Parallel Algorithms and Dynamic Data Structures on the Graphics Processing Unit: a warp-centric approach

By

SAMAN ASHKIANI

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair John D. Owens

Nina Amenta

Venkatesh Akella

Committee in Charge

2017

To my parents, Mina and Farshad; and my brother, Soheil.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Parallel Algorithms and Dynamic Data Structures on the Graphics Processing Unit: a warp-centric approach

Graphics Processing Units (GPUs) are massively parallel processors with thousands of active threads originally designed for throughput-oriented tasks. In order to get as much performance as possible given the hardware characteristics of GPUs, it is extremely important for programmers to not only design an efficient algorithm with good enough asymptotic complexities, but also to take into account the hardware limitations and preferences. In this work, we focus our design on two high level abstractions: work assignment and processing. The former denotes the assigned task by the programmer to each thread or group of threads. The latter encapsulates the actual execution of assigned tasks.

Previous work conflates work assignment and processing into similar granularities. The most traditional way is to have per-thread work assignment followed by per-thread processing of that assigned work. Each thread sequentially processes a part of input and then the results are combined appropriately. In this work, we use this approach in implementing various algorithms for the string matching problem (finding all instances of a pattern within a larger text). Another effective but less popular idea is per-warp work assignment followed by per-warp processing of that work. It usually requires efficient intra-warp communication to be able to efficiently process input data which is now distributed among all threads within that warp. With the emergence of warp-wide voting and shuffle instructions, this approach has gained more potential in solving particular problems efficiently and with some benefits compared to the per-thread assignment and processing. In this work, we use this approach to implement a series of parallel algorithms: histogram, multisplit and radix sort.

An advantage of using similar granularities for work assignment and processing is in problems with uniform per-thread or per-warp workloads, where it is quite easy to adapt warp-synchronous ideas and achieve high performance. However, with non-uniform irregular workloads, different threads might finish their processing in different times which can cause a sub-par performance. This is mainly because the whole warp continues to be resident in the device as long as all its

threads are finished. With these irregular tasks in mind, we propose to use different granularities for our work assignment and processing. We use a per-thread work assignment followed by a per-warp processing; each thread is still responsible for an independent task, but now all threads within a warp cooperate with each other to perform all these tasks together, one at a time, until all are successfully processed. Using this strategy, we design a dynamic hash table for the GPU, the *slab hash*, which is a totally concurrent data structure supporting asynchronous updates and search queries: threads may have different operations to perform and each might require an unknown amount of time to be fulfilled. By following our warp-cooperative strategy, all threads help each other perform these operations together, causing a much higher warp efficiency compared to traditional conflated work assignment and processing schemes.

ACKNOWLEDGMENTS

First and foremost, I want to thank John D. Owens, my Ph.D. advisor. I cannot express enough how much I owe John and how he has influenced me to be a better person, both academically and personally. Back in the Summer of 2013, I decided to make a change in my Ph.D. focus from electrical engineering to computer engineering. None of this chapter of my life would have happened if it were not for John, and how he trusted me, and how he patiently gave me the opportunity to learn and flourish. My respect and admiration for John is indescribable.

Furthermore, I owe a great deal to some of the best professors in my field. Prof. Nina Amenta, from the department of Computer Science at University of California, Davis, who if it were not for the amazing course she taught on "Parallel Algorithms", I would have not been introduced to my current field of study. Nina helped me as one of my dissertation committee members and advised me on many of my research projects and I thank her for all her support. Prof. Martin Farach-Colton, from the department of Computer Science at Rutgers University, who taught me a lot on theoretical aspects of data structures and advised me on many of my projects. Martin's support had a great influence on me and I cannot thank him enough for that.

I would also like to thank the rest of my dissertation committee members and my Ph.D. qualifying exam committee who advised me on my journey: Prof. Venkatesh Akella, Prof. Kent Wilken, and Prof. Chen-nee Chuah. I am honored to have such successful researchers evaluate my work. I would also thank Prof. Khaled Abdel-Ghaffar and Prof. Hussain Al-Assad who supported and helped me during my transition to join John's research group back in the Summer of 2013.

During my studies, I was lucky enough to be able to have many constructive discussions and helpful suggestions by experts from all over the field. I hereby thank Prof. Ulrich Meyer from Goethe-Universität Frankfurt am Main, Andrew Davidson at YouTube, Duane Merrill, Michael Garland and Anjul Patney from NVIDIA Research, Stephen Jones and Lars Nyland from NVIDIA, Mohamed Ebeida and Scott Mitchell at Sandia National Lab, and Sean Baxter.

Thanks to all the lovely people at the Department of Electrical and Computer Engineering at UC Davis, who helped me during my studies: Kyle Westbrook, Nancy Davis, Denise Christensen, Renee Kuehnau, Philip Young, Natalie Killeen, Yulia Suprun, Sacksith Ekkaphanh and many

more.

I enjoyed being a member of a bigger family in our research lab at UC Davis. I want to thank all who made me feel like being at home during all these years far from home, and all those who tolerated me at my best and worst days: Anjul Patney, Andrew Davidson, Pinar Muyan-Özçelik, Yangzihao Wang, Calina Copos, Afton Geil, Kerry Seitz, Leyuan Wang, Yuechao Pan, Collin McCarthy, Andy Riffel, Carl Yang, Weitang Liu, Muhammad Osama, Chenshan Shari Yuan, Shalini Venkataraman, Ahmed H. Mahmoud, Muhammad Awad, Yuxin Chen, Jason Mak, Vehbi Eşref Bayraktar, Edmund Yan, and Yudue Wu.

My research would not have been possible without the following financial support: UC Lab Fees Research Program Award 12-LR-238449, NSF awards CCF-1017399, OCI-1032859 and CCF-1637442, Sandia LDRD award #13-0144, a 2016–17 NVIDIA Graduate Fellowship, and a 2017 UC Davis Dissertation Writing Fellowship.

Chapter 1

Introduction

1.1 Overview

Graphics Processing Units (GPUs) are massively parallel devices with thousands of active threads. GPUs were initially designed to solely target graphics applications, which have a tremendous need for performing numerous more-or-less independent but less computationally expensive operations. GPUs were designed to fulfill such parallel tasks, with more emphasis on the number of operations done per fixed amount of time (maximizing throughput), rather than minimizing time required for each single operation (minimizing latency).

Researchers then began using the very same GPU hardware to perform other non-graphics general-purpose tasks. NVIDIA proposed the CUDA programming language to let programmers implement parallel algorithms on their GPUs as a parallel programming framework. However, implementing a high-performance algorithm on GPUs requires detailed knowledge of the hardware characteristics. Not only must a programmer design a well-designed algorithm with an acceptable theoretical asymptotic complexity, but the programmer must also pay attention to many practical issues, mostly based on the characteristics of the underlying GPU hardware, such as memory access patterns. Furthermore, some software capabilities may provide more options to the programmer as well (e.g., refer to Section 1.2).

These two factors are sometimes inconsistent with each other, where the most sophisticated algorithm with a very good asymptotic behavior might not result in the best performance as it cannot be programmed well enough to exploit all the available hardware resources. The best

high-performance results are usually crafted out of a compromise between theory and hardware characteristics, which as of today is a challenge to do on GPUs.

Though a complicated task, there has been many efforts to give high-level intuition to the efficient design of a GPU program. Some practical issues are more important than others. For example, as we discuss in Section 1.2, GPUs use numerous launched threads that are grouped into SIMD (single-instruction, multiple-data) units (called *warps*). These warps are automatically scheduled for execution on several streaming processors. Any instruction will eventually be run on these warps in a SIMD fashion. As a result, the programmer should avoid branching or else there will be a serialization in the execution. Similarly, memory accesses are physically done in a way that a warp can access a big block of memory (e.g., 4 bytes per thread, a total of 128 bytes in current GPUs). This is extremely important, since if the programmer requires the threads to access random arbitrary positions from the memory, each of those memory accesses requires a separate physical memory access. Yet, if the programmer would have used consecutive memory elements for consecutive threads, then a single physical memory access would be enough. This is usually known as a *coalesced* memory access, which is vital to an efficient GPU program.

Work assignment and processing: Intuitively, we can partition GPU computing problems into two separate phases: 1) work *assignment*, and 2) *processing*. The former encapsulates the input data distribution within the GPU. The latter focuses on the smallest independent group (a thread, or a group of threads like a warp or a thread-block) that actually performs the computation. Traditionally, these two abstractions are conflated together such that the same group (a thread, a warp, or a thread-block) that is assigned to a specific piece of work (i.e., specific portion of the input data) is also responsible for processing that same work. In dealing with regular workloads uniformly distributed among all assigned input portions, such a conflation between assignment and processing makes sense. There is no immediate incentive to depart from this high-level abstraction. However, for irregular workloads, we might be able to reach a better balance between the memory transactions and computations and overall to achieve improved performance by deliberately deviating from the one-to-one correspondence between assignment and processing phases. In such scenarios, we might be able to provide new opportunities by separating assignment from processing mostly to deliver better coalesced memory access and less

Approach	per-thread assignment and processing	per-warp assignment and processing	per-thread assignment, per-warp processing
Advantages	• good for uniform workloads • straightforward implementation and design • sequential local processing, no communications	• good for uniform workloads • fully coalesced memory accesses • warp-level privatization: less shared memory usage than per-thread methods.	• often coalesced memory accesses • good for irregular workloads • higher warp efficiency • reduced branch divergence
Disadvantages	• vectorized memory access • thread-level privatization: potentially more shared memory usage	• parallel processing requires warp-wide communications	• parallel processing requires warp-wide communications • requires all threads within a warp to be active (no intra-warp branches)
Examples	1. CUB's reduce, scan, histogram, and radix sort [66] 2. CUDPP's cuckoo hashing [3] 3. String matching (e.g., DRK, CRK, and HRK [5]) 4. Batched dynamic data structures (e.g., GPU LSM [9])	1. P-ary searches in B-trees and sorted lists [47]. 2. CUDA-based Reyes renderer [91]. 3. Ashkiani et al.'s multisplit, histogram, and radix sort [6, 7].	1. The verification stage in the DRK-2S string matching method [5]. 2. Fully concurrent dynamic data structures (e.g., the slab list and the slab hash [8]) 3. SlabAlloc: a warp-cooperative dynamic memory allocator [8]

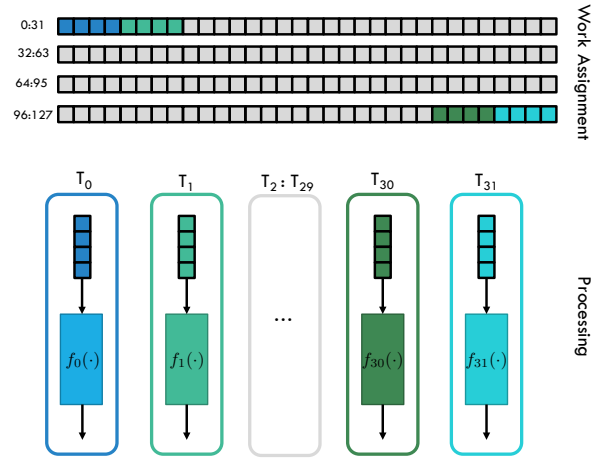
Table 1.1: Advantages, disadvantages and some examples for each pair of assignment-processing.

starvation due to code divergence (because of irregular workloads). Next, we elaborate on the existing conflated approaches and then propose our novel assignment-processing arrangements. Figure 1.1 shows some high-level schematic examples of the assignment-processing spectrum. Table 1.1 shows high-level aspects of each approach as well as some practical examples of the implementation of different algorithms under these approaches.

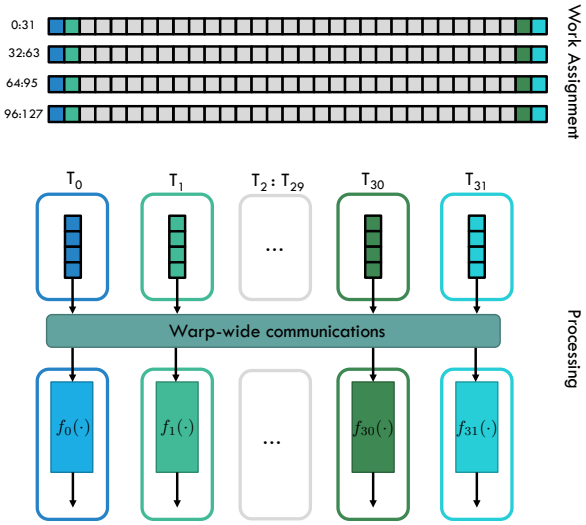
Per-thread assignment, per-thread processing: A traditional way of programming GPUs is to treat CUDA threads as generic processors (e.g., as in the parallel random-access machines (PRAM) model [45]). Each thread 1) is assigned to a portion of input data from the GPU's global memory (i.e., *per-thread* work assignment); 2) locally processes that portion (i.e., *per-thread* processing); and 3) appropriately stores its result so that it can be combined with others.[1] Figure 1.1a shows this approach in a problem with a regular workload.

This approach can be seen in most available GPU algorithms and implementations. To name a few, scan [68, 88], radix sort [69], merge sort [25], merge [11], histogram [19, 77, 89], and multisplit [41], are all implemented with the same strategy. There are some technical advantages

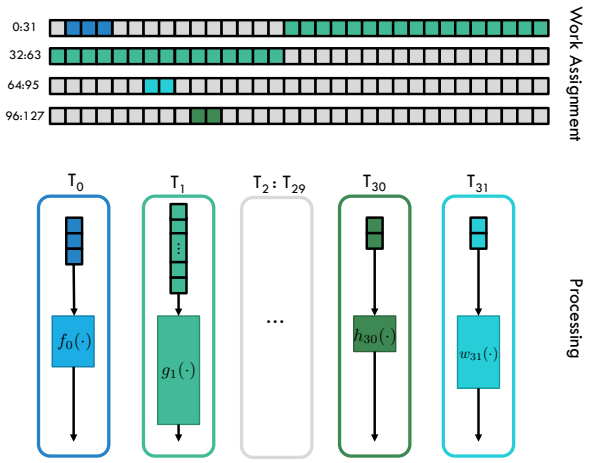
[1] In some algorithms the output does not have any dependence on the position of input data (i.e., any permutation of input results in the same output), e.g., performing a reduction with a binary associative operator. But in most algorithms, the output depends on the order of input data. As a result, each thread reads consecutive data elements to preserve locality and to locally process its portion (e.g., performing a prefix-sum operation). Throughout the rest of this work, we assume this most general case unless otherwise stated.



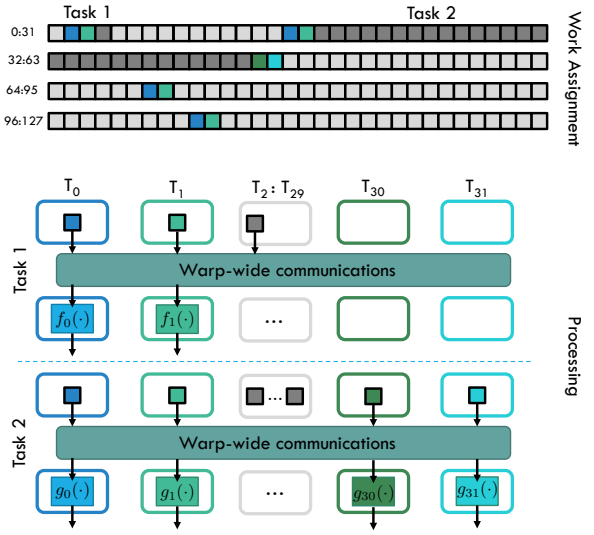
(a) Per-thread assignment, per-thread processing



(b) Per-warp assignment, per-warp processing



(c) Per-thread assignment, per-thread processing (irregular)



(d) Per-thread assignment, per-warp processing

Figure 1.1: Work assignment and processing: schematic demonstration. (a) and (b) represent regular workloads with conflated per-thread/per-warp assignment and processing. (c) represents an irregular workload with a conflated per-thread assignment and processing. (d) represents our proposed per-thread assignment, per-warp processing method to target irregular workloads. We have only considered 32 threads (a warp), and only 4 of these threads are thoroughly disambiguated through different colors. All gray elements, including light and dark, are placeholders. We have also only considered 128 data elements as input.

with this style of programming beyond its straightforward implementation: each thread is seen as a generic processor. Moreover, since threads process their assigned portion sequentially, all operations can be done in ALUs using register-level storage, which is the fastest memory resource on GPUs. If the workload is uniform, meaning that there is not any difference in the amount of processing required for each assigned workload (e.g., data-independent workloads), it is possible to use warp-synchronous programming ideas (by avoiding branches and extra synchronization barriers as much as possible) to better utilize available hardware resources.

There are some disadvantages as well. Since consecutive input elements are read by each thread, the memory access pattern is more complicated than we prefer for our GPU applications (vectorized access versus coalesced access). As a result, threads would first read every input portion for all threads within their own thread-block and store them into shared memory. After a synchronization barrier, they will access shared memory in the preferred vectorized format, but from a much faster shared memory. However, this approach requires storing intermediate results for each thread within a thread-block into shared memory, in order to be able to potentially combine those results together. In some scenarios, this might cause extra pressure on our shared memory usage (look at thread-level versus warp-level privatization in Section 3.5.8).

Per-warp assignment, per-warp processing An alternative to a per-thread work assignment is to assign work to larger groups of threads, for example a per-warp assignment approach. By using a larger portion of input data, there is naturally more locality to be extracted. But now, threads must synchronously cooperate with each other to perform the task, requiring extra shared memory/warp-wide communication.

Per-warp assignment naturally fits the coalesced memory access pattern that is preferred on GPUs. Since the data is distributed among the threads within a warp, there is less pressure on shared memory usage to store per-warp intermediate results (warp-level privatization in Section 3.5.8). Nevertheless, a disadvantage of this strategy is that the processing is supposed to be performed in parallel and within a warp, and as a result communication between threads is inevitable; this should either be performed through shared memory or by using warp-wide communication, either of which are more expensive than register-level accesses in traditional thread-level processing.

This approach is practiced rarely compared to the conflated per-thread assignment and processing, and part of the reason is that there should be a positive motivation behind tolerating extra intra-warp communication cost required processing. For example, Kaldewey and Di Blas study P-ary searches on B-trees that naturally suit larger processing groups and showed some promising results [47]. Tzeng et al. also used a warp-sized work granularity and processing for their CUDA-based Reyes renderer [91].

It is also customary to consider a conflated per-block assignment and processing. However, since there is currently no way for all threads within a block to communicate without using any synchronization barriers and through shared memory, such a per-block processing strategy is simply a combination of all the per-thread assignments and processing within that block. We believe that considering per-block scenarios as another group of conflated work assignment and processing does not add any new high level abstraction besides what already exists. So, we limit our abstraction to per-thread and per-warp pairs.

1.1.1 Our contributions:

Per-thread assignment, per-warp processing: The previous two strategies prove to be very effective in performing uniform workloads (i.e., each thread/warp performs relatively the same amount of work as others), each with their own advantages and disadvantages. But, they both suffer in dealing with irregular workloads, where neighboring processing units may be assigned different amounts of work. We propose a novel strategy of per-thread assignment and per-warp processing as follows, *Warp-Cooperative Work Sharing (WCWS)*. In WCWS, threads are still responsible for performing independent tasks. However, all tasks are put into a parallel work queue and processed one-by-one through cooperation among all threads within a warp. Processing continues until the work queue is empty, i.e., all assignments are fully processed.

We use the WCWS strategy to implement a dynamic hash table in Chapter 5 that supports fully concurrent operations (insertion/deletion/search) through the data structure [8]. In this case, since performing different operations might require a different amount of work, depending on the current data structure, the workload can be heavily irregular. In Chapter 2, our two-stage matching method (DRK-2S) also uses a similar approach to verify each potential match found by threads within a warp (Section 2.7.2). Here, since different threads can have various potential

matches (some do not even find any), then the overall workload is also irregular.

WCWS is a better match for irregular tasks in serializing the workload and cooperation among threads so that all threads are always actively helping rather than being stalled (thread starvation). If designed correctly, WCWS also better suits the coalesced memory access patterns that are preferred on GPUs.

As a disadvantage, all threads within a warp must be active to be able to cooperate; in other words no branching on threads is allowed. It also requires warp-wide communication within a warp, which is more expensive than register-level per-thread processing.

Our other contributions in this dissertation include:

1. In Chapter 2, we propose a series of string matching algorithms based on the Rabin-Karp matching method, suited for a single-pattern-single-text problem [5]. Our proposed methods either directly follow the per-thread assignment and per-thread processing approach (e.g., DRK in Section 2.4) or indirectly through using other traditional GPU primitives (e.g., Thrust's scan operation in CRK in Section 2.3.2).

2. In Chapter 3, we design and implement an efficient multisplit algorithm, an algorithm that takes an unordered set of elements and a user-defined bucket identifier (to assign labels/buckets to each element) and reorganizes input elements so that consecutive elements belong to the same bucket [6, 7]. Our multisplit implementation is based on per-warp assignment and per-warp processing, with extensive usage of warp-wide communications.

3. In Chapter 3, we also use the conflated per-warp assignment and per-warp processing approach to implement novel radix sort and histogram methods, based on our original multisplit design [6, 7].

4. In Chapter 4, we design and implement a dynamic dictionary data structure, GPU LSM, that supports bulk updates (insertions and deletions) as well as search, count and range queries [9]. Our design is based on traditional per-thread assignment and per-thread processing.

5. In Chapter 5, we design and implement a GPU-friendly linked list, called the slab list. We support concurrent updates such as insertions and deletions as well as search queries [8].

We provide an efficient implementation of a fully concurrent data structure (potentially an irregular workload itself) based on the per-thread assignment and per-warp processing approach.

6. By using the slab list, we implement a dynamic hash table for the GPU, the slab hash (Chapter 5).

7. We design a novel dynamic memory allocator, the SlabAlloc, to be used in the slab hash (Chapter 5).

As also summarized in Table 1.1, items 1 and 4 follow traditional per-thread assignment and per-thread processing. Items 2 and 3 follow per-warp assignment and per-warp processing. Items 5, 6 and 7 use per-thread assignment and per-warp processing (the WCWS strategy).

1.2 Preliminaries: the Graphics Processing Unit

The Graphics Processing Unit (GPU) is a throughput-oriented programmable processor. It is designed to maximize overall throughput, even by sacrificing the latency of sequential operations. Throughout this section and this work, we focus on NVIDIA GPUs and CUDA as our parallel computing framework. More details can be found in the CUDA programming guide [78]. Lindholm et al. [63] and Nickolls et al. [76] also provide more details on GPU hardware and the GPU programming model, respectively.

1.2.1 CUDA Terminology

In CUDA, the CPU is referred to as the *host* and all available GPUs are *devices*. GPU programs ("kernels") are launched from the host over a *grid* of numerous *blocks* (or thread-blocks); the GPU hardware maps blocks to available parallel cores (*streaming multiprocessors* (SMs)). The programmer has no control over scheduling of blocks to SMs; no programs may contain execution ordering assumptions. Each block typically consists of dozens to thousands of individual *threads*, which are arranged into 32-wide *warps*. CUDA v8.0 and all previous versions assume that all threads within warp execute instructions in lockstep (i.e., physically parallel). Hence, if threads within a warp fetch different instructions to execute (e.g., by using branching statements), then those instructions are serialized: similar instructions are executed in a SIMD fashion and one

at a time. This scenario is referred to as *branch divergence* and should be avoided as much as possible in order to reach high warp efficiency.

1.2.2 GPU Memory Hierarchy

GPUs possess several memory resources with organized into a memory hierarchy, each with different capacity, access time, and access scope (to be shared collectively). All threads have access to a global DRAM (*global memory*). For example, the Tesla K40c GPU has 12 GB of global memory available. This is the largest memory resource available in the device, but it is generally the slowest one as well, i.e., it requires the most instruction cycles to fetch specific memory contents. All threads within a thread-block have access to a faster but more limited *shared memory*. Shared memory is behaves like a manually controlled cache in the programmer's disposal, with orders of magnitude faster access time compared to global memory but much less available capacity (48 KB to be shared among all resident blocks per SM on a Tesla K40c). Shared memory contents are assigned to thread-blocks by the scheduler, and cannot be accessed by other thread-blocks (even those on the same SM). All contents are lost and the portion is re-assigned to a new block once a block's execution is finished. At the lowest level of the hierarchy, each thread has access to local registers. While registers have the fastest access time, they are only accessible by their own thread (64 KB to be shared among all resident blocks per SM on a Tesla K40c). On GPUs there are also other types of memory such as the L1 cache (on-chip 16 KB per SM, mostly used for register spills and dynamic indexed arrays), the L2 cache (on-chip 1.5 MB on the whole Tesla K40c device), constant memory (64 KB on a Tesla K40c), texture memory, and local memory [78].

On GPUs, memory accesses are done in units of 128 bytes. As a result, it is preferred that the programmer writes the program in such a way that threads within a warp require consecutive memory elements (*coalesced* memory access) to access 32 consecutive 4-byte units (such as float, integer, etc.). Any other memory access pattern results in a waste in the available memory bandwidth of the device.

1.2.3 Warp-wide communications

As we described before, in GPUs warps are the actual parallel units that operate in lockstep: all memory accesses and computations are done in SIMD fashion for all threads within a warp. CUDA also provides an efficient communication medium for all threads within a warp, without directly using any shared or global memory. Here we name the two most prominent types of warp-wide communication, and discuss each briefly. For a more detailed description refer to CUDA programming guide [78, Appendix B].

1.2.3.1 Warp-wide Voting:

There are a series of operations defined in CUDA so that threads can validate a certain binary predicate and share their results all together.

Any: `__any(pred)` returns true if there is at least one thread whose predicate is true.

All: `__all(pred)` returns true if all threads have their predicates validated as true.

Ballot: `__ballot(pred)` returns a 32-bit variable in which each bit represents its corresponding thread's predicate.

1.2.3.2 Warp-wide Shuffle:

The purpose of shuffle instructions is to read another thread's specific registers. It is particularly useful for broadcasting a certain value, or performing parallel operations such as reduction, scan, binary search, etc. within a single warp. There are four different types of shuffle instructions: (1) `__shfl`, (2)`__shfl_up`, (3) `__shfl_down`, (4)`__shfl_xor`.[2]

(1) is usually used for asking for the content of the specific register belonging to a thread. This can be any arbitrary thread, but we cannot ask for registers with dynamic indexing (i.e., register names should be known at compile time). For instance, in Section 3.5.5, histogram is computed within a warp so that each thread collects the results for specific buckets. Later when other threads need these results, they simply use shuffle instructions to ask for specific bucket counts from the corresponding responsible thread.

(2)–(4) are usually used when there is a fixed pattern of communication among the threads.

[2]Since CUDA 9.0, threads within a warp are not guaranteed to be in lockstep and there should be specific barriers to make sure all threads have reached a certain point in the program. As a result, all shuffle instructions are turned into their synchronized versions that have extra synchronization barriers (e.g., `__shfl_sync`) [79].

For example, warp-wide reduction can be computed using five rounds of `__shfl_xor` with 1, 2, 4, 8, and 16 values respectively. At each round, each thread computes its corresponding lane ID by XORing its own lane ID with a value and forming a butterfly network. In the end, all threads receive the final reduction result. Similarly, inclusive scan can be computed using five rounds of `__shfl_up/down` instructions. Here, each round values are added up/down with specific lane jumps. More details about these algorithm implementations can be found in the CUDA programming guide [78, Ch. B14]

1.2.4 CUDA Built-in intrinsics

CUDA provides several useful built-in integer intrinsics that are particularly useful for bitwise manipulations. We extensively use these intrinsics through all our implementations from Chapters 2–5, especially in order to process the result of ballot operations. Here we name a few examples, but more detailed descriptions and other intrinsics can be found in the CUDA Math API manual [80]:

Reverse bits: `__brev()` takes an unsigned integer and reverses its bits.

Number of high-order zero bits: `__clz()` takes an integer and returns the number of high-order zero bits before the first set bit. For example, the output of a 32-bit input variable is a value between 0 and 32. This instruction can also be used to find the most significant set bit of `x` (i.e., 32 - `__clz(x)`).

Finding the first set bit: `__ffs()` can be use to find the least significant set bit in an integer variable.

Population count: `__popc()` returns the total number of set bits in its input argument.

All these operations are provided in 32-bit and 64-bit versions. For 64-bit versions, there is a `ll` (i.e., long long) suffix added at the end (e.g., `__brevll()`).

Chapter 2

Parallel Approaches to the String Matching Problem on the GPU[1]

2.1 Introduction

Classic PRAM algorithms are proving to be a fertile source of ideas for GPU programs solving fundamental computational problems. But in practice, only certain PRAM algorithms actually perform well on GPUs. The PRAM model and the GPU programming model differ in important ways. Particularly important considerations suggest choosing algorithms where parallel threads perform uniform computation without branching or waiting ("uniformity"); where memory accesses across neighboring threads access neighboring memory locations ("coalescing"); and where the algorithm and memory accesses can take advantage of the computational and memory hierarchy of the modern GPU ("hierarchy"). Although these choices are usually the foremost considerations in GPU *implementations*, as in Schatz and Trapnell [86] and Lin et al. [62], they rarely play an important role in the process of *algorithm design*. In other words, they are considered optimization opportunities for programmers rather than desirable features of the algorithm itself.

In this chapter we address this issue through a case study. We consider the exact string matching problem, which is interesting in that it admits several different styles of parallelization. We focus on a classical PRAM algorithm of Karp and Rabin [49]. Forty years of research

[1]This chapter substantially appeared as "Parallel Approaches to the String Matching Problem on the GPU" published at SPAA 2016 [5], for which I was the first author and responsible for most of the research and writing.

into serial string matching has resulted in a plethora of elegant serial algorithms that perform very well in practice, but parallel algorithms are less well explored, particularly in the context of newer hierarchical parallel architectures like GPUs. We implement several variants of the Rabin-Karp algorithm (RK) on NVIDIA GPUs using the CUDA programming environment. Our experience leads us to believe that our general results apply to a broader set of hierarchical parallel processors, with either a compute hierarchy (multiple parallel processors, each with multiple parallel cores or functional units), a memory hierarchy (globally shared memory shared by all processors and faster locally shared memory of limited size shared by all cores within a processor), or both. Examples include other GPUs, accelerators such as the Intel Xeon Phi, and parallel multicore CPUs with vector instructions.

We begin with two main approaches to exploiting parallelism: *cooperative* and *divide-and-conquer*. In the former, all threads cooperate in computing a global solution, and in the latter, the main task is divided into smaller pieces, each piece is processed separately, and the results are combined. Rabin-Karp makes a good test case because it can be parallelized in both ways. In RK, the pattern of length m and all text substrings of length m are hashed (e.g., into a positive integer), and then the hashes are compared instead of the strings. Sting matching is inherently amenable to divide-and-conquer; the text can be divided into as many parallel sub-texts as needed, and the only drawback is due to overlap. RK is also amenable to cooperative parallelization, as described in the original 1987 paper, because with a wise choice of hash functions, the text can be hashed using a cooperative algorithm based on scan (aka parallel-prefix) operations. Fast implementation of scan operations has been one of the great successes in GPU software development for general computation. In both cases, the RK algorithm fits well with the GPU model in terms of uniformity, coalescing, and memory hierarchy. The downside of RK is that it always reads the entire text, while the best sequential algorithms are designed to skip as much as possible, so that they work especially well on "real data", such as natural language text, genomics data, and internet traffic.

We experimented extensively with two different parallel versions of RK, one with the cooperative strategy (CRK), and the other on using a simple divide-and-conquer strategy (DRK). We find that for short patterns (up to about 800 characters), DRK is much faster than leading

sequential codes running on an 8-core CPU. This is mostly because of its uniform load balance, relatively cheap operations, and high memory bandwidth due to coalesced memory accesses. The performance gap is especially large for very short patterns (e.g., 8 characters).

We then leverage the high performance of DRK's short-pattern matching to design a two-stage matching algorithm for longer patterns. It does a first round of searching on a small subset of the original pattern, and then groups of threads cooperate to verify all the potential matches. This new GPU-friendly algorithm outperforms the best multicore sequential codes for patterns of size up to almost 64k characters. This approach opens up some very interesting opportunities for skipping parts of the text and for doing approximate matching.

Our work provides both good news and bad news for parallel algorithm design. The good news is that parallel algorithms that pay attention to memory access patterns and load balancing can lead directly to high-performance GPU implementations. The bad news is that although scan operations can now be implemented very efficiently in the GPU environment, complex scans are still not necessarily competitive with more straightforward parallelization approaches.

We divide this chapter into the following parts:

- We propose three major RK-based binary matching algorithms: cooperative, divide-and-conquer, and a combination of both (hybrid). Using both theoretical demonstration and experimental validation, we address the following question: *for a given pattern and text size, what is the best method to use and why?* At a high level, we show that the divide-and-conquer approach is better for smaller patterns, while cooperative is superior for very large patterns (Sections 2.3–2.4).

- We provide several optimizations: both in terms of the general algorithm and computations, as well as some hardware specific ones (Section 2.6).

- We extend our RK-based algorithms to support characters from any general alphabets (Section 2.7.1).

- Using the divide-and-conquer approach, which is the fastest we have found for processing short patterns, we propose a novel two-stage matching method to process patterns of any

size efficiently. In this method, we first *skim* the text for a random small subset of the main pattern, and then *verify* all potential matches efficiently in parallel (Section 2.7.2).

2.2 Prior work

A naive approach to solve the single-pattern scenario is to compare all the possible strings of length m in our text with our pattern ($O(mn)$ time steps). There are several methods proposed so far to improve such quadratic running time. For instance, the KMP algorithm [53] saves work by jumping to the next possible index in case of a mismatch. The Boyer-Moore (BM) algorithm [16] is similar to KMP but starts its comparison from right to left. Some algorithms combine the existing tricks in other algorithms to enhance their performance. For example, the HASHq algorithm [57] hashes (similar to RK) all suffixes of shorter length (e.g., at most of length 8) and performs the matching on them. In case of a mismatch, like KMP, it shifts to the next potential match and like BM, starts comparing from the rightmost characters of each word. Faro and Lecroq's survey provides a comprehensive introduction and comparison between these and other methods [33].

Some of the earliest parallel work was from Galil [35] and Vishkin [93], who proposed a series of parallel string matching methods over a concurrent-read-concurrent-write PRAM memory model. The GPU literature features numerous papers that target related problems, but none of them address the fundamental problem of matching a single non-preprocessed pattern, implemented entirely on the GPU. Both Lin et al. [62] and Zha and Sahni [98] target multiple patterns (which is easier to parallelize) with an approach that leverages offline pattern compilation; the latter paper concludes that CPU implementations are still faster. Schatz and Trapnell used a CPU-compiled suffix tree for their matching [86]. Seamans and Alexander's virus scanner targeted multiple patterns with a two-step approach, but performed the second step on the CPU [87]. Kouzinopoulos and Margaritis [56] implement a brute-force non-preprocessed search and compare it to preprocessed-pattern methods.

2.3 Preliminaries

In this section we explain some of the basic algorithms and primitive parallel operations. Readers familiar with the serial RK algorithm (Section 2.3.1), and its parallel formulation (Section 2.3.2), can skip to Section 2.4.

In the basic scenario, $X = x_0 \dots x_{m-1}$ is a binary pattern of length m to be found in a binary text $Y = y_0 \dots y_{n-1}$ of length $n \geq m$. If $Y[r] = y_r y_{r+1} \dots y_{r+m-1}$, then our objective is to find all indices r such that $Y[r] = X$ for $0 \leq r < n - m + 1$. We will later generalize our results to arbitrary alphabets.

2.3.1 Serial Rabin-Karp

The main idea of the RK algorithm is to hash the pattern X and all the possible strings of length m in the text Y. The hashes are known as *fingerprints*, and any two strings with the same fingerprint match with high probability. Karp and Rabin proposed two families of hash functions [49]. The first produces integers as its fingerprints: for any $X \in \{0,1\}^m$: $F(X) = 2^{m-1}x_0 + \dots + 2x_{m-2} + x_{m-1}$. In order to avoid overflow, the result is computed modulo a random prime number $p \in (1, nm^2)$, so that $F_p(X) \stackrel{p}{\equiv} F(x)$. The second family of fingerprints is defined as follows. Define two matrices,

$$\mathbf{K}(0) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad \mathbf{K}(1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}. \tag{2.1}$$

Then for any binary string $X = x_0 \dots x_{m-1} \in \{0,1\}^m$, we define its fingerprint to be the product $\mathbf{K}(X) = \mathbf{K}(x_0) \dots \mathbf{K}(x_{m-1})$. Again to avoid overflows, the result is computed modulo a random prime number $p \in (1, mn^2)$ (i.e., $\mathbf{K}_p(X) \stackrel{p}{\equiv} \mathbf{K}(X)$). Regardless of which fingerprints we use, the RK algorithm can then be summarized as (1) choose a random prime number $p \in (1, mn^2)$; (2) compute all fingerprints (e.g., $\mathbf{F}_p(X)$, and $\mathbf{F}_p(Y[r])$ for $r \in [0, n - m + 1)$); and (3) compare: e.g., if $\mathbf{F}_p(Y[r])$ and $\mathbf{F}_p(X)$ are equal, then X and $Y[r]$ are equal with high probability, otherwise there is no match.

Step 2 can be done quite efficiently in a sequential manner [49]. In case we use the first class of fingerprints, $\mathbf{F}_p(Y[r+1])$ can be computed by using $\mathbf{F}_p(Y[r])$:

$$\mathbf{F}_p(Y[r+1]) \stackrel{p}{\equiv} 2\left(\mathbf{F}_p(Y[r]) - 2^{m-1}y_r\right) + y_{r+m+1}. \tag{2.2}$$

In case of the second class of fingerprints, we compute $\mathbf{K}_p(Y[r+1])$ based on $\mathbf{K}_p(Y[r])$ by a few operations:

$$\mathbf{K}_p(Y[r+1]) \stackrel{p}{\equiv} \mathbf{A}_p(y_r)\mathbf{K}_p(Y[r])\mathbf{K}_p(y_{r+m+1}), \tag{2.3}$$

where for any binary value $x \in \{0,1\}$, $\mathbf{A}_p(x)$ is defined as the left inverse of $\mathbf{K}_p(x)$ modulo p (i.e., $\mathbf{A}_p(x)\mathbf{K}_p(x) \stackrel{p}{\equiv} \mathbf{I}$ where $\mathbf{I}$ is an identity matrix). For each read text character, equation (2.2) requires two multiplications (each can be done by bitwise shift operation), two summations, and a modulo operation. Equation (2.3), on the other hand, requires 16 integer multiplications, 8 summations and 4 modulo operations per text character. As a result, the first class of fingerprints are significantly simpler to compute. We refer to sequential RK with first class of fingerprints as SRK algorithm. On the other hand, the second class of fingerprints can be updated using an associative operator, which can be leveraged in the following parallel formulation.

2.3.2 Cooperative Rabin-Karp

Karp and Rabin showed that the second class of fingerprints can be computed in parallel [49] using scan operations. It is well known that the *scan operation* takes a vector of input elements $\mathbf{u}$ and an associative binary operator $\oplus$, and returns an output vector $\mathbf{v}$ of the same size as $\mathbf{u}$. In exclusive (resp., inclusive) scan, $v_i = u_0 \oplus \cdots \oplus u_{i-1}$ (resp., 0 to i). A parallel scan on a vector of length n can be done with $O(\log n)$ depth and in $O(n)$ computations. With p processors, it is easy to show that a scan can be computed in $O(n/p + \log p)$ time steps. On the GPU, there are now highly optimized generic scan implementations that take an arbitrary associative operator as an input.

Let $\mathcal{K} = [\mathbf{K}_p(y_i)]_{i=0}^{n-1}$ and $\mathcal{A} = [\mathbf{A}_p(y_j)]_{j=0}^{n-m}$ represent vectors of all the required fingerprints and their inverses. We then define $\mathcal{S}$ and $\mathcal{T}$ vectors as follows:

$$\mathcal{S} = [\mathbf{K}_p(y_0), \mathbf{K}_p(y_0)\mathbf{K}_p(y_1), \ldots, \mathbf{K}_p(y_0)\ldots\mathbf{K}_p(y_{n-1})]\,,$$
$$\mathcal{T} = [\mathbf{I}, \mathbf{A}_p(y_0), \mathbf{A}_p(y_1)\mathbf{A}_p(y_0), \ldots, \mathbf{A}_p(y_{n-m})\ldots\mathbf{A}_p(y_0)]\,.$$

So $\mathcal{S}$ is an inclusive scan over vector $\mathcal{K}$ with right matrix multiplication modulo prime p as its associative operator. Similarly, $\mathcal{T}$ is an exclusive scan over vector $\mathcal{A}$ with left matrix multiplication modulo prime p as its associative operator. Then, it is clear that for $0 \leq r <$

$n - m + 1$:

$$\mathbf{K}_p(Y[r]) = \mathcal{T}[r]\mathcal{S}[r + m - 1] \tag{2.4}$$

$$= [\mathbf{A}_p(y_{r-1}) \dots \mathbf{A}_p(y_0)] \times$$

$$[\mathbf{K}_p(y_0) \dots \mathbf{K}_p(y_{r-1})\mathbf{K}_p(y_r) \dots \mathbf{K}_p(y_{r+m-1})] \tag{2.5}$$

$$= \mathbf{K}_p(y_r) \dots \mathbf{K}_p(y_{r+m-1}), \tag{2.6}$$

Thus, after computing two scans (i.e., computing $\mathcal{S}$ and $\mathcal{T}$), we can compute any required fingerprint by computing a single matrix multiplication (as in (2.4)). Based on Eq. (2.5), it is important to use different matrix multiplications (i.e., left/right) for each operator in order that each pair of matrices and their inverses correctly cancel each other out. Throughout this chapter, we refer to this method as Cooperative Rabin-Karp (CRK), because scan operations can be computed cooperatively by a set of independent threads or processors.

Our cooperative implementation uses this second fingerprint, but we should point out that it might be possible to improve the performance of the cooperative algorithm using a variant of the first fingerprint. Blelloch [15] points out, among many other things, that any sequential scan operation whose output can be represented by a recurrence of the form $x_i = a_i x_{i-1} + b_i$, where a_i and b_i are fixed input arrays and can be computed by a scan operation. In our case, $a_i = 2$, we have $b_i = y_i - 2^m y_{i-m}$, which can be computed in $O(1)$ from the input y, and all operations are done modulo p. Specialized to this situation, Blelloch's method is to define an associative operator $\cdot$ on pairs $c_i = (a_i, b_i)$, where $c_i \cdot c_j = (a_i a_j, b_i a_j + b_j)$. He proves that this is an associative operator on the set of pairs, and so it can be implemented using a scan operation. In addition to requiring many fewer operations than the second fingerprint, implementing this would only require one scan operation, not two. In practice, we have observed that this function produced more false positives due to hash collisions than the matrix-fingerprint, unless we used more expensive 64-bit operations. Thus our initial attempt to implement this did not lead to much of an improvement over our current CRK implementation. In any case, we expect neither approach would compete with our best methods, as we shall see in Section 2.8.

2.4 Divide-and-Conquer Strategy

Cooperative RK (Section 2.3.2) elegant exploits parallelism by having all of the processors cooperate through the two large scan operations. These parallel scans require intermediate communication between different processors and cores (threads from different blocks). However, CRK benefits little from the computational or memory hierarchy in its implementation (all operations are device-wide and global), so we also explored a straightforward *divide-and-conquer* approach. Dealing with smaller subproblems gives us a lot of flexibility in our design choices (e.g., work distribution among processors/cores, and exploiting the memory hierarchy), which allows us to tailor the computation to the GPU's strengths.

Divide and conquer is a natural approach to string matching; the easiest way to divide the work is to assign different parts of the text to different processors and process each part seperately. The final result is simply a union of matching results for each subproblem. In order to maintain independence between different subproblems, and to avoid extra communication between processors, this division should be done in an intelligent way. We must consider an overlap of length $m-1$ characters between consecutive subtexts to avoid losing any potential matches at the boundaries. Let L denote the number of subtexts, or equivalently, the granularity of the divide-and-conquer approach. Then, if we show each subtext as Y^ℓ, for $0 \leq \ell < L$, the division process can be shown as:

$$Y^\ell = \underbrace{y_{\ell g} y_{\ell g+1} \ldots y_{\ell g+g-1}}_{\text{exclusive characters}} \underbrace{y_{(\ell+1)g} \ldots y_{(\ell+1)g+m-1}}_{\text{overlapped characters}}, \tag{2.7}$$

where each subtext has $g = (n-m+1)/L$ *exclusive* characters, plus $m-1$ *overlapped* characters. For simplicity we assume that $(n-m+1)$ is a multiple of L, otherwise we can easily extend our text (increasing n with enough dummy characters) to be such a number. Now, each of these subtexts have $\bar{n} = g + m - 1$ characters and are independent from each other [2].

[2]This approach has commonality with stencil methods that operate over a large domain on hierarchical parallel machines: the domain is divided across cores and parallelism is exploited within a core. Typically, each core is assigned not only its own subdomain but also an overlapping region with its neighbors (the "halo"). Larger halos result in less communication at the cost of more redundant computation. In string matching, the overlap size is fixed, but the total overlap (i.e., total redundant work) is changed by the granularity of our division. Our objective will then be reaching a trade-off between minimizing redundant work (overlap) and maximizing achieved parallelism (maximizing the granularity).

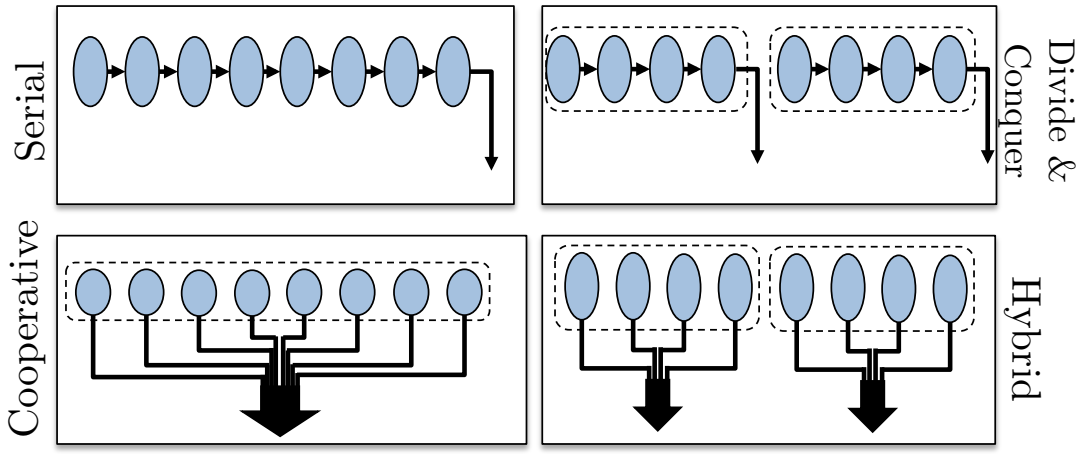


Figure 2.1: Schematic view of the four approaches considered in this chapter. The picture is simplified and shows the work distribution in each method. Dashed lines indicate subproblems.

In general, the possible cases for the number of subtexts is from $1 \leq L \leq n - m + 1$. We define *Divide-and-conquer RK* (DRK) as a method in which each processor performs the SRK on its own subtext individually. $L = 1$ corresponds to SRK applied to the whole text. We can also combine ideas of both the CRK and the DRK methods and form a *hybrid* method. We define *Hybrid RK* (HRK) as a method in which the main text is divided into subtexts (Eq. (2.7)) and then each subtext is assigned to a group of processors, which will perform CRK on it. If $L = 1$, HRK will be identical to CRK. Figure 2.1 shows our four possible formulations—serial, cooperative, divide-and-conquer, and hybrid—schematically.

Intuitively, a larger number of subtexts L (i.e., a finer granularity) yields more parallelism. At the same time, it means more potential overlaps and hence more redundant work because the total number of overlapped characters is $L(m - 1)$. One important question in this work is to determine the optimal choice of L depending on the method and other machine and problem characteristics.

2.4.1 Theoretical analysis with finite processors

In the following, we want to investigate each method's depth-work analysis under equal and finite number of p processors. We also assume that all processors have access to a globally shared memory.

We know that scan operations can be done with $O(n/p + \log p)$ depth complexity and $O(n)$

work. The CRK method requires two scan operations, followed by a constant round of final multiplications and comparisons, and $O(n)$ added work. Overall, the CRK requires $O(n)$ work and its running time will be

$$T_{\text{CRK}}(n,p) = O(n/p + \log p). \tag{2.8}$$

In the DRK method, each subtext is assigned to a processor. If we have $L > p$ subtexts of length $g + m - 1$, each subtext can be processed sequentially using SRK in $O(g + m - 1)$ time steps. Total work will be $O(n + Lm)$. Since at most p processors can be used at a time, $T_{\text{DRK}}(n,p)$ will be

$$T_{\text{DRK}}(n,p) = \frac{L}{p}O(g + m - 1) = O(\frac{n + Lm}{p}). \tag{2.9}$$

For the HRK, we assign each subtext to a group of processors, and each group runs CRK with its processors. Here we have a new degree of freedom: fewer groups of processors with more processors per group or vice versa. We assume p_1 groups and p_2 processors per group, for a total of $p = p_1 p_2$ processors. Subtexts are assigned to groups (p_1 at a time), while each subtext can be processed in parallel as in CRK with $O((g + m - 1)/p_2 + \log p_2)$. Overall, the running time $T_{\text{HRK}}(n,p)$ will be

$$\begin{aligned} T_{\text{HRK}}(n,p) &= \frac{L}{p_1}O(\frac{g + m - 1}{p_2} + \log p_2) \\ &= O(\frac{n + Lm}{p} + \frac{L}{p_1}\log p_2). \end{aligned} \tag{2.10}$$

For a fixed n, m, L and p, the runtime component $(L/p_1)\log p_2$ is minimized if $p_2 = 1$ and $p_1 = p$. It is interesting to note that in such case, HRK will be identical to the DRK method. For HRK, then, it is better to have more groups with fewer processors in each than fewer groups with more processors. Also, total work will be $O(n + Lm)$.

2.4.2 Theoretical Conclusions

Table 2.1 summarizes the step and work complexity for our three parallel methods with an equal number of processors. Although we have considered some simplifying assumptions in the above analysis, we can make some interesting general conclusions. First of all, since the sequential RK method is linear in time, as text size increases we expect to see an approximate linear increase in

Method	Step complexity	Work complexity
CRK	$O(n/p + \log p)$	$O(n)$
DRK	$O((n + mL)/p)$	$O(n + mL)$
HRK	$O((n + mL)/p + (L/p_1) \log p_2)$	$O(n + mL)$

Table 2.1: Step-work complexity for our methods with p processors (p_1 groups with p_2 processors in each).

the running time of all our parallel alternatives (because of the $O(n/p)$ term in Eq. (2.8), (2.9), and (2.10)).

As the pattern size m increases, more overlapped characters are included in DRK and HRK. These overlaps are redundant work and thus, there always exists a pattern size for which CRK is superior to the other algorithms, and for all larger patterns. This can also be seen in step complexities and the fact that CRK does not have any factor of pattern size in it. Also, with fixed number of processors (fixed p) and for large sizes of text and pattern our asymptotic analysis gets more accurate and hence we expect the DRK method to be asymptotically superior to the HRK method. However, for smaller text or pattern sizes, the order may differ.

2.5 Summary of Results

At this point in the chapter, we have introduced and analyzed three parallel algorithm formulations based on the RK algorithm: cooperative (CRK), divide-and-conquer (DRK), and hybrid (HRK). In Section 2.4.1 we analyzed the theoretical behavior of these algorithms and in Section 2.4.2, predicted their implementation behavior. Figure 2.2 shows the performance of these three algorithms, all implemented on an NVIDIA K40c GPU, with a fixed text size and varying pattern sizes. We observe that DRK is superior to the other methods for small patterns, but degrades as pattern size increases. Because CRK's performance is independent of pattern size, we note a crossover point where for all larger patterns the CRK method is superior. While HRK is better than CRK for small patterns, it is always inferior to the DRK method.

In the next three sections, we will build from the algorithms we detailed above, both analyzing and improving their performance as well as extending them to address practical concerns such as non-binary alphabets. We address the following issues:

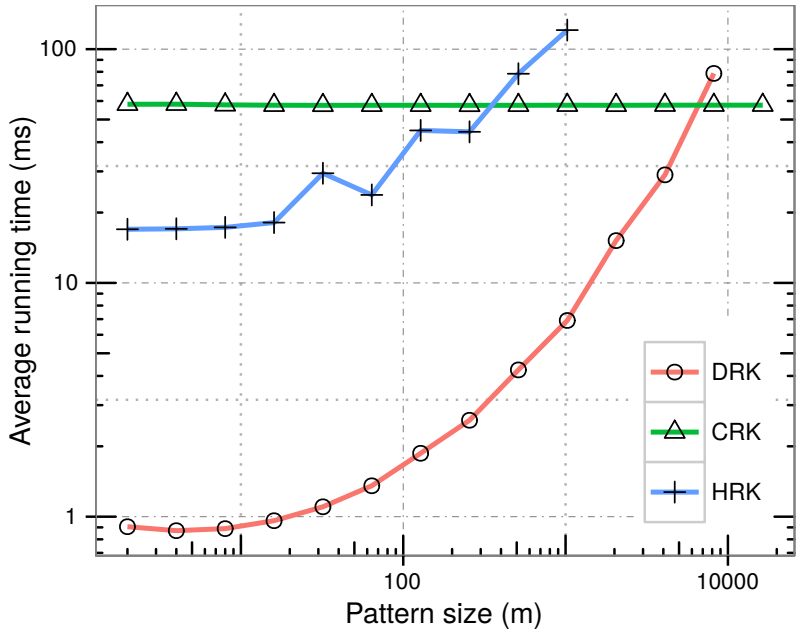


Figure 2.2: Average running time (ms) versus pattern size m, for a fixed binary text of size $n = 2^{25}$.

1. In Section 2.6, we provide implementation details for each of our parallel algorithms and describe a series of optimizations that improve their performance.

2. Karp and Rabin [49] only discussed binary alphabets and at this point in the chapter, so have we. In Section 2.7.1 we modify our algorithms to support any alphabet size.

3. Figure 2.2 demonstrates a large performance gap between DRK and CRK: DRK is up to 65x faster than CRK for small patterns. However, as pattern size increases, this performance gap narrows. We wish to preserve DRK's high performance for any pattern size and in Section 2.7.2, we propose a novel two-stage method (DRK-2S) based on the original DRK method that achieves this goal.

4. In Section 2.8 we thoroughly analyze the performance of all of our algorithms. Section 2.8.1 identifies the best parallel algorithm for any text and pattern size for binary alphabets. Then in Section 2.8.2, we compare our binary algorithms with the existing fastest CPU methods, and in Sections 2.8.3 and 2.8.4, extend this comparison for general alphabets. All our experiments up to this point are on randomly generated texts. In

Section 2.8.5, we expand our performance analysis to real-world text.

2.6 Implementation details

In the previous sections, we concentrated on our theoretical approach to parallel RK-based pattern matching. In this section, we dive into our implementation strategy. Since all our implementations are based on C, we assume that any character has a size of one byte; binary strings are handled as sequences of bytes. Throughout this section and beyond, some of our decisions we make are influenced by the CUDA programming model and GPU hardware.

2.6.1 Cooperative RK

For the CRK method we described in Section 2.3.2, we used the second class of fingerprints. We begin by mapping the input text to form the intermediate vectors $\mathcal{K}$ and $\mathcal{A}$. Then we perform two device-wide scan operations on these intermediate vectors to compute $\mathcal{S}$ and $\mathcal{T}$. In order to perform these scans, we use the scan operations provided by the Thrust library [13], with input and output stored in global memory and left/right matrix multiplications modulo a given prime number as their associative operators, respectively. All intermediate results are in the form of arrays of 2-by-2 integer matrices (as shown in Section 2.3.2): $\mathcal{K}$, $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{T}$ each requires 16B per input character. As a result, our CRK implementation not only uses the more computationally heavy fingerprints (second class), but also requires more global memory accesses per text character read.

2.6.2 Divide-and-Conquer RK

Our DRK implementations use the first class of fingerprints. Modulo operators do not have hardware implementations on current GPUs and hence are very slow compared to other operations. The only reason that we need this operator in equation (2.2) is to avoid overflows and hash the results; however, we can avoid these operations by making sure that an overflow never happens. For binary patterns of size less than or equal to 32 characters, we can use 32-bit registers (or 64-bit registers for $m \leq 64$) and guarantee no overflows. This change can significantly improve the performance for small patterns compared to the original updates. First, let us consider the most general case. Consider equation (2.2) and suppose we have previously computed 2^{m-1} modulo prime p and stored it in a register (called `base`). Then, if we denote y_r by `y_old`

and y_{n+r-1} by `y_new`, the updated fingerprint `F` will be `F` $\leftarrow$ `((F + p - R)` $\ll$ `1) + y_new) % p`, where `R` $\leftarrow$ `(base * y_old) % p`.

For 32-bit registers, if $p < 2^{30}$, then `R` and `F` never overflow (`base` is itself modulo `p`). Originally we should have `F-R` instead of `F+p-R`. But since `F` and `R` are both positive and less than `p`, we add a `p` value (neutral to modulo operator) to assure the result is positive before shifting ($0 \leq F + p - R < 2p$). The choice of the maximum of 30 bits is required because the result of both lines must not overflow before the modulo operator. Now, for patterns smaller than 32 characters we can have `F` $\leftarrow$ `((F & mask)` $\ll$ `1) + y_new`, where `mask` $\leftarrow$ $\sim$`(1` $\ll$ `m)`. Since we use the binary representation of characters as their fingerprints (exact hashing with theoretical zero collision), we can mask out the effect of y_r by an AND operation and avoid using an extra subtraction here. `mask` can be computed once, so on average we have one bitwise AND, one shift and one summation per character in this case. We refer to this version as DRK without modulo (DRK-WOM-32bit). With minimal (but non-zero) performance degradation, we can instead use 64-bit registers and support binary patterns up to 64 characters (DRK-WOM-64bit).

In DRK (or DRK-WOM), each thread is supposed to process $g+m-1$ consecutive characters from the text and verify whether or not there is a match for its g exclusive strings (Section 2.4). This is not an ideal memory access pattern for high-performance GPU implementation. We would instead prefer coalesced accesses, where consecutive threads within a warp access consecutive elements. To alleviate this problem, with N_T threads per block, we initially load N_T consecutive subtexts into each block's shared memory in a coalesced manner (total of gN_T+m-1 characters per block), and then each thread reads its own subtext locally from much faster shared memory (where coalescing is less important). In GPUs, memory accesses are in 4B alignments. Therefore we read elements as `char4` vectors (4 consecutive characters) to better exploit the available memory bandwidth[3].

2.6.3 Hybrid RK

As discussed in Section 2.4, HRK assigns each subtext to a group of processors, where a CRK method is performed on each subtext. On the GPU, we assign each subtext to a block, and threads

[3] `char4` is a built-in CUDA vector type with 4B alignment [78, Ch. B.3.].

within each block perform CRK on their own subtext. We use a block-wide scan operation from the CUB library [65]. In order to use CRK on each subtext, we need to store intermediate vectors in shared memory ($\mathcal{S}$ and $\mathcal{T}$ when using the second fingerprint). Current GPUs have very limited shared memory of at most 48 KB per block, so the extra storage forces us to either have fewer threads per block (small N_T) or very small subtext sizes (small g) and equivalently too many blocks (large L). In neither scenario are we utilizing our hardware's full potential, and consequently our HRK implementation compares poorly to our DRK- and CRK-based implementations.

2.7 Expanding beyond the simple RK

2.7.1 General alphabets

In their original paper, Karp and Rabin only considered binary alphabets [49]; we now extend this to general alphabets. Suppose we have an alphabet set Σ such that each character $a \in \Sigma$ requires at most $\sigma = \lceil \log \Sigma \rceil$ bits of information. Then, for any arbitrary pattern $X = x_0 \dots x_{m-1} \in \Sigma^m$, we can extend our first class of fingerprints to be $F_p(X) \stackrel{p}{\equiv} \sum_{i=0}^{m-1} x_i 2^{\sigma(m-i-1)}$. So, for this first class of fingerprints, the new update equation will be

$$F_p(Y[r+1]) \stackrel{p}{\equiv} 2^\sigma \left(F_p(Y[r]) - 2^{\sigma(m-1)} y_r \right) + y_{r+m+1}. \tag{2.11}$$

This update equation has a similar computational cost as equation (2.2). Its implementation will also be similar to what we outlined in Section 2.6.2. The `base` register we used in Section 2.6.2 is now precomputed to be $2^{\sigma(m-1)}$ modulo p. In order to avoid overflows, with the same reasoning as we had before, we should make sure that $p \leq 2^{31-\sigma}$. We also update our DRK-WOM implementation to incorporate σ: `F` $\leftarrow$ `((F & mask)` $\ll$ σ`) + y_new`, where `mask` $\leftarrow$ $\sim$`(((1`$\ll$ σ`) - 1)` $\ll$ `m)`. Here `mask` is intended to wipe out any possible remainder from the oldest added character. By using 64-bit registers, we can support pattern lengths of $m \leq 64/\sigma$, i.e., up to eight 8-bit ASCII characters.

For the second class of fingerprints, we first identify our fundamental fingerprints $\mathbf{K}_0 \dots \mathbf{K}_{2^\sigma - 1}$. Suppose we have an arbitrary character $z = c_{\sigma-1} \dots c_0$ where $c_i \in \{0, 1\}$. Then, $\mathbf{K}_z = \mathbf{K}_{c_{\sigma-1}} \times \cdots \times \mathbf{K}_{c_0}$. Inverse fingerprints are similarly defined as $\mathbf{A}_p(z) = \mathbf{A}_p(c_0) \times \cdots \times \mathbf{A}_p(c_{\sigma-1})$. These fundamental fingerprints and their inverses can either be precomputed and looked up from

a table at runtime or else can be computed on the fly. In either case, the amount of computation can become up to σ times more expensive and not any better than the binary CRK method, and hence for general alphabets, we prefer the first class of fingerprints and the DRK method.

2.7.2 Two-stage matching

For small patterns, our DRK implementations outperform our other methods on GPUs as well as the fastest CPU-based codes (Section 2.8 has more details). DRK-WOM, for instance, achieves up to a 66 GB/s processing rate. However, it works only for fairly small patterns (up to $64/\sigma$ characters if we use our 64-bit version). As the pattern size increases, DRK is the superior method up to almost 512 characters (for $1 \leq \sigma \leq 8$). However, DRK's performance degrades gradually as the pattern size increases for two reasons. 1) As m increases, the total number of characters that each thread must read until it can process g strings increases linearly ($g + m - 1$). 2) As m increases, the total number of characters stored in shared memory also increases ($gN_T + m - 1$) and hence our device occupancy (and overall performance) gradually decrease.

As we have discussed earlier in this chapter, the RK algorithm does not depend on the content of either the text or the pattern. This is a desirable property: the performance of RK implementations only depends on text and pattern sizes (n and m). We make no runtime decisions (e.g., branches) based on pattern or text content, which is beneficial for the uniformity goal we outlined in the introduction. In practice, most of our resources can be assigned statically at compile time (like the amount of required shared memory per block).

But an RK-based implementation processes every possible substring of length m in the text. Many well-known sequential matching algorithms (such as those introduced in Section 2.2) use different techniques to avoid this unnecessary work. In this subsection, we propose an algorithm that balances between the uniform workload of RK and avoids the unnecessary work of checking every possible substring.

Our *two-stage* DRK matching method (DRK-2S) first *skims* the text for a random small substring of the pattern and then *verifies* the potential matches that we identify. Suppose we search for a smaller substring of the pattern ("subpattern") of length $\bar{m} < m$. This subpattern can be chosen arbitrarily: it can be a prefix, a suffix, or any other arbitrary alignment from the main pattern. We then search our text using one of our DRK methods (preferably the DRK-WOM).

Each thread stores its potential matches, if any, into a local register-level stack. We note that any sequential method could be used for the skimming stage, but we focus on using one of our DRK methods because their implementations on GPUs balance load efficiently across the GPU.

For final verification, we expect that by choosing a large enough subpattern size, the expected number of potential matches will be much less than the text size n. As a result, we group threads together to verify potential matches altogether in parallel rather than individually and sequentially. In our implementation, we use a warp (32 threads) as the primary unit of verification. By doing so, we can have an efficient implementation by using warp-synchronous programming features and warp-wide voting schemes.

Initially we divide our pattern into several consecutive chunks of 32 characters. For each chunk, we compute the substring's fingerprint as $F_p(x_0 \dots x_{31}) \stackrel{p}{\equiv} \sum_{i=0}^{31} x_i 2^{\sigma(31-i)}$ by using a random prime number p as before. For example, for a pattern of length $m = 128$, we would compute 4 different fingerprints: $F_p(x_0 \dots x_{31}), \dots, F_p(x_{96} \dots x_{127})$. Now, we should verify whether for each potential match beginning at index r, $F_p(y_r \dots y_{r+31}), \dots, F_p(y_{r+96} \dots y_{r+127})$ are all equal to their pattern's counterparts.

After the skimming stage, each thread has a local stack of its own potential matches. Threads can find out about the existence of any non-empty stack within their warp (using a ballot). We start the verification based on threads' lane IDs (priority from 0 to 31). At this point all threads ask about the beginning index (say r) of that specific potential match (by using a shuffle) and then continue by coalesced reading of a chunk of 32 consecutive characters (from y_r until y_{r+31}). Then, by using at most five rounds of shuffles (performing a warp-wide reduction), we can compute the fingerprint corresponding to that chunk and compare it to that of the pattern's. In case it does not match, that specific potential match is not a final match and it is popped out of its thread's local stack. Otherwise, we continue to the next chunk of characters until all chunks are verified. The whole process is continued until all local stacks are empty. Algorithm 1 shows the high-level procedure of the DRK-2S.

We expect the DRK-2S method will significantly outperform the DRK method in dealing with large patterns, because we incur the cost of verifying a potential match (including reading m characters from the text) *only* when we first identify a subpattern. Consequently, we significantly

ALGORITHM 1: Two-stage matching method (DRK-2S)

for *each warp* w **parallel do**

  for *each thread* t *within* w **parallel do**

    Search for subpattern of length $\bar{m} < m$ using DRK $S_t \leftarrow$ potential matches $R_t \leftarrow$ beginning indices of S_t

  end

  for *each index* $r \in \bigcup_{t \in w}\{R_t\}$ **do**

    for $0 \leq k < \lceil m/32 \rceil$ **do**

      Compute $F \leftarrow F_p(y_{(r+32k)} \dots y_{(r+31+32k)})$ **if** F *is not equal to* $F_p(x_{32k} \dots x_{(32k+31)})$ **then**
        r is not a match

      end

    end

    if *all above* k *fingerprints matched* **then**
      r is a match

    end

  end

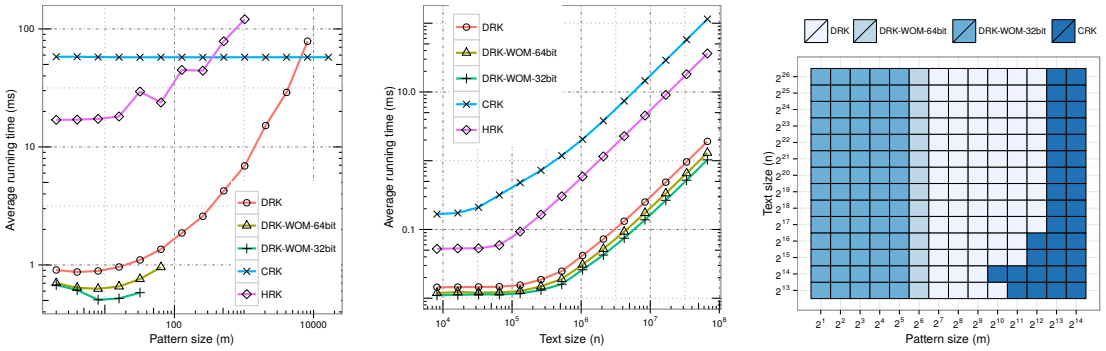
end

reduce the memory bandwidth requirements compared to DRK, and as we will see in the next section, achieve large speedups in practice.

2.8 Performance Evaluation

In this section, we experimentally assess the performance of all our proposed algorithms. Unless otherwise stated (Section 2.8.5), we use randomly generated texts and patterns with arbitrary alphabet sizes[4]. All our parallel methods are run on an NVIDIA Tesla K40c GPU, and compiled with NVIDIA's nvcc compiler (version 7.5.17). All our sequential codes are run on an Intel Xeon E5-2637 v2 3.50 GHz CPU, with 16 GB of DDR3 DRAM memory. We used OpenMP v3.0 to run our CPU codes in parallel over 8 cores (2 threads per core). All parallel GPU implementations are by the authors, while all CPU codes are from SMART library [32] and run in a divide-and-conquer fashion over our multi-core platform.

[4]Because the RK algorithm is independent of the content of the text or pattern, the runtime for any text or pattern for a fixed (m, n) will be identical.



(a) Avg. running time vs. m; $n = 2^{25}$ (b) Avg. running time vs. n; $m = 16$ (c) Superior Algorithms

Figure 2.3: Average running time vs. (a) pattern length (m) and (b) text length (n). (c) Superior algorithms in the input parametric space (m, n), where m is the pattern length and n is the text length. All GPU methods are implemented by the authors.

In this section, all our experiments are averaged over at least 200 independent random trials. All results in this section—average running times (ms) and processing rates (GB/s)—are measured without considering the transfer time (from disk or from CPU to GPU). In other words, for all GPU methods we assume that text is already stored on the off-chip DRAM memory of the device. As we will see shortly, our processing rates are indeed currently much faster than PCIe 3.0 transfer time. However, the recently announced CPU-GPU NVLink would allow us to eliminate this bottleneck by feeding the GPU at the rate of our fastest string matching implementations.

All our GPU codes are run with $N_T = 256$ threads per block. Based on the utilized number of registers per thread and also the size of shared memory per block, this amount of N_T has given us a very high occupancy ratio on our GPU device. Another important parameter in our divide-and-conquer methods is the total number of subtexts L, i.e., the total number of launched threads. Instead of optimizing for L, we instead optimize for $g = (n - m + 1)/L$, since it directly expresses the amount of work assigned to each thread, i.e., g is the total number of exclusive characters in each subtext. An advantage of RK-based methods is that by running a few tests on randomly generated texts and patterns, we can tune the optimum amount of g to fully exploit available GPU resources. Our simulations showed that the range of $12 \leq g \leq 44$ for $2 \leq m \leq 2^{16}$ achieves the best results.

2.8.1 Algorithm behavior in problem domains

In this part, we revisit the question of "which method is best suited for text size n and pattern size m?" We begin with a fixed text size n and varying pattern sizes m for a binary alphabet. Based on our earlier discussions, we expect all of our DRK family algorithms to perform better than CRK and HRK because of 1) significantly cheaper computations with the first class of fingerprints and 2) fewer and coalesced global memory accesses. All DRK methods only read text characters once and report results back to global memory only if necessary; however, CRK (and to some extent HRK) first process the text into 2-by-2 integer matrices and perform several device-wide computations over these intermediate vectors (requiring additional global memory operations). However, we also predicted that as pattern size becomes large, DRK methods fall behind CRK because of 1) excessive work due to overlap and 2) the limited capacity of available per-block shared memory in current GPUs.

Figure 2.3a shows the average running time over 200 independent random trials on texts with 32 M characters and various pattern sizes ($2 \leq m \leq 2^{16}$). In this simulation, the DRK methods are superior to CRK for $m < 2^{13}$. In addition, HRK performs better than CRK for $m \leq 256$ (because most intermediate operations are performed locally in each block), but it is never competitive to the DRK methods. All DRK methods have similar memory access behavior; the only difference is between their update costs. Consequently, on pattern sizes where they are applicable, DRK-WOM-32bit is better than DRK-WOM-64bit, and both are better than the regular DRK method, because they avoid expensive modulo operators.

We expected that all parallel algorithms would show linear performance with respect to text size (summarized in Table 2.1). Figure 2.3b depicts average running time versus varying text sizes and a fixed pattern size $m = 16$ for binary characters. This expected linear behavior is evident after we reach a certain text size, which is large enough to fully exploit GPU resources under each scenario's algorithmic and implementation limitations.

Let's turn from the above snapshots from the (m, n) problem domain to the overall behavior of our algorithms across the whole problem domain. Figure 2.3c shows the superior algorithms across the (m, n) space. As we expect, the DRK family is superior for small patterns, and the CRK method for larger patterns, but the crossover point differs based on the text size. Within the

		Pattern size (m)				
	Algorithm	4	16	64	256	1024
CPU	SRK	0.95	1.11	1.15	1.16	1.16
	SA	**1.57**	3.41	4.1	4.09	4.01
	AOSO2	0.80	4.25	4.53	4.54	4.54
	HASH5	—	**4.69**	5.25	5.68	5.63
	HASH8	—	4.62	**5.93**	**6.33**	**5.98**
GPU	CRK	0.58	0.58	0.58	0.58	0.58
	DRK	38.5	34.91	24.75	**12.98**	**4.86**
	DRK-WOM-32bit	**54.84**	**64.50**	—	—	—
	DRK-WOM-64bit	52.40	51.07	**35.06**	—	—
Speedup (GPU/CPU)		34.9x	13.8x	5.9x	2.1x	0.8x

Table 2.2: Processing rate (throughput) in GB/s. Texts and patterns are randomly chosen using a binary alphabet.

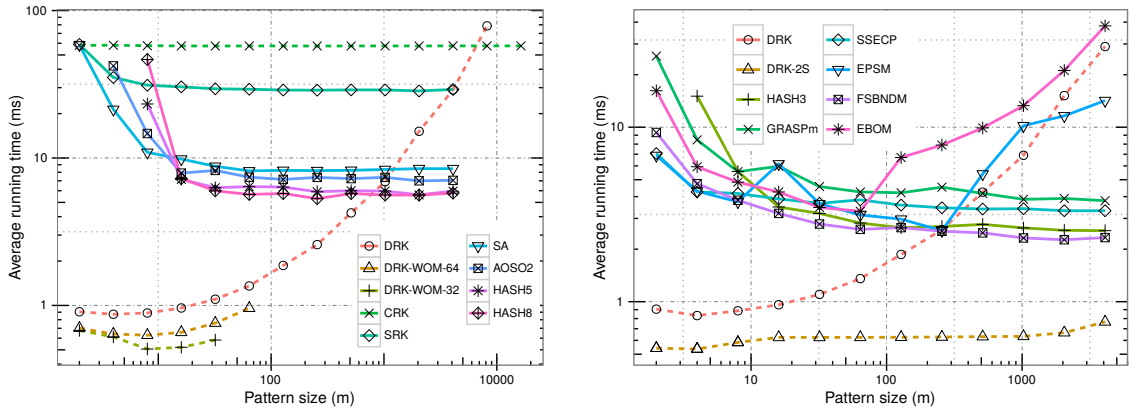
DRK family, the DRK-WOM-32bit has the cheapest computational cost and is hence dominant. However, it can only be used for patterns with $m \leq 32$ characters. The next best choice is the DRK-WOM-64bit, which supports up to $m \leq 64$ characters. For larger patterns we should choose the regular DRK method, per Section 2.6.2.

2.8.2 Binary sequential algorithms

Now that we have studied the competitive behavior of our parallel algorithms with each other, we compare their performance with the best sequential binary matching algorithms. Faro and Lecroq's comprehensive survey [33] covers the fastest sequential methods. We begin with available source codes from the SMART library [32][5] and parallelize them in a divide-and-conquer fashion (as described in Section 2.4) but distributed over 8 CPU cores (total of 16 CPU threads). We compile and run the results in parallel using OpenMP v3.0 with –O3 optimizations. Figure 2.4a shows the average running time of the best CPU implementations against our parallel algorithms implemented on the GPU.

We note that the CPU algorithms perform better as the pattern size increases. Most of these algorithms use different techniques to avoid unnecessary processing of text based on the content of read characters and the pattern (as we also leverage in our DRK-2S from Section 2.7.2). As

[5]We compare against all top-performing algorithms from Faro and Lecroq's survey [33] and SMART library [32] except for SSEF, which in our experiments did not deliver competitive performance.



(a) Binary alphabet ($\sigma = 1$)

(b) 8-bit alphabet ($\sigma = 8$)

Figure 2.4: Average running time (ms) versus pattern size m with fixed text size $n = 2^{25}$ for (a) $\sigma = 1$ and (b) $\sigma = 8$. Solid and dashed lines correspond to CPU and GPU implementations respectively.

the pattern size increases, this strategy avoids more work on average (the methods can take larger jumps through the text). On the other hand, as we noted previously, our DRK method degrades as the pattern size increases. As a result, our DRK family of implementations is superior to all binary CPU methods for patterns up to almost 1024 characters. Table 2.2 reports processing rate (throughput) for some specific pattern sizes measured over random texts with 2^{25} binary characters.

2.8.3 General sequential algorithms

Except for very minor low-level differences, our DRK implementations with any non-binary alphabet should perform as well as our binary DRK implementation. Figure 2.4b shows some of the fastest matching algorithms for $\sigma = 8$ (based on the survey from Faro and Lecroq [33]). As with our binary sequential comparison, we parallelize codes from the SMART library [32] across 8 CPU cores/16 threads. For the DRK-2S method, we used the DRK-WOM-64bit as our matching method for the skimming stage. Here, since we have $\sigma = 8$, we can use up to $\bar{m} = 8$ characters for our subpatterns.

Table 2.3 summarizes our achieved processing rates on random texts with 2^{25} 8-bit characters. Our DRK-2S method achieves a geometric mean speedup of 4.81x over the fastest CPU methods. Although we consider patterns up to 1024 characters in this table, the DRK-2S continues to be

		Pattern size (m)				
	Algorithm	4	16	64	256	1024
CPU	SA	3.96	4.20	4.92	4.91	5.26
	HASH3	2.24	9.69	11.16	12.89	12.08
	HASH5	–	7.22	10.29	11.63	12.21
	HASH8	–	7.33	11.56	10.69	12.18
	FJS	4.27	5.80	7.13	6.95	7.77
	SBNDM-BMH	4.82	5.22	6.90	7.03	7.80
	GRASPm	3.89	5.69	7.32	7.01	8.25
	FS	4.01	6.19	7.22	6.70	8.05
	SSECP	**7.66**	9.43	9.32	9.71	9.87
	EPSM	7.58	5.34	10.35	12.08	3.27
	FSBNDM	7.71	**11.00**	**12.55**	**12.92**	**14.59**
	EBOM	5.49	8.04	9.48	4.18	2.59
GPU	DRK	40.31	34.94	24.75	12.98	4.86
	DRK-2S	**62.68**	**53.77**	**53.73**	**53.51**	**53.04**
Speedup (GPU/CPU)		8.18x	4.89x	4.28x	4.14x	3.63x

Table 2.3: Processing rate (throughput) in GB/s. Text and patterns are randomly chosen with alphabet $\sigma = 8$ (8-bit ASCII characters).

superior to all the CPU methods up to patterns of almost 64k characters. For example, with 32k characters DRK-2S achieves 9.19 GB/s while the best CPU method, the FSBNDM, achieves 5.89 GB/s. The main reason behind our performance degradation for very large patterns is an increased number of memory accesses for each potential match ($O(m/32)$ accesses). We could potentially address this issue by modifying DRK-2S to consider chunks of 128 characters (instead of 32) and use `char4` memory accesses in its verification stage as well; this would reduce the number of accesses by a factor of 4. However, for realistic pattern sizes this modification would not significantly improve our performance.

2.8.4 Dependency on alphabet size

In the last two parts, we have evaluated our parallel GPU methods against the best CPU methods for two cases of binary characters (Section 2.8.2) and general 8-bit characters (Section 2.8.3). In this section, we focus on the DRK-2S method and continue our discussion over randomly generated texts over various alphabet sizes ($\Sigma = 2^\sigma$). Table 2.4 shows the processing rate of the DRK-2S for $1 \leq \sigma \leq 8$ and $4 \leq m \leq 1024$. For the skimming stage, we use a subpattern length

	Pattern size (m)						
σ	4	8	16	32	64	256	1024
1	52.40	56.55	55.04	49.96	49.86	49.74	49.23
2	59.98	58.91	55.75	51.04	51.03	51.01	50.15
3	60.57	58.94	55.86	51.13	51.11	50.91	50.45
4	60.58	58.85	55.78	51.09	51.01	50.71	49.82
5	60.59	58.88	53.47	53.52	53.44	53.38	52.76
6	60.63	58.94	52.36	53.46	53.42	53.30	52.29
7	60.59	58.87	53.50	53.45	53.42	53.18	52.11
8	62.68	63.66	53.77	53.69	53.73	53.51	53.04

Table 2.4: Processing rate (throughput) in GB/s of our DRK-2S method for various alphabet sizes ($\Sigma = 2^{\sigma}$) and different pattern lengths (m).

of $\bar{m} = 16$ for $1 \leq \sigma \leq 4$ and $\bar{m} = 8$ otherwise.

It is interesting to note that with a fixed pattern size, our performance increases as the alphabet size increases. This is mainly due to the fact that for randomly generated texts and with a fixed pattern size, as alphabet size increases, we get less likely to visit a match in our skimming stage (probability of $\sigma^{-\bar{m}}$), and hence fewer verifications are required in the second stage. It is also interesting to note the sudden performance degradation for $m > \bar{m}$ with each alphabet size, which is directly related to whether or not we have to perform the verification stage. For $m > \bar{m}$ as the pattern size increases, we witness a very mild rate decrease mainly due to the number of times that a consecutive number characters are verified (line 8 in Alg. 1). Overall, as we discussed earlier, we do not expect our performance to be affected much by change in alphabet size (evident in update equation (2.11)), which is also clear from this table as well.

2.8.5 Real-world scenarios

So far, we have only used randomly generated texts and patterns for our experiments, because all of our RK-based methods have performance independence of the content of the text and the pattern. For DRK-2S, however, the total number of verifications directly depends on the number of potential matches found in the skimming stage. As a result, the algorithm is no longer data-independent. In this part, we consider some real-world scenarios for evaluation:

Natural language text: Table 2.5 shows the average running time (ms) of the DRK-2S method as well as all the high-performing CPU algorithms from Section 2.8.3 in finding several

	Pattern size (m)						
Algorithm	4	8	16	32	64	256	1024
SA	16.68	17.23	17.51	17.19	15.08	16.20	15.67
HASH3	20.95	10.97	6.29	5.75	**4.67**	4.98	**3.77**
HASH5	–	14.56	7.40	5.99	5.32	4.96	5.77
HASH8	–	49.15	10.00	5.97	5.46	5.57	5.31
FJS	12.98	9.80	7.95	6.50	6.31	7.58	4.82
SBNDM-BMH	10.19	9.88	7.73	6.53	6.22	6.16	6.78
GRASPm	13.17	9.31	7.04	6.41	6.20	6.78	6.60
FS	13.06	9.58	7.43	5.73	6.06	6.38	6.52
FSBNDM	8.51	6.31	5.21	**4.92**	4.97	4.61	5.16
EBOM	7.55	6.17	**5.05**	5.42	5.25	5.49	6.47
SSECP	5.35	6.97	6.27	6.23	6.99	7.05	5.81
EPSM	**5.05**	**5.15**	5.69	5.43	4.75	**3.78**	6.64
DRK	2.46	2.62	2.84	3.25	3.99	7.59	20.34
DRK-2S	**1.58**	**1.63**	**1.88**	**1.84**	**1.84**	**1.84**	**1.83**
Speedup	3.2x	3.16x	2.69x	2.67x	2.54x	2.05x	2.06x

Table 2.5: Average running time (ms) for finding patterns of different sizes (m) on a sample text from Wikipedia with 100 M characters. The last row reports the achieved speedup of our DRK-2S method over the best CPU method.

patterns of different sizes in a 100 MB sample text from Wikipedia[6]. Here DRK-2S achieves a geometric mean speedup of 2.59x against the best CPU codes.

DNA sequences: Table 2.6 shows the average running time (ms) of regular DRK and DRK-2S methods as well as some of the best sequential algorithms over the E. coli genome sequence[7]. This is a text with almost 4.6 million characters drawn from four different possible characters ($\Sigma = 4$). The DRK-2S method achieves a geometric mean speedup of 5.45x against the best CPU codes.

Protein sequences: Table 2.7 shows the average running time (ms) of our DRK-based methods compared to the best CPU methods over processing of the Homo sapiens protein sequence, with 3.3 million characters and alphabet of 19 characters. The DRK-2S method achieves a geometric mean speedup of 6.88x against the best CPU codes.

[6]The enwik8 dataset includes the first 100 MB text from Wikipedia as of March 2006: `http://prize.hutter1.net/`.

[7]Data available in SMART library: `http://www.dmi.unict.it/~faro/smart/corpus.php`

	Pattern size (m)						
Algorithm	4	8	16	32	64	256	1024
SA	1.10	1.01	1.03	1.01	0.98	0.98	1.00
HASH3	1.35	0.80	0.60	**0.52**	0.49	**0.46**	0.52
HASH5	–	1.05	0.60	0.54	**0.44**	0.49	0.45
HASH8	–	2.74	0.73	0.54	0.45	0.52	**0.44**
FJS	1.83	1.51	1.55	1.47	1.64	1.59	1.50
SBNDM-BMH	1.32	1.01	0.79	0.56	0.58	0.57	0.65
GRASPm	1.36	1.09	1.07	0.83	0.74	0.71	0.67
FS	1.37	1.16	1.07	1.02	0.90	0.82	0.72
FSBNDM	1.24	0.88	0.64	0.56	0.54	0.58	0.61
EBOM	0.96	0.92	0.74	0.65	0.61	0.82	1.21
SSECP	**0.52**	**0.50**	0.74	0.65	0.66	0.65	0.65
EPSM	0.54	0.76	**0.50**	0.61	0.50	0.48	0.67
DRK	0.162	0.121	0.137	0.175	0.264	0.855	0.954
DRK-2S	**0.075**	**0.077**	**0.093**	**0.093**	**0.093**	**0.093**	**0.098**
Speedup	6.93x	6.49x	5.38x	5.59x	4.73x	4.94x	4.49x

Table 2.6: Average running time (ms) for finding patterns of different sizes (m) on E. coli genome with 4.6M characters. The last row reports the achieved speedup of our DRK-2S method over the best CPU method.

2.8.6 False positives

The RK algorithm, and hence all the proposed methods in this article so far, are randomized algorithms whose randomness comes from our choice of prime number and hashing process. While RK algorithms will always be successful in finding matches, it is possible to have false positives, i.e., multiple strings that hash to the same value. In the original RK algorithm, it is shown that if the chosen prime number is less than $n(n-m+1)^2$, then the probability of error is upper-bounded by $2.511/(n-m+1)$ [49]. Even by using 64-bit variables and operations, that leaves us with a non-zero probability of false positives. On the other hand, all our DRK-WOM methods assign a unique fingerprint to their strings, and hence false positives are not possible. Our DRK-2S has a great advantage here. First of all, by using a DRK-WOM method in the skimming stage, we significantly reduce the total number of potential matches. Secondly, DRK-2S divides the pattern into multiple smaller-sized chunks of size 32 and hashes them with a random prime number. So for a pattern to result in a false positive, all of its chunks

	Pattern size (m)						
Algorithm	4	8	16	32	64	256	1024
SA	0.84	0.84	0.84	0.88	0.78	0.78	0.78
HASH3	1.18	0.67	0.49	**0.43**	0.44	0.44	0.43
HASH5	–	0.86	0.60	0.49	0.46	0.47	0.50
HASH8	–	1.96	0.57	0.44	0.46	**0.43**	0.44
FJS	0.72	0.63	0.61	0.54	0.49	0.51	0.56
SBNDM-BMH	0.76	0.68	0.56	0.44	0.50	0.49	0.48
GRASPm	0.88	0.68	0.55	0.55	0.56	0.48	0.45
FS	0.79	0.60	0.54	0.51	0.46	0.46	0.49
FSBNDM	0.62	0.47	**0.45**	0.50	**0.41**	0.46	**0.43**
EBOM	0.51	0.48	0.53	0.55	0.53	0.63	1.46
SSECP	**0.45**	**0.41**	0.49	0.49	0.51	0.48	0.49
EPSM	0.56	0.53	0.55	0.50	0.46	0.53	0.66
DRK	0.080	0.086	0.098	0.125	0.188	0.609	0.684
DRK-2S	**0.053**	**0.055**	**0.065**	**0.065**	**0.065**	**0.065**	**0.071**
Speedup	8.49x	7.45x	6.92x	6.61x	6.31x	6.62x	6.01x

Table 2.7: Average running time (ms) for finding patterns of different sizes (m) on the Homo sapiens protein sequence with 3.3M characters. The last row reports the achieved speedup of our DRK-2S method over the best CPU method.

should independently hash to the same values as the original pattern's. So the probability is exponentially decreased. We did not encounter any false positives in our experiments with the DRK-2S. Still, we could arbitrarily decrease the false positive probability even more by hashing each chunk with a different prime number (or even using multiple prime numbers and multiple hashing per chunk). This would increase our computational load in the verification stage, but since the total number of potential matches after the skimming stage is significantly fewer than n, the impact on the overall performance would be negligible.

2.9 Conclusion

Our final outcome from this work is a novel two-stage algorithm that addresses the string matching problem. The DRK-2S algorithm fully exploits GPU computational and memory resources in its skimming stage, where we efficiently process the whole text to identify potential matches. We then test these potential matches using cooperative groups of threads. The result is a highly efficient string matching algorithm for patterns of any size, which we believe to be

the fastest string-matching implementation on any commodity processor for pattern sizes up to nearly 64k characters. Though very efficient, our string matching method can only be used for *exact* matching of a *single* pattern (or at least very few). We hope that this algorithm and the presented ideas embodied in it can be used as a cornerstone for other high-throughput string processing applications such as in regular expressions and approximate matching algorithms, as well as for multiple-pattern matching scenarios where a text is matched against a dictionary of patterns (e.g., in intrusion detection systems).

Chapter 3

GPU Multisplit[1]

3.1 Introduction

This chapter studies the multisplit primitive for GPUs.[2] Multisplit divides a set of items (keys or key-value pairs) into contiguous buckets, where each bucket contains items whose keys satisfy a programmer-specified criterion (such as falling into a particular range). Multisplit is broadly useful in a wide range of applications, some of which we will cite later in this introduction. But we begin our story by focusing on one particular example, the delta-stepping formulation of single-source shortest path (SSSP).

The traditional (and work-efficient) serial approach to SSSP is Dijkstra's algorithm [31], which considers one vertex per iteration—the vertex with the lowest weight. The traditional parallel approach (Bellman-Ford-Moore [10]) considers all vertices on each iteration, but as a result incurs more work than the serial approach. On the GPU, the recent SSSP work of Davidson et al. [26] instead built upon the delta-stepping work of Meyer and Sanders [70], which on each iteration classifies candidate vertices into *buckets* or *bins* by their weights and then processes the bucket that contains the vertices with the lowest weights. Items within a bucket are unordered and can be processed in any order.

Delta-stepping is a good fit for GPUs. It avoids the inherent serialization of Dijkstra's approach and the extra work of the fully parallel Bellman-Ford-Moore approach. At a high level,

[1] This chapter substantially appeared in two of our papers: "GPU Multisplit" published at PPoPP 2016 [6] and "GPU Multisplit: an extended study of a parallel algorithm" published at ACM Transactions on Parallel Computing (TOPC) [7]. For both papers, I was was responsible for most of the research and writing.

[2] The source code is available at `https://github.com/owensgroup/GpuMultisplit`.

delta-stepping divides up a large amount of work into multiple buckets and then processes all items within one bucket in parallel at the same time. How many buckets? Meyer and Sanders describe how to choose a bucket size that is "large enough to allow for sufficient parallelism and small enough to keep the algorithm work-efficient" [70]. Davidson et al. found that 10 buckets was an appropriate bucket count across their range of datasets. More broadly, for modern parallel architectures, this design pattern is a powerful one: expose just enough parallelism to fill the machine with work, then choose the most efficient algorithm to process that work. (For instance, Hou et al. use this strategy in efficient GPU-based tree traversal [42].)

Once we've decided the bucket count, how do we efficiently classify vertices into buckets? Davidson et al. called the necessary primitive *multisplit*. Beyond SSSP, multisplit has significant utility across a range of GPU applications. Bucketing is a key primitive in one implementation of radix sort on GPUs [69], where elements are reordered iteratively based on a group of their bits in their binary representation; as the first step in building a GPU hash table [3]; in hash-join for relational databases to group low-bit keys [30]; in string sort for singleton compaction and elimination [29]; in suffix array construction to organize the lexicographical rank of characters [28]; in a graphics voxelization pipeline for splitting tiles based on their descriptor (dominant axis) [82]; in the shallow stages of k-d tree construction [95]; in Ashari et al.'s sparse-matrix dense-vector multiplication work, which bins rows by length [4]; and in probabilistic top-k selection, whose core multisplit operation is three bins around two pivots [73]. And while multisplit is a crucial part of each of these and many other GPU applications, it has received little attention to date in the literature. The work we present here addresses this topic with a comprehensive look at efficiently implementing multisplit as a general-purpose parallel primitive.

The approach of Davidson et al. to implementing multisplit reveals the need for this focus. If the number of buckets is 2, then a scan-based "split" primitive [40] is highly efficient on GPUs. Davidson et al. built both a 2-bucket ("Near-Far") and 10-bucket implementation. Because they lacked an efficient multisplit, they were forced to recommend their theoretically-less-efficient 2-bucket implementation:

> The missing primitive on GPUs is a high-performance *multisplit* that separates

> primitives based on key value (bucket id); in our implementation, we instead use a sort; in the absence of a more efficient multisplit, we recommend utilizing our Near-Far work-saving strategy for most graphs. [26, Section 7]

Like Davidson et al., we could implement multisplit on GPUs with a sort. Recent GPU sorting implementations [69] deliver high throughput, but are overkill for the multisplit problem: unlike sort, multisplit has no need to order items within a bucket. In short, sort does more work than necessary. For Davidson et al., reorganizing items into buckets after each iteration with a sort is too expensive: "the overhead of this reorganization is significant: on average, with our bucketing implementation, the reorganizational overhead takes 82% of the runtime." [26, Section 7]

In this chapter we design, implement, and analyze numerous approaches to multisplit, and make the following contributions:

- On modern GPUs, "global" operations (that require global communication across the whole GPU) are more expensive than "local" operations that can exploit faster, local GPU communication mechanisms. Straightforward implementations of multisplit primarily use global operations. Instead, we propose a parallel model under which the multisplit problem can be factored into a sequence of local, global, and local operations better suited for the GPU's memory and computational hierarchies.

- We show that reducing the cost of global operations, even by significantly increasing the cost of local operations, is critical for achieving the best performance. We base our model on a hierarchical divide and conquer, where at the highest level each subproblem is small enough to be easily solved locally in parallel, and at the lowest level we have only a small number of operations to be performed globally.

- We locally reorder input elements before global operations, trading more work (the reordering) for better memory performance (greater coalescing) for an overall improvement in performance.

- We promote the warp-level privatization of local resources as opposed to the more traditional thread-level privatization. This decision can contribute to an efficient implementation

of our local computations by using warp-synchronous schemes to avoid branch divergence, reduce shared memory usage, leverage warp-wide instructions, and minimize intra-warp communication.

- We design a novel voting scheme using only binary ballots. We use this scheme to efficiently implement our warp-wide local computations (e.g., histogram computations).
- We use these contributions to implement a high-performance multisplit targeted to modern GPUs. We then use our multisplit as an effective building block to achieve the following:
  - We build an alternate radix sort competitive with CUB (the current fastest GPU sort library). Our implementation is particularly effective with key-value sorts (Section 3.7.1).
  - We demonstrate a significant performance improvement in the delta-stepping formulation of the SSSP algorithm (Section 3.7.2).
  - We build an alternate device-wide histogram procedure competitive with CUB. Our implementation is particularly suitable for a small number of bins (Section 3.7.3).

3.2 Related Work and Background

3.2.1 Parallel primitive background

In this chapter we leverage numerous standard parallel primitives, which we briefly describe here. A *reduction* inputs a vector of elements and applies a binary associative operator (such as addition) to reduce them to a single element; for instance, sum-reduction simply adds up its input vector. The *scan* operator takes a vector of input elements and an associative binary operator, and returns an output vector of the same size as the input vector. In exclusive (resp., inclusive) scan, output location i contains the reduction of input elements 0 to $i-1$ (resp., 0 to i). Scan operations with binary addition as their operator are also known as *prefix-sum* [40]. Any reference to a multi- operator (multi-reduction, multi-scan) refers to running multiple instances of that operator in parallel on separate inputs. *Compaction* is an operation that filters a subset of its input elements into a smaller output array while preserving the order.

3.2.2 Multisplit and Histograms

Many multisplit implementations, including ours, depend heavily on knowledge of the total number of elements within each bucket (bin), i.e., histogram computation. Previous competitive GPU histogram implementations share a common philosophy: divide the problem into several smaller sized subproblems and assign each subproblem to a thread, where each thread sequentially processes its subproblem and keeps track of its own *privatized* local histogram. Later, the local histograms are aggregated to produce a globally correct histogram. There are two common approaches to this aggregation: 1) using atomic operations to correctly add bin counts together (e.g., Shams and Kennedy [89]), 2) storing per-thread sequential histogram computations and combining them via a global reduction (e.g., Nugteren et al. [77]). The former is suitable when the number of buckets is large; otherwise atomic contention is the bottleneck. The latter avoids such conflicts by using more memory (assigning exclusive memory units per-bucket and per-thread), then performing device-wide reductions to compute the global histogram.

The hierarchical memory structure of NVIDIA GPUs, as well as NVIDIA's more recent addition of faster but local shared memory atomics (among all threads within a thread block), provides more design options to the programmer. With these features, the aggregation stage could be performed in multiple rounds from thread-level to block-level and then to device-level (global) results. Brown et al. [19] implemented both Shams's and Nugteren's aforementioned methods, as well as a variation of their own, focusing only on 8-bit data, considering careful optimizations that make the best use of the GPU, including loop unrolling, thread coarsening, and subword parallelism, as well as others. Recently, NVIDIA's CUDA Unbound (CUB) [67] library has included an efficient and consistent histogram implementation that carefully uses a minimum number of shared-memory atomics to combine per-thread privatized histograms per thread-block, followed by aggregation via global atomics. CUB's histogram supports any data type (including multi-channel 8-bit inputs) with any number of bins.

Only a handful of papers have explored multisplit as a standalone primitive. He et al. [41] implemented multisplit by reading multiple elements with each thread, sequentially computing their histogram and local offsets (their order among all elements within the same bucket and processed by the same thread), then storing all results (histograms and local offsets) into memory.

Next, they performed a device-wide scan operation over these histogram results and scattered each item into its final position. Their main bottlenecks were the limited size of shared memory, an expensive global scan operation, and random non-coalesced memory accesses.[3]

Patidar [84] proposed two methods with a particular focus on a large number of buckets (more than 4k): one based on heavy usage of shared-memory atomic operations (to compute block level histogram and intra-bucket orders), and the other by iterative usage of basic binary split for each bucket (or groups of buckets). Patidar used a combination of these methods in a hierarchical way to get his best results.[4] Both of these multisplit papers focus only on key-only scenarios, while data movements and privatization of local memory become more challenging with key-value pairs.

3.3 Multsiplit and Common Approaches

In this section, we first formally define the multisplit as a primitive algorithm. Next, we describe some common approaches for performing the multisplit algorithm, which form a baseline for the comparison to our own methods, which we then describe in Section 3.4.

3.3.1 The multisplit primitive

We informally characterize multisplit as follows:

- Input: An unordered set of keys or key-value pairs. "Values" that are larger than the size of a pointer use a pointer to the value in place of the actual value.
- Input: A function, specified by the programmer, that inputs a key and outputs the bucket corresponding to that key (*bucket identifier*). For example, this function might classify a key into a particular numerical range, or divide keys into prime or composite buckets.
- Output: Keys or key-value pairs separated into m buckets. Items within each output bucket must be contiguous but are otherwise unordered. Some applications may prefer output

[3] On an NVIDIA 8800 GTX GPU, for 64 buckets, He et al. reported 134 Mkeys/sec. As a very rough comparison, our GeForce GTX 1080 GPU has 3.7x the memory bandwidth, and our best 64-bucket implementation runs 126 times faster.

[4] On an NVIDIA GTX280 GPU, for 32 buckets, Patidar reported 762 Mkeys/sec. As a very rough comparison, our GeForce GTX 1080 GPU has 2.25x the memory bandwidth, and our best 32-bucket implementation runs 23.5 times faster.

order within a bucket that preserves input order; we call these multisplit implementations "stable".

More formally, let $\mathbf{u}$ and $\mathbf{v}$ be vectors of n *key* and *value* elements, respectively. Altogether m buckets $B_0, B_1, \ldots, B_{m-1}$ partition the entire key domain such that each key element uniquely belongs to one and only one bucket. Let $\boldsymbol{f}(\cdot)$ be an arbitrary bucket identifier that assigns a bucket ID to each input key (e.g., $\boldsymbol{f}(u_i) = j$ if and only if $u_i \in B_j$). Throughout this chapter, m always refers to the total number of buckets. For any input key vector, we define *multisplit* as a permutation of that input vector into an output vector. The output vector is densely packed and has two properties: (1) All output elements within the same bucket are stored contiguously in the output vector, and (2) All output elements are stored contiguously in a vector in ascending order by their bucket IDs. Optionally, the beginning index of each bucket in the output vector can also be stored in an array of size m. Our main focus in this chapter is on 32-bit keys and values (of any data type).

This multisplit definition allows for a variety of implementations. It places no restrictions on the order of elements within each bucket before and after the multisplit (intra-bucket orders); buckets with larger indices do not necessarily have larger elements. In fact, key elements may not even be comparable entities, e.g., keys can be strings of names with buckets assigned to male names, female names, etc. We do require that buckets are assigned to consecutive IDs and will produce buckets ordered in this way. Figure 3.1 illustrates some multisplit examples. Next, we consider some common approaches for dealing with non-trivial multisplit problems.

3.3.2 Iterative and Recursive scan-based splits

The first approach is based on binary split. Suppose we have two buckets. We identify buckets in a binary flag vector, and then compact keys (or key-value pairs) based on the flags. We also compact the complemented binary flags from right to left, and store the results. Compaction can be efficiently implemented by a scan operation, and in practice we can concurrently do both left-to-right and right-to-left compaction with a single scan operation.

With more buckets, we can take two approaches. One is to iteratively perform binary splits and reduce our buckets one by one. For example, we can first split based on B_0 and all remaining buckets ($\cup_{j=1}^{m-1} B_j$). Then we can split the remaining elements based on B_1 and $\cup_{j=2}^{m-1} B_j$. After

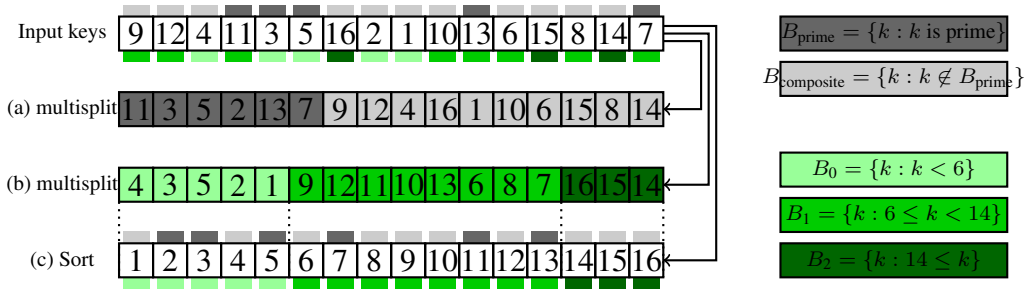


Figure 3.1: Multisplit examples. (a) Stable multisplit over two buckets (B_{prime} and $B_{\text{composite}}$). (b) Stable multisplit over three range-based buckets (B_0, B_1, B_2). (c) Sort can implement multisplit over ordered buckets (e.g., for B_0, B_1, B_3), but not for any general buckets (e.g., B_{prime} and $B_{\text{composite}}$); note that this multisplit implementation is not stable (initial intra bucket orders are not preserved).

m rounds the result will be equivalent to a multisplit operation. Another approach is that we can recursively perform binary splits; on each round we split key elements into two groups of buckets. We continue this process for at most $\lceil \log m \rceil$ rounds and in each round we perform twice number of multisplits and in the end we will have a stable multisplit. Both of these scan-based splits require multiple global operations (e.g., scan) over all elements, and may also have load-balancing issues if the distribution of keys is non-uniform. As we will later see in Section 3.6.1, on modern GPUs and with just two buckets this approach is not efficient enough.

3.3.3 Radix sort

It should be clear by now that sorting is not a general solution to a multisplit problem. However, it is possible to achieve a non-stable multisplit by directly sorting our input elements under the following condition: if buckets with larger IDs have larger elements (e.g., all elements in B_0 are less than all elements in B_1, and so on). Even in this case, this is not a work-efficient solution as it unnecessarily sorts all elements within each bucket as well. On average, as the number of buckets (m) increases, this performance gap should decrease because there are fewer elements within each bucket and hence less extra effort to sort them. As a result, at some point we expect the multisplit problem to converge to a regular sort problem, when there are large enough number of buckets.

Among all sorting algorithms, there is a special connection between radix sort and multisplit. Radix sort iteratively sorts key elements based on selected groups of bits in keys. The process

either starts from the least significant bits ("LSB sort") or from the most significant bits ("MSB sort"). In general MSB sort is more common because, compared to LSB sort, it requires less intermediate data movement when keys vary significantly in length (this is more of an issue for string sorting). MSB sort ensures data movements become increasingly localized for later iterations, because keys will not move between buckets ("bucket" here refers to the group of keys with the same set of considered bits from previous iterations). However, for equal width key types (such as 32-bit variables, which are our focus in this chapter) and with a uniform distribution of keys in the key domain (i.e., an equivalently uniform distribution of bits across keys), there will be less difference between the two methods.

3.3.4 Reduced-bit sort

Because sorting is an efficient primitive on GPUs, we modify it to be specific to multisplit: here we introduce our *reduced-bit sort* method (RB-sort), which is based on sorting bucket IDs and permuting the original key-value pairs afterward. For multisplit, this method is superior to a full radix sort because we expect the number of significant bits across all bucket IDs is less than the number of significant bits across all keys. Current efficient GPU radix sorts (such as CUB) provide an option of sorting only a subset of bits in keys. This results in a significant performance improvement for RB-sort, because we only sort bucket IDs (with $\log m$ bits instead of 32-bit keys as in a full radix sort).

Key-only In this scenario, we first make a *label* vector containing each key's bucket ID. Then we sort (label, key) pairs based on label values. Since labels are all less than m, we can limit the number of bits in the radix sort to be $\lceil \log m \rceil$.

Key-value In this scenario, we similarly make a label vector from key elements. Next, we would like to permute (key, value) pairs by sorting labels. One approach is to sort (label, (key, value)) pairs all together, based on label. To do so, we first pack our original key-value pairs into a single 64-bit variable and then do the sort.[5] In the end we unpack these elements to form the final results. Another way is to sort (label, index) pairs and then manually permute key-value

[5]For data types that are larger than 32 bits, we need further modifications for the RB-sort method to work, because it may no longer be possible to pack each key-value pair into a single 64-bit variable and use the current already-efficient 64-bit GPU sorts for it. For such cases, we first sort the array of indexes, then manually permute the arbitrary sized key-value pairs.

pairs based on the permuted indices. We tried both approaches and the former seems to be more efficient. The latter requires non-coalesced global memory accesses and gets worse as m increases, while the former reorders for better coalescing internally and scales better with m.

The main problem with the reduced-bit sort method is its extra overhead (generating labels, packing original key-value pairs, unpacking the results), which makes the whole process less efficient. Another inefficiency with the reduced-bit sort method is that it requires more expensive data movements than an ideal solution. For example, to multisplit on keys only, RB-sort performs a radix sort on (label, key) pairs.

Today's fastest sort primitives do not currently provide APIs for user-specified computations (e.g., bucket identifications) to be integrated as functors directly into sort's kernels; while this is an intriguing area of future work for the designers of sort primitives, we believe that our reduced-bit sort appears to be the best solution today for multisplit using current sort primitives.

3.4 Algorithm Overview

In analyzing the performance of methods from the previous section, we make two observations:

1. Global computations (such as a global scan) are expensive, and approaches to multisplit that require many rounds, each with a global computation, are likely to be uncompetitive. Any reduction in the cost of global computation is desirable.

2. After we derive the permutation, the cost of permuting the elements with a global scatter (consecutive input elements going into arbitrarily distant final destinations) is also expensive. This is primarily because of the non-coalesced memory accesses associated with the scatter. Any increase in memory locality associated with the scatter is also desirable.

The key design insight in this chapter is that we can reduce the cost of both global computation and global scatter at the cost of doing more local work, and that doing so is beneficial for overall performance. We begin by describing and analyzing a framework for the different approaches we study in this chapter, then discuss the generic structure common to all our implementations.

3.4.1 Our parallel model

Multisplit cannot be solved by using only local operations; i.e., we cannot divide a multisplit problem into two independent subparts and solve each part locally without any communication between the two parts. We thus assume any viable implementation must include at least a single global operation to gather necessary global information from all elements (or group of elements). We generalize the approaches we study in this chapter into a series of N rounds, where each round has 3 stages: a set of local operations (which run in parallel on independent subparts of the global problem); a global operation (across all subparts); and another set of local operations. In short: {local, global, local}, repeated N times; in this chapter we refer to these three stages as {prescan, scan, postscan}.

The approaches from Section 3.3 all fit this model. Scan-based split starts by making a flag vector (where the local level is per-thread), performing a global scan operation on all flags, and then ordering the results into their final positions (thread-level local). The iterative (or recursive) scan-based split with m buckets repeats the above approach for m (or $\lceil \log m \rceil$) rounds. Radix sort also requires several rounds. Each round starts by identifying a bit (or a group of bits) from its keys (local), running a global scan operation, and then locally moving data such that all keys are now sorted based on the selected bit (or group of bits). In radix sort literature, these stages are mostly known as up-sweep, scan and down-sweep. Reduced-bit sort is derived from radix sort; the main differences are that in the first round, the label vector and the new packed values are generated locally (thread-level), and in the final round, the packed key-value pairs are locally unpacked (thread-level) to form the final results.

3.4.2 Multisplit requires a global computation

Let's explore the global and local components of stable multisplit, which together compute a unique permutation of key-value pairs into their final positions. Suppose we have m buckets $B_0, B_1, \ldots, B_{m-1}$, each with $h_0, h_1, \ldots, h_{m-1}$ elements respectively ($\sum_i h_i = n$, where n is the total number of elements). If $u_i \in B_j$ is the ith element in key vector $\mathbf{u}$, then its final permuted

position $p(i)$ should be (from u_i's perspective):

$$p(i) = \underbrace{\sum_{k=0}^{j-1} h_k}_{\text{global offset}} + \underbrace{\left|\{u_r \in B_j : r < i\}\right|}_{\text{local offset } (u_i\text{'s bucket})}, \tag{3.1}$$

where $|\cdot|$ is the cardinality operator that denotes the number of elements within its set argument. The left term is the total number of key elements that belong to the preceding buckets, and the right term is the total number of preceding elements (with respect to u_i) in u_i's bucket, B_j. Computing both of these terms in this form and for all elements (for all i) requires global operations (e.g., computing a histogram of buckets).

3.4.3 Dividing multisplit into subproblems

Equation (3.1) clearly shows what we need in order to compute each permutation (i.e., final destinations for a stable multisplit solution): a histogram of buckets among all elements (h_k) as well as local offsets for all elements within the same bucket (the second term). However, it lacks intuition about how we should compute each term. Both terms in equation (3.1), at their core, answer the following question: to which bucket does each key belong? If we answer this question for every key and for all buckets (hypothetically, for each bucket we store a binary bitmap variable of length n to show all elements that belong to that bucket), then each term can be computed intuitively as follows: 1) histograms are equal to counting all elements in each bucket (reduction of a specific bitmap); 2) local offsets are equivalent to counting all elements from the beginning to that specific index and within the same bucket (scan operation on a specific bitmap). This intuition is closely related to our definition of the scan-based split method in Section 3.3.2. However, it is practically not competitive because it requires storing huge bitmaps (total of mn binary variables) and then performing global operations on them.

Although the above solution seems impractical for a large number of keys, it seems more favorable for input problems that are small enough. As an extreme example, suppose we wish to perform multisplit on a single key. Each bitmap variable becomes just a single binary bit. Performing reduction and scan operations become as trivial as whether a single bit is set or not. Thus, a divide-and-conquer approach seems like an appealing solution to solve equation (3.1): we would like to divide our main problem into small enough subproblems such that solving each

subproblem is "easy" for us. By an easy computation we mean that it is either small enough so that we can afford to process it sequentially, or that instead we can use an efficient parallel hardware alternative (such as the GPU's ballot instruction). When we solve a problem directly in this way, we call it a *direct solve*. Next, we formalize our divide-and-conquer formulation.

Let us divide our input key vector $\mathbf{u}$ into L contiguous subproblems: $\mathbf{u} = [\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{L-1}]$. Suppose each subvector $\mathbf{u}_\ell$ has $h_{0,\ell}, h_{1,\ell}, \ldots, h_{m-1,\ell}$ elements in buckets $B_0, B_1, \ldots B_{m-1}$ respectively. For example, for arbitrary values of i, s, and j such that key item $u_i \in \mathbf{u}_s$ and u_i is in bucket B_j, equation (3.1) can be rewritten as (from u_i's perspective):

$$p(i) = \underbrace{\overbrace{\sum_{k=0}^{j-1}}^{\text{previous buckets}} \left(\sum_{\ell=0}^{L-1} h_{k,\ell} \right) + \overbrace{\sum_{\ell=0}^{s-1}}^{u_i\text{'s bucket}} h_{j,\ell}}_{\text{global offset}} + \underbrace{\left| \{ u_r \in \mathbf{u}_s : (u_r \in B_j) \wedge (r < i) \} \right|}_{\text{local offset within } u_i\text{'s subproblem}}. \tag{3.2}$$

This formulation has two separate parts. The first and second terms require global computation (first: the element count of all preceding buckets across all subproblems, and second: the element count of the same bucket in all preceding subproblems). The third term can be computed locally within each subproblem. Note that equation (3.1) and (3.2)'s first terms are equivalent (total number of previous buckets), but the second term in (3.1) is broken into the second and third terms in (3.2).

The first and second terms can both be computed with a global histogram computed over L local histograms. A global histogram is generally implemented with global scan operations (here, exclusive prefix-sum). We can characterize this histogram as a scan over a 2-dimensional matrix $\mathbf{H} = [h_{i,\ell}]_{m \times L}$, where the "height" of the matrix is the bucket count m and the "width" of the matrix is the number of subproblems L. The second term can be computed by a scan operation of size L on each row (total of m scans for all buckets). The first term will be a single scan operation of size m over the reduction of all rows (first reduce each row horizontally to compute global histograms and then scan the results vertically). Equivalently, both terms can be computed by a single scan operation of size mL over a row-vectorized $\mathbf{H}$. Either way, the cost of our global operation is roughly proportional to both m and L. We see no realistic way to reduce m. Thus we concentrate on reducing L.

3.4.4 Hierarchical approach toward multisplit localization

We prefer to have small enough subproblems ($\bar{n}$) so that our local computations are "easy" for a direct solve. For any given subproblem size, we will have $L = n/\bar{n}$ subproblems to be processed globally as described before. On the other hand, we want to minimize our global computations as well, because they require synchronization among all subproblems and involve (expensive) global memory accesses. So, with a fixed input size and a fixed number of buckets (n and m), we would like to both decrease our subproblem size and number of subproblems, which is indeed paradoxical.

Our solution is a hierarchical approach. We do an arbitrary number of levels of divide-and-conquer, until at the last level, subproblems are small enough to be solved easily and directly (our preferred $\bar{n}$). These results are then appropriately combined together to eventually reach the first level of the hierarchy, where now we have a reasonable number of subproblems to be combined together using global computations (our preferred L).

Another advantage of such an approach is that, in case our hardware provides a memory hierarchy with smaller but faster local memory storage (as GPUs do with register level and shared memory level hierarchies, as opposed to the global memory), we can potentially perform all computations related to all levels except the first one in our local memory hierarchies without any global memory interaction. Ideally, we would want to use all our available register and shared memory with our subproblems to solve them locally, and then combine the results using global operations. In practice, however, since our local memory storage options are very limited, such solution may still lead to a large number of subproblems to be combined with global operations (large L). As a result, by adding more levels of hierarchy (than the available memory hierarchies in our device) we can systematically organize the way we fill our local memories, process them locally, store intermediate results, and then proceed to the next batch, which overall reduces our global operations. Next, we will theoretically consider such a hierarchical approach (*multi-level localization*) and explore the changes to equation (3.2).

λ-level localization For any given set of arbitrary non-zero integers $\{L_0, L_1, \ldots, L_{\lambda-1}\}$, we can perform λ levels of localizations as follows: Suppose we initially divide our problem into L_0 smaller parts. These divisions form our first level (i.e., the *global level*). Next, each subproblem

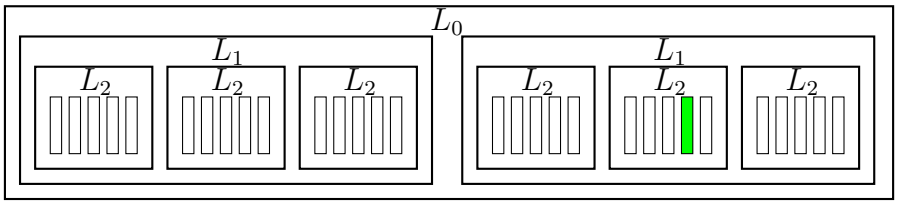


Figure 3.2: An example for our localization terminology. Here we have 3 levels of localization with $L_0 = 2$, $L_1 = 3$ and $L_2 = 5$. For example, the marked subproblem (in green) can be addressed by a tuple $(1, 1, 3)$ where each index respectively denotes its position in the hierarchical structure.

at the first level is divided into L_1 smaller subproblems to form the second level. We continue this process until the λth level (with $L_{\lambda-1}$ subproblems each). Figure 3.2 shows an example of our hierarchical division of the multisplit problem. There are total of $L_{\text{total}} = L_0 \times L_1 \times \dots L_{\lambda-1}$ smaller problems and their results should be hierarchically added together to compute the final permutation $p(i)$. Let $(\ell_0, \ell_1, \dots, \ell_{\lambda-1})$ denote a subproblem's position in our hierarchical tree: ℓ_0th branch from the first level, ℓ_1th branch from the second level, and so forth until the last level. Among all elements within this subproblem, we count those that belong to bucket B_i as $h_{i,(\ell_0,\dots,\ell_{\lambda-1})}$. Similar to our previous permutation computation, for an arbitrary i, j, and $(s_0, \dots, s_{\lambda-1})$, suppose $u_i \in \mathbf{u}_{(s_0,\dots,s_{\lambda-1})}$ and $u_i \in B_j$. We can write $p(i)$ as follows (from u_i's perspective):

$$
\begin{aligned}
p(i) = &\sum_{k=0}^{j-1} \left(\sum_{\ell_0=0}^{L_0-1} \sum_{\ell_1=0}^{L_1-1} \cdots \sum_{\ell_{\lambda-1}=0}^{L_{\lambda-1}-1} h_{k,(\ell_0,\ell_1,\dots,\ell_{\lambda-1})} \right) && \rightarrow \text{previous buckets in the whole problem} \\
&+ \sum_{\ell_0=0}^{s_0-1} \left(\sum_{\ell_1=0}^{L_1-1} \cdots \sum_{\ell_{\lambda-1}=0}^{L_{\lambda-1}-1} h_{j,(\ell_0,\ell_1,\dots,\ell_{\lambda-1})} \right) && \rightarrow u_i\text{'s bucket, first level, previous subproblems} \\
&+ \sum_{\ell_1=0}^{s_1-1} \left(\sum_{\ell_2=0}^{L_2-1} \cdots \sum_{\ell_{\lambda-1}=0}^{L_{\lambda-1}-1} h_{j,(s_0,\ell_1,\ell_2,\dots,\ell_{\lambda-1})} \right) && \rightarrow u_i\text{'s bucket, second level, previous subproblems} \\
&\vdots && \rightarrow u_i\text{'s bucket, previous levels, previous subproblems} \\
&+ \sum_{\ell_{\lambda-1}=0}^{s_{\lambda-1}-1} h_{j,(s_0,\dots,s_{\lambda-2},\ell_{\lambda-1})} && \rightarrow u_i\text{'s bucket, last level, previous subproblems} \\
&+ \left| \left\{ u_r \in \mathbf{u}_{(s_0,\dots,s_{\lambda-1})} : (u_r \in B_j) \wedge (r < i) \right\} \right|. && \rightarrow u_i\text{'s bucket, } u_i\text{'s subproblem}
\end{aligned}
\tag{3.3}
$$

There is an important resemblance between this equation and equation (3.2). The first and second terms (from top to bottom) are similar, with the only difference that each $h_{k,\ell}$ is now further broken into $L_1 \times \cdots \times L_{\lambda-1}$ subproblems (previously it was just $L = L_0$ subproblems). The other terms of (3.3) can be seen as a hierarchical disintegration of the local offset in (3.2).

Similar to Section 3.4.3, we can form a matrix $\mathbf{H} = [h_{j,\ell_0}]_{m \times L_0}$ for global computations where

$$h_{j,\ell_0} = \sum_{\ell_1=0}^{L_1-1} \cdots \sum_{\ell_{\lambda-1}=0}^{L_{\lambda-1}-1} h_{j,(\ell_0,\ell_1,\ldots,\ell_{\lambda-1})}. \tag{3.4}$$

Figure 3.3 depicts an schematic example of multiple levels of localization. At the highest level, for any arbitrary subproblem $(\ell_0, \ell_1 \ldots, \ell_{\lambda-1})$, local offsets per key are computed as well as all bucket counts. Bucket counts are then summed and sent to a lower level to form the bucket count for more subproblems ($L_{\lambda-1}$ consecutive subproblems). This process is continued until reaching the first level (the global level) where we have bucket counts for each $L_1 \times \cdots \times L_{\lambda-1}$ consecutive subproblems. This is where $\mathbf{H}$ is completed, and we can proceed with our global computation. Next, we discuss the way we compute local offsets.

3.4.5 Direct solve: Local offset computation

At the very last level of our localization, each element must compute its own local offset, which represents the number of elements in its subproblem (with our preferred size $\bar{n}$) that both precede it and share its bucket. To compute local offsets of a subproblem of size $\bar{n}$, we make a new binary matrix $\bar{\mathbf{H}}_{m \times \bar{n}}$, where each row represents a bucket and each column represents a key element. Each entry of this new matrix is one if the corresponding key element belongs to that bucket, and zero otherwise. Then by performing an exclusive scan on each row, we can compute local offsets for all elements belonging to that row (bucket). So each subproblem requires the following computations:

1. Mark all elements in each bucket (making local $\bar{\mathbf{H}}$)
2. m local reductions over the rows of $\bar{\mathbf{H}}$ to compute local histograms (a column in $\mathbf{H}$)
3. m local exclusive scans on rows of $\bar{\mathbf{H}}$ (local offsets)

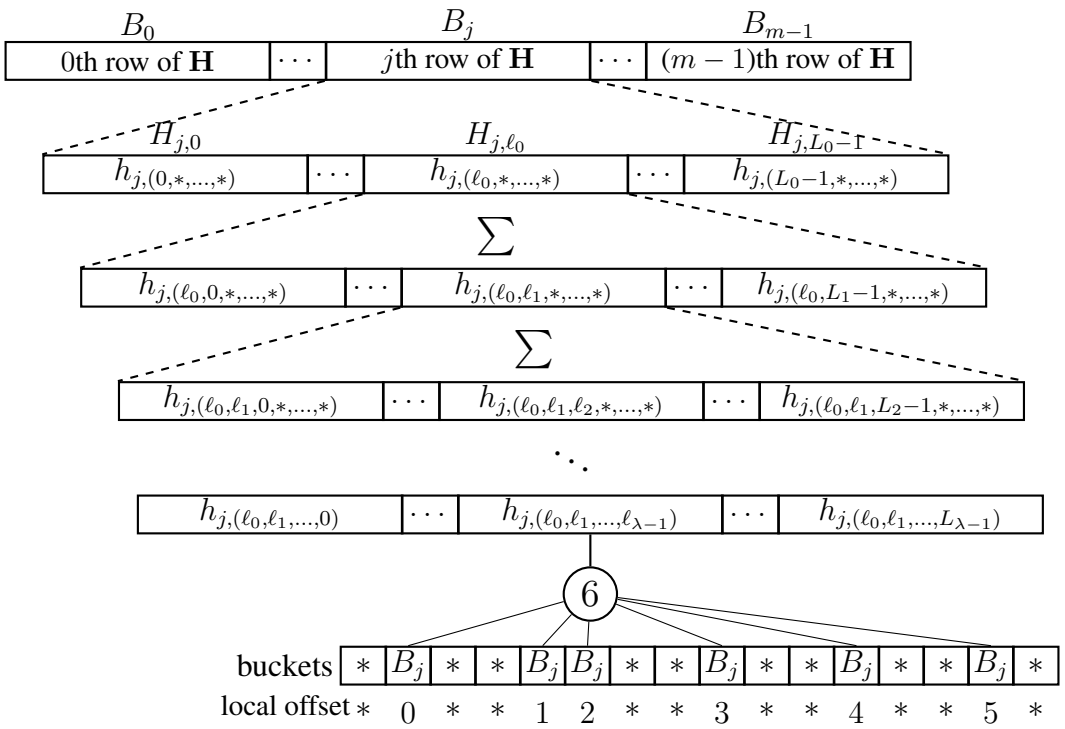


Figure 3.3: Each row of $\mathbf{H}$ belongs to a different bucket. Results from different subproblems in different levels are added together to form a lower level bucket count. This process is continued until reaching the first level where $\mathbf{H}$ is completed.

For clarity, we separate steps 2 and 3 above, but we can achieve both with a single local scan operation. Step 2 provides all histogram results that we need in equations (3.2) or (3.3) (i.e., all $h_{k,(\ell_0,...,\ell_{\lambda-1})}$ values) and step 3 provides the last term in either equation (Fig. 3.3).

It is interesting to note that as an extreme case of localization, we can have $L = n$ subproblems, where we divide our problem so much that in the end each subproblem is a single element. In such case, $\bar{\mathbf{H}}$ is itself a binary value. Thus, step 2's result is either 0 or 1. The local offset (step 3) for such a singleton matrix is always a zero (there is no other element within that subproblem).

3.4.6 Our multisplit algorithm

Now that we've outlined the different computations required for the multisplit, we can present a high-level view of the algorithmic skeleton we use in this chapter. We require three steps:

1. *Local*. For each subproblem at the highest level of our localization, for each bucket, count the number of items in the subproblem that fall into that bucket (direct solve for bucket counts). Results are then combined hierarchically and locally (based on equation (3.3))

until we have bucket counts per subproblem for the first level (global level).

2. *Global*. Scan the bucket counts for each bucket across all subproblems in the first level (global level), then scan the bucket totals. Each subproblem now knows both a) for each bucket, the total count for all its previous buckets across the whole input vector (term 1 in equation (3.3)) and b) for each bucket, the total count from the previous subproblems (term 2 in equation (3.3)).

3. *Local*. For each subproblem at the highest level of our localization, for each item, recompute bucket counts and compute the local offset for that item's bucket (direct solve). Local results for each level are then appropriately combined together with the global results from the previous levels (based on equation (3.3)) to compute final destinations. We can now write each item in parallel into its location in the output vector.

3.4.7 Reordering elements for better locality

After computing equation (3.3) for each key element, we can move key-value pairs to their final positions in global memory. However, in general, any two consecutive key elements in the original input do not belong to the same bucket, and thus their final destination might be far away from each other (i.e., a global scatter). Thus, when we write them back to memory, our memory writes are poorly coalesced, and our achieved memory bandwidth during this global scatter is similarly poor. This results in a huge performance bottleneck. How can we increase our coalescing and thus the memory bandwidth of our final global scatter? Our solution is to *reorder* our elements within a subproblem at the lowest level (or any other higher level) before they are scattered back to memory. Within a subproblem, we attempt to place elements from the same bucket next to each other, while still preserving order within a bucket (and thus the stable property of our multisplit implementation). We do this reordering at the same time we compute local offsets in equation (3.2). How do we group elements from the same bucket together? A local multisplit within the subproblem!

We have already computed histogram and local offsets for each element in each subproblem. We only need to perform another local exclusive scan on local histogram results to compute new positions for each element in its subproblem (computing equation (3.1) for each subproblem).

We emphasize that performing this additional stable multisplit on each subproblem does *not* change its histogram and local offsets, and hence does not affect any of our computations described previously from a global perspective; the final multisplit result is identical. But, it has a significant positive impact on the locality of our final data writes to global memory.

It is theoretically better for us to perform reordering in our largest subproblems (first level) so that there are potentially more candidate elements that might have consecutive/nearby final destinations. However, in practice, we may prefer to reorder elements in higher levels, not because they provide better locality but for purely practical limitations (such as limited available local memory to contain all elements within that subproblem).

3.5 Implementation Details

So far we have discussed our high level ideas for implementing an efficient multisplit algorithm for GPUs. In this section we thoroughly describe our design choices and implementation details. We first discuss existing memory and computational hierarchies in GPUs, and conventional localization options available on such devices. Then we discuss traditional design choices for similar problems such as multisplit, histogram, and radix sort. We follow this by our own design choices and how they differ from previous work. Finally, we propose three implementation variations of multisplit, each with its own localization method and computational details.

3.5.1 GPU memory and computational hierarchies

As briefly discussed in Section 3.2, GPUs offer three main memory storage options: 1) registers dedicated to each thread, 2) shared memory dedicated to all threads within a thread-block, 3) global memory accessible by all threads in the device.[6] From a computational point of view there are two main computational units: 1) threads have direct access to arithmetic units and perform register-based computations, 2) all threads within a warp can perform a limited but useful set of hardware based warp-wide intrinsics (e.g., ballots, shuffles, etc.). Although the latter is not physically a new computational unit, its inter-register communication among threads opens up new computational capabilities (such as parallel voting).

[6]There are other types of memory units in GPUs as well, such as local, constant, and texture memory. However, these are in general special-purpose memories and hence we have not targeted them in our design.

Based on memory and computational hierarchies discussed above, there are four primary ways of solving a problem on the GPU: 1) thread-level, 2) warp-level, 3) block-level, and 4) device-level (global). Traditionally, most efficient GPU programs for multisplit, histogram and radix sort [13, 41, 67] start from thread-level computations, where each thread processes a group of input elements and performs local computations (e.g., local histograms). These thread-level results are then usually combined to form a block-level solution, usually to benefit from the block's shared memory. Finally, block-level results are combined together to form a global solution. If implemented efficiently, these methods are capable of achieving high-quality performance from available GPU resources (e.g., CUB's high efficiency in histogram and radix sort).

In contrast, we advocate another way of solving these problems, based on a warp granularity. We start from a warp-level solution and then proceed up the hierarchy to form a device-wide (global) solution (we may bypass the block-level solution as well). Consequently, we target two major implementation alternatives to solve our multisplit problem: 1) warp-level $\rightarrow$ device-level, 2) warp-level $\rightarrow$ block-level $\rightarrow$ device-level (in Section 3.5.8, we discuss the costs and benefits of our approaches compared to a thread-level approach). Another algorithmic option that we outlined in Section 3.4.7 was to reorder elements to get better (coalesced) memory accesses. As a result of combining these two sets of alternatives, there will be four possible variations that we can explore. However, if we neglect reordering, our block-level solution will be identical to our warp-level solution, which leaves us with three main final options that all start with warp-level subproblem solutions and end up with a device-level global solution: 1) no reordering, 2) with reordering and bypassing a block-level solution, 3) with reordering and including a block-level solution. Next, we describe these three implementations and show how they fit into the multi-level localization model we described in Section 3.4.4.

3.5.2 Our proposed multisplit algorithms

So far we have seen that we can reduce the size and cost of our global operation (size of $\mathbf{H}$) by doing more local work (based on our multi-level localization and hierarchical approach). This is a complex tradeoff, since we prefer a small number of subproblems in our first level (global operations), as well as small enough subproblem sizes in our last levels so that they can easily

be solved within a warp. What remains is to choose the number and size of our localization levels and where to perform reordering. All these design choices should be made based on a set of complicated factors such as available shared memory and registers, achieved occupancy, required computational load, etc.

In this section we describe three novel and efficient multisplit implementations that explore different points in the design space, using the terminology that we introduced in Section 3.4.4 and Section 3.5.1.

Direct Multisplit Rather than split the problem into subproblems across threads, as in traditional approaches [41], Direct Multisplit (DMS) splits the problem into subproblems across warps (warp-level approach), leveraging efficient warp-wide intrinsics to perform the local computation.

Warp-level Multisplit Warp-level Multisplit (WMS) also uses a warp-level approach, but additionally reorders elements within each subproblem for better locality.

Block-level Multisplit Block-level Multisplit (BMS) modifies WMS to process larger-sized subproblems with a block-level approach that includes reordering, offering a further reduction in the cost of the global step at the cost of considerably more complex local computations.

We now discuss the most interesting aspects of our implementations of these three approaches, separately describing how we divide the problem into smaller pieces (our localization strategies), compute histograms and local offsets for larger subproblem sizes, and reorder final results before writing them to global memory to increase coalescing.

3.5.3 Localization and structure of our multisplit

In Section 3.4 we described our parallel model in solving the multisplit problem. Theoretically, we would like to both minimize our global computations as well as maximize our hardware utilization. However, in practice designing an efficient GPU algorithm is more complicated. There are various factors that need to be considered, and sometimes even be smartly sacrificed in order to satisfy a more important goal: efficiency of the whole algorithm. For example, we may

decide to recompute the same value multiple times in different kernel launches, just so that we do not need to store them in global memory for further reuse.

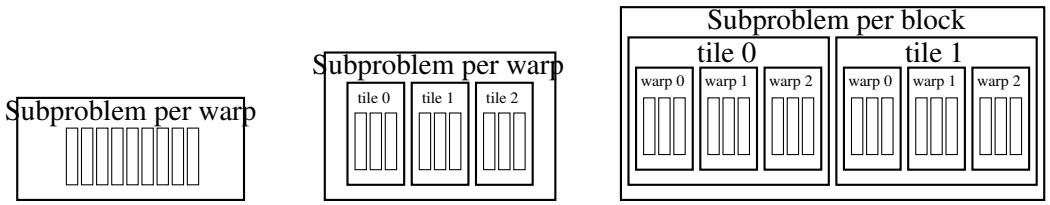
In our previous work [6], we implemented our multisplit algorithms with a straightforward localization strategy: Direct and Warp-level Multisplit divided problems into warp-sized subproblems (two levels of localization), and Block-level Multisplit used block-sized subproblems (three levels of localization) to extract more locality by performing more complicated computations needed for reordering. In order to have better utilization of available resources, we assigned multiple similar tasks to each launched warp/block (so each warp/block processed multiple independent subproblems). Though this approach was effective, we still faced relatively expensive global computations, and did not extract enough locality from our expensive reordering step. Both of these issues could be remedied by using larger subproblems within the same localization hierarchy. However, larger subproblems require more complicated computations and put more pressure on the limited available GPU resources (registers, shared memory, memory bandwidth, etc.). Instead, we redesigned our implementations to increase the number of levels of localization. This lets us have larger subproblems, while systematically coordinating our computational units (warps/blocks) and available resources to achieve better results.

Direct Multisplit Our DMS implementation has three levels of localizations: Each warp is assigned to a chunk of consecutive elements (first level). This chunk is then divided into a set of consecutive windows of warp-width ($N_{\text{thread}} = 32$) size (second level). For each window, we multisplit without any reordering (third level).

Warp-level Multisplit WMS is similar to DMS, but it also performs reordering to get better locality. In order to get better resource utilization, we add another level of localization compared to DMS (total of four). Each warp performs reordering over only a number of processed windows (a *tile*), and then continues to process the next tile. In general, each warp is in charge of a chunk of consecutive elements (first level). Each chunk is divided into several consecutive tiles (second level). Each tile is processed by a single warp and reordered by dividing it into several consecutive windows (third level). Each window is then directly processed by warp-wide methods (fourth level). The reason that we add another level of localization for each tile is simply because we do not have sufficient shared memory per warp to store the entire subproblem.

Algorithm	subproblem size ($\bar{n}$)
DMS	$N_{\text{(window/warp)}} N_{\text{thread}}$
WMS	$N_{\text{(tile/warp)}} N_{\text{(window/tile)}} N_{\text{thread}}$
BMS	$N_{\text{(tile/block)}} N_{\text{(warp/tile)}} N_{\text{(window/warp)}} N_{\text{thread}}$

Table 3.1: Size of subproblems for each multisplit algorithm. Total size of our global computations will then be the size of $\mathbf{H}$ equal to $mn/\bar{n}$.



(a) DMS localization (b) WMS localization (c) BMS localization

Figure 3.4: Different localizations for DMS, WMS and BMS are shown schematically. Assigned indices are just for illustration. Each small rectangle denotes a window of 32 consecutive elements. Reordering takes place per tile, but global offsets are computed per subproblem.

Block-level Multisplit BMS has five levels of localization. Each thread-block is in charge of a chunk of consecutive elements (first level). Each chunk is divided into a consecutive number of tiles (second level). Each tile is processed by all warps within a block (third level) and reordering happens in this level. Each warp processes multiple consecutive windows of input data within the tile (fourth level). In the end each window is processed directly by using warp-wide methods (fifth level). Here, for a similar reason as in WMS (limited shared memory), we added another level of localization per tile.

Figure 3.4 shows a schematic example of our three methods next to each other. Note that we have flexibility to tune subproblem sizes in each implementation by changing the sizing parameters in Table 3.1.

Next we briefly highlight the general structure of our computations (based on our model in Section 3.4.1). We use DMS to illustrate:

Pre-scan (local) Each warp reads a window of key elements (of size $N_{\text{thread}} = 32$), generates a local matrix $\bar{\mathbf{H}}$, and computes its histogram (reducing each row). Next, histogram results

are stored locally in registers. Then, each warp continues to the next window and repeats the process, adding histogram results to the results from previous windows. In the end, each warp has computed a single column of $\mathbf{H}$ and stores its results into global memory.

Scan (global) We perform an exclusive scan operation over the row-vectorized $\mathbf{H}$ and store the result back into global memory (e.g., matrix $\mathbf{G} = [g_{i,\ell}]_{m \times L_0}$).

Post-scan (local) Each warp reads a window of key-value pairs, generates its local matrix $\bar{\mathbf{H}}$ again,[7] and computes local offsets (with a local exclusive scan on each row). Similar to the pre-scan stage, we store histogram results into registers. We then compute final positions by using the global base addresses from $\mathbf{G}$, warp-wide base addresses (warp-wide histogram results up until that window), and local offsets. Then we write key-value pairs directly to their storage locations in the output vector. For example, if key $u \in B_i$ is read by warp ℓ and its local offset is equal to k, its final position will be $g_{i,\ell} + h_i + k$, where h_i is the warp-wide histogram result up until that window (referring to equation (3.3) is helpful to visualize how we compute final addresses with a multi-level localization).

Algorithm 2 shows a simplified pseudo-code of the DMS method (with less than 32 buckets). Here, we can identify each key's bucket by using a `bucket_identifier()` function. We compute warp histogram and local offsets with `warp_histogram()` and `warp_offsets()` procedures, which we describe in detail later in this section (Alg. 3 and 4).

3.5.4 Ballot-based voting

In this section, we momentarily change our direction into exploring a theoretical problem about voting. We then use this concept to design and implement our warp-wide histograms (Section 3.5.5). We have previously emphasized our design decision of a warp-wide granularity. This decision is enabled by the efficient warp-wide intrinsics of NVIDIA GPUs. In particular, NVIDIA GPUs support a warp-wide intrinsic `__ballot(predicate)`, which performs binary voting across all threads in a warp. More specifically, each thread evaluates a local Boolean predicate, and depending on the result of that predicate (true or false), it toggles a specific bit corresponding to its position in the warp (i.e., lane ID from 0 to 31). With a 32-

[7] Note that we compute $\bar{\mathbf{H}}$ a second time rather than storing and reloading the results from the computation in the first step. This is deliberate. We find that the recomputation is cheaper than the cost of global store and load.

ALGORITHM 2: The Direct Multisplit (DMS) algorithm

Input: key_in[], value_in[], bucket_identifier(): keys, values and a bucket identifier function.

Output: key_out[], value_out[]: keys and values after multisplit.

// key_in[], value_in[], key_out[], value_out[], H, and G are all stored in global memory. // L: number of subproblems

// ====== Pre-scan stage:

for *each warp* i=0:L-1 **parallel device do**

    histo[0:m-1] = 0;

    for *each window* j=0:N_window-1 **do**

        bucket_id[0:31] = bucket_identifier(key_in[32*i*N_window + 32*j + (0:31)]);

        histo[0:m-1] += warp_histogram(bucket_id[0:31]);

    end

    H[0:m-1][i] = histo[0:m-1];

end

// ====== Scan stage:

H_row = [H[0][0:L-1],H[1][0:L-1], . . . , H[m-1][0:L-1]];

G_row = exclusive_scan(H_row);

// [G[0][0:L-1],G[1][0:L-1], . . . , G[m-1][0:L-1]] = G_row;

for i = 0:m-1 *and* j = 0:L-1 **do**

    G[i][j] = G_row[i ∗ m + j];

end

// ====== Post-scan stage:

for *each warp* i=0:L-1 **parallel device do**

    histo[0:m-1] = 0;

    for *each window* j=0:N_window-1 **do**

        read_key = key_in[32*i*N_window + 32*j + (0:31)];

        read_value = value_in[32*i*N_window + 32*j + (0:31)];

        bucket_id[0:31] = bucket_identifier(read_key);

        offsets[0:31] = warp_offsets(bucket_id[0:31]);

        for *each thread* k=0:31 **parallel warp do**

            final_position[k] = G[bucket_id[k]][i] + histo[bucket_id[k]] + offsets[k];

            key_out[final_position[k]] = read_key;

            value_out[final_position[k]] = read_value;

        end

        histo[0:m-1] += warp_histogram(bucket_id[0:31]); // *updating histograms*

    end

end

element warp, this ballot fits in a 32-bit register, so that the ith thread in a warp toggles the ith bit. After the ballot is computed, every participant thread can access the ballot result (as a bitmap) and see the voting result from all other threads in the same warp.

Now, the question we want to answer within our multisplit implementation is a generalization to the voting problem: Suppose there are m arbitrary agents (indexed from 0 to $n-1$), each evaluating a personalized non-binary predicate (a vote for a candidate) that can be any value from 0 to $m-1$. How is it possible to perform the voting so that any agent can know all the results (who voted for whom)?

A naive way to solve this problem is to perform m separate binary votes. For each round $0 \leq i < m$, we just ask if anyone wants to vote for i. Each vote has a binary result (either voting for i or not). After m votes, any agent can look at the $m \times n$ ballots and know which agent voted for which of the m candidates.

We note that each agent's vote ($0 \leq v_i < n$) can be represented by $\log m$ binary digits. So a more efficient way to solve this problem requires just $\log m$ binary ballots per agent. Instead of directly asking for a vote per candidate (m votes/bitmaps), we can ask for consecutive bits of each agent's vote (a bit at a time) and store them as a bitmap (for a total of $\log m$ bitmaps, each bitmap of size n). For the jth bitmap ($0 \leq j < \lceil \log m \rceil$), every ith bit is one if only the ith agent have voted to a candidate whose jth bit in its binary representation was also one (e.g., the 0th bitmap includes all agents who voted for an odd-numbered candidate).

As a result, these $\lceil \log m \rceil$ bitmaps together contain all information to reconstruct every agent's vote. All we need to do is to imagine each bitmap as a row of a $m \times n$ matrix. Each column represent the binary representation of the vote of that specific agent. Next, we use this scheme to perform some of our warp-wide computations, only using NVIDIA GPU's binary ballots.

3.5.5 Computing Histograms and Local Offsets

The previous subsections described why and how we create a hierarchy of localizations. Now we turn to the problem of computing a direct solve of histograms and local offsets on a warp-sized (DMS or WMS) or a block-sized (BMS) problem. In our implementation, we leverage the balloting primitives we described in Section 3.5.4. We assume throughout this section that the

number of buckets does not exceed the warp width ($m \leq N_{\text{thread}}$). Later we extend our discussion to any number of buckets in Section 3.5.7.

3.5.5.1 Warp-level Histogram

Previously, we described our histogram computations in each subproblem as forming a binary matrix $\bar{\mathbf{H}}$ and doing certain computations on each row (reduction and scan). Instead of explicitly forming the binary matrix $\bar{\mathbf{H}}$, each thread generates its own version of the rows of this matrix and stores it in its local registers as a binary bitmap. Then per-row reduction is equivalent to a population count operation (`__popc`), and exclusive scan equates to first masking corresponding bits and then reducing the result. We now describe both in more detail.

To compute warp-level histograms, we assign each bucket (each row of $\bar{\mathbf{H}}$) to a thread. That thread is responsible for counting the elements of the warp (of the current read window of input keys) that fall into that bucket. We described in Section 3.5.4 how each agent (here, each thread) can know about all the votes (here, the bucket to which each key belongs). Since we assigned each thread to count the results of a particular bucket (here, the same as its lane ID ranging from 0 to 31), each thread must only count the number of other threads that voted for a bucket equal to its lane ID (the bucket to which it is assigned). For cases where there are more buckets than the warp width, we assign $\lceil m/32 \rceil$ buckets to each thread. First, we focus on $m \leq 32$; Algorithm 3 shows the detailed code.

Each thread i is in charge of the bucket with an index equal to its lane ID (0–31). Thread i reads a key, computes that key's bucket ID (0–31), and initializes a warp-sized bitvector (32 bits) to all ones. This bitvector corresponds to threads (keys) in the warp that might have a bucket ID equal to this thread's assigned bucket. Then each thread broadcasts the least significant bit (LSB) of its observed bucket ID, using the warp-wide ballot instruction. Thread i then zeroes out the bits in its local bitmap that correspond to threads that are broadcasting a LSB that is incompatible with i's assigned bucket. This process continues with all other bits of the observed bucket IDs (for m buckets, that's $\log m$ rounds). When all rounds are complete, each thread has a bitmap that indicates which threads in the warp have a bucket ID corresponding to its assigned bucket. The histogram result is then a reduction over these set bits, which is computed with a single population count (`__popc`) instruction.

ALGORITHM 3: Warp-level histogram computation

```
Function warp_histogram(bucket_id[0:31])
    Input: bucket_id[0:31] // a warp-wide array of bucket IDs
    Output: histo[0:m-1] // number of elements within each m buckets
    for each thread i = 0:31 parallel warp do
        histo_bmp[i] = 0xFFFFFFFF for (int k = 0; k < ceil(log2(m)); k++) do
            temp_buffer = __ballot((bucket_id[i] » k) & 0x01);
            if ((i » k) & 0x01) then histo_bmp[i] &= temp_buffer;
            else histo_bmp[i] &= ∼ temp_buffer;
        end
        histo[i] = __popc(histo_bmp[i]); // counting number of set bits
    end
    return histo[0:m-1];
```

For $m > 32$, there will still be $\lceil \log(m) \rceil$ rounds of ballots. However, each thread will need $\lceil m/32 \rceil$ 32-bit registers to keep binary bitvectors (for multiple `histo_bmp` registers per thread). Each of those registers is dedicated to the same lane IDs within that warp, but one for buckets 0–31, the second for buckets 32–63, and so forth.

3.5.5.2 Warp-level Local Offsets

Local offset computations follow a similar structure to histograms (Algorithm 4). In local offset computations, however, each thread is only interested in keeping track of ballot results that match its item's *observed* bucket ID, rather than the bucket ID to which it has been assigned. Thus we compute a bitmap that corresponds to threads whose items share our same bucket, mask away all threads with higher lane IDs, and use the population count instruction to compute the local offset.

For $m > 32$, we can use the same algorithm (Algorithm 4). The reason for this, and the main difference with our histogram computation, is that regardless of the number of buckets, there are always a fixed number of threads within a warp. So, we still need to perform $\lceil \log m \rceil$ ballots but our computations are the same as before, to look for those that share the same bucket ID as ours (only 32 other potential threads). This is unlike our histogram computations (Algorithm 3), that we needed to keep track of $\lceil m/32 \rceil$ different buckets because there were more buckets than

existing threads within a warp.

Histogram and local offset computations are shown in two separate procedures (Alg. 3 and 4), but since they share many common operations they can be merged into a single procedure if necessary (by sharing the same ballot results). For example, in our implementations, we only require histogram computation in the pre-scan stage, but we need both a histogram and local offsets in the post-scan stage.

ALGORITHM 4: Warp-level local offset computation

```
Function warp_offset(bucket_id[0:31])
    Input: bucket_id[0:31] // a warp-wide array of bucket IDs
    Output: local_offset[0:31] // for each element, number of preceding elements within the same bucket
    for each thread i = 0:31 parallel warp do
        offset_bmp[i] = 0xFFFFFFFF for (int k = 0; k < ceil(log2(m)); k++) do
            temp_buffer = __ballot((bucket_id[i] » k) & 0x01);
            if ((i » k) & 0x01) then offset_bmp[i] &= temp_buffer;
            else offset_bmp[i] &= ∼ temp_buffer;
        end
        local_offset[i] = __popc(offset_bmp[i]&(0xFFFFFFFF»(31-i))) - 1; // counting number of
          preceding set bits
    end
    return local_offset[0:31];
```

3.5.5.3 Block-level Histogram

For our Block-level MS, we perform the identical computation as Direct MS and Warp-level MS, but over a block rather than a warp. If we chose explicit local computations (described in Section 3.4) to compute histograms and local offsets, the binary matrix $\bar{\mathbf{H}}$ would be large, and we would have to reduce it over rows (for histograms) and scan it over rows (for local offsets). Because of this complexity, and because our warp-level histogram computation is quite efficient, we pursue the second option: the hierarchical approach. Each warp reads consecutive windows of keys and computes and aggregates their histograms. Results are then stored into consecutive parts of shared memory such that all results for the first bucket (B_0) are next to each other: results from the first warp followed by results from the second warp, up to the last warp. Then, the same

happens for the second bucket (B_1) and so forth until the last bucket B_{m-1}. Now our required computation is a segmented (per-bucket) reduction over m segments of N_{warp} histogram results.

In our implementation, we choose to always use a power-of-two number of warps per block. As a result, after writing these intermediate warp-wide histograms into shared memory and syncing all threads, we can make each warp read a consecutive segment of 32 elements that will include $32/N_{\text{warp}}$ segments (buckets). Then, segmented reduction can be performed with exactly $\log N_{\text{warp}}$ rounds of `__shfl_xor()`.[8] This method is used in our BMS's pre-scan stage, where each segment's result (which is now a tile's histogram) is written into global memory (a column of $\mathbf{H}$). The process is then continued for the next tile in the same thread-block.

3.5.5.4 Block-level Local Offsets

Our block-wide local offset computation is similar to our warp-level local offsets with some additional offset computations. First, for each warp within a thread-block, we use both Alg. 3 and Alg. 4 to compute histogram and local offsets per window of input keys. By using the same principle showed in our hierarchical computations in equation 3.3, we use these intermediate results to compute block-wide local offsets (the tile's local offset in BMS). To compute the local offset of each element within its own bucket, we require three components:

1. local offsets for each window (same tile, same bucket, same window)

2. total number of elements processed by the same warp, same bucket, but from previous windows

3. total number of elements in the same tile, same bucket, but from previous warps (each with multiple windows)

To illustrate, suppose, $h_{j,(x,y)}$ shows the histogram results of bucket B_j for warp x and window y. Then for a key u_i which is read by warp X in its window Y, we have:

$$\begin{aligned}\text{local offset}(u_i) = &\left|\{u_r \in (\text{warp } X, \text{window } Y) : (u_r \in B_j) \wedge (r < i)\}\right| \\ &+ \sum_{y=0}^{Y-1} h_{j,(X,y)} + \sum_{x=0}^{X-1} \sum_{y=0}^{N_{\text{window}}-1} h_{j,(x,y)}. \end{aligned} \tag{3.5}$$

[8] This is a simple modification of a warp-wide reduction example given in the CUDA Programming guide [78, Chapter B14].

The first term (item) is exactly the outcome of Alg. 4, computed per window of input keys. By having all histogram results per window, each warp can locally compute the second term. However, each thread only has access to the counts corresponding to its in-charge bucket (equal to its lane ID). By using a single shuffle instruction, each thread can ask for this value from a thread that has the result already computed (within the same warp). For computing the third term, we replicate what we did in Section 3.5.5.3. Histogram results (aggregated results for all windows per warp) are stored into consecutive segments of shared memory (total of $32/N_{\text{warp}}$ segments). Then, we perform a segmented scan by using $\log(N_{\text{warp}})$ rounds of `__shfl_up()` (instead of `__shfl_xor`). All results are then stored back into shared memory, so that all threads can access appropriate values that they need (to compute equation 3.5) based on their warp ID. As a result, we have computed the local offset of each key within each thread-block.

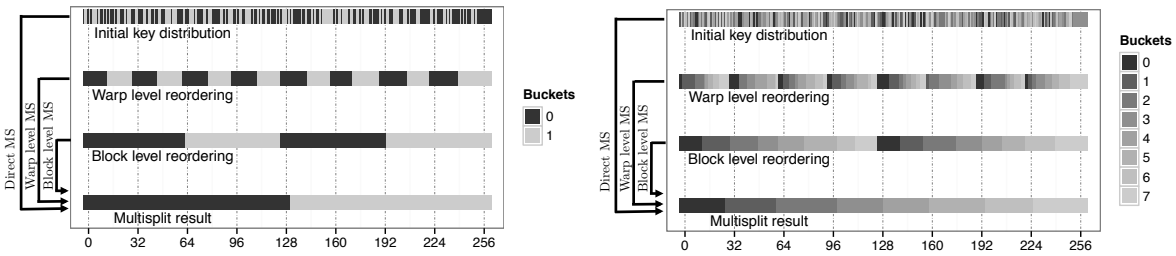
3.5.6 Reordering for better locality

As described in Section 3.4, one of the main bottlenecks in a permutation like multisplit is the random scatter in its final data movement. Figure 3.5 shows an example of such a case. As we suggested previously, we can improve scatter performance by reordering elements locally in each subproblem such that in the final scatter, we get better coalescing behavior (i.e., consecutive elements are written to consecutive locations in global memory).

However, while a higher achieved write memory bandwidth will improve our runtime, it comes at the cost of more local work to reorder elements. Warp-level reordering requires the fewest extra computations, but it may not be able to give us enough locality as the number of buckets increases (Fig. 3.5). We can achieve better locality, again at the cost of more computation, by reordering across warps within a block.

3.5.6.1 Warp-level Reordering

WMS extends DMS by reordering each tile (a group of consecutive windows in WMS) before the final write for better memory coalescing behavior. Our first question was whether we prefer to perform the reordering in our pre-scan stage or our post-scan stage. We know that in order to compute the new index for each element in a tile, we need to know about its histogram and we need to perform a local (warp-level) exclusive scan over the results. We have already computed the warp level histogram in the pre-scan stage, but we do not have it in the post-scan stage and



(a) Key distribution with 2 buckets

(b) Key distribution with 8 buckets

Figure 3.5: Key distributions for different multisplit methods and different number of buckets. Key elements are initially uniformly distributed among different buckets. This window shows an input key vector of length 256; each warp is 32 threads wide and each block has 128 threads.

thus would either have to reload it or recompute it.

However, if we reorder key-value pairs in the pre-scan stage, we must perform two coalesced global reads (reading key-value pairs) and two coalesced global writes (storing the reordered key-value pairs before our global operation) per thread and for each key-value pair. Recall that in DMS, we only required one global read (just the key) per thread and per key in its pre-scan stage.

In the end, the potential cost of the additional global reads was significantly more expensive than the much smaller cost of recomputing our efficient warp-level histograms. As a result, we reorder in the post-scan stage and require fewer global memory accesses overall.

The main difference between DMS and WMS is in post-scan, where we compute both warp-level histogram and local offsets (Algorithm 3 and 4). As we described before, in WMS each subproblem is divided into several tiles. Each tile is assigned to a warp and is processed in a consecutive number of windows of length 32 each. Each warp, however, only performs reordering for all elements within a tile (not among all tiles). This decision is mostly because of limited available shared memory per block.

Algorithms 3 and 4 provide histogram and local offsets per window of read data. So, in order to perform warp-level reordering (performing a local multisplit within a tile), we need the following items to be computed for each key-value pair to be able to compute their new positions in shared memory:

1. local offsets for each window (same tile, same bucket, same window)

2. total number of elements processed by the same warp (same tile), from the same bucket,

but from previous windows

3. total number of elements in the same tile, but from previous buckets

The first item is exactly the warp-level local offset computed in Section 3.5.5.2. The second item can be computed as follows: each thread within a warp directly computes all histogram results for all its windows. So, it can locally compute an exclusive scan of these results. However, each thread only has access to the counts per window of the bucket for which it is in charge (i.e., the bucket equal to its lane ID). As a result, it is enough for each thread to ask for the appropriate value of (2) by asking the thread who owns it (via a single shuffle instruction).

For the third item, we do the following: each thread within a warp has already computed the total number of elements per window (for specific buckets). So, it can easily add them together to form the histogram for all the windows (tile histogram). All that remains is to perform another exclusive scan among these values such that each thread has the total number of items from previous buckets within the same tile. This result too can be provided to threads by using a single shuffle instruction. We can now perform reordering for key-value pairs into shared memory.

After performing the warp-level reordering, we need to write back results into global memory and into their final positions. To do so, all threads within a warp once again read key-value pairs from shared memory (it will be in N_{window} coalesced reads). Since items are reordered, previous threads are not necessarily reading the same items as before and hence they should identify their buckets once again. Since elements are already reordered (consecutive elements belong to the same bucket), the new local offset among all other keys within the same bucket and same tile can be recomputed in a much easier way: each item's index (within shared memory) minus the offset computed in step (3) of reordering (which can be accessed by a single shuffle instruction). Finally we can add the computed local tile-offsets to the global offset offsets computed by the scan-stage, and perform final data movements in a (more likely) coalesced way.

3.5.6.2 Block-level Reordering

The benefit from warp-level reordering is rather modest, particularly as the number of buckets grows, because we only see a small number of elements per warp (a WMS's tile) that belong to the same bucket. For potentially larger gains in coalescing, our BMS reorders entire blocks (larger tiles by a factor of N_{warp}). That being said, an important advantage of our WMS is

that almost everything can be computed within a warp, and since warps perform in lockstep, there will not be any need for further synchronizations among different warps. In contrast, any coordination among warps within a thread-block requires proper synchronization.

As we mentioned before, each subproblem in BMS is assigned to a thread-block and is divided into several tiles. Each tile is assigned to multiple warps within a block. Each warp divides its share into multiple consecutive windows of 32 elements each. Final positions can be computed in a hierarchical approach with 5 levels of localization, as described in equation 3.3.

In BMS, although each block can process multiple consecutive tiles, we only perform reordering per tile (mostly because of limited available shared memory per block). Reordering is equivalent to performing local multisplit over each tile. We can summarize the required computations for this reordering with the following two items:

1. block-level local offsets

2. total number of keys in the same tile, from previous buckets

The first item can be computed as described in Section 3.5.5.4. During the above computation, at some point we stored results from a segmented exclusive scan into shared memory. Since our shuffle-based segmented scan is initially an inclusive scan, each thread has to subtract its initial bucket count from it to make it an exclusive scan. So, during that computation, with minimal extra effort, we could also store results for the sum of all elements within each segment (i.e., the total number of elements within each bucket in that tile) into another location in shared memory as well (for a total of m elements). We refer to this result as our tile histogram. Now, here each warp can reload the tile histogram from shared memory and perform a warp-wide exclusive scan operation on it. In order to avoid extra synchronizations, every warp performs this step independently (as opposed to the option that a single warp computes it and puts it into shared memory, thus making it available for all other warps). Thus, each thread can easily ask the required value for item (2) by using a single shuffle instruction to fetch the value from the thread that owns that bucket's result.

After all threads within a block finish storing reordered key-value pairs into shared memory (for a single tile), we perform the final data movements. Threads read the reordered tile (different

key-value pairs than before), identify their buckets, and compute the final positions based on the following items:

(i) The new block-level local offset

(ii) total number of elements from previous subproblems (the whole device) and from previous buckets

Since elements are now already reordered, consecutive elements belong to the same bucket. As a result, the first item is equivalent to the index of that element (within shared memory) minus the starting index for that specific bucket (which is exactly item (2) in reordering). The second item is also already computed and available from our scan stage. So, threads can proceed to perform the final data movement.

3.5.7 More buckets than the warp width

Warp-level histogram and local offset computation Throughout the previous discussion, our primary way of computing a histogram in each warp (which processes a group of windows one by one) is to make each thread responsible to count the total number of elements with the bucket ID equal to its lane ID. Since the current generation of GPUs has $N_{\text{thread}} = 32$ threads per warp, if the number of buckets is larger than the warp width, we must put each thread in charge of multiple buckets (each thread is in charge of $\lceil m/32 \rceil$ buckets as described in Section 3.5.5.1). The total number of ballots required for our warp-level histogram procedure scales logarithmically ($\lceil \log m \rceil$). Local offset computations are also as before (with $\lceil \log m \rceil$ ballots).

Block-level histogram and local offsets computations If $m > 32$, besides the changes described above, to perform our segmented scan and reductions (Section 3.5.5.3), each thread should participate in the computations required by $\lceil m/32 \rceil$ segments. Previously, for $m > 32$ buckets we used CUB's block-wide scan operation [6]. However, although CUB is a very efficient and high-performance algorithm in performing scan, it uses a lot of registers to achieve its goal. As a result, we prefer a more register-friendly approach to these block-level operations, and hence implemented our own segmented scan and reductions by simply iterating over multiple segments, processing each as described in Section 3.5.5.3 and Section 3.5.5.4.

3.5.8 Privatization

Traditionally, multisplit [41] (as well as histogram and radix sort [13, 67] with some similarities) were implemented with a thread-level approach with thread-level memory *privatization*: each thread was responsible for a (possibly contiguous) portion of input data. Intermediate results (such as local histograms, local offsets, etc.) were computed and stored in parts of a memory (either in register or shared memory) exclusively dedicated to that thread. Privatization eliminates contention among parallel threads at the expense of register/shared memory usage (valuable resources). Memory accesses are also more complicated since the end goal is to have a sequence of memory units per thread (commonly known as *vectorized* access), as opposed to natural coalesced accesses where consecutive memory units are accessed by consecutive threads. That being said, this situation can be remedied by careful pipelining and usage of shared memory (initially coalescing reads from global memory, writing the results into shared memory, followed by vectorized reads from shared memory).

In contrast, in this work, we advocate a *warp-level* strategy that assigns different warps to consecutive segments of the input elements and stores intermediate results in portions of memory (distributed across all registers of a warp, or in shared memory) that are exclusively assigned to that particular warp [6]. An immediate advantage of a warp-level strategy is a reduction in shared memory requirements per thread-block (by a factor of warp-width). Another advantage is that there is no more need for vectorized memory accesses, relieving further pressure on shared memory usage. The chief disadvantage is the need for warp-wide intrinsics for our computations. These intrinsics may be less capable or deliver less performance. On recent NVIDIA GPUs, in general, using warp-wide intrinsics are faster than regular shared memory accesses but slower than register-level accesses. A more detailed comparison is not possible since it would heavily depend on the specific task at hand. However, efficient warp-wide intrinsics open up new possibilities for warp-wide computations as fundamental building blocks, allowing algorithm designers to consider using both thread and warp granularities when constructing and tuning their algorithms.

NVIDIA GPU	Tesla K40c	GeForce GTX 1080
Architecture	Kepler	Pascal
Compute capability	3.5	6.1
Number of SMs	15	20
Global Memory size	12 GB	8 GB
Global memory bandwidth	288 GB/s	320 GB/s
Shared memory per SM	48 KB	96 KB

Table 3.2: Hardware characteristics of the NVIDIA GPUs that we used in this chapter.

3.6 Performance Evaluation

In this section we evaluate our multisplit methods and analyze their performance. First, we discuss a few characteristics in our simulations:

Simulation Framework All experiments are run on a NVIDIA K40c with the Kepler architecture, and a NVIDIA GeForce GTX 1080 with the Pascal architecture (Table 3.2). All programs are compiled with NVIDIA's nvcc compiler (version 8.0.44). The authors have implemented all codes except for device-wide scan operations and radix sort, which are from CUB (version 1.6.4). All experiments are run over 50 independent trials. Since the main focus of this chapter is on multisplit as a GPU primitive within the context of a larger GPU application, we assume that all required data is already in the GPU's memory and hence no transfer time is included.

Some server NVIDIA GPUs (such as Tesla K40c) provide an error correcting code (ECC) feature to decrease occurrence of unpredictable memory errors (mostly due to physical noise perturbations within the device in long-running applications). ECCs are by default enabled in these devices, which means that hardware dedicates a portion of its bandwidth to extra parity bits to make sure all memory transfers are handled correctly (with more probability). Some developers prefer to disable ECC to get more bandwidth from these devices. In this work, in order to provide a more general discussion, we opt to consider three main hardware choices: 1) Tesla K40c with ECC enabled (default), 2) Tesla K40c with ECC disabled, and 3) GeForce GTX 1080 (no ECC option).

Bucket identification The choice of bucket identification directly impacts performance results of any multisplit method, including ours. We support user-defined bucket identifiers. These can be as simple as unary functions, or complicated functors with arbitrary local arguments.

For example, one could utilize a functor which determines whether a key is prime or not. Our implementation is simple enough to let users easily change the bucket identifiers as they please.

In this section, we assume a simple user-defined bucket identifier as follows: buckets are assumed to be of equal width Δ and to partition the whole key domain (*delta-buckets*). For example, for an arbitrary key u, bucket IDs can be computed by a single integer division (i.e., $\boldsymbol{f}(u) = \lfloor u/\Delta \rfloor$). Later, in Section 3.7.1 we will consider a simpler bucket identifier (*identity buckets*): where keys are equal to their bucket IDs (i.e., $\boldsymbol{f}(u) = u$). This is particularly useful when we want to use our multisplit algorithm to implement a radix sort. In Section 3.7.3 we use more complicated identifiers as follows: given a set of arbitrary splitters $s_0 < s_1 < \cdots < s_{m-1}$, for each key $s_0 < u < s_{m-1}$, finding those splitters (i.e., bucket B_j) such that $s_j \leq u < s_{j+1}$. This type of identification requires performing a binary search over all splitters per input key.

Key distribution Throughout this chapter we assume uniform distribution of keys among buckets (except in Section 3.6.4 where we consider other distributions), meaning that keys are randomly generated such that there are, on average, equal number of elements within each bucket. For delta-buckets and identity buckets (or any other linear bucket identifier), this criteria results in uniform distribution of keys in the key domain as well. For more complicated nonlinear bucket identifiers this does not generally hold true.

Parameters In all our methods and for every GPU architecture we have used either: 1) four warps per block (128 threads per block), where each warp processes 7 consecutive windows, or 2) eight warps per block where each warp processes 4 consecutive windows. Our key-only BMS for up to $m \leq 32$ uses the former, while every other case uses the latter (including WMS and BMS for both key-only and key-value pairs). These options were chosen because they gave us the best performance experimentally.

This is a trade off between easier inter-warp computations (fewer warps) versus easier intra-warp communications (fewer windows). By having fewer warps per block, all our inter-warp computations in BMS (segmented scans and reductions in Section 3.5.5.3 and 3.5.5.4) are directly improved, because each segment will be smaller-sized and hence fewer rounds of shuffles are required ($\log N_{\text{warp}}$ rounds). On the other hand, we use subword optimizations to pack the intermediate results of 4 processed windows into a single 32-bit integer (a byte per bucket per

window). This lets us communicate among threads within a warp by just using a single shuffle per 4 windows. Thus, by having fewer warps per block, if we want to load enough input keys to properly hide memory access latency, we would need more than 4 windows to be read by each warp (here we used 7), which doubles the total number of shuffles that we use.

It is a common practice for GPU libraries, such as in CUB's radix sort, to choose their internal parameters at runtime based on the GPU's compute capability and architecture. These parameters may include the number of threads per block and the number of consecutive elements to be processed by a single thread. The optimal parameters may substantially differ on one architecture compared to the other. In our final API, we hid these internal parameters from the user; however, our experiments on the two GPUs we used (Tesla K40c with `sm_35` and GeForce GTX 1080 with `sm_61`) exhibited little difference between the optimal set of parameters for the best performance on each architecture.

In our algorithms, we always use as many threads per warp as allowed on NVIDIA hardware ($N_{\text{thread}} = 32$). Based on our reliance on warp-wide ballots and shuffles to perform our local computations (as discussed in Section 3.5.5), using smaller-sized logical warps would mean having smaller sized subproblems (reducing potential local work and increasing global computations), which is unfavorable. On the other hand, providing a larger-sized warp in future hardware with efficient ballots and shuffles (e.g., performing ballot over 64 threads and storing results as a 64-bit bitvector) would directly improve all our algorithms.

3.6.1 Common approaches and performance references

Radix sort As we described in Section 3.3, not every multisplit problem can be solved by directly sorting input keys (or key-values). However, in certain scenarios where keys that belong to a lower indexed bucket are themselves smaller than keys belonging to larger indexed buckets (e.g., in delta-buckets), direct sorting results in a non-stable multisplit solution. In this work, as a point of reference, we compare our performance to a full sort (over 32-bit keys or key-values). Currently, the fastest GPU sort is provided by CUB's radix sort (Table 3.3). With a uniform distribution of keys, radix sort's performance is independent of the number of buckets; instead, it only depends on the number of significant bits.

	Tesla K40c (ECC on)		Tesla K40c (ECC off)		GeForce GTX 1080	
Method	time	rate	time	rate	time	rate
Radix sort (key-only)	25.99 ms	1.29 Gkeys/s	19.41 ms	1.73 Gkeys/s	9.84 ms	3.40 Gkeys/s
Radix sort (key-value)	43.70 ms	0.77 Gpairs/s	28.60 ms	1.17 Gpairs/s	17.59 ms	1.90 Gpairs/s
Scan-based split (key-only)	5.55 ms	6.05 Gkeys/s	4.91 ms	6.84 Gkeys/s	3.98 ms	8.44 Gkeys/s
Scan-based split (key-value)	6.96 ms	4.82 Gpairs/s	5.97 ms	5.62 Gpairs/s	5.13 ms	6.55 Gpairs/s

Table 3.3: On the top: CUB's radix sort. Average running time (ms) and processing rate (billion elements per second), over 2^{25} randomly generated 32-bit inputs (keys or key-value pairs). On the bottom: our scan-based split. Average running time (ms) and processing rate (billion elements per second), over 2^{25} randomly generated 32-bit inputs uniformly distributed into two buckets.

Reduced-bit sort Reduced-bit sort (*RB-sort*) was introduced in Section 3.3 as the most competitive conventional-GPU-sort-based multisplit method. In this section, we will compare all our methods against RB-sort. We have implemented our own kernels to perform labeling (generating an auxiliary array of bucket IDs) and possible packing/unpacking (for key-value multisplit). For its sorting stage, we have used CUB's radix sort.

Scan-based splits Iterative scan-based split can be used on any number of buckets. For this method, we ideally have a completely balanced distribution of keys, which means in each round we run twice the number of splits as the previous round over half-sized subproblems. So, we can assume that in the best-case scenario, recursive (or iterative) scan-based split's average running time is lower-bounded by $\log(m)$ (or m) times the runtime of a single scan-based split method. This ideal lower bound is not competitive for any of our scenarios, and thus we have not implemented this method for more than two buckets.

3.6.2 Performance versus number of buckets: $m \leq 256$

In this section we analyze our performance as a function of the number of buckets ($m \leq 256$). Our methods differ in three principal ways: 1) how expensive are our local computations, 2) how expensive are our memory accesses, and 3) how much locality can be extracted by reordering.

In general, our WMS method is faster for a small number of buckets and BMS is faster for a large number of buckets. Both are generally faster than RB-sort. There is a crossover between

WMS and BMS (a number of buckets such that BMS becomes superior for all larger numbers of buckets) that may differ based on 1) whether multisplit is key-only or key-value, and 2) the GPU architecture and its available hardware resources. Key-value scenarios require more expensive data movements and hence benefit more from reordering (for better coalesced accesses). That being said, BMS requires more computational effort for its reordering (because of multiple synchronizations for communications among warps), but it is more effective after reordering (because it reorders larger sized subproblems compared to WMS). As a result, on each device, we expect to see this crossover with a smaller number of buckets for key-value multisplit vs. key-only.

3.6.2.1 Average running time

Table 3.4 shows the average running time of different stages in each of our three approaches, and the reduced bit sort (RB-sort) method. All of our proposed methods have the same basic computational core, warp-wide local histogram, and local offset computations. Our methods differ in performance as the number of buckets increases for three major reasons (Table 3.4):

Reordering process Reordering keys (key-values) requires extra computation and shared memory accesses. Reordering is always more expensive for BMS as it also requires inter-warp communications. These negative costs mostly depend on the number of buckets m, the number of warps per block N_{warp}, and the number of threads per warp N_{thread}.

Increased locality from reordering Since block level subproblems have more elements than warp level subproblems, BMS is always superior to WMS in terms of locality. On average and for both methods, our achieved gain from locality decreases by $\frac{1}{m}$ as m increases.

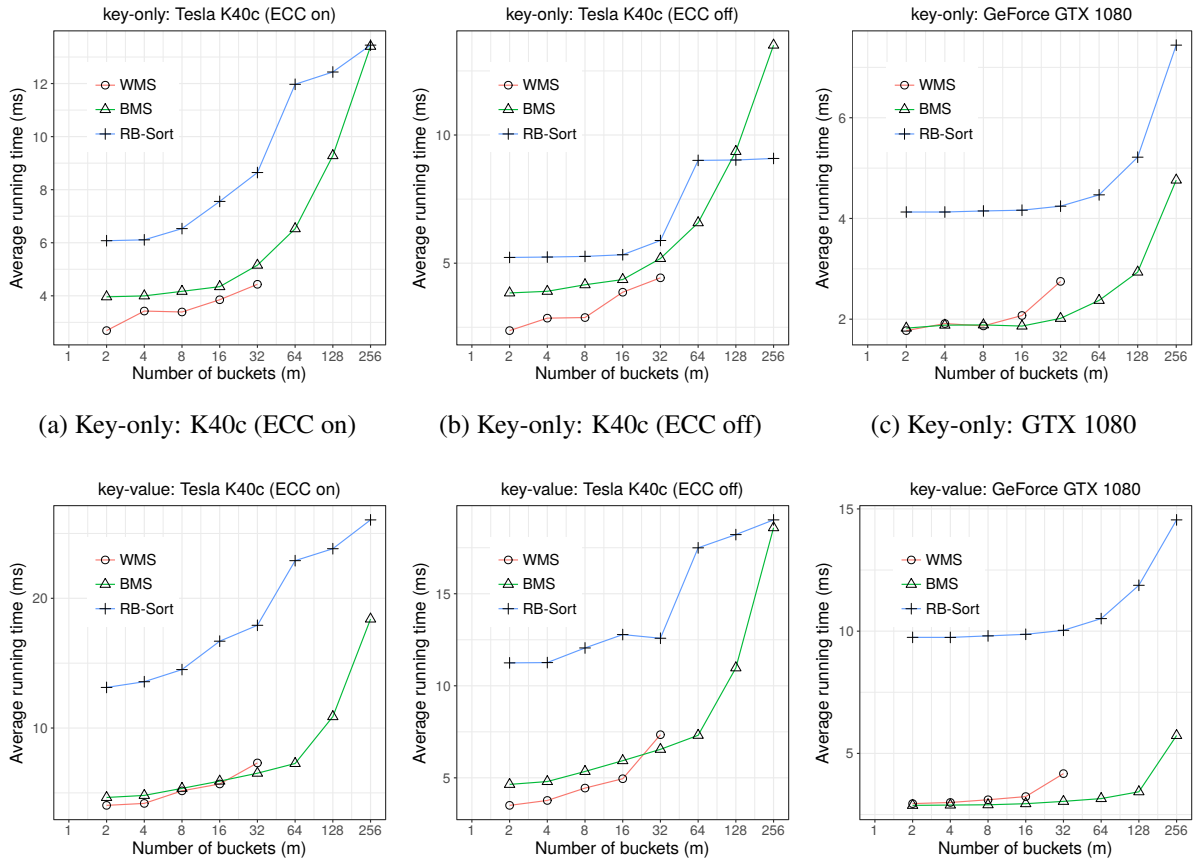
Global operations As described before, by increasing m, the height of the matrix $\mathbf{H}$ increases. However, since BMS's subproblem sizes are relatively larger (by a factor of N_{warp}), BMS requires fewer global operations compared to DMS and WMS (because the smaller width of its $\mathbf{H}$). As a result, scan operations for both the DMS and WMS get significantly more expensive, compared to other stages, as m increases (as m doubles, the cost of scan for all methods also doubles).

		Tesla K40c (ECC on)						GeForce GTX 1080					
		Key-only			Key-value			Key-only			Key-value		
		Number of buckets (m)						Number of buckets (m)					
Algorithm	Stage	2	8	32	2	8	32	2	8	32	2	8	32
DMS	Pre-scan	1.40	1.53	3.98	1.40	1.53	3.98	0.61	0.72	1.80	0.61	0.72	1.80
	Scan	0.13	0.39	1.47	0.13	0.39	1.47	0.10	0.31	1.16	0.09	0.31	1.16
	Post-scan	2.29	2.94	4.85	3.34	4.05	11.84	1.19	2.02	3.10	2.29	3.71	6.60
	Total	3.82	4.86	10.29	4.87	5.97	17.28	1.90	3.05	6.06	3.00	4.74	9.56
WMS	Pre-scan	0.79	0.93	1.38	0.89	0.97	1.39	0.58	0.60	0.93	0.59	0.62	0.93
	Scan	0.05	0.08	0.40	0.06	0.13	0.39	0.04	0.06	0.31	0.04	0.10	0.31
	Post-scan	1.85	2.38	2.66	3.09	4.06	5.53	1.15	1.20	1.51	2.32	2.38	2.94
	Total	**2.69**	**3.39**	**4.43**	**4.04**	**5.16**	7.31	**1.77**	**1.87**	2.75	2.95	3.11	4.17
BMS	Pre-scan	0.88	0.84	1.11	0.83	0.93	1.35	0.57	0.58	0.62	0.57	0.58	0.62
	Scan	0.04	0.05	0.08	0.04	0.05	0.08	0.03	0.04	0.06	0.03	0.04	0.06
	Post-scan	3.04	3.28	3.97	3.78	4.37	5.08	1.22	1.27	1.33	2.27	2.29	2.36
	Total	3.96	4.17	5.15	4.65	5.35	**6.52**	1.82	1.89	**2.02**	**2.88**	**2.90**	**3.04**
RB-sort	Labeling	1.69	1.67	1.67	1.69	1.67	1.67	1.16	1.15	1.14	1.13	1.15	1.13
	Sorting	4.39	4.87	6.98	5.81	7.17	10.58	2.97	3.00	3.11	4.11	4.16	4.38
	(un)Packing	–	–	–	5.66	5.67	5.67	–	–	–	4.51	4.50	4.52
	Total	6.08	6.53	8.65	13.13	14.51	17.92	4.13	4.15	4.24	9.75	9.81	10.04

Table 3.4: Average running time (ms) for different stages of our multisplit approaches and reduced-bit sort, with $n = 2^{25}$ and a varying number of buckets.

Figure 3.6 shows the average running time of our multisplit algorithms versus the number of buckets (m). For small m, BMS has the best locality (at the cost of substantial local work), but WMS achieves fairly good locality coupled with simple local computation; it is the fastest choice for small m (≤ 32 [key-only, Tesla K40c], ≤ 16 [key-value, Tesla K40c], and ≤ 2 [key-only, GeForce GTX 1080]). For larger m, the superior memory locality of BMS coupled with a minimized global scan cost makes it the best method overall.

Our multisplit methods are also almost always superior to the RB-sort method (except for the $m \geq 128$ key-only case on Tesla K40c with ECC off). This is partly because of the extra overheads that we introduced for bucket identification and creating the label vector, and packing/unpacking stages for key-value multisplit. Even if we ignore these overheads, since RB-sort performs its operations and permutations over the label vector as well as original key (key-value) elements, its data movements are more expensive compared to all our multisplit



(a) Key-only: K40c (ECC on) (b) Key-only: K40c (ECC off) (c) Key-only: GTX 1080

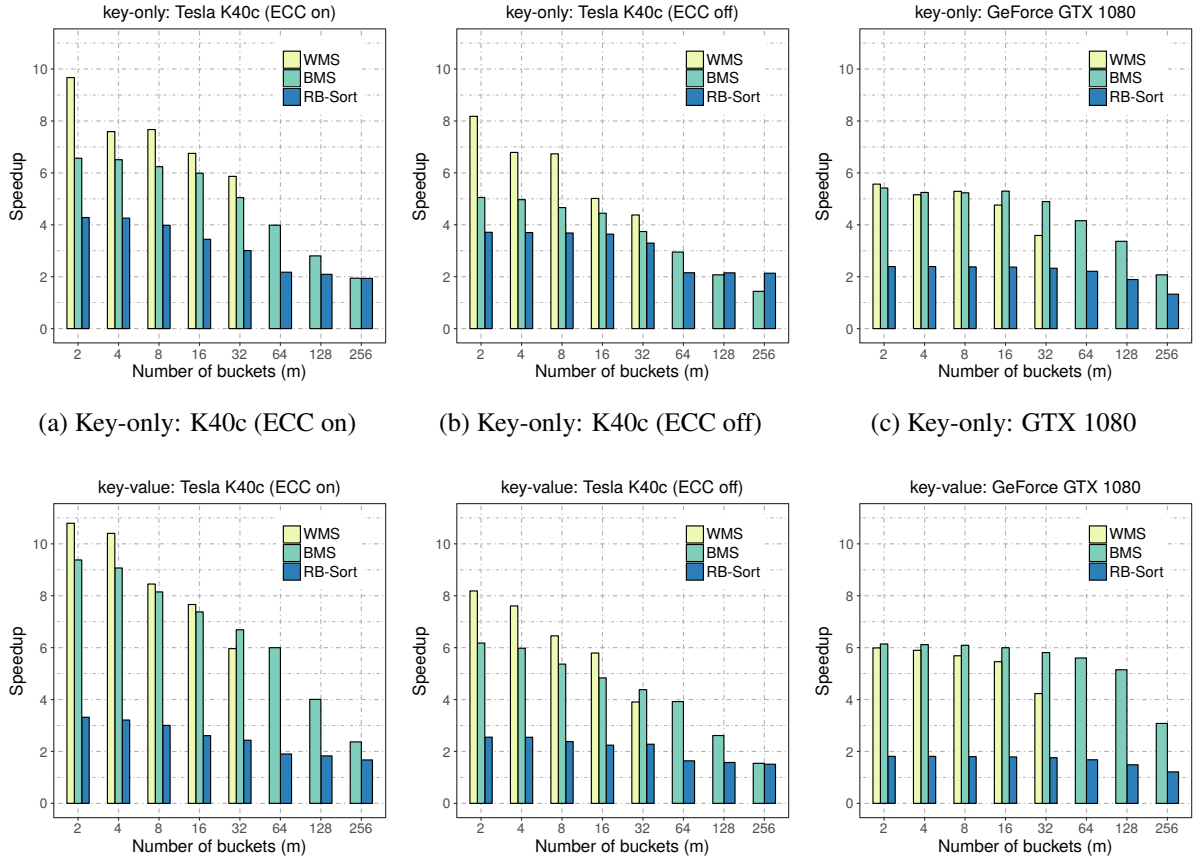
(d) Key-value: K40c (ECC on) (e) Key-value: K40c (ECC off) (f) Key-value: GTX 1080

Figure 3.6: Average running time (ms) versus number of buckets for all multisplit methods: (a,b,c) key-only, 32 M elements (d,e,f) key-value, 32 M elements.

methods that instead only process and permute original key (key-value) elements.[9]

For our user-defined delta-buckets and with a uniform distribution of keys among all 32-bit integers, by comparing Table 3.3 and Table 3.4 it becomes clear that our multisplit method outperforms radix sort by a significant margin. Figure 3.7 shows our achieved speedup against the regular 32-bit radix sort performance (Table 3.3). We can achieve up to 9.7x (and 10.8x) for key-only (and key-value) multisplits against radix sort.

[9]In our comparisons against our own multisplit methods, RB-sort will be the best sort-based multisplit method as long as our bucket identifier cannot be interpreted as a selection of some consecutive bits in its key's binary representation (i.e., $\boldsymbol{f}(u) = (u \gg k)\&(2^r - 1)$ for some k and r). Otherwise, these cases can be handled directly by a radix sort over a selection of bits (from the k-th bit until the $(k + r)$-th bit) and do not require the extra overhead that we incur in RB-sort (i.e., sorting certain bits from input keys is equivalent to a stable multisplit solution). We will discuss this more thoroughly in Section 3.7.1.



(a) Key-only: K40c (ECC on) (b) Key-only: K40c (ECC off) (c) Key-only: GTX 1080

(d) Key-value: K40c (ECC on) (e) Key-value: K40c (ECC off) (f) Key-value: GTX 1080

Figure 3.7: Achieved speedup against regular radix sort versus number of buckets for all multisplit methods: (a,b,c) key-only, (d,e,f) key-value. Both scenarios are over 32 M elements random elements uniformly distributed among buckets, and delta-bucket identifiers.

3.6.2.2 Processing rate, and multisplit speed of light

It is instructive to compare any implementation to its "speed of light": a processing rate that could not be exceeded. For multisplit's speed of light, we consider that computations take no time and all memory accesses are fully coalesced. Our parallel model requires one single global read of all elements before our global scan operation to compute histograms. We assume the global scan operation is free. Then after the scan operation, we must read all keys (or key-value pairs) and then store them into their final positions. For multisplit on keys, we thus require 3 global memory accesses per key; 5 for key-value pairs. Our Tesla K40c has a peak memory bandwidth of 288 GB/s, so the speed of light for keys, given the many favorable assumptions we have made for it, is 24 Gkeys/s, and for key-value pairs is 14.4 G pairs/s. Similarly, our GTX

1080 has 320 GB/s memory bandwidth and similar computations give us a speed of light of 26.6 G keys/s for key-only case and 16 Gpairs/s for key-value pairs.

Table 3.5 shows our processing rates for 32M keys and key-value pairs using delta-buckets and with keys uniformly distributed among all buckets. WMS has the highest peak throughput (on 2 buckets): 12.48 Gkeys/s on Tesla K40c (ECC on), 14.15 Gkeys/s on Tesla K40c (ECC off), and 18.93 Gkeys/s on GeForce GTX 1080. We achieve more than half the speed of light performance (60% on Tesla K40c and 71% on GeForce GTX 1080) with 2 buckets. As the number of buckets increases, it is increasingly more costly to sweep all input keys to compute final permutations for each element. We neglected this important part in our speed of light estimation. With 32 buckets, we reach 7.57 G keys/s on Tesla K40c and 16.64 G keys/s on GeForce GTX 1080. While this is less than the 2-bucket case, it is still a significant fraction of our speed of light estimation (32% and 63% respectively).

The main obstacles in achieving the speed of light performance are 1) non-coalesced memory writes and 2) the non-negligible cost that we have to pay to sweep through all elements and compute permutations. The more registers and shared memory that we have (fast local storage as opposed to the global memory), the easier it is to break the whole problem into larger subproblems and localize required computations as much as possible. This is particularly clear from our results on the GeForce GTX 1080 compared to the Tesla K40c, where our performance improvement is proportionally more than just the GTX 1080's global memory bandwidth improvement (presumably because of more available shared memory per SM).

3.6.2.3 Performance on different GPU microarchitectures

In our design we have not used any (micro)architecture-dependent optimizations and hence we do not expect radically different behavior on different GPUs, other than possible speedup differences based on the device's capability. Here, we briefly discuss some of the issues related to hardware differences that we observed in our experiments.

Tesla K40c It is not yet fully disclosed whether disabling ECC (which is a hardware feature and requires reboot after modifications) has any direct impact besides available memory bandwidth (such as available registers, etc.). For a very small number of buckets, our local computations are relatively cheap and hence having more available bandwidth (ECC off compared to ECC on)

			Throughput (speedup against radix-sort)							
			Number of buckets (m)							
		Method	2	4	8	16	32	64	128	256
K40c (ECC on)	key-only	DMS	8.79 (6.8 x)	8.36 (6.5 x)	6.91 (5.4 x)	6.90 (5.4 x)	3.26 (2.5 x)	–	–	–
		WMS	**12.48 (9.7 x)**	**9.79 (7.6 x)**	**9.90 (7.7 x)**	**8.71 (6.8 x)**	**7.57 (5.9 x)**	–	–	–
		BMS	8.47 (6.6 x)	8.39 (6.5 x)	8.05 (6.2 x)	7.72 (6.0 x)	6.51 (5.0 x)	**5.14 (4.0 x)**	**3.61 (2.8 x)**	**2.50 (1.9 x)**
		RB-sort	5.52 (4.3 x)	5.49 (4.3 x)	5.14 (4.0 x)	4.44 (3.4 x)	3.88 (3.0 x)	2.80 (2.2 x)	2.70 (2.1 x)	2.50 (1.9 x)
	key-value	DMS	6.90 (9.0 x)	6.31 (8.2 x)	5.62 (7.3 x)	5.62 (7.3 x)	1.94 (2.5 x)	–	–	–
		WMS	**8.31 (10.8 x)**	**8.01 (10.4 x)**	**6.51 (8.5 x)**	**5.90 (7.7 x)**	4.59 (6.0 x)	–	–	–
		BMS	7.22 (9.4 x)	6.98 (9.1 x)	6.27 (8.1 x)	5.68 (7.4 x)	**5.15 (6.7 x)**	**4.62 (6.0 x)**	**3.09 (4.0 x)**	**1.82 (2.4 x)**
		RB-sort	2.56 (3.3 x)	2.47 (3.2 x)	2.31 (3.0 x)	2.01 (2.6 x)	1.87 (2.4 x)	1.47 (1.9 x)	1.41 (1.8 x)	1.29 (1.7 x)
K40c (ECC off)	key-only	DMS	8.99 (5.2 x)	8.52 (4.9 x)	6.98 (4.0 x)	4.94 (2.9 x)	3.26 (1.9 x)	–	–	–
		WMS	**14.15 (8.2 x)**	**11.74 (6.8 x)**	**11.65 (6.7 x)**	**8.68 (5.0 x)**	**7.57 (4.4 x)**	–	–	–
		BMS	8.74 (5.1 x)	8.59 (5.0 x)	8.07 (4.7 x)	7.69 (4.4 x)	6.47 (3.7 x)	**5.10 (2.9 x)**	3.59 (2.1 x)	2.48 (1.4 x)
		RB-sort	6.42 (3.7 x)	6.40 (3.7 x)	6.37 (3.7 x)	6.30 (3.6 x)	5.70 (3.3 x)	3.72 (2.2 x)	**3.72 (2.1 x)**	**3.69 (2.1 x)**
	key-value	DMS	8.99 (7.7 x)	7.05 (6.0 x)	5.71 (4.9 x)	3.98 (3.4 x)	1.96 (1.7 x)	–	–	–
		WMS	**9.58 (8.2 x)**	**8.90 (7.6 x)**	**7.55 (6.5 x)**	**6.78 (5.8 x)**	4.57 (3.9 x)	–	–	–
		BMS	7.23 (6.2 x)	6.99 (6.0 x)	6.28 (5.4 x)	5.66 (4.8 x)	**5.13 (4.4 x)**	**4.59 (3.9 x)**	**3.06 (2.6 x)**	**1.81 (1.5 x)**
		RB-sort	2.98 (2.6 x)	2.98 (2.5 x)	2.78 (2.4 x)	2.63 (2.2 x)	2.67 (2.3 x)	1.92 (1.6 x)	1.84 (1.6 x)	1.76 (1.5 x)
GTX 1080	key-only	DMS	17.67 (5.2 x)	14.38 (4.2 x)	11.00 (3.2 x)	7.73 (2.3 x)	5.54 (1.6 x)	–	–	–
		WMS	**18.93 (5.6 x)**	17.54 (5.2 x)	**17.98 (5.3 x)**	16.18 (4.8 x)	12.20 (3.6 x)	–	–	–
		BMS	18.42 (5.4 x)	**17.84 (5.2 x)**	17.79 (5.2 x)	**18.01 (5.3 x)**	**16.64 (4.9 x)**	**14.14 (4.2 x)**	**11.43 (3.4 x)**	**7.05 (2.1 x)**
		RB-sort	8.13 (2.4 x)	8.13 (2.4 x)	8.09 (2.4 x)	8.06 (2.4 x)	7.91 (2.3 x)	7.51 (2.2 x)	6.43 (1.9 x)	4.51 (1.3 x)
	key-value	DMS	11.17 (5.9 x)	9.75 (5.1 x)	7.07 (3.7 x)	4.95 (2.6 x)	3.51 (1.8 x)	–	–	–
		WMS	11.38 (6.0 x)	11.21 (5.9 x)	10.81 (5.7 x)	10.37 (5.5 x)	8.04 (4.2 x)	–	–	–
		BMS	**11.67 (6.1 x)**	**11.62 (6.1 x)**	**11.57 (6.1 x)**	**11.40 (6.0 x)**	**11.04 (5.8 x)**	**10.64 (5.6 x)**	**9.78 (5.1 x)**	**5.85 (3.1 x)**
		RB-sort	3.44 (1.8 x)	3.44 (1.8 x)	3.42 (1.8 x)	3.40 (1.8 x)	3.34 (1.8 x)	3.19 (1.7 x)	2.83 (1.5 x)	2.31 (1.2 x)

Table 3.5: Multisplit with delta-buckets and 2^{25} random keys uniformly distributed among m buckets. Achieved processing rates (throughput) are shown in Gkeys/s (or Gpairs/s for key-value pairs). In parenthesis speedup against regular CUB's radix-sort over input elements are shown.

results in better overall performance (Table 3.5). The performance gap, however, decreases as the number of buckets increases. This is mainly because of computational bounds due to the increase in ballot, shuffle, and numerical integer operations as m grows.

CUB's radix sort greatly improves on Tesla K40c when ECC is disabled (Table 3.3), and because of it, RB-sort improves accordingly. CUB has particular architecture-based fine-grained optimizations, and we suspect it is originally optimized for when ECC is disabled to use all hardware resources to exploit all available bandwidth as much as possible. We will discuss CUB further in Section 3.7.1. RB-sort's speedups in Fig. 3.7 are relatively less for when ECC is disabled compared to when it is enabled. The reason is not because RB-sort performs worse (Table 3.5 shows otherwise), but rather because CUB's regular radix sort (that we both use in RB-sort and compare against for speedup computations) improves when ECC is disabled (Table 3.3).

GeForce GTX 1080 This GPU is based on NVIDIA's latest "Pascal" architecture. It both increases global memory bandwidth (320 GB/s) and appears to be better at hiding memory latency caused by non-coalesced memory accesses. The GTX 1080 also has more available shared memory per SM, which results in more resident thread-blocks within each SM. As a result, it is much easier to fully occupy the device, and our results (Table 3.5) show this.

3.6.3 Performance for more than 256 buckets

So far, we have only characterized problems with $m \leq 256$ buckets. As we noted in Section 3.3.3, we expect that as the number of buckets increases, multisplit converges to a sorting problem and we should see the performance of our multisplits and sorting-based multisplits converge as well.

The main obstacle for efficient implementation of our multisplits for large bucket counts is the limited amount of shared memory available on GPUs for each thread-block. Our methods rely on having privatized portions of shared memory with m integers per warp (total of $32m$ bits/warp). As a result, as m increases, we require more shared storage, which limits the number of resident thread-blocks per SM, which limits our ability to hide memory latency and hurts our performance. Even if occupancy was not the main issue, with the current GPU shared memory sizes (48 KB per SM for Tesla K40c, and 96 KB per SM to be shared by two blocks on GeForce GTX 1080), it would only be physically possible for us to scale our multisplit up to

about $m = 12\text{k}/N_{\text{warp}}$ (at most 12k buckets if we use only one warp per thread-block).

In contrast, RB-sort does not face this problem. Its labeling stage (and the packing/unpacking stage required for key-value pairs) are independent of the number of buckets. However, the radix sort used for RB-sort's sorting stage is itself internally scaled by a factor of $\log m$, which results in an overall logarithmic dependency for RB-sort vs. the number of buckets.

Solutions for larger multisplits ($m > 256$) Our solution for handling more buckets is similar to how radix sort handles the same scalability problem: iterative usage of multisplit over $m' \leq 256$ buckets. However, this is not a general solution and it may not be possible for any general bucket identifier. For example, for delta-buckets with 257 buckets, we can treat the first 2 buckets together as one single super-bucket which makes the whole problem into 256 buckets. Then, by two rounds of multisplit 1) on our new 256 buckets, 2) on the initial first 2 buckets (the super-bucket), we can have a stable multisplit result. This approach can potentially be extended to any number of buckets, but only if our bucket identifier is suitable for such modifications. There are some hypothetical cases for which this approach is not possible (for instance, if our bucket identifier is a random hash function, nearby keys do not necessarily end up in nearby buckets).

Nevertheless, if iterative usage is not possible, it is best to use RB-sort instead, as it appears to be quite competitive for a very large number of buckets. As a comparison with regular radix sort performance, on Tesla K40c (ECC on), RB-sort outperforms radix sort up to almost 32k keys and 16k key-value pairs. However, we reiterate that unlike RB-sort, which is always a correct solution for multisplit problems, direct usage of radix sort is not always a possible solution.

3.6.4 Initial key distribution over buckets

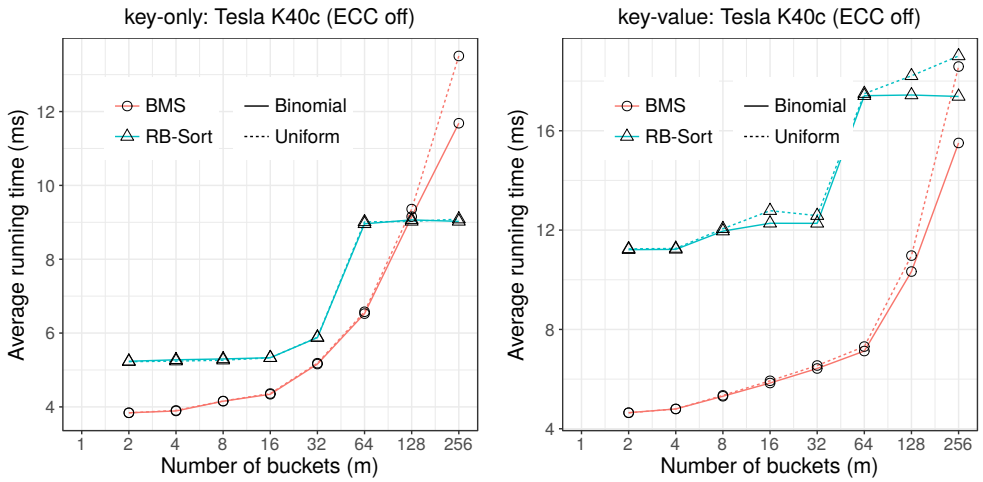
So far we have only considered scenarios in which initial key elements were uniformly distributed over buckets (i.e., a uniform histogram). In our implementations we have considered small subproblems (warp-sized for WMS and block-sized for BMS) compared to the total size of our initial key vector. Since these subproblems are relatively small, having a non-uniform distribution of keys means that we are more likely to see empty buckets in some of our subproblems; in practice, our methods would behave as if there were fewer buckets for those subproblems. All of our *computations* (e.g., warp-level histograms) are data-independent and, given a fixed bucket

count, would have the same performance for any distribution. However, our *data movement*, especially after reordering, would benefit from having more elements within fewer buckets and none for some others (resulting in better locality for coalesced global writes). Consequently, the uniform distribution is the worst-case scenario for our methods.

As an example of a non-uniform distribution, consider the binomial distribution. In general $B(m-1, p)$ denotes a binomial distribution over m buckets with a probability of success p. For example, the probability that a key element belongs to bucket $0 \leq k < m$ is $\binom{m-1}{k} p^k (1-p)^{m-k-1}$. This distribution forces an unbalanced histogram as opposed to the uniform distribution. Note that by choosing $p = 0.5$, the expected number of keys within the kth bucket will be $n\binom{m-1}{k} 2^{1-m}$. For example, with $n = 2^{25}$ elements and $m = 256$ total buckets, there will be on average almost 184 empty buckets (72%). Such an extreme distribution helps us evaluate the sensitivity of our multisplit algorithms to changes in input distribution.

Figure 3.8 shows the average running time versus the number of buckets for BMS and RB-sort with binomial and uniform distributions, on our Tesla K40c (ECC off). There are two immediate observations here. First, as the number of buckets increases, both algorithms become more sensitive to the input distribution of keys. This is mainly because, on average, there will be more empty buckets and our data movement will resemble situations where there are essentially much fewer number of buckets than the actual m. Second, the sensitivity of our algorithms increases in key-value scenarios when compared to key-only scenarios, mainly because data movement is more expensive in the latter. As a result, any improvement in our data movement patterns (here caused by the input distribution of keys) in a key-only multisplit will be almost doubled in a key-value multisplit.

To get some statistical sense over how much improvement we are getting, we ran multiple experiments with different input distributions, with delta-buckets as our bucket identifiers, and on different GPUs. Table 3.6 summarizes our results with $m = 256$ buckets. In this table, we also consider a milder distribution where αn of total keys are uniformly distributed among buckets and the rest are within one random bucket (α-uniform). BMS achieves up to 1.24x faster performance on GeForce GTX 1080 when input keys are distributed over buckets in the binomial distribution. Similarly, RB-sort achieve up to 1.15x faster on Tesla K40c (ECC off) with the



(a) Key-only (b) Key-value

Figure 3.8: Average running time (ms) vs. number of buckets (m) for two different initial key distributions: (a) a uniform distribution and (b) the binomial distribution $B(m-1, 0.5)$.

Type	Method	Distribution	Tesla K40c (ECC on)		Tesla K40c (ECC off)		GeForce GTX 1080	
Key-only	BMS	Uniform	2.50		2.48		7.05	
		0.25-uniform	2.64	1.06x	2.61	1.05x	7.36	1.05x
		Binomial	2.89	1.15x	2.87	1.16x	7.89	1.11x
	RB-sort	Uniform	2.50		3.69		4.51	
		0.25-uniform	2.69	1.08x	3.71	1.00x	4.56	1.01x
		Binomial	2.80	1.12x	3.72	1.01x	4.95	1.10x
Key-value	BMS	Uniform	1.82		1.81		5.85	
		0.25-uniform	1.99	1.10x	2.00	1.11x	6.71	1.15x
		Binomial	2.18	1.20x	2.16	1.20x	7.28	1.24x
	RB-sort	Uniform	1.29		1.77		2.31	
		0.25-uniform	1.45	1.13x	1.81	1.02x	2.33	1.01x
		Binomial	1.48	1.15x	1.93	1.9x	2.52	1.09x

Table 3.6: Processing rate (billion elements per second) as well as speedup against the uniform distribution for delta-bucket multisplit with different input distributions. All cases are with $m = 256$ buckets.

binomial distribution. In general, compared to our methods, RB-sort seems to be less sensitive to changes to the input distribution.

3.7 Multisplit, a useful building block

3.7.1 Building a radix sort

In Section 3.6, we concentrated on the performance evaluation of our multisplit methods with user-defined bucket identifiers, and in particular the delta-bucket example. In this section, we focus on identity buckets and how we can modify them to implement our own version of radix sort.

Multisplit with identity buckets Suppose we have identity buckets, meaning that each key is identical to its bucket ID (i.e., $\boldsymbol{f}(u_i) = u_i = j$, for $0 \leq j < m$). In this case, sorting keys (at least the first $\lceil \log m \rceil$ bits of keys) turns out to be equivalent to the stable multisplit problem. In this case, there is no need for the extra overheads inherent in RB-sort; instead, a direct radix sort can be a competitive alternate solution.

Radix sort In Section 3.3.3 we briefly discussed the way radix sort operates. Each round of radix sort sorts a group of bits in the input keys until all bits have been consumed. For CUB, for example, in the Kepler architecture (e.g., Tesla K40c) each group consists of 5 consecutive bits, while for the more recent Pascal architecture (e.g., GeForce GTX 1080), each group is 7 bits.[10]

Multisplit-sort Each round of radix sort is essentially bucketing its output based on the set of bits it considers in that round. If we define our buckets appropriately, multisplit can do the same. Suppose we have $\boldsymbol{f}_k(u) = (u \gg kr)\&(2^r - 1)$, where $\gg$ denotes bitwise shift to the right and $\&$ is a bitwise AND operator. r denotes our radix size (i.e., the size of the group of bits to be considered in each iteration). Then, with $0 \leq k < \lceil 32/r \rceil$ iterations of multisplit with $\boldsymbol{f}_k$ as each iteration's bucket identifier, we have built our own radix sort.

High level similarities and differences with CUB radix sort At a high level, multisplit-sort is similar to CUB's radix sort. In CUB, contiguous chunks of bits from input keys are sorted iteratively. Each iteration includes an up-sweep where bin counts are computed, a scan operation to compute offsets, and a down-sweep to actually perform the stable sorting on a selected chunk of bits (similar roles to our pre-scan, scan and post-scan stages in our multisplit). The most important differences are 1) shared memory privatization (CUB's thread-level versus our

[10] These stats are for CUB 1.6.4.

			Throughput (speedup against CUB's)							
			Number of bits in each key							
		Method	1	2	3	4	5	6	7	8
K40c (ECC on)	key-only	WMS	**14.08 (1.12 x)**	**13.88 (1.12 x)**	12.17 (0.99 x)	10.12 (0.94 x)	7.67 (1.05 x)	–	–	–
		BMS	13.84 (1.10 x)	12.94 (1.04 x)	12.10 (0.99 x)	**10.88 (1.01 x)**	**9.45 (1.29 x)**	**6.89 (1.18 x)**	4.55 (0.79 x)	2.69 (0.50 x)
		CUB	12.56 (1x)	12.45 (1x)	**12.28 (1x)**	10.75 (1x)	7.33 (1x)	5.83 (1x)	**5.74 (1x)**	**5.42 (1x)**
	key-value	WMS	**8.62 (1.13 x)**	**7.94 (1.04 x)**	6.58 (0.96 x)	5.37 (0.94 x)	4.56 (0.95 x)	–	–	–
		BMS	7.79 (1.02 x)	7.85 (1.03 x)	**7.55 (1.10 x)**	**7.16 (1.25 x)**	**6.76 (1.41 x)**	**5.23 (1.60 x)**	**3.41 (1.09 x)**	1.92 (0.67 x)
		CUB	7.61 (1x)	7.60 (1x)	6.87 (1x)	5.72 (1x)	4.79 (1x)	3.27 (1x)	3.12 (1x)	**2.86 (1x)**
K40c (ECC off)	key-only	WMS	**17.57 (1.35 x)**	**16.47 (1.26 x)**	**13.36 (1.04 x)**	10.18 (0.80 x)	7.80 (0.68 x)	–	–	–
		BMS	15.26 (1.17 x)	13.89 (1.06 x)	12.76 (0.99 x)	**10.91 (0.85 x)**	9.49 (0.82 x)	6.85 (1.06 x)	4.53 (0.70 x)	2.68 (0.42 x)
		CUB	13.05 (1x)	13.06 (1x)	12.86 (1x)	12.76 (1x)	**11.54 (1x)**	**6.50 (1x)**	**6.46 (1x)**	**6.44 (1x)**
	key-value	WMS	**10.31 (1.13 x)**	**9.67 (1.07 x)**	7.73 (0.86 x)	6.21 (0.70 x)	4.53 (0.59 x)	–	–	–
		BMS	9.01 (0.99 x)	9.00 (1.00 x)	8.50 (0.94 x)	7.69 (0.86 x)	6.78 (0.88 x)	**5.20 (1.16 x)**	3.31 (0.74 x)	1.92 (0.43 x)
		CUB	9.14 (1x)	9.00 (1x)	**9.00 (1x)**	**8.92 (1x)**	**7.69 (1x)**	4.50 (1x)	**4.46 (1x)**	**4.47 (1x)**
GTX 1080	key-only	WMS	18.87 (1.15 x)	17.70 (1.07 x)	16.76 (1.03 x)	16.33 (0.96 x)	12.56 (0.71 x)	–	–	–
		BMS	**19.37 (1.18 x)**	**19.15 (1.15 x)**	**18.99 (1.17 x)**	**18.61 (1.09 x)**	**17.89 (1.02 x)**	16.83 (0.93 x)	13.41 (0.92 x)	8.17 (0.94 x)
		CUB	16.35 (1x)	16.59 (1x)	16.24 (1x)	17.05 (1x)	17.62 (1x)	**18.05 (1x)**	**14.50 (1x)**	**8.65 (1x)**
	key-value	WMS	11.39 (1.01 x)	11.04 (1.00 x)	10.73 (0.96 x)	10.05 (0.90 x)	8.29 (0.76 x)	–	–	–
		BMS	**11.68 (1.03 x)**	**11.61 (1.05 x)**	**11.58 (1.04 x)**	**11.41 (1.02 x)**	**11.18 (1.03 x)**	**10.89 (1.07 x)**	**10.20 (1.23 x)**	**6.42 (1.20 x)**
		CUB	11.30 (1x)	11.06 (1x)	11.18 (1x)	11.14 (1x)	10.88 (1x)	10.20 (1x)	8.30 (1x)	5.34 (1x)

Table 3.7: Multisplit with identity buckets. 2^{25} random keys are uniformly distributed among buckets. Achieved throughput (Gkeys/s or Gpairs/s) are shown as well as the achieved speedup against CUB's radix sort (over limited number of bits).

warp-level), which decreases our shared memory usage compared to CUB, and 2) our extensive usage of warp-wide intrinsics versus CUB's more traditional register-level computations.

3.7.1.1 Performance Evaluation

Our multisplit-based radix sort has competitive performance to CUB's radix sort. Our experiments show that, for example on the GeForce GTX 1080, our key-only sort can be as good as 0.9x of CUB's performance, while our key-value sort can be as good as 1.1x faster than CUB's.

Multisplit with identity buckets We first compare our multisplit methods (WMS and BMS) with identity buckets to CUB's radix sort over $\lceil \log m \rceil$ bits. Multisplit and CUB's performance are roughly similar. Our multisplit methods are usually better for a small number of buckets, while CUB's performance is optimized around the number of bits that it uses for its internal iterations (5 bits for Kepler, 7 bits for Pascal). Table 3.7 shows our achieved throughput (billion elements sorted per second) as a function of the number of bits per key.

There are several important remarks to make:

- Our multisplit methods outperform CUB for up to 4 bits on the Tesla K40c and up to 6 bits on the GeForce GTX 1080. We note CUB is highly optimized for specific bit counts: 5-bit radixes on Kepler (Tesla K40c) and 7-bit radixes on Pascal (GeForce GTX 1080).

- By comparing our achieved throughputs with those of delta-buckets in Table 3.5, it becomes clear that the choice of bucket identifier can have an important role in the efficiency of our multisplit methods. In our delta-bucket computation, we used integer divisions, which are expensive computations. For example, in our BMS, the integer division costs 0.72x, 0.70x, and 0.90x geometric mean decrease in our key-only throughput for Tesla K40c (ECC on), Tesla K40c (ECC off) and GeForce GTX 1080 respectively. The throughput decrease for key-value scenarios is 0.87x, 0.82x and 0.98x respectively. The GeForce GTX 1080 is less sensitive to such computational load variations. Key-value scenarios also require more expensive data movement so that the cost of the bucket identifier is relatively less important.

- On the GeForce GTX 1080, our BMS method is always superior to WMS. This GPU appears to be better at hiding the latency of BMS's extra synchronizations, allowing the marginally better locality from larger subproblems to become the primary factor in differentiating performance.

Multisplit-sort Now we turn to characterizing the performance of sort using multisplit with identity buckets. It is not immediately clear what the best radix size (r) is for achieving the best sorting performance. As a result, we ran all choices of $4 \leq r \leq 8$. Because our bucket identifiers are also relatively simple (requiring one shift and one AND), our multisplit performance should be close to that of identity buckets.

Since BMS is almost always superior to WMS for $r \geq 4$ (Table 3.7), we have only used BMS in our implementations. For sorting 32-bit elements, we have used $\lfloor 32/r \rfloor$ iterations of r-bit BMS followed by one last BMS for the remaining bits. For example, for $r = 7$, we run 4 iterations of 7-bit BMS then one iteration of 4-bit BMS. Table 3.8 summarizes our sort results.

By looking at our achieved throughputs (sorting rates), we see that our performance increases up to a certain radix size, then decreases for any larger r. This optimal radix size is different

Method	r	K40c (ECC on) time	K40c (ECC on) throughput	K40c (ECC on) speedup	K40c (ECC off) time	K40c (ECC off) throughput	K40c (ECC off) speedup	GeFroce GTX 1080 time	GeFroce GTX 1080 throughput	GeFroce GTX 1080 speedup
Our sort (key-only)	4	25.84	1.299	1.01x	26.00	1.290	0.75x	14.11	2.368	0.70x
	5	**24.82**	**1.35**	**1.05x**	**24.81**	**1.352**	**0.78x**	12.65	2.654	0.78x
	6	26.18	1.282	0.99x	26.41	1.271	0.74x	11.44	2.933	0.86x
	7	34.59	0.970	0.75x	34.93	0.961	0.56x	**11.21**	**2.994**	**0.88x**
	8	50.17	0.669	0.52x	50.58	0.663	0.38x	14.59	2.300	0.68x
CUB (key-only)		25.99	1.291	–	19.42	1.728	–	9.88	3.397	–
Our sort (key-value)	4	36.86	0.910	1.18x	35.17	0.954	0.81x	14.11	1.441	0.75x
	5	34.90	0.962	1.25x	**34.32**	**0.978**	**0.83x**	12.65	1.619	0.85x
	6	**34.79**	**0.97**	**1.26x**	34.70	0.967	0.82x	18.01	1.852	0.97x
	7	43.90	0.764	1.00x	44.59	0.753	0.64x	**15.97**	**2.101**	**1.10x**
	8	67.47	0.497	0.65x	67.94	0.494	0.42x	18.01	1.863	0.97x
CUB (key-value)		43.73	0.767	–	28.58	1.174	–	17.56	1.911	–

Table 3.8: Our Multisplit-based radix sort is compared to CUB. We have used various number of bits (r) per iteration of mutlisplit for our various sort implementations. Average running time (ms) for sorting 2^{25} random 32-bit elements, achieved throughput (Gkeys/s for key-only, and Gpairs/s for key-value), and speedup against CUB's radix sort.

for each different GPU and depends on numerous factors, for example available bandwidth, the efficiency of warp-wide ballots and shuffles, the occupancy of the device, etc. For the Tesla K40c, this crossover point is earlier than the GeForce GTX 1080 (5 bits compared to 7 bits). Ideally, we would like to process more bits (larger radix sizes) to have fewer total iterations. But, larger radix sizes mean a larger number of buckets (m) in each iteration, requiring more resources (shared memory storage, more register usage, and more shuffle usage), yielding an overall worse performance per iteration.

Comparison with CUB CUB is a carefully engineered and highly optimized library. For its radix sort, it uses a persistent thread style of programming [38], where a fixed number of thread-blocks (around 750) are launched, each with only 64 threads (thus allowing many registers per thread). Fine-tuned optimizations over different GPU architectures enables CUB's implementation to efficiently occupy all available resources in the hardware, tuned for various GPU architectures.

The major difference between our approach with CUB has been our choice of privatization.

CUB uses thread-level privatization, where each thread keeps track of its local processed information (e.g., computed histograms) in an exclusively assigned portion of shared memory. Each CUB thread processes its portion of data free of contention, and later, combines its results with those from other threads. However, as CUB considers larger radix sizes, it sees increasing pressure on each block's shared memory usage. The pressure on shared memory becomes worse when dealing with key-value sorts as it now has to store more elements into shared memory than before.

In contrast to CUB's thread privatization, our multisplit-based implementations instead target warp-wide privatization. An immediate advantage of this approach is that we require smaller privatized exclusive portions of shared memory because we only require a privatized portion per warp rather than per thread. The price we pay is the additional cost of warp-wide communications (shuffles and ballots) between threads, compared to CUB's register-level communication within a thread.

The reduced shared memory usage of our warp privatization becomes particularly valuable when sorting key-value pairs. Our key-value sort on GeForce GTX 1080 shows this advantage: when both approaches use 7-bit radixes, our multisplit-sort achieves up to a 1.10x higher throughput than CUB. On the other hand, CUB demonstrates its largest performance advantage over our implementation (ours has 0.78x the throughput of CUB's) for key-only sorts on Tesla K40c (ECC off). In this comparison, our achieved shared memory benefits do not balance out our more costly shuffles.

Our multisplit-based radix sort proves to be competitive to CUB's radix sort, especially in key-value sorting. For key-only sort, our best achieved throughputs are 1.05x, 0.78x, and 0.88x times the throughput that CUB provides for Tesla K40c (ECC on), Tesla K40c (ECC off), and GeForce GTX 1080, respectively. For key-value sorting and with the same order of GPU devices, our multisplit-based sort provides 1.26x, 0.83x, and 1.10x times more throughput than CUB, respectively. Our highest achieved throughput is 3.0 Gkeys/s (and 2.1 Gpairs/s) on a GeFroce GTX 1080, compared to CUB's 3.4 Gkeys/s (and 1.9 Gpairs/s) on the same device.

Future of warp privatized methods We believe the exploration of the difference between thread-level and warp-level approaches has implications beyond just multisplit and its extension

to sorting. In general, any future hardware improvement in warp-wide intrinsics will reduce the cost we pay for warp privatization, making the reduction in shared memory size the dominant factor. We advocate further hardware support for warp-wide voting with a generalized ballot that returns multiple 32-bit registers, one for each bit of the predicate. Another useful addition that would have helped our implementation is the possibility of shuffling a dynamically addressed register from the source thread. This would enable the user to share lookup tables among all threads within a warp, only requesting the exact data needed at runtime rather than delivering every possible entry so that the receiver can choose.

3.7.2 The Single Source Shortest Path problem

In Section 3.1 we argued that an efficient multisplit primitive would have helped Davidson et al. [26] in their delta-stepping formulation of the Single Source Shortest Path (SSSP) problem on the GPU. In this section, we show that by using our multisplit implementation, we can achieve significant speedups in SSSP computation, especially on highly connected graphs with low diameters.

3.7.2.1 The Single Source Shortest Path (SSSP) problem

Given an arbitrary graph $G = (V, E)$, with non-negative weights assigned to each edge and a source vertex $s \in V$, the SSSP problem finds the minimum cost path from the source to every other vertex in the graph. As described in Section 3.1, Dijkstra's [31] and Bellman-Ford-Moore's [10] algorithms are two classical approaches to solve the SSSP problem. In the former, vertices are organized in a single priority queue and are processed sequentially from the lowest to the highest weight. In the latter, for each vertex we process all its neighbors (i.e., processing all edges). This can be done in parallel and is repeated over multiple iterations until convergence. Dijkstra is highly work-efficient but essentially sequential and thus unsuitable for parallelization. Bellman-Ford-Moore is trivially parallel but does much more work than necessary (especially for highly connected graphs).

As an alternative algorithm between these two extremes (sequential processing of all vertices vs. processing all edges in parallel), delta-stepping allows the selective processing of a subset of vertices in parallel [70]. In this formulation, nodes are put into different buckets (based on their assigned weights) and buckets with smaller weights are processed first. Davidson et al. [26]

proposed multiple GPU implementations based on the delta-stepping formulation. Their two most prominent implementations were based on a *Near-Far* strategy and a *Bucketing* strategy. Both divide vertices into multiple buckets, which can be processed in parallel. Both use efficient load-balancing strategies to traverse all vertices within a bucket. Both iterate over multiple rounds of processing until convergence is reached. The main difference between the two is in the way they organize the vertices to be processed next (work frontiers):

Near-Far strategy In this strategy the work queue is prioritized based on a variable splitting distance. In every iteration, only those vertices less than this threshold (the *near set*) are processed. Those falling beyond the threshold are appended to a *far pile*. Elements in the far pile are ignored until work in the near set is completely exhausted. When all work in the near set is exhausted (this could be after multiple relaxation phases), this strategy increases the splitting distance (by adding an incremental weight Δ to it) and removes invalid elements from the far pile (those which have been updated with similar distances), finally splitting this resulting set into a new near set and far pile. This process continues until both the near set and far pile are empty (the convergence criterion).

Bucketing strategy In this strategy, vertices are partitioned into various buckets based on their weights (Davidson et al. reported the best performance resulted from 10 buckets). This strategy does a more fine-grained classification of vertices compared to Near-Far, resulting in a greater potential reduction in work queue size and hence less work necessary to converge. The downside, however, is the more complicated bucketing process, which due to lack of an efficient multisplit primitive was replaced by a regular radix sort in the original work. As a result of this expensive radix sort overhead, Near-Far was more efficient in practice [26].

3.7.2.2 Multisplit-SSSP

Now that we have implemented an efficient multisplit GPU primitive in this chapter, we can use it in the Bucketing strategy explained above to replace the costly radix sort operation. We call this new Bucketing implementation *Multisplit-SSSP*. Our Multisplit-SSSP should particularly perform well on highly connected graphs with relatively large out degrees and smaller diameters

(such as in social graphs), causing fewer iterations and featuring large enough work fronts to make multisplit particularly useful. However, graphs with low average degrees and large diameters (such as in road networks) require more iterations over smaller work frontiers, resulting in high kernel launch overheads (because of repetitive multisplit usage) without large enough work frontiers to benefit from the efficiency of our multisplit. We note that this behavior for different graph types is not limited to our SSSP implementation; GPU graph analytics in general demonstrate their best performance on highly connected graphs with low diameters [94].

3.7.2.3 Performance Evaluation

In this part, we quantitatively evaluate the performance of our new Multisplit-SSSP compared to Davidson et al.'s Bucketing and Near-Far approaches. Here, we choose a set of graph datasets listed in Table 3.9.[11] For those graphs that are not weighted, we randomly assign a non-negative integer weight between 0 and 1000 to each edge.

Table 3.10 shows the convergence time for Near-Far, Bucketing, and Multisplit-SSSP (in million traversed edges per second, MTEPS), with Multisplit-SSSP's speedup against Near-Far. Multisplit-SSSP is always better than Bucketing, on both devices and on every graph we tested (up to 9.8x faster on Tesla K40c and 9.1x faster on the GeForce GTX 1080). This behavior was expected because of the performance superiority of our multisplit compared to a regular radix-sort (Fig. 3.7).

Against Near-Far, our performance gain depends on the type of graph. As we expected, on highly connected graphs with low diameters (such as rmat), we achieve up to 1.58x and 2.17x speedup against Near-Far, on the Tesla K40c and GeForce GTX 1080 respectively. However, for high diameter graphs such as road networks (e.g., belgium_osm), we are closer to Near-Far's performance: Multisplit-SSSP is slower than Near-Far on Tesla K40c (0.93x) and marginally faster on GeForce GTX 1080 (1.04x). Road graphs have significantly higher diameter and hence more iterations. As a result, the extra overhead in each phase of Multisplit-SSSP on large diameters can become more important than the saved operations due to fewer edge re-relaxations.

[11] All matrices except for rmat are downloaded from University of Florida Sparse Matrix Collection [27]. Rmat was generated with parameters $(a, b, c, d) = (0.5, 0.1, 0.1, 30)$.

Graph Name	Vertices	Edges	Avg. Degree
cit-Patents [39]	3.77 M	16.52 M	8.8
flickr [27]	0.82 M	9.84 M	24.0
belgium_osm [54]	1.44 M	1.55 M	2.2
rmat [21]	0.8 M	4.8 M	12.0

Table 3.9: Datasets used for evaluating our SSSP algorithms.

	Tesla K40c (ECC on)					GeForce GTX 1080				
Graph Name	Near-Far	Bucketing	Multisplit-SSSP			Near-Far	Bucketing	Multisplit-SSSP		
–	time (ms)	time (ms)	time (ms)	MTEPS	speedup	time (ms)	time (ms)	time (ms)	MTEPS	speedup
cit-Patents	458.4	3375.9	343.1	96.3	1.34x	444.2	3143.0	346.8	95.2	1.28x
flickr	96.0	163.0	64.5	305.2	1.49x	66.7	111.1	36.5	539.0	1.83x
belgium_osm	561.4	3588.0	604.5	5.12	0.93x	443.8	3014.2	427.0	7.3	1.04x
rmat	20.9	28.7	13.2	727.3	1.58x	12.17	14.9	5.8	1655.2	2.17x

Table 3.10: Near-Far, Bucketing, and our new Multisplit-SSSP methods over various datasets. Speedups are against the Near-Far strategy (which appears to be always better than the Bucketing strategy).

3.7.3 GPU Histogram

All three of our multisplit methods from Section 3.5 (DMS, WMS and BMS) have a pre-scan stage, where we compute bucket histograms for each subproblem using our warp-wide ballot-based voting scheme (Algorithm 3). In this section, we explore the possibility of using our very same warp-wide histogram to compute a *device-wide* (global) histogram. We define our histogram problem as follows: The inputs are n unordered input elements (of any data type) and a bucket identifier $\boldsymbol{f}$ that assigns each input element to one of m distinct buckets (bins). The output is an array of length m representing the total number of elements within each bucket.

Our Histogram In the pre-scan stage of our multisplit algorithms, we store histogram results for each subproblem so that we can perform a global scan operation on them, then we use this result in our post-scan stage to finalize the multisplit. In the GPU Histogram problem, however, we no longer need to report per-subproblem histogram details. Instead, we only must sum all subproblems' histograms together to form the output global histogram. Clearly, we would prefer the largest subproblems possible to minimize the cost of the final global summation. So, we base our implementation on our BMS method (because it always addresses larger subproblems than

the other two). We have two main options for implementation. 1) Store our subproblem results into global memory and then perform a segmented reduction, where each bucket represents a segment. 2) Modify our pre-scan stage to atomically add histogram results of each subproblem into the final array. Based on our experiments, the second option appears to be more efficient on both of our devices (Tesla K40c and GeForce GTX 1080).

Experimental setup In order to evaluate a histogram method, it is common to perform an extensive set of experiments with various distributions of inputs to demonstrate the performance and consistency of that method. A complete and thorough benchmarking of all possible distributions of inputs is beyond the scope of this short section. Nevertheless, just to illustrate the potentials in our histogram implementation, we continue this section with a few simple experimental scenarios. Our goal is to explore whether our warp-wide histogram method can potentially be competitive to others (such as CUB's histogram), under what conditions, and most importantly why. To achieve this goal, we consider the following two scenarios:

1. Even Histogram: Consider a set of evenly spaced splitters $\{s_0, s_1, \ldots, s_m\}$ such that each two consecutive splitters bound a bucket (m buckets, each with a width of $|s_i - s_{i-1}| = \Delta$). For each real number input $s_0 < x < s_m$, we can easily identify its bucket as $\lfloor (x - s_0)/\Delta \rfloor$.

2. Range Histogram: Consider a set of arbitrarily ranged splitters $\{s_0, s_1, \ldots, s_m\}$ such that each two consecutive splitters bound a bucket. For each real number input $s_0 < x < s_m$, we must perform a binary search (i.e., an upper bound operation) on splitters to find the appropriate bucket (requires at most $\lceil \log m \rceil$ searches).

We generate $n = 2^{25}$ random floating point numbers uniformly distributed between 0 and 1024. For the Even histogram experiment, splitters are fixed based on the number of buckets m. For Range histogram experiment, we randomly generate a set of $m - 1$ random splitters (s_0 and s_m are already fixed). We use our histogram method to compute the global histogram for $m \leq 256$ buckets. We compare against CUB Histogram, which supports equivalents (`HistogramEven` and `HistogramRange`) to our Even and Range test scenarios. We have repeated our experiments over 100 independent random trials.

Example	GPU	Method	Number of buckets (bins) 2	4	8	16	32	64	128	256
Even Histogram	Tesla K40c (ECC on)	Our Histogram	**45.3**	**42.5**	**45.4**	**37.8**	**32.1**	**26.5**	**24.2**	18.8
		CUB	13.7	14.9	16.9	19.1	21.4	21.8	20.8	**19.5**
		Speedup vs. CUB	3.30x	2.86x	2.69x	1.98x	1.50x	1.21x	1.16x	0.96x
	Tesla K40c (ECC off)	Our Histogram	**53.0**	**47.2**	**48.1**	**38.3**	**32.3**	**26.5**	**24.0**	18.7
		CUB	13.6	14.7	16.7	18.9	21.3	21.8	20.7	**19.5**
		Speedup vs. CUB	3.90x	3.20x	2.88x	2.03x	1.52x	1.21x	1.16x	0.96x
	GeForce GTX 1080	Our Histogram	**61.0**	**61.1**	**60.9**	60.7	60.2	45.2	59.6	52.7
		CUB	60.5	60.7	60.5	**60.7**	**61.1**	**60.6**	**60.3**	**60.9**
		Speedup vs. CUB	1.01x	1.01x	1.01x	1.00x	0.98x	0.75x	0.99x	0.87x
Range Histogram	Tesla K40c (ECC on)	Our Histogram	**28.0**	**22.1**	**18.4**	**14.6**	**11.9**	**9.0**	**7.7**	**7.3**
		CUB	8.7	6.8	6.2	5.8	5.7	5.5	5.2	4.8
		Speedup vs. CUB	3.21x	3.26x	2.96x	2.51x	2.09x	1.63x	1.50x	1.51x
	Tesla K40c (ECC off)	Our Histogram	**27.6**	**22.2**	**17.8**	**14.5**	**11.7**	**8.7**	**7.6**	**7.1**
		CUB	8.4	6.8	6.2	5.8	5.6	5.4	5.1	4.8
		Speedup vs. CUB	3.29x	3.28x	2.89x	2.50x	2.10x	1.61x	1.50x	1.50x
	GeForce GTX 1080	Our Histogram	**56.7**	**51.4**	**45.4**	**39.8**	**33.9**	**28.4**	**24.8**	**20.0**
		CUB	42.4	35.2	30.9	27.1	24.4	22.3	19.3	14.8
		Speedup vs. CUB	1.34x	1.46x	1.47x	1.47x	1.39x	1.28x	1.29x	1.35x

Table 3.11: Histogram computation over two examples of even bins (Even Histogram) and customized bins (Range Histogram). Procesing rates (in billion elements per second) are shown for our Histogram and CUB as well as our achieved speedup. Experiments are repeated on three different hardware settings.

3.7.3.1 Performance Evaluation

Table 3.11 shows our achieved average processing rate (in billion elements per second) as well as our speedup against CUB, and for different hardware choices. For the Even Histogram, we observe that we are better than CUB for $m \leq 128$ buckets on Tesla K40c, but only marginally better for $m \leq 8$ on GeForce GTX 1080. For Range Histogram, we are always better than CUB for $m \leq 256$ on both devices.

Even Histogram CUB's even histogram is designed to operate with any number of buckets (even $m \gg 256$). This generality has consequences in its design choices. For example, if an implementation generalizes to any number of buckets (especially large $m > 256$), it is not

possible to privatize histogram storage for each thread (which requires size proportional to m) and then combine results to compute a block-level histogram solution. (This is a different scenario than radix sort, because radix sort can choose the number of bits to process on each iterations. CUB's radix sort only supports histograms of size up to 128 buckets [at most 7 bits] on Pascal GPUs). As a result, CUB uses shared memory atomics to directly compute histograms in shared memory. Then these intermediate results are atomically added again into global memory to form the output global histogram. (Since CUB is mostly bounded by atomic operations, ECC does not have much effect on its performance on Tesla K40c.)

On the other hand, our focus here is on a limited number of buckets. As a result, by using warp-level privatization of histograms, we avoid shared memory atomics within blocks, resulting in better performance for the histogram bucket counts that we target. As the number of buckets increases, we gradually increase pressure on our shared memory storage until CUB's histogram becomes the better choice.

Atomic operation performance has improved significantly on the Pascal architecture (GeForce GTX 1080), which helps CUB's histogram performance. On this architecture, Table 3.11 shows that that we are barely better than CUB for a very small number of buckets ($m \leq 8$), and then we witness a decrease in our performance because of the shared-memory pressure that we explained above.

Range Histogram We see better performance for range histograms for two main reasons. First, bucket identification in this case requires a binary search, which is much more expensive than the simpler floating point multiplications required for the even histogram. Our histogram implementation is relatively insensitive to expensive bucket-computation operations because they help us hide any extra overheads caused by our extensive shuffle and ballot usage. A thread-level approach like CUB's would also have to do the same set of expensive operations (in this case, expensive memory lookups), but then they would lose the comparative benefit they would otherwise gain from faster local register-level computations.

Another reason for our better performance is again because of CUB's generality. Since CUB must operate with an arbitrary number of splitters, it does not store those splitters into shared memory, which means every binary search is directly performed in global memory. On the

other hand, for $m \leq 256$, we can easily store our splitters into shared memory in order to avoid repetitive global memory accesses.

Summary Specializing a histogram computation to support only a limited number of buckets allows potential performance increases over more general histogram implementations. For some applications, this may be desirable. For other applications—and this is likely CUB's design goal—consistent performance across an arbitrary number of buckets may be more important. Nonetheless, multisplit was the key building block that made these performance increases possible.

3.8 Conclusion

The careful design and analysis of our GPU multisplit implementations allow us to provide significant performance speedups for multisplit operations over traditional sort-based methods. The generality of our multisplit algorithm let us extend it further into other interesting applications, such as building a competitive GPU radix sort, getting significant improvements in Single Source Shortest Path problem, and providing a promising GPU histogram mostly suitable for small number of buckets. Beyond simply demonstrating the design and implementation of a family of fast and efficient multisplit primitives, we offer three main lessons that are broadly useful for parallel algorithm design and implementation: Considering a warp-synchronous programming by leveraging warp-wide hardware intrinsics and promoting warp-level privatization of memory, where applicable, can potentially brings interesting and efficient implementations; Minimize global (device-wide) operations, even at the cost of increased local computation; the benefit of more coalesced memory accesses outweighs the cost of local reordering.

Chapter 4

GPU LSM: Dynamic Dictionary Data Structure for the GPU[1]

4.1 Introduction

The choice of the right data structure for a computation can have an enormous impact on performance. The CPU world contains a wealth of data structures optimized for particular operations; the GPU world does not. For general-purpose use, GPUs basically have three data structures: an unordered array, a sorted array, and a hash table. Application-specific data structures, such as acceleration tree data structures used in ray tracing [99], may also be useful for general-purpose tasks. Historically, complex data structures used by the GPU have been built on the CPU and then copied to the GPU (for example, perfect spatial hash tables [58]), but recent work in, for example, cuckoo [3] and Robin Hood hashing [36] and bounding volume hierarchy (BVH) trees [50] has focused on data structures that can be efficiently constructed directly on the GPU. Efficient construction of a data structure on the GPU is a complicated task that, if efficiently implemented, is rewarded by much higher available memory bandwidth compared to any CPU-GPU data transfers (e.g., the Tesla K40c has 288 GB/s internal memory bandwidth, compared to the 16 GB/s provided by PCIe 3.0 interconnects).

None of these mentioned data structures are efficiently mutable (whether built on the GPU or the CPU). GPU programmers mostly do not even consider applications that require modifications

[1]This chapter substantially appeared as "GPU LSM: A Dynamic Dictionary Data Structure for the GPU" [9], for which I was responsible for most of the research and writing.

to their data structures. Providing concurrency to and guaranteeing correctness within a mutable data structure on a GPU with such massive parallelism is a significant challenge. This becomes an even more complicated task if we also expect to fully exploit the potential of our hardware to achieve high performance. As a result, programmers hardly ever consider taking this path and almost all data structures on GPUs cannot be modified without rebuilding the whole data structure from scratch. NVIDIA's OptiX ray tracing engine [83], for instance, with very limited exceptions [55], entirely rebuilds its BVH tree every frame.

We believe that supporting *dynamic* GPU data structures—data structures that can be efficiently updated on the GPU—will enlarge the scope of problems that can be addressed within GPU computing applications. Currently, the complete lack of dynamic data structures on the GPU has inhibited the development of applications that might benefit from them. Many potential applications might benefit from efficient dynamic GPU data structures: processing dynamic graphs and trees on the GPU; processing moving objects (e.g., real-time range queries to find k nearest neighbors for all moving objects in a 2D plane [60]); processing spatial data (e.g., real-time tweet visualization from a user-defined geographical region [75]); and dynamic memory allocators on the GPU [92].

In this chapter we begin this study by considering a dynamic version of one of the most basic abstract data types, the dictionary. Serial examples of dictionary data structures include skip lists and balanced trees such as red-black trees or AVL trees.

4.1.1 Dynamic data structures for the GPU: the challenge

Consider any dynamic data structure on the GPU, that is, one that supports inserting and deleting items; for simplicity, we will discuss only insertions. Depending on the number of items to be inserted, we might find ourselves in one of three scenarios:

Insert one/a few items In this case, we could choose to fall back to serial algorithms and insert items one at a time. It is unlikely that such an insertion will be able to fully occupy a modern GPU because it does not involve enough work.

Insert a very large number of items If the number of items to insert is on the order of the number of items already in the data structure, or larger, it is likely most efficient to simply

rebuild the whole data structure.

Insert a moderate number of items The interesting case, which we address in this chapter, is the very large middle ground between serial and rebuild. We focus our efforts on efficiently supporting dynamic operations where an update of the data structure is potentially less expensive than a full rebuild.

We will define a *batch* operation on the GPU as one that falls into this third scenario; the number of items to insert or delete is large enough that the operations can profitably be done in parallel but not so large that we might as well rebuild the data structure. The reason this third case is such a challenge is that by increasing the number of concurrent modifications to a shared data structure, it increasingly becomes harder to cooperate while maintaining correctness. For example, in a balanced tree data structure, inserting an item requires rebalancing the tree, and parallel inserts might require rebalancing operations that interfere with each other. Our challenge is to find or design data structures that support batched operations and to implement efficient algorithms for them.

4.1.2 Dictionaries

Formally, a *dictionary* maintains a set $\mathcal{S}$ of $\langle \textbf{\textit{key}}, \textbf{\textit{value}} \rangle$ pairs, where keys are totally ordered, that supports the following operations:

- INSERT(k, v): $\mathcal{S} \leftarrow (\mathcal{S} - \{\langle k, * \rangle\}) \cup \{\langle k, v \rangle\}$. That is, if we insert a new item whose key matches an item already in the set, we replace the old item.
- DELETE(k): $\mathcal{S} \leftarrow \mathcal{S} - \{\langle k, * \rangle\}$. That is, remove all key-value pairs with key k.
- LOOKUP(k): return $\langle k, v \rangle \in \mathcal{S}$, or $\perp$ if no such key exists.
- COUNT(k_1, k_2): return the number of pairs $\langle k, * \rangle$ such that $k_1 \leq k \leq k_2$.
- RANGE(k_1, k_2): returns all pairs $\langle k, * \rangle$ such that $k_1 \leq k \leq k_2$.

We will discuss our assumptions for the semantics of batch operations later in Section 4.3.1.

Existing GPU data structures (sorted array and hash tables) can implement all of these operations, but not necessarily efficiently. For example, a sorted array performs the read-only

operations well (lookup, count, and range queries). However, modifying a sorted array (even by the insertion or deletion of a single element) potentially requires resorting or merging the entire data structure and hence a large number of memory operations. Cuckoo hashing, for instance, supports efficient lookup [3]. But, it does not support any updates to the hash table after building. It is also an unordered data structure, which makes count and range queries impractical.

The lack of existing mutable data structures on GPUs emphasizes the challenge in identifying and/or designing a data structure that supports a set of interesting operations with good concurrency and performance while simultaneously allowing mutability. We began this project with the following data-structure requirements: 1) enable efficient, concurrent mutable insertion and deletion operations ("updates") at a cheaper cost than completely rebuilding the data structure; 2) support a full set of queries (e.g., lookup, count and range); and 3) allow parallel execution of these queries to efficiently exploit the massive parallelism offered by modern GPUs while guaranteeing correctness at any time. Although some GPU data structures focus on a subset of these requirements (with the notable exception of mutability), we are interested in supporting all three. We fully expect that a data structure with no support for mutability should outperform one with that support, but we expect that incremental changes to a mutable data structure will be considerably cheaper than an entire rebuild of a non-mutable data structure.

4.2 Background and Previous Work

GPU Data Structures On the GPU, data structure research began as an outgrowth of computer graphics, with the seminal GPU-computing language Brook [20] supporting uniform 2D and 3D grids. Glift [59] was the first to identify GPU data structures as a standalone field of study and implemented the GPU's first adaptive and multiresolution 2D and 3D grids. CUDA supports uniform 1D, 2D, and 3D grids natively, and grid data structures for simulation remain an active area of study (e.g., Breitbart [17]).

In this work, we concentrate on a dictionary data structure that supports numerous queries (Section 4.1.2), including ordered queries (such as count, range, predecessor, successor, etc.). On GPUs, there have been some efforts to support range queries in particular data structures. To name a few: Yang et al. implemented grid files (built on CPU) for multidimensional database

queries [96]. Kim et al. built an architecture sensitive binary tree [52] for both CPUs and GPUs. Fix et al. implemented a brute force GPU method for lookup and range queries on a CPU-built B+ tree, targeted for database systems [34]. Range queries have also been used for spatial data structures such as R-trees [64, 97] or for processing moving objects [60, 61]. In general, although dictionaries with range queries are useful and some GPU implementations deliver impressive query performance, there are not yet any such data structures that support general updates on the GPU. All the work we cite above requires a complete rebuild for updating the data structure, which is even more expensive considering the fact that most of them can only be built on the CPU and require data transfers back and forth to the GPU. Other related work includes Kaczmarski [46], who specifically targets the bulk-update problem with a combined CPU-GPU approach that contains optimizations beyond rebuilding the entire data structure; and Huang et al. [44], who extend Kaczmarski's work but with non-clustered indexes that would be poorly suited for range queries.

If range queries were not of interest, we might choose hash tables [3, 36] or non-clustered B-trees, but prior GPU work in these areas is also not efficiently updatable.

In-memory dictionaries Dozens of data structures implement dictionaries, including 2-3 trees, red-black trees, AVL tree, treaps, skip lists, and many more [24]. These data structures are used on CPUs when the data is memory-resident; they are not optimized for locality of reference. They all share the trait that they have $O(\log n)$ levels, and each level might induce a memory fault.

External memory dictionaries A different set of data structures were devised for cases where memory faults dominate the cost of using the dictionary. Because this setting emphasizes locality of reference, it has interesting analogies to the GPU context. The most widely used external memory dictionary is the B-tree [12, 22]. A B-tree is a search tree in which each non-leaf node (except possibly the root) has $\Theta(B)$ children, where B is chosen so that $B/2$ keys fit in a page. Internal nodes contain only pivot keys and child pointers; the actual key-value pairs are stored in the leaves. A B-tree with n items has height $\Theta(\log_B n)$. Since all the keys in a page are fetched together, a B-tree is faster than a balanced binary tree by a factor of $\log B$, which can be quite significant, depending on the size of keys and the size of pages.

B-trees enjoy an optimal number of page misses during searches, but they are suboptimal for insertions, deletions and updates [18]. There are some data structures that target better insertions while providing optimal searches (or somewhat worse). The two most influential data structures among these are inspirations for this work: the Log-structured Merge-tree (LSM) [81] and the Cache Oblivious Lookahead Array (COLA) [14]. The basic idea of an LSM is to keep a set of dictionaries, each a constant factor larger than the one before. As insertions arrive, they are placed in a small dictionary. When a dictionary reaches capacity, it is merged with the next larger dictionary, and so on. In an LSM, a query must be performed at each level. Therefore, LSMs usually have a B-tree at each level. Since $O(\log n)$ searches must be performed, the search time in an LSM takes $O(\log n \log_B n)$ I/Os in the worst case, making them slower for queries than B-trees. The improvement comes from reducing the number of I/Os needed to insert. Also, since items are moved from one level of the LSM to the next in a batch, the insertions in the next level enjoy some locality of reference. LSMs are asymptotically much faster than B-trees for insertions.

A COLA replaces the B-tree at each level with a sorted array. This would seem to make searches slower, since, as noted above, B-trees perform searches faster than binary search into an array. To get fast search times, pointers are maintained between levels via fractional cascading. Thus COLAs enjoy the same search speed as B-trees while achieving the same insertion complexity as the LSM.

4.3 The GPU LSM

In this chapter, we propose a GPU dictionary data structure—the *GPU LSM*—that combines the general structure of the LSM and the use of sorted arrays as in the COLA. Our goal in design is to leverage the theoretical approach of building a cache-efficient data structure while at the same time using the best practices from a GPU implementation perspective. In general LSMs are not cache-oblivious (e.g., with a B-tree per level). However, we use a sorted array (a cache-oblivious data structure) for each level (standard for COLAs and not for LSMs). This makes our GPU LSM a good candidate for GPU hardware that provides relatively small cache sizes.[2]

[2]For example, the NVIDIA Tesla K40c provides 16 KB L1-cache per SM, 1.5 MB L2-cache to be shared among all SMs, and 48 KB manually managed shared memory per SM.

In our data structure, we do not maintain inter-level pointers for fractional cascading as in the COLA because of the implementation complexity and performance implications that it would create for parallel updates on the GPU. In practice, we observe (Section 4.5.3.1) that this deficiency does not affect our query results much (theoretically we can perform lookup queries in $O(\log^2 n)$ compared to the optimal $O(\log n)$ in the COLA with fractional cascading [14]). Our GPU LSM supports concurrent updates (insertions and deletions) as well as all queries listed in Section 4.1.2.[3]

What we particularly like about the LSM and the COLA (introduced in Section 4.2) is that by their hierarchical (exponentially scaled) structure, they almost always prevent some parts of the data structure from being touched during insertions or deletions. Moreover, as we shall see, we can implement the updates in a consistent way using only two primitives, sort and merge. Both sort and merge are bulk primitives that have efficient parallel formulations for GPUs. As a result, we would expect to be able to handle high rates of insertions (and deletions). Having a sorted array per level also gives us good locality of reference, a requirement for any high-performance GPU program.

4.3.1 Batch Operation Semantics

Because we expect serial operations on a GPU data structure to be uncompetitive, we instead focus on *bulk* operations, consistent with the GPU's bulk-synchronous programming model. We do not send one single query or one single update to the data structure but instead a batch of queries or updates, so that we can exploit concurrency across multiple operations. For queries this is straightforward: they do not modify the data structure, so multiple queries can run simultaneously without correctness issues.[4] For updates, correctness is more challenging. The original LSM and COLA have serial but not bulk semantics, so we begin by defining the specific semantics we use in bulk operations.

1. The GPU LSM has a fixed batch size, defined by a parameter b, which is also the size of the first level.

[3] In this work we have only considered LOOKUP, COUNT and RANGE queries. However, it is straightforward to support other order based queries such as finding a successor or a predecessor of a certain key.

[4] For completeness, we have also considered individual query implementations, where, for instance, a single thread can issue a new lookup query asynchronously and independently from others.

2. All update operations are in a batch of size b. We may have mixed batches of insertions and deletions. The choice of b depends on the user's preference. Query operations can be in batches of any size.

3. If we insert items with the same key in different batches, only the most recently inserted is reported by a query. All previously inserted items with that key become *stale*. The notion of time is discretized based on the order of batch insertions.

4. If we insert multiple items with the same key in the same batch, an arbitrary one is chosen as the current valid item.

5. After deleting a key, all instances of that key inserted previously are considered deleted and become *stale*. Multiple deletions of the same key within a batch have the same effect as one deletion.

6. A key that is inserted and deleted within the same batch is considered deleted.

Our strategy for implementing these semantics is to tolerate stale elements (deleted elements and duplicate insertions) in the data structure, but to guarantee that they do not affect query results. Periodically, the user can choose to clean up (flush) the data structure, removing all stale elements and improving query efficiency (as a direct result of size reduction).

4.3.2 Insertion

Insertions are where LSMs and COLAs really shine, and in external memory, their insertion performance far outperforms B-trees.

In the GPU LSM, since we assumed all insertions are in units of b elements, the size of level i in the GPU LSM is $b2^i$, and at any time the whole data structure can accommodate multiples of b elements. Each level is completely full or completely empty. To insert a batch of size b, the batch is first sorted. If the first level is empty, the batch becomes the new first level. Otherwise, we merge the sorted batch with the sorted list already residing in the first level, forming a sorted array of size $2b$, and proceed. Figure 4.1 schematically shows the insertion process in the GPU LSM.

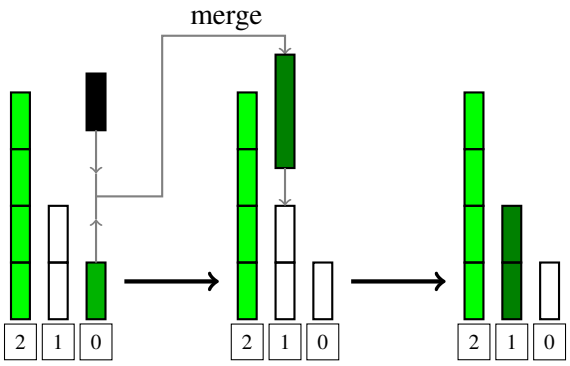


Figure 4.1: Insertion example in GPU LSM; adding a new batch of b elements into a GPU LSM with $5b$ elements. Blocks with similar colors are sorted among themselves.

Note that in a GPU LSM with $n = rb$ elements (*resident elements*), the levels that contain sorted lists correspond to the set bits in the the binary representation of the integer r. The insertion process corresponds to incrementing r, with binary additions and carries corresponding to the merging procedure described above. The larger our choice of b is, the more parallelism can be exploited in our insertion process by the GPU. Smaller b sizes lead to inefficiency in the first few levels for each operation, and indirectly increase the total number of levels (r) in the data structure. Also note that since we start insertion from lower indexed (smaller) levels, any item residing in larger levels is certainly inserted before any element in a smaller level. We also make sure that after merging, insertion order is preserved among all elements within the same level (Section 4.4.1).

LSMs and COLAs were designed to exploit I/O parallelism (with an analysis based on memory block size) [14]. Similarly, we can analyze the GPU LSM's insertions to show that any sequence of r insertions of batches of size b requires at most $O(rb \log r)$ work, that is, $O(\log r)$ work per item. While a worst-case individual insertion will require a cascade of merges, ultimately placing the final list in the level corresponding to the most significant bit of r, and hence time $\Omega(rb)$, such a worst-case operation can only occur infrequently. In particular, an element that gets stored in a list of length $O(2^i b)$ has participated in $O(i) \leq O(\log r)$ merge operations, so the total work performed over all rb elements in the GPU LSM at any time is $O(rb \log r)$.

4.3.3 Deletion

Deletion was not initially included in the COLA. To include deletion, we need to make sure item 5 in Section 4.3.1 is satisfied. The standard way to delete an item is to insert a *tombstone* item with the same key, indicating that previously inserted items with that key, if any, should be considered deleted. Since deletion, then, is the insertion of a tombstone, deletion and insertion are in practice the same, and we can combine any insertion and deletion requests into a mixed batch. As we shall see, this tombstoning scheme allows the GPU LSM to perform insertions and deletions very efficiently, at the cost of accumulating stale elements.

4.3.4 Lookup

Recall that LOOKUP(k) should return the most recently inserted value corresponding to k, if it was not subsequently deleted, or otherwise report that such a key does not exist. To ensure this, we guarantee the following *building invariants* during insertion and deletion.

1. Within each level, all elements are sorted by key and thus all elements with the same key are next to each other (segmented);

2. Every tombstone at each level is placed as the first element of its segment; and

3. All elements of each segment (regular elements and tombstones) are ordered from the lowest index to the highest based on their insertion time (from least recent to most recent).

With these invariants, it suffices to start our lookup process from the smallest full level (most recent) and look for the first index with key greater than or equal to k. If we find a key equal to k, we return it and are done, otherwise we continue to the next full level. If we find a tombstone with key k at any time during our search, then k is deleted and our lookup returns no results.

With $n = rb$ total elements, finding the lower bound (a modified binary search) in each level takes $O(\log(b2^i))$ steps over $\log r$ steps, which in the worst case results in $O(\log^2(r) + \log(r)\log(b))$ individual memory accesses per query (the same cost as in the basic LSM).

4.3.5 Count and Range queries

Both COUNT and RANGE operations take a tuple (k_1, k_2) as an input argument. The former returns the total number of elements within that range of keys while the latter returns all those

valid elements as an output. Based on the same building semantics described in Section 4.3.4, we implement the following procedure.

Since each full level is sorted by key, we can use binary search to find indices corresponding to the first key k such that $k \geq k_1$ or $k > k_2$ (also known as lower bound and upper bound operation, respectively).

If duplicates and deletions were not allowed in the data structure, we could compute a result by subtracting these indices to have the number of elements within those bounds (for COUNT), or just collecting all elements within those bounds (for RANGE). However, with duplicates and deletions allowed, this approach would include three types of stale entries in our results: 1) tombstones, 2) deleted elements (because of an existing tombstone in a higher level), and 3) elements with the same keys that were inserted in different times (and are hence in different levels). To address these stale entries and return an accurate answer, we perform a post-processing stage (discussed in Sections 4.4.3–4.4.4) to correct our preliminary potential results (Afshani et al. discuss why counts are harder than lookups [2]). It should be noted this extra validation could be avoided if we were not allowing stale elements in the data structure (item 3 in Section 4.3.1), but that is the price that we pay to support deletion.

4.3.6 Cleanup

The structure we have described so far produces correct query results even in the presence of tombstones and stale elements (either deleted or duplicate). This allows us to perform faster insertions and deletions, but as tombstones and stale elements accumulate, we will see increased memory usage and more occupied levels, resulting in reduced query performance.

Thus we provide a CLEANUP operation, in which all tombstones, their corresponding deleted elements, and all duplicate elements (except for the most recent one) are removed, followed by a reorganization of the GPU LSM. When is a cleanup appropriate? Cleanups are not free, but they may reduce the number of LSM levels and increase query performance when there are a significant number of deletes and/or duplicate inserts. Frequent cleanups also may be appropriate in applications where query performance is paramount.

If a cleanup is performed after $O(rb)$ operations, its cost is at most $O(rb)$: rebuilding the GPU LSM from scratch requires a radix sort, compaction to remove stale elements and tombstones,

and slicing up the remaining sorted list of valid elements into a sequence of levels. A schedule in which a cleanup is performed every time the GPU LSM doubles in size can be amortized so that the total work does not increase.

4.4 Implementation details

In this section we dive deeper into the implementation details of each operation and the design choices that we make. Our design decisions are partially influenced by hardware characteristics (NVIDIA GPUs), programming model (CUDA), and available open-source primitive libraries (moderngpu[5] and CUB [67]).

Our GPU LSM is built using 32-bit variables (both keys and values). We use the term "element" to refer to what is stored at a specific index in arrays (key arrays, value arrays, etc.) allocated in the GPU's global memory.

4.4.1 Insertion and Deletion

In Sections 4.3.2 and 4.3.3, we noted that the implementation of insertion and deletion in the GPU LSM were similar. In order to distinguish between a tombstone and a regular element to be inserted, we dedicate one bit as a flag; we refer to this bit as the *status bit*. The 32-bit *key variable* is the 31-bit *original key* shifted once and placed next to the status bit. The cost of this decision is that we lose one bit in the key domain.

As described in Section 4.3.2, when inserting a batch of b elements, we first sort the new batch by their keys. In order to fulfill the building invariants of Section 4.3.4 while also satisfying the semantics in Section 4.3.1, we implement sorting and merging as follows. For sorting, we use regular radix sort over all key variables including the status bit. For merging, we merge different levels just based on the original keys, excluding the status bit. We make sure that stability is preserved: new levels merged into existing levels appear first in the merged result, to preserve batch insertion order. We use moderngpu's merges with a modified comparison operator and CUB's radix sort for this process.

Since a tombstone is marked with a zero LSB, after sorting a batch prior to insertion, a tombstone will appear before any insertions with the same key. And, since our merge is stable and

[5]Moderngpu is available at `https://github.com/moderngpu/moderngpu`.

we neglect the status bits, a regular element from a smaller (more recent) level always appears before all later elements with the same key (regular or tombstone) that came from previously inserted batches. Note that after merging we do not remove stale elements. Nonetheless, because those stale elements appear after the replacement or tombstone element that made them stale, they are "invisible" to the queries. In case there are only $b' < b$ new elements to be inserted, a user can pad the batch by duplicating enough $(b - b')$ copies of an arbitrary element within the batch (e.g., the last one); only one of those duplicates will be visible to queries. In case our GPU LSM achieves an average R insertion rate, here we are effectively only using Rb'/b of our available insertion rate. As a result, users should choose b wisely to balance the cost of partial batches with the efficiency of larger batches.

In the original LSM data structures on the CPU, higher levels of the data structure are small enough to fit in cache, so insertions are effectively performed inside cache before getting merged with lower levels of the data structure. In our GPU LSM, based on our choice of having a sorted array (a cache-oblivious data structure) in each level of LSM, top levels will automatically cache efficiently, aided by the high associativity of the L2 cache on the GPU. Nevertheless, insertions/deletions are only done in batches of size $b \gg 1$, which means fewer levels can be fit into cache, but each level has more elements in it (making it practically effective as in CPU versions). As well, sort and merge primitives aggressively use shared memory to achieve coalesced global memory accesses so that final results (sorted/merged chunks of input) are first stored into shared memory and then stored into global memory in larger batches.

4.4.2 Lookup queries

We could implement lookup queries in two ways: a bulk approach and an individual approach. For the bulk approach, we would first sort all queries and then perform a sorted search (similar to moderngpu's) over all queries and all occupied levels. However, sorting all queries is an expensive operation and having all queries ready synchronously may also be a strong assumption. Thus we focus on an individual approach where each query can be performed independently.

Each thread can individually perform a lookup query by simply performing a lower-bound binary search[6] in each occupied level, starting from the smallest level. There are three possible

[6] A lower-bound binary search is similar to STL's `std::lower_bound` operation, which finds the first instance

outcomes: 1) we find a regular element (set LSB) that matches our query and then return its value, 2) we find a tombstone (zero LSB) that matches our query and then return $\perp$, or 3) we search in all levels and nothing is found (return $\perp$). The main bottleneck for our lookups is the random memory accesses required in all binary searches that we perform.

4.4.3 Count queries

We first consider our bulk implementation of COUNT. We assume there are multiple queries to be performed (each with its (k_1, k_2) argument) and each query can have any arbitrary size as its outcome. We assign each GPU thread to a query and then perform device-wide operations to find all results together (with possible collaboration among threads). The procedure takes five steps:

1. Initial count estimate: we perform two separate binary searches for the lower and upper limits of each query for each level. By subtracting resulting indices at each level, we get an upper bound on the number of potentially valid elements. For each query, we write the per-level upper bounds next to each other in memory.

2. Scanning stage: we perform a device-wide scan operation over the upper bounds to compute a global index for each query result.

3. Initial key storage: we use the lower limits of the queries and the global indices computed in the previous step to copy all potential keys into an array.

4. Segmented sort: we perform a segmented sort over the array (each segment belongs to a query). LSBs (status bits) are neglected in sorting comparisons.

5. Final counting: all identical keys (tombstones or regular) are now next to each other and sorted by time step. We count the first element of each segment only if it is not a tombstone. As a result, if there are multiple keys, only one of them is counted.

We use CUB's exclusive scan (prefix-sum) for stage 2. For stage 3, we assign each thread to a query but force the threads in a warp to collaborate with each other in writing out results from all 32 queries involved (for coalesced accesses). We use moderngpu's segmented sort for stage 4.

of a key greater than or equal to its argument in a sorted list.

In stage 5, we assign each query to a thread but force the threads in a warp to collaborate with each other in validating and counting (via warp-wide ballots) the results for all potential matches from 32 consecutive queries.

An individual approach (a group of threads performing the query on their own) will not be competitive if there is no bound on the arguments ($k_2 - k_1$), because it may require too many serialized memory accesses for the validation process. However, if we are given an a priori bound on the ranges, we can efficiently use all shared resources and perform a single kernel count query, both searching and then locally validating the results.

4.4.4 Range queries

We implement RANGE operations similarly to COUNT, differing in the final three steps. In stage 3, we store not only all potential keys from each query but also their corresponding values. Stage 4 becomes a segmented sort over key-value pairs rather than just keys. Finally, in stage 5, we first mark each valid element (by overwriting the LSBs) and then perform a segmented compaction based on all set LSBs (each segment represents a query) to gather all non-stale non-tombstone elements. The final result is the beginning memory offsets of each query, followed by valid elements (both keys and values) belonging to each query, sorted by their keys.

4.4.5 Cleanup

As we described in Section 4.3.6, CLEANUP removes all stale elements (tombstones, deleted and duplicates) and rebuilds the data structure. In its implementation, we require 1) cross-validation among different levels to remove deleted and duplicate elements from different levels (caused by elements from more recent levels) and 2) that the total number of elements should remain a multiple of b after removals. To satisfy the first item, we make sure all our intermediate operations within the cleanup are "stable", meaning that among all instances of an arbitrary key (tombstone or regular), temporal information is preserved. The second item is important since we assume that the number of elements in our data structure is a multiple of b, and this may no longer be true after removing stale elements. We could either remove enough tombstones and duplicates to make sure the condition is satisfied and leave the rest unchanged (possibly to be processed in future cleanups), or we can simply remove all stale elements and then pad

with enough ($< b$) *placebo* elements.[7] We choose the second approach because it requires less processing and appears to be more efficient.

Our bulk strategy for cleanup is to 1) iteratively merge all occupied levels from the smallest to the largest (neglecting the LSB to preserve time ordering), 2) mark all unmarked stale elements, 3) compact all valid elements together, 4) add enough placebos, and 5) redistribute (already sorted) elements to different (new) levels. Since all levels are already sorted, merging them together iteratively is much faster than resorting all of them together. Our cleanup implementation preserves timing order among elements with the same original key, but not across different keys (smaller keys will end up in smaller levels).

4.5 Performance Evaluation

4.5.1 What does the GPU LSM provide?

We compare our results with two data structures (Table 4.1): a GPU hash table (cuckoo hashing [3]), and a sorted array (SA).

Cuckoo hashing Cuckoo hashing has bulk build and lookup operations, but it does not support deletions and in general it is not possible to insert new elements or increase table sizes at runtime (i.e., not mutable).[8]

Sorted array Another simple but effective GPU data structure can be a single sorted array (SA) on GPU memory. In the SA, insertions (or deletions) can happen by adding (or removing) elements and resorting the whole array, which, as we shall see, is much slower than applying updates to a GPU LSM. Merging a sorted set of elements into an existing SA is faster. All queries in an SA are similar to the GPU LSM, but only on a single occupied level (of arbitrary size). In general, all SA queries (lookup, count, and range) have a faster worst-case scenario than the GPU LSM's, because it is easier to search a single sorted level ($O(\log n)$) than to search through multiple smaller occupied levels in the GPU LSM ($O(\log^2 n)$); in practice, though, we find that the difference is small.

[7] We pad our data structure with tombstone status bits and maximum keys (e.g., $2^{32} - 1$ in 32-bit scenarios) at the end of the last level's sorted array. These elements will be invisible to queries since they are tombstones, and will always remain at the end of the last level since no larger key is possible.

[8] Cuckoo hashing code is from CUDPP: `https://github.com/cudpp/cudpp`.

	Cuckoo hashing	Sorted Array (SA)	GPU LSM
INSERT	—	$O(n)$	$O(\log n)$
DELETE	—	$O(n)$	$O(\log n)$
LOOKUP	$O(1)$	$O(\log n)$	$O(\log^2 n)$
COUNT/RANGE	$O(n)$	$O(\log n + L)$	$O(\log^2 n + L)$

Table 4.1: High-level comparison of capabilities in a GPU hash table, a sorted array and a GPU LSM. Bounds on the total work are normalized to be per item in a data structure with n elements. L denotes the size of the output.

Other data structures Our principal design goal in this project is to design and build an efficient mutable data structure that supports insertions and deletions. We targeted mutable dictionaries with lookup and range queries, which led us to our GPU LSM. While we aimed to maximize query performance, we did so under the constraint of mutability. Assessing the quality of our achieved performance is indeed a challenging task, since there are not yet any other general-purpose mutable data structures for GPUs.

Numerous GPU data structures are specifically designed to only satisfy high-performance queries, without any support for mutability. Many of these data structures are sequentially built on the CPU and then transformed into the GPU, just because efficient parallel building of these data structures is challenging enough to be avoided (Section 4.2). For example, if we are interested in having only fast queries and no other operations, B-trees will be more suitable for coalesced memory accesses in GPUs (compared to a binary search that requires random memory accesses).[9] However, the same B-tree data structures appear to be suboptimal for insertions and deletions, to the extent that, to the best of our knowledge to date, there is not any publicly available B-tree library for comparison on GPUs (even for just building the data structure and performing queries).[10] Given the state of the art in this area, constructing even a non-mutable

[9]Kaldewey et al. compared efficient uses of P-ary searches on ideal B-trees vs. regular binary search on the GPU, with a focus on the SIMD structure of GPUs in lookup operations [48]. They concluded that, on a NVIDIA GTX285 GPU and with 128 million keys, P-ary searches on a CPU-built B-tree are ideally up to an order of magnitude faster than regular binary search on a sorted array, which itself is an order of magnitude better than P-ary searches on a sorted array. Given the suboptimal insertions in B-trees, we believe that having a sorted array in each level of our LSM and performing queries based on regular binary search is our best option to support both mutability and fast queries.

[10]As an example, Fix et al. experimented on lookup queries on a CPU-built B+ tree [34]. Their data structure is

B-tree GPU implementation for comparison is a significant research project in itself.

Sorted arrays and hash tables are the only fully developed GPU data structures for general-purpose tasks with well-known and optimized performance characteristics (in contrast to other data structures like B-trees that are not yet fully developed on GPUs). As a result, although neither of them is a mutable data structure, we believe comparing our GPU LSM against them is the right way to benchmark our data structure. This comparison is not meant to disprove the mutability capability of our GPU LSM but instead to expose the price we paid for achieving it (in terms of query performance).

In the following we discuss insertions and deletions in Section 4.5.2, then lookups, count, and range queries in Section 4.5.3. Finally, we discuss our cleanup operation and its potential influence on faster queries in Section 4.5.4. All experiments are run on an NVIDIA Tesla K40c GPU (ECC enabled), with Kepler architecture and 12 GB DRAM, and an Intel Xeon CPU E5630. All programs are compiled with NVIDIA's nvcc compiler (version 7.5.17) with the -O3 optimization flag.

4.5.2 Insertions and deletions

For any batch, including an arbitrary set of insertions and deletions, the GPU LSM provides the same performance (i.e., performance does not depend on status bits). As a result, here we just consider pure insertion. Our principal target is an application scenario where we repeatedly insert batches of elements into our data structure. On this scenario—which we quantify as "effective insertion rate"—the GPU LSM has considerably better theoretical and experimental behavior than a hash table or a sorted array, as we will see at the end of this subsection. However, before we explore this scenario, we must first consider two preliminary scenarios: 1) building the data structure from scratch ("bulk build") and 2) inserting one single batch into an already-built data structure ("batch insertion").

first built on the CPU and put into a contiguous block of memory so that it can be transferred easily to the GPU. They assigned all threads within a thread-block to perform each single lookup query on their data structure. On an NVIDIA GTX 480 and with 100k elements in their B+ tree, without considering memory transfer time between the CPU and the GPU, they reported 20 μs for each query on the GPU and 2 μs for performing the same query on their CPU (Intel Core 2 Quad Q9400). We do not believe their implementation represents a competitive data structure from the B-tree family for comparison with our LSM. As we will see in Section 4.5.3.1, with 16 M elements, our GPU LSM can achieve an average lookup rate up to 75.7 M queries/s over 16 M queries (0.013 μs per query).

	GPU LSM			Sorted Array (SA)		
b	min rate	max rate	mean rate	min rate	max rate	mean rate
2^{27}	727.8	727.8	727.8	727.8	727.8	727.8
2^{26}	585.3	727.7	648.8	583.1	727.7	647.4
2^{25}	421.1	727.1	585.5	472.2	726.8	561.2
2^{24}	270.3	727.1	537.9	346.0	727.0	459.2
2^{23}	155.5	726.3	485.2	224.0	723.2	334.0
2^{22}	84.3	714.8	441.2	132.6	709.9	219.2
2^{21}	44.0	694.0	398.1	74.2	691.2	131.2
2^{20}	22.4	664.2	354.1	39.8	658.7	73.8
2^{19}	11.3	558.2	289.4	20.5	555.9	39.4
2^{18}	5.6	432.0	220.7	10.5	422.1	20.3
2^{17}	2.8	326.8	159.9	2.6	319.1	10.3
2^{16}	1.4	194.3	98.0	1.3	184.8	5.2
2^{15}	0.7	101.4	58.4	0.6	95.9	2.6
mean			225.3			16.7
Cuckoo Hash			361.7			

Table 4.2: Minimum/maximum/harmonic mean insertion rates (M elements/s) for GPU LSM (and SA) and with various batch sizes. These numbers are recorded over all possible number of resident batches for $1 \leq r \leq 2^{27}/b$. We also note the bulk build rate of cuckoo hashing with an 80% load factor.

Bulk build Suppose there are initially a set of kb available elements from which we want to build a GPU LSM. This operation requires a sort and is similar to building a sorted array (SA); the bulk build of either is faster than cuckoo hashing (up to 2x). Our GPU sustains 770 M elements/s for key-value radix sort. Then we must segment this array into at most $\log k$ sorted levels corresponding to its GPU LSM levels (in-memory transfers with 288 GB/s = 36 G elements/s); this time is negligible compared to sorting.[11] Cuckoo hashing with an 80% load factor has a build rate of 361.7 M elements/s, roughly 2x slower than both the GPU LSM and the SA.

Batch insertion To insert a new batch into an existing GPU LSM, we first sort the batch then iteratively merge with the lowest-indexed full levels until we reach the first empty level. As a result, if there are r resident batches in the GPU LSM prior to the new insertion, the first empty level's index will be the least significant zero bit in r (we call it $\mathit{ffz}(r)$). The total amount of time required for inserting a new batch to a GPU LSM (total of rb elements), also depicted in Fig. 4.2a, is $T^b_{\text{ins}}(r) = T^b_{\text{sort}} + (2^{\mathit{ffz}(r)} - 1)T^b_{\text{merge}}$, where T^b_{sort} is the time spent to sort a batch of size b and T^b_{merge} is the time spent to merge two batches of size b. The second term can be derived because merge is a linear operation, and hence merging two arrays each of size $2^k b$ takes about $2^{k+1}T_{\text{merge}}$.

Because of this complex behavior and in order to better compare the GPU LSM's performance with SA and hash tables, we conduct the following experiment with a fixed batch size b: We randomly generate $n = 2^{27}$ elements and incrementally insert batches of b elements ($\lceil n/b \rceil$ batches) into the GPU LSM. After each insertion, we compute the insertion rate for that batch (b divided by the insertion time). By doing so, we have considered all possible resident batches of $1 \leq r \leq \lceil n/b \rceil$. We continue this approach for different batch sizes.

Table 4.2 shows the minimum rate (when all levels are full), the maximum rate (when the first level is empty), and the harmonic mean of all possible other outcomes. We have also repeated the same experiment for the SA. The minimum rate for GPU LSM is usually worse than the SA's because of iterative merges. However, the mean rate is always better than the SA, because in

[11] If we have multiple batches to insert into a non-empty GPU LSM, all batches can be sorted together and then iteratively merged from the highest valued batch until the lowest valued batch (to make sure deletions and time information are correctly preserved).

practice, on average, very few merges are performed before finding an empty level. For a fixed n, if a smaller b is chosen for the GPU LSM, there will be more full levels and hence more iterative merges and as a result slower performance. Averaged (harmonic mean) over all batch sizes, the GPU LSM's insertion rate is 225.3 M elements/s, which is 13.5x faster than SA.

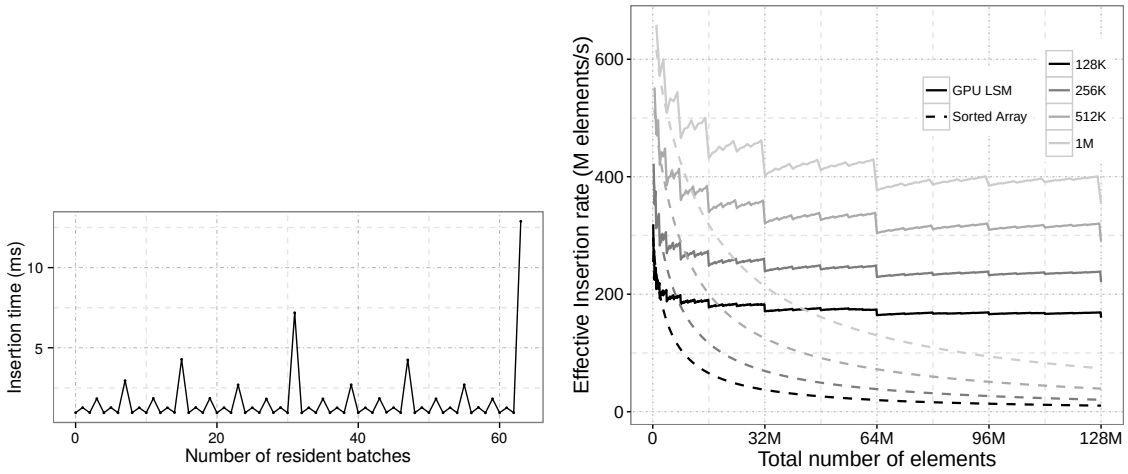
Effective insertion rate How does the GPU LSM compare with the SA with repeated inserts? The effective insertion rate for the GPU LSM gets increasingly better than the SA as more and more batches are inserted. The reason is that for the SA to handle insertions, each time the new batch must be sorted and then merged with all other elements ($O(1/n)$ rate). However, it is possible (by summing and bounding $T^b_{\text{ins}}(r)$ over a series of batch insertions) to show that for the GPU LSM the effective insertion rate is instead $O(1/\log n)$. To show this experimentally, we start with an empty GPU LSM and incrementally insert batches. After each insertion we compute the effective rate (number of resident elements divided by total time). Figure 4.2b shows the results for various batch sizes. Repeating the same experiment for SA, it is clear that its performance degrades at a higher rate as the number of elements increases. In short, while merging into an SA is fast and the SA could potentially be used for dynamic inserts, it is clear both theoretically and in practice that the GPU LSM is the superior data structure for insertions, and, with the tombstoning scheme, for deletions as well.

4.5.3 Queries: Lookup, count and range operations

The performance of queries in the GPU LSM depends on the specific arrangement of full levels in the current data structure. Searching more levels requires more work. Also the larger the levels to be searched, the more work. We observe in this section that query performance depends mainly on the average number of full levels, and hence for a fixed number of elements, a GPU LSM with larger batch size usually has superior query performance.

For lookups, we start from the first full level and continue until we find the query key, so the worst case happens when the query does not exist in the data structure. For count and range queries all levels must be searched regardless. The larger the range in a count or range query (e.g., $k_2 - k_1$), the more elements are expected to fall within it and hence more work is required for their final validation, which results in reduced performance.

Because of the complex relation of query performance with level distributions, we conduct



(a) Batch insertion time (ms) in GPU LSM with $b = 2^{19}$.

(b) Effective insertion rate versus total number of inserted elements.

Figure 4.2: (a) Batch insertion time in GPU LSM. (b) Effective insertion rate for a series of batch insertions in GPU LSM, and a sorted array (SA). Each color represents a different batch size.

the following experiment: For any fixed batch size b and total of $n = 2^{24}$ elements, we build every possible GPU LSM with $1 \leq r \leq n/b$ resident batches. In each case, we generate the same number of queries as there are elements in the data structure. We report the minimum rate, the maximum rate, and the harmonic mean of queries. The same experiment is repeated on a sorted array (SA) of the same size.

4.5.3.1 Lookup queries

Table 4.3 shows the result of this experiment in two scenarios: 1) all queries exist (right) or 2) none exist (left). In this experiment we see that the mean query rate of the GPU LSM decreases as the batch size decreases, because on average the total number of occupied levels increases. Averaged (harmonic mean) over all batch sizes in Table 4.3, our GPU LSM gets 67.9 (or 75.7) M queries/s for none (or all) scenarios. Similarly, the SA reaches 119.0 (or 133.3) M queries/s, which are both almost 1.75x faster than their respective GPU LSM rate. The cuckoo hash table is also 7.34x (or 10.01x) faster than the GPU LSM. Recall, however, that both the SA and hash tables are immutable data structures.

	none existing				all existing			
b	min	max	mean	SA	min	max	mean	SA
2^{24}	116.8	116.8	116.8	106.8	106.8	106.8	106.8	116.7
2^{23}	116.8	133.1	124.4	112.4	106.8	118.6	112.4	124.3
2^{22}	61.6	183.1	105.9	113.9	74.7	142.2	104.7	128.9
2^{21}	41.4	237.4	84.0	118.4	51.7	194.7	91.1	132.9
2^{20}	30.7	291.3	68.4	120.3	39.3	251.1	77.6	135.0
2^{19}	25.2	331.7	57.7	123.4	32.0	299.2	67.5	138.7
2^{18}	22.0	361.5	51.1	125.4	27.2	332.0	60.7	141.2
2^{17}	19.5	376.7	47.4	126.8	24.3	347.3	56.7	143.0
2^{16}	18.1	386.1	45.5	127.6	22.7	365.7	54.9	144.1
Cuckoo hash			498.9				758.0	

Table 4.3: Lookup rate (M queries/s): searching for the same number of keys as there are elements in the data structure such that no (left) or all (right) queries exist. For GPU LSM, we have gathered results from all possible number of batches for $1 \leq r \leq 2^{24}/b$. For SA we have only reported the harmonic mean. Cuckoo hashing is with 80% load factor.

4.5.3.2 Count and Range queries

Unlike lookup queries, count and range queries will always search through all full levels of a GPU LSM. However, there is a new important factor in their performance: the *expected range* (L) of each argument. The larger the L, the more keys fall within its range, and hence more potential results must be validated. Table 4.4 shows the same experiment we describe at the beginning of this section, followed by the same number of queries with two different L values. Averaged over all batch sizes, count queries for $L = \{8, 1024\}$ reach {32.7, 2.8} M queries/s, which is {1.84x, 1.45x} slower than performing the same queries on a SA with the same size. Similarly for range queries and averaged over all batch sizes, we reach {23.3, 1.3} M queries/s, which is {1.39x, 1.36x} slower than SA. Noted that it is not possible to efficiently use hash tables for any of these queries at all. Count queries are also always faster than range queries because range queries have to perform a more expensive validation stage (Section 4.4.4), and in the end they have to return all valid elements as opposed to just sending back a total count.

4.5.4 Cleanup and its relation with queries

Our observations show that the cleanup operation—discussed in Section 4.3.6 and detailed in Section 4.4.5—is much more efficient than rebuilding the whole data structure from scratch.

		$L = 8$				$L = 1024$			
	b	min	max	mean	SA	min	max	mean	SA
Count Query	2^{20}	27.0	103.7	48.0	73.5	2.58	4.17	3.01	4.10
	2^{19}	22.4	94.1	39.8	68.6	2.51	4.13	2.86	4.09
	2^{18}	19.0	82.9	35.1	63.0	2.44	4.10	2.79	4.08
	2^{17}	16.2	67.6	27.4	56.1	2.36	4.06	2.63	4.04
	2^{16}	15.1	44.7	23.9	47.3	2.32	3.99	2.57	3.99
Range Query	2^{20}	23.2	72.1	38.4	42.2	1.27	1.80	1.43	1.81
	2^{19}	19.4	63.2	31.9	38.2	1.25	1.81	1.37	1.81
	2^{18}	14.6	50.5	24.9	33.8	1.23	1.81	1.35	1.81
	2^{17}	12.9	36.3	19.8	29.1	1.20	1.81	1.28	1.81
	2^{16}	11.0	23.0	15.1	25.0	1.18	1.76	1.27	1.80

Table 4.4: Count and Range queries in GPU LSM and SA with expected range $L = 8, 1024$. We have gathered results from all possible number of batches for $1 \leq r \leq 2^{24}/b$.

Also, it can greatly improve query performance by reducing the number of full levels and hence may be useful to perform regularly.

Our experiments show that the speed of the cleanup operation mostly depends on the total number of resident elements in the data structure, and less significantly on the percentage of elements that need to be removed (stale elements). The more elements to be removed the better. For example, with $n = (2^6 - 1)b$ elements where $b = 2^{20}$, cleanup operations when $\{10, 50\}\%$ of elements should be removed runs at {1870.2, 1828.2} M elements/s. A GPU LSM with roughly the same size ($n = (2^7 - 1)b$ with $b = 2^{19}$) with $\{10, 50\}\%$ of elements removed results in {1842.5, 1794.3} M elements/s. Since the GPU LSM's bulk build sustains 728 M elements/s (Section 4.5.2), cleanup is up to 2.5x faster than building all elements from scratch.

Depending on the number of resident batches, cleanup can speed up queries by potentially reducing the total number of full levels. For example, with 10% removals, $n = (2^7 - 1)$, and $b = 2^{18}$, cleanup takes 19.23 ms to finish. After cleanup, we can perform 32 million lookup queries in 132.5 ms, which is almost 4.8x faster than performing the exact same queries before the cleanup (including the cleanup time). This is an important result, since it motivates the user to regularly perform cleanups if she needs to perform a lot of queries during the lifetime of a GPU LSM.

4.6 Conclusion

We proposed and implemented the GPU LSM, a dynamic dictionary data structure suitable for GPUs, with fast batch update operations (insertions and deletions) as well as a variety of parallel queries (lookup, count, and range). We find that we can update the GPU LSM much more efficiently than we can a sorted array, especially for small batch sizes, at the cost of a small constant factor in query time. It might be possible to improve the query time by employing the fractional cascading idea used in COLA [14], at the cost of a more complicated insertion and more memory; it is not clear whether this would be practical or not.

Similar ideas might be useful in other GPU data structures, for example BVH trees, which might be useful for applications such as collision detection and ray tracing in dynamic scenes.

Chapter 5

Slab Hash: A Dynamic Hash Table for the GPU[1]

5.1 Introduction

A key deficiency of the GPU ecosystem is its lack of *dynamic* data structures, which allow incremental updates (such as insertions and deletions). Instead, GPU data structures (e.g., cuckoo hash tables [3]) typically address incremental changes to a data structure by rebuilding the entire data structure from scratch. A few GPU data structures (e.g., the dynamic graph data structure in cuSTINGER [37]) implement phased updates, where updates occur in a different execution phase than lookups. In this work we describe the design and implementation of a hash table for GPUs that supports truly concurrent insertions and deletions that can execute together with lookups.

Supporting high-performance concurrent updates of data structures on GPUs represents a significant design challenge. Modern GPUs support tens of thousands of simultaneous resident threads, so traditional lock-based methods that enforce concurrency will suffer from substantial contention and will thus likely be inefficient. Non-blocking approaches offer more potential for such massively parallel frameworks, but most of the multi-core system literature (e.g., classic non-blocking linked lists [71]) neglects the sensitivity of GPUs to memory access patterns and branch divergence, which makes it inefficient to directly translate those ideas to the GPU.

[1]This chapter substantially appeared as "A Dynamic Hash Table for the GPU" [8], for which I was the first author and responsible for most of the research and writing.

In this chapter, we present a new GPU hash table, the *slab hash*, that supports bulk and incremental builds. One might expect that supporting incremental insertions and deletions would result in significantly reduced query performance compared to static data structures. However, our hash table not only supports updates with high performance but also sustains build and query performance on par with static GPU hash tables. Our hash table is based on a novel linked list data structure, the *slab list*. Previous GPU implementations of linked lists [72], which operate on a thread granularity and contain a data element and pointer per linked list node, exhibit poor performance because they suffer from control and memory divergence and incur significant space overhead. The slab list instead operates on a warp granularity, with a width equal to the SIMD width of the underlying machine and contains many data elements per linked list node. Its design minimizes control and memory divergence and uses space efficiently. We then construct the slab hash from this slab list as its building block, with one slab list per hash bucket. Our contributions in this work are as follows:

- The slab list is based on a node structure that closely matches the GPU's hardware characteristics.
- The slab list implementation leverages a novel warp-cooperative work sharing strategy that minimizes branch divergence, using warp-synchronous programming and warp-wide communications.
- The slab hash, based on the slab list, supports concurrent operations with high performance.
- To allow concurrent updates, we design and implement a novel memory allocator that dynamically and efficiently allocates and deallocates memory in a way that is well-matched to our underlying warp-cooperative implementation.
- Our memory allocator is scalable, allowing us to support data structures up to 1 TB (far larger than the memory size of current GPUs) and without any CPU intervention.
- Slab hashes bulk-build and search rates are comparable to those of static methods (e.g., cuckoo hash table [3]), while also achieving efficient incremental updates.

5.2 Background & Related Work

Hash tables There are several efficient static hash tables implemented for GPUs. Alcantara et al. [3] proposed an open-addressing cuckoo hashing scheme for GPUs. This method supports bulk build and search, both of which require minimal memory accesses in the best case: a single atomic operation for inserting a new element, and a regular memory read for a search. As the load factor increases, it is increasingly likely that a bulk build using cuckoo hashing fails. Garcia et al. [36] proposed a method based on Robin Hood hashing that focuses on higher load factors and uses more spatial locality for graphics applications, at the expense of performance degradation compared to cuckoo hashing. Khorasani et al.'s stadium hashing is also based on a cuckoo hashing scheme but stores two tables instead of one. Its focus is mainly on out-of-core hash tables that cannot be fit on a single GPU's memory. In the best case (i.e., an empty table) and with a randomly generated key, an insertion in this method requires one atomic operation and a regular memory write. A search operation in stadium hashing requires at least two memory reads. Although hash tables may be specifically designed for special applications, Alcantara's cuckoo hashing appears to be the best general-purpose in-core hash table option with the best performance measures. We use this method for our comparisons in Section 5.6.

Misra and Chaudhuri [72] implemented a lock-free linked list, which led to a lock-free hash table with chaining that supported concurrent insertion, deletion and search. Their implementation is not fully dynamic, because it pre-allocates all future insertions into an array (which must be known at compile time), and it does not address the challenge of dynamically allocating new elements and deallocating deleted elements at runtime. However, we briefly compare it to the slab hash in Section 5.6.3. Inspired by Misra and Chaudhuri's work, Moscovici et al. [74] recently proposed a lock-based GPU-friendly skip list (GFSL) with an emphasis on the GPU's preferred coalesced memory accesses. We will also discuss in Section 5.6.3 why we believe GFSL (either by itself or as a building block of a larger data structure) cannot beat the peak performance of our lock-free slab hash in updates and searches.

Dynamic memory allocation Although a mature technology for single and multi-core systems, dynamic memory allocation is still considered a challenging research problem on massively parallel frameworks such as GPUs. Massive parallelism makes it difficult to directly exploit

traditional allocation strategies such as lock-based or private-heap schemes without a significant performance degradation.

CUDA [78] provides a built-in `malloc` that dynamically allocates memory on the device (GPU). However, it is not efficient for small allocations (less than 1 kB). To address `malloc`'s inefficiencies for small allocations, almost every competitive proposed method so far is based on the idea of allocating numerous large enough memory pools (with different terminology), assigning each memory pool to a thread, a warp, or a thread block (to decrease parallel contention), dynamically allocating or deallocating small portions of it based on received requests, and finally implementing a mechanism to use another memory pool once fully allocated. Some methods use hashing to operate on different memory pools (e.g., `ScatterAlloc` [90] and `Halloc` [1]). Other methods use various forms of linked lists to move into different memory pools (e.g., `XMalloc` [43] and `CMalloc` [92]). All these methods maintain various flags (or bitmaps) and operate on them atomically to be able to allocate or deallocate memory.

Vinkler et al. has provided an extensive study of all these methods and some benchmarks to compare their performance [92]. The most efficient ones, `CMalloc` and `Halloc`, perform best when there are multiple allocation requests within each warp that can be formed into a single but larger allocation per warp (a *coalesced* allocation). However, for the warp-cooperative work sharing strategy we use in this work (Section 5.4.1), we need an allocator that can handle numerous independent but asynchronous allocations per warp, which cannot be formed into a single larger coalesced allocation to avoid divergence overheads. As we will see in Section 5.5, existing allocators perform poorly in such scenarios. Instead, we propose a novel warp-synchronous allocator, *SlabAlloc*, that uses the entire warp to efficiently allocate fixed-size slabs with modest register usage and minimal branch divergence (more details in Section 5.5).

5.3 Design description

A linked list is a linear data structure whose elements are stored in non-contiguous parts of the memory. These arbitrary memory accesses are handled by storing the memory address (i.e., a pointer) of the next element of the list alongside the data stored at each node. The simplicity of linked lists makes concurrent updates relatively easy to support, using compare-and-swap

(CAS) operations [71]. New nodes can be inserted by (1) allocating a new node, (2) initializing the data it contains and storing the successor's address into its *next* pointer, then (3) atomically compare-and-swapping the new node's address with its predecessor's next pointer. Similarly, nodes can be deleted by (1) atomically marking a node as deleted (to make sure no new node is inserted beyond it) and then (2) compare-and-swapping its predecessor's pointer with its successor's address.

On GPUs, it is possible to implement the same set of operations for a linked list, and then use it as a building block of other data structures (e.g., in hash tables) [72]. However, this implementation requires an arbitrary random memory access per unit of stored data, which is not ideal for any high-performance GPU program. Furthermore, making any change to a linked list data structure requires a dynamic memory allocator, which itself is challenging to perform efficiently, especially on massively parallel devices such as GPUs (Section 5.2). In this work, we propose a new linked list data structure, the slab list, and then use it to implement a dynamic hash table (slab hash). In our design, we have two major goals in mind: (1) maximizing performance in maintaining several slab lists concurrently (suited for hash tables), and (2) having better memory utilization by reducing the memory overhead in classic linked lists. We present the slab list and slab hash in this section, and then provide implementation details in Section 5.4.

5.3.1 Slab list

Classic singly linked lists consist of nodes with two main distinctive parts: a single unit of data element (a key, a key-value pair, or any other arbitrary metadata associated with a key), and a *next* pointer to its successor node. Storing the successor's memory address makes linked lists flexible and powerful in dealing with mutability issues. However, it introduces additional memory overhead per stored unit of data. Moreover, the efficiency of linked list operations is one of our primary concerns.

In classic linked list usage, an operation (insertion/deletion/search) is often requested from a single independent thread. These high-level operations translate into lower-level operations on the linked list itself. In turn, these lower-level operations result in a series of random, sequential memory operations per thread. Because we expect that a parallel program that accesses such a data structure will feature numerous simultaneous operations on the data structure, we can

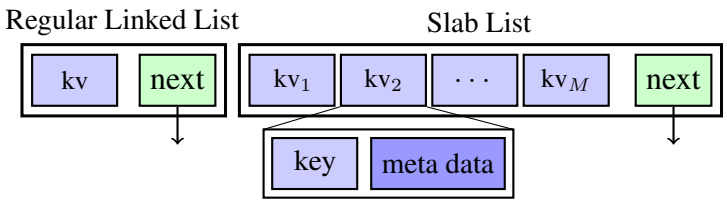


Figure 5.1: Regular linked list and the slab list.

easily parallelize operations across threads. But in modern parallel hardware with SIMD cores, including GPUs, the peak memory bandwidth is only achieved when threads within each SIMD unit (i.e., a warp in GPUs) access consecutive memory indices with a certain fixed alignment (e.g., on NVIDIA GPUs, each thread fetches a 32-bit word per memory access, i.e., 128 bytes per warp).

There are some well-known tactics to avoid coalesced-memory issues in GPUs, such as using structure-of-arrays instead of array-of-structures data layouts, or first fetching a big block of items into a faster but locally shared memory (with coalesced memory accesses) and then accessing the local memory with an arbitrary alignment. However, none of these methods is effective with a linked list data structure that requires singleton structures distributed randomly in the memory domain. As a result, we propose to use an alternate linked list design that is more suitable for our hardware platform.

In our design, we use a larger linked list node (called a *slab*, or interchangeably a *memory unit*) that consists of multiple data elements and a single pointer to its successor slab (shown in Fig. 5.1). An immediate advantage of slab lists is that their memory overhead is reduced by approximately a factor of M (if there are M data elements per slab). However, our main motivation for using large slabs is to be able to maintain them in parallel, meaning that the whole slab is accessed with a minimum number of memory accesses and in parallel (distributed among multiple threads), and then operations are also performed in parallel. The optimal size of these slabs will depend on the hardware characteristics of the target platform, including both the way memory accesses are handled as well as communication possibilities among different threads. On GPUs, we operate on each slab with a single SIMD unit (a warp) and use available warp-wide intrinsics such as shuffles and ballots for communications. So, the size of a slab will be a modest multiple of the warp width (e.g., 32 consecutive 32-bit words).

5.3.2 Supported Operations in Slab Lists

Suppose our slab list maintains a set of keys (or key-value pairs), here represented by $\mathcal{S}$. Depending on whether or not we allow duplicate keys in our data structure, we support the following operations:

- INSERT(k, v): $\mathcal{S} \leftarrow \mathcal{S} \cup \{\langle k, v\rangle\}$. Insert a new key-value pair into the slab list.
- REPLACE(k, v): $\mathcal{S} \leftarrow (\mathcal{S} - \{\langle k, *\rangle\}) \cup \{\langle k, v\rangle\}$. Insert a new key-value pair with an extra restriction on maintaining uniqueness among keys (i.e., replace a previously inserted key if it exists).
- DELETE(k): $\mathcal{S} \leftarrow \mathcal{S} - \{\langle k, v\rangle \in \mathcal{S}\}$. Remove the least recently inserted key-value pair $\langle k, v\rangle$.
- DELETEALL(k): $\mathcal{S} \leftarrow \mathcal{S} - \{\langle k, *\rangle\}$. Delete all instances of a key in the slab list.
- SEARCH(k): Return the least recent $\langle k, v\rangle \in \mathcal{S}$, or $\perp$ if not found.
- SEARCHALL(k) : Return all found instances of k in the data structure ($\{\langle k, *\rangle \in \mathcal{S}\}$), or $\perp$ if not found.

5.3.2.1 Search (SEARCH and SEARCHALL)

Searching for a specific key in slab list is similar to classic linked lists. We start from the head of the list and look for the key within that memory unit. If none of the data units possess such a key, we load the next memory unit based on the stored successor pointer. In SEARCH we return the first found matching element, but in SEARCHALL we continue searching the whole list. In both cases, if no matching key is found we return $\perp$.

5.3.2.2 Insertion (INSERT and REPLACE)

For the INSERT operation, we make no extra effort to ensure uniqueness among the keys, which makes the operation a bit easier. We simply start from the head of the list and use an atomic CAS to insert the new key-value pair into the first empty data unit we find. If the CAS operation is successful, then insertion is done. Otherwise, it means that some other thread has successfully inserted a pair into that empty data unit, and we have to try again and look for a new empty spot.

If the current memory unit is completely full (no empty spot), we load the next memory unit and repeat the procedure. If we reach the end of the list, it means that the linked list requires more memory units to contain the new element. As a result, we dynamically allocate a new memory unit and use another atomic CAS to replace the null pointer currently existing in the tail's successor address with the address of the newly allocated memory unit. If it is successful, we restart our insertion procedure from the tail again. If it failed, it means some other thread has successfully added a new memory unit to the list. Hence, we release the newly allocated memory unit and restart our insertion process from the tail.

REPLACE is similar to INSERT except that we have to search the entire list to see if there exists a previously inserted key k. If so, then we use atomic CAS to replace it with the new pair. If not, we simply perform INSERT starting from the tail of the list.

5.3.2.3 Deletion (DELETE and DELETEALL)

All our deletion operations are performed in a lazy fashion, meaning that we mark elements as deleted (put a *tombstone*) and do not immediately remove data elements from the list. We later describe our FLUSH operation that locks the list and removes all stale elements (marked to be deleted) and rebalances the list to have the minimum number of necessary memory units, releasing extra memory units for later allocations.

To delete a key, we start from the head and look for the matching key. If found, we mark the element with a tombstone. If not, we continue to the next slab. We continue this process until we reach the end of the list. For DELETE, we return after deleting the first matching element, but for DELETEALL we process the whole list.

In case we allow duplicates, we can simply mark a to-be-deleted element as empty. In this case, later insertions that use INSERT can potentially find these empty spots down the list and insert new items in them. However, if we do not allow duplicates, we must mark deleted elements differently to avoid inserting a key that already exists in the list (somewhere in its successive memory units).

5.3.3 Slab Hash: A Dynamic Hash Table

Our *slab hash* is a dynamic hash table built from a set of B slab lists (buckets). This hash table uses chaining as its collision resolution. More specifically, we use a direct-address table

of B buckets (*base slabs*), where each bucket corresponds to a unique hashed value from 0 to $B-1$ [23]. Each base slab is the head of an independent slab list consisting of slabs, as introduced in Section 5.3.1, each with M data points to be filled. In general, base slabs and regular slabs can differ in their structures in order to allow additional implementation features (e.g., pointers to the tail, number of slabs, number of stored elements, etc.). For simplicity and without loss of generality, here we assume there is no difference between them.

We use a simple universal hash function such as $h(k; a, b) = ((ak + b) \bmod p) \bmod B$, where a, b are random arbitrary integers and p is a random prime number. As a result, on average, keys are distributed uniformly among all buckets with an *average slab count* of $\beta = n/(MB)$ slabs per bucket, where n is the total number of elements in the hash table. For searching a key that does not exist in the table (i.e., an unsuccessful search), we should perform $\Theta(1 + \beta)$ memory accesses. A successful search is slightly better, but has similar asymptotic behavior.

In order to be able to compare our memory usage with open-addressing hash tables that do not use any pointers (e.g., cuckoo hashing [3]), we define the *memory utilization* to be the amount of memory actually used to store the data over the total amount of used memory (including pointers and unused empty slots). If each element and pointer take x and y bytes of memory respectively, then each slab requires $Mx + y$ bytes. As a result, on average, our slab list would achieve a memory utilization equal to $nx/(\beta(Mx + y))$. In order to achieve a certain memory utilization given a particular slab structure, we compute the required β, which is directly affected by the total number of buckets in the hash table. For open-addressing hash tables, memory utilization is equal to the load factor, i.e., the number of stored elements divided by the table size.

5.4 Implementation details

In this section we focus on our technical design choices and implementation details, primarily influenced by the hardware characteristics of NVIDIA GPUs.

5.4.1 Our warp-cooperative work sharing strategy

A traditional, but not necessarily efficient, way to perform a set of independent tasks on a GPU is to assign and process an independent task on each thread (e.g., classic linked list operations

on GPU [72]). An alternative approach is to do a per-warp work assignment followed by a per-warp processing (e.g., warp-wide histogram computation [6]). In this work we propose a new approach where threads are still assigned to do independent tasks (per-thread assignment), but works are done in parallel (per-warp processing). We call this a *warp-cooperative work sharing (WCWS)* strategy. In our data structure context, this means that we form a work queue of arbitrary requested operations from different threads within a warp, and all threads within that warp cooperate to process these operations one at a time (based on a pre-defined intra-warp order) and until the whole work queue is empty.

If data is properly distributed among the threads, as it is naturally so in our slab based design, then regular data structure operations such as looking for a specific element can simply be implemented in parallel using simple warp-wide instructions (e.g., using ballots and shuffles). An immediate advantage of the WCWS strategy is that it significantly reduces branch divergence when compared to traditional per-thread processing. A disadvantage of it is that we should always keep all threads within a warp active in order to correctly perform even a single task (avoiding branches on threads). But, this limitation already exists on many CUDA warp-wide instructions and can be easily avoided by using the same tricks [78, Chapter B.15].

5.4.2 Choice of parameters

As we emphasized in Section 5.3, the main motivation behind introducing slabs in our design is to have better coalesced memory accesses. Hence, we chose our slab sizes to be a multiple of each warp's physical memory access width, i.e., at least 32×4 B for current architectures. Throughout the rest of the chapter, we assume each slab is exactly 128 B, so that once a warp accesses a slab each thread has exactly 1/32 of the slab's content. So, when we use the term "lane" for a slab, we mean that portion of the slab that is read by the corresponding warp's thread. We currently support two item data types: 1) 32-bit entries (key-only), 2) 64-bit entries (key-value pairs), but our design can be extended to support other data types. In both cases, slab lanes 0–29 contain the data elements (in the key-value case, even and odd lanes contain keys and values respectively). We refer to lane 31 as the *address lane*, while lane 30 is used as an auxiliary element (flags and pointer information if required). As a result, slab lists (and the derived slab hash) can achieve a maximum memory utilization of 94%.

5.4.3 Operation details

Here we provide more details about some of slab hash operations discussed in Section 5.3.2. We thoroughly discuss SEARCH and REPLACE (insertion when uniqueness is maintained), and then briefly explain our methodology for DELETE and FLUSH operations. In our explanations, we use some simplified code snippets. For example, ReadSlab() takes a 32-bit address layout of a slab as input; each thread reads its corresponding data portion. SlabAddress() extends a 32-bit address layout to a 64-bit memory address (more details about memory address layouts are in Section 5.5).

5.4.3.1 SEARCH

Figure 5.2 shows a simplified pseudo-code for the SEARCH procedure in our slab hash. As an input, any thread that has a search query to perform sets `is_active` to true. Keys are stored in `myKey` and the result will be stored in `myValue`. By following the WCWS strategy introduced before, all threads within a warp participate in performing every search operation within that warp, one operation at a time. First, we form a local warp-wide work queue (line 3) by using a ballot instruction and asking whether any thread has something to search for. Then, all threads go into a while loop (line 4) and repeat until all search queries are processed. At each round, all threads can process the work queue and find the next lane within the warp that has the priority to perform its search query (the source lane, line 6). This is done by using a pre-defined procedure `next_prior()`, which can be implemented as simply as finding the first set bit in the work queue (using CUDA's `__ffs`). Then all threads ask for the source lane's query key using a shuffle instruction (line 6), and hash it to compute its corresponding bucket id (line 7).

The whole warp then performs a coalesced memory read from global memory (ReadSlab()), which takes the 32-bit address layout of the slab as well as the lane id of each thread. If we are currently at the linked list's base slab (the bucket head), we will find the corresponding slab's contents in a fixed array. Otherwise, we use our SlabAlloc allocator and compute the unique 64-bit address of that allocated slab by using the 32-bit `next` variable. Now, every thread has read its portion of the target slab. By using a ballot instruction we can ask whether any valid thread possesses the source lane's query (`src_key`), and then compute its position `found_lane` (line 14). If found, we ask for its corresponding value by using a shuffle instruction and asking

for its subsequent thread's `read_data`, which stores the result from the requested source lane. The source lane then stores back the result and marks its query as resolved (line 18). If not found, we must go to the next slab. To find it, we ask the address lane for its address (line 21) and update the `next_ptr`. If the `next_ptr` was empty, it means that we have reached the slab list's tail and the query does not exist (line 24). Otherwise, we update the `next` variable and continue within the next loop. At each loop, we initially check whether the work queue has changed (someone has successfully processed its query) or we are still processing the same query (but are now searching in allocated slabs rather than the base slab).

5.4.3.2 REPLACE

The main skeleton of the REPLACE procedure (Fig. 5.2) is similar to search, but now instead of looking for a particular key, we look for either that same key (to replace it), or an empty spot (to insert it for the first time). Any thread with an insertion operation will mark `is_active` as true. As with search, threads loop until the work queue of all insertion operations are completely processed. Within a loop, all threads read their corresponding portions of the target slab (lines 2–7), searching for the source key or an empty spot (an empty key-value pair) within the read slab (called the destination lane). If found, the source lane inserts its key-value pair into the destination lane's portion of the slab with a 64-bit atomicCAS operation. If that insert is successful, the source lane marks its operation as resolved, which will be reflected in the next work queue computation. If the insert fails, it means some other warp has inserted into that empty spot and the whole process should be restarted.

If no empty spot or source key is found at all, all threads fetch the next slab's address from the address lane. If that address is not empty, the new slab is read and the insertion process repeats. If the address is empty, it means that a new slab should be allocated. All threads use the SlabAlloc routine, allocating a new slab, then the source lane uses a 32-bit atomicCAS to update the empty address previously stored in the address lane. If the atomicCAS is successful, the whole insertion process is repeated with the newly allocated slab. If not, it means some other warp has successfully allocated and inserted the new slab and hence, this warp's allocated slab should be deallocated. The process is then restarted again with the new valid slab.

```
1: __device__ void warp_operation(bool &is_active, uint32_t &myKey, uint32_t &myValue) {
2: next ← BASE_SLAB;
3: work_queue ← __ballot(is_active);
4: while (work_queue != 0) do
5:     next ← (if work_queue is changed) ? (BASE_SLAB) : next;
6:     src_lane ← next_prior(work_queue); src_key ← __shfl(myKey, src_lane);
7:     src_bucket ← hash(src_key); read_data ← ReadSlab(next, laneId);
8:     warp_search_macro() OR warp_replace_macro()
9:     work_queue ← __ballot(is_active);
10: end while
11: }
12: // ========================================
13: warp_search_macro()
14: found_lane ← __ffs(__ballot(read_data == src_key) & VALID_KEY_MASK);
15: if (found_lane is valid) then
16:     found_value ← __shfl(read_data, found_lane + 1);
17:     if (laneId == src_lane) then
18:         myValue ← found_value; is_active ← false;
19:     end if
20: else
21:     next_ptr ← __shfl(read_data, ADDRESS_LANE);
22:     if (next_ptr is an empty address pointer) then
23:         if (laneId == src_lane) then
24:             myValue ← SEARCH_NOT_FOUND; is_active ← false;
25:         end if
26:     else
27:         next ← next_ptr;
28:     end if
29: end if
30: // ========================================
31: warp_replace_macro()
32: dest_lane ← __ffs(__ballot(read_data == EMPTY || read_data == myKey) & VALID_KEY_MASK);
33: if dest_lane is valid then
34:     if (src_lane == laneId) then
35:         old_pair ← atomicCAS(SlabAddress(next, dest_lane), EMPTY_PAIR, ⟨ myKey, myValue ⟩);
36:         if (old_pair == EMPTY_PAIR) then
37:             is_active ← false;
38:         end if
39:     end if
40: else
41:     next_ptr ← __shfl(read_data, ADDRESS_LANE);
42:     if next_ptr is empty then
43:         new_slab_ptr ← SlabAlloc::warp_allocate();
44:         if (laneId == ADDRESS_LANE) then
45:             temp ← atomicCAS(SlabAddress(next, ADDRESS_LANE), EMPTY_POINTER, new_slab_ptr);
46:             if (temp != EMPTY_POINTER) then
47:                 SlabAlloc::deallocate(new_slab_ptr);
48:             end if
49:         end if
50:     else
51:         next ← next_ptr;
52:     end if
53: end if
```

Figure 5.2: Psuedo-code for search (SEARCH) and insert (REPLACE) operations in slab hash.

5.4.3.3 DELETE

Deletion is very similar to our search procedure. Similarly, we use a boolean variable, `is_active`, to indicate whether each thread has something to delete or not. We form a work queue, and one by one threads fetch the source key to be deleted at that round (similar to lines 2–7 in Fig. 5.2). Starting from the source key's bucket, we look for the key in each slab. If found, we mark that key and its value as deleted.

5.4.3.4 FLUSH

Since we do not physically remove deleted elements in the slab hash but instead mark them as deleted, after a while it is possible to have slab lists that can be reorganized to occupy fewer slabs. FLUSH operation takes a bucket as an argument and then a warp processes all slabs within that bucket's slab list and compacts them into fewer slabs. In the end, we deallocate those emptied buckets in the SlabAlloc so that they can be reused by others. In order to guarantee correctness, we implement this operation as a separate kernel call so that no other thread can perform an operation in those buckets while we are flushing its contents.

5.5 Dynamic memory allocation

Motivation Today's GPU memory allocators (Section 5.2) are generally designed for variable-sized allocations and aim to avoid too much memory fragmentation (so as not to run out of memory with large allocations). These allocators are designed for flexibility and generality at the cost of high performance; for instance, they do not emphasize branch and memory-access divergence. For high-throughput mutability scenarios such as hash table insertions (e.g., the slab hash) that require many allocations, the memory allocator would be a significant bottleneck.

The WCWS strategy (Section 5.4.1) that we chose for the slab hash results in the following allocation problem: numerous independent fixed-size slab allocations per warp that may occur asynchronously and at different times during a warp's lifetime. These allocations cannot be formed into a single larger coalesced allocation that suits other allocators.

Consequently, current allocators perform poorly on this pattern of allocations. For example, on a Tesla K40c (ECC disabled), with one million slab allocations, 128 bytes per slab, one allocation per thread and with similar total used memory for each allocator, CUDA's `malloc`

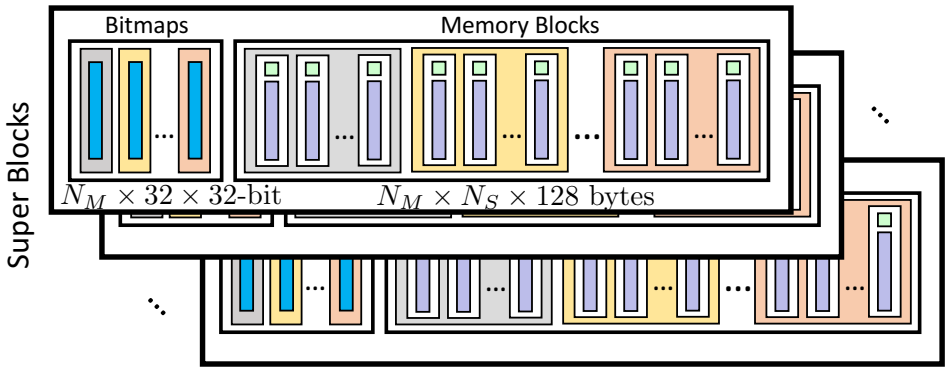


Figure 5.3: Memory layout for SlabAlloc.

spends 1.2s (0.8 M slabs/s). `Halloc` takes 66 ms (16.1 M slabs/s). We designed our own memory allocator that is better suited for this allocation workload. Our `SlabAlloc` takes 1.8 ms (600 M slabs/s), which is about 37x faster than the `Halloc`.

Terminology We use a hierarchical memory structure as follows: several (N_S) memory pools called *super blocks* are each divided into several (N_M) smaller *memory blocks*. Each memory block consists of fixed $N_U = 1024$ *memory units* (i.e., slabs). Figure 5.3 shows this hierarchy.

SlabAlloc There are a total of $N_S N_M$ memory blocks. We distribute memory blocks uniformly among all warps, such that different warps may be assigned to each memory block. We call each warp's assigned memory block *resident*. A resident block is used for all allocations requested from its warp owners for up to 1024 memory units (slabs). Once a resident block gets full, its warp randomly chooses (with a hash) another memory block and uses that as its new resident block. After a threshold number of resident changes, we add new super blocks and reflect them in the hash functions. In its most general case, both the super block and its memory block are chosen randomly using two different hash functions (taking the global warp ID and the total number of resident change attempts as input arguments). This creates a probing effect in the way we assign resident blocks. Since there are 1024 memory units within each memory block, by using just one 32-bit bitmap variable per thread (total of $32 \times$ 32-bit), a warp can fully store a memory block's full/empty availability.

Upon each allocation request, all threads in the warp look into their local resident bitmap (stored in a register) and announce whether there are any unused memory units in their portion of the memory block. For example, thread 0 is in charge of the first 32 memory units of its

resident block, thread 1 has memory units 32–63, etc. Following a pre-defined priority order (e.g., the least indexed unused memory unit), all threads then know which thread should allocate the next memory unit. That thread uses an atomicCAS operation to update its resident bitmap in global memory. If successful, then the newly allocated memory unit's address is shared with all threads within that warp (using shuffle instructions). If not, it means some other warp has previously allocated new memory units from this memory block and the local register-level resident bitmap should be updated. As a result, in the best case scenario and with low contention, each allocation can be addressed with just a single atomic operation. If necessary, each resident change requires a single coalesced memory access to read all the bitmaps for the new resident block. Deallocation is done by first locating the slab's memory block's bitmap in global memory and then atomically unsetting the corresponding bit.

Memory structure In general, we need a 64-bit pointer variable to uniquely address any part of the GPU's memory space. Almost all general-purpose memory allocators use the same format. Since our main target is to improve our data structure's dynamic performance, we trade off the generality of our allocators to gain performance: we use 32-bit *address layouts*, which are less expensive to store and share (especially because shuffle instructions work only with 32-bit registers). In order to uniquely address a memory unit, we use a 32-bit variable: 1) the first 10 bits represent the memory unit's index within its memory block, 2) the next 14 bits are used for memory block's index within its super block, and 3) the next 8 bits represent the super block. Each super block is assumed to be allocated continuously on a single array ($<$ 4 GB). Each memory unit is at least 2^7 bytes, and there are total of 1024 units in each memory block. Considering 2^7 bytes per each block's bitmap, we can at most put 2^{14} memory blocks within each super block (i.e., $(2^7 + 2^{17})N_M \leq 2^{32} \Rightarrow N_M < 2^{15}$). As a result, with this version we can dynamically allocate memory up to $2^7 N_S N_M N_U < 1$ TB (much larger than any current GPU's DRAM size).

In SlabAlloc, we assume that each super block is allocated as a contiguous array. In order to look up allocated slabs using our 32-bit layout, we store the actual beginning address (64-bit pointers) of each super block in shared memory. Before each memory access, we must first decode the 32-bit layout variables into an actual 64-bit memory address. This requires a single

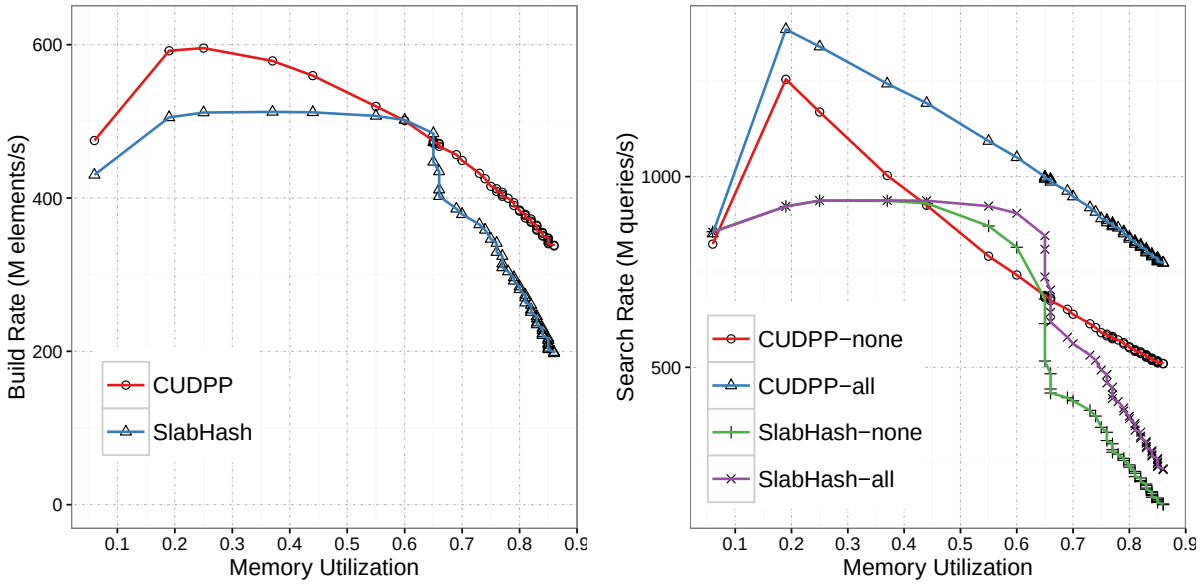
shared memory access per memory lookup (using the above 32-bit variable), which is costly, especially when performing search queries. To address this cost, we can also implemented a lightweight memory allocator, *SlabAlloc-light*, where all super blocks are allocated in a single contiguous array. In this case, a single beginning address for the first super block, which is stored as a global variable, is enough to find the address of all memory units, resulting in a less expensive memory lookup, but with less scalability (at most about 4 GB). In scenarios where memory lookups are heavily required (e.g., the bulk search scenarios in Section 5.6.1), SlabAlloc-light gives us up to 25% performance improvement compared to the regular SlabAlloc.

5.6 Performance Evaluation

We evaluate our slab list and slab hash on an NVIDIA Tesla K40c GPU (with ECC disabled), which has a Kepler microarchitecture with compute capability of 3.5, 12 GB of GDDR5 memory, and a peak memory bandwidth of 288 GB/s. We compile our codes with the CUDA 8.0 compiler (V8.0.61). In this section, all our insertion operations maintain uniqueness (REPLACE). We have also used SlabAlloc-light with 32 super blocks, 256 memory blocks, and 1024 memory units, 128 bytes each. We believe this is a fair comparison, because all other methods that we compare against (CUDPP and Misra's) pre-allocate a single contiguous array for use. If necessary, our SlabAlloc can be scaled up to 1 TB allocations. We divide our performance evaluations into two categories. First, we compare against other static hash tables (such as CUDPP's cuckoo hashing implementation [3]) in performing operations such as building the data structure from scratch and processing search queries afterwards. Second, we design a concurrent benchmark to evaluate the dynamic behavior of our proposed methods with a random mixture of certain operations performed asynchronously (insertion, deletion, and search queries). We compare the slab hash to Misra and Chaudhuri's lock-free hash table [72].

5.6.1 Bulk benchmarks (static methods)

There are two major operations defined for static hash tables such as CUDPP's hash table: (1) building the data structure given a fixed load factor (i.e., memory utilization) and an input array of key-value pairs, and (2) searching for an array of queries (keys) and returning an array of corresponding values (if found). By giving the same set of inputs into our slab hash, where each



(a) Building rate

(b) Search rate

Figure 5.4: Performance (M operations/s) versus memory efficiency. 2^{22} elements are stored in the hash table in each trial. (a) The whole table is built from scratch, dynamically, and in parallel. (b) There are 2^{22} search queries where all (or none) of them exist.

thread reads a key-value pair and dynamically inserts it into the data structure, we can build a hash table. Similarly, after the hash table is built, each thread can read a query from an input array and search for it dynamically in the slab hash and store back the search results into an output array. By doing so, we can compare slab hash with other static methods.[2]

For many data structures, the performance cost of supporting incremental mutability is significant: static data structures often sustain considerably better bulk-build and query rates when compared to similar data structures that additionally support incremental mutable operations. We will see, however, that the performance cost of supporting these additional operations in the slab hash is modest.

Figure 5.4a shows the build rate (M elements/s) for various memory utilizations. $n = 2^{22}$ elements are stored in the table. For CUDPP's hash, memory utilization (load factor) can be directly fixed as an input argument, but the situation is slightly more involved for the slab hash.[3]

[2]In the slab hash, there is no difference between a bulk build operation and incremental insertions of a batch of key-value pairs. However, for a bulk search we assign more queries to each thread.

[3]By choosing various average slab counts (from 0.1 to 5.0 in Fig. 5.4a), we can compute the initial number of buckets (from 4.2 M to 55 k buckets respectively). As a result, for each data point in the figure, we first build the slab hash and compute its memory utilization (the total size of stored elements divided by the total used memory),

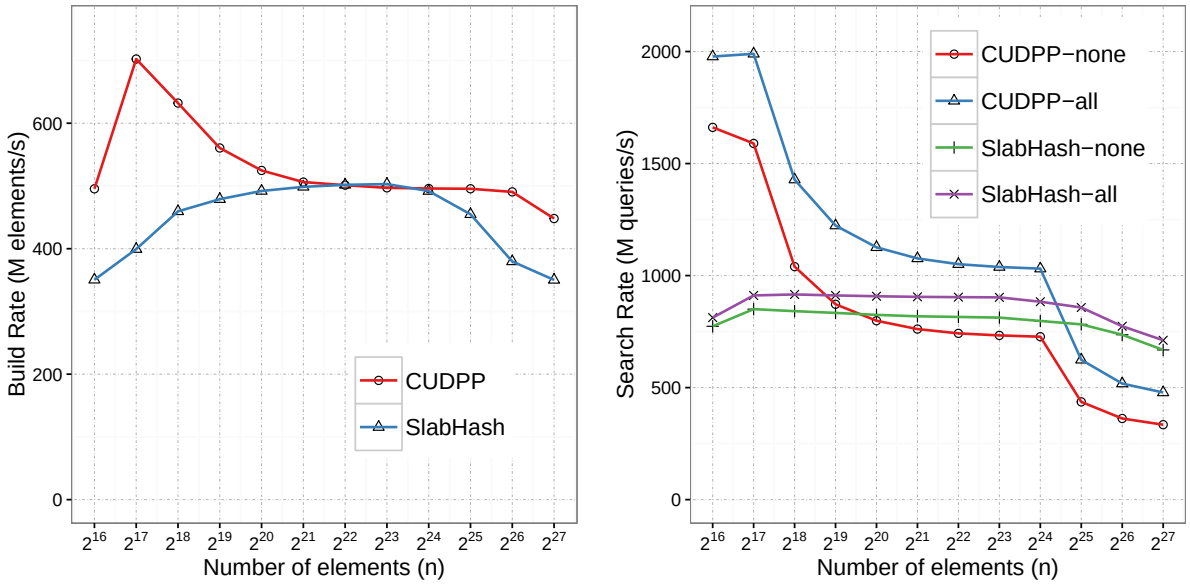
The process is averaged over 50 independent randomly generated trials. For search queries on a hash table with $n = 2^{22}$ elements, we generate two sets of $n = 2^{22}$ random queries: 1) all queries exist in the data structure; 2) none of the queries exist. These two scenarios are important as they represent, on average, the best and worst case scenarios respectively. Figure 5.4b shows the search rate (M queries/s) for both scenarios with various memory utilizations.

The slab hash gets its best performance from 19–60% memory utilization; these utilizations have a 0.2–0.7 average slab count. Intuitively, this is when the average list size fits in a single slab.[4] Peak performance is 512 M insertion/s and 937 M queries/s. At about 65% memory utilization there is a sudden drop in performance for both insertions and search queries. This drop happens when the average slab count is around 0.9–1.1, which means that almost all buckets will have more than one slab and most of the operations will have to visit the second slab. The slab hash appears to be competitive to cuckoo hashing, especially around 45–65% utilization. For example, our slab hash is marginally better in insertions (at 65%) and 1.1x faster in search queries when no queries exist. But, using a geometrical mean over all utilizations and $n = 2^{22}$, cuckoo hashing is 1.33x, 2.08x, and 2.04x faster than the slab hash for build, search-all, and search-none respectively.

Figure 5.5 shows the build rate (M elements/s) and search rate (M queries/s) vs. the total number of elements (n) stored in the hash table, where memory utilization is fixed to be 60% (an average slab count of 0.7). Here we witness that CUDPP's building performance is particularly high when the table size is small, which is because most of the atomic operations can be done in cache level. The slab hash saturates the GPU's resources for $2^{20} \leq n \leq 2^{24}$, where both methods perform roughly the same. For very large table sizes, both methods degrade, but the slab hash's performance decline starts at smaller table sizes. For search queries, the slab hash shows a relatively consistent performance with a harmonic mean of 861 and 793 M queries/s for search-all and search-none. In this experiment, with a geometric mean over all table sizes and 65% memory utilization, the speedup of CUDPP's cuckoo hashing over the slab hash is 1.19x, 1.19x, and 0.94x for bulk build, search-all, and search-none respectively.

and then build a CUDPP hash with the same utilization and with the same input elements.

[4]To choose the optimal number of initial buckets, we can choose the preferred memory utilization and its corresponding average slab count β from Fig. 5.4. Then $B = n/(M\beta)$, where $M = 15$, as discussed in Section 5.3.3 and 5.4.2.



(a) Building rate

(b) Search rate

Figure 5.5: Performance (M operations/s) versus total number of stored elements in the hash table. Memory utilization is fixed to be 60%. (a) The whole table is built from scratch, dynamically, and in parallel. (b) There are as many search queries as there are elements in the table where either all (or none) of them exist.

Ideally, the "fast path" scenario for CUDPP's cuckoo hash table requires a single atomicCAS for insertion and a single random memory access for a search. Unless there is some locality to be extracted from input elements (which does not exist in most scenarios), any hash table is doomed to have at least one global memory access (atomic or regular) per operation. This explains why CUDPP's peak performance is hard to beat, and other proposed methods such as stadium hashing [51] and Robin Hood hashing [36] are unable to compete with its peak performance. In the slab hash, for insertion, ideally we will have one memory access (reading the slab) and a single atomicCAS to insert into an empty lane. For search, it will be a single memory access plus some overhead from extra warp-wide instructions (Section 5.4).

5.6.2 Incremental insertion

Suppose we periodically add a new batch of elements to a hash table. For CUDPP, this means building from scratch every time. For the slab hash, this means dynamically inserting new elements into the same data structure. Figure 5.6 shows both methods in inserting new batches of different sizes (32k, 64k, and 128k) until there are 2 million elements stored in the hash table.

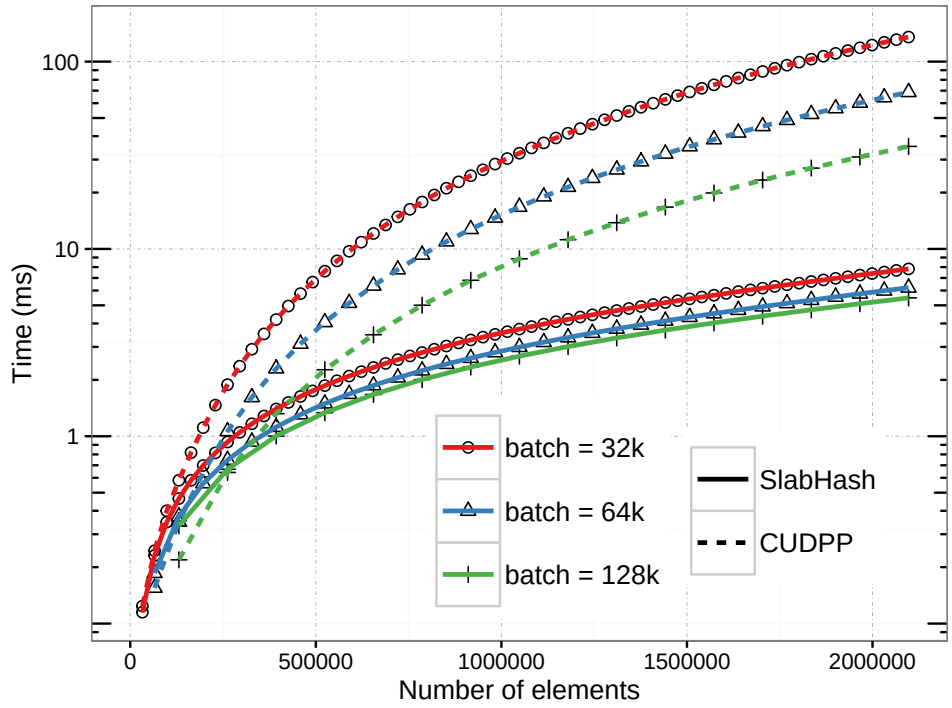


Figure 5.6: Incremental batch update for the slab hash, as well as building from scratch for the CUDPP's cuckoo hashing. Final memory utilization for both methods are fixed to be 65%. Time is reported in logarithmic scale.

For CUDPP, we use a fixed 65% load factor. For the slab hash, we choose initial number of buckets so that its final memory utilization (after inserting all batches) is 65%. As expected, the slab hash significantly outperforms cuckoo hashing by reaching final speedup of 6.4x, 10.4x, and 17.3x for batches of size 128k, 64k, and 32k. As the number of inserted batches increases (as with smaller batches), the performance gap increases.

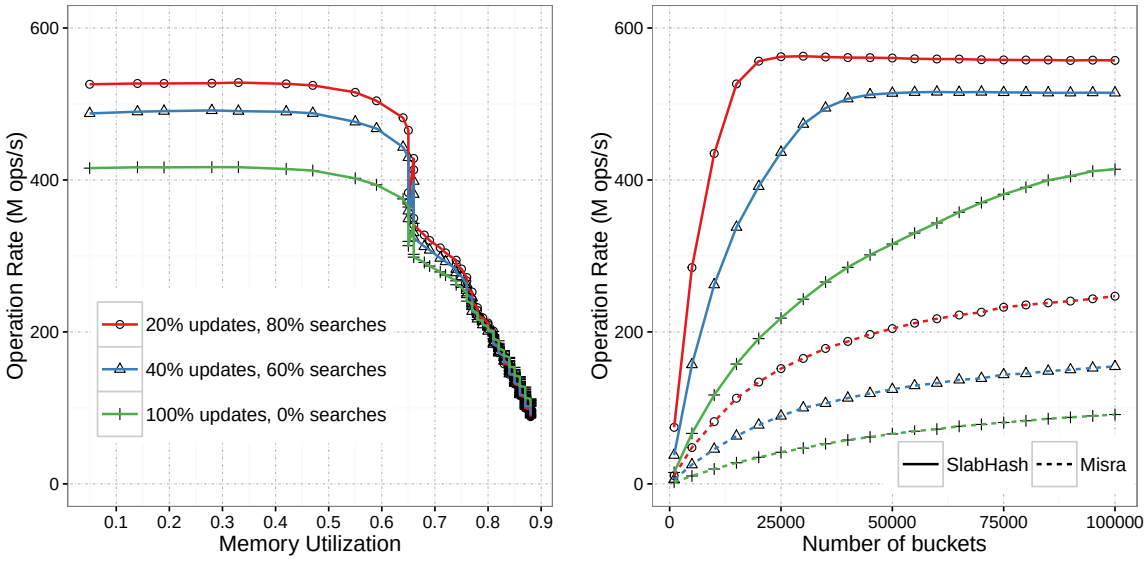
5.6.3 Concurrent benchmarks (dynamic methods)

One notable feature of the slab hash is its ability to perform truly concurrent query and mutation (insertion/deletion) operations without having to divide different operations into different computation phases. To evaluate our concurrent features, we design the following benchmark. Suppose we build our hash table with an initial number of elements. We then continue to perform operations in one of the following four categories: a) inserting a new element, b) deleting a previously inserted element, c) searching for an existing element, d) searching for a non-existing element. We define an *operation distribution* $\Gamma = (a, b, c, d)$, such that every item is non-negative and $a + b + c + d = 1$. Given any Γ, we can construct a random workload where, for instance, a

denotes the fraction of new insertions compared to all other operations. To ensure correctness, we generate operations in batches and process batches one at a time, but each in parallel. For each batch, operations are randomly assigned to each thread (one operation per thread) such that all four operations may occur within a single warp. In the end, we average the results over multiple batches. We consider three scenarios: 1) $\Gamma_0 = (0.5, 0.5, 0, 0)$ where all operations are updates, 2) $\Gamma_1 = (0.2, 0.2, 0.3, 0.3)$ where there are 40% updates and 60% search queries, and 3) $\Gamma_2 = (0.1, 0.1, 0.4, 0.4)$ where there are 20% updates and 80% search queries.

Figure 5.7a, shows the slab hash performance (M ops/s) for three different operation distributions and various initial memory utilizations. Since updates are computationally more expensive than searches, given a fixed memory utilization, performance gets better with fewer updates ($\Gamma_0 < \Gamma_1 < \Gamma_2$). Similar to the behavior in Fig. 5.4, the slab hash sharply degrades in performance with more than 65% memory utilization, falling to about 100 M ops/s with about 90% utilization. Comparing against our bulk benchmark in Fig. 5.4, it is clear that the slab hash performs slightly worse in our concurrent benchmark (e.g., Γ_0 in Fig 5.7a and Fig. 5.4a). There are two main reasons: (1) Since it is assumed that in static situations all operations are available, we can assign multiple operations per thread and hide potential memory-related latencies, and (2) in concurrent benchmarks we run three different procedures (one for each operation type) compared to the bulk benchmark that runs just one.

Misra's hash table Misra and Chaudhuri have implemented a lock-free hash table using classic linked lists [72]. This is a key-only hash table (i.e., an unordered set), without any pointer dereferencing or dynamic memory allocation; based on the required number of insertions, an array of linked list nodes are allocated at compile time, and then indices of that array are used in the linked lists. Since it uses a simplified version of a classic linked list (32-bit keys and 32-bit *next* indices), it theoretically can reach at most 50% memory utilization. In order to compare its performance with our slab hash, we use our concurrent benchmarks and the three operation distributions discussed above. Figure 5.7b shows performance (M ops/s) versus number of buckets, where each case has exactly one million operations to perform. The slab hash significantly outperforms Misra's hash table, with geometric mean speedup of 5.1x, 4.3x, and 3.1x for distributions with 100%, 40% and 20% updates respectively.



(a) Concurrent benchmark

(b) Performance vs. Misra's

Figure 5.7: (a) Concurrent benchmark for the slab hash: performance (M ops/s) versus initial memory utilization. (b) Performance (M ops/s) versus Misra and Chaudhuri's lock-free hash table [72]. Three different operation distributions are shown in different colors as shown in (a).

As discussed in Section 5.2, Moscovici et al. has recently proposed a lock-based skip list (GFSL). On a GeForce GTX 970, with 224 GB/s memory bandwidth, they report that its peak performance is about 100 M queries/s for searches and 50 M updates/s for updates (compared to our peak results of 937 and 512 M op/s respectively). In the best case, GFSL requires at least two atomic operations (lock/unlock) and two other regular memory accesses for a single insertion. This cost makes it unlikely that GFSL can outperform static cuckoo hashing (1 atomic/insert) or our dynamic slab hash (1 read and 1 atomic per insert) in their peak performance.

5.7 Conclusion

The careful consideration of GPU hardware characteristics as well as our warp-cooperative work sharing strategy lead us to design and implementation of an efficient dynamic hash table for GPUs. Beyond getting significant speedup compared to previous semi-dynamic hash tables, slab hash proves to be competitive to the fastest static hash tables too. We believe our slab list design and its utilization in slab hash can be a promising first step to provide a larger family of dynamic data structures with specialized analytics for them, which can also be used to target other interesting problems such as sparse data representation and dynamic graph analytics.

Chapter 6

Conclusion

Careful consideration of GPU hardware characteristics allowed us to design and implement a series of parallel algorithms and complex data structures in this dissertation. Developing efficient programs for GPUs requires an in-depth knowledge of both hardware characteristics as well as the problem to be solved. This dissertation shows that for many problems, the programmer can find a balance to craft an algorithm that not only shows good theoretical behavior but also exploits the hardware capabilities as much as possible.

Throughout this work, we have demonstrated the benefit of using warp-wide communication schemes whenever possible. Warp-wide shuffles and ballots provide another degree of freedom for the programmer to have a more preferred memory access pattern (a coalesced access) while also permitting the distribution of computational tasks among the threads. We also showed that by following our per-warp processing schemes and as a result of warp-privatization (either for per-thread/per-warp assignments), we can save some usage of more limited local resources (shared memory usage). This might sometimes appear very useful, such as in our multisplit-sort implementation. That being said, excessive usage of warp-wide instructions might not be a solution. For example, in our multisplit work, we could implement a warp-wide reordering scheme that only uses shuffles in its implementation, which would have completely removed our use of shared memories. That proved to be ineffective as the number of shuffle instructions was large enough that it canceled out the possible benefits from shared memory savings.

We believe separation of work assignment and processing has given us a new intuitive solution to deal with irregular workloads, especially in the context of complex data structures.

We want to remain cautious in drawing any strict conclusions about "the right" way to use GPUs; however, we believe any attempt to give an intuition to the programmer to understand certain methods in attacking certain problems can be a useful starting point to possibly design new high-level classifications for attacking new problems in the future. With the introduction of the cooperative-group feature in CUDA 9.0, it is possible in the future to consider per-thread assignment and per-group processing, for any arbitrary group that might benefit us in that particular application, or that particular real-time input data. However, current developments are still not complete and require much more maturity.

The underlying reason it made sense for us to consider per-warp processing was the existence of useful warp-wide communication instructions in CUDA. That made it feasible for us to be able to consider other positive benefits of per-warp processing, such as better memory access and fewer branch divergence. Any improvement in the performance for these instructions, as well as introducing more capabilities (e.g., being able to shuffle registers dynamically, or wider ballots to share more than a single bit per thread) would directly improve our performance as well. The possibility of having larger cooperative-groups (e.g., multiple warps) and then having shuffle/ballot communication scheme between them (or any communication medium that does not need costly synchronization or memory usage) will open a new world of possibilities to allow novel, efficient, and diverse algorithms and data structures.

Although we followed different high level strategies to perform specific tasks based on our collective needs, we see plenty of work to be done as an extension to these projects. Here we name some future directions:

Dynamic Data Structures: We considered two different perspectives toward dynamic data structures for GPUs: 1) bulk synchronous updates, 2) totally concurrent and asynchronous updates. We proposed the GPU LSM as an ordered dictionary for the former and the slab hash as an unordered hash table for the latter. Nevertheless, there is still a huge difference between the number of available dynamic data structures for a CPU programmer compared to a GPU programmer. Efficient implementation of other types of GPU data structures that also support dynamic updates is an interesting direction to follow. There are no efficient general-purpose implementation of B-trees on GPUs, even without supporting dynamic updates. Similarly,

an efficient trie data structure is also not available on GPUs (as opposed to multi-core CPU versions of concurrent tries [85]). Both of these data structures should provide a better search performance on GPUs (because of their parallel nature of dividing searched elements), but with a more challenging updates. They can be an alternative to our GPU LSM data structure, with better search performance but a worse update performance. But naturally, we only prefer these data structures over hash tables if we are space-limited or require ordered operations (e.g., iterating through elements with an order). Both of these data structures have large intermediate nodes that can be fit into the warp-cooperative strategy we used for our slab hash. We believe the same set of ideas can be a leading force to design and implement such data structures for the GPU.

Sparse Data Representation: We used slab lists as our building blocks for the slab hash. We believe slab lists can be extended to be used for more generic sparse data representations to potentially replace the traditional Compressed Sparse Row (CSR) format. CSR is very compact, which makes it hard to support any dynamic changes to the data structure. Instead, we can use a slab list per row to support dynamic updates. One possible problem of this approach is that our main design in the slab list was to provide the maximum performance for a hash table with a uniform distribution of elements per bucket. In dealing with a more general sparse data representation, and especially if we use it for high-level applications such as graphs, then it is very likely that we would see a very skewed distribution of elements among the rows, i.e., some vertices might have much larger out-degree than others. Consequently, there would be some lists that are much longer than others (in terms of average slab counts). These slab lists (i.e., vertices) are also much more likely to be touched for each update operation (to make sure whether the new element already exists in the list or not).

To solve this problem, instead of all slabs having the same size, we can tolerate different slab sizes. If we can tolerate phase updates (all updates being requested within the same kernel, without any search queries being performed), then we can easily implement a set of slab fusions (to fuse multiple smaller sized slabs and building a larger sized slab), or slab defusions (to do it in reverse). By processing these larger slabs with a thread-block or even the whole device we, can provide faster updates for those lists with a larger number of elements and with much less pointer dereferencing.

GPU Memory Management: Memory management on CPUs is a well-studied and mature field. For example, smart pointers in C++ or garbage collection in Java are universally used methods to increase the efficiency of overall utilized memory. There is absolutely nothing similar on the GPU, even in much weaker scenarios. As we discussed in Chapter 5, dynamic memory allocation is itself a challenging task on the GPU. There are several different proposed methods, each with some advantages and disadvantages. All these methods have no management over who will get to deallocate an already allocated memory. For example, what if some other thread is using an allocated memory (just by reading its contents in the global memory), while some other thread decides to deallocate it? We believe that the design and implementation of a complete memory management unit that controls such allocations and deallocations is a very interesting research topic that should be addressed in future work.

REFERENCES

[1] Andrew V. Adinetz and Dirk Pleiter. Halloc: a high-throughput dynamic memory allocator for GPGPU architectures, March 2014. https://github.com/canonizer/halloc.

[2] Peyman Afshani, Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Mayank Goswami, and Meng-Tsung Tsai. Cross-referenced dictionaries and the limits of write optimization. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, 2017.

[3] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics*, 28(5):154:1–154:9, December 2009. doi: 10.1145/1661412.1618500.

[4] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 781–792, November 2014. doi: 10.1109/SC.2014.69.

[5] Saman Ashkiani, Nina Amenta, and John D. Owens. Parallel approaches to the string matching problem on the GPU. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA 2016, pages 275–285, July 2016. doi: 10.1145/2935764.2935800.

[6] Saman Ashkiani, Andrew A. Davidson, Ulrich Meyer, and John D. Owens. GPU multisplit. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2016, pages 12:1–12:13, March 2016. doi: 10.1145/2851141.2851169.

[7] Saman Ashkiani, Andrew A. Davidson, Ulrich Meyer, and John D. Owens. GPU multisplit: an extended study of a parallel algorithm. *ACM Transactions on Parallel Computing*, 2017.

[8] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. A dynamic hash table for the GPU. *CoRR*, abs/1710.11246(1710.11246v1), October 2017.

[9] Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D. Owens. GPU LSM: A dynamic dictionary data structure for the GPU. *CoRR*, abs/1707.05354

(1707.05354v1), July 2017.

[10] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs: Theory, Algorithms and Applications*, chapter 3.3.4: The Bellman-Ford-Moore Algorithm, pages 97–99. Springer-Verlag London, 2009. doi: 10.1007/978-1-84800-998-1.

[11] Sean Baxter. Moderngpu: Patterns and behaviors for GPU computing. http://moderngpu.github.io/moderngpu, 2013–2016.

[12] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, February 1972. doi: 10.1007/BF00288683.

[13] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2, chapter 26, pages 359–371. Morgan Kaufmann, October 2011. doi: 10.1016/B978-0-12-385963-1.00026-5.

[14] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 81–92, June 2007. ISBN 978-1-59593-667-7. doi: 10.1145/1248377.1248393.

[15] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[16] Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977. doi: 10.1145/359842.359859.

[17] Jens Breitbart. Data structure design for GPU based heterogeneous systems. In *International Conference on High Performance Computing & Simulation*, HPCS '09, pages 44–51, June 2009. doi: 10.1109/HPCSIM.2009.5192780.

[18] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 546–554, Baltimore, MD, 2003.

[19] Shawn Brown and Jack Snoeyink. Modestly faster histogram computations on GPUs. In *Proceedings of Innovative Parallel Computing*, InPar '12, May 2012. doi: 10.1109/

InPar.2012.6339589.

[20] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004. doi: 10.1145/1015706.1015800.

[21] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446, April 2004. doi: 10.1137/1.9781611972740.43.

[22] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979. doi: 10.1145/356770.356776.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[25] Andrew Davidson, David Tarjan, Michael Garland, and John D. Owens. Efficient parallel merge sort for fixed and variable length keys. In *Proceedings of Innovative Parallel Computing*, InPar '12, May 2012. doi: 10.1109/InPar.2012.6339592.

[26] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-efficient parallel GPU methods for single source shortest paths. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2014, pages 349–359, May 2014. doi: 10.1109/IPDPS.2014.45.

[27] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, December 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663.

[28] Mrinal Deo and Sean Keely. Parallel suffix array and least common prefix for the GPU. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 197–206, February 2013. doi: 10.1145/2442516.2442536.

[29] Aditya Deshpande and P J Narayanan. Can GPUs sort strings efficiently? In *20th*

International Conference on High Performance Computing, HiPC 2013, pages 305–313, December 2013. doi: 10.1109/HiPC.2013.6799129.

[30] Gregory Frederick Diamos, Haicheng Wu, Ashwin Lele, Jin Wang, and Sudhakar Yalamanchili. Efficient relational algebra algorithms and data structures for GPU. Technical Report GIT-CERCS-12-01, Georgia Institute of Technology Center for Experimental Research in Computer Systems, February 2012. URL http://www.cercs.gatech.edu/tech-reports/tr2012/git-cercs-12-01.pdf.

[31] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0029-599X. doi: 10.1007/BF01386390.

[32] Simone Faro and Thierry Lecroq. Smart: a string matching algorithm research tool, 2011. URL http://www.dmi.unict.it/~faro/smart/.

[33] Simone Faro and Thierry Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Computing Surveys (CSUR)*, 45(2):13, February 2013. doi: 10.1145/2431211.2431212.

[34] Jordan Fix, Andrew Wilkes, and Kevin Skadron. Accelerating braided B+ tree searches on a GPU with CUDA. In *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance*, A4MMC 2011, June 2011.

[35] Zvi Galil. Optimal parallel algorithms for string matching. *Information and Control*, 67(1–3):144–157, October–December 1985. doi: 10.1016/S0019-9958(85)80031-0.

[36] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent parallel hashing. *ACM Transactions on Graphics*, 30(6):161:1–161:8, December 2011. doi: 10.1145/2070781.2024195.

[37] Oded Green and David A. Bader. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *2016 IEEE High Performance Extreme Computing Conference*, HPEC 2016, September 2016. doi: 10.1109/HPEC.2016.7761622.

[38] Kshitij Gupta, Jeff Stuart, and John D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Proceedings of Innovative Parallel Computing*, InPar '12, May 2012. doi: 10.1109/InPar.2012.6339596.

[39] Bronwyn H. Hall, Adam B. Jaffe, and Manuel Trajtenberg. The nber patent citation data file: Lessons, insights and methodological tools. Technical report, National Bureau of Economic Research, 2001.

[40] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Herbert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[41] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 511–524, June 2008. doi: 10.1145/1376616.1376670.

[42] Qiming Hou, Xin Sun, Kun Zhou, Christian Lauterbach, and Dinesh Manocha. Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):466–474, April 2011. doi: 10.1109/TVCG.2010.88.

[43] Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen-mei Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *IEEE 10th International Conference on Computer and Information Technology*, IEEE CIT-2010, pages 1134–1139. IEEE, June 2010. doi: 10.1109/CIT.2010.206.

[44] Yulong Huang, Benyue Su, and Jianqing Xi. CUBPT: Lock-free bulk insertions to B+ tree on GPU architecture. *Computer Modelling & New Technologies*, 18(10):224–231, 2014.

[45] Joseph JáJá. *An Introduction to Parallel Algorithms*, volume 17. Addison-Wesley Reading, 1992.

[46] Krzysztof Kaczmarski. B+-tree optimized for GPGPU. In Robert Meersman, Hervé Panetto, Tharam Dillon, Stefanie Rinderle-Ma, Peter Dadam, Xiaofang Zhou, Siani Pearson, Alois Ferscha, Sonia Bergamaschi, and Isabel F. Cruz, editors, *On the Move to Meaningful Internet Systems: OTM 2012*, volume 7566 of *Lecture Notes in Computer Science*, pages 843–854. Springer Berlin Heidelberg, September 2012. doi: 10.1007/978-3-642-33615-7_27.

[47] Tim Kaldewey and Andrea Di Blas. Large-scale GPU search. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2, chapter 1, pages 3–14. Morgan Kaufmann, October

2011. doi: 10.1016/B978-0-12-385963-1.00001-0.

[48] Tim Kaldewey and Andrea Di Blas. Large-scale GPU search. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2, chapter 1, pages 3–14. Morgan Kaufmann, October 2011. doi: 10.1016/B978-0-12-385963-1.00001-0.

[49] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987. doi: 10.1147/rd.312.0249.

[50] Tero Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *High-Performance Graphics*, HPG '12, pages 33–37, June 2012. doi: 10.2312/EGGH/HPG12/033-037.

[51] Farzad Khorasani, Mehmet E. Belviranli, Rajiv Gupta, and Laxmi N. Bhuyan. Stadium hashing: Scalable and flexible hashing on GPUs. In *International Conference on Parallel Architecture and Compilation*, PACT 2015, pages 63–74, October 2015. doi: 10.1109/PACT.2015.13.

[52] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 339–350, June 2010. doi: 10.1145/1807167.1807206.

[53] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi: 10.1137/0206024.

[54] Moritz Kobitzsh. 10th dimacs implementation challenge. 2010. URL http://www.cc.gatech.edu/dimacs10/index.shtml.

[55] Daniel Kopta, Thiago Ize, Josef Spjut, Erik Brunvand, Al Davis, and Andrew Kensler. Fast, effective BVH updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 197–204, March 2012. doi: 10.1145/2159616.2159649.

[56] Charalampos S. Kouzinopoulos and Konstantinos G. Margaritis. String matching on a multicore GPU using CUDA. In *13th Panhellenic Conference on Informatics*, PCI '09,

pages 14–18. IEEE, September 2009. doi: 10.1109/PCI.2009.47.

[57] Thierry Lecroq. Fast exact string matching algorithms. *Information Processing Letters*, 102(6):229–235, June 2007. doi: 10.1016/j.ipl.2007.01.002.

[58] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics*, 25(3):579–588, July 2006. doi: 10.1145/1141911.1141926.

[59] Aaron E. Lefohn, Joe Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, January 2006. doi: 10.1145/1122501.1122505.

[60] Francesco Lettich, Claudio Silvestri, Salvatore Orlando, and Christian S. Jensen. GPU-based computing of repeated range queries over moving objects. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 640–647, February 2014. doi: 10.1109/PDP.2014.27.

[61] Wei Liao, Zhimin Yuan, Jiasheng Wang, and Zhiming Zhang. Accelerating continuous range queries processing in location based networks on GPUs. In *Management Innovation and Information Technology*, pages 581–589, 2014. doi: 10.2495/MIIT130751.

[62] Cheng-Hung Lin, Chen-Hsiung Liu, and Shih-Chieh Chang. Accelerating regular expression matching using hierarchical parallel machines on GPU. In *IEEE Global Telecommunications Conference*, GLOBECOMM 2011, pages 1–5. IEEE, December 2011. doi: 10.1109/GLOCOM.2011.6133663.

[63] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March/April 2008. doi: 10.1109/MM.2008.31.

[64] Lijuan Luo, Martin D. F. Wong, and Lance Leong. Parallel implementation of R-trees on the GPU. In *2012 17th Asia and South Pacific Design Automation Conference*, ASP-DAC 2012, pages 353–358, January 2012. doi: 10.1109/ASPDAC.2012.6164973.

[65] Duane Merrill. CUB, 2011. URL nvlabs.github.io/cub/.

[66] Duane Merrill. Cub: Flexible library of cooperative threadblock primitives and other utilities for CUDA kernel programming. https://nvlabs.github.io/cub/, 2013–2016.

[67] Duane Merrill. CUDA UnBound (CUB) library, 2015. https://nvlabs.github.io/cub/.

[68] Duane Merrill and Andrew Grimshaw. Parallel scan for stream architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, December 2009.

[69] Duane Merrill and Andrew Grimshaw. Revisiting sorting for GPGPU stream architectures. Technical Report CS2010-03, Department of Computer Science, University of Virginia, February 2010. URL https://sites.google.com/site/duanemerrill/RadixSortTR.pdf.

[70] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, October 2003. doi: 10.1016/S0196-6774(03)00076-2. 1998 European Symposium on Algorithms.

[71] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82. ACM, August 2002. doi: 10.1145/564870.564881.

[72] Prabhakar Misra and Mainak Chaudhuri. Performance evaluation of concurrent lock-free data structures on GPUs. In *IEEE 18th International Conference on Parallel and Distributed Systems*, ICPADS 2012, pages 53–60. IEEE, December 2012. doi: 10.1109/ICPADS.2012.18.

[73] Laura Monroe, Joanne Wendelberger, and Sarah Michalak. Randomized selection on the GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 89–98, August 2011. doi: 10.1145/2018323.2018338.

[74] Nurit Moscovici, Nachshon Cohen, and Erez Petrank. Poster: A GPU-friendly skiplist algorithm. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 449–450, February 2017. ISBN 978-1-4503-4493-7. doi: 10.1145/3018743.3019032.

[75] Todd Mostak. Using GPUs to accelerate data discovery and visual analytics. In *Future Technologies Conference*, FTC 2016, pages 1310–1313. IEEE, December 2016. doi: 10.1109/FTC.2016.7821771.

[76] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, March/April 2008. doi: 10.1145/1365490.1365500.

[77] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Bart Mesman. High performance predictable histogramming on GPUs: Exploring and evaluating algorithm trade-offs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 1:1–1:8, 2011. ISBN 978-1-4503-0569-3. doi: 10.1145/1964179.1964181.

[78] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2016. Version 8.0.

[79] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2016. Version 9.0.

[80] NVIDIA Corporation. CUDA math API, 2017. Version 9.0.

[81] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996. doi: 10.1007/s002360050048.

[82] Jacopo Pantaleoni. VoxelPipe: A programmable pipeline for 3D voxelization. In *Proceedings of High Performance Graphics*, HPG '11, pages 99–106, August 2011. ISBN 978-1-4503-0896-0. doi: 10.1145/2018323.2018339.

[83] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, 29(4): 66:1–66:13, July 2010. doi: 10.1145/1778765.1778803.

[84] Suryakant Patidar. Scalable primitives for data mapping and movement on the GPU. Master's thesis, International Institute of Information Technology, Hyderabad, India, June 2009.

[85] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 151–160, February 2012. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145836.

[86] Michael C. Schatz and Cole Trapnell. Fast exact string matching on the GPU. Technical

report, University of Maryland, 2007.

[87] Elizabeth Seamans and Thomas Alexander. Fast virus signature matching on the GPU. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 35, pages 771–783. Addison-Wesley, August 2007.

[88] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, August 2007. doi: 10.2312/EGGH/EGGH07/097-106.

[89] R. Shams and R. A. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proceedings of the International Conference on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422, Gold Coast, Australia, December 2007.

[90] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *Innovative Parallel Computing*, InPar '12. IEEE, May 2012. doi: 10.1109/InPar.2012.6339604.

[91] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of High Performance Graphics*, HPG '10, pages 29–37, June 2010. doi: 10.2312/EGGH/HPG10/029-037.

[92] M. Vinkler and V. Havran. Register efficient dynamic memory allocator for GPUs. *Computer Graphics Forum*, 34(8):143–154, December 2015. ISSN 1467-8659. doi: 10.1111/cgf.12666.

[93] Uzi Vishkin. Optimal parallel pattern matching in strings. *Information and Control*, 67 (1–3):91–113, October–December 1985. doi: 10.1016/S0019-9958(85)80028-0.

[94] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2016, pages 11:1–11:12, March 2016. doi: 10.1145/2851141.2851145.

[95] Zhefeng Wu, Fukai Zhao, and Xinguo Liu. SAH KD-tree construction on GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG

'11, pages 71–78, August 2011. doi: 10.1145/2018323.2018335.

[96] Ke Yang, Bingsheng He, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, Pedro Sander, and Jiaoying Shi. In-memory grid files on graphics processors. In *Proceedings of the 3rd International Workshop on Data Management on New Hardware*, DaMoN '07, pages 5:1–5:7, 2007. doi: 10.1145/1363189.1363196.

[97] Simin You, Jianting Zhang, and Le Gruenwald. Parallel spatial query processing on GPUs using R-trees. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, BigSpatial '13, pages 23–31, November 2013. doi: 10.1145/2534921.2534949.

[98] Xinyan Zha and Sartaj Sahni. GPU-to-GPU and host-to-host multipattern string matching on a GPU. *IEEE Transactions on Computers*, 62(6):1156–1169, June 2013. ISSN 0018-9340. doi: 10.1109/TC.2012.61.

[99] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):126:1–126:11, December 2008. doi: 10.1145/1409060.1409079.