Porting tree-based hash table compression to GPGPU model checking

master's thesis

by Danny Bergsma

University of Twente, Enschede, The Netherlands
Electrical Engineering, Mathematics & Computer Science (EEMCS)
Formal Methods & Tools (FMT)

Enschede, 11 December 2019

Table of Contents

Voorwoord (preface)

Het schrijven van deze scriptie heeft mij veel dingen geleerd: Hiervoor had ik nooit serieus met C gewerkt. Ook CUDA en GPGPU programmeren in het algemeen waren nieuw voor mij. Het boek 'Professional CUDA C Programming' van John Cheng, Max Grossman en Ty McKercher leerde je, net als bij de propedeusevakken 'Programmeren 1' en 'Programmeren 2', vooral de ideeën en concepten van GPGPU programmeren en "toevallig" ook nog CUDA C (analoog aan object-georiënteerd programmeren en Java bij de genoemde vakken). Gaandeweg kreeg ik steeds beter de essentie van GPGPU programmeren door.
Het project was groter dan welk project ik ook eerder gedaan had. Het managen van alle informatie is dan lastig, zeker als je het grotendeels alleen moet doen; je wilt zoveel vertellen en bent bang iets te vergeten. En je lezer zit niet zo in de materie als jij…
Op het laatst bleek dat er serieuze tekortkomingen zaten in de code waarop ik had voortgeborduurd. Daardoor waren de resultaten van de experimenten niet meer valide. Het verhelpen van de tekortkomingen en vooral het herhalen van al die experimenten vergden veel doorzettingsvermogen en tijd.
Maar nu ligt hij er, de scriptie. Af en wel, voor zover dat kan. Maar voor nu af genoeg.

Met deze scriptie komt er ook een einde aan mijn masterstudie en mijn leven aan de Universiteit van Twente. Het heeft een tijd geduurd, met uitstapjes naar Groningen en Utrecht. Maar elke keer kwam ik terug naar Enschede. Niet zozeer voor de stad, maar meer voor de sfeer van de universiteit en de opleiding. Beide relatief klein, wat betekent dat je meer je best moet doen om aandacht te trekken van (toekomstige) studenten. Ook nieuw, en niet zo suf als sommige andere universiteiten. En er lijkt altijd wel ergens een nieuw gebouw te worden neergezet of te worden verbouwd…
Het lukte me nooit om goed te integreren. In het midden van het collegejaar kwam ik vanuit Friesland, na eerst toch maar enkele maanden naar Groningen te zijn gegaan. Mijn propedeuse haalde ik zo snel als mogelijk was, maar daarna begonnen de moeilijkheden. Na twee jaar pakte ik mijn bachelorstudie weer op. Aan het eind daarvan ontmoette ik eindelijk een groepje met wie het wel goed klikte, ook op inhoudelijk gebied; daarvóór trok ik een groepsopdracht meestal maar naar me toe, ook omdat mijn ideeën over gewenst niveau geregeld anders waren.
Na een mislukt avontuur in Utrecht, heb ik me door de master in Enschede heen geworsteld. Inhoudelijk vond ik het wel interessant, zeker de vakken van mijn specialisatie, al had ik soms het gevoel dat het nog wel wat dieper mocht gaan. Het afronden duurde echter lang, heel lang. De motivatie was geregeld weg en ik gedij beter in een meer schoolse omgeving.

Nu begint een nieuwe fase in mijn leven. Het trage afronden stokte niet alleen mijn studie, maar ook andere ontwikkelingen. Deze nieuwe fase valt samen met een nieuw kalenderjaar, 2020. Ook als maar een gedeelte van de verwachtingen en wensen voor 2020 uitkomen, zal dit jaar een keerpunt vormen. Een keerpunt naar meer verbondenheid, ontplooiing en ervaring.

Acknowledgements

I would like to thank my graduation committee, and especially Marieke Huisman and Anton Wijs, for their feedback and support.

Abstract

To reduce the costs of faulty software, methods to improve software quality are very popular nowadays. One of these methods is model checking: verifying the functional correctness of the model of a hardware or software system. The model implies a state space, which consists of all possible states of the system and all possible transitions between them.

For complex systems, the number of states can be millions or even more. Consequently, exploring the state space and checking that the system satisfies its specification is computationally very intensive. Multiple model checking algorithms have been adapted to make use of the massive parallelism of GPUs by GPGPU programming, resulting in spectacular performance improvements.

One such implementation is the GPU-based model checker GPUexplore. GPUexplore uses a hash table for the shared store of visited states in the model checking process. Considering the spectacular speedups, this hash table is now the bottleneck, as GPU memory size is relatively limited. Compression of the hash table can be used to reduce space requirements.

But we first tackled two large flaws in GPUexplore's uncompressed hash table: replication of states in the hash table and a hash function that resulted in the inferior distribution of states across the table. We successfully designed and implemented a replication-free hash table with a probabilistic-optimal distribution, whose performance was equal to or even better than the flawed implementation.

Subsequently, we have successfully ported an existing tree-based hash table compression algorithm designed for multi-core CPUs to stand-alone GPGPU hash tables. The recursion in the compression algorithm was the main challenge, as this is implemented differently on GPUs than on CPUs and impacts performance. We made multiple improvements to the original implementation, aimed at reducing recursive overhead. Moreover, we designed and implemented a solution without any recursion at all, consisting of fifteen variants that differ in aspects related to GPU performance.

We used parameterised random data input and state sequences extracted from real-world models, consisting of up to 375M states, for performance analysis: We exhaustively examined the impact of different GPU, input and hash table parameters on performance, both uncompressed and compressed, including the improved versions. We used the results to find optimal settings for each input/program combination, which enabled a fair evaluation of compression overhead and for measuring the performance impact of the improvements.

Ultimately, there was still compression overhead, but very limited, up to a 3.8x slowdown, corresponding to the scattered memory accesses needed for tree-based compression. Considering the spectacular speedups of GPU model checking, the maximum slowdown of 3.8x would not be an issue for integrating compression into GPU model checkers, which would enable checking bigger models and models with data. As integration enables possibilities for reducing recursive overhead and the amount of required memory accesses, compression overhead may then even reduce.

Now, there is only one step left: integrating our compression algorithm into a GPU-based model checker, *e.g.*, into GPUexplore, and examining whether the results of our performance analysis also apply when integrated.

1. Introduction

Writing correct software is not an easy task, especially if concurrency is involved. Faulty software costs billions of dollars a year: costs to remove the bugs, loss of productivity by users of the software, *et cetera*. As software is getting more complex and customers demand higher quality software, methods to improve the reliability of software are very popular nowadays.
One of the approaches is the use of formal methods: by applying mathematical methods, *e.g.*, logic, the correct functioning of software can be proven. Whereas testing can only show the presence of bugs, not their absence (quoting Dijkstra), formal methods can. In our project, we will use one such a method: model checking.

In model checking [1], a model of the system under consideration is given by a formal description (specification) of its (concurrent) behaviour. In addition, properties that the system should satisfy, are formalised. Then the model checker can be used to verify that the model meets its functional requirements (or does not).
The model implies a state space, which consists of all reachable states of the system and all possible transitions between them. It is constructed by starting from the initial state and determining what successor states are reachable by applying one of the possible transitions from this initial state; this process, called reachability, is repeated for those successor states, repeated for the successor states of those successor states, *et cetera*.
As an example, consider a traffic light with three states: 'red', 'yellow' and 'green'. Possible transitions are from 'red' to 'green', from 'green' to 'yellow' and from 'yellow' to 'red'. Several traffic lights can be combined to compose a more complex system. One of the properties of interest in this complex system is the absence of an overall system state where two (or more) traffic lights are in their 'green' state. Model checking can be used to prove that such a state is not reachable from the initial state.
For complex systems, the state space may consist of millions or even more states. Consequently, constructing this state space and verifying the validity of required system properties is computationally very demanding and also very memory-intensive. As performance improvements for Central Processing Units (CPUs) to execute sequential code have stalled [2], model checkers have been developed that make efficient use of the multiple cores of modern CPUs by parallelising core model checking algorithms [3,4].
Recently, Wijs *et al.* further parallelised model checkers by adopting the massive parallelism of Graphics Processing Units (GPUs) [5,6]. Originally, GPUs were used for graphics processing, but they can also be used for tasks the CPU was used to execute, called General Purpose GPU (GPGPU) programming [7]. Areas of application, besides model checking, are media processing [8], medical imaging [9] and eye-tracking [10].
In the GPGPU programming model, a massive number of threads, typically thousands or even more, run the same program concurrently, but on different data: the Single Program, Multiple Data (SPMD) execution model. Many (parallel) algorithms have an implementation for GPUs, often resulting in spectacular speedups, even compared to optimised parallel algorithms running on the most sophisticated multi-core CPUs.
Porting algorithms designed for multi-core CPUs to GPUs, however, is not a straightforward task. For example, the explicit memory hierarchy of GPUs is different from CPUs' single-level memory model. To get the most out of GPGPU programming, crucial for computationally intensive tasks as model checking, implementations need to be tailored to the specifics of GPUs. Moreover, the programming models and frameworks are different from those for CPU programming: OpenCL [11] is a cross-platform, cross-vendor GPGPU programming framework; CUDA [12] only works on NVIDIA GPUs.

GPUexplore

GPUexplore is a fully GPU-based model checker, written in C and using the CUDA framework. It can check the reachability of states in a system, and can also check functional properties on-the-fly: currently, it can check for deadlocks and safety properties; the support of liveness properties is planned [5,13].
GPUexplore uses a GPU adaptation of the lockless hash table implementation by Laarman *et al.* [14]: When building the state space, it is important to store the states that have already been visited, for termination and performance reasons. As in parallel model checking this shared store is frequently used by multiple threads for lookups and insertions of states, using ordinary locks to enforce mutual exclusion would result in very poor performance: the contention is too high.
Laarman *et al.* developed a shared hash table store and associated lookup and insertion algorithm without locks and tailored the implementation to the specifics of multi-core model checking, including efficient use of the steep (implicit) memory hierarchy of modern CPUs. They show that the resulting performance is excellent and scales very well with the number of cores.
Several GPU implementations and variations of the algorithm exist [5,13,15,16,17,18]. As typically thousands of threads share the same data structure to store visited states, the contention is even higher here and designing an effective shared store is even more important for performance.
GPUexplore’s adaptation has been shown to be very efficient and scalable with more powerful GPUs; the resulting GPGPU state space exploration solution makes efficient use of the enormous power of modern GPUs and outperforms the most sophisticated multi-core state space exploration algorithms running on modern CPUs.

Hash table compression

Laarman *et al.* also developed a compressed hash table for multi-core CPU model checking [19]. The compression algorithm is built on top of the same lockless hash table implementation from [14], and allows for the compression of states in the hash table by sharing common components of the states. They show that this compression technique results in a giant reduction in the space needed for state space exploration with no performance penalty.
This algorithm would also be very relevant to GPUs, as GPU memory size (~10GB) is limited compared to main memory size (~100GB); as now algorithms exist that make efficient use of GPU’s enormous computing power, memory size becomes the new bottleneck, *e.g.*, for models with data.

1.1 Contribution

Our contribution relates to both the uncompressed and compressed hash tables.

Uncompressed hash table

While experimenting with porting compression to GPU-based hash tables, we encountered multiple flaws in the uncompressed hash table of GPUexplore and fixed them all.
First, we found the possibility for corruption in the hash table, *i.e.*, a hash table entry could be a mixture of two input values. This is disastrous for a model checker, which is often used to verify safety-critical systems. We fixed this issue by reverting to the hash table implementation of the initial version of GPUexplore [5].

This version, however, suffers from replication of entries in the hash table. Moreover, the used hash function results in an inferior distribution of entries across the table. Both flaws increase the (effective) table fill rate and decreases performance. This is very disadvantageous to the model checking process, which is very demanding. It also hinders a fair comparison to other hash table implementations, *e.g.*, hash tables without replication, such as our compressed hash table implementation.
We fixed both the replication and the inferior distribution. We removed the replication by using fine-grained locking. We integrated a very fast, but still mathematically grounded hash function [20] to get a probabilistic-optimal distribution. Our experiments show that the resulting performance is the same as or even better than that of the flawed implementation of GPUexplore. Our compressed hash table implementation, that is built on top of the uncompressed implementation, also benefits from the new hash function, again without a performance penalty.
Having removed the flaws from both the uncompressed and compressed hash tables, we were finally able to do a fair comparison.

Compressed hash table

We have successfully ported the multi-core compression algorithm to stand-alone GPU hash tables, using the CUDA framework. The main challenge was the recursion in the algorithm, as the call stack is saved to GPU device memory, which is relatively slow and has a large latency. Initially, this resulted in slowdowns up to 40x, compared to the uncompressed hash table. After optimising GPU and table parameters, we managed to reduce the slowdowns, with a maximal slowdown of 5.3x.
We even further reduced the slowdowns by implementing improved versions of the compressed hash table, aimed at reducing the recursive overhead: versions with less recursion and more work per recursive call. In this way, we reduced the recursive overhead to such an extent that now the scattered memory accesses required by tree-based hash table compression turned out to be the bottleneck.
Indeed, we designed and implemented versions without recursion, but the performance benefits over our most optimised recursive version were small, with a maximum of 14%. We created fifteen variants without recursion, that differ in aspects related to GPU performance. But they achieved almost the same performance each time, as they all share the same bottleneck of the scattered memory accesses.
Using the optimal version, the maximum slowdown is now 3.8x, small enough for performance-effective integration of compression into GPU model checkers, *e.g.*, GPUexplore, which would enable checking bigger models and models with data. Integration enables possibilities for reducing the amount of memory accesses required, possibly further reducing compression overhead.

Extensive performance evaluation

For our performance analysis, we used parameterised random data inputs and state sequences extracted from real-world models, up to 345M states. To make a fair performance evaluation, we first exhaustively examined the impact of GPU, input and table parameters on performance, of the uncompressed and all compressed versions. We used the results to get optimal settings for each input/program combination. All mutual comparisons use optimal settings for both contestants.

Summary of contribution

Thus, to summarise the major contributions of this thesis are:

- fixing the main flaws of the uncompressed hash table implementation by designing and implementing a replication-free hash table with a probabilistic-optimal distribution
- successfully porting tree-based compression to GPU-based hash tables by tackling the main performance limiter, *i.e.*, recursive overhead; enabling larger models and models with data in GPGPU model checking
- exhaustively examining the impact of GPU, input and table parameters on performance, of both the uncompressed and compressed hash tables; the results are used for finding parameters that are optimal for performance and they give an in-depth insight into the performance determiners of the tables, which can guide performance optimisation efforts when the tables are integrated into a GPU-based model checker such as GPUexplore

1.2 Overview of thesis

The remainder of this thesis is as follows.

Chapter 2 gives background information on model checking, GPGPU programming and hash functions. Chapter 3 discusses the CPU-based lockless hash table implementations, both uncompressed and compressed, and GPUexplore's adaptation of the uncompressed hash table. Then, Chapter 4 gives an overview of the project and the test setup.
Chapter 5 outlines our stand-alone implementation of the uncompressed GPU hash table, including several fixes and improvements to the original table. This chapter also features the performance evaluation of our fixed and improved stand-alone uncompressed GPU hash table, using parameterised random data. Chapter 6 repeats this for the recursive stand-alone compressed hash table implementations, including mutual comparisons and comparisons to the uncompressed hash table of Chapter 5. Next, Chapter 7 presents our solution for a non-recursive stand-alone compressed hash table, consisting of fifteen variants; the subsequent performance evaluation features mutual comparisons and comparisons to the uncompressed and recursive compressed hash tables from Chapters 5 and 6, respectively.
Chapter 8 highlights the practical results of our performance evaluation with a summary of the practical random-data experiments of Chapters 5-7 and with the performance analysis of data extracted from real-world models. Finally, Chapter 9 gives conclusions and suggestions for future work.

Appendix A has all information about the random data inputs we used and its generation.
Appendix B lists all experimental data of our exhaustive performance analysis.

2. Background

This chapter gives (more) background information on the topics that are used in the remainder of this thesis: model checking (Section 2.1), GPGPU programming (Section 2.2) and hash functions (Section 2.3).

2.1 Model checking

In model checking [1], a model, *i.e.*, the formal description of a system, implies a structure. This structure is a Kripke structure, an extension of a transition system; a transition system is a directed graph in which nodes represent system states and arcs represent transitions between states; one or more states are designated as initial states.
In a Kripke structure, a set of atomic propositions is defined and a labelling function maps each state to the set of atomic propositions that holds in that state. In model checking, each state is uniquely defined by the set of atomic propositions that holds in that state. Thus, if there are three atomic propositions, only eight states can be defined, as the size of the powerset of three elements is $2^3 = 8$.
The number of states thus grows exponentially with the number of atomic propositions; the set of *reachable* states, *i.e.*, the state space, however, is usually much smaller. In general, the state space is not explicitly given, but implicitly, by its specification; consequently, the state space size is not known *a priori*.

Traffic light example

As an example, consider this Kripke structure of the traffic light example from Chapter 1:

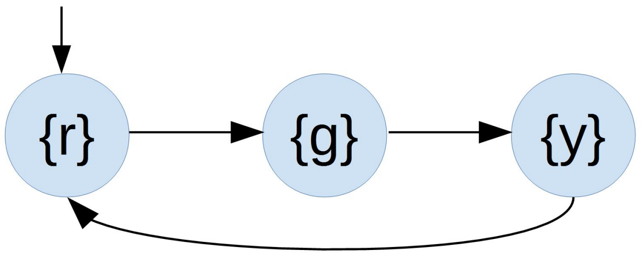


Figure 2.1: Kripke structure of a traffic light

where `r`, `g`, `y` are atomic propositions with the meanings ‘red light is on’, ‘green light is on’ and ‘yellow light is on’, respectively.
In the state on the left, the set `{r}` states that only proposition `r` holds (propositions `g` and `y` do not hold), meaning that the red light is on and that both the green and yellow lights are off. Therefore this is the ‘red state’, whereas the middle node refers to the ‘green state’ and the right node to the ‘yellow state’. Transitions are only possible from the ‘red state’ to the ‘green state’, from the ‘green state’ to the ‘yellow state’ and from the ‘yellow state’ to the ‘red state’; the ‘red state’ is the initial state.
With three propositions, eight states can be identified. In this example, only three out of the eight states are reachable, *e.g.*, a state with red and green lights on (`{r,g}`) is not reachable. State space exploration refers to the process of determining what states are reachable, *i.e.*, building the state space, starting from (the initial state in) a formal description of the system.
Multiple traffic lights, or processes, can be combined to get a more complex system, *e.g.*, by specifying how they interact together.

State vectors

In general, each state is defined by a state vector, each element of the vector representing the state of the corresponding process or value of the corresponding variable (in a system with variables/data). For example, in a model with integer variables `x` and `y`, a state in which `x`=2 and `y`=3 is represented by the vector (2,3) and a state in which `x`=4 and `y`=7 by the vector (4,7).
As variables can have a large number of values (*e.g.*, 2^{32} for a 32-bit integer), this often leads to *state space explosion*. There are ways to combat this problem, but, nevertheless, model checking is computationally very demanding and memory-intensive, even if elements of state vectors are restricted to a limited number of values (*e.g.*, with finite-state processes).

Properties

Modelling the system is one thing, defining the required properties is another. These properties are usually formalised in temporal logic, as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). The semantics of these logics are formally defined over Kripke structures. Algorithms exist that determine whether a model satisfies some LTL or CTL formula or does not; conceptually, they do a full search through the state space.
For example, the LTL property `G r` intuitively expresses that the proposition `r` should hold in every state (`G` stands for 'Globally'). This is clearly not the case in our traffic light example. However, the LTL property `F y` holds (`F` stands for 'Finally'), as for the only (infinite) execution possible `y` will hold eventually (in the 'yellow state').
Properties can be classified into safety and/or liveness properties (or none of them). Safety properties express that something "bad" should not happen. Invariants, as `G r`, are safety properties. Liveness properties assert that eventually something "good" will happen, *e.g.*, `F y`. Freedom from deadlock is another interesting property; deadlock occurs when there are no outgoing transitions from a state: the system is stuck.
To check for invariant violations and deadlocks, it is not always necessary to build the entire state space beforehand. When exploring the state space, this kind of properties can be checked on-the-fly; as soon as a property violation is detected, the model checking process can stop (and the user can correct the error in the system and/or modify the property). In this way, only one part of the state space needs to be explored, saving a lot of time while model checking complex systems.

Types of model checking

In explicit-state model checking, each state is represented individually, *i.e.*, by a state vector. For complex systems with a large state vector and/or large number of states, the space requirements would be enormous. Symbolic model checking manipulates *sets* of states, symbolically represented by boolean functions in the form of Binary Decision Diagrams (BDDs). Some model checking problems are well suited for symbolic model checking, others are best solved by explicit-state model checking.
As model checking is computationally and memory-wise very intensive, algorithms have been developed that make efficient use of the power of modern multi-core CPUs. These algorithms have been implemented in the state-of-the-art multi-core model checkers as LTSmin [3] and DiVinE [4]. Recently, model checkers have been developed that use the massive parallelism of GPUs to achieve even better results [5,6]. Section 3.1.2 discusses several GPGPU implementations of the CPU multi-core lockless hash table solution, discussed in Section 3.1.1.

2.2 GPGPU programming

GPGPU programming is different from CPU programming. For example, it features an explicit memory hierarchy, enabling for a software-managed cache. This section discusses the basic principles and constructs of GPGPU programming.
Two GPGPU programming frameworks are in widespread use: First, the cross-vendor and cross-platform Open Computing Language (OpenCL)[1] framework of the nonprofit technology consortium The Khronos Group [11]. Second, NVIDIA's Compute Unified Device Architecture (CUDA)[2] framework [12], which has support for NVIDIA GPUs only. As CUDA was available first and has some features that are not (natively) supported by OpenCL, it is still used extensively for GPGPU programming, *e.g.*, by GPUexplore [5].
First, we discuss the main differences between CPUs and GPUs and how they are combined in heterogeneous systems for efficient computation (Section 2.2.1). Then, we discuss OpenCL (Section 2.2.2), whose principles are also valid for CUDA. Finally, we discuss some CUDA-specific concepts (Section 2.2.3), in particular warps.

2.2.1 Heterogeneous systems

CPUs have sophisticated control logic to maximise the performance of a single thread, *e.g.*, by branch prediction and out-of-order execution. They also feature an extensive cache hierarchy, to decrease memory latency, also essential for single-thread performance. For years, that performance was further improved by increasing the clock frequency of CPUs. Due to power and thermal limits, CPU frequencies reached their maximum years ago. Instead, CPUs are now getting more and more cores. Still, the architectures of modern CPUs are latency-oriented [7].
GPUs, on the other hand, do not have sophisticated control logic or an extensive cache hierarchy. Instead, the transistors are used for featuring hundreds or even thousands of simple "cores". Due to the absence of an extensive cache hierarchy, latency is higher, but GPU memory bandwidth is also higher than that of main memory. As modern GPUs can execute thousands of threads concurrently (even more than the number of cores), zero-overhead context-switching of those threads hides the higher latency to a large extent: a thread waiting for data from memory is simply switched for a thread that was also waiting for data from memory, but whose data is now ready. The simple, but many, cores and efficient context-switching make GPU architectures throughput-oriented [7].
Due to the differences between CPUs and GPUs, each is suitable for a specific kind of problems: CPUs for control-intensive computations, GPUs for compute- and/or data-intensive computations that can be effectively parallelised. This parallelisation can be achieved by dividing the problem into multiple smaller, simpler subtasks (task-parallelism) and/or by executing the same operation on smaller subsets of data in parallel (data-parallelism) [7].

In heterogeneous computing, tasks that are most suited to GPUs are executed by those and more general purpose tasks by the CPU. Heterogeneous programming models and frameworks, such as OpenCL and CUDA, can be used for this purpose. Whereas CUDA supports off-loading (parallel) tasks to NVIDIA GPUs only, OpenCL supports off-loading to GPUs of various vendors, including AMD and Intel, and to other kinds of devices, such as CPUs and Field-Programmable Gate Arrays (FPGAs). This makes OpenCL code more portable, but its genericity makes it also more difficult to get the most performance out of a specific device [7].

[1] https://www.khronos.org/opencl/
[2] https://developer.nvidia.com/cuda-zone

Heterogeneous computing also refers to splitting the execution of a task over the CPU and GPU: the GPU may execute the task faster than the CPU, but as transferring data to and from GPU memory causes serious overhead, a balanced, combined execution may be faster than execution by the GPU alone. In some hardware architectures, the CPU and GPU share the same physical memory, so they can cooperate on a task even more closely, *e.g.*, the CPU executes the first (sequential) phase of a task and then the GPU executes the second (parallel) phase of a task directly on the (in-memory) results from the first phase, without the need of data transfer to and from separate GPU memory.
OpenCL and CUDA also support the concept of shared virtual memory, in which the CPU and GPU share a common (virtual) view of memory and the framework takes the responsibility for all necessary data transfers between main and GPU memory [7].

2.2.2 OpenCL

The OpenCL framework defines, among other things, an API and the OpenCL C programming language, an extended subset of C99, adapted to massive parallelism. Figure 2.2 gives an overview of the OpenCL platform model:

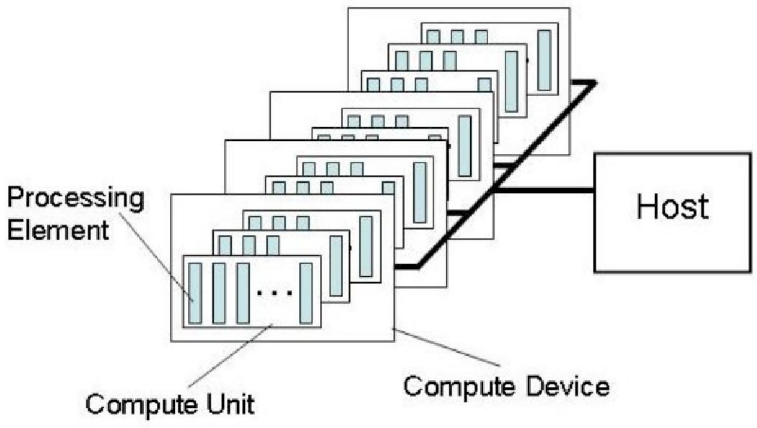


Figure 2.2: overview of OpenCL platform model (from [11])

The host program, running on the CPU, uses the API to communicate with one or more Compute Devices. Usually these are GPUs, but CPUs and FPGAs are supported as well. The Compute Devices execute OpenCL parallel code, called kernels, *i.e.*, device code. The API is used to send kernels to Compute Devices, send input data before execution, receive output data after execution, *et cetera* [21].
Compute Devices are usually composed of multiple Compute Units, which are similar to cores in multi-core CPUs. The coarse-grained Compute Units, on their turn, contain multiple fine-grained Processing Elements, *i.e.*, very simple cores. Modern GPUs consist of hundreds or even more of such Processing Elements [21].
In the OpenCL nomenclature, threads are called work-items. Work-items are grouped together into independent work-groups, enabling task-parallelism. The OpenCL kernel is parameterised by work-group and/or work-item id(s), allowing each work-item to work on different data, depending on its id(s), enabling data-parallelism. Each work-item has its own private memory and each work-group has also its own shared memory, called local memory. All work-groups share global memory. The host can only read from and write to global memory; global memory serves as an interface between host and device.

Work-items are usually mapped to Processing Elements and work-groups to Compute Units, but the actual implementation is up to the vendor of the Compute Device. The logical memory spaces private, local and global memory are usually mapped to Processing Element-local memory, *e.g.*, registers, to Compute Unit-local memory and to device memory, respectively [21].

Execution model

Multiple work-groups usually execute the same *program*, but on different data. This execution model is called Single Program, Multiple Data (SPMD). Work-items in a workgroup usually execute the same *instruction*, but on different data, according to the Single Instruction, Multiple Data (SIMD) execution model. To be more precise, its execution model is called Single Instruction, Multiple Thread (SIMT), as the work-items can have different control flows, *e.g.*, when some work-items take the true-branch and the others the false-branch of an if-statement. This is called branch divergence [22].

Achieving optimal performance

Global memory is very large (GBs), but its latency is also very high; private memory is very small, but has a very low latency. Local memory is somewhere in between. To get optimal performance, especially for memory-bound programs, access to global memory and, to a lesser extent, local memory, should be considered carefully [21].
In general, it is optimal for work-items to operate on adjacent memory locations, as their data can be fetched in just one memory transaction: usually, a cache resides between the Compute Units/Processing Elements and device memory; if the requested memory locations all correspond to one cache line, only one memory transaction is needed. For local or private memory operations this notion is less important [7].
For optimal performance, branch divergence should be avoided: in the SIMT execution model, when one group of work-items executes one branch, the group that will take the other branch, just waits till the other group has finished executing their branch, resulting in suboptimal performance [7]. A related, even more serious issue is barrier divergence, explained in more detail on page 16.
To keep as most Processing Elements busy as possible, a large number of work-items should be used (thread-level parallelism). Another way to achieve this, are more independent instructions within a kernel, *e.g.*, by unrolling loops (instruction-level parallelism). Work-group sizes are also important in achieving the best performance: they should not be too small or too big. Finding optimal work-group sizes and total number of work-items are guided by heuristics, but still requires experimentation.

Data races and race conditions

As work-items usually execute the same instruction, this introduces another possibility for data races. In general, a data race occurs when two or more threads access the same shared memory location concurrently and at least one of these threads does a write action. A related, but different, concept is a race condition: in that case, the correctness of a program depends on the timing or ordering of events.
For example, consider the concurrent execution of the statement `x := x+1` by two threads (where `x` is an integer, initialised to zero), possibly running on two different CPU or GPU cores. This example suffers from both a data race and a race condition: both threads read from and write to variable `x` concurrently and then execution is undefined.

Even if no data race actually takes place, the program has a race condition: Depending on the exact thread interleaving, `x` would be 1 or 2 after completion of both threads, as addition is not atomic here. As thread interleavings can be different for every execution of the program, errors due to data races and race conditions are often very difficult to reproduce and may not manifest during testing, but can be often formally detected.

Weakly-ordered memory model

The weakly-ordered memory model of OpenCL, in which memory operations can be reordered to a large extent, introduces even more possibilities for a data race to occur: the order in which a work-item writes to memory locations may not be the same as the order observed by another work-item. For example, consider the parallel execution of

```
x := 30
y := 40
```

and

```
r1 := y
r2 := x
```

by two different work-items, where `x` and `y` are shared variables, both initialised to zero. OpenCL's memory model allows the second work-item to read 40 for `y` and 0 (not 30) for `x`. Memory fences can be used to enforce (some) memory ordering (at a performance penalty): For example, if memory fences are placed between both assignments in both code fragments, reading 40 for `y` guarantees reading 30 for `x`. (Note that both fences are required.) Memory fences also make previous updates by the work-item visible to the other work-items, by flushing them from work-item-local caches to shared memory: global and/or local memory, depending on the memory type(s) specified [11].

Barriers

To be more precise, the example above would still contain data races, so execution is undefined. Barriers can be used to fix this issue, as they are used to synchronise between work-items in the same work-group: each work-item waits at the barrier until all other work-items in the work-group have reached the same barrier; only then the execution of all work-items in the work-group, is continued. Fences for local and/or global memory can be defined at barriers [11].
Appropriate use of barriers is crucial for avoiding data races and race conditions, as, in general, no assumptions can be made about work-item interleavings. Some work-items may execute in lockstep, *i.e.*, they will execute the same instruction in parallel, but usually this is only a subset of all work-items in a work-group [11]. But, as barriers also result in serious overhead, they should only be used if they are really required.
As an example, consider this kernel (all example kernels use a simplified syntax):

```
void fenceExample(global int a[], local int b[]) {
      b[ltid] = ltid;
      barrier(CLK_LOCAL_MEM_FENCE);
      a[ltid] = b[(ltid+1)%gsize];
}
```

Code fragment 2.3: barrier and memory fence example

where ltid is the local work-item id (id of work-item local to work-group) and gsize the work-group size; a is an int-array in global memory and b is an int-array in local memory, shared by all work-items in the work-group.
Suppose there is only one work-group. After execution of the kernel, a[ltid]'s value will be (ltid+1) modulo gsize, for every 0 ≤ ltid < gsize, *e.g.*, a[0] = 1. If we remove the barrier, we cannot guarantee this outcome anymore, as there will be a data race on the entire b array; the memory fence is needed to ensure that work-item ltid indeed reads the value set by work-item ltid+1 (modulo gsize) before.

Atomics

Barriers can only be used to synchronise between work-items in the same work-group; atomics can be used to coordinate concurrent read/write operations on the same memory location between work-items in different work-groups (or the same), even across Compute Devices.
Atomic operations, such as atomic addition, are executed atomically, *i.e.*, they are never interrupted, in contrast to normal addition as x = y+z. The latter operation is usually executed as a sequence of multiple instructions, which can be interrupted, *e.g.*, between (1) fetching the values of y and z, and (2) the addition and storing of the result to x. Atomics can be seen as very fine-grained locks and are usually implemented on the hardware level [11].
As an example, consider this kernel:

```
void atomicsExample(global int values[], global int sum) {
      atomic_add(sum, values[gtid]);
}
```

Code fragment 2.4: atomics example

where gtid is the global work-item id; values is an int-array in global memory and sum is an int variable in global memory.
After execution of the kernel, sum's value will be the sum of values[0]..values[ksize], where ksize is the total number of work-items. This works even if the work-items are grouped together into multiple independent work-groups.

Sequential consistency

OpenCL 1.2 supports atomic read-modify-write functions only; OpenCL 2.0 adds support for atomic load (read/get) and store (write/set) functions [11]. Concurrent atomic operations are not considered racy, whereas a concurrent *non-atomic* load/store operation and an atomic read-modify-write operation on the same variable is racy. Atomic operations always operate on the actual memory locations (not on cached versions), so their results are immediately visible to other work-items (provided that the other work-items do not use non-atomic loads) [23].
In OpenCL 2.0, it is now possible to specify a memory order for an atomic operation. At one extreme, this is still the original weak memory model; at the other extreme, it corresponds to the sequential consistency memory model [11]. The latter is the most easy and intuitive memory model for programmers to reason about programs. In this model, the execution of a concurrent program corresponds to some (global) sequential interleaving of the operations of all threads (work-items) and the order of operations of each thread in this interleaving is the same as the order in the thread's code [24].

OpenCL 2.0 guarantees that an OpenCL kernel without data races and in which all atomic operations utilise the sequential consistency memory model, *appears* to execute with sequential consistency [11], *i.e.*, the execution is serialisable.

Barrier divergence

Barrier divergence occurs when one subset of the work-items in a work-group waits at one barrier and the other subset at another barrier (or at none at all) [22], for example, in this code fragment:

```
if (a[ltid])
      barrier(...)
else
      barrier(...)
```

Code fragment 2.5: barrier divergence example

where `ltid` is the local work-item id and `a` is a bool-array in global memory. If `a[ltid]` evaluates to true for some work-items and false for others, barrier divergence occurs and execution of the kernel is undefined.
In (nested) loops, things get even more complicated: For example, consider a main loop containing a barrier and a nested loop containing another barrier. What happens if all work-items in a work-group execute the same number of barriers in total, but the *distribution* of executions over both barriers is different for subsets of work-items? Indeed, OpenCL implementations by different vendors give inconsistent results [22].
In conclusion, barrier divergence should be avoided.

2.2.3 CUDA

Most of the principles discussed above also apply to CUDA, although the nomenclature is different: Compute Units are Streaming Multiprocessors (SMs) and Processing Elements are CUDA cores. Work-items are called CUDA threads and work-groups are called blocks. All blocks/threads allocated to a kernel are collectively referred to as the grid; by specifying grid and block dimensions (execution configuration) of a kernel, one specifies the number of blocks and the number of threads in each block, respectively [12].
Per-thread local memory is usually stored in fast on-chip registers, but large arrays, *et cetera*, are stored in local memory, which resides in (slow) device memory. Block-local memory is called shared memory, which is partitioned into 32-bit wide banks; two or more threads in a warp (see next) should not access the same bank in parallel [12].
As (multiple) blocks are assigned to a fixed, single SM, its shared memory is partitioned among thread blocks resident on the SM and its registers are partitioned among threads of those blocks. This restricts the number of blocks (and threads) that can be resident on a single SM; registers may be *spilled* to local memory to make room for other blocks. Resources are only released when a block fully completes execution; only then a new block can be loaded onto the Streaming Multiprocessor [12].
Global memory resides in device memory, but is usually accessed via a device-wide L2 cache and, depending on configuration, also via per-SM L1 caches. The read-only memory spaces constant and texture memory also reside in device memory, but have their own caches and are optimal for specific memory access patterns [12].
In host code for setting up and communicating with the GPU device, CUDA makes more use of extensions to the C language, whereas OpenCL uses more explicit API calls [12].

Warps

CUDA also features a concept that is not natively supported by OpenCL: warps. Threads in a block, usually 32, are grouped together into warps. Originally, the threads in a warp operate in lockstep, thus they are implicitly synchronised, instead of explicitly by a barrier. This can be used for intra-warp communication via shared memory, by declaring it volatile. When memory is declared volatile, it is always accessed directly; otherwise, it is possible that memory operations work on thread-locally cached versions instead [12].
When the example from Code fragment 2.3 (page 14) is executed by a single warp (`gsize` = 32) and `b` is declared volatile, the barrier, and included memory fence, are not necessary anymore, resulting in Code fragment 2.6 on this page: due to the lockstep nature of warps, data races on `b` cannot occur and as `b` is declared volatile, the values written to `b` are read subsequently. Removing the barrier saves a serious amount of explicit synchronisation overhead.

```
void warpSynchronous(global int a[], local int b[]) {
      b[ltid] = ltid;
      a[ltid] = b[(ltid+1)%gsize];
}
```

Code fragment 2.6: barrier and memory fence example (warp-synchronous)

Although this so-called warp-synchronous programming was advocated by NVIDIA for a long time, it is now considered unsafe [25]. Indeed, NVIDIA's newest Volta architecture has Independent Thread Scheduling (ITS) [12]. Many CUDA kernels, explicitly or implicitly, however assume a warp size of 32 threads.
Also, reordering of memory operations in the execution of the warp is still allowed by the memory model. In Code fragment 2.6, this issue can be addressed by putting a memory fence at the location of the removed barrier. In CUDA, memory fences are only used for memory ordering; to make memory updates visible to other threads, it is still needed to declare relevant memory volatile (in our example, the `b` array) or to use a barrier (which, in CUDA, always also includes a global and shared memory fence).
Now, NVIDIA states that CUDA kernels should not rely on implicit warp synchronisation anymore and that intra-warp communication should take place via warp-level primitives [25]. The CUDA model checker GPUexplore [5], however, still uses warp-synchronous programming, to achieve optimal performance.
Note that the (disadvantageous) effects of branch divergence can only appear within warps (as only they can operate in lockstep) [22]; the (disastrous) effects of barrier divergence are not restricted to one warp, but to all warps/threads in a block.
AMD, NVIDIA's main discrete GPU competitor, calls warps wavefronts, but it is not possible to address them directly via OpenCL, which AMD uses for GPGPU programming on their GPUs. OpenCL 2.0 introduces sub-groups, which are very similar to warps (and will probably be mapped to wavefronts, on AMD GPUs, and warps, on NVIDIA GPUs). This is, however, an optional extension [23].

Atomics

In CUDA, only atomic read-modify-write operations are available, no atomic load or store operations [12]. Therefore, for a load operation, only a non-atomic version is available. A concurrent execution with an atomic read-modify-write operation, such as an atomic compare-and-set, is, consequently, considered racy in the strict sense.

2.3 Hash functions

A hash function maps a, usually large, universe of values to a, usually much smaller, range of values. A hash function can, for example, be used in a hash table to index the (primary) bucket for a given input key. For many applications, it is necessary that the hash function is fast, *e.g.*, because it is called often. For most applications of hash functions, it is also necessary that the output values have a good distribution over the input keys, even with patterns in the input, *i.e.*, that each key is assigned a pseudo-random output value.

Universal hash function

A universal hash function captures this notion: a random hash function h is called universal if the probability that two arbitrary, but distinct keys are hashed to the same output value (called a collision) is $\leq 1/m$, where m is the number of possible output values; h is usually a family of functions, parameterised by random constants. Strong universality takes this notion even further, to pairwise independence: a random hash function h is called strongly universal if the probability that key x hashes to arbitrary z_1 and distinct key y hashes to arbitrary z_2 is $1/m^2$ [20].

Multiply-shift

A fast and strongly universal function family for hashing integers is called 'Multiply-shift'. This (partial) instantiation of the function hashes 32-bit integer `x` to a 32-bit value:

```
(a * x + b) >> 32
```

where `a` and `b` are random 64-bit seeds. Multiplication (and addition) is 64-bits here, discarding any overflow [20].

A generalisation of this universal function family allows to hash vectors of 32-bit integers to a 32-bit value:

```
((a0 * x0) + (a1 * x1) + (a2 * x2) + ... + b) >> 32
```

where `x0`, `x1`, `x2`, … are the 32-bit elements of the vector and `a0`, `a1`, `a2`, …, `b` are random 64-bit seeds [20].
The 'Pair-multiply-shift' trick substitutes one (fast) addition for one (slow) multiplication, for each pair of vector elements. For example, the partial hash `(a0 * x0) + (a1 * x1)` is now being calculated by `(a0 + x1) * (a1 + x0)`. If the number of elements in a vector is odd, the partial hash value for the last element can be calculated in the ordinary 'Multiply-shift' way, *i.e.*, by a single multiplication [20].

Hashing to arbitrary ranges

The above functions hash to a 32-bit value. A modulo operation can be used to get a smaller range of output values, but this operation is expensive. Instead, this function can be used to restrict the 32-bit (hash) value `h` to a smaller range of `m` values:

```
(h * m) >> 32
```

Multiplication and especially shifting operations are cheap, in particular compared to the modulo operation. The restricted value is still pseudo-random. For this to work, multiplication should again be 64-bits (discarding any overflow) [20].

3. Previous work: lockless hash tables

Reachability, *i.e.*, building/exploring the state space, is a subtask of many verification problems in model checking (Section 2.1), but it can also be used stand-alone, for example to detect deadlocks or invariant violations on-the-fly. Starting from the initial state, the successor states are determined from the system's formal description, the successor states of those states are determined, *et cetera*. The states whose successors still need to be determined are kept in the so-called open set, which can be implemented as a stack (depth-first search) or as a queue (breadth-first search) [1].
To remember the states whose successors have already been determined, a so-called closed set is used. This is needed for performance and termination, as states are often on a cycle in the state space graph and would otherwise be visited infinitely often. A hash table can be used to implement the closed set. Note that we consider explicit-state model checking here [1].
In a concurrent reachability algorithm, each worker/thread works on its own part of the open set. Various algorithms exist for load balancing. As the closed set's implementation, the hash table, is, however, shared by all threads, it is critical to ensure thread-safety, *e.g.*, that data races and race conditions are not possible. Using ordinary, coarse-grained locking, would result in very poor performance, due to the high contention. Instead, a lockless way, *i.e.*, a synchronisation mechanism without locks, should be used [14].

In this chapter, we explain the already existing implementations of lockless hash tables, both CPU multi-core and GPGPU, in detail. We start with the multi-core uncompressed hash table and its existing GPGPU implementations (Section 3.1). Then, we discuss the multi-core compression algorithm that is built on top of this hash table (Section 3.2); no GPGPU implementation exists yet.

3.1 Uncompressed hash table

This section first describes the CPU multi-core uncompressed hash table (Section 3.1.1) and then its existing GPGPU implementations (Section 3.1.2).

3.1.1 Multi-core implementation

Laarman *et al.* [14] developed a hash table design and associated operations that are optimised for modern multi-core hardware architectures. The size of main memory is usually very big, but its latency is also high. Modern multi-core CPUs compensate this latency by using multiple levels of cache; some of these caches are local to the core, some are shared among multiple or all cores.

To ensure that each core has the same global view of memory, cache coherence protocols are used: if a core modifies the contents of a memory location and that memory location has been cached in some (other) local cache before, the protocol ensures that the local cache is updated, resulting in serious overhead. This introduces the problem of cache line sharing (also known as false sharing): if two cores work on different memory locations that are accidentally located at the same cache line, the extensive, but unnecessary, application of the cache coherence protocol causes a great amount of overhead.

One of the ways to mitigate this overhead, is to minimise the memory working set, *i.e.*, the number of different memory locations the algorithm updates in the time window that these usually stay in the local cache. This is one of the key factors used in the design of Laarman's hash table, next to simplicity whenever possible.
As the closed set grows monotonically, the hash table needs only one operation: `find-or-put`, with a state vector as argument (so, the table key is also the data). This operation is used to insert the data (state vector) into the table (closed set). The function returns true when the state vector is already in the hash table; when it is not, the function returns false and the state vector is being added to the hash table. We first present the general working of the find-or-put algorithm, then we describe the data structure the actual algorithm operates on. Finally, we explain the algorithm in detail.

Overview of the `find-or-put` algorithm

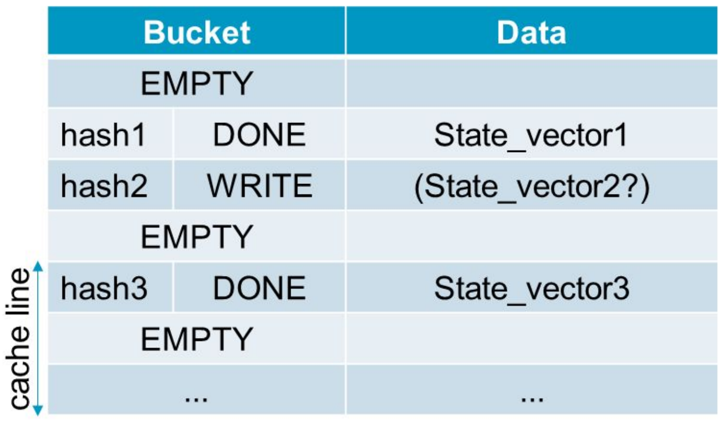
The elements of the hash table are called buckets: a bucket is empty or contains a state vector. The algorithm hashes the state vector and uses this hash to index the table.

- If that bucket is not empty, its contents will be compared to the vector; as two different vectors may hash to the same value (called a collision), the bucket may contain a different vector. If this is the case, the next bucket will be examined; otherwise the algorithm concludes that the vector is already present in the table and returns true.
- If the bucket is empty, the algorithm concludes that the vector does not exist in the table yet and tries to insert the vector by claiming the empty bucket. This may fail, as a concurrent invocation of the algorithm may try to do the same and only one succeeds.
    - If it succeeds, it returns false, indicating that the vector was not present in the table yet and has now successfully been inserted.
    - In the other case, it will compare its vector to the vector just inserted by the concurrent invocation as both invocations may have tried to insert the same vector. If this is indeed the case, true will be returned, as the vector is now present in the table and has not been inserted by this invocation; otherwise, the next bucket will be examined.

Any examining of next buckets will work in the same way as for the initial bucket.

Hash table design

Figure 3.1 gives an overview of the data structure of the actual hash table, which the `find-or-put` algorithm operates on. Its elements are explained in turn.



Bucket		Data
EMPTY		
hash1	DONE	State_vector1
hash2	WRITE	(State_vector2?)
EMPTY		
hash3	DONE	State_vector3
EMPTY		
...		...

Figure 3.1: overview of the design of the hash table

- To handle hash collisions, open addressing is used, instead of chaining. So, when a non-existent state vector hashes to an index that has already been occupied in the hash table, a different index will be determined repeatedly, till a free spot is found; chaining handles hash collisions by a linked list. Chaining would require in-operation memory allocation, resulting in a larger memory working set.
- Walking-the-line means linear probing on the cache line (benefitting from an already loaded cache line), followed by (bounded) double hashing (better distribution), ensuring worst-case constant time complexity. So, if a bucket is occupied by a different vector, the next bucket in the same cache line will be tried, then the next next one, *et cetera*. If all buckets in the cache line are occupied by different vectors, the vector is rehashed with a different hash function to get a new bucket index.
- A separate data array, whose length is the same as the number of buckets (= length of bucket array), stores the actual state vectors, which can be large. This ensures that the bucket array stays short and can be cached to a large extent, speeding up subsequent probes. The bucket with index `i` corresponds one-to-one to the element in the data array with the same index.
- Hash memoisation stores (a part of) the hash in the bucket array. In most occasions, comparing hashes suffices to conclude that the probed state vector is different from the one stored in the data array. This saves a lookup in the (large) data array. Hash memoisation is useful because there is no one-to-one correspondence between hashes and bucket indices, *e.g.*, due to open addressing.
- Lockless operation on the bucket array: a dedicated value is used to indicate EMPTY buckets; one bit of the memoised hash is used to indicate that the state vector has been written to the data array (DONE) or that writing is still in progress (WRITE).
- Compare-and-swap is used as the operation to change each bucket atomically from EMPTY, via WRITE, to DONE. This is the only sequence possible.
  The compare-and-swap operation `CAS()` has three arguments: `mem_loc`, `old` and `new`. If the value at `mem_loc` is `old`, `new` is written at `mem_loc` and the operation returns true; if the value at `mem_loc` is not `old`, nothing is written and the operation returns false. This is all done atomically. As the `CAS()` operation costs multiple instruction cycles, it should be used with care.

`find-or-put` algorithm in detail

Algorithm 3.2 (next page) lists the `find-or-put` algorithm.

The variable `count` is used to count (line 15) and index the various hash functions (lines 2 and 16); `THRESHOLD` is used to limit the amount of different hash applications (line 4). The function `walkTheLineFrom(index)` returns all indices in the same cache line as `index`, in a circular way, starting from `index` itself and ending with `index`-1. They are used in the `for`-loop starting at line 5.

If an empty bucket is found while walking-the-line (line 6), this means that the state vector is not in the hash table yet. Using atomic compare-and-swap (line 7), the algorithm tries to claim the empty bucket by changing it from EMPTY to WRITE (and storing its hash). If this succeeds, the actual state vector is written to the data array (line 8). Now, the bucket is changed to DONE (line 9) and false is returned (line 10), to indicate that the state vector was not present in the hash table yet and has been inserted by the current invocation of the `find-or-put` operation.

```
Data: size, Bucket[size], Data[size]
input  : vector
output : seen
1  $count \leftarrow 1$;
2  $h \leftarrow \mathtt{hash}_{count}(vector)$;
3  $index \leftarrow h \mod size$;
4  while $count <$ THRESHOLD do
5      for $i$ in $\mathtt{walkTheLineFrom}(index)$ do
6          if EMPTY $= Bucket[i]$ then
7              if $\mathtt{CAS}(Bucket[i], \text{EMPTY}, \langle h, \text{WRITE} \rangle)$ then
8                  $Data[i] \leftarrow vector$;
9                  $Bucket[i] \leftarrow \langle h, \text{DONE} \rangle$;
10                 return false;
11         if $\langle h, - \rangle = Bucket[i]$ then
12             while $\langle -, \text{WRITE} \rangle = Bucket[i]$ do ..wait.. done
13             if $Data[i] = vector$ then
14                 return true;
15     $count \leftarrow count + 1$;
16     $index \leftarrow \mathtt{hash}_{count}(vector) \mod size$;
```

Algorithm 3.2: multi-core `find-or-put` algorithm (from [14])

Claiming an empty bucket (line 7) may also fail, as another, concurrent, invocation of the `find-or-put` operation may try to claim the same bucket and only one succeeds.
This other invocation may even insert the same state vector. In that case, the non-succeeding concurrent invocation will see at line 11 that the memoised hash just set by the other invocation is the same as its own hash. It then waits till the succeeding invocation finishes writing the state vector to the data array (line 12, resembling a spinlock); afterwards, it concludes that the state vector has (just) been inserted (line 13) and returns true (line 14).
If the succeeding concurrent invocation inserts a different state vector, the non-succeeding invocation is able to conclude this at line 11, based on the memoised hash, or otherwise at line 13. It then continues by proceeding to the next bucket (lines 5-14).

When the state vector that is being probed has been inserted already a long time before, the algorithm sees that its hash is the same as the memoised hash at line 11. As the bucket is now already in its DONE state, it proceeds immediately to line 13 and it can conclude that the state vector was already present in the hash table (line 14).

Essentially, locking takes place at the bucket level and is implemented by the `while`-loop at line 12. However, due to the hash memoisation check at line 11, the loop is rarely hit under normal circumstances.
The algorithm requires exact guarantees from the underlying memory model. For example, if the memory operations at lines 8 and 9 are re-ordered, the correct working of the algorithm cannot be guaranteed anymore.
The model checking algorithm has been model checked itself for deadlocks. Indeed, one bug was found and corrected.

This lockless hash table has successfully been implemented in the multi-core model checker LTSmin [3].

3.1.2 GPGPU implementations

Various GPGPU implementations of the multi-core algorithm outlined above exist: one by Neele [15], multiple variations implemented in GPUexplore [5,13] and one by Verkleij [16]. In this subsection, we discuss the differences of each to the original algorithm.

The first one, by Neele [15], is a one-to-one implementation of the original algorithm in OpenCL. The achieved speedup using up to 256 work-items is almost linear.
Speedup expresses the relative performance of two systems working on the same problem and is often used in assessing the (relative) performance of parallel processing. For example, when running *n* threads in parallel is *n* times faster than running one thread only, linear speedup is achieved. This implies excellent scalability.

GPUexplore

Wijs *et al.* [5] developed a version that is more adapted to the specifics of GPUs in their explicit-state on-the-fly model checker GPUexplore[3], implemented in NVIDIA's CUDA. GPUexplore is the first model checker that uses GPUs for building the state space; other GPGPU model checkers are a hybrid: the state exploration algorithm runs on the CPU, the actual checking of the properties is done on the GPU. Therefore, those are not on-the-fly model checkers, whereas GPUexplore can check on-the-fly for deadlocks and safety properties [26].

GPUexplore's `find-or-put` operation

Algorithm 3.3 lists GPUexplore's `find-or-put` pseudocode:

```
extern volatile __shared__ unsigned int $cache$ []
< process work tile and fill cache with successors >
$WarpNr \leftarrow ThreadId\ /\ WarpSize$
$WarpTId \leftarrow ThreadId\ \%\ WarpSize$
$i \leftarrow WarpNr$
while $i < |cache|$ do
    $\overline{s} \leftarrow cache[i]$
    if isNewVector($\overline{s}$) then
        for $j = 0$ to $H$ do
            $BucketId \leftarrow h_1(\overline{s})$
            $entry \leftarrow Visited[BucketId + WarpTId]$
            if $entry = \overline{s}$ then
                setOldVector($cache[i]$)
            $\overline{s} \leftarrow cache[i]$
            if isNewVector($\overline{s}$) then
                for $l = 0$ to $WarpSize$ do
                    if $Visited[BucketId + l]$ = empty then
                        if $WarpTId = 0$ then
                            $old$ = atomicCAS($\&Visited[BucketId + l]$, empty, $\overline{s}$)
                            if $old$ = empty then
                                setOldVector($\overline{s}$)
                    if ¬isNewVector($\overline{s}$) then
                        break
            if ¬isNewVector($\overline{s}$) then
                break
            $BucketId \leftarrow BucketId + h_2(\overline{s})$
    $i \leftarrow i + BlockSize/WarpSize$
```

Algorithm 3.3: GPUexplore's `find-or-put` algorithm (from [5])

[3] https://www.win.tue.nl/~awijs/GPUMC/index.html

In this pseudocode, the state vectors are an integer (32-bit) wide, whereas in the actual tool implementation wider state vectors are supported.

As cache coherence protocols do not exist (yet) in GPUs, minimising the memory working set is less important, *e.g.*, no separate data array is used. Each bucket is 32 times 32-bit wide, corresponding to exactly one GPU cache line, and separated into slots, each capable of storing exactly one vector; with 32-bit vectors, each bucket has 32 slots. The `cache` (line 1) is shared by all CUDA threads that belong to the same thread block. CUDA threads add discovered successors to that cache in shared memory. In this way, local duplication detection takes place (saving much slower probes in the global hash table), *i.e.*, if two threads find the same successor state, it is only probed at the global hash table once. The local cache is also implemented as a lockless hash table.

Each warp is assigned a state vector from the `cache` (line 7). The 32 threads in the warp then read one full bucket in parallel (line 11). This is called *warping*-the-line instead of *walking*-the-line and is a type of data-parallelism. As a bucket corresponds to exactly one cache line, memory access is aligned (request's first address is a multiple of the load granularity) and coalesced (contiguous), and it can be fetched in one memory operation. If one of the 32 threads in the warp finds a match (line 12), this is recorded by `setOldVector(cache[i])` (line 13). Then, for every warp thread, `isNewVector(`$\bar{s}$`)` at line 15 evaluates to false, `!isNewVector(`$\bar{s}$`)` at line 24 evaluates to true, and the warp is assigned another state vector from the cache (line 7), if there are any left (line 6).
If none of the 32 threads find a match (detected at line 15), the warp leader (thread id in warp is 0) tries to insert the state vector into the first empty bucket slot (line 19); as concurrent warps may try to claim the same empty slot, this may fail and then the warp leader tries the next empty bucket slot.

The algorithm uses warp-synchronous programming, *i.e.*, it exploits the implicit synchronisation of all threads in a warp, saving the overhead from explicit synchronisation; coordination with threads in other warps (possibly in different blocks) takes place via the `atomicCAS()` operation (which, in the CUDA version, returns the old value). The shared `cache` is declared `volatile`, thus updates to `cache` are never cached thread-locally and no reads from `cache` will be handled locally.

Race condition

The actual implementation has a race condition: When the state vectors (and, consequently, bucket slots) are wider than 64 bits, the entire slot cannot be claimed by an `atomicCAS()` operation anymore, as that operation supports only integers up to 64-bit; only a part of the slot can be claimed. When two warps (from different blocks) are trying to insert the same vector, into the same (empty) slot, only one succeeds in claiming the first part of the slot. If the non-succeeding warp does not wait till the succeeding one has also filled the other entries of the slot, it cannot detect that its state vector is already being inserted, leading to a so-called false negative. This results in some states being visited multiple times and also in replication in the hash table.
The race condition can be removed by introducing a spinlock solution, similar to the one in the original algorithm. However, it turned out that the overhead introduced by the spinlock harms performance more than visiting some states multiple times; in practice, only a small part, about 2%, of the states will be re-visited.

GPUexplore 2.0 [13] fixes the race condition by observing that atomic memory requests are scheduled in half-warps. Padding can be used to ensure that slots are not crossing a half-warp/half-bucket boundary. Then all threads in a warp that work on the same slot, *i.e.*, a vector group, try to claim the *full* slot by doing an `atomicCAS()` operation each, in parallel; only (the threads from) one warp will succeed in claiming the full slot. Any competing warp that tried to insert the same state vector at the same time, into the same slot, but failed, can detect immediately that the state vector has just been inserted, by comparing its elements to the return values of the `atomicCAS()` operations.

GPUexplore 2.0 also shows that it scales very well: it was much faster running on a new generation of GPUs; this is only possible when its hash table scales as well.

Selecting empty slot by data race

Verkleij [16] introduces a data race to the original GPUexplore implementation: if the state vector is in none of the slots, it selects an empty slot by data race. The warp's thread belonging to that slot then tries to claim it. In the original implementation, outlined above, only the warp leader tries to claim empty slots, starting with the first empty one encountered. As noted, data races should be avoided.

Configurable bucket size

Cassee *et al.* [18] made an extension for more data-parallelism to the hash table design of GPUexplore 2.0, allowing for a configurable bucket size: instead of a fixed bucket size of 32 integers, a lower bucket size can be configured, allowing (the threads in) a warp to operate on multiple vectors and buckets in parallel. For example, with a bucket size of 8, a warp can operate on four buckets and four vectors in parallel, *i.e.*, a warp now consists of four bucket groups of eight threads each.
In general, this reduces the amount of inactive threads in a warp, but it also leads to uncoalesced memory access, as the (smaller) buckets that are examined in parallel by a warp, are most likely not located next to each other in global memory. Further, divergence can increase, as one bucket group may need to examine more slots, possibly including rehashing, before finding a free slot than another bucket group needs to. Independent Thread Scheduling may reduce the negative effects of divergence.
Bucket sizes cannot be smaller than the length (number of elements) of each input vector. Lower bucket sizes may lead to "lost" hash table entries. For example, with a vector length of 3 elements/integers, a bucket with size 32 can store up to 10 vectors, whereas four buckets with size 8 can only store up to 4 * 2 = 8 vectors.
Results showed that a lower bucket size was faster with input with many duplicated vectors (duplication factor of 25 or more), using stand-alone hash tables. However, integrated into GPUexplore, performance was worse in almost all cases.

3.2 Tree-based compression

In practice, state vectors usually have many subvectors in common with other (reachable) state vectors. For example, in a successor state often only one element (slot/variable) of the state vector changes, compared to its originating state, and the originating and successor states have all other elements in common. This observation can be used to dramatically reduce the memory space required in explicit-state model checking, *e.g.*, to compress the closed set's hash table, by sharing common subvectors among state vectors.

Compressed tree database

In its original form [27], each vector is represented by a balanced binary tree, as can be seen in Figure 3.4: Pairs of vector elements are grouped together in hash tables at the fringe of the tree, returning a reference (an index in the hash tables). Then a pair of those references are grouped together in another hash table, returning another reference, *et cetera*. Thus, in this example seven hash tables are used, of which four are at the fringe. Common subtrees are shared, providing the compression.

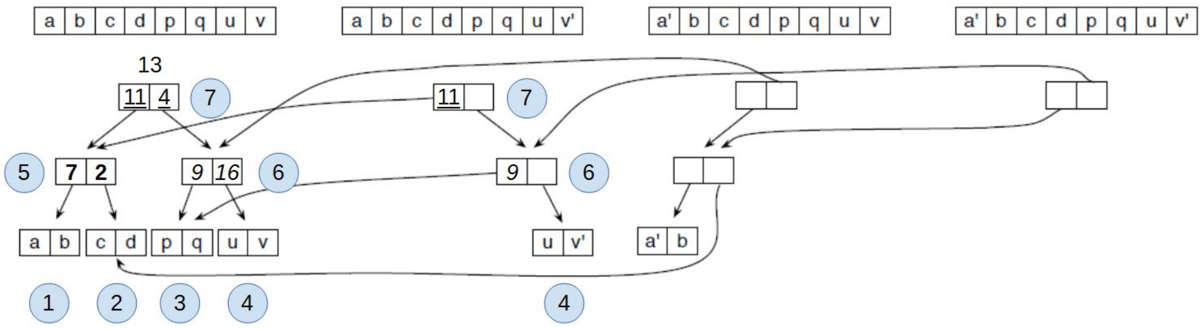


Figure 3.4: tree compression (adapted, from [19])

Using this tree structure, a `find-or-put` operation can be implemented.
Applying this operation on the (a,b,c,d,p,q,u,v) state vector in a fresh situation (all hash tables empty), results in four references at the fringe (see Figure 3.4). For example, the pair (a,b) is hashed to index **7** in hash table #1, (c,d) to **2** in hash table #2, (p,q) to *9* in hash table #3 and (u,v) to *16* in hash table #4. Then, the reference pair (**7**,**2**) is hashed to 11 in hash table #5 and (*9*,*16*) to 4 in hash table #6. Finally, the reference pair (11,4) is hashed to 13 in hash table #7 (root of tree).
Now suppose a subsequent application on the (a,b,c,d,p,q,u,v’) vector (see Figure 3.4). The pair (a,b) now already exists in hash table #1, so instead of creating a new item in the hash table, a reference to the existent pair is returned. The same happens for the pair (c,d) in hash table #2, the pair (p,q) in hash table #3 and for the reference pair (**7**,**2**) in hash table #5. However, new items (and references) are created in hash tables #4, #6 and #7.
If the complete vector already exists in (the collection of) the hash tables, this is detected at the hash table in the root node: then, the pair consisting of the references returned by the left and right subtree already exists in the hash table. In this case, the operation returns true and no hash tables have been altered.

Maximal sharing

Laarman *et al.* [19] extend this approach by combining all hash tables in one big table. This ensures maximal sharing. For example, this makes it possible to share the subvectors (a,b) and (c,d) in the vectors (a,b,c,d) and (c,d,a,b). This also fixes the problem of finding optimal sizes for each individual hash table.
Note that, in practice, state vector elements are just integers (or more general, bit patterns), and so are the references, hence they are indistinguishable from each other. This leads to the following issue: Suppose a *sub*vector (0,1) has been inserted before. Now, suppose a non-existent complete vector V is inserted and the left subvector results in reference 0 and the right one in reference 1. As (0,1) already exists in the hash table, the algorithm falsely concludes that V already exists.

This is fixed by including one additional `is_also_root_node` bit to each bucket in the hash table. In the example, this bit would not be set initially, as the inserted subvector (0,1) is not the root node of a (implicit) tree. The algorithm then concludes that V does not exist in the hash table yet, as the non-set bit states that (0,1) is not the root node of a (implicit) tree. The algorithm now sets the bit, hence subsequent probes of vectors equal to V will return true, indicating that V already exists in the hash table.

Results show that in most (larger) models, an (almost) optimal compression ratio is achieved, *i.e.*, the average memory usage per state vector is about the size of a (root) node; in most cases, adding a new state vector only adds a root node, no other nodes. This also implies that, in general, the achieved compression ratio is better for models with larger vector lengths.
Especially for smaller models, the order of elements (slots/variables) in a state vector may impact the compression ratio that is achieved. Finding an optimal order is, however, an exponential problem. Laarman *et al.* could not find a good heuristic.
The notions above are related to the principles of tree-based compression, not to a CPU or GPU implementation; effects on performance may, however, differ.

Memory structure

To use this compression technique in a hash table that is shared by multiple threads, the hash table should be designed carefully. Due to the high contention, ordinary locking is not feasible. Instead, a variation on the lockless hash table design from Section 3.1.1 will be used: a separate data array and hash memoisation are not used anymore. Figure 3.5 shows the resulting memory structure:

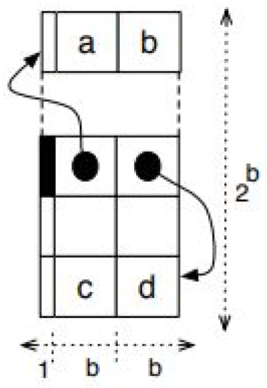


Figure 3.5: memory structure (from [19])

where `b` is the length of each state vector slot in bits; one bit is used for the `is_also_root_node` bit and a black rectangle indicates that this bit is set. In this example, a state vector has four slots. The figure shows the memory structure after the vector (a,b,c,d) has been inserted.

Compression of the open set

As the closed set is now compressed, the open set can now be the memory bottleneck. Using the same principles, it is also possible to compress the open set: the open set, which is usually implemented as a stack or queue, now contains references to the closed set. Using those references, it is possible to reconstruct the full state vector, which can then be used to find successors.

Incremental tree database

Finally, Laarman *et al.* describe an incremental tree structure: If a successor state differs in only one element, which is often the case, (the tree structure of) the originating state can be used in the `find-or-put` operation that is applied on the successor's state vector. For example, if the left subvector is the same in both state vectors, the reference to the corresponding subtree is already known from the tree structure of the originating state. Therefore, it is not necessary anymore to compute this reference bottom-up. The same principle applies for smaller common subvectors/subtrees, for example, if the left subvector of the right subvector is the same. This technique results in a serious performance improvement, as a much less number of memory locations need to be read.

Performance

Despite the multiple table accesses needed for each `find-or-put` invocation, results show that compression adds almost no overhead in CPU multi-core model checking. Compressed runs were sometimes even faster than uncompressed ones, especially with small models, due to better cache utilisation and a lower memory bandwidth. The compressed hash table also scales better with more cores than the uncompressed table, due to the lower memory footprint.

4. Overview of project

This chapter gives a high-level overview of the project. We start with giving the motivation, research goals and questions, research methodology and products that have been delivered. Then, we explain the test program we used (Section 4.1) and the test setup (Section 4.2), as they are used in all our experiments, which are explained in more detail in subsequent chapters.

Motivation

The compression algorithm of Section 3.2 has not been ported to GPUs yet. However, the benefits would be significant, as GPU memory sizes are relatively small. Laarman *et al.* show that the space reductions in multi-core model checking are enormous, at a negligible performance penalty [19]. Incorporating the state reduction technique Partial Order Reduction in GPUexplore [13] already reduced the space requirements, but the promise of the compression algorithm is to reduce it even further, enabling the verification of larger models and models with data.

The multi-core compression algorithm has been implemented recursively, *i.e.*, applying the same method on the left and right subvectors/-trees recursively. Implementing the compression technique on a GPU in a performant way will be a challenge, as recursion works differently on GPUs, *e.g.*, the call stack is saved to local memory, which resides in (slow) global memory.

Research goals and questions

Thus, our main research goal is:

- Reducing the space requirements for GPGPU explicit-state on-the-fly model checking by implementing an efficient GPGPU compression algorithm for hash tables, enabling the verification of larger models and models with data.

To achieve this goal, we will address these research questions:

- How can we implement an efficient GPGPU compression algorithm, in OpenCL and/or CUDA? Can we improve it even further?
    - What is the performance compared to an uncompressed hash table? What are the performance benefits of the improvements?
      To do a fair comparison, we need to address these questions first:
        - What is the impact of GPU parameters, such as the execution configuration, on the performance of both the uncompressed and compressed hash tables?
        - What is the impact of input parameters, such as the vector length, on the performance of both the uncompressed and compressed hash tables?
        - What is the impact of table parameters, such as the table size/fill rate, on the performance of both the uncompressed and compressed hash tables?
        - Using the results from the questions above, what are the optimal GPU and table parameters for each type of input, for both the uncompressed and compressed hash tables?

Research methodology

We started with a stand-alone version of the uncompressed hash table from GPUexplore 2.0 [13], but with a configurable bucket size. The stand-alone implementation is loosely based on the work of Cassee [18]. Chapter 5 provides more details about it. All those implementations are only available as CUDA source code. Ultimately, we aim at integration into GPUexplore. Therefore, we chose CUDA as the GPGPU programming framework for our project; results may also be valid for OpenCL implementations.
Cassee also implemented a hash table using Cuckoo hashing [17]. Cuckoo hashing handles collisions by relocating hash table entries. Stand-alone results were promising [17], but integrated into GPUexplore the Cuckoo hashing caused too many false negatives, as the relocation process is not atomic for vectors larger than 64 bits [5].
We implemented our stand-alone compression hash table on top of the uncompressed hash table and its `find-or-put` operation, just as Laarman *et al.* did [19]. This would make integration into GPUexplore more straight-forward. Using Cuckoo hashing is not possible, as this would break stable indexing, required for the references in the compressed table. We made several improvements to our original implementation, aimed at tackling the main performance limiters. Chapters 6 and 7 describe more details.

For performance evaluation, we used parameterised random data and state vector sequences extracted from real-world models (see Section 4.2 for more details).
For each implementation, we first determined the optimal execution configuration (block and grid dimensions). As this possibly depends on input and table parameters, we examined the impact of those parameters on the optimal execution configuration. All subsequent experiments use the optimal execution configuration(s) that are found.
Next, we evaluated the impact of (combinations of) input parameters (vector length, duplication, number of vectors and, in the case of the compressed versions, compression ratio) on (relative) performance. We did the same with table parameters (table size/fill rate and bucket size). Finally, we examined the impact of the combination of input and table parameters. We also tried to identify any dependencies between parameters.
Using the results from those experiments, we identified optimal table parameters and an optimal execution configuration for each type of input. This enabled us to do a fair comparison, for each type of input, between the compressed hash table implementations and the uncompressed implementation, and between the compressed hash table implementations themselves. In Chapter 8, this comparison is repeated for real-world data. It also features a recap of the practical random-data experiments of Chapters 5-7.
Table 4.1 (next page) gives an overview of all experiments.
Integrated into GPUexplore, performance effects may be different, but our extensive performance evaluation can be used in guiding optimisation efforts when integrated.

Products

Our project has resulted in these products:

- stand-alone CUDA implementations of the uncompressed hash table from GPUexplore, including a configurable bucket size and other improvements
- stand-alone CUDA implementations of a tree-based hash table compression algorithm, both recursive and non-recursive, including improvements
- experimental data: the results of our experiments with GPU, input and table parameters on the performance of the uncompressed and compressed hash tables
- this thesis: a report that contains the results of the research, *i.e.*, the answers to the research questions and a discussion of the results

experiment (table identifier)	**description**
parameterised random data (Chapters 5-7)	
optimal execution configuration (`-E`)	determining optimal block and grid dimensions for each input and table parameter (+ combinations); all other experiments use the found configurations
different input *[fixed default low fill rate 0.24 → adjusted table size + default bucket size 32]* (`-I`)	determining the impact of different input parameters (vector length, duplication, number of vectors, compressed: compression ratio) on performance, in isolation and combined (trying to find performance dependencies); comparing with other versions
different table parameters *[fixed default input]* (`-T`)	determining the impact of different table parameters (table size/fill rate and bucket size) on performance, in isolation and combined (trying to find performance dependencies); comparing with other versions
different input + table parameters (combined):	
different input [**high fill rate 0.80** + *default bucket size 32*] (`-I-hfr`)	determining the impact of different input on performance (as `-I`, trying to find performance dependencies), on a high fill rate of 0.80; comparing with low fill rate (`-I`) and other versions
different input [*low fill rate 0.24* + **bucket size 8/4/2**] (`-I-s8`/`-s4`/`-s2`)	determining the impact of different input on performance (as `-I`, trying to find performance dependencies), on lower bucket sizes of 8, 4 and 2 (if possible); comparing with bucket size 32 (`-I`), each other and other versions
different input [**high fill rate 0.80 + bucket size 8/4/2**] (`-I-hfr-s8`/`-s4`/`-s2`)	determining the impact of different input on performance (as `-I`, trying to find performance dependencies), on a high fill rate of 0.80 and on lower bucket sizes of 8, 4 and 2 (if possible); comparing with bucket size 32 (`-I-hfr`), low fill rate (`-I-s8`/`-s4`/`-s2`), each other and other versions
in practice, fixed table size of 256MiB:	
different input [**fixed table size of 256MiB** + *bucket size 32*] (`-I-fn`)	determining the impact of different input on performance (as `-I`), in a fixed-sized 256MiB table; comparing with fixed low fill rate 0.24 (`-I`) and other versions
different input [**fixed table size of 256MiB + optimal bucket size**] (`-I-fn-os`, `PRD` (Chapter 8))	determining the impact of different input on performance (as `-I`), in a fixed-sized 256MiB table and with optimal bucket size for each input (determined by experiments above); comparing with bucket size 32 (`-I-fn`) and other versions
real-world data (Chapter 8)	
different input [**fixed table size of 8GiB + optimal bucket size**] (`RWD`)	determining the impact of different real-world input on performance, in a fixed-sized 8GiB table and with optimal bucket size for each input (determined by experiments above); comparing with other versions

Table 4.1: overview of all experiments

4.1 Test program

All hash table implementations are wrapped in a C++ test program. This program enables users to specify an input vector file. The program reads this file and prints the vector length (should be constant) and total number of vectors. It then initialises the hash table, in GPU global memory, and copies all vectors, also to GPU global memory.
Next, the program runs a GPU kernel that inserts all vectors into the hash table. This is done by a grid-stride loop, *i.e.*, a grid-wide loop that iterates over all vectors (cyclic partitioning). In this way, the user can specify arbitrary execution configurations, as long as the block dimension is a multiple of the warp size, *i.e.*, 32.
When all vectors have been inserted, the test program runs a GPU kernel that counts all non-empty entries in the hash table. The program then prints this number. For the compressed versions, the program also runs a GPU kernel that counts all vectors in the table (*i.e.*, nodes with their root bit set) and also prints this number.

The program allows users to specify GPU and table parameters:

- block dimension (number of CUDA threads in each block)
- grid dimension (number of blocks)
- number of entries (table size)
- bucket size

4.2 Test setup

This section gives an overview of the test setup; for more details, see Appendix B.1.
The hash tables and the test program require CUDA Compute Capability (CC) 3.0, *i.e.*, Kepler and all newer NVIDIA GPU architectures are supported. We also specified CC 3.0 as target architecture for compilation.
We have used the `nvprof` command line profiler for our experiments. This tool not only gives runtimes for each kernel, but can also be used for collecting metrics, *e.g.*, the degree of branch divergence. All runtimes are averages of three runs. We measured a small (startup) overhead of only 0.03 ms, by inserting just one vector.
Appendix B lists all experimental data.

Ceteris paribus

In our performance evaluation, we first examine the impact of the change of a parameter in isolation, *i.e.*, keeping all other parameters unchanged. But, for example, when we change the amount of duplicated vectors, we also change the table size to keep the same fill rate. We also account for "lost" entries. For example, for vector lengths 3 we increased the uncompressed table size by 32/30 (bucket sizes 32 and 16) or 4/3 (bucket sizes 4 and 8), to keep the same (effective) fill rate.
The default (low) fill rate is 0.24 and the default bucket size is 32 (as in GPUexplore).
We also included experiments with a fixed table size, and, consequently, different fill rates for different types of input. This is more like practice, as one, in general, claims a fixed amount of memory for the hash table, *i.e.*, as much memory as possible. For the same reason, we use a fixed table size for the experiments with, for each type of input, optimal settings (in particular, optimal bucket size).
In both experiments with a fixed-sized table, we use the same table size for both compressed and uncompressed tables, to account for the space reduction achieved by compression; in the other experiments, we reduce the table size of the compressed tables to get the same fill rate when we compare to the uncompressed table.

Parameterised random data

We created a Java program that generated random sequences of vectors. Those sequences, however, were parameterised by:

- vector length, *i.e.*, the number of elements in each vector
- total number of vectors, including duplicates
- ratio of duplicate vectors
- amount of internal duplication, *i.e.*, replication of individual vector elements, which affects the amount of compression possible

Duplicate vectors are randomly distributed over the sequence.
In the compressed tables, vector element values can also be considered as references to other nodes in the table. To account for this, all vector element values are restricted in such a way that they can also be used as a reference to the other nodes of a 512MiB table. Note that this only applies to compressible inputs (vector length 3 or larger).
For more details, we refer to Appendix A.1; Appendix A.2 contains an overview of all input files[4] we used in our experiments, including all their parameters and meta-data.

Real-world data

Using a modified version of REFINER[5], we have extracted state vector sequences from models often used in (the benchmarking of) model checking. Each process in the model gets its own (32-bit integer) element in the vector. Using those sequences, which also include duplicates, we are able to mimic the (order of) `find-or-put` calls that actually would take place during model checking. We used models from the BEEM[6], CADP[7] and mCRL2[8] databases; models with a `.1` suffix have been modified to obtain a larger state space. For all model details, we refer to Appendix B.4.

GPU accelerator: Titan Xp

We use an NVIDIA Titan Xp for our experiments. This graphics card is based on the Pascal GPU architecture and has a GP102 chip. Its Compute Capability (CC) is 6.1. It has 30 Streaming Multiprocessors (SMs), each consisting of 128 CUDA cores, making a total of 30 SMs * 128 cores/SM = 3840 cores; its memory configuration is 12GiB GDDR5X. Theoretical limits are 12,150 GFLOPS (computation) and 547.6 GiB/s (memory).
The GPU accelerator features a per-SM unified L1 and texture cache. When the compiler detects any reads from read-only global data, they are loaded via this cache (and the L2 cache); the programmer can guide the compiler by using restricted (non-aliasing) pointers or do an explicit read-only cache read by using the `__ldg()` operation [12]. Other global reads (and writes) are loaded via the per-device L2 cache only, but additional L1 caching (for reads) can be enabled by a compiler option. Thread-local memory, *e.g.*, the call stack, resides in device memory, but is cached in L1 and L2 [28].
All global memory transactions are 32B, regardless of whether loading is via the L1 (and L2) caches or via the L2 cache only. This differs from previous NVIDIA GPU architectures, where transactions via the L1 cache were 128B [28]. L1 and L2 cache lines are still 128B, but now consist of four 32B segments; only the segments requested are loaded from global memory, any other segments are not pre-fetched [29].

[4] https://drive.google.com/file/d/1SPXJnfJzNxb3TjwSdhIHp58eRAFEq6oy/view
[5] https://www.win.tue.nl/~awijs/refiner/index.html
[6] http://paradise.fi.muni.cz/beem/
[7] https://cadp.inria.fr/demos.html
[8] https://www.mcrl2.org/web/user_manual/download.html

Execution configurations

Block dimension, shared memory size and the number of registers used by each thread restrict the number of warps that can be resident on a single Streaming Multiprocessor (SM), *i.e.*, the number of active warps. Each Pascal SM can host up to 64 warps, *i.e.*, 64 warps * 32 threads/warp = 2048 CUDA threads. Occupancy is defined as the ratio of the number of active warps to the number of maximal warps (64).
In general, one should strive for maximal occupancy, as this increases the amount of thread-level parallelism and enables more possibilities for latency hiding. The CUDA Occupancy Calculator, included in the CUDA SDK, can help to achieve maximal occupancy, as it shows the impact of varying block dimensions, shared memory sizes and register usage on occupancy.
The compiler tries to find the optimal register usage versus occupancy by sometimes spilling registers to thread-local memory, as this would increase occupancy. The CUDA compiler also has an option for specifying maximum register usage, forcing register spilling. Block dimensions are directly controlled by the execution configuration given at the launch of a GPU kernel. In our experiments, we only use block dimensions that achieve the highest level of occupancy possible (given some register usage and shared memory size).

We calculate our initial grid dimension as follows: Based on the block dimension, we calculate how many blocks we need to achieve maximum occupancy on a single SM. For example, when the block dimension is 256 threads (256 threads / 32 threads/warp = 8 warps), we need 2048 threads [maximum] / 256 threads/block (or 64 warps [maximum] / 8 warps/block) = 8 blocks. We then multiply this number by the SM count to get the grid dimension. In the example, 30 SMs * 8 blocks/SM = 240 blocks. We also try other grid dimensions, but they are always a multiple of the number of SMs; as we are primarily interested in high-level patterns, we do not account for micro-adjustments, which could, however, yield (a little bit) better performance.
Overloading an SM and/or device, however, may have benefits. For example, when it is difficult to get an even load distribution per block. The overloading mechanism can then be used for load distribution: Suppose 120 blocks each have two times more workload as 240 other blocks. Then, the former 120 blocks are loaded onto the device, along with 120 of the latter blocks; when the latter ones have completed execution, the 120 remaining ones are loaded and are finished at the same time as the first 120 blocks.
Overloading can also be used for automatic scalability. Suppose a new GPU has two times more SMs, *e.g.*, 60 instead of 30. When the overloading factor was two times (or even more), the new GPU could complete execution twice as fast. Of course, in reality this is more complicated, but the same principle holds.

When not using a grid-stride loop (*i.e.*, a grid-wide loop that iterates over all vectors), the execution configuration has to be adapted to the input size to ensure processing of the full input; *e.g.*, the total number of threads in the grid should be equal to the total number of vectors or elements. But, in general, execution configurations are only related to performance, not correctness.

5. Uncompressed GPU hash table

This chapter starts with a discussion of the stand-alone implementation of the uncompressed hash table from GPUexplore (Section 5.1). We found several flaws in it, including corruption of vectors, replication of vectors and a hash function with inferior distribution properties. We fixed them all. Results show that performance has improved.
Then, we present our extensive random-data performance evaluation using the fixed implementation (Section 5.2), *i.e.*, the impact of GPU, input and table parameters on performance. Using the results, we found optimal GPU and table settings for each type of input. The results obtained with these settings are in Chapters 6 and 7 used for a fair comparison between the compressed hash tables and the uncompressed hash table.
Finally, in Section 5.3, we give the most important conclusions of the extensive performance evaluation of Section 5.2.

5.1 Stand-alone implementation

As stated in Chapter 4, we started with a stand-alone implementation of the uncompressed hash table from GPUexplore 2.0 [13], loosely based on the work of Cassee [18] (Subsection 5.1.1). As this implementation may lead to corruption of vectors, we reverted to the implementation of GPUexplore 1.0 [5] (Subsection 5.1.2).
This implementation, however, has its own flaws: replication of vectors across the table and a hash function with an inferior distribution. This does not impact correctness, but during our experiments we found that the flaws led to a worse performance. Therefore, we designed and implemented versions without replication (Subsection 5.1.3) and with a probabilistic-optimal hash function (Subsection 5.1.4). Results show that we were right.

As the hash table is used in a model checker, which is often used to verify safety-critical systems, we examined correctness aspects in detail, next to performance aspects.

5.1.1 Original version (GPUexplore 2.0)

The general idea of the stand-alone version is the same as that of Algorithm 3.3 (page 23), but does not load vectors from the shared cache; it just operates on the vector that is given as an argument to the `find-or-put` call. It also allows for vectors consisting of multiple elements; it tries to claim a complete bucket slot at once. In contrast to the integrated version, the stand-alone version has a configurable bucket size.

We removed warp-synchronous programming by replacing warp-level primitives with synchronised (`_sync`) variants. This made the code CUDA 9.0+ compliant, which is important for Independent Thread Scheduling (ITS), introduced in NVIDIA's Volta GPU architecture. ITS may in particular show benefits on lower bucket sizes, as those lower sizes introduce bucket groups within each warp and those groups may follow different code paths; ITS may mitigate the effects of the introduced branch divergence.
CUDA 9.0 also introduced Cooperative Groups, a mechanism for the synchronisation of groups of threads. Warps are a type of Cooperative Groups, but Cooperative Groups allow other group sizes as well. For example, bucket and vector groups could be implemented as Cooperative Groups. We have not done so, as bucket and vector groups never span more than a warp in our implementation; instead, we implemented them using warp-level primitives. As Cooperative Groups are built on the same primitives, there would not be any difference in performance, but only at source code level.

During our experiments, we encountered replication of vectors, but only when they consisted of more than four elements. Apparently, atomic memory requests are not scheduled in half-warps anymore, but per four CUDA threads and this introduces a race condition. Therefore, when a duplicated vector is inserted by two concurrent `find-or-put` invocations, it is possible that both invocations store that vector in the hash table, in different locations, as an invocation may see the updated contents of only a part of the bucket slot. In other words, storing a full vector into a bucket slot is now not atomic anymore.
Even worse, we were able to construct an experiment with corrupted vectors, *i.e.*, the bucket slot contained a mix of elements from two different vectors. This would be disastrous for a model checker, as a model checker may falsely conclude it has already seen a vector and then decides to not explore the vector further (a false positive). Successor states of that vector may violate invariants, which is then not detected. The model checker may then falsely conclude that verification was successful. Therefore, it is crucial that we remove the possibility for corrupted vectors.

5.1.2 Version without corrupted vectors (GPUexplore 1.0)

To eliminate this possibility for corrupted vectors, we reverted to the insertion procedure of GPUexplore 1.0 [5]: The algorithm tries to claim the first entry of a bucket slot only. When it succeeds, it fills the other entries. Otherwise, it immediately proceeds to the next bucket slot, without waiting for the current slot to be filled (*i.e.*, to check for a duplicate vector).
Replication still occurs, but now also for vectors that consist of four or of a smaller number of elements: As the apparent scheduling of atomic memory operations per four threads is nowhere documented and could change any moment, we stick to 32-bit atomic operations only. This means now only bucket slots of exactly one 32-bit entry (one vector element) could be fully claimed by a vector group (of one thread); other vector lengths would require waiting for the completion of writing the full bucket slot by a concurrent invocation. For reasons of simplicity, our implementation only checks the return value of the `atomicCAS()` operation for the empty entry (*i.e.*, for a successful claim). Therefore, also input with vector length 1 may get replicated.
As we are not using the assumption anymore that atomic memory requests are scheduled in half-warps, slots can now cross half-warp and half-bucket boundaries again.

Although replication (false negatives) is still possible, corruption of vectors (false positives) is not possible anymore. False negatives hurt performance, but does not affect correctness, as false positives do.

Reading non-`volatile`

When reading hash table entries for comparing them to the elements of the vector that is being inserted (Algorithm 3.3 (page 23), line 11), this is done in a non-volatile way, *i.e.*, a stale (old) value may be read, *e.g.*, when it has been cached (remember that cache coherence protocols do not exist in GPUs yet).
This does not lead to false positives, as the only stale value that can be read is the empty entry value, which is a restricted value that vector elements are not allowed to have; when one or more entries in a bucket slot have the empty entry value, the algorithm always concludes that the bucket slot does not contain a vector, including the vector that is being inserted.

It, however, may lead to (more) false negatives, as the algorithm may falsely conclude that a bucket slot does not contain the vector that is being inserted. The subsequent `atomicCAS()` operation (line 19) operates on the actual value, but as its return value is only checked for a successful claim (by comparing it to the empty entry value), the algorithm just assumes that the bucket slot was taken by another vector and proceeds with inserting the vector into another bucket slot.
We did experiments with a version that reads hash table entries in a volatile way (line 11). We could, however, not measure any differences in the amount of replication regardless of whether L1 loading was enabled for all global reads or not. Runtimes were worse, especially with CC 6.1 as target architecture. Therefore, we decided to keep reading in a non-volatile way.

Data races

In the strict sense, the non-atomic hash table reads are racy, as there can be concurrent writes to the same hash table entries, both atomic (first entry of a bucket slot) and non-atomic (the other entries of a bucket slot). We did experiments with atomic loads, but as those are not natively supported by CUDA, we mimicked them using atomic increments by zero, which return the original (and, in this case, also new) values.
We measured a slowdown in performance, especially with bucket sizes 8 and 16. Because of this and because atomic loads (and stores) are apparently not needed in CUDA (and thus not supported), we decided to keep non-atomic reads (and non-atomic writes for the non-first entries of a bucket slot).

Low-level optimisations

We tried several low-level optimisations, for processing multiple types of inputs and both on the default bucket size of 32 entries and on bucket size 8. We then still used an old hash function from Cassee, whose higher register pressure lowered occupancy.

Most optimisations have no or almost no effects:
- enabling L1 loading for all global reads: no effect with CC 3.0 as target architecture (`sm_30`), small positive effect with CC 6.1 as target architecture (`sm_61`)
- using restricted pointers (`__restrict__`), where appropriate, to indicate that the object the (array) pointer references to is not aliased by another pointer, possibly enabling some compiler optimisations, *e.g.*, enabling reading from the read-only cache: almost no effect, regardless of L1 loading for all global reads or not
- forced inlining (`__forceinline__`), especially of the `find-or-put` function: no effect, regardless of L1 loading or not
- removing `const` qualifiers (from the input array, *i.e.*, trying to force loading via L2 cache only): no effect, independent of L1 loading or target architecture
- forcing register spilling (`maxrregcount`), which impacts the optimal execution configuration, has a large beneficial effect with `sm_61` and then performance is getting close to the original performance of `sm_30`; slight effect with `sm_30`

Other "optimisations" have effects, but mostly negative:
- CC 6.1 as target architecture (`sm_61`): negative, as this leads to a higher register pressure (and, consequently, a different optimal execution configuration)
- a mixture of CC 3.0 and CC 6.1 as target architectures (`compute_30, sm_61`): little bit slower (same register pressure as `sm_30`)

In the end, we have not used any low-level optimisation we tried.

5.1.3 Removal of replication

Although replication does not impact correctness, its (performance) effects can be large: Replication leads to a decrease in the effective table size. It may hurt performance in several ways: Inserting vectors may take more time as (atomically) writing a replicated vector takes more time than reading the identical vector that has already been inserted. But even if the vector does not exist in the table yet, insertion may take longer as it takes longer to find an empty slot in a highly filled table. False negatives may also lead to more redundant work that needs to be done by the model checker that uses the table. Indeed, during our experiments with real-world data we experienced a replication up to 41%, especially on lower bucket sizes. We could only measure a very small replication in our experiments with random data, as the distribution of duplicated vectors is then different: randomly (random data) versus very local (real-world data); very local duplication offers more possibilities for replication, as more duplicate vectors are being inserted concurrently, especially on lower bucket sizes. The results are different from GPUexplore (only 2% replication with real-world models), as the hash table access patterns are different and local duplicate detection takes place, via the shared cache.
As replication natively does not appear in the compressed hash tables of Chapters 6 and 7, we need to design and implement a replication-free uncompressed hash table to allow a fair comparison.

Replication-free implementation

To get a replication-free implementation, we re-introduced the spinlock of the multi-core implementation of Laarman *et al.* (Subsection 3.1.1). As our GPU implementation does not use distinct bucket and data arrays, but only a data array, we use the first entry in a bucket slot for locking purposes. Our method to get rid of replication is very similar to the solution of Laarman *et al*.
A bucket/vector group now tries to claim an (apparently) empty bucket slot by writing a temporary ‘writing-in-progress’ element into the first entry of that slot. The most significant bit (MSB) of this element is set, to indicate that writing is in progress (for this to work, we require that the MSB of all input vector elements is not set). The other bits contain the memoised hash.
If a bucket/vector group successfully claimed an empty slot, it subsequently writes all vector elements, except the first element. Only if those have been written, the vector group leader writes its own element (*i.e.*, the first element of the vector) into the first entry of the slot, indicating that writing has been completed. A `__syncwarp()` warp-level synchronisation primitive ensures that this first element is only being written after the other threads in the vector group have written their elements. Writing is done in a volatile way to make sure that the writes are visible to other threads (as long as they also read in a non-volatile way).
If a bucket/vector group did not succeed in claiming an (apparently) empty slot, it will read the value returned by the `atomicCAS()` operation, *i.e.*, the actual value of the first entry of the slot. If this is the temporary ‘writing-in-progress’ element, it will check the memoised hash. In many cases, it can already conclude that the vector that is being inserted by a concurrent invocation is different, as the hashes do not match. But if the memoised hash is the same as its own hash, the vector that is being inserted is possibly identical and the vector group spinlocks on the first entry of the slot, till the ‘writing-in-progress’ element has been replaced with the first element of the vector that is being inserted. It can now check the full slot for an identical vector; this is done immediately if the `atomicCAS()` operation did not return a ‘writing-in-progress’ element.

For this to work, reading the slot entries from the hash table should be done in a non-volatile way; otherwise, a stale (old) value may be read and the algorithm may falsely conclude that the vector has not been inserted yet. The initial reading (before claiming an (apparently) empty slot) can still be done volatilely, as long as every bucket slot that does not contain an entire vector (*i.e.*, one or more slot entries are empty or the first entry in the bucket slot is a (hash-matching) 'writing-in-progress' element) is considered an empty slot and is subsequently read in a non-volatile way.

Risk of deadlock

When two or more bucket groups in the same warp try to claim the same bucket (bucket size is lower than 32), there is a risk of deadlock: Only one bucket group succeeds and the other bucket group(s) then spinlocks on (the first entry of) the bucket slot. If the threads of the warp operate in lock-step, the constant spinlocking of the non-succeeding bucket group(s) may obstruct the succeeding bucket group from reaching its "unlocking" operation (*i.e.*, replacing the 'writing-in-progress' element in the first entry of the bucket slot with the first element of the vector).
During the experiments with our replication-free implementation, we had no deadlocks: Apparently, the execution of the succeeding bucket group is completed first, including the unlocking operation (the threads of the other bucket group(s) are then temporarily disabled). Only then the execution of the non-succeeding bucket group(s) is resumed; as the first entry in the bucket slot does not contain a 'writing-in-progress' element anymore, spinlocking will not take place and the entire bucket slot can be read immediately (in a non-volatile way).
This execution order, however, may differ with GPU architecture or compiler version, as it is nowhere specified. But it then will only lead to deadlock, not to a false 'verification successful' claim. Moreover, the most recent NVIDIA GPU architectures feature Independent Thread Scheduling: this fixes the issue as threads in a warp do not need to operate in lock-step anymore. Therefore, we decided to keep our current replication-free implementation and did not try to design a solution that is not dependent on the execution order.

Experimental results

With our random-data input, the performance benefits of the replication-free implementation are very limited, up to 3%. But the replication was already very small initially, due to the random distribution of duplicated vectors in the random-data input. In contrast to the situation in GPUexplore [5], the spinlocking does not hurt either. We cannot explain the difference, as the tried spinlocking implementation is not available.
With our real-world data, the performance benefits are more profound, up to 18%. The benefits are more profound as the initial replication was more severe, up to 41%, especially on lower bucket sizes, due to the very local duplication in the real-world data.
As there are no false negatives anymore, the program that uses the hash table does not need to do redundant work anymore. This may lead to even more performance benefits.

All experimental data can be found in the `[U-H/R]` table in Appendix B.2.

In conclusion, we designed and implemented a replication-free uncompressed hash table, with an equal or even better performance compared to GPUexplore's implementation with replication. The full table size can now effectively be used and the program that uses the hash table will not get false negatives returned anymore.

5.1.4 Integration of strongly universal hash function

The replication-free implementation of Subsection 5.1.3 fixed one important flaw in the hash table implementation of GPUexplore 1.0. But during our experiments with real-world data we found another flaw: Even with a large table size of 8GiB and a (relatively) small model (and, consequently, a low table fill rate), the amount of rehashing was very large, much more than expected. Sometimes, even the maximum bound in rehashing was reached. As this might hurt performance, we carefully looked at (the distribution of) the used hash function.

GPUexplore's hash function

The hash function of GPUexplore works as follows:
It is parameterised by two random 32-bit hash constants, to allow for rehashing. A 64-bit `hashtmp` integer variable is used to store the (partial) hash. The hash function adds every 32-bit state vector element to `hashtmp`; between each addition, `hashtmp` is shifted 5 bits to the left. The resulting `hashtmp` value is the same for each parameterised hash function.
The `hashtmp` value is then multiplied by one of the hash constants and the other hash constant is added. The resulting `hashtmp` value is now (almost always) different for each parameterised hash function. The final hash is calculated by `hashtmp` % `P`, where `P` is a large prime constant. The hash table index is calculated by restricting the final hash to the number of hash table buckets, again by a modulo operation.
One of the issues with this hash function arises with large vector lengths: At some point, the bits related to the first vector elements are shifted out. Essentially, the hash value now only depends on the last vector elements; in the extreme case, in which the last vector elements are always the same, each vector will get the same hash, even if the first elements are different. We fixed this issue by shifting in (on the right) the bits that are shifted out (on the left), *i.e.*, we implemented circular shifting.
But we also experienced extreme rehashing with smaller vector lengths. Therefore, we decided to integrate a mathematically grounded hash function: the strongly universal hash function of Section 2.3.

Parallel pair-multiply-shift

We not only integrated the fast 'Pair-multiply-shift' hash function of Section 2.3, but also a parallel version: The first threads in a bucket group (*i.e.*, a thread group that tries to insert a vector into the same bucket slot, consisting of one or more vector groups) calculate each pair. Then, parallel reduction is used to sum all pairs. Finally, the bucket group leader adds the `b` hash constant and shifts; all other threads in the bucket group receive this final hash by using a warp shuffle function. Note that the hash function of GPUexplore cannot be parallelised, due to the shifting between each addition.

Experimental results

Our results show that, with our random-data input, the amount of rehashing was low using GPUexplore's hash function. But even then switching to our strongly universal hash function improved performance. Apparently, our hash function itself is faster, probably because no slow modulo operations takes place, in contrast to GPUexplore's hash function. Our parallel version improved performance even more.

This was especially the case on bucket size 32; lower bucket sizes show less benefits, probably because the performance bottleneck is now different and the speed of the hash function has less impact. With vector length 1, the parallel version was slower, due to the overhead of parallel processing and the absence of benefits from parallel processing. But, again, this was more profound on bucket size 32 than on lower bucket sizes.

The results with real-world data are different: In contrast to the experiments with random data, in which we use a low table size, we now use a large table of 6GiB. We could now measure massive reductions in rehashing by switching to our strongly universal hash function, up to 75x. Apparently, the distribution of GPUexplore's hash function is especially worse in large tables.
The massive reduction in rehashing has also effects on performance: Runtimes were up to 32% lower. As we only experimented with inputs of vector lengths 3 and 8, our parallel version of the 'Pair-multiply-shift' hash function was always as fast or faster than the sequential version.
Optimal execution configurations and bucket sizes were different, compared to using GPUexplore's hash function; to mitigate the effects of extensive rehashing, "sub-optimal" execution configurations and bucket sizes need to be selected to get the best performance when using GPUexplore's hash function.

All experimental data can, again, be found in the `[U-H/R]` table in Appendix B.2.

In conclusion, we integrated the strongly universal (*i.e.*, probabilistic-optimal) 'Pair-multiply-shift' hash function into the uncompressed hash table implementation. This not only led to a reduction in rehashes, sometimes even massive, but also to equal or better performance, compared to the uncompressed hash table that uses GPUexplore's hash function instead. By implementing a parallel version of 'Pair-multiply-shift', we made use of the massive parallelism provided by GPUs.

5.2 Performance evaluation (random data)

As we now have fixed all flaws in the uncompressed hash table, we can start with our extensive performance evaluation, using the stand-alone implementation from Subsection 5.1.4. We use the parallel version of 'Pair-multiply-shift', which is in most cases faster than the sequential version; as we are primarily interested in compressible inputs (see Chapters 6 and 7), *i.e.*, inputs with vector length 3 or higher, we do not mind if the parallel version is a little bit slower for vector length 1 (and possibly 2 as well).

We start our performance evaluation by determining optimal execution configurations for each input and table parameter, including combinations (Subsection 5.2.1); they are used in all subsequent experiments.
We then examine the impact of different input (Subsection 5.2.2) and table (Subsection 5.2.3) parameters on performance. We also try to find performance dependencies between parameters. Next, we examine the impact of the *combination* of different input and table parameters on performance (Subsection 5.2.4), including comparison to the experiments of Subsection 5.2.2. The results are used to find optimal table parameters, in particular the optimal bucket size, for each type of input.
Whereas the experiments listed above are synthetic, our last experiments resemble a more practical situation (Subsection 5.2.5), with a fixed-sized table of 256MiB. We first experiment using the default bucket size of 32. Next, we use the optimal bucket size(s) found before and compare the results to those obtained using bucket size 32.

Table 4.1 (page 31) gives an overview of all experiments. For more details, see 'Research methodology' (page 30) and Section 4.2: Test setup (page 32).

In this performance evaluation we use parameterised random data. In Chapter 8, we do performance evaluation with real-world data. That chapter also recaps the results of our final experiment, using the found optimal bucket size(s), from Subsection 5.2.5.

The first four subsections (Subsections 5.2.1-5.2.4) give an in-depth understanding of the factors that impact the performance of the hash table, but are not necessary for understanding the general ideas; the last subsection (Subsection 5.2.5) and Section 5.3: Conclusions present the general ideas.

The table identifiers in the headings refer to the tables in Appendix B.3.1.
In the tables, a blue colour indicates a relevant value; green and red colours are used to indicate the lowest and highest runtimes, respectively. A bold font (in the 'speedup' and 'slowdown' columns) indicates a dependency on performance for two or more input or table parameters.

5.2.1 Optimal execution configuration – `[U-E]`

Our stand-alone implementation uses 32 registers per thread (no shared memory usage). Consequently, we can achieve maximal occupancy, *i.e.*, 64 warps per SM (= 2048 threads/SM). This maximises the possibilities for latency hiding. Our default block and grid dimensions are 256 threads and 240 blocks, respectively. We also tried block dimensions 128 and 512, and various grid dimensions (under- and overloading).
Our default block and grid dimensions were also the optimal execution configuration, but there were other execution configurations with (almost) the same runtimes: for example, grid dimension 480 (two times overloading), or block dimension 128 and grid dimension 480 (same total number of threads as our default execution configuration) or 960 (two times overloading). The overloading configurations can be used for automatic scalability. Underloading clearly hurt performance.
Our default execution configuration was also optimal for different input and table parameters. With bucket size 8 and especially bucket size 4, there was much less difference to the underloading configurations: as there are more concurrent (uncoalesced) memory transactions per warp, compared to higher bucket sizes, the memory bus gets saturated and there are fewer benefits from having more warps active.

We used the default execution configuration (block dimension of 256 threads and grid dimension of 240 blocks), which turned out to be optimal, for the following experiments.

5.2.2 Different input – `[U-I]` (fixed low fill rate 0.24 + fixed default bucket size 32)
see Table 5.1 (next page) for a summary of table *`[U-I]`*

We first examine the impact of input parameters (vector length, duplication and number of vectors) on performance in isolation. Next, we look at the impact of a combination of input parameters on performance, to find any performance dependencies.

Vector length: When varying the vector length, but keeping the total number of elements the same (and thus changing the total number of vectors), the effects are almost linear: for example, vector length 1 is almost four times as slow as vector length 4 and vector length 8 is 1.84x as fast.

The total number of vectors impacts the runtime so much, as it has an almost linear relation to the number of (memory) instructions per warp; the vector length is less important, as it is the strength of warps, or parallel GPU processing in general, to process multiple (vector) elements at once. The number of vectors also defines the number of (slow) atomic operations (compare-and-swaps): in the current experiment, going from vector length 4 to 1 quadruples the number of atomic operations.

input	**runtime (ms)**	**speedup vs. default of vector length**	**speedup vs. vector length 4**
default (vector length: 4)			
default (8,000,000 vectors, duplication 2.00)	9.31	*1*	*1*
less duplication: 1.12	10.41	0.89	*1*
0.5x number of (unique) vectors	4.60	2.02	*1*
2x number of (unique) vectors	18.94	0.49	*1*
lower vector length: 1			
default (same total number of elements → 32,000,000 vectors)	32.50	*1*	0.29
0.25x number of (unique) vectors: 8,000,000 vectors	7.72	4.21	[1.21]
lower vector length: 2			
default (same total number of elements → 16,000,000 vectors)	17.46	*1*	0.53
0.5x number of (unique) vectors: 8,000,000 vectors	8.62	2.02	[1.08]
lower vector length: 3 ***[with 32/30 compensation for "lost" entries]***			
default (same total number of elements → 10,666,666 vectors)	13.37	*1*	0.70
lower number of (unique) vectors: 8,000,000 vectors	9.98	1.34	[0.93]
higher vector length: 8			
default (same total number of elements → 4,000,000 vectors)	5.05	*1*	1.84
2x number of (unique) vectors: 8,000,000 vectors	10.17	0.50	[0.92]

[z.zz] is speedup vs. vector length 4, 8,000,000 vectors (+ same duplication)

Table 5.1: effects different input (uncompressed) **–** `[U-I]`

These effects can also be observed when varying the vector length, but now keeping the total number of vectors the same (and thus changing the total number of elements): the effects are minimal. For example, vector length 1 is only 1.21x as fast as vector length 4 and vector length 8 is 0.92x as fast. This indicates that the algorithm has a large part whose runtime is independent of the vector length. As a result, the number of vectors is far more important for performance than the vector length or total number of elements.

Duplication: The amount of duplication has a small effect on performance: changing the duplication factor from 2.00 to 1.12 slows down performance by 11%. Duplication has an impact on the read/write ratio: with less duplication, more (atomic) writes are needed, which are slower than (non-atomic) reads.

Number of vectors: Keeping the vector length the same, but changing the total number of vectors (and thus changing the total number of elements linearly), has a linear effect, as expected: doubling the number of vectors also doubles the runtime, as each warp now executes (almost) twice as many (memory) instructions.

Compensation for "lost" entries (see Section 4.2): We could not measure any effect of the compensation for the "lost" entries of vector length 3. The compensation, however, was very limited: 32/30.

Combinations/dependencies: We could not find any clear dependency, apart from a caching effect with 0.25x number of vectors (vector length 1): as the table size is also 0.25x as small (to keep the same fill rate), a (too) large part of the table can be kept in cache, bypassing (slow) device memory; in reality, much larger tables and input data are used and caching effects will appear much less.

5.2.3 Different table parameters – `[U-T]` (fixed default input)

see Table 5.2 (next page) for a summary of table *`[U-T]`*

We first examine the impact of table parameters (table size/fill rate and bucket size) on performance in isolation. Next, we look at the impact of the combination of the table parameters on performance, to find any performance dependencies.

Table size/fill rate: When lowering the table size to get a high fill rate (0.80 instead of 0.24), there is a small effect of 5% slowdown, probably due to more rehashing (when the initial bucket is full): computing a new hash costs time and so requesting a new bucket (memory location). We do not see effects with fill rates 0.12 and 0.48, as they are still low.

Bucket size: Lowering the bucket size clearly improves performance, by 30% (bucket size 16) and 45% (bucket size 8, compared to bucket size 32). But the relation is far from linear or logarithmic: the advantageous effects of the parallel processing of multiple vectors by a single warp and a better distribution (more buckets) are tempered by saturation of the memory bus, branch divergence (different execution paths within warps) and more rehashing (as there are fewer slots in each bucket); all effects are stronger on bucket size 8 than on bucket size 16, but, apparently, this is even more the case for the negative effects than for the positives ones.

Bucket size 4 is even a little bit slower than bucket size 8 (but still faster than bucket sizes 16 and 32): Remember that the memory transactions of the Pascal GPU architecture of our Titan Xp accelerator are 32B (see Section 4.2), so when each thread in a warp requests a 32-bit integer and those integers form an aligned and contiguous chunk of memory of 128B (which is the case for bucket size 32), this boils down to four 32B memory transactions (4 transactions * 32 B/transaction = 128B); on bucket sizes 16 and 8, two and four contiguous chunks of memory of 64B and 32B, respectively, are requested, but this still boils down to four 32B memory transaction.
On bucket size 4, however, eight 32B memory transactions are needed, as eight contiguous chunks of 16B are requested; 16B (50%) of each memory transaction is wasted, giving a global load efficiency of only 50%, compared to 100% (other bucket sizes). This really hurts our memory-bound application.

Combination/dependencies: With a high fill rate, the (positive) effect of lowering the bucket size is lower, due to even more branch divergence and more rehashing: on a high fill rate of 0.80, bucket size 16 is now only 22% faster and bucket size 8 24% (compared to bucket size 32); bucket slot 4 is now even slower than bucket size 16.
The same effect, although less profound, is seen with a medium fill rate of 0.48; on the other hand, on a low fill rate of 0.12, bucket size 4 is even a little bit faster than bucket size 8 (1.48 vs. 1.46, compared to bucket size 32).

table parameters	**runtime (ms)**	**speedup vs. bucket size 32**	**speedup vs. low fill rate 0.24**
default: low fill rate (table size: 256MiB → fill rate 0.24)			
default (bucket size: 32)	9.32	*1*	*1*
bucket size: 16	7.18	1.30	*1*
bucket size: 8	6.44	1.45	*1*
bucket size: 4	6.52	1.43	*1*
very low fill rate: 2x table size: 512MiB → 0.5x fill rate: 0.12			
default (bucket size: 32)	9.31	*1*	1.00
bucket size: 8	6.36	1.46	1.01
bucket size: 4	6.30	1.48	**1.04**
medium fill rate: 0.5x table size: 128MiB → 2x fill rate: 0.48			
default (bucket size: 32)	9.35	*1*	1.00
bucket size: 4	7.14	1.31	**0.91**
high fill rate: 0.3x table size: 76.8MiB → 3.33x fill rate: 0.80			
default (bucket size: 32)	9.86	*1*	0.95
bucket size: 16	8.11	1.22	**0.88**
bucket size: 8	7.97	1.24	**0.81**
bucket size: 4	9.03	1.09	**0.72**

Table 5.2: effects different table parameters (uncompressed) – [U-T]

Optimal table parameters: For each fill rate, except the very low fill rate of 0.12, a bucket size of 8 is optimal; bucket size 32 gives the highest runtimes. Additionally, a low fill rate yields the best performance, especially on lower bucket sizes. For our default input, a high fill rate and a bucket size of 32 are the most worse parameters and are 9.86 ms / 6.30 ms = 1.6x as slow as the optimal ones.

Linear probing – `[U-T (_lp8)]` (fixed default input)
see Table 5.3 (this page) for a summary of table `[U-T (_lp8)]`

To compensate for the reduced global load efficiency of 50% on the bucket size of 4, we have implemented a version with linear probing up to 8 hash table entries: any rehashing will not immediately take place after a (full) bucket has been examined, but instead the next bucket, consisting of another four entries, will be examined first; 'next' is to be interpreted as the next bucket in a 32B cache line segment, in a circular way.
This saves rehashing costs (calculating a new hash plus requesting a new bucket from global memory) and benefits from an already loaded cache line segment of exactly eight (32-bit) entries. The benefits will be even larger on bucket size 2, as its global load efficiency is only 25%; then, the next *three* buckets will be examined before rehashing.

Linear probing introduces some code overhead and causes a slowdown of 0-3% for bucket sizes 8 to 32, which do not benefit from linear probing as it is up to 8 entries only; the slowdown is larger for higher bucket sizes, as the code overhead is at `find-or-put` operation level and there are more `find-or-put` invocations on higher bucket sizes. On bucket size 4, however, linear probing gives a speedup of 3-17%; with higher fill rates, the speedup is larger, as more rehashing would be needed otherwise.

table parameters	**runtime (ms)**	**speedup vs. no `_lp8` [Table 5.2]**
default: low fill rate (table size: 256MiB → fill rate 0.24)		
default (bucket size: 32)	9.52	0.98
bucket size: 16	7.21	**1.00**
bucket size: 8	6.45	**1.00**
bucket size: 4	6.20	**1.05**
very low fill rate: 2x table size: 512MiB → 0.5x fill rate: 0.12		
bucket size: 4	6.11	**1.03**
medium fill rate: 0.5x table size: 128MiB → 2x fill rate: 0.48		
bucket size: 4	6.48	**1.10**
high fill rate: 0.3x table size: 76.8MiB → 3.33x fill rate: 0.80		
bucket size: 4	7.69	**1.17**

Table 5.3: effects different table parameters (uncompressed lp8) – `[U-T (_lp8)]`

We have also implemented linear probing up to 32 entries: after examining a full 32B cache line segment, we then proceed with the other segments in the 128B cache line, again in a circular way. This saves even more rehashing. We, however, do not benefit from additionally loaded cache line segments, as those are not pre-fetched (unless those segments are explicitly requested before). But it prevents many cache lines with “holes”, *i.e.*, cache line segments that are not filled and are wasted cache memory.
Our experiments, however, show no additional performance benefits over linear probing up to 8 entries only. Apparently, the performance benefits of linear probing are very related to benefiting from an already loaded cache line segment.

Optimal table parameters: Now, a bucket size of 4 is optimal, on all fill rates. This remains when we also consider the runtimes of the non-`_lp8` original (Table 5.2). But note that the bucket size can never be smaller than the vector length.
Still, a low fill rate gives the best performance.

5.2.4 Different input + table parameters

In the previous two subsections, we examined the performance effects of input (Subsection 5.2.2) and table (Subsection 5.2.3) parameters separately. In this subsection, we examine the effects of the *combination* of all input and table parameters on performance.
This subsection presents the results of multiple experiments: We start with the performance effects of different input on a high fill rate of 0.80. Then, we present the performance effects of different input on lower bucket sizes (8 and 4). Finally, we examine the performance effects of different input on a high fill rate of 0.80 plus lower bucket sizes (again 8 and 4). We compare the results to each other and to the results on a low fill rate of 0.24 plus default bucket size of 32 (Table 5.1).
The results are used to find optimal table parameters, in particular optimal bucket size, for each type of input.

Different input [high fill rate] – `[U-I-hfr]` (default bucket size 32)
see Table 5.4 (this page) for a copy of table `[U-I-hfr]`

input	runtime (ms)	speedup vs. low fill rate 0.24 [Table 5.1]
default (vector length: 4, duplication 2.00)	4.88 – 19.98	0.94 – 0.95
less duplication: 1.12	5.63 – 22.57	**0.91 – 0.92**
lower vector length: 1	7.59 – 36.58	**1.00 – 1.02**
lower vector length: 2	8.63 – 20.06	**0.99 – 1.00**
lower vector length: 3 [with 32/30 compensation for “lost” entries]	10.45 – 16.05	**0.94 – 0.96**
lower vector length: 3 [no compensation for “lost” entries]	10.64 – 16.54	0.92 – 0.94
higher vector length: 8	5.70 – 13.08	**0.86 – 0.90**

Table 5.4: effects different input [high fill rate] (uncompressed) – `[U-I-hfr]`

We measured a slowdown of 0-14% with different input types, moving from a low fill rate (0.24) to a high fill rate (0.80) table, dependent on vector length: almost no effect on vector length 1; more effects on higher vector lengths, due to a higher increase in rehashing (higher vector lengths have less slots in a bucket). The effects are also more profound in cases with less duplication: in a high fill rate table, claiming a bucket slot (by `atomicCAS()`) will fail more often and the resulting negative performance effects are stronger for cases that need to claim more slots, as they have more vectors to insert.
We can now see the effect of compensation for lost entries, on vector length 3: slowdown is 4-6% compensated and 6-8% non-compensated.

Different input [bucket size 8/4] – `[U-I-s8/-s4/-s4 (_lp8)]` (low fill rate 0.24)
see Table 5.5 (this page) for a copy of table `[U-I-s8]`

Bucket size 32 → 8 (Table 5.5): Moving from bucket size 32 to 8, we measured a speedup of 13-54%.
The speedup is less with lower vector lengths and/or less duplication: A lower bucket size increases the amount of memory operations that are concurrently in progress, including the amount of `atomicCAS()` operations. This increases the probability that an `atomicCAS()` operation has to wait for the completion (unlocking) of a concurrent atomic operation on the same hash table entry (and, in practice, locking granularity may even be more coarse, *e.g.*, at 32B or 128B level). As cases with lower vector lengths and/or less duplication have a higher level of `atomicCAS()` invocations, they are more hurt by this phenomenon.
Note the anomaly of vector length 3: speedup is *better* than on vector length 4. We do not have an explanation for this behaviour, but it is probably related to "lost" entries.

Bucket size 8 → 4: Moving from bucket size 8 to 4, the program was 3% slower – 4% faster, with an average of 1% slower (except vector length 8, as bucket size 4 is then not possible). Enabling linear probing, on bucket size 4, has a speedup effect of 0-7%, with an average of 3% faster (no effect on vector length 1).
Therefore, we only need to take a look at bucket size 4 with linear probing: comparing bucket size 4, with linear probing, to bucket size 8, without linear probing, gives a small speedup, up to 5% (except a few cases of vector length 1 that were up to 2% slower).

input	runtime (ms)	speedup vs. bucket size 32 [Table 5.1]
default (vector length: 4, duplication 2.00)	3.18 – 13.00	1.44 – 1.46
less duplication: 1.12	4.19 – 16.91	**1.22 – 1.24**
lower vector length: 1	5.79 – 32.29	**1.13 – 1.33**
lower vector length: 2	6.14 – 16.46	**1.20 – 1.40**
lower vector length: 3 [with **4/3** compensation for "lost" entries]	8.48 – 11.34	**1.34 – 1.54**
lower vector length: 3 [no compensation for "lost" entries]	8.57 – 11.45	1.32 – 1.51
higher vector length: 8	3.48 – 8.91	1.25 – 1.46

Table 5.5: effects different input [bucket size 8] (uncompressed) – `[U-I-s8]`

Different input [high fill rate + bucket size 8/4] – `[U-I-hfr-s8/-s4/-s4 (_lp8)]`
see Table 5.6 (this page) for a summary of table `[U-I-hfr-s8]`

Low fill rate → high fill rate (bucket size 8, Table 5.6): Changing low fill rate 0.24 to high fill rate 0.80, on bucket size 8, we measured a speedup up to 5%, for the inputs of vector length 1, and a slowdown of 5-32%, for the inputs of other length; the effects of a high fill rate are more profound on bucket size 8 than on bucket size 32 (Table 5.4, page 47), due to more rehashing. Again, the effects are stronger for inputs of higher vector lengths, as those inputs result in buckets with less (but larger) slots.
The inputs of vector length 1 are now faster, especially those with a less number of vectors, probably due to (L2) caching effects, as the high fill rate is achieved by a small table, in global memory. As the input, which resides in global memory as well, is also loaded via the L2 cache (and via L1), this may also impact the performance related to input loading. Inputs of other vector length probably also benefit from caching effects, but the negative effects, apparently, outweigh the caching effects. In reality, we have larger tables and input (model) sizes and caching effects will be much less profound.
In contrast to the situation on bucket size 32 (Table 5.4, page 47), the effects are now *less* profound in cases with less duplication: Apparently, those cases are already so much hurt by more `atomicCAS()` operations on bucket size 8, that changing to a high fill rate have less negative effects than in cases with more duplication.

Bucket size 32 → 8 (Table 5.6): For all inputs, bucket size 8 is still faster than bucket size 32 (Table 5.4, page 47), in a high fill rate table: 12-37% faster, for the inputs of vector length 1, and 4-34% faster, for the inputs of other length. Apart from vector length 1, the speedup is less than on a low fill rate table (Table 5.5, previous page).
Bucket size 8 → 4: Moving from bucket size 8 to 4, the program was 1-4% slower, for inputs of vector length 1, and 4-12% slower for inputs of other lengths; the effects are more profound than in a low fill rate table.
Enabling linear probing, up to 8 entries, has a speedup effect of 1-19%, with an average speedup of 11%. Again, effects are stronger than in a low fill rate table.
Comparing bucket size 4, with linear probing, to bucket size 8, without linear probing, gives a speedup of 0-8%, with an average of 3% (except a few cases of vector length 1 that were up to 3% slower). Now, effects are similar to the effects in a low fill rate table.

input	**speedup vs. low fill rate [Table 5.5]**	**speedup vs. bucket size 32 [Table 5.4]**
default (vector length: 4, duplication 2.00)	0.81	1.23 – 1.25
less duplication: 1.12	**0.84**	**1.12**
lower vector length: 1	**0.99 - 1.05**	1.12 – 1.37
lower vector length: 2	**0.92 - 0.95**	**1.14 – 1.34**
lower vector length: 3 [with **4/3** compensation for "lost" entries]	**0.78 - 0.81**	1.15 – 1.25
higher vector length: 8	**0.68 – 0.72**	**1.04 – 1.13**

Table 5.6: effects different input [high fill rate + bucket size 8] (uncompressed)
`[U-I-hfr-s8]`

Optimal table parameters

Considering both low and high fill rate tables, bucket size 4, with linear probing, gives the best performance in our experiments (except for a few cases of vector length 1 with an optimal bucket size of 8, probably due to caching effects); as the minimal bucket size for inputs of vector length 8 is 8, bucket size 8 gives the best runtimes for those inputs. In general, a table with a low fill rate turned out to have a better performance, thus, in reality, this boils down to claiming as much memory as possible for the hash table. This also means that, in reality, the size of a table is fixed.

5.2.5 In practice: different input (fixed table size)

The experiments in the subsections above (Subsections 5.2.1-5.2.4) have an artificial nature. In this subsection, we present a more practical situation, in which we use a fixed-sized table of 256MiB. We start by examining the effects of different input on performance, using the default bucket size of 32. We then examine the performance effects of using optimal bucket sizes, determined in the previous subsection.

Different input [fixed table size] – `[U-I-fn]` (fixed default bucket size 32)

When fixing the table size, we observe both the effects of different input (Table 5.1, page 43) and a different fill rate for almost all inputs, including a higher fill rate for some inputs (Table 5.4, page 47). Ultimately, we only see effects for the two cases that now move to a high fill rate of 0.85 (less duplication, 2x number of (unique) vectors; vector lengths 4 and 8): a slowdown of 11% and 17%, respectively; the latter slowdown is larger, due to a higher increase in rehashing.

Different input [fixed table size + optimal bucket size] – `[U-I-fn-os]`
see Table 5.7 (this page) for a summary of table `[U-I-fn-os]`

Based on our experiments, the lowest bucket size possible seems to be the optimal setting. But we have to take into account the ratio of “lost” entries: for example, with vector length 5, bucket size 16 might be a better choice than bucket size 8, as the ratio of lost entries of the former is 1/16 compared to 3/8 = 6/16 of the latter; the amount of lost entries impacts the (effective) fill rate and, consequently, performance. We also saw that enabling linear probing helps, with bucket sizes of 4 or less.

input	**runtime (ms)**	**speedup vs. bucket size 32** `[U-I-fn]`
default (vector length: 4)	3.03 – 20.81	1.11 – 1.52
lower vector length: 1	5.93 – 32.43	1.12 – 1.36
lower vector length: 2	5.97 – 16.67	1.19 – 1.45
lower vector length: 3 [no compensation for “lost” entries]	6.24 – 11.90	1.27 – 1.60
higher vector length: 8	3.69 – 12.23	1.11 – 1.37

Table 5.7: effects different input [fixed table size + optimal bucket size] (uncompressed) `[U-I-fn-os]`

While experimenting with those optimal bucket sizes, we, however, discovered that bucket size 4 (including linear probing) was optimal for vector lengths 1 and 2, and not bucket sizes 1 and 2, respectively. The positive effects of lower bucket sizes are, apparently, too much tempered by saturation of the memory bus and high levels of branch divergence. Indeed, on a high fill rate, bucket size 4 turns out to be even *more* faster. We already saw before that lowering the bucket size had less benefits for lower vector lengths.
Similarly, we found an optimal bucket size of 16 for inputs of vector length 8.

Using those optimal bucket settings, we achieved speedups of 11-60%, compared to a fixed bucket size of 32.
The effects are very similar to the effects of a lower bucket size (Table 5.5, page 48), combined with the effects of a different fill rate, including a higher fill rate for some inputs (Table 5.4, page 47). We see less caching effects, especially for low-vector input of vector length 1, as the table size is now larger for those cases.

5.3 Conclusions

In this section, we present the most important conclusions of our extensive performance evaluation (Section 5.2).

To get a performance evaluation that can be used for fair comparisons, we first fixed the main flaws in the uncompressed hash table implementation from GPUexplore: corruption, replication and an inferior distribution.

The number of vectors and bucket size have a large impact on performance, whereas vector length and duplication have a minor impact; the total number of vectors determines performance more than the total number of elements does. The impact of fill rate is limited up to a certain level (around 0.80), but from that level, the impact becomes more and more visible.

Based on our experiments, we found these optimal settings:

- for all inputs: a low fill rate, and, as model size is, in general, not known *a priori*, the largest hash table possible
- for inputs with vector lengths 1-4: bucket size 4 + linear probing (up to 8 entries)
- for inputs with higher vector lengths: smallest or second smallest bucket size possible (8, 16 or 32; no linear probing), but taking into account the ratio of "lost" entries as this may greatly impact the (effective) fill rate and, consequently, performance (*e.g.*, for inputs of vector length 5, bucket size 8 gives ⅜ as that ratio, compared to a ratio of 1/16 for bucket size 16)

Our results differ from the results of Cassee *et al.* [18]: Considering isolated hash tables, they found that a lower bucket size only had benefits with input with a large duplication factor (20+); in our experiments a lower bucket size *always* pays off, including cases with much smaller duplication factors (1.12 and 2.00).
We do not know yet where the difference comes from. It is probably related to different GPU architectures: Cassee *et al.* used a Titan X, based on the Maxwell GPU architecture; we used a Titan Xp, based on the Pascal GPU architecture. Maxwell does not cache thread-local memory (*e.g.*, spilled registers) in the unified L1/texture cache and global memory load transactions are 128B when loaded via L1, instead of 32B [28].

Our actual implementation, including the hash function, is also different and Cassee's implementation may, consequently, use a different amount of registers, which has an impact on register spilling. We also do not know if global L1 loading was activated; in that case, memory load transactions were 128B instead of 32B and this would impact the effects of lower bucket sizes to a large extent, as we have explained before.
This may also impact the results when the hash table implementations are integrated into GPUexplore. Using our implementation, settings or our Titan Xp accelerator (based on the Pascal GPU architecture), results may be different; a lower bucket size will now maybe also pay off when our hash table is integrated into GPUexplore, in contrast to the conclusions of Cassee *et al.* [18].

6. Compressed GPU hash table (recursive)

In the previous chapter, we presented the existing uncompressed GPU-based hash table (although we had to fix multiple flaws first) and its extensive performance evaluation. In this chapter, we present a novel GPU-based hash table, using tree-based compression, based on the recursive algorithm of Laarman *et al.* (Section 3.2); the next chapter, Chapter 7, introduces another GPU-based hash table using tree-based compression, but now using a non-recursive algorithm.

This chapter starts with an overview of the stand-alone implementation of our GPU-based hash table, using tree-based compression (Section 6.1).
We then examine the impact of GPU, input and table parameters on the performance of the compressed GPU-based hash table and compare the results to the performance evaluation of the uncompressed hash table of Chapter 5 (Section 6.2). Using the results of our extensive performance evaluation, we determined optimal GPU and table parameters for each input and managed to speedup execution up to 8 times. Then, we compare those optimal results to the optimal results in the uncompressed hash table; as optimisation had much more effects with the compressed table, we managed to reduce the slowdowns of the compressed versus the uncompressed table, from 40x to 5.3x.
Next, we discuss multiple improvements we implemented to our original implementation, aimed as reducing recursive overhead: versions with less recursion and more work per recursive call (Section 6.3). We explore the impact on performance and the results show that we managed to reduce the performance gap even further.
Finally, we give the conclusions of our performance evaluation (Section 6.4).

6.1 Stand-alone implementation

Our original implementation closely follows Laarman's algorithm [19] (Section 3.2): the (implicit) compression tree is constructed by recursion (top-down).
The massive parallelism provided by GPUs is used in two ways, as in the uncompressed hash table (more on this in the 'Sequential and parallel operation' section):

- Bucket groups: Parallel insertion of vectors, by multiple bucket groups. Each warp (32 threads) inserts one vector and even more with a lower bucket size than 32.
- Node groups: A bucket is examined in parallel, each node group checking a single slot in that bucket. Node groups are similar to the vector groups of the uncompressed hash table; the elements of a compressed hash table are tree nodes, whereas the elements of an uncompressed hash table are vectors.

We start with an example to illustrate how the algorithm works (Subsection 6.1.1). Then, we explain the algorithm in detail (Subsection 6.1.2). Next, we describe several aspects related to correctness and performance (Subsection 6.1.3). Finally, we give the maximum table size, related to the number of nodes that can be referenced, and ways to improve this bound (Subsection 6.1.4).

6.1.1 Illustrative example

To illustrate how the algorithm, and tree-based hash table compression in general, works, we give an example of the subsequent insertion of the vectors (1,2,3,4), (5,6,1,2) and, again, (1,2,3,4) into an empty hash table.

To insert vector (1,2,3,4), it is used as an argument to the wrapper function `treeFindOrPut()`. This function calls the recursive function `treeRec()`, with the same vector as argument. This top-level `treeRec()` invocation now splits the vector and recursively calls `treeRec()` on the left and right subvectors, (1,2) and (3,4), respectively. The two mid-level `treeRec()` invocations now split their subvector arguments again, into subvectors (1), (2) and (3), (4), respectively, and recursively call `treeRec()` on those four subvectors. Subvectors of length 1 are base cases and the four bottom-level `treeRec()` invocations just return their subvector arguments, *i.e.*, (1), (2), (3) and (4), to their callers, the (two) mid-level `treeRec()` invocations.
The "left" mid-level `treeRec()` invocation now creates a tree node from the return values of its (two) recursive `treeRec()` calls: it creates the node (1,2), out of (1) and (2), and calls `findOrPut()` to insert this node into the hash table. The `findOrPut()` operation is an adaptation of its implementation in the uncompressed hash table and now returns a reference (*i.e.*, hash table index) *L*, to the node (1,2), that has just been inserted into the table. The `treeRec()` invocation returns this reference to its caller, the top-level `treeRec()` invocation. The "right" mid-level `treeRec()` invocation returns a reference *R*, to the node (3,4), to the same caller.
The top-level `treeRec()` invocation creates a tree node from the return values of its (two) recursive `treeRec()` calls: it creates the node (*L*,*R*), out of *L* and *R*. It then calls `findOrPut()` to insert this node into the table and returns the reference *RN* it gets to its caller, *i.e.*, to the invocation of the wrapper function `treeFindOrPut()`.
The wrapper function now checks the root bit of the tree node *RN* references to, which is in this case still unset. Now, the wrapper function sets the root bit and returns that the vector is new and has now been inserted.

During the insertion of the vector (5,6,1,2), the "right" mid-level `treeRec()` invocation tries to insert the node (1,2) by calling `findOrPut()`. The `findOrPut()` operation, however, returns a reference (*i.e.*, *R*) to the existing (1,2) tree node, instead of inserting it for another time.
The same happens with an existing tree node that contains one or two references to other nodes, instead of only vector element values. For example, during the insertion of, again, the vector (1,2,3,4), the node (*L*,*R*) already exists and the reference *RN* is returned; the wrapper function `treeFindOrPut()` now detects a set root bit and returns that the vector has already been inserted before.

6.1.2 Compression algorithm in detail

See Algorithm 6.1 (next page) for a simplified version of our recursive C function `treeRec()` that constructs the compression tree.
The type for references to tree nodes in the hash table is `indextype`, the type for vector elements and hash table entries is `inttype`; in our implementation, both are aliases for the type of an unsigned 32-bit integer (`uint32_t`). As a node consists of two 32-bit references or vector elements (or a mixture of both), it is 64-bit wide and occupies two 32-bit hash table entries; the MSB of the first entry is used as root bit.
In constructing the tree, we start with the complete vector and then recursively call `treeRec()` to construct the subtrees for the "left" and "right" parts of the vector; this process is repeated till the base case is reached, *i.e.*, a subvector of length 1. The base case just returns the subvector, *i.e.*, its sole element; the other calls return a reference to the root node of the just constructed subtree, *i.e.*, the index of that root node in the hash table. The calling invocation then creates a node, consisting of those returned references or sole vector elements, inserts the node into the hash table by calling `findOrPut()` and returns a reference to it, *i.e.*, the index of that node in the hash table.

```
indextype treeRec(inttype *vector, int vectorLength) {
  indextype result;

  if (vectorLength == 1)
    result = *vector;
  else { // length > 1
    int split = (vectorLength / 2);
    indextype node[] = { treeRec(vector, split),
                         treeRec((vector + split), // pointer arithmetic
                           (vectorLength - split)) };

    result = findOrPut(node);
  }

  return result;
}
```

Algorithm 6.1: `treeRec()`, the function that creates the compression tree (simplified)

Note that the created node may already exist in the hash table; then, the `findOrPut()` function returns a reference to the existing node.

Wrapper function: `treeFindOrPut()`

The initial `treeRec()` call returns a reference to the root node of the compression tree for the inserted vector. This reference is then used in our top-level `treeFindOrPut()` wrapper function to check whether this was already a root node, indicating that the vector was already present in the hash table, or not, by atomically checking the root bit (bitmask `ROOT_BIT_32`) and setting it:

```
FoundOrPut treeFindOrPut(inttype *vector) {
  FoundOrPut result;

  indextype rootIndex = treeRec(vector, d_vectorLength);
  result = ((atomicOr((d_table + rootIndex), ROOT_BIT_32) & ROOT_BIT_32) ?
             SEEN : NEW);

  return result;
}
```

Algorithm 6.2: `treeFindOrPut()`, top-level function for inserting vectors (simplified)

In Algorithm 6.2, `FoundOrPut` is the `enum` for indicating the result of the `find-or-put` operation: a new vector (`NEW`), an existing vector (`SEEN`) or, in the actual implementation, a full table (the table is considered full when for rehashing no hash functions are left anymore). The constants `d_vectorLength` and `d_table` refer to the length of each input vector and (a pointer to) the start address of the hash table, respectively. The actual implementation accounts for sole-element vectors, by by-passing the `treeRec()` function and inserting the vector, padded with an empty (32-bit) element to get a 64-bit node, directly into the hash table.

Adaptations to `findOrPut()`

As `findOrPut()` now only operates on 64-bit (tree) nodes, *i.e.*, a combination of two 32-bit hash table entries, each node group (was: vector group) now consists of two CUDA threads, in which one thread checks the first entry of a slot in the table and the other thread the other (last) entry. If the slot is empty, the node group leader (the first thread in a node group) tries to claim the full slot, by a 64-bit `atomicCAS()` operation; the other thread synchronises on the result. This result now also includes the case that a concurrent invocation inserted the same node; then the hash table reference to that node is returned, instead of inserting the same node again (in a different slot/bucket), as the uncompressed implementation from GPUexplore 1.0 (Subsection 5.1.2) does.
We have implemented an improvement (Subsection 6.3.2), in which we have reduced each node group to only one thread, by each thread now checking a full 64-bit slot, instead of only a 32-bit wide entry of it.

The `findOrPut()` function now returns a reference, to the inserted node, instead of returning whether the inserted vector was new or not; now, the `treeFindOrPut()` wrapper function has the responsibility for doing this. The adapted `findOrPut()` also allows for a full table, by returning a reserved value in that case; the actual implementation of `treeRec()` propagates this value to its caller(s) and, eventually, returns this value to the calling `treeFindOrPut()` invocation.

Sequential and parallel operation

Our implementation of tree-based compression supports the parallel processing of multiple vectors within a warp, *i.e.*, by multiple bucket groups. This is directly related to the set bucket size: with a bucket size of 8, each warp has four bucket groups, each consisting of eight threads. In this case, each bucket group has four node groups, consisting of two threads each; node groups are only relevant in `findOrPut()`.
Bucket groups are also used in the `treeRec()` and `treeFindOrPut()` functions. Hence, when a warp executes a `treeRec()` or `treeFindOrPut()` operation, each thread in a warp only operates on the vector of his bucket group; all threads in a bucket group do exactly the same, except in the operation of the `findOrPut()` function, in which the threads of a bucket group are separated into node groups and each node group operates on his own slot (of the same bucket) in the hash table; each thread in a node group operates on his own entry in that slot.
For each bucket group (vector), constructing the compression tree, *i.e.*, inserting its nodes to the hash table, is, however, done sequentially and follows the recursive call order (depth-first search). Chapter 7 introduces a method to construct the tree in parallel and bottom-up, *i.e.*, starting at the leaf nodes and then working upwards.

6.1.3 Correctness and performance aspects

This subsection describes several aspects related to correctness and performance.

Intrinsic benefits over uncompressed implementation

As the final version of our uncompressed table, our compressed hash table is replication-free. But this is natively supported, in contrast to the spinlocking solution in the uncompressed hash table. As spinlocking may hurt performance, a native solution, as in our compressed hash table, is preferred.

In contrast to the uncompressed hash table implementation, which supports vectors up to 32 entries, our compressed implementation supports vectors of arbitrary length. Additionally, hash table entries are only "lost" with vectors of length 1, when they are padded; but, in the case of input of vector lengths 1 and 2, benefits are very limited anyway, as no compression is possible and the benefits can only come from implementation differences, *e.g.*, the absence of spinlocking.

Reading non-`volatile`

When initially checking the full bucket in parallel, the memory reads are still done in a non-volatile way, *i.e.*, may return stale values (in our case, the empty entry value). This still cannot lead to false positives (falsely concluding vector is not new), but, in contrast to the uncompressed implementation from GPUexplore 1.0 (Subsection 5.1.2), can now also not lead to replication (false negatives): as the subsequent `atomicCAS()` function call returns the actual (non-cached) value and this value is now checked, an already existing node will be detected at this point. This may, however, impact performance, but experimenting with a version that reads in a volatile way, did not lead to a better performance, independent of target architecture and L1 loading for global reads.
In conclusion, our compressed hash table implementation is correct (no false positives and no false negatives), despite reading in a non-volatile way. Enabling `volatile` reading does not increase performance, hence we keep reading in a non-volatile way.

Integration of strongly universal hash function

As in the uncompressed hash table (Subsection 5.1.4), we integrated the strongly universal hash function of Section 2.3. We chose for the 'Pair-multiply-shift' variant, as a parallel version has no benefits: we need to hash 64-bit tree nodes, consisting of a *pair* of 32-bit entries; parallel processing has only benefits for multiple pairs.

We see the same effects as in the uncompressed hash table:
With random-data input, rehashing was already low using GPUexplore's hash function and performance benefits come from a faster hash function, with much less benefits on lower bucket sizes.
Much more effect with real-world data, using a large table of 6GiB. Then, rehashing decreased up to 100x and resulting runtimes were up to 16% lower. Optimal execution configurations and bucket sizes were again different, as using GPUexplore's hash function requires "sub-optimal" settings to mitigate the effects of massive rehashing.

All experimental data can be found in the `[Clr64-H]` table in Appendix B.2.

Low-level optimisations

We again tried some low-level optimisations, on bucket sizes 32 and 8, and still using the old hash function implementation of Cassee:

- CC 6.1 as target architecture (`sm_61`): negative, as this leads to a higher register pressure and stack frames, including those for the recursive calls
- forcing register spilling (`maxrregcount`) has a slight effect with `sm_61` (more effect with enabled L1 loading for global reads); no effect with `sm_30`, as its register usage does not inhibit achieving maximal occupancy

Again, we have not used any optimisation we tried.

6.1.4 Maximum table size

Having one large hash table, instead of several, smaller, hash tables for each node location in the tree, not only enables maximal sharing, but we can also claim as much memory as possible for the table (see Section 3.2, 'Maximal sharing', page 26).
As one bit of a node is used to indicate whether it is a root node, each node entry can reference up to 2^{31} hash table entries. One value is, however, reserved for indicating the empty entry. Additionally, for implementation matters, we require that the number of entries is a multiple of 32 (number of threads in a warp). Therefore, our hash table can have at most 2^{31} - 32 entries.
As each entry is 32-bit (= 4 bytes) wide, this corresponds to a table of (2^{31} - 32) entries * 4 B/entry ≈ 8GiB. In practice, the average size per compressed state vector is almost the size of a root node, which is, in our case, 8 bytes. This means that our 8GiB table can store up to 8GiB / 8 B/root node ≈ 1,000,000,000 state vectors, of arbitrary length.

As GPU memory sizes are relatively small and compression dramatically reduces space requirements for model checking, this may be enough for now, but several options exist for increasing this maximum.
First, we can use a bit array to indicate root nodes; this would also fix the issue that our current implementation supports 31-bit integers only, instead of full 32-bit integers. A bit array was not an option for the CPU multi-core implementation, as this would introduce false sharing, caused by the cache coherence protocol. GPUs do not have a cache coherence protocol, but an additional global memory access, to the bit array, may hurt performance much.
Another option is changing the hash table references in such a way, that they reference (64-bit) nodes instead of (32-bit) entries. We do so in our 64-bit improvement (Subsection 6.3.2), mentioned before.
Both options double the maximum table size and can be used in conjunction.

6.2 Performance evaluation (random data)

In all experiments, we compare the results to the results of the uncompressed hash table (Section 5.2), without linear probing, as linear probing will only be added to our compressed implementation in Subsection 6.3.2. We focus on the results of inputs with vector length 3 or higher, as only they are compressible.

Performance evaluation is similar to that of the uncompressed table (Section 5.2):
We start our performance evaluation by determining optimal execution configurations for each input and table parameter, including combinations (Subsection 6.2.1); they are used in all subsequent experiments.
We then examine the impact of different input (Subsection 6.2.2) and table (Subsection 6.2.3) parameters on performance. We have now also included compression ratio as input parameter. We also try to find performance dependencies between parameters. Next, we examine the impact of the *combination* of different input and table parameters on performance (Subsection 6.2.4), including comparison to the experiments of Subsection 6.2.2. The results are used to find optimal table parameters, in particular the optimal bucket size, for each type of input.
Whereas the experiments listed above are synthetic, our last experiments resemble a more practical situation (Subsection 6.2.5), with a fixed-sized table of 256MiB. We first experiment using the default bucket size of 32. Next, we use the optimal bucket size(s) found before and compare the results to those obtained using bucket size 32.

Table 4.1 (page 31) gives an overview of all experiments. For more details, see 'Research methodology' (page 30) and Section 4.2: Test setup (page 32).

In this performance evaluation we use parameterised random data. In Chapter 8, we do performance evaluation with real-world data. That chapter also recaps the results of our final experiment, using the found optimal bucket size(s), from Subsection 6.2.5.

The first four subsections (Subsections 6.2.1-6.2.4) give an in-depth understanding of the factors that impact the performance of the hash table, but are not necessary for understanding the general ideas; the last subsection (Subsection 6.2.5) and Section 6.4: Conclusions present the general ideas. Section 6.4 also presents the conclusions of the performance evaluation of our improvements from Section 6.3.

The table identifiers in the headings refer to the tables in Appendix B.3.2.

6.2.1 Optimal execution configuration – `[C-E, -vl1, -vl2]`

Our stand-alone compressed hash table uses 32 registers (no shared memory usage), enabling maximal occupancy. Consequently, we again started with our default block dimension of 256 threads and grid dimension of 240 blocks.

However, *lowering* the grid dimension, resulted in lower runtimes: 114ms (grid dimension 90, 0.375x underloading) versus 147ms (grid dimension 240), a decrease of 22%. Lowering the grid dimension even further, to 60 blocks, was counterproductive, yielding a runtime of 126ms. We tried block dimensions of 128 and 512 threads, with various grid dimensions, but we could not measure any differences.
Using other input or table parameters resulted in the same optimal execution configuration (*i.e.*, a grid dimension of 90 blocks), except for lower bucket sizes and non-compressible input of vector length 1.
To get optimal runtimes on the lower bucket sizes 8, 4 and 2, we had to lower the grid dimension further, to 60 blocks. For inputs of vector length 2, we had to lower the grid dimension even more, to 30, on bucket size 2.
For the non-compressible input of vector length 1, the 4x overloading execution configuration of 256 threads per block and 960 blocks yielded the best runtime, of 64.8ms, compared to the default runtime of 69.8ms. When changing the bucket size, we also had to adapt the grid dimension to get the lowest runtimes: 150 blocks (bucket size 16; 0.625x underloading;), 90 blocks (bucket size 8; 0.375x underloading), 60 blocks (bucket size 4; 0.25x underloading) and 30 blocks (bucket size 2; 0.125x underloading), all using a block dimension of 256 threads.

We had to lower the grid dimension, *i.e.*, exposing, counterintuitively, *less* parallelism to the GPU, to damp the disadvantageous effects of recursion: The (recursive) call stack resides in device memory, but is cached in (per-SM) L1 caches and the (device-wide) L2 cache. Apparently, from a certain point of (concurrent) recursion, those caches are overloaded and too many stack frames have to be loaded from device memory, which is relatively slow and has a large latency. The amount of (concurrent) recursion is only defined by the vector length (depth of recursion) and execution configuration, not by other parameters, such as bucket size (a lower bucket size only increases the number of processed vectors per recursive call).

This observation is reinforced by our experiments:
As vector length 1 is the base case, no recursion will take place for inputs of that length and even an *overloading* execution configuration resulted in the lowest runtimes. We had to lower the grid dimension on lower bucket sizes to lower the amount of (too many) concurrent memory requests, which only led to a saturated memory bus. In the uncompressed table, we saw the same saturation, but we did not have to lower the grid dimension, as it did not suffer from the (code) overhead of the compression algorithm.
For inputs of vector length 2, we had to lower the grid dimension on lower bucket sizes as well, as recursive depth is limited here. For the other vector lengths, this also applies, but to a lesser extent, as recursive depth is larger and the effects of recursion determine performance more than saturation of the memory bus does; hence, we still need a grid dimension of 60 blocks to get optimal performance on bucket size 2, in contrast to a grid dimension of 30 blocks on vectors lengths 1 and 2.

Our improvements (Section 6.3) aim at removing the observed recursion overhead: First, by lowering the amount of recursive calls through incorporating more base cases, *i.e.*, decreasing the depth of recursion (Subsection 6.3.1). Next, by doing more work per recursive call: doubling the number of processed vectors per call (Subsection 6.3.2).

6.2.2 Different input – `[C-I]` (fixed low fill rate 0.24 + fixed default bucket size 32)

see Table 6.3 (next page) for a summary of table `[C-I]`

We first examine the impact of input parameters (vector length, duplication, number of vectors and now also compression ratio) on performance in isolation. Next, we look at the impact of a combination of input parameters on performance, to find any performance dependencies.

Vector length: The effect of a different vector length is the opposite of the effect in the uncompressed case: Now, a lower vector length, keeping the total number of elements the same, results in a *lower* runtime. For example, vector length 2 is 1.25x as fast as vector length 4 and vector length 8 is 0.87x as fast. This directly relates to the depth of the recursion; vector length 1 is 2.1x as fast, as no recursion takes place. But the effects of a different vector length are weaker than in the uncompressed table.
Consequently, the related effect of a different vector length, but now keeping the number of vectors the same (and varying the total number of elements), still shows a positive correlation between vector length and runtimes, but the effect is now much stronger, especially for vector length 1: vector length 1 is now 8.6x as fast as vector length 4 (versus 1.2x for the uncompressed table), length 2 2.5x as fast (versus 1.1x) and length 8 0.4x as fast (versus 0.9x).
This also impacts the slowdowns versus the uncompressed table, which primarily depend on vector length, corresponding to recursive depth: Vector length 1 is only 1.7x as slow, due to the (code) overhead introduced by the compression algorithm, *e.g.*, setting the root bit (which is not necessary in this case). But vector length 2 is already 5.2x as slow, increasing to 8.7x for vector length 3, 12.2x for vector length 4 and a huge 25.8x for vector length 8.

Duplication: The effect of less duplication (more (atomic) writes) is very limited; the recursive calls are the bottleneck to performance and less duplication does not impact that. Compared to the uncompressed case, the effect is even more limited, even for vector length 1, and, consequently, the resulting slowdown is about 10% smaller.

input	runtime (ms)	speedup vs. default of vector length	speedup vs. vector length 4	slow-down vs. unc. [Table 5.1]
default (vector length: 4)				
default (8,000,000 vectors, duplication 2.00, compr. ratio 0.51)	113.7	*1*	*1*	12.2
less duplication: 1.12	115.9	0.98	*1*	**11.1**
0.5x number of (unique) vectors	57.1	1.99	*1*	**12.4**
2x number of (unique) vectors	221.7	0.51	*1*	**11.7**
worse compression ratio: 0.73	115.9	0.98	*1*	**12.5**
lower vector length: 1 ***[with 2/1 compensation for "lost" entries]******				
default (same total number of elements → 32,000,000 vectors)	55.4	*1*	2.05	**1.7**
0.25x number of (unique) vectors: 8,000,000 vectors	13.2	4.21	[8.64]	1.7
lower vector length: 1 ***[no compensation for "lost" entries]******				
default (same total number of elements → 32,000,000 vectors)	63.3	*1*	1.80	2.0
*lower vector length: 2**				
default (same total number of elements → 16,000,000 vectors)	91.0	*1*	1.25	**5.2**
0.5x number of (unique) vectors: 8,000,000 vectors	45.4	2.03	[2.50]	5.3
*lower vector length: 3***				
default (same total number of elements → 10,666,666 vectors; ***compr. ratio 0.73****)*	116.7	*1*	0.99	**8.7**
lower number of (unique) vectors: 8,000,000 vectors	87.1	1.34	[1.33]	8.7
higher vector length: 8				
default (same total number of elements → 4,000,000 vectors)	130.2	*1*	0.87	**25.8**
2x number of (unique) vectors: 8,000,000 vectors	257.7	0.51	[0.44]	25.3

* speedup vs. vector length 4: comparison to compression ratio 0.51
** slowdown vs. uncompressed: comparison to no compensation for "lost" entries

[z.zz] is speedup vs. vector length 4, 8,000,000 vectors (+ same duplication)

Table 6.3: effects different input (compressed) **–** *[C-I]*

Number of vectors: Varying the number of vectors has a (positive) linear effect on runtimes, as in the uncompressed hash table. The effect is a little bit different from the effect in the uncompressed hash table, probably due to caching effects (to get the same fill rate, the compressed table size is ½ of the uncompressed table size, for inputs with compression ratio 0.51). Hence, the slowdowns versus the uncompressed hash table are different from 12.2x.

Compression ratio: When switching to less internal duplication, resulting in a worse compression ratio (0.73 instead of 0.51), runtimes are now a little bit higher (2%); as in the case of less duplication, more (atomic) writes are needed, but as, again, the amount of recursion is not affected, the effect is very limited. As the compression ratio does not impact the uncompressed hash table, the resulting slowdown is a little bit larger.

Compensation for "lost" entries (see Section 4.2): For most inputs of vectors 1, compensation paid off and resulted in much lower runtimes, up to 12%, probably due to the large compensation factor of 2/1; in some other cases, runtimes were equal or even higher, probably, again, due to caching effects (no compensation means a smaller table).

Combinations/dependencies: We could not find any (other) clear dependencies, apart from some minor caching effects.

6.2.3 Different table parameters – `[C-T]` (fixed default input)
see Table 6.4 (next page) for a summary of table `[C-T]`

We first examine the impact of table parameters (table size/fill rate and bucket size) on performance in isolation. Next, we look at the impact of the combination of the table parameters on performance, to find any performance dependencies.

Table size/fill rate: Changing the fill rate has minimal effects. As the uncompressed table showed a small effect when switched to a high fill rate of 0.80, the associated slowdown is a little bit smaller (11.7x versus 12.2x). We also see the effects of compression, when comparing to the same table size (instead of the same fill rate): a table size of 76.8MiB gives a high fill rate uncompressed, but a medium one compressed.

Bucket size: By contrast, lowering the bucket size has a large effect, as more work is done per (recursive) call, *i.e.*, more vectors are processed per call. For example, the fastest bucket size, of 2, is 5.5x as fast as bucket size 32. The effect of a one step lower bucket size is almost constant (ca. 1.7x speedup), except when lowering the bucket size from 4 to 2 (ca. 1.1x speedup). As the effect is much stronger than in the uncompressed table and is sustained up to a bucket size of 2 (instead of 8), the associated slowdowns, comparing to the uncompressed table, dramatically decrease from 12.2x (bucket size 32) to 3.2x (bucket size 2, compared to bucket size 4 uncompressed).

Combination/dependencies: With a high fill rate of 0.80, the effect of lowering the bucket size is less, up to 13%. Now, bucket size 2 is only 4.8x as fast as bucket size 32. The benefits of a lower bucket size are now more tempered by increased branch divergence and rehashing. But as this phenomenon is even stronger for the uncompressed table, slowdowns are getting *more* smaller for lower bucket sizes than in a low fill rate table: now, lowering the bucket size from 32 to 2 results in a 4.4x lower slowdown, compared to a 3.9x lower slowdown in the low fill rate (0.24) table. This is even more apparent when comparing to the same table size, of 76.8MiB: the slowdown corresponding to bucket size 2 is now only 2.3x and lowering the bucket size from 32 to 2 results in a 5.0x lower slowdown (from 11.6x to 2.3x).

Optimal table parameters: For all fill rates, bucket size 2 gives the lowest runtimes and bucket size 32 the highest.
Additionally, a low fill rate (large table) gives the best performance.

---

table parameters	**runtime (ms)**	**speedup vs. bucket size 32**	**speedup vs. low fill rate 0.24**	**slowdown vs. unc. [Table 5.2] *(same fill rate)***	**slowdown vs. unc. [Table 5.2] *(same table size)***
default: low fill rate (table size: 128MiB → fill rate 0.24)					
default (bucket size: 32)	113.5	*1*	*1*	12.2	12.1
bucket size: 16	64.2	1.8	*1*	**8.9**	8.8
bucket size: 8	38.6	2.9	*1*	**6.0**	5.7
bucket size: 4	23.1	4.9	*1*	**3.5**	3.2
bucket size: 2	20.6	5.5	*1*	**3.2**	2.9
very low fill rate: 2x table size: 256MiB → 0.5x fill rate: 0.12					
default (bucket size: 32)	114.1	*1*	0.99	12.3	**12.3**
very, very low fill rate: 4x table size: 512MiB → 0.25x fill rate: 0.06					
default (bucket size: 32)	114.2	*1*	0.99	-	12.3
medium fill rate: 0.5x table size: 64MiB → 2x fill rate: 0.48					
default (bucket size: 32)	113.0	*1*	1.00	12.1	-
high fill rate: 0.3x table size: 38.4MiB → 3.33x fill rate: 0.80					
default (bucket size: 32)	114.3	*1*	0.99	**11.7**	-
bucket size: 16	63.9	1.8	1.00	**7.9**	-
bucket size: 8	41.1	2.8	**0.94**	**5.2**	-
bucket size: 4	26.5	4.3	**0.87**	**2.9**	-
bucket size: 2	23.7	4.8	**0.87**	**2.6**	-
medium fill rate: 0.6x table size: 76.8MiB → 1.67x fill rate: 0.40					
default (bucket size: 32)	114.4	*1*	0.99	-	**11.6**
bucket size: 16	64.1	1.8	1.00	-	**7.9**
bucket size: 8	38.8	3.0	0.99	-	**4.9**
bucket size: 4	23.5	4.9	**0.98**	-	**2.6**
bucket size: 2	21.0	5.4	**0.98**	-	**2.3**

bucket size 2 compressed is compared to bucket size 4 uncompressed

Table 6.4: effects different table parameters (compressed) **–** *[C-T]*

6.2.4 Different input + table parameters

In the previous two subsections, we examined the performance effects of input (Subsection 6.2.2) and table (Subsection 6.2.3) parameters separately. In this subsection, we examine the effects of the *combination* of all input and table parameters on performance.
This subsection presents the results of multiple experiments: We start with the performance effects of different input on a high fill rate of 0.80. Then, we present the performance effects of different input on lower bucket sizes (8, 4 and 2). Finally, we examine the performance effects of different input on a high fill rate of 0.80 plus lower bucket sizes (again 8, 4 and 2). We compare the results to each other and to the results on a low fill rate of 0.24 plus default bucket size of 32 (Table 6.3).
The results are used to find optimal table parameters, in particular optimal bucket size, for each type of input.

Different input [high fill rate] – `[C-I-hfr]` (default bucket size 32)
see Table 6.5 (this page) for a copy of table `[C-I-hfr]`

Changing to a high fill rate (0.80) table, we could only measure significant slowdowns for inputs of vector length 1. We do not have an explanation for the latter.
In the uncompressed table (Table 5.4, page 47), we measured opposite effects: the (negative) effects were stronger for higher vector lengths, each bucket having less slots. In the compressed case, the tree nodes have, however, a fixed length of two (32-bit) elements and rehashing is only needed when all 16 slots of a bucket (of 32 elements) are occupied. Apparently, this does not happen often, even when on a high fill rate.

Consequently, the slowdowns of the compressed table, compared to the uncompressed one, are now in all cases except vector length 1, less than in the low fill rate table, dependent on vector length: ranging from ca. 0% less (vector length 2) to ca. 12% less (vector length 8); as the slowdown was (and is) much larger for higher vector lengths, the absolute reduction in slowdown is even larger: for inputs of vector length 8, the slowdown now ranges from 19.7x to 23.4x instead of 22.9x to 26.2x.

input	**speedup vs. low fill rate [Table 6.3]**	**slowdown vs. uncomp. [Table 5.4]**
default (vector length: 4, duplication 2.00)	1.00 – 1.01	11.1 – 11.9
less duplication: 1.12	0.99 – 1.01	**9.9 – 10.5**
lower vector length: 1 [with 2/1 compensation for “lost” entries]	**0.81 - 1.00**	**1.9 – 2.1**
lower vector length: 2	0.96 - 1.00	**4.7 – 5.5**
lower vector length: 3	0.99 - 1.01	**7.1 – 8.2***
higher vector length: 8	0.98 - 1.01	**19.7 – 23.4**

* comparison to no compensation for “lost” entries

Table 6.5: effects different input [high fill rate] (compressed) – `[C-I-hfr]`

Different input [bucket size 8/4/2] – `[C-I-s8/-s4/-s2]` (low fill rate 0.24)
see Table 6.6, 6.7 (this page) for copies of [C-I-s8/-s2]

Bucket size 32 → 8 (Table 6.6): Lowering the bucket size from 32 to 8 for all input types, results in speedups of 54-102% (vector length 1; non-recursive) and 166–229% (other vector lengths; recursive); the speedup effect is larger for higher vector lengths, related to recursive depth. We could not measure a dependence on duplication in this case (but we could when changing the bucket size from 8 to 4 and from 4 to 2). We could, however, measure a dependence on compression ratio: with a higher (worse) compression ratio, the speedup effects were less.

input	**speedup vs. bucket size 32 [Table 6.3]**	**slowdown vs. uncomp. [Table 5.5]**
default (vector length: 4, duplication 2.00)	2.83 – 3.01	5.7 – 6.3
less duplication: 1.12	2.75 – 3.07	**4.4 – 5.0**
lower vector length: 1 [with 2/1 compensation for "lost" entries]	**1.54 - 2.02**	**1.1 – 1.5**
lower vector length: 1 [no compensation for "lost" entries]	1.64 – 1.81	1.1 – 1.5
lower vector length: 2	**2.66 - 2.74**	**2.1 – 2.8**
lower vector length: 3	2.93 – 2.99	**3.5 – 4.4****
higher vector length: 8	**3.07 - 3.29**	**9.0 – 12.2**

Table 6.6: effects different input [bucket size 8] (compressed) – [C-I-s8]

input	**speedup vs. bucket size 4 `[C-I-s4]`**	**slowdown vs. uncomp. (bucket size 8) [Table 5.5]**
default (vector length: 4, duplication 2.00)	1.09 – 1.12	3.2 – 3.7
less duplication: 1.12	1.08 – 1.12	**2.5 – 3.0**
lower vector length: 1 [with 2/1 compensation for "lost" entries]	**0.93 - 1.07**	**1.1 – 1.6**
lower vector length: 1 [no compensation for "lost" entries]	0.80 – 1.00	1.1 – 1.8
lower vector length: 2	**0.98 - 1.17**	**1.3 – 1.7**
lower vector length: 3	1.11 – 1.12	**2.1 – 2.7****
higher vector length: 8	**1.14 - 1.38**	**4.0 – 5.9**

Table 6.7: effects different input [bucket size 2] (compressed) – [C-I-s2]

* comparison to no compensation for "lost" entries

The reason is the same as the reason why duplication made lowering the bucket size less effective in the uncompressed hash table: A higher compression ratio means more atomic operations and a lower bucket size increases the number of concurrent atomic operations. This increases the probability that an `atomicCAS()` operation has to wait for the completion (unlocking) of a concurrent atomic operation on the same table entry.

As the effect of a lower bucket size is much stronger for the compressed table, especially for higher vector lengths, the slowdowns compared to the uncompressed table are reduced to a large extent. For example, inputs of vector length 8 are now 9.0x – 12.2x as slow, on bucket size 8, compared to 22.9x – 26.2x, on bucket size 32.

Bucket size 8 → 4: Lowering the bucket size even further, to 4, has almost no effects for inputs of vector length 1, but results in lower runtimes for the other inputs, ranging from 47% to 79%; again, the effect is stronger for larger vectors.
We now compare to bucket size 8 of the uncompressed table, as bucket size 4 is uncompressed not possible for inputs of vector length 8 and bucket size 4 is slower than 8 for the other lengths. This again means a reduction in slowdowns. For example, inputs of vector length 8 are now 5.2x – 7.3x as slow as in the uncompressed table, compared to 9.0x – 12.2x (bucket size 8) and 22.9x – 26.2x (bucket size 32).

Bucket size 4 → 2 (Table 6.7): The lowest bucket size possible, 2, yields even lower runtimes for vector length 2 and higher, up to 38%, but the effect is much weaker than when the bucket size is lowered from 8 to 4. Again, almost no effects on inputs of vector length 1.
We can now see the effects of compensation for inputs of vector length 1: runtimes are increasing less for the compensated cases.
Comparing again to bucket size 8 uncompressed, slowdowns are further reduced for inputs of vector length 2 and higher; now, vector length 8 is only 4.0x – 5.9x as slow.

Different input [high fill rate + bucket size 8/4/2] – `[C-I-hfr-s8/-s4/-s2]`

Low fill rate → high fill rate (bucket size 8): Changing from a low fill rate (0.24) to a high fill rate (0.80) table, on bucket size 8, now has much more effects, in contrast to the same change on bucket size 32: runtimes are up to 21% higher. With a bucket size of 8, a high fill rate causes more rehashing, which leads to higher runtimes; with a bucket size of 32, this was not the case yet.
The effects are stronger for cases with less duplication or a higher (worse) compression ratio: in a high fill rate table, claiming a bucket slot (by `atomicCAS()`) will fail more often and the resulting negative performance effects are stronger for cases that need to claim more slots.

Bucket size 32 → 8: The effect of lowering the bucket size from 32 to 8 is a little bit smaller than in a low fill rate table (Table 6.6, previous page), but still gives much lower runtimes and still mainly depends on the vector length. As that effect is even a little bit stronger in the uncompressed table, for vector lengths 4 and 8, slowdowns, comparing to the uncompressed table, are a little bit smaller in a high than in a low fill rate table.

The effects of the same change in fill rate, but now on bucket sizes 4 and 2, are similar to the effects on bucket size 8.

Optimal table parameters

For inputs of vector lengths 3, 4, and 8, requiring a large amount of recursion, the optimal bucket size is 2, both in a low and high fill rate table; apparently, mitigating the effects of recursion, by processing more vectors per recursive call, is the most important factor. For inputs of vector length 2, only requiring two recursive calls, the best bucket size is 2 in the case of a low fill rate and 4 in the case of a high fill rate. The non-recursive inputs of vector length 1 have the lowest runtimes on bucket size 4 or 8 (low fill rate) and 8 (high fill rate).
Again, a low fill rate gives, in general, the best results and in practice this boils down to claim as much memory as possible for the hash table, *i.e.*, a fixed table size.

6.2.5 In practice: different input (fixed table size)

The experiments in the subsections above (Subsections 6.2.1-6.2.4) have an artificial nature. In this subsection, we present a more practical situation, in which we use a fixed-sized table of 256MiB; this is the same size as in the experiments with the uncompressed hash table, to account for the space (fill rate) reductions achieved by compression. We start by examining the effects of different input on performance, using the default bucket size of 32. We then examine the performance effects of using optimal bucket sizes, determined in the previous subsection.

Different input [fixed table size] – `[C-I-fn]` (fixed default bucket size 32)
see Table 6.8 (this page) for a summary of table *`[C-I-fn]`*

We now observe both the effects of different input (Table 6.3, page 61) and a different fill rate (Table 6.5, page 64). We could only measure significant effects for inputs of vector length 1: slowdowns up to 21%. We still do not have an explanation for this.
As switching to a fixed table size caused a high fill rate for some inputs of vector length 4 and 8 in the uncompressed table (but not in the compressed table) and, consequently, higher runtimes, we now see a small reduction in slowdowns for those vector lengths, compared to Table 6.3 (page 61).

input	speedup vs. fixed low fill rate 0.24 [Table 6.3]	slowdown vs. uncompr. `[U-I-fn]`
default (vector length: 4)	0.98 – 1.03	9.7 – 12.6
lower vector length: 1 [no compensation for “lost” entries]	**0.79 – 0.98**	**1.9 – 2.1**
lower vector length: 2	1.00 – 1.06	**4.8 – 5.3**
lower vector length: 3	0.99 – 1.00	**7.7 – 8.7***
higher vector length: 8	0.99 – 1.02	**19.1 – 25.8**

* comparison to no compensation for “lost” entries

Table 6.8: effects different input [fixed table size] (compressed)
`[C-I-fn]`

Different input [fixed table size + optimal bucket size] – `[C-I-fn-os]`
see Table 6.9 (this page) for a summary of table `[C-I-fn-os]`

Using the optimal bucket sizes determined before, we achieved speedups of 66-117%, for inputs of vector length 1, and speedups of 268-727%, for the other inputs, compared to a bucket size of 32 (Table 6.8, previous page); we achieved higher speedups with larger vectors and, consequently, larger vectors are now again faster (with the same number of elements). As the speedups achieved by optimal bucket sizes are much smaller in the uncompressed table (Table 5.7, page 50), we have managed to dramatically reduce the slowdowns, especially with larger vectors, up to a reduction of 6.1x.

input	speedup vs. bucket size 32 [Table 6.8]	slowdown vs. uncompr. [Table 5.7]
default (vector length: 4)	4.0 – 5.7	2.2 – 3.8
lower vector length: 1 [no compensation for “lost” entries]	1.7 – 2.2	1.1 – 1.5
lower vector length: 2	3.7 – 4.9	1.3 – 1.6
lower vector length: 3	4.6 – 4.8	2.2 – 2.9*
higher vector length: 8	5.0 – 8.3	3.1 – 5.3

* comparison to no compensation for “lost” entries

Table 6.9: effects different input [fixed table size + optimal bucket size] (compressed) `[C-I-fn-os]`

6.3 Optimised implementations

As recursive overhead seems to be an important performance limiter, we implemented several variants aimed at reducing that overhead: First, we decreased the recursive depth by incorporating more base cases (Subsection 6.3.1). On top of that optimisation, we increased the work that is done per (recursive) call by doubling the number of vectors that is processed per each call (Subsection 6.3.2), including the additional optimisations of linear probing and fixing the bucket size to 2.
We have again used our extensive performance evaluation method and compared each optimisation to its previous implementation. As the effects that we found are similar to the effects we already described in detail in Section 6.2, we only describe the highlights and main differences. All experimental data can be found in Appendix B.3.2.
In Chapter 8, we will use the results to measure the achieved reductions in slowdowns with respect to the uncompressed table.

6.3.1 Less recursion: more base cases

In our original implementation (Section 6.1), only subvectors of length 1 were base cases in the recursive `treeRec()` function and the wrapper function `treeFindOrPut()` bypasses the `treeRec()` function for sole-element vectors. In our optimised implementation, vectors of length 1 and (sub)vectors of length 2 are now base cases in `treeRec()`, saving two recursive calls for the latter; `treeFindOrPut()` does not need to bypass `treeRec()` anymore for vectors of length 1 (the `treeRec()` function does the required padding now).

Only one recursive call is now required for (sub)vectors of length 3, for the subvector of length 2, instead of, in total, four calls, saving three recursive calls. Larger (sub)vectors still require two (direct) recursive calls, but in those cases at least four recursive calls are saved overall.

Performance evaluation (random data)
see Table 6.10 (this page), 6.11 (next page) for a summary of `[Clr-I-fn, -os]`

The implementation now only requires 26 registers, instead of 32, but as both do not restrict achieving maximal occupancy, there are no benefits stand-alone.
As the pressure of recursion on the caches is now less, we need less underloading and can increase the grid dimension to get maximal performance: Compressible inputs (vector length 3 and higher) now use a grid dimension of 120 blocks (instead of 90) on bucket sizes 32 and 16, and a grid dimension of 90 blocks (instead of 60) on bucket size 8. Inputs of vector length 2 now achieve maximal performance with grid dimension 150 on bucket size 32 (was: 90), 120 on bucket size 16 (was: 60) and 90 on bucket size 8 (was: 60); they are the same as for inputs of vector length 1.
The optimal grid dimensions on bucket sizes 4 and 2 have not changed and are still low (30 or 60). Apparently, the performance bottleneck here is not recursion anymore, but the saturated memory bus.

The optimal bucket size for inputs of vector lengths 4 and 8 is still 2, the lowest possible. The optimal bucket size for smaller vector lengths is now 4, as they need no or only one recursive call and do not benefit from the additional vector that is processed by switching to bucket size 2.

On bucket size 32 (Table 6.10, this page), less recursion really pays off and leads to a speedup of 29-91%, for vector lengths 2 and higher. The effects are stronger for lower vector lengths, as the number of recursive calls are then relatively more reduced; *e.g.*, inputs of vector length 4 now require 2 calls (less rec.)/6 (more rec.) = 0.33x as many recursive calls, whereas inputs of vector length 8 now require 6 (less rec.)/14 (more rec.) = 0.43x as many recursive calls.
Inputs of vector lengths 1 are slower, probably because no bypassing takes place for those inputs anymore and calling `treeRec()` (non-recursively) leads to one call stack frame being allocated, as the CUDA compiler cannot detect at compile time that `treeRec()` will only be called non-recursively for those inputs.

input	**speedup vs. more rec. [Table 6.8]**
default (vector length: 4)	1.50 – 1.58
lower vector length: 1 [no compensation for "lost" entries]	**0.59 – 0.74**
lower vector length: 2	**1.89 – 1.91**
lower vector length: 3	**1.45 – 1.46**
higher vector length: 8	**1.29 – 1.34**

Table 6.10: effects different input [fixed table size] (compressed, less recursion)
`[Clr-I-fn]`

Although the runtimes of the implementation with less recursion are decreased to a large extent when instead an optimal bucket size is used (Table 6.11, this page), the effects of a lower bucket size are less profound than in the implementation with more recursion (Table 6.9, page 68). Consequently, most of the performance benefits seen on bucket size 32 (Table 6.10, previous page) disappear on the optimal bucket size. Now, the implementation with less recursion is only up to 21% faster (instead of 91%); inputs of vector length 1 are, however, now less slower than on bucket size 32.
Apparently, changing the bucket size to the optimal one in the implementation with more recursion already removed recursive overhead to such a large extent, that the (additional) effects of incorporating more base cases (*i.e.*, less recursion) are limited.

input	**speedup vs. bucket size 32 [Table 6.10]**	**speedup vs. more rec. [Table 6.9]**
default (vector length: 4)	2.8 – 4.2	1.09 – 1.15
lower vector length: 1 [no compensation for "lost" entries]	1.8 – 2.5	0.79 – 0.93
lower vector length: 2	2.5 – 2.6	1.01 – 1.21
lower vector length: 3	3.7 – 3.8	1.15 – 1.17
higher vector length: 8	4.2 – 7.2	1.08 – 1.14

Table 6.11: effects different input [fixed table size + optimal bucket size] (compressed, less recursion)
[Clr-I-fn-os]

6.3.2 More work per recursive call: fully 64-bit operations

We can even further reduce the recursive overhead by letting each thread operate on a full 64-bit node in the `findOrPut()` function, instead of reading a single 32-bit entry (the atomic compare-and-swap operation is 64-bit, and is done by the node group leader only). Each node group now consists of only one thread, instead of two. This essentially means there is no notion of node groups anymore.
We now also use references to (64-bit) nodes instead of (32-bit) entries. This means a node entry can now reference up to 2^{31} - 1 64-bit nodes, instead of 2^{31} - 1 32-bit entries. This corresponds to a (almost) 16GiB table, instead of a 8GiB one, and about 2,000,000,000 state vectors of arbitrary length, instead of 1,000,000,000.
As each warp now hosts 32 "node groups", each thread operating on a full 64-bit node (two 32-bit entries), the maximum bucket size increases to 64 entries (32 nodes); with a bucket size of 2, each warp now hosts 32 bucket groups (each bucket group containing one thread), instead of 16 (each bucket group containing two threads).

Linear probing

As in the uncompressed hash table (page 46), we can use linear probing to mitigate the effects of reduced global load efficiency, on bucket sizes 2 or 4. This also saves costly rehashing. This is even more important for the compressed hash table, as the bucket size can always be 2 (or 4), whereas this is not the case in the uncompressed table (in which the bucket size cannot be smaller than the vector length).

Fixed bucket size (of 2)

Based on the observation that a bucket size of 2 gives (almost) the best compressed performance in general, we can set it fixed. Previous experiments show that making a parameter configurable can hurt performance, even if the parameter's value is set to the default value [18]. As each bucket group now exists of (a "node group" of) one thread, the notion of bucket groups essentially disappears.
Fixing the bucket size is not possible in the uncompressed hash table, as the bucket size has to be at least as large as the length of the input vectors, which is not fixed.

Performance evaluation (random data)
see Table 6.12, 6.13 (next page) for a summary of `[Clr64-I-fn, -os]`

Register usage is the same as for the 32-bit version. But the size of the stack frame for the recursive `treeRec()` function is dramatically reduced from 40 to 24 bytes. This means more recursive calls can be "active" concurrently before the caches are overloaded with stack frames, *i.e.*, recursion now limits performance to a lesser extent.
The optimal grid dimensions are now the same for each vector length: 120 (bucket size 32), 90 (bucket size 16), 60 (bucket size 8) and 30 (bucket sizes 4 and 2). Apparently, the optimal grid dimension is not determined by vector length/recursive depth anymore, but by saturation of the memory bus: a lower bucket size means more concurrent memory transactions and they saturate the memory bus.

With bucket size 4 and especially bucket size 2, adding linear probing up to 8 entries increases performance up to 24%. Adding linear probing has more effects in a table with a high fill rate, as it then saves more rehashing than in a table with a low fill rate.
We could not measure any significant benefits from increasing linear probing up to 32 entries. The same happened when we set the bucket size fixed to 2, even with linear probing (up to 8 entries).

The optimal bucket size for inputs of vector lengths 4 and 8 is still 2, the lowest possible, to reduce recursive overhead as much as possible. But the optimal bucket size for the smaller vector lengths 1, 2 and 3 is now 4 (low fill rate) or 8 (high fill rate; including vector length 4), as they still need no or only one recursive call and do not benefit from the additional vector that is processed by switching to bucket size 2.
In conclusion, a lower vector length and/or a high(er) fill rate require a higher bucket size to get optimal performance.

On bucket size 32 (Table 6.12, next page), the 64-bit implementation is much faster than the 32-bit one, up to 227%. The effects are much stronger than the effects of less recursion (previous subsection). The effects are still dependent on vector length, but we now see opposite effects: a lower vector length gives a lower speedup, related to recursive depth; as lower vector lengths have less recursive depth, they benefit less from reducing recursive overhead (*i.e.*, by doing more work per recursive call).
Changing to the optimal bucket size (Table 6.13, next page) improved performance, but to a much lesser extent than in the 32-bit version (Table 6.11, page 70) and in the version with more recursion (Table 6.9, page 68). Hence, much of the performance benefits of the 64-bit version seen on bucket size 32 (Table 6.12, next page) disappear when using the optimal bucket size. The resulting speedups are, however, still up to 41%, much more than the benefits of the version with less recursion (Table 6.11, page 70), especially for larger vector lengths.

input	**speedup vs. 32b [Table 6.10]**
default (vector length: 4)	2.42 – 2.62
lower vector length: 1 [no compensation for "lost" entries]	**1.97 – 2.03**
lower vector length: 2	**2.03 – 2.05**
lower vector length: 3	**2.52 – 2.59**
higher vector length: 8	**2.84 – 3.27**

Table 6.12: effects different input [fixed table size] (compressed, less recursion, 64-bit)
`[Clr64-I-fn]`

---

input	**speedup vs. bucket size 32 [Table 6.12]**	**speedup vs. 32b [Table 6.11]**
default (vector length: 4)	1.51 – 2.04	1.24 – 1.29
lower vector length: 1 [no compensation for "lost" entries]	1.18 – 1.33	1.06 – 1.31
lower vector length: 2	1.33 – 1.38	1.06 – 1.11
lower vector length: 3	1.83 – 1.86	1.26 – 1.27
higher vector length: 8	1.99 – 3.08	1.34 – 1.41

Table 6.13: effects different input [fixed table size + optimal bucket size] (compressed, less recursion, 64-bit)
`[Clr64-I-fn-os]`

6.4 Conclusions

In this section, we present the most important conclusions of our extensive performance evaluation (Section 6.2) and optimised implementations (Section 6.3).

We implemented tree-based hash table compression using the recursive algorithm of Laarman *et al.* [19]. As the call stack is saved to local memory, which resides in slow device memory, recursive overhead turned out to be severe. Local memory is cached in L1 (and L2), but those caches are overloaded with too much concurrent recursion. Therefore, we had to reduce the grid dimension, *i.e.*, concurrent recursion, to get optimal performance.
To reduce recursive overhead itself, we had to lower the bucket size, all the way down to 2. Now, more work is done per recursive call, as those calls are at warp-level. We implemented two optimisations to reduce recursive overhead even further: less recursion by incorporating more base cases and, again, more work per recursive call by switching to 64-bit read/write operations only. Lowering the bucket size had the most effect, up to a performance speedup of 8.3x. The two implementation optimisations had minor effects, especially the incorporation of more base cases.

Inputs of higher vector length require more recursion and, consequently, recursive overhead was more profound. But as most optimisations had more effect on those inputs, this reduced the performance differences with respect to inputs of lower length.

Initially, the performance gap between the compressed and uncompressed hash table was large, dependent on vector length/recursive depth: up to 26.2x for an input of vector length 8. But our optimisations narrowed that gap, especially for inputs of higher vector length, also because the effects of a lower bucket size were much stronger in the compressed table than in the uncompressed one.

Using our most optimised recursive implementation (less recursion, 64-bit, linear probing up to 8 entries), we conclude that the number of vectors and bucket size have a large impact on performance, whereas duplication has a minor impact, especially on the default bucket size of 32.
On the default bucket size of 32, compression ratio and vector length have a minor impact; the effect of the latter is even the opposite of that in the uncompressed hash table: when keeping the total number of elements the same, a lower vector length now gives a *lower* runtime, as it requires less recursion.
On a much lower, optimal, bucket size (2-8), compression ratio and vector length have a higher impact. Now, when keeping the total number of elements the same, a lower vector length gives a *higher* runtime. The effects of both parameters are much stronger than in the uncompressed table; compression ratio does not impact performance in an uncompressed table at all. The total number of vectors determines performance more than the total number of elements does, as in the uncompressed hash table.
On the default bucket size of 32, the impact of fill rate is limited, even when it is high. On lower bucket sizes, especially 2, the impact is limited up to a certain level (around 0.80), but from that level, the impact becomes more and more visible.

The slowdown versus the uncompressed hash table primary depends on vector length: As we have removed recursive overhead to a large extent, this is now more related to the number of scattered global memory accesses required for tree-based compression; a higher vector length requires more of such accesses, hence it has a larger slowdown versus the uncompressed hash table, whose memory accesses are all non-scattered.

Based on our experiments, we found these optimal settings (for our most optimised recursive implementation):

- for all inputs: a low fill rate, and, as model size is, in general, not known a *priori*, the largest hash table possible
- for inputs with vector lengths 1-3: bucket size 4 + linear probing, up to 8 entries (low fill rate); bucket size 8 (high fill rate); in general, the fill rate is not known *a priori* (although the use of compression often leads to a low fill rate), then the best bucket size is 8, as bucket size 4 would hurt performance more in a high fill rate table than bucket size 8 would in a low fill rate table
- for inputs with higher vector length: bucket size 2 + linear probing, up to 8 entries; this is the same for both a low and a high fill rate, as mitigating recursive overhead is still more important

7. Compressed GPU hash table (non-recursive)

In the previous chapter, we reduced recursive overhead by lowering the bucket size, incorporating more base cases and using 64-bit read/write operations only. In this chapter, we will remove recursive overhead completely by designing and implementing a solution without recursion, but still using tree-based compression.
In this solution, we begin with the leaf nodes and then work upwards, finishing at the root node. While building the compression tree bottom-up, we need to (temporarily) save the references to the constructed nodes; in the recursive solution, they are saved in the recursive call stack (in local memory), but we now can specify that location by ourselves. That memory space is one of the parameters that differentiate our fifteen variants, along with two other parameters that are also relevant for GPU performance: parallelisation within each vector and type of "reduction".

First, we discuss our stand-alone implementation, including the fifteen variants we created (Section 7.1). Then, we apply our performance analysis on all variants (Section 7.2). We use this analysis to compare the variants to each other, and to the recursive compressed hash table of Chapter 6.

7.1 Stand-alone implementation

We used the most optimised recursive compressed hash table (Subsection 6.3.2) as a basis for our non-recursive implementation. This includes fully 64-bit operations, linear probing (up to 8 entries) and a fixed bucket size of 2. However, as those optimisations are primarily aimed at reducing recursive overhead, they may not be the best choice for the non-recursive implementation. For reasons of time, we have not examined whether the optimisations are optimal for the non-recursive implementation as well.

We start with an example to illustrate how the algorithm works (Subsection 7.1.1). Then, we explain the algorithm in detail (Subsection 7.1.2). Finally, we give an overview of the fifteen variants we created, including the three parameters that are used to differentiate them (Subsection 7.1.3).

7.1.1 Illustrative example

To illustrate how the algorithm works, we give an example of the insertion of the vector (1,2,3,4) into an empty hash table. In this example, when building the tree, we save the references to the constructed tree nodes to shared memory, but in a different version this could also be global memory or a thread-local register.
To insert vector (1,2,3,4), it is used as an argument to the wrapper function `treeFindOrPut()`; this is the same wrapper function as in the recursive implementation of Chapter 6. This function calls the, now, non-recursive function `treeNoRec()`, with the same vector as argument.
First, the bottom row of the compression tree is constructed. The `treeNoRec()` function creates two tree nodes: (1,2) and (3,4), corresponding to the elements of the vector. It then calls `findOrPut()` to insert these nodes into the hash table; this is the same `findOrPut()` function as in the recursive implementation of Chapter 6. Both `findOrPut()` invocations return a reference, *L* and *R*, to the tree nodes that have been inserted. The `treeNoRec()` function saves both references to shared memory.

Now, the next row of the compression tree is constructed (bottom-up), by creating new tree nodes for each pair of references and inserting them into the table. In the example, the `treeNoRec()` function creates a new tree node (*L*,*R*) and inserts this node into the table by calling `findOrPut()`. The latter again returns a reference, *RN*. The `treeNoRec()` function now replaces, in shared memory, the *L* reference by the *RN* reference. This process is repeated for any other pair of nodes in the bottom row (none in the example). As now a new row of the tree has fully been constructed, each pair of references in this new row is again bundled to get the next row bottom-up. This process is repeated until there is only one reference left: the reference to the root node. In our example, *RN* is the only reference left and this is indeed a reference to the root node of the (implicit) compression tree for the vector (1,2,3,4).
Now, the *RN* reference is returned to the caller of `treeNoRec()`, which is the wrapper function `treeFindOrPut()`; this wrapper function then checks and sets the root bit, as in the recursive implementation.

7.1.2 Compression algorithm in detail

We use the massive parallelism provided by GPUs in at least one way:

- Parallel insertion of vectors. Each warp (32 threads) inserts at least one vector.

We can also apply parallelism *within* each vector. This is one of the parameters that differentiate our fifteen versions. Then additional parallelism takes place:

- Multiple threads insert a single vector together, by constructing the bottom row and all other rows in parallel (bottom-up).

Note that no parallelisation takes place in `findOrPut()` (fixed bucket size of 2).
For more details, we refer to the description of the variants (Section 7.1.3).

Non-recursive `treeNoRec()` function

Algorithm 7.1 (next page) gives a simplified version of our non-recursive solution (see Algorithm 6.1 (page 55) for the recursive counterpart).
The type for references to tree nodes in the hash table is still `indextype`, the type for vector elements and hash table entries is `inttype`; in our implementation, both are aliases for the type of an unsigned 32-bit integer (`uint32_t`). The function parameter `buildTree` references the array where the references to the constructed nodes are saved while building the compression tree (*e.g.*, an array in shared memory). The constant `d_lowestFullRowLength` contains the number of nodes in the lowest tree row that is still full (*i.e.*, has the maximum number of nodes for that row); the constant `d_incompleteRowLength` contains the number of nodes in the incomplete row of leaves underneath (if there is one; there is such a row iff the vector length is not a power of 2).

We start by constructing the lowest rows of the compression tree: the lowest row of the tree that is still full and an incomplete row of leaves underneath (if there is one). First, all the leaves of the incomplete row are constructed by inserting two vector elements into the hash table each time; each time, two returned node references are used to construct a non-leaf node in the lowest full row and the reference that has now been returned by `findOrPut()` is saved to the corresponding location in the `buildTree` array. After the incomplete row is fully processed, the remaining (leaf) nodes of the lowest full row are constructed by inserting each time two vector elements into the hash table and saving the returned reference to the corresponding location in the `buildTree` array. Now all vector elements have been "distributed" to nodes in the compression tree and the lowest full row has been filled completely.

```
indextype treeNoRec(inttype *vector, indextype *buildTree) {
  // processing incomplete, lowest row and lowest full row:
  int fullRowIdx = 0, incompleteRowIdx;
  while ((incompleteRowIdx = (fullRowIdx * 2)) < d_incompleteRowLength) {
    indextype node[] = {findOrPut(vector), findOrPut(vector + 2)};
    buildTree[fullRowIdx] = findOrPut(node);
    vector += 4;
    fullRowIdx++;
  }
  while (fullRowIdx < d_lowestFullRowLength) {
    buildTree[fullRowIdx] = findOrPut(vector);
    vector += 2;
    fullRowIdx++;
  }
  // processing other rows bottom-up, in-place (neighboured pairs):
  int rowLength = (d_lowestFullRowLength / 2);
  while (rowLength >= 1) {
    for (fullRowIdx = 0; (fullRowIdx < rowLength); fullRowIdx++) {
      indextype *children = (buildTree + (fullRowIdx * 2));
      buildTree[fullRowIdx] = findOrPut(children);
    }
    rowLength /= 2;
  }
  // returning root node:
  return buildTree[0];
}
```

Algorithm 7.1: `treeNoRec()`, the function that creates the compression tree (simplified)

The actual implementation accounts for vectors of odd length, which requires building a (non-leaf) node that consists of a vector element plus the reference to a (leaf) node of two vector elements.

As the lowest full row has now been fully constructed, we can construct the other rows bottom-up, by constructing new nodes consisting of two references, each referencing a node in the row underneath. This is done in-place in the `buildTree` array, by overwriting a reference by the reference to the node that is constructed, in the row above, from that to-be-overwritten reference and another reference from the same row. Hence, the length of `buildTree` is the number of nodes in the lowest full row.
We repeat the procedure listed above until we arrive at the root node. Then, we return a reference to this root node to the caller of `treeNoRec()`, the wrapper function `treeFindOrPut()` (page 55); the actual implementation accounts for full tables, *i.e.*, propagating the reserved value that indicates a full table.

Different compression tree

The resulting (implicit) compression trees may be different from the ones created by the recursive solution; not in number of nodes, but in structure. Figure 7.2 (next page) gives the two different tree structures for a vector of length 6 (*R* denotes a reference; a number (*0..5*) denotes a vector element index). This may also impact the compression ratio that is achieved; in general, achieving a lower compression ratio, *i.e.*, more compression has been achieved, gives a lower fill rate and a better performance.

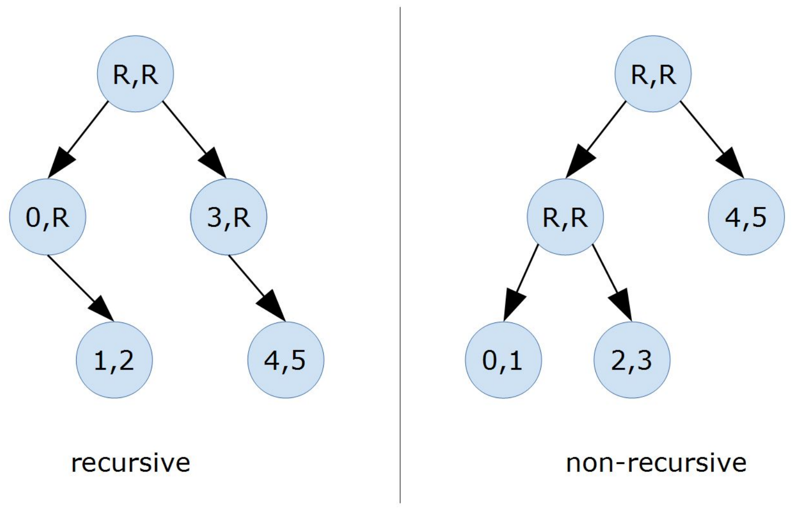


Figure 7.2: recursive/non-recursive trees (vector length 6)

Low-level optimisations

We again tried some low-level optimisations, using one variant (`l_np_sm`, see next). We still used the old hash function implementation of Cassee.

- CC 6.1 as target architecture (`sm_61`): no effect, even though this often led to a higher register pressure and larger stack frames
- forcing register spilling (`maxrregcount`) has a negative effect (independent of L1 loading for global reads enabled or not), for all cases (`sm_61`)/down-sizing vector group (`ld`) cases only (`sm_30`)
- reading in a volatile way: negative effect (both for `sm_30` and `sm_61`, both for L1 enabled or not), even more negative for the down-sizing vector group (`ld`) cases and the `l_np_gm` case (`sm_61`, L1 enabled)

Again, we have not used any optimisation we tried.

7.1.3 Variants

We have implemented fifteen variants of our non-recursive solution. They differ in the following ways, related to GPU specifics (not all combinations are possible or relevant):

- Parallelisation within vector.
    - No parallelisation (`1`), see Algorithm 7.1 (previous page): As the recursive implementation, there is no parallelisation within each vector. The nodes of the compression tree of a vector are all constructed sequentially. Parallelisation still exists, as multiple vectors are inserted in parallel (*i.e.*, 32 vectors per warp).
    - Constant-sized vector group (`l`): Each warp now consists of multiple vector groups; each vector group, which in general consists of multiple threads, is responsible for inserting a single vector. The number of threads in a vector group equals the number of nodes required for the lowest full row in the compression tree.
      First, each thread in the vector group handles the construction of a single node in the lowest full row, in parallel. Then, half of the threads each handles the construction of a single node in the row above, again in parallel. For the next row above, the number of active threads is again halved, *et cetera*, until the root node is reached.

    Although the number of inactive (divergent) threads constantly increases, we only need to synchronise across a warp, not across a (larger) thread block (as in the next case). We could make use of those inactive threads in the `findOrPut()` operation for a particular node, by letting them examine other slots in the hash table, essentially increasing the bucket size. We, however, have not implemented that extension yet.
    Still, multiple vectors are inserted in parallel within a warp, but, in general, less than in the previous case. For example, with a vector group of four threads, eight vectors are inserted in parallel within a warp, instead of 32.
  - Down-sizing vector group (`ld`): This is the same as the previous case, but the degree of divergence within warps is now reduced. This is achieved by vector groups that are down-sized to the number of threads that are required for the tree row that is being processed, *i.e.*, the number of threads equals the number of nodes in that row.
    For example, with a block dimension of 256 threads and a lowest full tree row consisting of four nodes, all 256 threads, in 256 threads / 32 threads/warp = 8 warps, are active to construct those tree rows (for 256 threads / 4 threads/vector = 64 vectors). But when constructing the nodes in the row above, only 256 threads / 2 = 128 threads, in 128 threads / 32 threads/warp = 4 warps, are active; all (other threads in the) other four warps are inactive, but as divergence can only occur within a warp, this is not an issue.
    This variation requires block-level synchronisation (`__syncthreads()`), which is, in general, slower than the warp-level synchronisation from the previous case. When the number of active threads in a block has been down-sized to 32 (or less) threads, *i.e.*, to a single warp, we could use explicit warp-level synchronisation (`__syncwarp()`); we, however, could not measure any benefits from doing so, probably because the other threads in the block are inactive and do not add any significant overhead to block-level synchronisation.
- Type of "reduction", which impacts memory access patterns and is, in particular, important when accessing global memory.
  - Neighboured pairs (`np`), see Algorithm 7.1 (page 76): Nodes are paired with their immediate neighbour node to construct a new node.
  - Interleaved pairs (`ip`): Paired nodes are separated by a given stride. As nodes are now paired in a different way, the resulting compression tree may be different from the one created by the 'neighboured pairs' method (in structure, not in number of nodes); consequently, the achieved compression ratio (and associated performance) may also be different.
- Memory space where (the hash table references to) the tree nodes are (temporarily) saved while constructing the compression tree.
  - Global memory (`gm`): This is the same memory space as that of the hash table and the vector input. Large, but slow.
  - Shared memory (`sm`): This memory space is shared by all threads in a thread block, so it can (also) be used in constant-sized and down-sizing vector groups. Shared memory usage may restrict occupancy.
    To avoid bank conflicts, we have also implemented variants that accesses the shared memory in a shuffled way (`s_sm`, only relevant when the 'neighboured pairs' (`np`) method is used and not relevant for constant-sized vector groups (`l`)).

- Registers/thread-local memory (`lm`): Synchronised warp shuffle functions (`__shfl_sync()`) are used to access the registers (nodes) of other threads (within the same warp). Register usage may restrict occupancy.
  Only relevant when the 'neighboured pairs' (`np`) method is used with a constant-sized vector group (`l`).

7.2 Performance evaluation (random data)

We have again used our extensive performance evaluation method and compared each version to each other and to the most optimised recursive hash table implementation (Section 6.3.2). As the effects that we found are similar to the effects we already described in detail in Section 6.2, we only describe the highlights and main differences. All experimental data can be found in Appendix B.3.3.
In Chapter 8, we will use the results to measure the achieved reductions in slowdowns with respect to the uncompressed table.

Optimal execution configuration

The variants with no parallelisation within a vector (`1`) need 32 registers (more than the recursive implementations, except for our initial implementation); the variants with a constant-sized vector group (`l`) require 32-40 registers and with a down-sizing vector group (`ld`) even more, 40-46 registers, which restricts achieving maximal occupancy. We will, however, see that we do not need to achieve maximal occupancy, but this may be different when the hash table is integrated into a model checker.

As we use a fixed bucket size of 2, saturation of the memory bus determines the optimal grid dimension and, consequently, this dimension is low: 30 blocks; in a high fill rate table (low table size), using a higher grid dimension (60 or 90) gives better runtimes, probably due to caching effects (when more memory requests are loaded from L1 or L2, a higher grid dimension can be used before saturation occurs, as L1 and L2 caches serve memory requests faster).
The patterns of performance versus grid dimension are different for variants with different types of parallelisation within a vector. For example, using a vector group (constant-sized (`l`) or down-sizing (`ld`)), a grid dimension of 120 blocks is required to get the best performance with inputs of vector length 8: those inputs have four nodes in the bottom row of their compression tree and, consequently, only eight vectors are inserted in parallel by each warp; to compensate for this, a higher grid dimension is needed to saturate the memory bus.

Variants

The performance of all variants were similar to each other. Apparently, performance is determined by all (scattered) global memory accesses to the hash table; those accesses are the same for each variant.
Consequently, when integrating the hash table into a model checker, one can choose a variant suited for that model checker. For example, when the model checker uses shared memory for other purposes, one can choose a variant that uses global or local memory to temporarily store (the references to) the constructed nodes of the compression tree.

Comparison to recursive hash table

As Table 7.3 (this page) shows, the performance benefits over the recursive implementation are very limited, with speedups of 3-14% (and an exceptional slowdown case for vector length 1). Apparently, in the most optimised recursive implementation, using an optimal bucket size, recursive overhead was already reduced to such a large extent, that the (scattered) memory accesses are now the bottleneck and not the recursion anymore.

input	**speedup vs. rec.** **[Table 6.13]**
default (vector length: 4)	1.03 – 1.09
lower vector length: 1 [no compensation for "lost" entries]	0.89 – 1.09
lower vector length: 2	1.07 – 1.10
lower vector length: 3	1.06 – 1.08
higher vector length: 8	1.08 – 1.14

Table 7.3: effects different input [fixed table size] (compressed, no recursion)
[Cnr-I-fn-s2 (l_np_sm)]

8. Performance evaluation: in practice

In the previous three chapters, we presented our stand-alone implementations of an uncompressed, compressed (recursive) and compressed (non-recursive) hash table, including an extensive performance evaluation using parameterised random data.

In this chapter, we present the practical highlights of our performance evaluation.
First, we give a summary of the results of our random-data performance evaluation using a fixed-sized table and optimal bucket size (Section 8.1); we use those results to compare the performance of the most optimal compressed hash table to that of the uncompressed hash table.
Then, we give the results of our performance analysis using state vector sequences extracted from real-world models (Section 8.2); those results show the performance of the uncompressed and compressed hash table implementations in practice and we, again, use the results to compare the performance of the most optimal compressed hash table to that of the uncompressed hash table.

8.1 Summary of practical random-data experiments

This section gives a summary of the results using a fixed-sized table of 256MiB and an optimal bucket size, for the uncompressed (Subsection 5.2.5), compressed (recursive) (Subsections 6.2.5, 6.3.1 and 6.3.2) and compressed (non-recursive) (Section 7.2) hash table implementations.

For all inputs, except vector length 1:
Of all compressed (recursive) hash table implementations, our most optimised version, the 64-bit version with less recursion (Subsection 6.3.2), gives the lowest runtimes; our initial version, the 32-bit version with more recursion (Section 6.1), gives the highest runtimes. The runtimes of the best version are 7-60% better than the runtimes of the worst version; the performance benefits are larger for inputs of higher vector length.
The runtimes of our non-recursive solution (Chapter 7), are even better: 3-14%, compared to our most optimised recursive compressed hash table, and 16-77%, compared to our initial recursive hash table implementation.

For inputs of vector length 1:
Now, the version with less recursion, but still 32-bit (Subsection 6.3.1), gives the worst runtimes. The runtimes of the other two recursive implementations are very similar. In all cases, except one, the non-recursive implementation gives the lowest runtimes and they are 13-16% lower than the runtimes of the worst version.

Table 8.1 (next page) presents the results of the comparison between the best performing compressed solution (*i.e.*, the non-recursive solution) and the uncompressed hash table (Subsection 5.1.4): Slowdowns are limited, in the range of 1-223%, and are larger for higher vector lengths; the trees of inputs of higher vector length consist of more nodes and, consequently, require more scattered memory accesses in the compressed hash table, whereas a single memory access suffices in the uncompressed hash table.

The full `[PRD]` table can be found in Appendix B.3.4.

input	**slowdown best compressed vs. uncompressed**
default (vector length: 4)	1.37 – 2.55
lower vector length: 1 [no compensation for "lost" entries]	1.01 – 1.39
lower vector length: 2	1.00 – 1.36
lower vector length: 3	1.36 – 1.88
higher vector length: 8	1.78 – 3.23

Table 8.1: effects different input [fixed table size + optimal bucket size]
[PRD]

8.2 Real-world data

Next to parameterised random data (Section 8.1), we also used state vector sequences extracted from real-world models to evaluate the performance of our implementations.
As initial test configuration, we used the optimal settings determined in the previous chapters, but for each parameter we also tried one step up and one step down (whenever possible); if that step resulted in a lower runtime, we tried another step up or down, respectively. For example, with an initial bucket size of 2, we also tried a bucket size of 4; if bucket size 4 gave a lower runtime, we also tried bucket size 8, *et cetera*.
As, in practice, one wants to claim as much memory as possible for the hash table, we used a large hash table of (almost) 8GiB (larger is not possible for the 32-bit compressed versions, as they can only reference up to 2^{31} - 1 hash table entries); as the input sizes of the `lamport8` and `szymanski5` cases are very large (multiple GiB) and inputs reside in global memory, as the hash table does, we used hash tables of 4GiB and 3GiB, respectively, for those cases.

The structure of the compression tree in the recursive implementation was for some cases different and as this affected compression ratio, and probably performance too, we then re-ordered the elements in the input data to achieve the same (or almost the same) compression ratio; we did not need this in the experiments with random-data input, as in that input internal duplication is symmetrical, in contrast to (some) real-world data.
Another important difference between the real-world data and our random-data input is the locality of duplication of vectors: in the random-data input, duplicate vectors are randomly distributed over the input, whereas, in the real-world data, duplicated vectors are often clustered. Moreover, for most real-world data, the ratio of duplication is higher than the ratio in our random-data input.

Table 8.2 (next page) presents the results of our experiments using real-world data.
In general, they are very similar to the results of our experiments with random-data input (Section 8.1): the initial compressed (recursive) implementation (Section 6.1) gives the worst runtimes; the most optimised recursive implementation (Subsection 6.3.2) and the non-recursive implementation (Chapter 7) give the lowest runtimes.

As we do not account for micro-adjustments in grid dimensions (Section 4.2), the lowest possible runtimes may actually be a little bit different. As the runtimes of the most optimised recursive implementation and the non-recursive implementation are very close to each other, they can be considered as being equal to each other. This, again, shows that performance is not limited by recursion anymore, but by the scattered hash table memory accesses, inherent to tree-based compression.
Comparing the best performing compressed implementation to the uncompressed table, we see limited slowdowns, in the range of 1.2x-3.8x, slightly related to vector length.

The full [RWD] table can be found in Appendix B.4.

input (number of vectors, duplication, compression ratio)	runtime uncompr. (ms)	runtime compressed (ms)				slowdown best compr. vs. uncompr.
		base	less rec.	less rec. + 64b	non-rec.	
vector length: 3 (min. compr. ratio: 0.67)						
1394 (355,339, 1.8, 0.84)	0.51	0.89	0.88	0.82	0.80	1.6
1394.1 (23,792,770, 2.4, 0.75)	25.7	42.4	41.5	38.4	37.4	1.5
vector length: 5 (min. compr. ratio: 0.40)						
odp (641,227, 7.0, 0.44)	0.64	1.68	1.58	1.47	1.41	2.2
odp.1 (31,091,555, 4.0, 0.40)	33.9	69.2	63.7	57.0	58.9	1.7
transit (39,925,525, 10.6, 0.57)	29.4	107	106	102	97	3.3
vector length: 6 (min. compr. ratio: 0.33)						
lamport8 (269,192,486, 4.3, 0.33)	169	654	552	375	388	2.2
szymanski5 (375,297,914, 4.7, 0.33)	269	877	727	349	335	1.2
vector length: 8 (min. compr. ratio: 0.25)						
wafer_stepper.1 (16,977,693, 4.5, 0.27)	15.8	48.0	43.5	36.2	40.4	2.3
vector length: 9 (min. compr. ratio: 0.22)						
acs (895,005, 4.5, 0.24)	0.99	3.23	2.62	1.95	2.06	2.0
vector length: 10 (min. compr. ratio: 0.20)						
asyn3 (80,686,290, 5.1, 0.20)	61.8	321	289	235	266	3.8

Table 8.2: effects different input [fixed table size + optimal bucket size] [RWD]

9. Conclusions

We first fixed the main flaws in the uncompressed hash table implementation of GPUexplore: corruption of vectors, replication of vectors and a hash function with an inferior distribution. We implemented a stand-alone uncompressed hash table, without corruption or replication, and with a fast, but still probabilistic-optimal, hash function. Results showed that performance is equal to or even better than the original implementation from GPUexplore.
Using our improved implementation, we extensively examined the impact of GPU, input and table parameters on performance. We used the results to find optimal GPU and table parameters for each type of input. This enabled a fair comparison to the compressed hash table implementations.

Then, we implemented a compressed hash table, using tree-based compression. This implementation uses a recursive algorithm and this was a large bottleneck to performance, as the call stack is saved to local memory, which resides in slow device memory. We were able to reduce the recursive overhead by lowering the grid dimension, using a low bucket size, incorporating more base cases and switching to 64-bit read/write operations only. This dramatically reduced the performance gap with the uncompressed hash table, especially for inputs of higher vector length.
We, again, extensively examined the impact of GPU, input and table parameters on performance and found optimal GPU and table parameters for each type of input. We used the results to assess the performance benefits of each optimisation and to compare the performance of the compressed to the uncompressed hash table.
To reduce recursive overhead completely, we designed and implemented a compressed hash table, still using tree-based compression, but now using a non-recursive algorithm. We created fifteen variants, based on three differentiating parameters that affect GPU performance. But the performance of all variants was similar to each other and close to the performance of our most optimised recursive implementation; we already reduced recursive overhead to such a large extent, that the scattered memory accesses inherent to tree-based compression are now the performance bottleneck.

We used both parameterised random-data input and state vector sequences extracted from real-world models to evaluate performance. Results were similar: with our optimisations we dramatically reduced the performance gap with the uncompressed hash table, from over 40x to 3.8x.
As the speedups of GPU-based model checkers, such as GPUexplore, are enormous, compared to the most sophisticated CPU multi-core solutions, that performance gap is small enough for allowing a performance-efficient integration of our compressed table into such model checkers. This would allow the verification of larger models and models with data, which is currently not possible in GPUexplore, primarily due to memory space limitations. As a result, also the verification of those kinds of models can achieve the enormous speedups provided by GPU-based model checking.

9.1 Future work

The main future work is to integrate our hash table implementations into GPUexplore, to examine whether the stand-alone results also apply when the hash tables are integrated. Previous results [18] show that effects may be different integrated. Our extensive performance evaluation could help to determine why this would be the case and to provide possibilities for addressing the causes.

Integration of uncompressed hash table

Our uncompressed hash table implementation may already speedup the current (uncompressed) version of GPUexplore. It will fix the dangerous possibility for corruption of vectors anyway. The effects of a replication-free hash table are probably limited, as previous results [5] showed that the performance effects of false negatives are not severe in GPUexplore. GPUexplore may, however, benefit from our strongly universal hash function, which is not only very fast, but also probabilistic-optimal; the current version of GPUexplore uses a hash function with an inferior distribution, especially in large hash tables. It is also interesting to examine if a lower bucket size now pays off.

Integration of compressed hash table

Most interesting is the integration of our compressed hash table, as this would allow GPUexplore to verify larger models and models with data; this may also enlarge the possibilities to specify models that already can be verified in a more convenient way. As the performance of the most optimised recursive and the non-recursive table is similar, one could integrate both, as there might be performance differences integrated.
Integration into GPUexplore would enable an important possibility to address the main performance limiter, *i.e.*, the number of scattered memory accesses: By using the incremental tree database of Section 3.2, a saved tree of the originating state can be reused in constructing the trees of successor states. Then, the hash table only needs to be accessed for the tree nodes that are different. With a vector of length *k*, this means only O(log(*k*)) (= height of tree) accesses are needed instead of O(*k*) accesses, assuming only one vector element changed. This may reduce the performance gap with the uncompressed hash table even further.
In GPUexplore, the open set (*i.e.*, the states that still need to be visited/explored) could now also be compressed, *e.g.*, by letting the open set containing references to (the root nodes of) the to-be-visited states in the closed set (hash table); before a state is added to the open set, the closed set is checked and, after that, always contains the state.

When the compressed hash table has been integrated into a GPU-based model checker, its (integrated) performance can also be compared to that of a CPU-based multi-core compressed hash table (*e.g.*, the one integrated in LTSmin [3]). We have not done a stand-alone comparison, as we think this would mean comparing apples to oranges. Our compressed hash table is also very specific to model checking (although it may also be used for other purposes), hence it makes more sense to do the comparison when it really matters, *i.e.*, integrated.

In our performance analysis using real-world data (Section 8.2), we allocated a full 32-bit vector element for each process in the model; GPUexplore applies state vector compression by allocating exactly the number of bits that are required for that process (*i.e.*, the 2-log of the number of states in the automaton). When the compressed hash table is integrated into GPUexplore, we can examine how this would affect our results.

Other future work

Unfortunately, for reasons of time, we have not used a different GPU to examine how well our compressed hash table scales with a more or less powerful GPU.
Our compressed hash table implementations are written in CUDA C and use some CUDA specifics, such as warp-level synchronisation. Therefore, it would be interesting to port the implementation to OpenCL, to examine whether the results are then similar.

References

1. C. Baier and J.-P. Katoen. 2008. *Principles of model checking*. The MIT Press, Cambridge, MA, USA.
2. H. Sutter. 2005. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's journal* 30, 3, 202-210.
3. A. Laarman, J. van de Pol, and M. Weber. 2011. Multi-core LTSmin: marrying modularity and scalability. In *Proceedings of the third international conference on NASA Formal methods (NFM'11)*, volume 6617 of *LNCS*, M. Bobaru, K. Havelund, G.J. Holzmann, and R. Joshi (editors). Springer-Verlag, Berlin Heidelberg, Germany, 506-511.
4. J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. 2013. DiVinE 3.0: an explicit-state model checker for multithreaded C & C++ programs. In *Proceedings of the 25th international conference on Computer Aided Verification (CAV'13)*, volume 8044 of *LNCS*, N. Sharygina and H. Veith (editors). Springer-Verlag, Berlin Heidelberg, Germany, 863-868.
5. A. Wijs and D. Bošnački. 2014. GPUexplore: many-core on-the-fly state space exploration using GPUs. In *Proceedings of the 20th international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, volume 8413 of *LNCS*, E. Ábrahám and K. Havelund (editors). Springer-Verlag, Berlin Heidelberg, Germany, 233-247.
6. D. Bošnački, S. Edelkamp, D. Sulewski, and A. Wijs. 2010. GPU-PRISM: an extension of PRISM for General Purpose Graphics Processing Units. In *Proceedings of the 2010 ninth international workshop on Parallel and Distributed Methods in Verification, and second international workshop on High Performance Computational Systems Biology (PDMC-HIBI '10)*, Juan E. Guerrero (editor). IEEE Computer Society, Los Alamitos, CA, USA, 17-19.
7. D.B. Kirk and W.W. Hwu. 2016. *Programming massively parallel processors: a hands-on approach* (third edition). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
8. B. Cowan and B. Kapralos. 2011. GPU-based acoustical occlusion modeling with acoustical texture maps. In *Proceedings of the 6th Audio Mostly conference: a conference on interaction with sound (AM'11)*, L. Roque and V. Alves (editors). ACM, New York, NY, USA, 55-61.
9. S.S. Stone, J.P. Haldar, S.C. Tsao, W.W. Hwu, Z.-P. Liang, and B.P. Sutton. 2008. Accelerating advanced MRI reconstructions on GPUs. In *Proceedings of the 5th conference on Computing Frontiers (CF'08)*. ACM, New York, NY, USA, 261-272.
10. J. Mulligan. 2012. A GPU-accelerated software eye tracking system. In *Proceedings of the symposium on Eye Tracking Research and Applications (ETRA'12)*, S. Spencer (editor). ACM, New York, NY, USA, 265-268.
11. Khronos OpenCL Working Group. 2017. The OpenCL specification 2.2. Retrieved 25 September 2019 from https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf.
12. NVIDIA. 2019. CUDA C programming guide 10.1.243. Retrieved 25 September 2019 from http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
13. A. Wijs, T. Neele, and D. Bošnački. 2016. GPUexplore 2.0: unleashing GPU explicit-state model checking. In *Proceedings of the 21st international symposium on Formal Methods (FM'16)*, volume 9995 of *LNCS*, J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou (editors). Springer International Publishing, Cham, Switzerland, 694-701.

14. A. Laarman, J. van de Pol, and M. Weber. 2010. Boosting multi-core reachability performance with shared hash tables. In *Proceedings of the 2010 conference on Formal Methods in Computer-Aided Design (FMCAD'10)*, R. Bloem and N. Sharygina (editors). FMCAD Inc, Austin, TX, USA, 247-256.
15. T. Neele. 2014. *Design of a scalable hash table on a GPU*. Bachelor's thesis. University of Twente, Enschede, The Netherlands.
16. L. Verkleij. 2016. *Boosting shared hash tables performance on GPU*. Bachelor's thesis. University of Twente, Enschede, The Netherlands.
17. N. Cassee and A. Wijs. 2017. Analysing the Performance of GPU Hash Tables for State Space Exploration. In *Proceedings of the 3rd workshop on Graphs as Models (GaM'17)*, volume 263 of *EPTCS*, T. Kehler and A. Millers (editors). Open Publishing Association, 1-15.
18. N. Cassee and A. Wijs. 2017. On the Scalability of the GPUexplore Explicit-State Model Checker. In *Proceedings of the 3rd workshop on Graphs as Models (GaM'17)*, volume 263 of *EPTCS*, T. Kehler and A. Millers (editors). Open Publishing Association, 38-52.
19. A. Laarman, J. van de Pol, and M. Weber. 2011. Parallel recursive state compression for free. In *Proceedings of the 18th international SPIN workshop on Model checking software*, volume 6823 of *LNCS,* A. Groce and M. Musuvathi (editors). Springer-Verlag, Berlin Heidelberg, Germany, 38-56.
20. M. Thorup. 2015. *High Speed Hashing for Integers and Strings*. CoRR, abs/1504.06804.
21. J. Tompson and K. Schlachter. 2012. An introduction to the OpenCL programming model. Retrieved 25 September 2019 from http://cims.nyu.edu/~schlacht/OpenCLModel.pdf.
22. A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. ACM, New York, NY, USA, 113-132.
23. E. Bardsley and A.F. Donaldson. 2014. Warps and atomics: beyond barrier synchronization in the verification of GPU kernels. In *Proceedings of the 6th international symposium on NASA Formal Methods (NFM'14)*, volume 8430 of *LNCS*, J. Badger and K. Rozier (editors). Springer International Publishing, Cham, Switzerland, 230-245.
24. L. Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* 28, 9 (September 1979), 690-691.
25. NVIDIA. 2019. Tuning CUDA applications for Kepler 10.1.243. Retrieved 25 September 2019 from http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html.
26. A. Wijs and D. Bošnački. 2016. Many-core on-the-fly model checking of safety properties using GPUs. *International Journal on Software Tools for Technology Transfer (STTT)* 18, 2 (April 2016), 169-185.
27. S. Blom, B. Lisser, J. van de Pol, and M. Weber. 2008. A database approach to distributed state space generation. *Electronic Notes in Theoretical Computer Science (ENTCS)* 198, 1 (February 2008), 17-32.
28. NVIDIA. 2019. Tuning CUDA applications for Pascal 10.1.243. Retrieved 25 September 2019 from https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html.
29. NVIDIA. 2019. Pascal L1 cache. Retrieved 25 September 2019 from https://devtalk.nvidia.com/default/topic/1006066/.