Multi-core LTSmin: Marrying modularity and scalability. In *Proceedings of the Third International Symposium on NASA Formal Methods, NFM 2011, Pasadena, CA, USA*, volume 6617 of *Lecture Notes in Computer Science*, pages 506–511. Springer Verlag, July 2011. doi: 10.1007/978-3-642-20398-5_40.

[9] A. W. Laarman, J. C. van de Pol, and M. Weber. Parallel recursive state compression for free. In *Proceedings of the 18th International SPIN Workshop, SPIN 2011, Snow Bird, Utah*, volume 6823 of *Lecture Notes in Computer Science*, pages 38–56. Springer Verlag, July 2011.

[10] P. Misra and M. Chaudhuri. Performance Evaluation of Concurrent Lock-free Data Structures on GPUs. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 53–60, Singapore, Dec. 2012. IEEE. doi: 10.1109/ICPADS.2012.18.

[11] F. I. van der Berg and A. W. Laarman. SpinS: Extending LTSmin with Promela through SpinJa. In *11th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2012, London, UK*, volume 296 of *Electronic Notes in Theoretical Computer Science*, pages 95–105. Elsevier, September 2012. doi: 10.1016/j.entcs.2013.07.007.