Boosting Shared Hash Tables Performance on GPU

Luuk Verkleij
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
l.h.verkleij@student.utwente.nl

ABSTRACT

Since 2005 it has become apparent that CPU sequential speed fails to increase exponentially, making parallel computing like GPU computing more important. The development of GPUExplore has shown that model checkers are one of the applications that could benefit from this trend. This paper contributes to increasing model checkers speed by redesigning a hash table algorithm, used by the model checkers LTSmin and GPUExplore. In this paper we compare the performance of three different designs, including a new proposed design. Based on the data we have gather, we claim that our design performance is better than the others.

Keywords

GPU, OpenCL 1.2, Hash Table, model checker, LTSmin, GPUExplore

1. INTRODUCTION

Until recently clock speed of cores increased exponentially, together with transistors. Unfortunately this trend stopped after 2005, as clock speed failed to increase exponentially, unlike transistors[5]. Motivated by this lack of performance scaling, CPU went more parallel like Graphic Cards or Graphic Processing Units (GPUs) already did. The extreme parallelism on GPUs, which are designed for processing images, are now becoming the focus of more algorithms. This is because vendors decided to open up their GPUs for general purposes computing.

One of the applications that could benefit by this parallel programming trend are model checkers. Model checkers go through as many states as possible, checking variables if they satisfy some pre-set conditions. It is easy to imagine why such an application could benefit from parallel computing, as every state has multiple state transitions. Some of these states could be checked in parallel, where GPU is strong. Wijs et al. created such a model checker, named GPUExplore[7].

An important part of a model checker is it's hash table, in which it searches for and stores states. As model checkers have specific and simpler requirements, which means that common used hash tables, like Cuckoo hashing, does not have optimal performance. This has been found with Laarman et al. with the model checker LTSmin[3] and Wijs et al. with the model checker GPUExplore[7]. Therefore Laarman et al. proposes a design based on CPU architecture that achieves good results. Wijs et al. proposes a redesign of the Laarman et al. design taking GPU architecture in mind.

In this paper we propose a redesign of the Laarman et al. algorithm. We are going to measure this algorithm against our implementation on GPU of the Laarman et al. and Wijs et al. algorithm. Using these measurements we are going to prove that our algorithm performance is better. Therefore we conclude that our redesign could be used in future model checkers to increase their performance.

In this paper we first discuss background information, which include how the Laarman et al. hash table algorithm works and was designed. The background also includes a short explanation on how GPU architecture and programming works. Afterwards we discuss Laarman et al.'s and Wijs et al.'s design of their algorithm and discuss our redesign. We discuss the implementation, in which we explain some problems and decision we have made. After this we will going to discuss the research method and results, which includes verification and testing. At last we analyse the results, make a conclusion and discuss further work.

2. BACKGROUND

2.1 Hash table

The hash table algorithm used for this research is described in a previous paper by Laarman et al [3]. The goal of this paper[3] was to realize an efficient shared state storage for model checking algorithms.

With this goal in mind, they could identify the following requirements.

- The storage only needs one operation: find-or-put (Algorithm 1). This lowers strain on memory and avoids cache line sharing in comparison to more generalized hash tables.
- The storage should not require memory allocation during operation, as this would increase the memory footprint
- The use of pointers should be avoided, as pointers take a considerable amount of memory when large spaces are explored.
- The time efficiency of find-or-put should scale with the number of processes executing it, ideally resulting in almost linear speed up.
- The storage is not required to be resizable

There where two types of hashing for the hash table considered. Hopscotch and Cuckoohashing. Cuckoo hashing has been both be considered by Laarman et al. [3] and Wijs et al. [7]. It was decided that these do not performance well and often have features that where unused. Afterwards the following design choices have been made by Laarman et al. [3] for the hash function.

- Open Addressing
- A form of collision resolution by probing.
- Linear probing on the cache line

  Probing over the cache line is by first by getting a part of the hash table of the size of the cache line. Then this part is being probed untill it founds it or inserts it. If those fail it rehash and probes the cache line again.
- Separated data in an indexed array
- Hash memorization in buckets
- A power of 2 table size
- Lockless
- Using compare-and-swap (CAS) on the buckets

2.1.1 *find-or-put*

```
   Data   : $size, Bucket[size], Data[size]$
   Input  : $vector$
   Output: $seen$
1  num ← 1 ;
2  hash ← hashnum($vector$) ;
3  ← hash % size;
4  while $true$ do
5      for $i \in$ walkTheLineFrom($index$) do
6          if $empty = Bucket[i]$ then
7              if CAS($Bucket[i], empty,$ (hash , $write$))
                then
8                  $Data[i] \leftarrow vector$ ;
9                  $Bucket[i] \leftarrow$ (hash , $done$) ;
10                 return $false$ ;
11             end
12         end
13         if hash $= Bucket[i]$ then
14             while $(-, write) = Bucket[i]$ do ..wait.. ;
15             if $(-, done) = Bucket[i] \wedge Data[i] = vector$
                then
16                 return $true$ ;
17             end
18         end
19     end
20     num ← num +1 ;
21     $index$ ← hashnum($vector$) % size;
22 end
```

Algorithm 1: The find-or-put algorithm of Laarman et al.[3] in pseudocode

In this section all code in algorithm 1 will be referenced by line number using the following notation: (x), whereby x is a line in algorithm 1.

The `find-or-put` function probes continually (4) until either a free `bucket` is found (8-10) or the data is found to be in the hash table (15-17). The `for`-loop (5-19) handles sequential probing behaviour.

The `walkTheLineFrom` (5) returns a set of integers starting from index, with a maximal size of the cache line, making optimal use of fetching behaviour of the cache.

Buckets store memorized hashes and the write status bit of the data in the Data array. A bucket can be in three states: empty, being written and done. A bucket can never become empty again once it has the state being written and can never become being written again once it has the state done.

At line 6 the bucket is checked if it the empty state. If so, it sets the state to being written, writes the vector to the data and sets the bucket to done. This can be seen as locking the data.

Line 14 resembles a spinlock, which means that when it finds that the bucket is in the state of being written, it waits untill the write has been completed.

2.2 GPU Architecture

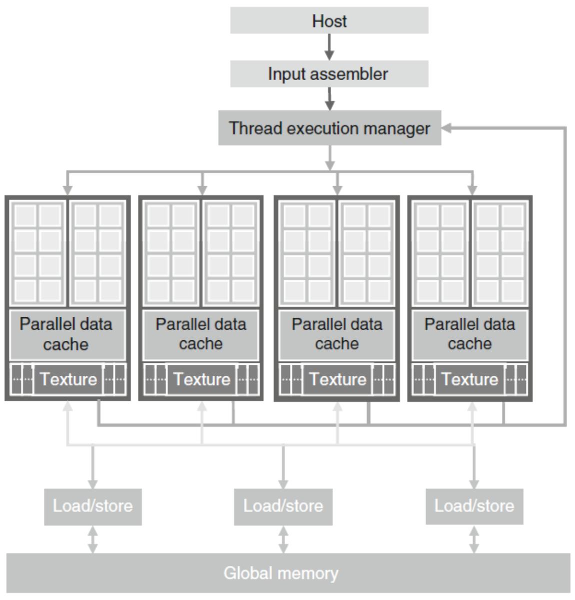

Figure 1. GPU Architecture [2]

GPU architectures differ for each vendor. Here we will explain GPU architecture through the abstraction of the OpenCL framework making use of Figure 1. There is a host, a CPU-based device, which controls multiple compute devices. In our case the host, the CPU, controls the CPU and the GPU. Each of these devices (GPU and CPU) contains multiple compute units. In figure 1 the compute units are the four blocks. In the case of the CPU there are multiple cores, and in the case of GPUs we have workgroups. In a compute unit (workgroup) there are multiple processing units (workers). In figure 1 these are the 16 blocks within the compute units. NVidia calls their workgroups "Warps", while AMD calls their workgroups "Wavefronts"[6].

In the GPU workers share a memory, called local memory. In figure 1 these are the cache blocks in the workgroups. This local memory can only be accessed by the workers in the same workgroup, while global memory can be accessed by all workers. Each worker also has a private memory. Memory hierarchy is as follows, from fastest to slowest, from least shared to most shared and from smallest to largest[4][2].

1. Private Memory
2. Local Memory
3. Global Memory

This is the case for most consumer GPUs

To make optimal use of the parallel features of the GPU, GPUs make use of SPMD (Single Program, Multiple Data) and SIMD (Single Instruction, Multiple Data) techniques. All workgroups run the same program and every worker within a workgroup share a program counter and therefore run the same instruction. This is different from a CPU were threads run different instructions with different data.

2.3 OpenCL 1.2

There are two competing GPU programming languages: CUDA and OpenCL. CUDA is made and only supported by NVidia. CUDA is released in 2006 and is more advanced compared to OpenCL 1.2. Unlike CUDA, OpenCL is cross-vendor, supported by all major GPU manufacturers. As we do not want to be constraint by the vendor of the GPU, we will be using OpenCL.

NVidia has been late in updating there OpenCL 1.2 support, but finally started to update their OpenCL 1.2 support starting this May 2015[1]. Therefore we can safely assume that the majority of the graphic cards will support OpenCL 1.2.

OpenCL 1.2 is a standard developed by the Kronos Group. It provides a top-level abstracting for low level hardware routines. The advantage of the abstracting is that OpenCL code runs from microcontroller to CPUs to General Purpose GPUs.

2.4 GPU Optimalizations

2.4.1 Branch Divergence

Branch divergence occurs in GPUs, because of previously discussed SIMD [2]. Take for example branch in figure 2. If within a local thread group there is at least one thread that goes in the if-branch and at least one thread that goes in the else-branch, the time it takes for the workgroup to complete doubles. As every worker that goes in the else has to wait for the if-branch to finish and every worker that goes in the if-branch has to wait for the else-branch to finish. This means that if/else and other branch divergences should be avoided for optimal performance. nobranch is the improved version of branch for GPUs

```
__kernel void branch(   int a,
                        int b,
                        global int *c) {
    if(a == -1) (*c) -= b;  // a == -1
    else (*c) += b;         // a == 1
}

__kernel void nobranch( int a,
                        int b,
                        global int *c) {
    (*c) -= a * b; // a == 1 || a == -1
}
```

Figure 2. Kernel branch often takes twice as long

2.4.2 Memory Hierarchy

As previously discussed, GPU has a memory hierarchy which if used correctly can increase your speed significantly. For example, if there are a lot of read/write operations between variable A and workgroup G than moving variable A to the local memory of workgroup G increases speed. And if a thread does a lot of operations on a variable within the global or local memory, it is faster to read the variable to the private memory of the thread, do the operations and then write it back than to write each operation directly to the variable.

2.4.3 Atomics and Barriers

An atomic operation is an important part of parallel programming, as atomic operations are done without an interrupt. The memory does not change between the start and the end of the function. In practice this is done through temporary memory locking. Therefore using an excessive atomic operations result in increased waiting time and reduce speed up.

Another functionality are barriers. There are two barriers, a barrier for local and a barrier for local memory. A barrier guarantees that all threads are at the barrier before the threads can move on.

These functions are both useful, but as they are synchronize functions, using often will increase waiting time in threads and therefore reduce performance. Therefore they should be used only when absolutely necessary.

3. RESEARCH QUESTIONS

The goal of the research is to find out which of the three algorithms, Laarman et al., Wijs et al and our design has the best performance. Therefore our main research question is as follows:

- Does Laarman et al., Wijs et al or our design have the best overall performance?

To answer this question it is divided in the following sub questions that we answer for each design.

1. What is the speedup for large number of vectors?
2. What is the speedup for a single vector?
3. How does it perform when the hash table is getting full?

4. RELATED WORK

4.1 Thomas Neele

This research is built upon the research of Neele. The implementations have been built on the implementation delivered in this research. Neele in his research implements the Laarman et al. hash table on GPU. He test the implementation on both CPU and GPU, using the speedup and absolute speed as performance measure. We see that the GPU has good speedup using all the workers. We also see that eventually the GPU absolute performance overtakes the CPU performance. This is argued using two measurements, one which performs inserts and one that performs finds.

Neele concludes that the Laarman design can benefit from GPUs parallel architecture. However this conclusion is made with only the workgroups and one worker. For example in Neele's research the GTX770 was used, which we also used. The graphs of Neele's research makes it apparent there were 128 workers used, while there are 128 workgroups and 32 workers each. In our research we will make use of the 32 workers available.

4.2 GPUExplore by Wijs et al.

The paper of Wijs et al. [7] is important for this paper. Wijs et al. demonstrates that GPU based model checkers are not only possible, it is a necessary evolution for model checkers to increase their speed. They present a model checker entirely for GPU, called GPUExplore. One of the citations it uses is Laarman et al. [3] and the find-or-put algorithm. They also redesigned Laarman et al. version of find_or_put based on the architecture of the GPU. They confirm that it has better performance than cuckoo hashing. However it misses crucial data and experimentation to confirm or deny if the redesign is effective. The redesign of the GPUExplore compared to Laarman et al. original design is expanded in the following section.

5. HASH TABLE DESIGNS

5.1 LTSmin

The design of the LTSmin version has been extensively discussed in the section background and related work. In this section we will discuss the problems with the design when implementing it on GPUs in contrast to CPUs.

In the background we have discussed branch divergence. Laarman et al.'s version experiences this problem on GPU. For example one worker can complete execution at the first position, yet he has to wait for another worker who is still searching. All workers have to wait until the slowest worker is done. This can be avoided by only using a single worker per workgroup, however this also means the GPU is not used optimally.

The design proposed in GPUExplore in Wijs et al.[7] differs from the pseudocode in algorithm 2. It is more in line with how it is implemented in OpenCL, while still conceptual the same.

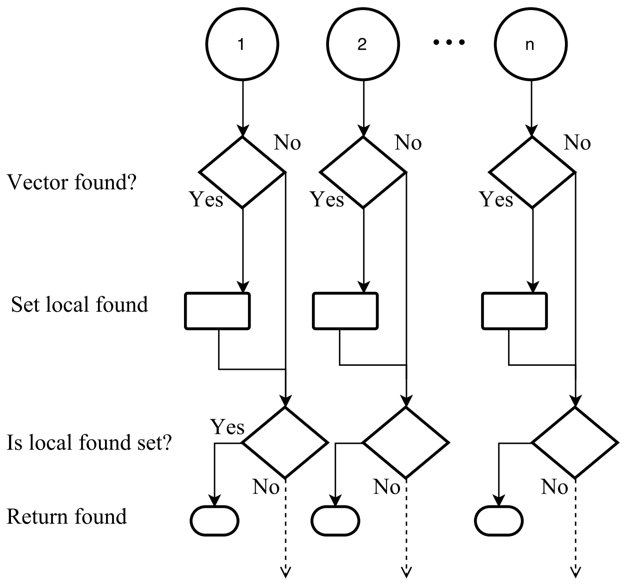

Figure 3. Parallel search found in both GPUExplore and in the proposed design

There are two important changes compared to Laarman et al.. The first change is the addition of a parallel search.

```
   Data   : $size, Bucket[size], Data[size]$
   Input  : $vector$
   Output: $seen$
 1 threadId ← get_local_id(0);
 2 numOfWorkers ← get_local_size(0);
 3 num ← 1 ;
 4 $local$ found ← $false$ ;
 5 $local$ written ← $false$;
 6 while $true$ do
 7     hash ← hash_num($vector$) ;
 8     index ← hash + threadId % size;
 9     for $i$ to cacheLineSize / numOfWorkers do
10         $local$ selected ← numOfWorkers;
11         do
12             if $Bucket[index] = FULL \wedge Data[index] = vector$ then
13                 found ← $true$;
14             if found = $true$ then
15                 return $true$;
16             if threadId = 0 then
17                 for $j$ ← 1 to numOfWorkers do
18                     if CAS(&$Bucket[index], EMPTY,$
19                     $WRITE) = EMPTY$ then
20                         $Data[index] = vector$;
21                         $Bucket[index] = WRITE$;
22                         written ←= $true$;
23             if written then
24                 return $false$;
25         while selected ! = numOfWorkers;
26         index ← index + numOfWorkers % size;
27     num ← num +1;
```

Algorithm 2: The find-or-put algorithm of Wijs et al.[7] in pseudocode. It is an hybrid between our proposed design and the original Laarman et al. design, as it features parallel search and sequential insertion

The parallel search can be found in block 12-15 in algorithm 2. How it works is shown in figure 3. We have n workers and each of these workers checks if they find the given vector. If it is found, the worker sets the local variable "found" to true from false. Afterwards we check if the local found variable is set to true. If this is the case, we know that a worker has found the vector, therefore the workgroup can report it has found the vector. The parallel search is faster than Laarman et al.'s sequential search incase the hash table is not empty. This is because the workers inspect multiple positions in the hash table concurrently.

The second change is the removal of the spinlock, which we would have expected after the if block at line 16-24 in algorithm 2. This increases insert speed, yet gives the possibility of false negatives. False negatives occur when two threads have the same vector and try to insert them concurrently. Both get inserted, while only one should. This results in a small performance decrease, as the model checker is now going to explore this state twice.

5.2 Our Design

The pseudocode of the proposed design is found at algorithm 3. Compared to GPUExplore, we have changed how vectors are inserted. A problem that occurs in the GPUExplore design is that when the workers have not found the vector, and all positions are full, GPUExplore still has to go through all positions in attempt to insert the vector. This is solved in our design by making the insertion parallel. This is found in line 16-23 at algorithm 3.

```
    Data   : $size, Bucket[size], Data[size]$
    Input  : $vector$
    Output: $seen$
 1  threadId ← get_local_id(0);
 2  numOfWorkers ← get_local_size(0);
 3  num ← 1 ;
 4  $local$ found ← $false$ ;
 5  $local$ written ← $false$;
 6  while $true$ do
 7      hash ← hash_num($vector$) ;
 8      index ← hash + threadId % size;
 9      for $i$ to cacheLineSize / numOfWorkers do
10          $local$ selected ← numOfWorkers;
11          do
12              if $Bucket[index] = FULL \wedge Data[index] = vector$ then
13                  found ← $true$;
14              if found = $true$ then
15                  return $true$;
16              if $Bucket[index] = EMPTY$ then
17                  selected ← index;
18              if selected = index ∧ CAS(&$Bucket[index], EMPTY, WRITING$) = $EMPTY$ then
19                  $Data[index] \leftarrow true$ ;
20                  $Bucket[index] \leftarrow FULL$;
21                  written ← $true$;
22              if written then
23                  return $false$;
24          while selected ! = numOfWorkers;
25          index ← index + numOfWorkers % size;
26      num ← num +1;
```

Algorithm 3: Our proposed design in pseudocode. It features parallel searching and insertion

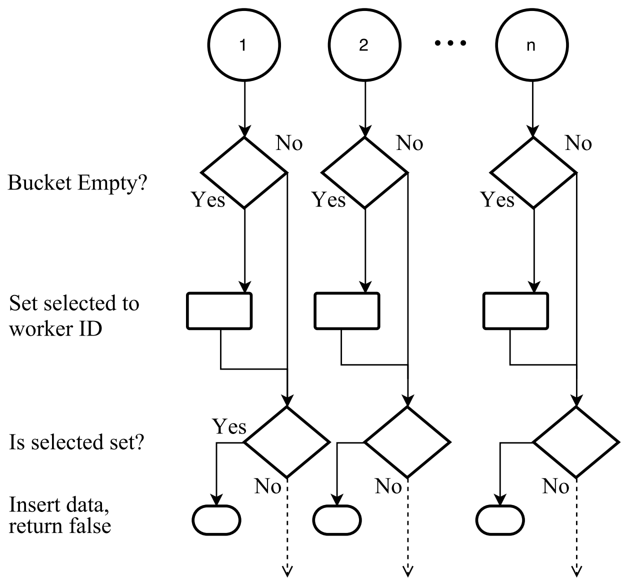

Figure 4. Parallel insert in the proposed design. If there is an empty bucket available, it selects it using by data racing.

The parallel insertion is explained in figure 4. In our design it is essentially the same as parallel searching. First we set a local variable "selected" to the number of workers there are. When a worker has found an empty spot where it is searching, it sets the local "selected" variable to its worker ID. If the "selected" variable is set to a worker id, the worker who has found it tries to write it to the hash table. When this CAS operation succeeds, it sets the local "written" variable. This is so that other workers know that the vector is inserted and the workgroup can finish the execution. However, if the CAS operation fails, the workers come in the while from line 25 in algorithm 3. This is essentially a spinlock. However this does not prevent all false negatives. When the workers didn't find an empty spot they go to the next part of the cache line, shown in algorithm 3 in line 26.

The algorithm does have false negatives. However, we expect to have less false negatives than GPUExplore, because our inserts are faster. This means that there is a smaller window for two vectors to both be inserted concurrently. There is small chance that both vectors are inserted at the same position. This would mean that one workgroup repeats and finds the vector in this iteration, preventing a false negative.

6. IMPLEMENTATION

This section goes into some of the challenges that where encountered using OpenCL. OpenCL is low level, without the abstraction away from the hardware as most popular language have these days. This resulted in some unexpected problems while implementing the design. This section goes into the encountered problems and chosen solutions.

6.1 Intel SDK

While OpenCL has been standardized, it is to the manufacturers to implement it and make it work with their hardware. For this paper the Intel SDK have been used for implementing the designs. These has some specific bugs, for example when working with de HD4000 integrated graphic cards, running driver version 10.18.10.4226 from 25-5-2015, it could not build kernels, while it worked fine on other devices. It appears to be an Intel SDK / driver bug, as it would get stuck in building. We tried to avoid it, but there may be some quirks in the implementations influenced by the Intel SDK.

6.2 Less than 32 bit in OpenCL

Working with OpenCL a problem occurred when using short scalar types. A short type is 16-bit in OpenCL. [4]. When working with an array made of unsigned shorts, problems arise. When using the array indexing operator or using pointer algorithm it would act as if the shorts where 32-bit. Selecting a part of the array would return a pointer twice as far and changing than expected. When changing the value which the pointer points to, it also would change the value indexed one further. These problems can be overcome using some pointer algorithm and bitwise operators, yet for simplicity sake the 16-bit array was quickly replaced by a 32-bit version. In the process also using up more memory.

6.3 Printf Problems

The printf function was heavily used in the debugging of the implementations. However the printf function was harder to use than was initially expected. There were two problems that where encountered. The first is the strictness of the printf. If you do not use the correct flag for a given variable, for example when you do not specify a long when the variable is in fact long, it prints garbage values. The second problem has not been verified, but it appears that when the printf has more than 3 arguments, it will print some of these wrong. This may be a problem with the implementation of OpenCL library that was used.

6.4 Spinlock Struggles

The first spinlock was implemented naive, using a while loop to check the given value, the implementation would even end up not terminating. The thread went into a deadlock. Apparently the bucket was not updated to the local memory. The correct way to implement a spinlock is having a global memory barrier call within the while loop, or using another function that forces the global memory to be updated. Experience proved that printf is one of the functions that forces global memory to be updated.

7. VALIDATION METHOD

Verification of the code would be ideal, but unfortunately it is out of scope for this paper. Therefore a set of requirements has been chosen to get an indication of the correctness of the code. All implementations used in this paper were successfully tested using the methods below.

7.1 Filling It Until It's Full

We use three different tests for checking the code. The first step exists of filling the hashtable with unique vectors, as much as the hashtable can hold, and checking if it always returns false. This verifies that each insertion results in a returned false. If there is a true returned, then insertion went wrong. If the database was full before the all vectors were inserted of was not full, it implies that there are overwritten values or double values. Therefore if this is not the case, it implies that each vector was correctly inserted.

The second step is finding each unique vector we inserted in the hashtable earlier. This should only return true. If not, this implies that either searching went wrong or that inserting went wrong. If it went right, it implies that we can insert a least the used vectors and find it at least ones.

The third step is again finding each unique vector we inserted earlier. This is to verify that finding values does not alter the hashtable.

7.1.1 Workgroups and workers

An important part of GPU programming is that it is corrent concurrently. To verify this we use the above validation method using different configurations of the number of workgroups and workers in each workgroup.

8. MEASUREMENT METHOD

To answer the research questions, we have implemented multiple ways to measure the implementation of the hash table algorithms. There are two states an algorithm can have. The first is that the vector is not yet in the hash table, therefore it will insert the vector. The second state is that vector is already in the hash table, therefore the algorithm finds the vector.

Using these two states we have divided the measurements in three test cases. The first test case is that we only find vectors already in the hash table. The second test case is where we only insert new vectors. Yet this is unrealistic as there will be seldom a case where there are only inserts of finds. Therefore the thirth test case is where we do a combination of both, where half the vectors are inserted and the other half are found.

These test cases have been executed while varying the number of workgroups, workers and how full the initial hash table is.

Table 1: Hardware and software used to gather the measurements

Operating System	Red Hat 4.4.7
GPU	GeForce GTX770
Driver	319.23

8.1 Setup

The setup used to gather data is in table 1.

9. MEASUREMENT RESULTS

All previous mentioned designs have been tested. The Laarman et al. design has been spit into two, one with one worker and the one with 32 workers. These have been split to find out how mutch branch divergence effects the design. The names used in the graphs are as follows:

Laarman et al. - 1 Worker
The Laarman et al. algorithm using a single worker per workgroup

Laarman et al. - 32 Workers
The Laarman et al. algorithm using 32 workers per workgroup

GPUExplore
The GPUExplore algorithm from Wijs et al. using 32 workers per workgroup

Proposed design
The design that we propose in this paper, using 32 workers per workgroup

All the measurements have been exectued on the GTX770, including Laarman et al. which originaly is designed for CPU. Workers, like metioned in the background, are proccessing units in a compute unit (workgroup). For the GTX770 we have found that there are 128 workgroups and each workgroup has 32 workers. When there where made use of multiple workgroups, the number of workers in these workgroups where the same.

9.1 Speedup for multiple vectors

Figure 5 shows the measurements using 1 - 128 workgroups, which is the maximum for GTX770. This graph is based on measurements test case 1, where we only find vectors. This means that all the vectors already in the hash table before we execute the algorithms. The y-axis is the speedup compared to a one workgroup. What we see is that Laarman et al. 1 worker, GPUExplore and the proposed design have linear speedup. The Laarman et al. 32 workers has the best speedup, but slightly flattens at the end.

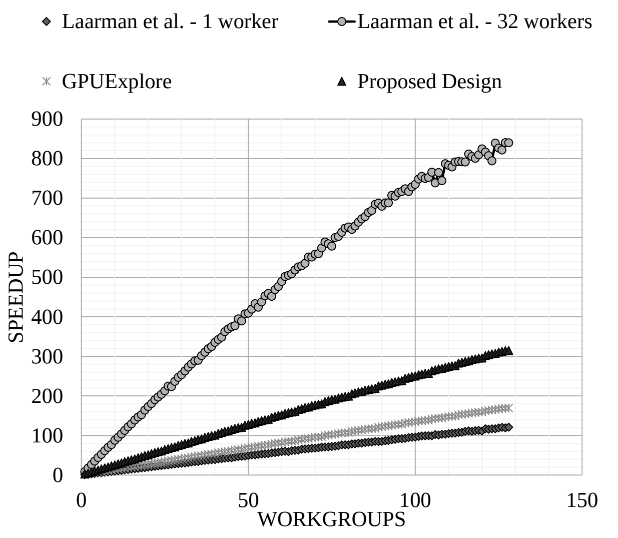

Figure 5. Test case 1 - the speed up of finding vectors compared to a single workgroup

Figure 6 shows the measurements of using 1 - 128 workgroups. This is the graph of test case 2, where vectors are only inserted. The y-axis is the speed up compared to a single workgroup. Here we see that Laarman et al - 1 worker and GPUExplore are both linear and overlap each other. Our proposed design also is linear, but is steeper. Finally Laarman et al. - 32 workers after around 75 workgroups flattens, taking shape of half a parabola.

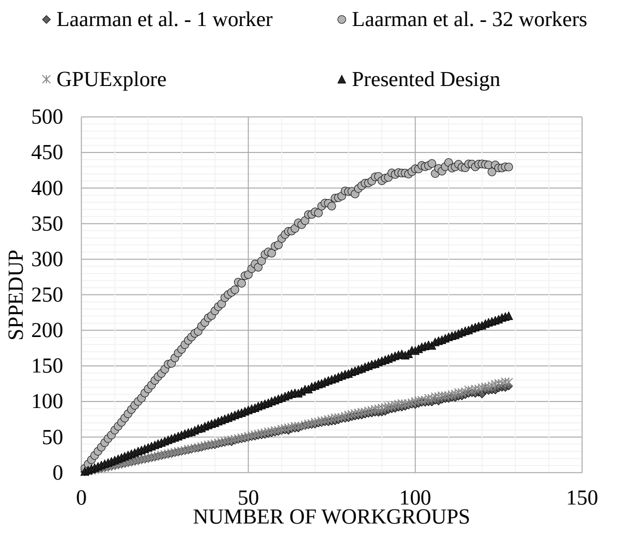

Figure 6. Test case 2 - the speed up of inserting vectors compared to a single workgroup

Figure 7 shows the measurements of using 1 - 128 workgroups. This graph is a graph of test case 3, where vectors are both inserted and found. The y-axis is the speed up based on the execution time of a single workgroup. Where we saw Laarman et al - 32 workers previously had speedups of 400 in test case 2 and even 900 in test case 1, here it doesn't reach a speed up of 50 and flattening quickly. Moreover, where GPUExplore had both a steeper speed up in test case 1 and 2 than Laarman et al. - 1 worker, here it is less steep.

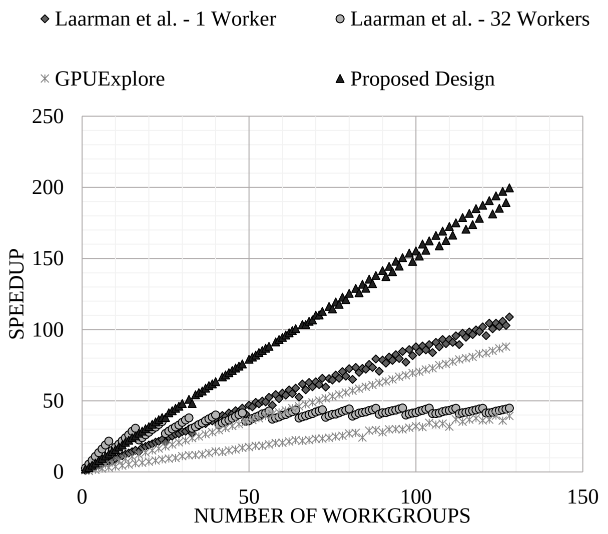

Figure 7. Test case 3 - the speed up of inserting and finding vectors compared to a single workgroup

9.2 Speedup Single Vector

Figure 8 shows the speed up of a single vector in test case 3. The speedup is compared to the speed of one workgroup with one worker. Here we see that the speed up of a single vector is reduced per worker increases, but unlike GPUExplore and our proposed design, which each worker a extra vector is able to be processed. We see that a single vector is processed faster with the first 8 workers with GPUExplore and our proposed design.

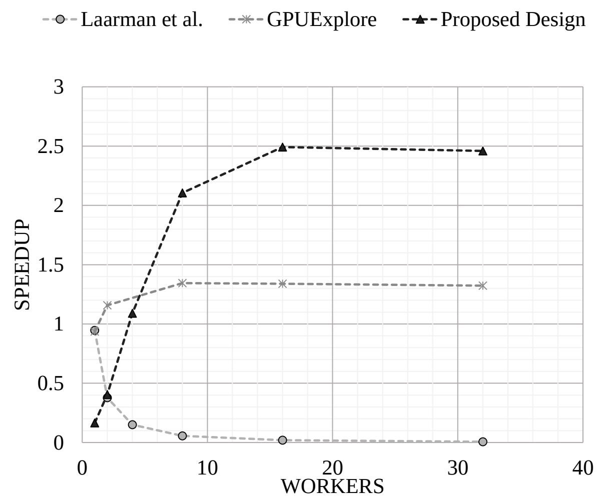

Figure 8. Test case 3 - The speed up for a single vector compared to a single workgroup with a single worker

9.3 Hash table size differents speedup

While in theory we often assume hash tables have a constant execution rate, in practice they do not. The graph has this ideal line of a constant execution speed. In Figure 9 we show the speedup of the designs when they are used in a database that is filled. The speedup is based on the performance of an empty database. We see our proposed design has an almost constant speedup until the database is half. Laarman et al - 32 Workers has the worst performance which execution time almost dubbles when the hash table is one tenth filled.

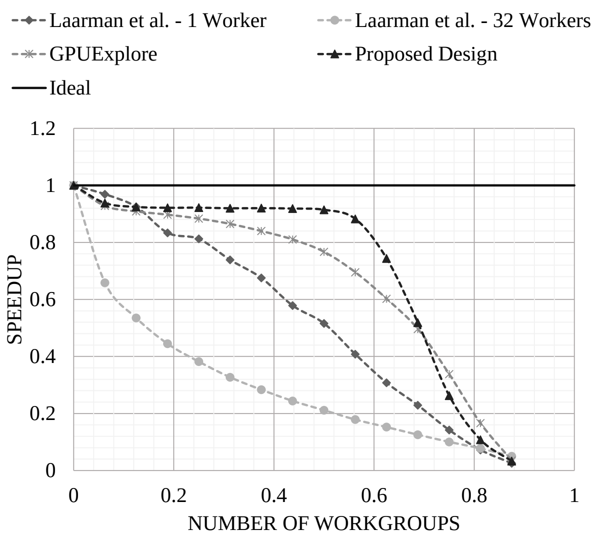

Figure 9. The measured speedup of the three algorithms inserting vectors and finding them at the same time.

10. DISCUSSION

10.1 Laarman et al. - 32 Workers

Laarman et al. with 32 workers uses the GPU optimal, which in this case is 128 workgroups with each 32 workers. In this case we have the best finding and inserting performance. However, in the context of figure 8 this becomes clear that if we have a low number of vectors instead of batches of 128*32 vectors, the slowdown is significant. What we see with 32 workers is that it is possible for single vector to actually take 5 times as long, which would not be the case if we only had one worker per workgroup. This is caused by branch divergence in the workers.

This branch divergence problem with Laarman et al. becomes obvious in test case 3, as shown in fig 7, where it had the worst speed up.

10.2 Laarman et al. - 1 Worker

The two problems, reduced single vector performance and bad performance while searching and inserting, which we have with 32 workers is solved by only using a single worker in each workgroup. This also means we only use 1/32 of the GPUs potential. This version was tested by Neele as mentioned in the related works section. Neele found a speedup of 0.43 per workgroup on the GTX770. We however found an average speedup of 0.89 per workgroup, and a speedup of 0.85 per workgroup when using all the workgroups. Even though we don't use the GPU optimal, the speedup and performance are still good.

In Figure 9 we find that Laarman et al. design does not have a constant execution time. It gets around 0.13 times slower for each 10 percent that the database is filled.

10.3 Wijs et al.

The first that stands out is that Wijs et al. has a worse speed up in test case 1 our design, as shown in figure 5. As they both make use of parallel searching, we would expect the same speedup. This is because when Wijs et al. has not found the vector in the part where the workers are searching, while this part is filled, it has to go through the insertion loop before it can search further. This explains why it performs better than Laarman et al. but worse than our design. Wijs et al. performance slightly better than the Laarman et al.. We find in figure 9 that GPUExplore performance near constant in execution time, with only a slight steep down. Test case 2, as shown in figure 6, speedup is equal to Laarman et al. - 1 worker, as expected.

Wijs et al. has overall better performance than Laarman et al - 1 Worker, yet an unexpected bad performance when we are both inserting and finding. In conclusion the GPU-Explore algorithm is slightly better than Laarman et al on GPU.

10.4 Our Design

It is clear that the proposed design has the best performance overall. What we find in figure 9 is that our proposed design has an almost constant execution time for the database, from empty until it is half full. The speedup for all the three test cases is the steepest. However it needs around 16 workers to scale well, as shown in figure 8. Compared to a single worker in a single workgroup, a vector would be 6 times slower inserted with the maximum workgroups.

11. CONCLUSION

In conclusion our design performance is the best among the three designs. It overtakes both finding and inserting by making efficient use of the workers that are available. With these workers we get a speedup of around 1.6 per workgroup. Unlike Laarman et al. which does not scale using multiple workers as a consequence of branch divergence.

Our design scales when we are inserting, unlike GPUExplore. Insertion takes longer than finding a vector, because it needs to know that the previous positions where full, followed by a CAS operation. That in combination with the expected lower number of false negatives, we can conclude our proposed design performance better.

We have an indication that our design could increase the performance of model checkers, however we cannot conclude it. Like mentioned in the section on the designs, the proposed design has some false negatives, which impacts the performance negative. To conclude the proposed design performance better we need to implement it in a model checker.

Even through the scaling is promising, there is still a problem of memory. From Neele: "Even the most expensive GPUs have a memory size of only 12 GB, whereas a high end server may have more than 100 GB of main memory. The potential of a model checker largely depends on the amount of memory it can allocate."

12. FURTHUR WORK

Unfortunately we have not had the time to implement all ideas for the design. The original idea of the algorithm is splitting it in the local memory of the workgroups. Where the vector was to be placed is decided by the hash. The hash can be calculated by a different workgroup, which can do 32 at once because of the workers. In theory this should increase the performance even above the proposed design in this paper. Further work could try to implement these features in the algorithm.

Further research could be performed using the algorithm proposed in this paper and implement in a model checker. For example it could be tested in LTSmin. To evaluate if using GPUs for the hash table is useful and does increase performance.

As last I would like to question if the hash table is the best part of the model checker for parallel computing. A hash table makes use of a hash function with low collision to find a random position that has a good chance to be unique. This results in fast finding and inserting. This means that until the hash table is getting full, the workers are often in a position where they are doing nothing. What we suggest is looking at another storage system, like a tree, where parallel computing has the potential to scale optimal with the number of workers.

13. REFERENCES

[1] Khronos products. `https://www.khronos.org/conformance/adopters/conformant-products`. Accessed: 2015-10-7.

[2] D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach.* Newnes, 2012.

[3] A. Laarman, J. Van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 247–256. FMCAD Inc, 2010.

[4] L. H. Munshi and Aaftab. The opencl specification.

[5] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[6] J. Tompson and K. Schlachter. An introduction to the opencl programming model. *Person Education*, 2012.

[7] A. Wijs and D. Bošnački. Gpuexplore: Many-core on-the-fly state space exploration using gpus. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 233–247. Springer, 2014.