Lock-based Data Structures on GPUs with Independent Thread Scheduling

vorgelegt von

Phillip Grote

Matrikelnummer: 371570

dem

Fachgebiet Datenbanksysteme und Informationsmanagement
der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin

Bachelor Thesis

February 3, 2020

Gutachter:
Prof. Dr. Volker Markl

Betreuer:
Clemens Lutz

Zeitraum:
September 2019 - Februar 2020

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Die selbständige und eigenhändige Anfertigung versichert an Eides statt:

The independent and unaided completion of the thesis is affirmed by affidavit:

Berlin, den 02.02.2020 Phillip Grote

Abstract

GPUs support thousands of concurrent threads and are able to accelerate throughput oriented workloads by an order of magnitude. In order to implement highly concurrent data structures, efficient synchronization is necessary. However, correct and efficient synchronization requires knowledge of the underlying hardware. Especially since the handling of control flow on GPUs can lead to livelock conditions, which prevent the system from making forward progress.

In this thesis we investigate efficient synchronization techniques on GPUs with Independent Thread Scheduling. By exploring how the hardware handles control flow, we are able to show that Independent Thread Scheduling prevents SIMT deadlocks from happening. However, we identify situations in which livelocks occur on GPUs with Independent Thread Scheduling. To address this problem, we present a technique that prevents such livelocks from occurring.

By evaluating the performance of different synchronization techniques, we are able to demonstrate that choosing a different synchronization technique could improve performance by a factor of 3.4. We show that the performance of a given synchronization technique depends on the workload and no synchronization technique outperforms the others in every situation. Finally, we are comparing fine-grained and coarse-grained locking and give advice on when to choose which granularity.

Zusammenfassung

GPUs unterstützen tausende von nebenläufigen Threads und sind in der Lage Aufgaben, die einen hohen Datendurchsatz benötigen, um mehrere Größenordnungen zu beschleunigen. Um Datenstrukturen, die nebeläufigen Zugriff ermöglichen, zu implementieren, werden effiziente Synchronisationstechniken benötigt. Eine effektive und korrekte Umsetzung von Synchronization macht umfangreiche Kenntnisse der verwendeten Hardware erforderlich. Insbesondere ist zu beachten, dass Programme, die auf der GPU ausgeführt werden, einen Zustand (Livelock) erreichen können, in dem ein weiterer Fortschritt des Programms verhindert wird.

In dieser Bachelorarbeit untersuchen wir effiziente Synchronisationstechniken unter der Verwendung einer GPU mit Independent Thread Scheduling. Wir demonstrieren, wie die Hardware Kontrollfluss umsetzt und können bestätigen, dass Independent Thread Scheduling SIMT deadlocks verhindert. Jedoch können wir andere Situationen identifizieren, in denen ein Livelock eintritt, auch wenn eine GPU mit Independent Thread Scheduling verwendet wird. Um dieses Problem zu beheben, stellen wir eine Technik vor, mit der entsprechende Livelocks verhindert werden können.

Durch Evaluierung verschiedener Synchronisationstechniken sind wir in der Lage zu zeigen, dass mit der Auswahl einer für die Situation angemessenen Synchronisationstechnik, die Leistung um das 3,4 fache verbessert werden kann. Im Allgemeinen hängt die Leistungsfähigkeit der Synchronisationstechnik von der jeweiligen Aufgabe ab. Wir zeigen, dass keine Synchronisationstechnik in allen Situationen besser als alle anderen ist. Schließlich untersuchen wir die Granularität von Locks und geben an, in welchen Situation welche Granularität gewählt werden sollte.

Contents

1 Introduction

Concurrent access to the same data structure exposes its internal consistency. Therefore, it becomes necessary to implement coordination among concurrent threads. This coordination among threads is called synchronization. But it does not suffice to just ensure correctness by protecting the consistency of a given data structure. Moreover, synchronization needs to be efficient. Otherwise, the performance impact of synchronization would cancel out the overall purpose of increasing concurrent access. And with thousands of concurrent threads on a modern *Graphical Processing Unit* (GPU), efficient synchronization is especially challenging.

In this chapter we briefly explain why it has become necessary to scale computer systems (e.g. database systems) to many-core architectures and we state why we chose the GPU as a target architecture for our investigation. Next, we point out why efficient synchronization is required. Finally, we present our contributions and outline the remaining parts of this work.

1.1 Many-Core Hardware Architectures and the GPU

In the past, the exponentially increasing performance of computing systems was achieved by a combination of reducing transistor sizes and improvements in hardware architecture, compiler technology and algorithms [1]. Half of those gains were achieved by higher clock frequencies [17]. However, power density constraints finally resulted in stagnating clock frequencies. As a consequence, we observed a general interest in multi-threaded programming [32] and in finding more efficient hardware architectures [1]. Although it is possible to exploit hardware specialization in order to achieve higher efficiency, an ideal architecture should also support a wide range of programs [1]. GPUs are one example for such a specialized hardware architecture, which differs greatly from the traditional architecture implemented on *Central Processing Units* (CPUs). While CPUs optimize latency, GPUs are focusing on throughput. Figure 1.1 illustrates the different design philosophies of CPUs and GPUs. CPUs are designed for optimizing sequential code. Therefore, they spend a lot of their chip area on sophisticated control logic and large caches [19]. In contrast, GPUs spend the vast majority of their chip area on process-

ing elements [19]. As a result, many throughput oriented workloads could achieve substantial speed-ups when implemented on a GPU [20][24].

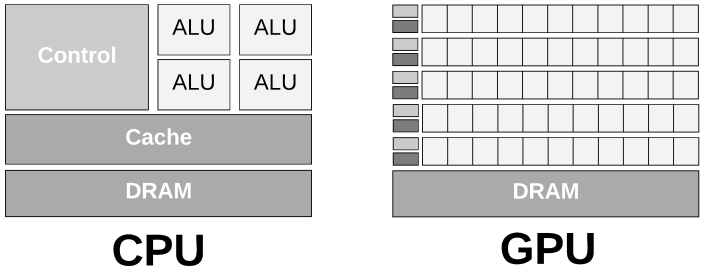

Figure 1.1: Different Design Philosophies on CPUs and GPUs [19].

In order to obtain the desired flexibility needed to support a wide range of applications, the GPU has transitioned from a fixed-function graphic pipeline to a programmable parallel processor which supports a Turing Complete programming model [1][22]. One recent hardware feature, which underscores this development, is *Independent Thread Scheduling* (ITS) . ITS was introduced with the Volta microarchitecture [26]. It changes how threads are scheduled and increases programmability.

The ever-increasing flexibility of GPUs enabled the acceleration of non-graphics applications. Today, accelerating read-heavy *Online Analytical Processing* (OLAP) workloads is a highly active research field [4][8][9][16]. But other parts of a database system could also benefit from the computing power of modern GPUs. He and Yu have shown that even write-heavy *Online Transactional Processing* (OLTP) workloads could benefit from GPUs [16]. Finally, the growing interest in supporting dynamic (mutable) data structures, that are built, queried and updated on the GPU [5][6], indicates a more general applicability of GPUs for database systems.

1.2 Efficient Synchronization

One major impediment in scaling database systems to highly concurrent architectures such as GPUs is synchronization [10][21]. The capability of modern GPUs to support thousands of concurrent threads contributes to the importance of efficient synchronization. Coordination among threads imposes an additional coordination overhead. But it is of crucial importance that this overhead does not cancel out the performance improvements gained by allowing concurrent access. Thus, synchronization needs to be efficient.

Furthermore, correct and efficient synchronization depends highly on the underlying hard-

ware. In order to accelerate database systems with GPUs we need to understand how synchronization behaves on this particular architecture. The same reasoning applies for implementing mutable data structures on such a highly concurrent architecture.

1.3 Contributions

In this thesis, we investigate common low-level synchronization primitives on GPUs with ITS. Thereby, we explore how the hardware handles control flow. This knowledge helps us to implement different synchronization primitives on the GPU. Finally, we investigate the presented lock variations in respect to their scalability. In summary, our contributions are:

Livelock. Although Independent Thread Scheduling prevents SIMT deadlocks, we identified a new livelock condition.

Architectural Details. We investigate the architectural details of GPUs and how they affect the implementation of common synchronization techniques. In particular, how to prevent the aforementioned livelock condition

Study of Synchronization. Study of different synchronization primitives on GPUs with ITS. Our Investigation explains why certain synchronization performs better than others on GPUs. Furthermore, our result can be used to assess the cost of synchronization and help to identify suitable synchronization primitives.

1.4 Outline

First, in **chapter** 2 we separate the terms locking and latching. Furthermore, we summarize the formal requirements of synchronization and present the most important architectural details of modern GPUs which directly affect the implementation of synchronization primitives.

In **chapter** 3 we investigate ITS. We show that ITS avoids SIMT deadlocks. However, we also present a scenario in which a livelock condition could occur despite of ITS. Finally, we demonstrate how such a livelock could be prevented.

The synchronization primitives we are investigating and their implementation on GPUs are shown in **chapter** 4. We differentiate between centralized spin locks and queue based locks.

In **chapter** 5 we evaluate the previously presented synchronization primitives with three distinct benchmarks. In the Counter-Benchmark concurrent threads compete to increment a single counter object. Next, we investigate a more realistic workload: building a hash table. Finally, we use a sorted list as a case study to evaluate our lock variants with lock coupling and compare fine-grained locking with coarse-grained locking.

We present related work in **chapter** 6.

We summarize the results of this thesis in **chapter** 7.

2 Background

In the previous chapter we pointed out why efficient synchronization is important and why we are investigating synchronization on GPUs with ITS. In this chapter we summarize the challenges that need to be addressed in order to implement efficient synchronization on GPUs. The following chapters will build on the presented insights.

We structured this chapter as follows: First, we give a precise definition of the term locking and how we are using the term throughout this work. Next, we present the requirements of a correct implementation of synchronization. Finally, we summarize the most important architectural details of modern GPUs.

2.1 Defining Locking

Locking means different things to two different research communities [15]: On the one hand, it could mean high-level concurrency control. And on the other hand, locking could mean low-level data structure synchronization. In contrast to high-level concurrency control, which protects the database content by separating transactions, low-level data structure synchronization protects the data structure by separating multiple concurrent threads. The mechanisms to achieve high-level concurrency control and the mechanisms for low-level data structure synchronization are different. The literature on operating systems and programming environments usually uses the term lock for a mechanism that accomplishes low-level data structure synchronization. In order to distinguish between the two mentioned purposes the database research community uses two different terms. It is common to use latch in the sense of low-level synchronization and lock as a mechanism for high-level concurrency control. Nevertheless, similar to Leis et al. [21] we are using the term lock instead of latch, since we focus on low-level data structure synchronization and hence have no need to differentiate between low-level synchronization and high-level concurrency control.

2.2 Synchronization and its Requirements

The purpose of synchronization is to prevent all incorrect interleavings of instructions. This can be achieved by ensuring that a specified sequence of instructions appears to execute as a single, indivisible unit. The simplest method to implement this notion of atomicity is to force different threads to execute their operations one at a time (mutual exclusion) [30].

In this section we formally define linearizability and explain how this definition relates to correct synchronization. Furthermore, we provide a brief introduction to safety and liveness; two requirements for correct synchronization.

2.2.1 Linearizability

Linearizability is a consistency condition, and as such it formalizes the correctness of a given execution. Moreover, linearizability of a system as a whole depends only on the linearizability of its parts [28][30]. In order to define linearizability we introduce some terms in accordance with Raynal [28]:

Processes and Operations. Given a finite set of processes $P := \{p_1, ..., p_n\}$. Each $p_i \in P$ is accessing concurrent objects by executing operations on them. The execution of an operation is modeled by its invocation and its response event. Let $p_i \in P$, the invocation of an operation *op* with parameters *arg* on an object X by process p_i, is denoted with: $inv[X.op(arg)\ by\ p_i]$. The response with return value *ret* is denoted with: $resp[X.op() \rightarrow ret\ by\ p_i]$. Depending on the context, it is possible to omit the name of the object as well as the name of the invoking process. Each process is assumed to be sequential: it executes one operation at a time.

Definition 1. *Let $P := \{p_1, ..., p_n\}$ be a finite set of processes:*

1. *A **history** $\hat{H} = (H, <_H)$ is a sequence of invocation and reply events, where H is a finite set of events generated by $p_1, ..., p_n$ and $<_H$ is a total order on those events.*

2. *The **local** history of p_i ($p_i \in P$), denoted with $\hat{H}|p_i$, is the sub-sequence of $\hat{H}$ containing only those events generated by p_i.*

3. *Two histories $\hat{H}$ and $\hat{H}'$ are said to be **equivalent** if they have the same local histories: for each $p_i \in P$, $\hat{H}|p_i = \hat{H}'|p_i$.*

A history $\hat{H}$ induces an irreflexive partial order on its operations:

Definition 2. *Let* op *and* op′ *be two operations and* $\hat{H} = (H, <_H)$ *be a history.*

1. *we define the* ***partial order*** $\rightarrow_H$*:*

$$(op \rightarrow_H op') := (resp[op] <_H inv[op'])$$

2. *Two operations* op *and* op′ *are said to be* ***concurrent*** *if neither*

$$resp[op] <_H inv[op']$$
$$nor$$
$$resp[op'] <_H inv[op]$$

Sequential specification. Objects are defined by a sequential specification which states how the objects must behave when accessed sequentially. Typically, each operation on a given object is therefore associated with preconditions and postconditions. Furthermore, each operation must preserve certain invariants.

Definition 3. *Given a finite set of processes* $P := \{p_1, ..., p_n\}$

1. *A history is* ***sequential*** *if it has no concurrent operations.*

2. *A sequential history* $\hat{S}$ *is* ***legal*** *if, for each object X, the sub-sequence of* $\hat{S}$ *that consists only of events involving X is compliant with the sequential specification of X.*

After we have introduced some important terms in Definition 1-3 we are now able to give a formal definition of linearizability:

Definition 4. *A history* $\hat{H}$ *is* ***linearizable*** *if there is a history* $\hat{S}$ *such that:*

1. $\hat{H}$ *and* $\hat{S}$ *are equivalent*

2. $\hat{S}$ *is sequential and legal*

3. $\rightarrow_H \subseteq \rightarrow_S$

Each execution of a concurrent program results in a specific history. We can check if a given history is linearizable in accordance with the given definition. But correct synchronization needs to ensure that all histories that can be generated are linearizable. In the next two sections we explain how this can be achieved.

2.2.2 Safety

A sequential implementation is safe, if each operation on an object X is compliant with the sequential specification of X [30]. If we want to allow concurrent access on the same object, we have to extend the implementation to allow concurrent operation calls. Safety requires atomicity and deadlock freedom.

Atomicity. Each operation should appear to occur atomically [30]. Only if the methods of an object appear to occur atomically, it is properly synchronized. In this sense synchronization means to achieve the appearance of a total order on high-level operations. Furthermore, the ordering must be consistent with the sequential program of each participating process. If such a total order exists, each operation occurred atomically [30]. Linearizability is one way to formalize atomicity. Given a history $\hat{H}$ of an execution, we are able to evaluate if $\hat{H}$ is linearizable, but the definition of linearizability does not formulate how to prevent non-linearizable executions.

One method to actually achieve linearizability is by using a single global lock. All participating threads[1] need to acquire the lock before they can invoke operations on any shared object. After a thread has completed the critical section it releases the lock and another thread can proceed. With this protocol only sequential histories can be produced. We remind ourselves that a sequential history has per definition no concurrent operations (see Definition 3). We call such a locking scheme coarse-grained locking. But it is possible to increase the potential concurrency, or to allow concurrency at all, by adopting fine-grained locking. For instance, it can be achieved by protecting each list element with a separate lock (fine-grained locking) instead of protecting the whole list with a single lock (coarse-grained locking). It has been shown that linearizability can be ensured on a concurrent sorted list by a fine-grained locking protocol in which each thread holds at most two locks at a time [7]. This protocol is known as lock coupling. First, a thread needs to acquire the lock which is associated with the first element of the list (*head*). After acquiring the first lock a traversing thread proceeds as follows:

1. Acquire the lock of the successor
2. Release the lock of the current node
3. Successor node becomes current node
4. Repeat if necessary

This protocol ensures that a thread is never overtaken by another thread during its traversal

[1] We are describing how correct synchronization is actually achieved and therefore we are not talking about abstract processes anymore.

of the list. Similar locking protocols are commonly used in concurrent trees and other pointer based data structures [30].

Deadlock. One important safety property on concurrent objects is deadlock freedom. Especially with fine-grained locking it becomes necessary to ensure deadlock freedom. As an example, let us assume, we have a locking protocol similar to the above mentioned lock coupling. If we allow threads to traverse a doubly-linked list from both ends, a deadlock might occur. If we assume the following situation:

- Thread *A* and thread *B* are holding a lock on neighbouring nodes
- *A* is traversing the list from *head* to *tail*
- *B* is traversing the list from *tail* to *head*

In the given situation each thread will only release its lock, if it succeeds in acquiring the lock currently held by the other one. As a consequence a deadlock prevents both threads from continuing their execution.

In general, a deadlock occurs when four conditions are simultaneously met [30]:

1. **Exclusive use.** Threads access non-shareable resources (e.g. a node within a list).
2. **Hold and wait.** Threads wait for unavailable resources while continuing to hold resources they have already acquired.
3. **Irrevocability.** It is not possible to revoke the access to a resource which was already granted.
4. **Circularity.** A circular chain of threads in which each thread is holding a resource needed by a previous one. For instance, thread A wants to acquire a lock held by thread B, while thread B is trying to acquire the lock held by thread A.

A common strategy to prevent deadlocks is to break the circularity condition by imposing a static order on locks and by requiring that every operation acquires its locks according to that static order [30].

2.2.3 Liveness

Liveness ensures that each operation will finish eventually. In other words, liveness guarantees forward progress. Forward progress is related to the following definition:

Definition 5. *According to Scott [30] an operation is said to be **lock free** if some thread is guaranteed to make progress: the operation is completed in some bounded number of steps*

We say a livelock occurs, if the systems as a whole is prevented from making progress. If a operation is not lock free, a livelock might occur.

Lock-based algorithms, the topic of this investigation, are inherently blocking. Each thread which tries to acquire an already taken lock is unable to proceed and has to wait until the lock is released [30]. Livelock freedom requires that each thread that acquires a lock releases this particular lock eventually. Therefore we must ensure that each critical section is free of infinite loops. However, on GPUs this is not sufficient. As we describe in chapter 2.3, livelock freedom is linked to how the hardware handles control flow.

2.3 GPU Hardware Architecture

In the previous section we defined linearizability and described how to synchronize concurrent accesses to shared objects. In this section we summarize some particularly important architectural details of GPUs. They directly affect the previously defined correctness and the performance of synchronization. We start with a general overview of the hardware and programming model. Next, we describe how control flow is handled on GPUs. Finally, we summarize issues related to cache coherence and memory consistency and present the available atomic instructions used to implement synchronization.

2.3.1 General Overview

The programming model organizes the threads which are executing a program on the GPU hierarchically. Multiple threads are grouped together in a thread-block and multiple threads-blocks compose a grid. When invoking a GPU program (kernel), the programmer specifies the number of threads per thread-block (block size) and the number of thread-blocks per grid (grid size).

GPUs are subdivided in several *Streaming Multiprocessors* (SMs) shown in Figure 2.1. All threads within the same thread-block are mapped onto the same SM. Threads of the same thread-block have access to a manually managed cache called shared memory which is associated with the SM. All threads have access to the same global memory.

All threads within a thread-block are grouped by the hardware in warps. Such a warp typically consists of 32 threads. The hardware manages, schedules and executes warps and not individual threads. Threads within a warp share a common instruction fetch and instruction decode unit (front-end). The execution unit is split in multiple lanes (back-end): each lane corresponds to a single thread. In effect, this hardware architecture allows individual threads to execute a single instruction on multiple data points (*SIMD*). In contrast to SIMD instructions on CPUs, the GPU hardware abstracts away the underlying SIMD back-end. Therefore, NVIDIA labels the hardware architecture as Single Instruction Multiple Thread (*SIMT*) architecture [25]. NVIDIA claims that for "[...] the purposes of correctness, the programmer can essentially ignore the SIMT behavior [...]" [25]. But as we will point out in chapter 3, current GPUs cannot live up to this claim, even after Independent Thread Scheduling has been introduced.

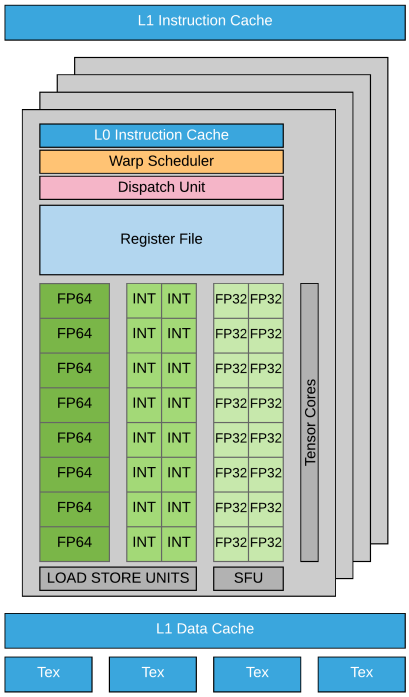

Figure 2.1: Streaming Multiprocessor [26].

2.3.2 Arbitrary Control Flow

The programming model on GPUs abstracts away the underlying SIMD hardware. This becomes especially apparent if we consider how control flow is handled by the underlying hardware. We present two common approaches: stack-based and stack-less reconvergence.

2.3.2.1 Stack-based Branch Reconvergence

On pre-volta architectures the SIMT model is implemented by stack-based masking of execution [12]. This allows different threads within a warp to take different control flow paths (thread divergence). This flexibility is achieved by serializing the execution while masking the inactive threads. Masking inactive threads reduces the SIMD utilization and potentially degrades performance. But SIMD utilization is regained by forcing divergent threads to reconverge as soon as possible [12]. A limitation of this implementation is that it creates implicit scheduling constraints for diverging threads: threads that need to communicate must be mapped to different warps. If the programmer fails to consider this limitation, deadlocks will occur.

Listing 2.1 shows how mutual exclusion could be achieved by a simple spin lock. On GPUs with stack-based execution masking this code would cause a deadlock if threads from the same warp are competing against each other to enter the critical section. This deadlock occurs if thread *A* and thread *B* are grouped together into the same warp and are trying to execute the critical section. The *atomicCAS* instruction on line 1 ensures that only one thread is able to acquire the lock. If thread *A* acquires the lock, the hardware will force thread *A* to wait on line 2 for thread *B* in order to improve SIMD utilization. As a result thread *B* will spin on line 1 forever since thread *A* is not able to execute the unlock operation on line 7. Such a case where forward progress of a diverged thread is prevented is known as SIMT deadlock[2] [12].

[2] In this situation the system as a whole is prevented from making progress: a livelock condition has been reached. Nevertheless, we decided to adopt the already established terminology here.

Listing 2.1: Deadlock due to diverging threads.

```
1 while(atomicCAS(&(data->lock),0,1) != 0);
2 __threadfence();
3
4 //critical section
5
6 __threadfence();
7 atomicExch(&(data->lock),0);
```

2.3.2.2 Stack-less Branch Reconvergence

NVIDIA addressed the possibility of a SIMT deadlock by changing the implementation of how control flow is handled by the hardware. Beginning with the Volta microarchitecture, all NVIDIA GPUs have a new hardware feature called Independent Thread Scheduling, which is an integral part of our work. In this section we summarize the description of this feature [2][11][26]:

As a first step, the stack is replaced by convergence barriers. Each warp maintains various data fields to manage the execution of arbitrary control flow. For instance, the Barrier Participation Mask is used to track which threads within a warp are participating in a given convergence barrier. In order to allow nested control flow, there may be more than one barrier participation mask for any given warp. During execution, threads with the same convergence barrier will wait for each other. To achieve this, the Barrier State tracks which threads have arrived at a given convergence barrier. Thread divergence may occur when the warp is executing a branch instruction. If this happens, the scheduler will select a warp-split (subset of threads with a common program counter) for execution. The main advantage of the convergence barrier implementation over a stack-based implementation is that the scheduler is in principle free to switch between different warp-splits. This enables forward progress even in situations which would lead to a SIMT deadlock on a stack-based implementation.

2.3.3 Cache Coherence

On a shared-memory parallel system, data in upper levels of the memory hierarchy may not be consistent with lower levels. A cache-coherent system is one in which changes to data are guaranteed to become visible to all threads and changes to the same location are seen in the

same order by all threads [30].

Each SM has it own L1 cache which is unified with shared memory and all SMs share the same L2 cache. But on NVIDIA GPUs writes to global memory are not kept coherent per default [27].

In order to implement correct synchronization we need to either disable the L1 completely [31] or annotate individual memory accesses with cache operators via inline assembly. In our study we use both approaches: disabling of the L1 cache in chapter 3 and annotating of individual accesses in chapter 4.

On CPUs, some synchronization primitives perform better than others because they are avoiding frequent cache invalidations. Since GPUs do not implement cache coherence this advantage does not apply. Instead, we need to investigate how different synchronization primitives behave and explain why.

2.3.4 Memory Consistency

GPUs [3] and other modern multicore-processors [17] implement a relaxed memory model. This means, that multiple memory operations are not sequentially consistent: Accesses by different threads, or to different locations by the same threads, occur out-of-order from the perspective of threads on other cores. When consistency is required, special synchronization instructions are necessary to ensure a specific ordering of memory operations. Inserting synchronization instructions is difficult. For instance Alglave et al. [3] found missing fences in a variety of peer-reviewed publications and even vendor guides.

To illustrate the issue, we adapted Figure 2.2 from Scott [30]. If we assume that x and y are initially set to *zero*, it is possible on a machine with a *relaxed* memory model that both i and j will be set to *zero*. We observe such a behaviour if the writes to x and y are delayed. In this case both threads read a *zero*. We say the reads bypass the writes. It appears that the second instruction of each thread is executed before the first of the other threads. This ordering combined with the program order results in the ordering loop shown in Figure 2.2. As a result, the state held in memory is inconsistent.

2.3.5 Atomic Instructions

Atomic instructions are one of the main building blocks for implementing synchronization algorithms. These instructions are able to read and modify a memory location in a single atomic

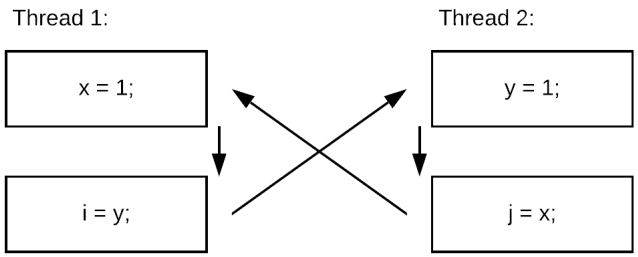

Figure 2.2: Ordering Loop.

operation. The signatures of the most important atomic instructions for our investigation are shown in Listing 2.2.

Listing 2.2: Atomic Instructions.

```
1 int atomicAdd(int* address, int val);
2
3 int atomicExch(int* address, int val);
4
5 int atomicCAS(int* address, int compare, int val);
```

The semantics of these instructions are described in the CUDA Programming Guide [25]:

- **atomicAdd:** Reads an integer *old* at memory location *address*, computes $old + val$ and stores the result at the same memory location. These operations are performed in one atomic transaction. The return value of *atomicAdd* is *old*.
- **atomicExch:** Reads an integer *old* at memory location *address*, and stores *val* at the same memory location. These operations are performed in one atomic transaction. The return value of *atomicExch* is *old*.
- **atomicCAS:** Reads an integer *old* at memory location *address*. If *old* equals *compare*, val will be stored at the same memory location. These operations are performed in one atomic transaction. The return value of *atomicCAS* is *old*.

3 Independent Thread Scheduling

In the previous chapter we presented how a GPU manages control flow and why SIMT deadlocks occur on GPUs with the traditional stack-based reconvergence approach. In this chapter we analyze Independent Thread Scheduling and evaluate its utility.

3.1 SIMT Deadlock

First, we consider the simple GPU program shown in Listing 3.1. All concurrent threads access the same Counter object (Listing 3.2) through the pointer *c*. It therefore becomes necessary to implement coordination among them. The synchronization mechanism we use is a simple spin lock. Within the critical section we are incrementing the *counter* (line 6 Listing 3.1).

In Chapter 2 we pointed out that, if we execute this kernel, a SIMT deadlock will occur on all GPUs without ITS. To confirm that ITS actually prevents SIMT deadlock from occurring we tested the kernel on a GPU with a Turing microarchitecture. As the immediate successor of the Volta microarchitecture, it provides ITS. We address the lack of cache coherence by deactivating the L1 cache completely by using the following compiler option:

```
-Xptxas -dlcm=cg
```

Furthermore, the hardware will only make use of ITS, if we compile the given kernel with the following additional compiler option:

```
-arch=sm_75
```

This option instructs the compiler to make use of all features available on the Turing microarchitecture, including ITS.

We compiled two executable programs: one which makes use of ITS, while the other does

Listing 3.1: Simple Spin Lock.

```
__global__ void increment_counter(Counter* c) {
    while(atomicCAS(&(c->lock),0,1) != 0);
    __threadfence();

    //critical section
    c->counter++;

    __threadfence();
    atomicExch(&(c->lock),0);
}
```

Listing 3.2: Counter object.

```
struct Counter {
    int lock;
    int counter;
}
```

not. We ran both programs on the same GPU. The program which makes use of Independent Thread Scheduling produces the correct result. The other program gets stuck in a deadlock.

We conclude that Independent Thread Scheduling does increase programmability by abstracting away the actual thread scheduling and thus frees the programmer from considering SIMT deadlocks.

3.2 Livelock

In this section we present a condition in which the systems as a whole is prevented from making forward progress (livelock) despite Independent Thread Scheduling.

3.2.1 Problem

We observed that in situations which are similar to the pseudo code shown in Listing 3.3, the kernel does not terminate. We say a livelock occurs. Moreover, the livelock occurs even on a GPU with ITS.

Listing 3.3: Livelock.

```
1  void livelock( int* flag) {
2      while (true) {
3          if (flag[THREAD_ID] != 0) {
4              //BB1
5          } else {
6              //BB2
7              return;
8          }
9      }
10
11 void run() {
12     int flag = {1, 0, 0, 0};
13     livelock( flag );
14
15     if (THREAD_ID < 3) flag[THREAD_ID + 1] = 1;
16 }
```

3.2.2 Explanation and Solution

To understand the situation it is important to consider, that threads within the same warp share the same instruction fetch and instruction decode unit. Thus, despite the naming, Independent Thread Scheduling does not mean that the hardware schedules individual threads. Threads are still scheduled and executed together in warps. All threads are executing every instruction, even if the control flow diverges. If control flow diverges, inactive threads are masked and therefore their results within the execution stage are not written back to the register file.

In general, an *If*-branch instruction, results in a warp split. Some threads will execute the *if*-block while others will execute the *else*-block. This pattern is shown in Listing 3.3 on line 3-8. But in Listing 3.3 the control flow is wrapped in an infinite loop. Only if the *else*-block is executed, a thread exits the while loop (line 7). Furthermore the condition on line 3 depends on the execution of the *else*-block. Only if a thread exits at line 7, the condition will change eventually (line 15). Due to the outer while loop, the hardware chooses to schedule the threads which are executing the *if*-block repeatedly. Thus, the *else*-block will never be executed. The program reaches a livelock condition. To prevent such a livelock condition we need to force the hardware to schedule those threads which are executing the *else*-block.

If we consider Listing 3.4 and Listing 3.5. Figure 3.1 shows a possible execution, where threads *t1* and *t3* evaluate the condition to *true*, while *t0* and *t4* evaluate the condition to *false*. The execution shown on the left, corresponds to Listing 3.4 and the execution on the rights shows the execution of Listing 3.5. We force the hardware to switch to the *else*-block by issuing a

Listing 3.4: Simple Control Flow Example with *syncwarp*.

```
1 if (/*Condition*/) {
2     //BB1.1
3     //BB1.2
4 } else {
5     //BB2.1
6     //BB2.2
7 }
```

Listing 3.5: Simple Control Flow Example without *syncwarp*.

```
1 if (/*Condition*/) {
2     //BB1.1
3     __syncwarp();
4     //BB1.2
5 } else {
6     //BB2.1
7     __syncwarp();
8     //BB2.2
9 }
```

syncwarp instruction on line 3 and line 7 in Listing 3.5. This instruction forces the warp splits to wait for each other. First, BB1.1 is executed and the *syncwarp* instruction forces the hardware to schedule the *else*-block. After BB2.1 has been executed the *syncwarp* instruction on line 7 has the effect that the warp split which is executing the *if*-block is in principle able to proceed. But the hardware decides to continue with BB2.2 until the reconvergence point is reached. When the warp split arrives at the reconvergence point, the scheduler switches back to the other warp split. Finally, both warp splits converge and are executing BB3 together.

The same idea of forcing the hardware to switch to the other warp split can be applied to prevent livelock conditions. Therefore, we need to ensure that a *syncwarp* instruction is issued by all threads within the warp. The result is shown Listing 3.6: we inserted a *syncwarp* instruction on line 4 and line 17.

3.2.3 Discussion

By manipulating the scheduling of basic blocks, we are able to prevent livelock conditions. The *syncwarp* instruction is only available on GPUs with Independent Thread Scheduling. The

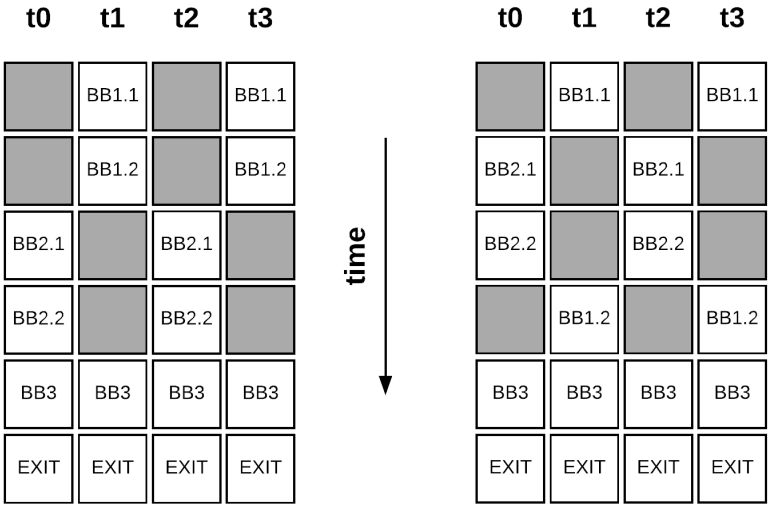

Figure 3.1: Warp Scheduling. On the left without a syncwarp instruction. On the right with a syncwarp instruction.

behaviour of *syncwarp* is undefined[1] on all Pre-Volta architectures the GPUs without Independent Thread Scheduling serialize the execution of control flow and thus it is not possible for individual warp-splits to wait for each other.

3.3 Summary

Independent Thread Scheduling solves the problem of SIMT deadlocks. Nevertheless, a livelock condition could still occur. It is possible to prevent livelock conditions by manipulating the scheduling of basic blocks. But it is necessary to understand how control flow is actually implemented on the GPU. In summary, even for the purposes of correctness, the programmer cannot ignore the SIMT behavior, contrary to the NVIDIA programming guide [25].

[1] In our tests the instruction was simply ignored.

Listing 3.6: Livelock Prevention: *syncwarp*.

```
1  void livelock( int* flag) {
2      while (true) {
3          if (flag[THREAD_ID] != 0) {
4              __syncwarp();
5              //BB1
6          } else {
7              //BB2
8              return;
9          }
10     }
11
12 void run() {
13     int flag = {1, 0, 0, 0};
14     livelock( flag );
15
16     if (THREAD_ID < 3) flag[THREAD_ID + 1] = 1;
17     __syncwarp();
18 }
```

4 Synchronization Primitives: Spin Locks

Efficient synchronization needs to be scalable. To achieve scalability an understanding of the underlying hardware is required. After having gained insight into the hardware architecture of modern GPUs, we are able to present well-known lock variations and show how they can be implemented on GPUs. In this chapter we set the scope for the following evaluation.

Locks guarantee atomicity of operations via mutual exclusion. In general, there are two alternatives to implement mutual exclusion. A thread which failed to acquire a needed lock could either retry until it succeeds (busy waiting) or suspend its execution for a while and retry later. The second alternative involves direct control over the scheduling process. On a GPU we do not have an operating system and we cannot directly control the scheduling of particular threads. Therefore, only busy waiting can be used.

We structure this chapter in two parts: *Centralized Spin Locks* and *Queued Spin Locks*.

4.1 Centralized Spin Locks

The following locks make use of the atomic instructions supported on GPUs. During lock acquisition all competing threads are spinning on the same central location.

4.1.1 Test-And-Set (TAS) Lock

The TAS lock implementation can be seen as a baseline. This lock is the default synchronization primitives and is commonly used (e.g. locking in NVIDIA's CUDA by Example [29]). The signature of the TAS lock implementation is shown in Listing 4.1. The constructor (line 4) allocates memory in the global memory of the GPU to hold an integer. The pointer to this allocated memory is stored in *mutex*. The destructor on line 5 is responsible for reclaiming the memory. The value to which *mutex* points indicates if a thread is currently holding the lock.

Listing 4.1: TAS Lock Signature.

```
1 struct TAS_Lock {
2     int *mutex;
3
4     TAS_Lock( void );
5     ~TAS_Lock( void );
6
7     __device__ void lock( void );
8     __device__ void unlock( void );
9 };
```

The lock method in Listing 4.2 calls the *atomicCAS* instruction in a loop (line 2). If no thread is currently holding the lock, the *atomicCAS* will store a 1 at *mutex*, return a 0 and therefore proceed with line 2. Otherwise, another thread is currently holding the lock, the *atomicCAS* will not change the value at *mutex*, return a *one* and has to repeat the *atomicCAS* instruction until it eventually succeeds.

Listing 4.2: TAS Lock Operation.

```
1 __device__ void lock( void ) {
2     while( atomicCAS( mutex, 0, 1 ) != 0 );
3     __threadfence();
4 }
```

The memory fence instruction on line 3 ensures that the lock is held before the thread enters the critical section. This is necessary due to the weak memory consistency.

The unlock operation shown in Listing 4.3 writes a *zero* at *mutex*. Similar to the lock operation, the memory fence instruction on line 2 ensures that all instructions within the critical section have completed before the lock is released.

Listing 4.3: TAS/TTAS Unlock Operation.

```
1 __device__ void unlock( void ) {
2     __threadfence();
3     atomicExch( mutex, 0 );
4 }
```

4.1.2 Test-and-Test-And-Set (TTAS) Lock

The TTAS lock differs from the TAS lock only in the implementation of the lock method (see Listing 4.4). On a CPU TTAS locks perform significantly better than TAS locks. Because they minimize coherence traffic by avoiding frequent cache invalidations [18]. However, as we point out in chapter 2 GPUs do not implement cache coherence. Thus, it is interesting to investigate if the performance of TTAS differs from TAS on GPUs.

On line 3 we are making use of a special instruction to load the lock value directly from global memory, by-passing the L1 cache. We achieved this through inline assembly wrapped into a inline function. The function is shown in Listing 4.5. In line 3 of Listing 4.5 we annotate the load instruction with the *.cg* cache operator. Any *ld.cg* instruction loads from global memory, by-passing the L1 cache [27]. In this regard a normal memory access differs from an atomic one. Due to their implementation atomic instruction are by-passing the L1 cache per default.

We load the lock value directly from global memory, because other threads may have changed the value and due to the lack of cache coherence the value in the L1 could be out-of-date.

It is important to note, that in this lock variant the *atomicCAS* instruction is not "spinning". Instead the lock "spins" on line 3. Only if a "normal" read of the lock value suggests that we have a chance to acquire the lock, the lock is trying to do so (line 4).

Listing 4.4: TTAS Lock Operation.

```
1 __device__ void lock( void ) {
2     while (true) {
3         while (ld_gbl_cg((int *) mutex)) {}
4         if (!(atomicCAS(mutex, 0, 1) != 0)){
5             __threadfence();
6             return;
7         }
8     }
9 }
```

Listing 4.5: Global Memory Load.

```
1 __device__ __inline__ double ld_gbl_cg(const int *addr) {
2     int return_value;
3     asm volatile ("ld.global.cg.s32 %0, [%1];" : "=r"(return_value) : "l"(
          addr));
4     return return_value;
5 }
```

4.1.3 Ticket Lock

A ticket lock tracks two values: *current* and *next*. Every time a thread wants to acquire the lock, it will read the value of *next* while incrementing *next* by one. The value read by the thread is called its *TICKET ID*. The thread waits until *TICKET ID* is equal to *current*. If a thread exits the critical section, the value at *current* will be incremented and the next waiting thread is able to proceed.

The signature of the ticket lock is shown in Listing 4.6. Listing 4.7 shows the implementation of the corresponding lock method. On line 2 the thread receives its *TICKET ID*. The thread checks repeatedly if it is allowed to enter the critical section (line 4). In order to ensure that the value read at *current* is not out-of-date, the thread performs a L1 by-passing load (Listing 4.5) similar to the *TTAS* lock.

Listing 4.6: Ticket Lock Signature.

```
1  struct TICKET_Lock {
2      int *current;
3      int *next;
4
5      TICKET_Lock( void );
6      ~TICKET_Lock( void );
7
8      __device__ void lock( void );
9      __device__ void unlock( void );
10 };
```

A branch instruction might result in a warp split. If a warp split occurs, a GPU with Independent Thread Scheduling will be able to schedule any of them. But if the hardware decides to schedule only the warp split whose threads are not authorized to access the critical section, a livelock will occur. We prevent the livelock by forcing the hardware to schedule the other warp

Listing 4.7: Ticket Lock Operation.

```
1 __device__ void lock( void ) {
2     unsigned own_ticket = atomicAdd(next, 1 );
3     while (true) {
4         if (own_ticket != ld_gbl_cg(current)) {
5             __syncwarp();
6         } else {
7             __threadfence();
8             return;
9         }
10    }
11 }
```

split. We can do so by executing a *syncwarp* instruction (line 5). This instruction forces the warp split to wait until the other warp split has executed a corresponding *syncwarp* instruction as well. Thus the following executions are possible:

1. No thread within the warp is authorized to access the critical section. All threads are executing the *syncwarp* instruction and the warp as a whole can check again if any thread is authorized to enter the critical section.

2. Because threads can operate on different data points (e.g. a different lock), it is possible that multiple threads are authorized to enter the critical section. After the threads which are not allowed to enter the critical section execute the *syncwarp* instruction, the hardware immediately schedules those threads which are authorized to enter the critical section. These threads execute the *syncwarp* instruction within the unlock operation (Listing 4.8). The execution switches back to those threads which still have to execute the critical section.

3. All threads are allowed to enter the critical sections. All threads are executing the *syncwarp* instruction within the unlock operation.

Ticket locks have another important property: they guarantee fairness. With TAS and TTAS it is possible that a thread bypasses another thread that has already been waiting for a long time.

The values *current* and *next* could overflow, but such a rollover is harmless as long as the maximum number of threads is less than the largest representable integer.

Listing 4.8: Ticket Unlock Operation.

```
__device__ void unlock( void ) {
    __threadfence();
    atomicAdd(current, 1 );
    __syncwarp();
}
```

4.2 Queued Spin Locks

The basic idea behind queued spin locks is that all participating threads form a queue. Each thread knows its successor. When a thread exits the critical section, it signals its successor to enter the critical section.

4.2.1 MCS Lock

Listing 4.9: MCS Lock Signature.

```
struct QNode {
    int waiting;
    int next;
};

struct MCS_Lock {
    int * tail;
    int * head;
    QNode * qNode;

    MCS_lock( void );

    ~MCS_lock( void );

    __device__ void lock( void );
    __device__ void unlock( void );
};
```

The signature of our implementation of the *Mellor-Crummy and Scott* (MCS) lock [23] and the auxiliary data structure *QNode* is shown in Listing 4.9. The constructor allocates memory for two integers on the GPU and stores their address in *head* and *tail*. Furthermore, we store a pointer to an array of *QNode* elements. The *QNode* array is allocated separately, because we need only one *QNode* array which is shared among all locks. During initialization we set this pointer for each MCS lock. The size of the *QNode* array is determined by the number of con-

current threads.

The lock operation of the MCS lock is presented in Listing 4.10. First, a locking thread must determine its own *QNode* element. Therefore each thread accesses its *THREAD ID* (*TID*) on line 2. The corresponding *QNode* element is identified on line 3. Because the *TID* is unique we do not synchronize the accesses to the *QNode* array. We shift the *TID* by one, because the first element within the *QNode* is used as a *NULL* element. Next, we build the list element (line 5-6). On line 8 we add the new list element to the queue. By using the *atomicExch* instruction the thread identifies its predecessor. If the thread has no predecessor (*prev* is equal to 0), it will enter the critical section immediately. Otherwise, the threads spins on *waiting* (line 12) until it has been set to 0 by its predecessor. As a first instruction within the critical section, each thread stores the used slot (line 15) in order to identify the correct list element during an unlock operation.

Listing 4.10: MCS Lock Operation.

```
1  __device__ void lock() {
2      int tid = threadIdx.x + blockIdx.x * blockDim.x;
3      int slot = tid + 1;
4
5      qNode[slot].next = 0;
6      qNode[slot].waiting = 1;
7
8      int prev = atomicExch(tail, slot);
9
10     if(prev != 0) {
11         qNode[prev].next = slot;
12         while( atomicCAS(&qNode[slot].waiting, 0 , 0) != 0) {}
13     }
14     __threadfence();
15     *head=slot;
16 }
```

To unlock a MCS lock (Listing 4.11) the thread needs to restore its own position in the *QNode* array. (line 2). Therefore this instruction precedes the memory fence on line 3 it is done within the critical section. If the thread does not have a successor (*succ* is equal to 0), we need to distinguish two possible situations:

1. If the thread is the last element in the queue, the value at *tail* must be equal to the threads position in the *QNode* array and the thread has successfully unlocked the lock. This condition is checked on line 9-10.

2. If the value at *tail* is not equal to the threads position in the *QNode* array, then the cur-

Listing 4.11: MCS Unock Operation.

```
1  __device__ void lock() {
2      __device__ void unlock() {
3          int slot = *head;
4          __threadfence();
5
6          int succ = qNode[slot].next;
7
8          if (succ == 0) {
9              if (atomicCAS(tail, slot, 0) == slot) {
10                 return;
11             }
12             while ( atomicCAS(&qNode[slot].next, 0, 0) == 0);
13         }
14         succ = qNode[slot].next;
15         qNode[succ].waiting = 0;
16     }
17 }
```

rent thread needs to wait (line 12). This situation occurs when another thread is already queued behind the current thread (line 8 in Listing 4.10), but has not updated its predecessor (line 11 in Listing 4.10).

At this point, if the thread has not already unlocked the MCS lock, it has been assured that the current thread has an successor. To unlock the MCS lock the thread signals its successor to proceed (line 15).

4.2.2 MCS2 Lock

With the MCS lock we are managing the lock by maintaining different lists. A single list element is represented as a *QNode*. We are pre allocating all QNode elements in an array. If we want to allow a single thread to hold multiple locks simultaneously, we need to modify the previously presented MCS lock.

Instead of using a *QNode* array, MCS2 uses a *QNode2* array shown in Listing 4.12. This array is named *qNode*. To support n threads and two locks per thread we need $(2n + 1)$ *QNode2* elements. MCS2 differs from MCS mainly in how the position in the *qNode* is calculated. Instead of shifting the *TID* by 1, we calculate the position pos as follows:

$$pos = 2 * TID + 1$$

After we calculated *pos*, it becomes necessary to check if $qNode[pos]$ is already used by accessing *active*. If the element at *pos* is already being used the thread increments *pos* by one.

Listing 4.12: QNode2 Signature.

```
1 struct QNode2 {
2     int waiting;
3     int next;
4     int active
5 };
```

Finally, the thread needs to ensure that *active* is kept up-to-date by setting *active* to 1 before the elements is queued and to 0 when it is dequeued.

5 Evaluation

In chapter 4 we presented the synchronization primitives and how they could be implemented on GPUs. In this chapter we evaluate them in different scenarios.

This investigation reveals insights into how the synchronization primitives behave on GPUs. Moreover, we are able to compare different lock variants with each other.

In the first section we concentrate on the synchronization primitives and evaluate them under two different scenarios. In the second section, we evaluate different synchronization primitives using fine grained locking with lock coupling and compare fine grained locking against coarse grained locking.

5.1 Hardware and Software Setup

Tabel 5.1 summarizes our hardware and software setup.

GPU	
Model:	GeForce RTX 2060
Architecture:	Turing
Cuda Cores	1920
Number of Multiprocessors	30
Compute Capability	7.5
CUDA Driver Version	10.1
CUDA Runtime Version	10.1
Warp Size	32
Global Memory	6 GB

Table 5.1: Hardware & Software Setup.

5.2 Synchronization Primitives

We begin our evaluation with a very easy-to-understand microbenchmark: the Counter microbenchmark. After that, we investigate the insertion into a hash table.

5.2.1 Counter

The objective behind the Counter microbenchmark is to isolate contention and investigate how synchronization affects execution time. The GPU kernel operates on a shared array of *Counter* objects. Each *Counter* objects is identified by its position within the array. The total number of *Counter* objects is configurable. After the kernel is initialized, each thread uses its own thread ID to access a shared buffer with work-items (work queue). Each work-item instructs a thread to increment the counter of a specific *Counter* object by providing a counter ID.

We configured this microbenchmark to run on a single *Counter* object and with 32768 work-items. Figures 5.1-5.4 show different synchronization primitives. Different grid size (1, 8, 16, 32) are plotted in each Figure. The grid size is synonymous to the number of thread blocks. On the x-axis we scale the number of threads per thread block from 32 to 1024. The total number of concurrent threads is calculated by the value at the x-axis times the grid size: We have at least 32 and up to 32768 concurrent threads which are competing for the same *Counter* object. The minimal and maximal value on the x-axis are not randomly chosen: 32 corresponds to the warp size and 1024 is the maximum number of threads per thread block our GPU supports. As a baseline we used the execution time of a single thread, because this shows the effect of contention.

5.2.1.1 TAS Lock

We start our evaluation on the Counter benchmark with the TAS Lock (Figure 5.1). With a grid size of 1 the TAS is able scale reasonably. Only if more than 384 threads are used, we observe a departure from our baseline. With other grid sizes, we observe a rapid departure from the baseline.

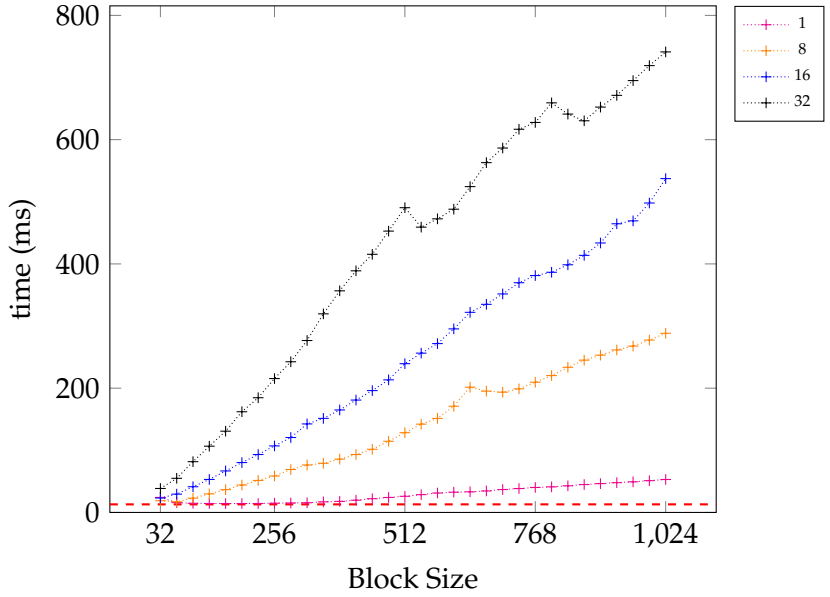

Figure 5.1: TAS-lock: Incrementing counter with different grid sizes.

5.2.1.2 TTAS Lock

The plot of the TTAS lock in Figure 5.2 locks similar to the plot of the TAS lock shown in Figure 5.1. The only difference is that the effect of contention is slightly mitigated. It is important to note the different scales used for TAS and TTAS. Another interesting aspect is that runtime drops if we increase the number of threads from 32 to 64. Thus we are able to make use of the increased concurrency.

5.2.1.3 Ticket Lock

The ticket lock presented in Figure 5.3 shows good scaling behavior for lower grid sizes. With a grid size of 1 and 8 the runtime is unaffected by the number of thread per thread block. Furthermore, we observe that the ticket lock is able to make use of increasing the grid size from 1 to 8, if we compare the magenta colored line with the orange colored line. With higher grid sizes the execution time increases. If more than 256 threads per thread block are used (black/blue line). With less than 256 threads per thread block the tested grid sizes of 8, 16 and 32 perform equally and are able to make use of increased concurrency.

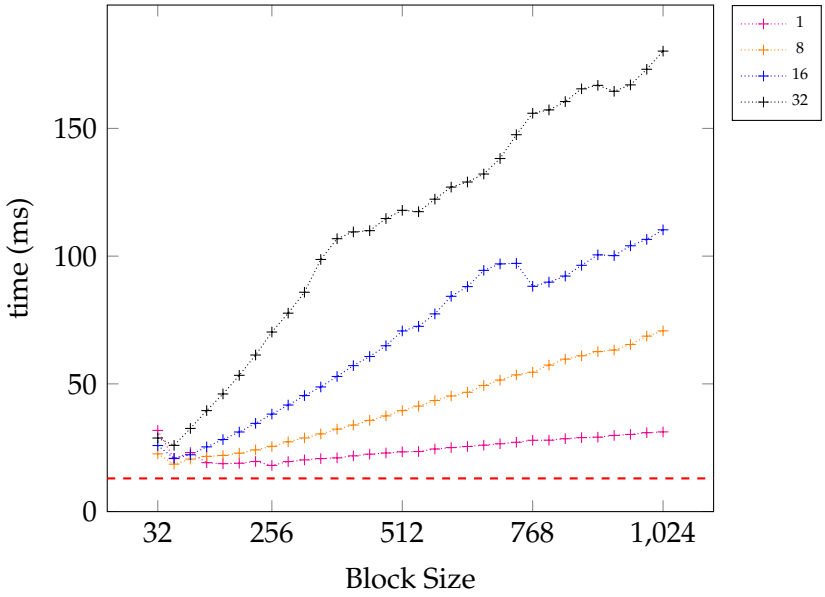

Figure 5.2: TTAS lock: Incrementing counter with different grid sizes.

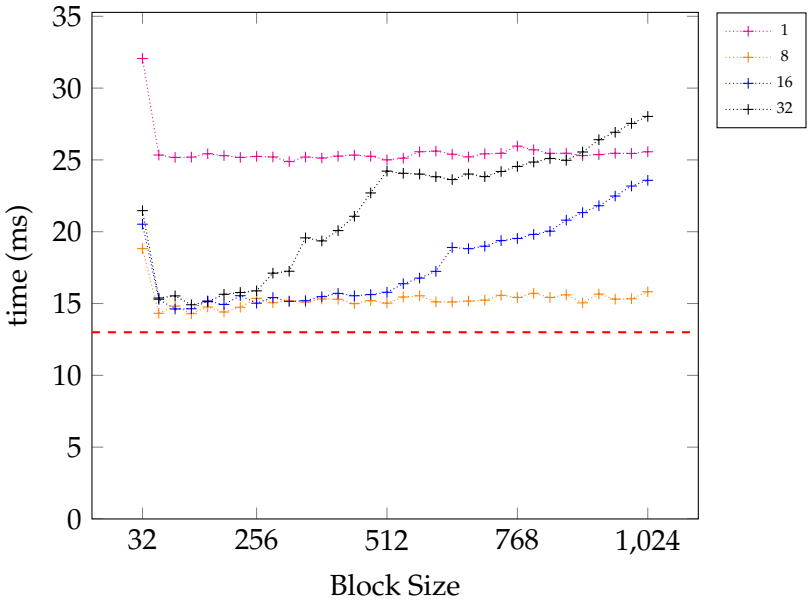

Figure 5.3: Ticket lock: Incrementing counter with different grid sizes.

5.2.1.4 MCS Lock

Figure 5.4 shows that all tested grid sizes are affected by the number of threads per thread block, similiar to TAS and TTAS. But in contrast to TAS and TTAS, the scaling behaviour of the tested grid sizes is roughly the same. If less than 256 threads per thread block are used, MCS is able to make use of higher grid sizes.

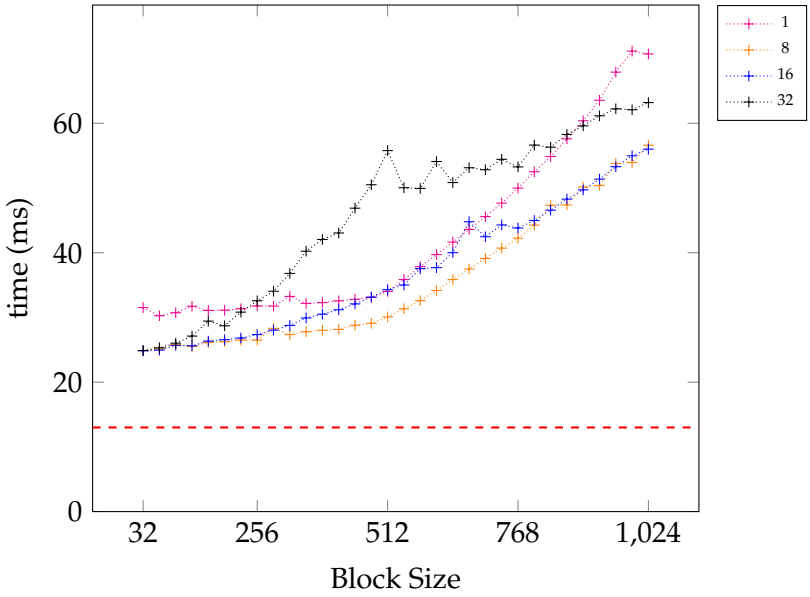

Figure 5.4: MCS Lock: Incrementing counter with different grid sizes.

5.2.1.5 Discussion

In this section we evaluated different lock variants with regard to their scalability. For this purpose we used the simplest possible task: incrementing a counter. We were able to validate the correctness without great effort by comparing the end results.

We explain the huge difference between TAS and TTAS by the fact that TTAS is able to avoid atomic instructions. Only if a non-atomic read suggests a chance to acquire the lock, TTAS executes an atomic instruction. This correlation between scalability and the ability to avoid atomic instructions also explains why a ticket lock performs even better. With a ticket lock each thread executes exactly two atomic instruction: one during the lock operation and one during the unlock operations. But with TTAS multiple waiting threads are simultaneously able to recognize an unlock operation. Although only one thread would be able to acquire the lock next, all of them would execute an *atomicCAS* instruction.

MCS achieves good scalability even though the threads are spinning with an *atomicCAS*, because each thread executes on a different memory location. Unfortunately, we are not able to explain why an *atomicCAS* is even necessary. But in all our tests where MCS was spinning with a non-atomic instruction the execution got stuck and we had to terminate the execution manually. This issue needs to be adressed in future work.

5.2.2 Hash Table

In this section we investigate how the synchronization primitives behave if we use them in a more realistic workload. Therefore, we adapted similar to ElTantawy and Aamodt [13] the code from NVIDIA's CUDA by Example [29], which implements locking on a hash table, and integrated our lock variations.

We insert 26.2 million key-value pairs. The keys are uniformly distributed. Contention is controlled by the number of hash entries within the hash table. Each hash entries correspond to a bucket which is implemented by a linked list. New key-value pairs are inserted at the beginning of the corresponding bucket list. We use four different contention levels. The four levels correspond to the following numbers of hash table entries: 256, 64, 32 and 16. We compare our four lock variations (TAS, TTAS, Ticket, MCS) and a lock-free variation. The lock-free variation uses *atomicCAS* to swap pointers directly in order to insert into a bucket list. We ran our experiments with a grid size of 30 and a block size of 256 which results in 7680 concurrent threads.

5.2.2.1 Results

The results are shown in Figure 5.5. With low contention (16 hash table entries) execution time is roughly the same on all tested variants. But with higher levels of contention the differences become apparent. On our high contention scenario with 16 hash table entries, the MCS is 3.4 times faster than the standard spin lock (TAS). MCS performs even better than the lock-free implementation. With 32 hash entries Ticket performs on par with MCS and TTAS could catch up to lock-free. While MCS and Ticket are 1.4 times faster than lock-free respectively TTAS, they are 3 times faster than TAS. If we lower the contention further (64 hash entries), all variants but TAS perform equally. With 256 hash entries no variation stands out.

5.2.2.2 Discussion

The results indicate the main performance bottleneck in scaling concurrent accesses is contention. Therefore, high contention should be avoid at all cost. If we cannot avoid high contention, the choice of the synchronization primitive contributes significantly to the overall performance.

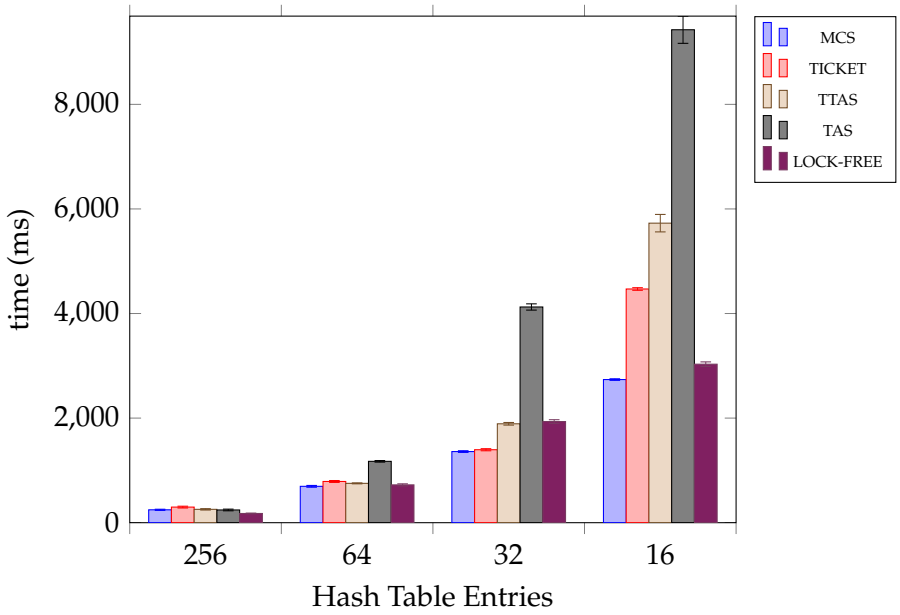

Figure 5.5: Insertion into hash table with different levels of contention (low to high).

5.3 Lock Granularity

Lock granularity determines the potential concurrency. Lock coupling ensures linearizability with fine-grained locking. It is used to allow concurrent inserts into a sorted linked list or other order preserving data structures (e.g. B-Tree).

In this section we evaluate the implementation of our synchronization primitives, when used with lock coupling. Next, we evaluate how coarse-grained locking performs on the same workload. Finally, we compare fine-grained with coarse-grained locking.

5.3.1 Setup

We insert 262,000 key-value pairs into an ordered linked list. The list is composed of multiple ordered sublists. Each sublist holds all keys within a certain range. The last element in each sublist points to the first element of the next sublist. We call the first element of a sublist entry point. All entry points are managed by a hash table. The hash function determines for a given key the corresponding entry point. Is is implemented by a shift right operation. The shift amount depends on the range a sublist is responsible for. For instance, if a sublist is responsible for 1024 keys, we need to shift any given key by 10 to the right. In other words, the most significant bits are used to determine the sublist, whereas the least significant bits determine

the position in a given sublist.

5.3.2 Lock Coupling

In this section we evaluate our lock variations with lock coupling. Each list element is protected by its own lock. Each traversing thread holds two locks at most. If a thread holds a lock on a given list element, lock coupling ensures that this thread is never overtaken by another thread. But multiple threads are able to traverse the list in a pipelined fashion.

We distinguish a high contention scenario in Figure 5.6 and a low contention scenario in Figure 5.7. We configured the kernel with a grid size of 30 and a block size of 128: 3840 concurrent threads. In both figures we scale the number of entry points (x-axis): few entry points result in high contention while many entry points result in low contention. For all lock variations we observed a step function behaviour. With decreasing contention the performance of the different lock variations converges. TTAS and TAS perform better than Ticket and MCS2 with high and low contention.

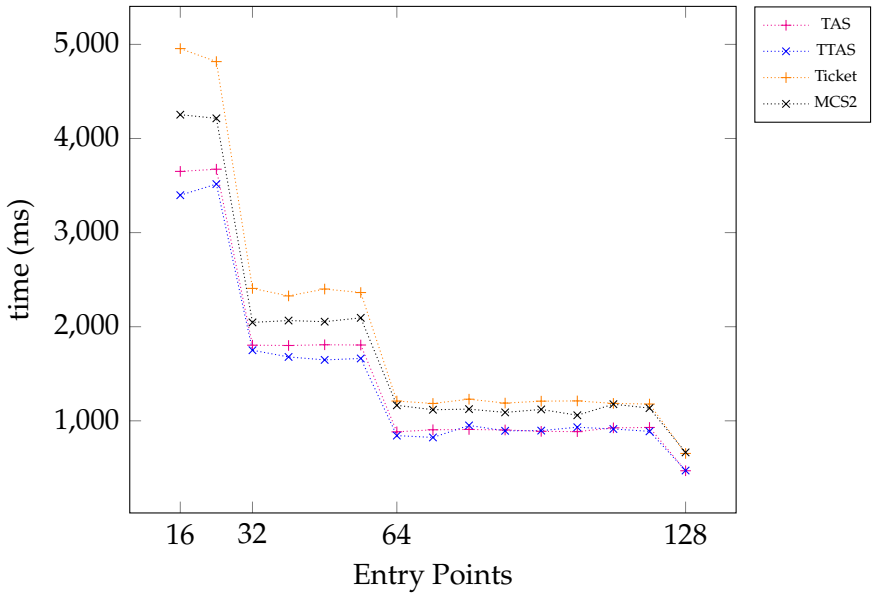

Figure 5.6: Lock Coupling: High Contention.

Discussion. In contrast to the previous benchmark, we observed that a more complex lock implementation does not pay off. Instead, the simple TAS lock performs the best. However, we could confirm the importance of low contention.

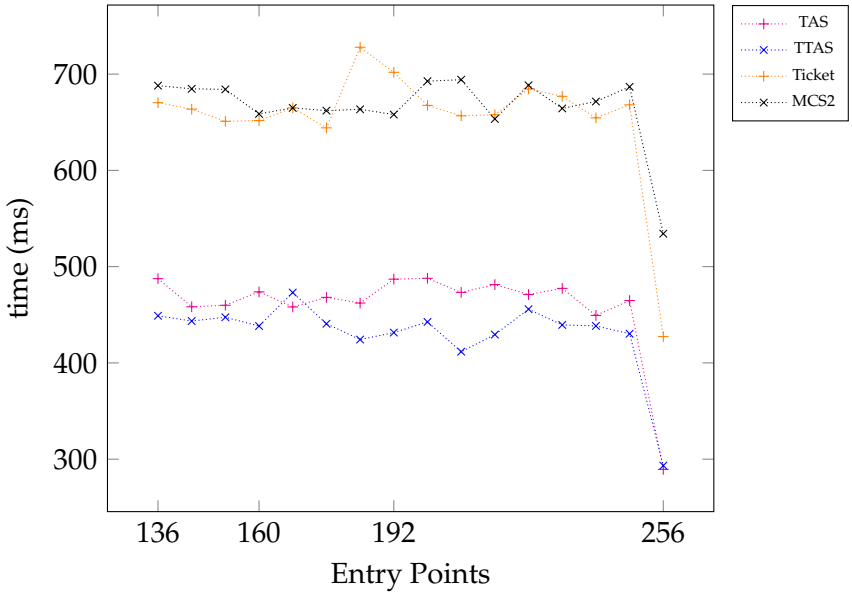

Figure 5.7: Lock Coupling: Low Contention.

5.3.3 Coarse-Grained Locking

After we measured how our lock variations perform with lock coupling, we are evaluating them with coarse-grained locking. In this setting we do not protect each list element individually. Instead we only lock the entry points. If a thread holds a lock, no other thread is able to insert into the same sublist. We control the level of contention by configuring the number of entry points. In all scenarios we scale the block size (x-axis). All kernels were run with a grid size of 30.

The results are shown in Figure 5.8-5.11. In order to differentiate coarse-grained locking and lock coupling we annotate the lock variation with the suffix "-CG". In Figure 5.10 and Figure 5.11 we are comparing all lock variations. In Figure 5.8 and Figure 5.9 we omitted TAS-CG, because of its poor scaling behaviour.

Each plot shows that the runtime is decreasing, if we increase the block size. TTAS reaches its optimum faster than the others. With a 128 threads per block each lock reaches it optimal performance. But if we scale the block size further than 256 threads per block, performance decreases again. With high contention (16 and 32 entry points) the scaling behaviour of TTAS, MCS2 and Ticket is quite similar, whereas with low contention (64 and 128 entry points) each lock reacts differently. In the range where all locks reach their optimum (128-256 threads), TTAS performs better than MCS2 and Ticket. In this range, even TAS is often able to outperform MCS2 and Ticket.

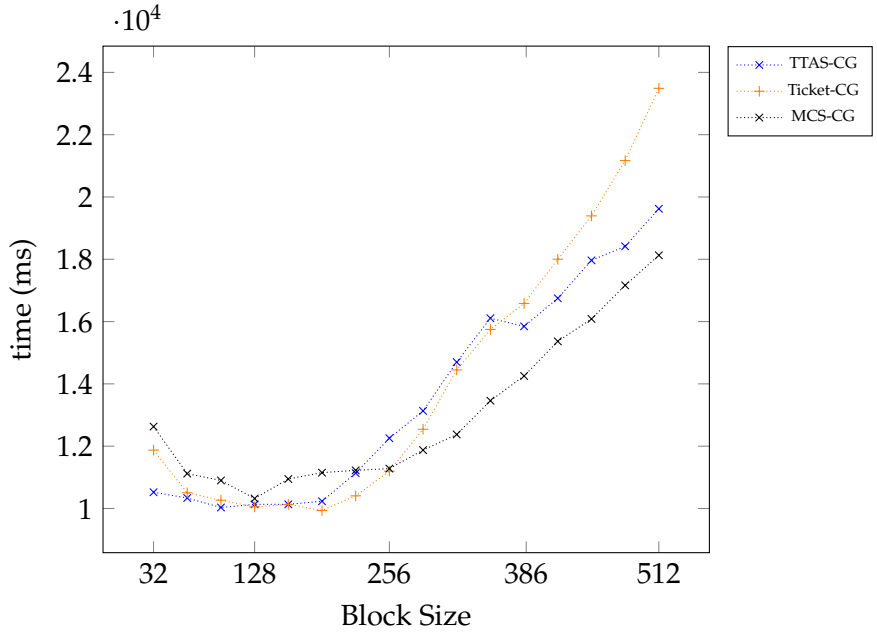

Figure 5.8: Coarse Grained Locking: 16 Entry Points, 30 Thread Blocks.

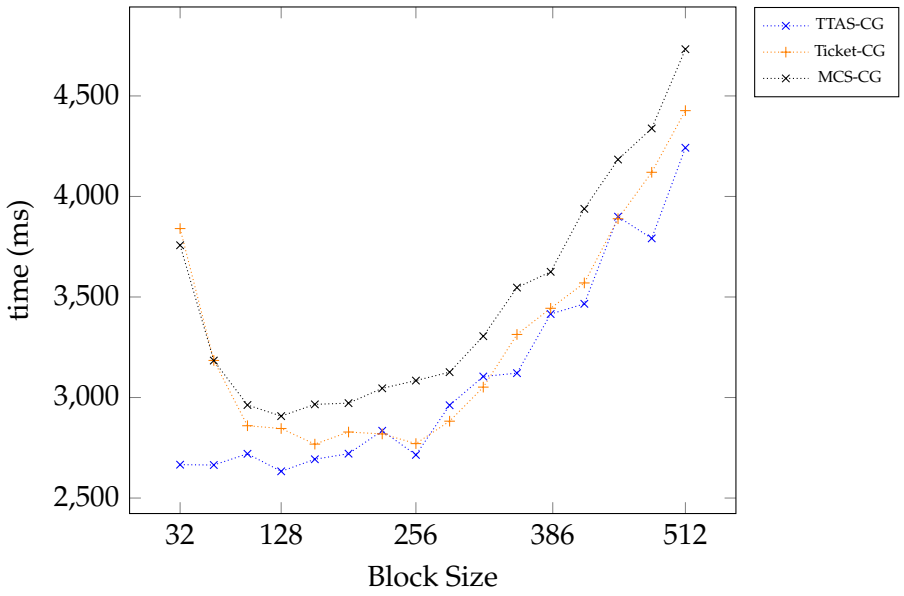

Figure 5.9: Coarse Grained Locking: 32 Entry Points, 30 Thread Blocks.

5.3.4 Comparison

In our last evaluation we are using TAS and TTAS to compare their performance using lock coupling with their performance using coarse-grained locking. We chose them because the previous evaluations indicated their superior performance. We configured the benchmark with

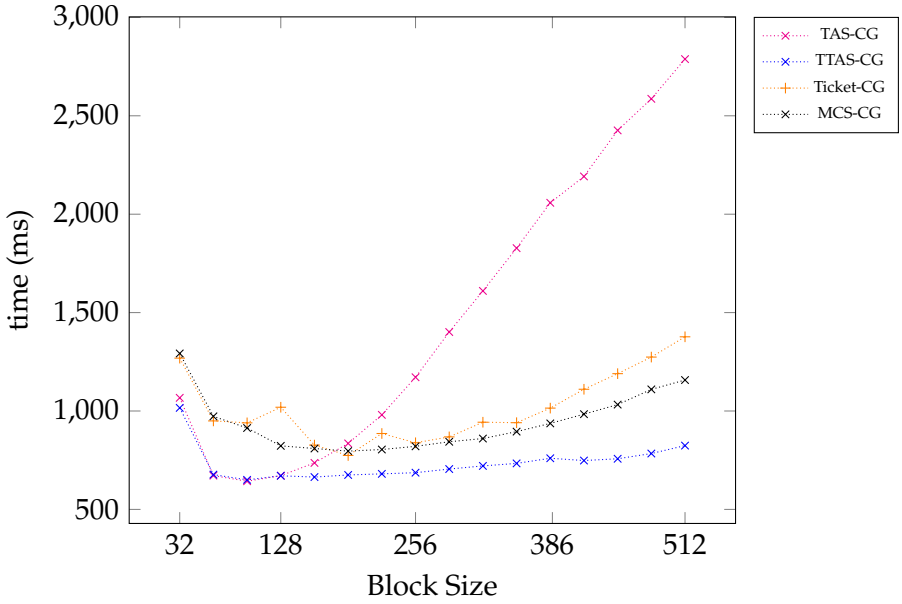

Figure 5.10: Coarse Grained Locking: 64 Entry Points, 30 Thread Blocks.

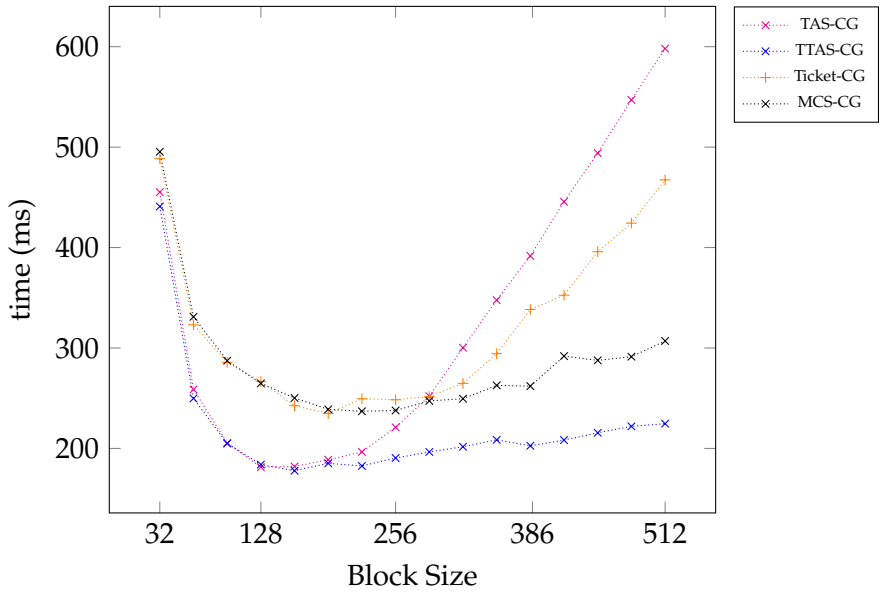

Figure 5.11: Coarse Grained Locking: 128 Entry Points, 30 Thread Blocks.

a fixed grid size of 30 and a block size of 128. Figure 5.10 and Figure 5.11 suggest that both TAS-CG and TTAS-CG reach their optimal performance with this configuration.

The results of our comparison are shown in Figure 5.12 and Figure 5.13. Figure 5.12 can be divided into three phases. First, if we have fewer than 64 entry points, lock coupling is up to 2.4 times faster than coarse grained locking. Next, with 64 or more, but less than 128, entry

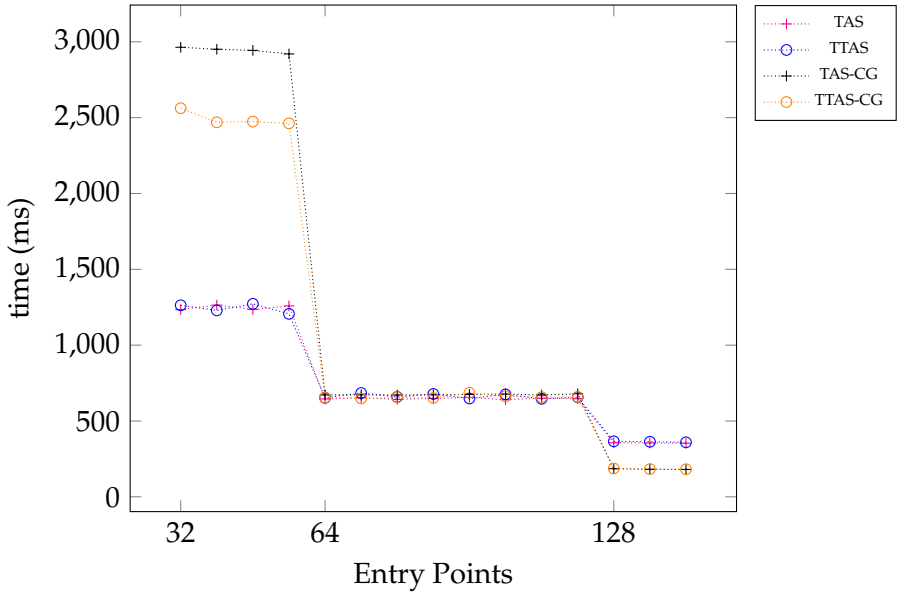

Figure 5.12: Coarse Grained Locking and Fine Grained Locking: High Contention.

points lock coupling is on par with coarse grained locking. Finally, with 128 or more entry points coarse grained-locking is 2.0 times faster than lock coupling. Figure 5.13 can be seen as a continuation of the third phase: coarse-grained locking remains faster than lock coupling.

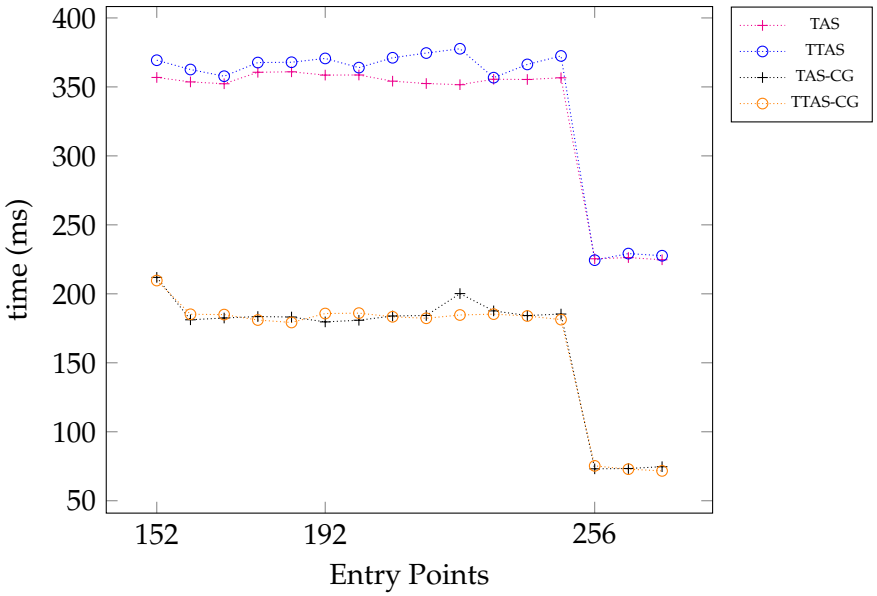

Figure 5.13: Coarse Grained Locking and Fine Grained Locking: Low Contention.

5.4 Summary

The impact of a given lock variation depends on the given situation. If we have high contention combined with a very short critical section (e.g. in our Counter benchmark), the ticket lock will outperform the others. In some workloads, like inserting in a hash table, the performance can be improved by using a more sophisticated lock implementation (e.g. MCS). But in others a simple TAS lock achieves the best performance. Finally, increasing the potential concurrency with fine-grained locking does not necessarily improve performance.

6 Related Work

This thesis relates to different research areas. We present related work on mutable data structures first. Next, we summarize the work concerning synchronization on modern in-memory database systems. Finally, we present related work on GPU Hardware Architecture.

6.1 Mutable Data Structures

Traditionally, GPUs are used to accelerate read-heavy OLAP queries. But the following papers demonstrate that highly concurrent data structures could be implemented on GPUs as well.

Ashkiani et al. [5] propose a dynamic hash table for GPUs, which allows incremental updates (such as insertion and deletions). They propose a new work sharing strategy which reduces branch divergence.

Awad et al. [6] implemented a GPU implementation of a B-Tree that supports concurrent queries (point, range, successor) and updates (insertion and deletions). They demonstrate that the key limiter of performance on GPU is contention.

6.2 Synchronization

The following papers investigate synchronization of concurrent data structures used in modern in-memory database systems. All of them evaluate synchronization on CPUs, whereas our work evaluate synchronization on GPU with Independent Thread Scheduling.

Leis et al. [21] identified that database systems often use fine-grained locking in order to protect their data structures. They point out that the performance of transactional database systems is critically dependent on efficient synchronization. Additionally, they argue that lock-free synchronization is hard to implement but does scale significantly better than traditional

fine-grained locking approaches. Finally, they present Optimistic Lock Coupling and Read-Optimized Write EXclusion (ROWEX), two locking protocols which are both easy to reason about and achieve high scalability.

The work of Faleiro and Abadi [14] questions the superior scalability of lock-free synchronization. They conclude that lock-free synchronization outperforms lock-based synchronization only when the number of threads exceeds the number of processing cores. But modern main-memory database systems usually implement a one-to-one mapping between threads and processing cores. In such an environment their evaluation shows that lock-free synchronization is not able to outperform the lock-based alternative. Finally, they identify contention as the main bottleneck of synchronization.

Wang et al. [33] confirm the difficulty of implementing correct synchronization without locks. Nevertheless, they were able to improve the performance of the lock-free Bw-Tree, which was originally proposed by Microsoft Research. Despite their improvements, the Bw-Tree ist not able perform as well as other concurrent data structures that use locks.

6.3 GPU Hardware Architecture

The hardware architecture on modern GPUs differs significantly from contemporary CPUs. The research on GPU hardware improvements tries to optimize the performance on throughput oriented workloads and to increase the general applicability of GPUs.

For instance, ElTantawy and Aamodt [13] recognize fine-grained synchronization as an integral part of many parallel algorithms. They propose changes to the current GPU hardware to better support fine-grained synchronization. To evaluate their proposed changes, they used GPGPU-Sim which provides a detailed simulation model of GPUs running CUDA workloads.

In another paper ElTantawy and Aamodt [12] proposed a static analysis technique that detects SIMT deadlocks and a *Control Flow Graph* (CFG) transformation that avoids SIMT deadlocks. Finally, they propose an hardware reconvergence mechanism that prevents SIMT deadlocks without a CFG transformation.

Instead of proposing changes to the hardware, we are focusing on contemporary GPU hardware architectures. Therefore, we are able to evaluate different synchronization techniques on actual GPU hardware.

7 Conclusion

In this work, we investigated different synchronization techniques and evaluated their performance in different scenarios. We investigated Independent Thread Scheduling a new hardware feature introduced with the Volta microarchitecture. Independent Thread Scheduling increases programmability by preventing SIMT deadlocks. We demonstrated that it is still necessary to consider the underlying hardware in order to prevent livelock conditions. Furthermore, we proposed a technique which is able to prevent livelocks.

Next, we presented four different lock variations: TAS, TTAS, Ticket and MCS. We showed how the lack of cache coherence and the relaxed memory consistency could be addressed in order to ensure correct synchronization. Additionally, we proposed MCS2, a modification of the MCS lock. With MCS2 a thread is able to hold up to two locks simultaneously.

We evaluated the scalability of the presented lock variations. By isolating the effects of contention, we were able to demonstrate that the ticket lock scales the best. Nevertheless, when used to synchronize concurrent inserts into a hash table, this superior scalability does not result in better performance. Instead MCS achieved the best performance in such a scenario. With MCS we were able to achieve a speed up to a factor of 3.4, when compared with TAS. Furthermore, we showed that under high contention MCS is able to outperform lock-free synchronization. If locking is used to synchronize inserts into a sorted list, we observed that TAS outperforms MCS.

Finally, in our sorted list benchmark we showed that with high contention and long list sizes fine-grained locking performs better than coarse-grained locking. But if we increase the contention and decrease the list size respectively, coarse grained locking will outperform fine-grained locking.

Bibliography

Printed References

[1] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. "General-purpose graphics processor architectures". In: *Synthesis Lectures on Computer Architecture* 13.2 (2018), pp. 1–7.

[2] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. *General-purpose graphics processor architectures*. Vol. 13. 2. Morgan & Claypool Publishers, 2018. Chap. 3, pp. 22–33.

[3] Jade Alglave et al. "GPU Concurrency: Weak Behaviours and Programming Assumptions". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 577–591. ISBN: 9781450328357. DOI: 10.1145/2694344.2694391. URL: https://doi.org/10.1145/2694344.2694391.

[4] Iya Arefyeva et al. "Low-Latency Transaction Execution on Graphics Processors: Dream or Reality?" In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, August 27, 2018*. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2018, pp. 16–21. URL: http://www.adms-conf.org/2018-camera-ready/low-latency.pdf.

[5] S. Ashkiani, M. Farach-Colton, and J. D. Owens. "A Dynamic Hash Table for the GPU". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2018, pp. 419–429. DOI: 10.1109/IPDPS.2018.00052.

[6] Muhammad A. Awad et al. "Engineering a High-Performance GPU B-Tree". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 145–157. ISBN: 9781450362252. DOI: 10.1145/3293883.3295706. URL: https://doi.org/10.1145/3293883.3295706.

[7] Rudolf Bayer and Mario Schkolnick. "Concurrency of operations on B-trees". In: *Acta informatica* 9.1 (1977), pp. 1–21.

[8] Sebastian Breß. "The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS". In: *Datenbank-Spektrum* 14.3 (2014), pp. 199–209.

[9] Sebastian Breß et al. "Gpu-accelerated database systems: Survey and open challenges". In: *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*. Springer, 2014, pp. 1–35.

[10] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. "Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 33–48. ISBN: 9781450323888. DOI: 10.1145/2517349.2522714. URL: https://doi.org/10.1145/2517349.2522714.

[11] Gregory Frederick Diamos et al. "Execution of divergent threads using a convergence barrier". US20160019066A1. 2015.

[12] A. ElTantawy and T. M. Aamodt. "MIMD synchronization on SIMT architectures". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–14. DOI: 10.1109/MICRO.2016.7783714.

[13] A. ElTantawy and T. M. Aamodt. "Warp Scheduling for Fine-Grained Synchronization". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 375–388. DOI: 10.1109/HPCA.2018.00040.

[14] Jose M Faleiro and Daniel J Abadi. "Latch-free synchronization in database systems: Silver bullet or fool's gold?" In: *CIDR*. 2017, p. 9.

[15] Goetz Graefe. "A Survey of B-Tree Locking Techniques". In: *ACM Trans. Database Syst.* 35.3 (July 2010). ISSN: 0362-5915. DOI: 10.1145/1806907.1806908. URL: https://doi.org/10.1145/1806907.1806908.

[16] Bingsheng He and Jeffrey Xu Yu. "High-Throughput Transaction Executions on Graphics Processors". In: *Proc. VLDB Endow.* 4.5 (Feb. 2011), pp. 314–325. ISSN: 2150-8097. DOI: 10.14778/1952376.1952381. URL: https://doi.org/10.14778/1952376.1952381.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[18] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[19] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors, Third Edition: A Hands-on Approach*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 9780128119860.

[20] Victor W. Lee et al. "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: Association for Computing Machinery, 2010, pp. 451–460. ISBN: 9781450300537. DOI: 10.1145/1815961.1816021. URL: https://doi.org/10.1145/1815961.1816021.

[21] Viktor Leis et al. "The ART of Practical Synchronization". In: *Proceedings of the 12th International Workshop on Data Management on New Hardware*. DaMoN '16. San Francisco, California: Association for Computing Machinery, 2016. ISBN: 9781450343190. DOI: 10.1145/2933349.2933352. URL: https://doi.org/10.1145/2933349.2933352.

[22] E. Lindholm et al. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2 (Mar. 2008), pp. 39–55. ISSN: 1937-4143. DOI: 10.1109/MM.2008.31.

[23] John M Mellor-Crummey and Michael L Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors". In: *ACM Transactions on Computer Systems (TOCS)* 9.1 (1991), pp. 21–65.

[24] John Nickolls and William J Dally. "The GPU computing era". In: *IEEE micro* 30.2 (2010), pp. 56–69.

[28] Michel Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media, 2012.

[29] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming, portable documents*. Addison-Wesley Professional, 2010.

[30] Michael L Scott. "Shared-memory synchronization". In: *Synthesis Lectures on Computer Architecture* 8.2 (2013), pp. 1–221.

[31] I. Singh et al. "Cache coherence for GPU architectures". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2013, pp. 578–590. DOI: 10.1109/HPCA.2013.6522351.

[32] Herb Sutter and James Larus. "Software and the Concurrency Revolution". In: *Queue* 3.7 (Sept. 2005), pp. 54–62. ISSN: 1542-7730. DOI: 10.1145/1095408.1095421. URL: https://doi.org/10.1145/1095408.1095421.

[33] Ziqi Wang et al. "Building a Bw-Tree Takes More Than Just Buzz Words". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 473–488. ISBN: 9781450347037. DOI: 10.1145/3183713.3196895. URL: https://doi.org/10.1145/3183713.3196895.

Online References

[25] NVIDIA. *CUDA C++ PROGRAMMING GUIDE*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 01/17/2020).

[26] NVIDIA. *Inside Volta: The World's Most Advanced Data Center GPU*. URL: https://devblogs.nvidia.com/inside-volta/ (visited on 01/21/2020).

[27] NVIDIA. *Parallel Thread Execution ISA Version 6.5*. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/ (visited on 01/29/2020).