INDEX-BASED SEARCH TECHNIQUES FOR VISUALIZATION AND DATA ANALYSIS ALGORITHMS ON MANY-CORE SYSTEMS

by

BRENTON JOHN LESSLEY

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2019

DISSERTATION APPROVAL PAGE

Student: Brenton John Lessley

Title: Index-Based Search Techniques for Visualization and Data Analysis Algorithms on Many-Core Systems

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Hank Childs	Chair
Boyana Norris	Core Member
Christopher Wilson	Core Member
Eric Torrence	Institutional Representative

and

Janet Woodruff-Borden	Vice Provost and Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2019

DISSERTATION ABSTRACT

Brenton John Lessley

Doctor of Philosophy

Department of Computer and Information Science

June 2019

Title: Index-Based Search Techniques for Visualization and Data Analysis Algorithms on Many-Core Systems

Sorting and hashing are canonical index-based methods to perform searching, and are often sub-routines in many visualization and analysis algorithms. With the emergence of many-core architectures, these algorithms must be rethought to exploit the increased available thread-level parallelism and data-parallelism. Data-parallel primitives (DPP) provide an efficient way to design an algorithm for scalable, platform-portable parallelism. This dissertation considers the following question: What are the best index-based search techniques for visualization and analysis algorithms on diverse many-core systems? To answer this question, we develop new DPP-based techniques, and evaluate their performance against existing techniques for data-intensive visualization and analysis algorithms across different many-core platforms. Then, we synthesize our findings into a collection of best practices and recommended usage. As a result of these efforts, we were able to conclude that our techniques demonstrate viability and leading platform-portable performance for several different search-based use cases. This dissertation is a culmination of previously-published co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Brenton John Lessley

DEGREES AWARDED:

Ph.D. in Computer Science	University of Oregon, 2019
M.S. in Computer Science	University of Washington Tacoma, 2011
B.S. in Business Administration	University of Washington Tacoma, 2009

AREAS OF SPECIAL INTEREST:

Data-Parallel Algorithms
Parallel Data Structures
High-Performance Computing
Scientific Visualization

PROFESSIONAL EXPERIENCE:

Graduate Research & Teaching Fellow	University of Oregon, 2013 - 2019
Graduate Student Researcher	Lawrence Berkeley Nat'l Lab, Summer 2017
Graduate Student Researcher	Lawrence Berkeley Nat'l Lab, Summer 2016
Student Developer	Google Summer of Code, Summer 2015

PUBLICATIONS:

Brenton Lessley, Samuel Li, and Hank Childs (2019). HashFight: A Platform-Portable Hash Table for Multi-Core and Many-Core Architectures (in preparation).

Brenton Lessley and Hank Childs (2019). Data-Parallel Hashing Techniques for GPU Architectures (in submission).

Brenton Lessley, Talita Perciano, Colleen Heinemann, David Camp, Hank Childs, and E. Wes Bethel (2018). DPP-PMRF: Rethinking Optimization for a Probabilistic Graphical Model Using Data-Parallel Primitives. *Proceedings of the IEEE Symposium on Large Data Visualization and Analysis (LDAV)*, Berlin, Germany, pp. 1–11.

Brenton Lessley, Kenneth Moreland, Matthew Larsen and Hank Childs (2017). Techniques for Data-Parallel Searching for Duplicate Elements. *Proceedings of the IEEE Symposium on Large Data Visualization and Analysis (LDAV)*, Phoenix, AZ, pp. 1–5.

Brenton Lessley, Talita Perciano, Manish Mathai, Hank Childs and E. Wes Bethel (2017). Maximal Clique Enumeration with Data-Parallel Primitives. *Proceedings of the IEEE Symposium on Large Data Visualization and Analysis (LDAV)*, Phoenix, AZ, pp. 16–25.

Brenton Lessley, Roba Binyahib, Robert Maynard, and Hank Childs (2016). External Facelist Calculation with Data-Parallel Primitives. *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, Groningen, The Netherlands, pp. 10–20.

Sergio Davalos, Altaf Merchant, Gregory Rose, **Brenton Lessley**, and Ankur Teredesai (2015). 'The Good Old Days': An Examination of Nostalgia in Facebook Posts. *International Journal of Human-Computer Studies*, vol. 83, pp. 83–93.

Daniel Lowd, **Brenton Lessley**, and Mino De Raj (2014). Towards Adversarial Reasoning in Statistical Relational Domains. *AAAI-14 Workshop on Statistical Relational AI (Star AI 2014)*, Quebec City, Canada.

Brenton Lessley, Matthew Alden, Daniel Bryan, and Arindam Tripathy (2012). Detection of Financial Statement Fraud Using Evolutionary Algorithms. *Journal of Emerging Technologies in Accounting*, vol. 9, no. 1, pp. 71–94.

ACKNOWLEDGEMENTS

The path towards completing my Ph.D. has been quite an adventure, always seeing the light at the end of the tunnel, yet always one more challenging task to complete. In short, the past six years have more or less been a mind-training game that teaches me how to stay focused on a goal and work towards it tirelessly until completion, despite countless thoughts of giving up. To reach this point took more self-motivation than I ever knew I had and even more motivation and encouragement from those around me.

Throughout the Ph.D. process, my parents have provided the best support, advice, and encouragement that I could have asked for. They have been my number one cheering squad and, with great humor, make sure that I always remain optimistic, from one paper or milestone to the next. I cannot thank them enough.

The same thank you goes to my very good friend, Agrima, who has patiently heard every high note and low note from me while completing this long process. I know she sacrificed a lot of fun to join in on my many work sessions or postpone a trip due to a looming milestone. I truly appreciate her patience and sense of humor, and can't thank her enough.

If it wasn't for my advisor, Hank Childs, I likely would not have even completed the PhD degree. From the day I joined the CDUX research group and planned out a road map towards completing the PhD, Hank has helped me become a self-sufficient researcher and provided countless opportunities to hone my area of expertise. I thank him for all the guidance and support he has provided me, particularly in the past year. I am proud of all the work that we have completed and published during the four years under his advisement.

During my time at the University of Oregon, I have made some of my best friends, who provided so many nice memories and made the hard work much more enjoyable, even if it was as simple as going to a basketball or football game after completing some goal, holding a house party, or going to Starbucks for a coding session. You know who you are—thank you, you are the best.

I was fortunate to have many awesome officemates during my six years in Deschutes Hall, all of whom provided plenty of great humor, language learning, and a benchmark for working harder. I can confidently say that our whiteboards were rarely empty. To Ali, Pedram, Azaad, Soheil, Sam, Vince, Stephanie, Abhishek, Manish, Garrett, Sudhanshu, and Steve, it was a great pleasure. This thank you also extends to my fellow labmates and friends in CDUX. From the group parties and practice presentations to random office conversations, your sense of humor and friendship were awesome.

While most of my time was spent at the University of Oregon, I also spent two great summers at the Lawrence Berkeley National Lab with the Data Analytics & Visualization Group. The research projects and papers I completed during this time really boosted my confidence as a researcher and led to a pipeline of exciting work. I was lucky to have Wes Bethel and Talita Perciano as my mentors, and I appreciated the discussions and collaboration with David Camp, Colleen Heinemann, Dmitriy Morozov, Sugeerth Murugesan, Dani Ushizima, Burlen Loring, and Hari Krishnan.

Finally, I would like to thank Boyana Norris, Chris Wilson, and Eric Torrence for their willingness to serve on my PhD committee and provide great feedback throughout the dissertation phase.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

Figure Page

LIST OF TABLES

Table Page

LIST OF ALGORITHMS

Algorithm Page

LIST OF LISTINGS

Listing Page

CHAPTER I

INTRODUCTION

Within the scientific visualization and data analysis domains, a large body of algorithms are built upon search-based subroutines, in which one or more elements are queried within a larger set of elements. Canonical methods to perform this search task are primarily based on sorting, hashing, and spatial partitioning [66], each using an implementation-specific data structure to efficiently facilitate the searching operation. These data structures can be as simple as a sorted (ordered) variant of the original unordered elements, a hash table, or a tree-based partitioning of the elements (e.g., with a kd-tree, octree, or binary search tree). In searching via sorting and hashing, the sorted array and hash table can be considered *index-based* data structures, since elements are logically stored and indexed within a linear array.

The evolution of hardware architectures over the past two decades have informed the research directions for index-based search techniques. With the emergence of multi-processor CPU architectures and thread-based programming, significant research initially focused on the design of lock-free, parallel index-based search techniques for single-node, shared memory [101, 134, 45]. Subsequently, studies began to investigate external-memory (off-chip) and multi-node, distributed-memory parallel techniques that could accommodate the oncoming shift towards large-scale data processing [15, 22]. Over the past decade, computational platforms have advanced from nodes containing small numbers of CPU cores to nodes containing many-core GPU accelerators with massive threading and memory bandwidth capabilities. With this increase in available parallelism comes the ability to process larger workloads and data in parallel [95, 54]. By explicitly parallelizing fine-grained computations that operate on this data, scalable *data-parallelism* can be attained, whereby a single instruction is performed over multiple data elements (SIMD) in parallel (e.g., via a vector instruction), as opposed to over single scalar data values (SISD).

Traditional single-threaded (non-parallel) and multi-threaded index-based search designs, however, do not demonstrate node-level scalability for the massive number of concurrent threads and data-parallelism offered by emerging many-core architectures, particularly GPU accelerators. Hardware differences in on-chip caching, shared memory, instruction execution (e.g., vectorization and branch prediction), and threading models require new code implementations to achieve optimal performance on these emerging architectures. Additionally, algorithms that use these search techniques are not structurally designed to exploit increased available parallelism, e.g., due to significant non-parallel code and an insufficient amount of parallel work.

For software developers and projects, this has led to a significant change. Algorithms cannot simply be ported over to a new architecture or platform while retaining optimal performance, and they may need to be supported on multiple architectures simultaneously. One approach to this problem is to maintain a different algorithm implementation for each different architecture. For example, developers can provide an efficient NVIDIA CUDA implementation [115] for NVIDIA GPUs and an efficient Intel TBB implementation [53] for Intel CPUs (or $N \times M$ separate implementations for N algorithms and M architectures). However, this approach has drawbacks, as it increases software development time and is neither adaptable nor future-proof to new processor architectures, and also would require separate algorithm implementations.

A more preferred approach is to re-think an algorithm in terms of *data-parallel primitives* (DPPs), which are highly-optimized “building block” operations that are combined together to compose a larger algorithm. To qualify as a DPP, an operation must execute in $O(\log N)$ time on an array of size N, given the availability of N or more processing units (e.g., parallel hardware threads or cores) [11]. Well-known operations, such as map, reduce, gather, scatter, scan (prefix sum), and sort all meet this property, and are some of the most commonly-used DPPs. By providing efficient implementations of each DPP for each different platform architecture, a single algorithm composed of DPPs can be executed efficiently across multiple platforms. The DPP paradigm thus provides a way to explicitly design and program an algorithm for scalable, platform-portable data-parallelism, as increases in processing units and data enable unrestricted increases in speedup.

In this dissertation work, we answer the following dissertation question: *What are the best index-based search techniques for visualization and analysis algorithms on diverse many-core systems?* To answer this question, we needed to develop a novel hashing-based search technique that is designed entirely in terms of DPPs. Additionally, we propose novel methods of using existing sorting- and hashing-based search techniques. We introduce these techniques in two phases:

1. Design sorting- and hashing-based techniques for the search of duplicate elements.
2. Expand the hashing-based technique to a general-purpose hash table data structure.

Then, we consider three data-intensive visualization and/or analysis algorithms, each of which has a search-oriented procedure as part of the algorithm. In each case, our first step is to re-think these algorithms in terms of these techniques and other DPPs. Finally, we look for patterns and commonalities across these solutions, and synthesize our findings with a set of best practices and a decision tree flowchart that guides the selection of a most-suitable search technique.

The culmination of this work answers our dissertation question, i.e., identifying the best index-based search techniques for visualization and analysis algorithms on diverse many-core systems. This question is the subject of this dissertation.

1.1 Dissertation Outline

This dissertation is organized into the following three parts:

1. Techniques: Chapters II through IV.
2. Applications: Chapters V through VIII.
3. Best Practices: Chapters IX and X.

Part I surveys existing index-based search techniques and then introduces new data-parallel index-based search techniques. Part II incorporates the techniques of Part I into the design of data-parallel algorithms for different scientific visualization and data analysis algorithms. Part III synthesizes the findings and best practices of Parts I and II.

In particular, the content of the individual dissertation chapters is as follows. Chapter II provides a background data-parallelism and existing index-based search techniques. Chapter III introduces new data-parallel sorting- and hashing-based techniques for duplicate element searching. Chapter IV expands the hashing-based techniques of Chapter III to a general-purpose hash table data structure. Chapter V applies the techniques of Chapter III to the scientific visualization algorithm of external facelist calculation (EFC). Chapter VI applies the techniques of Chapter IV to the task of hashing unsigned integers. Chapter VII applies the techniques of Chapter III to the graph algorithm of maximal clique enumeration (MCE). Chapter VIII applies the techniques of Chapter III to graph-based image segmentation using Markov random fields (MRF). Chapter IX identifies specific test configurations and use cases that lead to the best performance for the duplicate element searching task of Chapters III and V. Chapter X concludes this dissertation by synthesizing the findings and best practices of the previous chapters and offering recommendations for future research.

1.2 Co-Authored Material

A significant portion of the content in this dissertation is adopted from collaborative research work and manuscripts that I have completed, as lead author, during my PhD program. Each manuscript has been either already published or is currently under submission at a journal. The content of each manuscript includes the text, figures, and experimental results, all of which are primarily composed by myself. The following listing indicates the chapters that contain manuscript content and the authors that contributed to the manuscript (i.e., myself and co-authors); note that a detailed division of labor for each manuscript is provided at the beginning of its corresponding chapter.

- Chapter II is mainly based on an under-submission journal publication, and is a collaboration between Hank Childs and myself.
- Chapters III and V are mainly based on an accepted conference publication, and is a collaboration between Roba Binyahib, Robert Maynard, Hank Childs, and myself. The content of this publication is divided between the two chapters, without any overlap.
- Chapters IV and VI are mainly based on an under-submission journal publication, and is a collaboration between Samuel Li, Hank Childs, and myself. The content of this publication is divided between the two chapters, without any overlap.
- Chapter VII is mainly based on an accepted conference publication, and is a collaboration between Talita Perciano, Manish Mathai, Hank Childs, Wes Bethel, and myself.
- Chapter VIII is mainly based on an accepted conference publication, and is a collaboration between Talita Perciano, Colleen Heinemann, David Camp, Hank Childs, Wes Bethel, and myself.

- Chapter IX is mainly based on an accepted conference publication, and is a collaboration between Kenneth Moreland, Matthew Larsen, Hank Childs, and myself.

Part I

Techniques

In this part of the dissertation, we provide a background on data-parallelism and index-based searching, and then introduce our collection of DPP-based search techniques for duplicate element detection and hashing.

CHAPTER II

BACKGROUND

The task of searching for elements in an indexed array is a well-studied problem in computer science. Canonical methods for this task are primarily based on sorting and hashing [66]. This chapter provides the necessary background to understand these methods and the work of this dissertation. Specifically, the following concepts are covered in sequence, each in their own section:

1. Parallel Computing and Data-Parallelism
2. Data-Parallel Primitives
3. General Purpose GPU Computing
4. Index-Based Searching
5. Data-Parallel Hashing Techniques

The material in this chapter is primarily adopted from a collaborative survey manuscript completed by myself and Hank Childs [80]. As lead author of this manuscript, I contributed the majority of the literature review and paper writing. Hank Childs provided significant guidance towards optimizing the scope of the survey, improving the message of our contributions, and editing the final submission. This manuscript is currently in preparation for a journal submission and is based on the work of my Ph.D. area exam.

2.1 Parallel Computing & Data-Parallelism

Lamport [73] defines *concurrency* as the decomposition of a process into independently-executing events (subprograms or instructions) that do not causally affect each other. *Parallelism* occurs when these events are all executed at the same time and perform roughly the same work. According to Amdahl [4], a program contains both non-parallelizable, or serial, work and parallelizable work. Given *P* processors (e.g., hardware cores or threads) available to perform parallelizable work, *Amdahl's Law* defines the *speedup* S_P of a program as $S_P \leq T_1/T_P$, where T_1 and T_P are the times to complete the program with a single processor and P processors, respectively. As $P \to \infty$, $S_\infty \leq \frac{1}{f}$, where f is the fraction of serial work in the program. So, the speedup, or *scalability*, of a program is limited by its inherent serial work, as the number of processors increases. Ideally, a linear speedup is desired, such that P processors achieve a speedup of P; a speedup proportional to P is said to be scalable.

Often a programmer writes and executes a program without explicit design for parallelism, assuming that the underlying hardware and compiler will automatically deliver a speedup via greater processor cores and transistors, instruction pipelining, vectorization, memory caching, etc [54]. While these automatic improvements may benefit *perfectly parallelizable work*, they are not guaranteed to address *imperfectly parallelizable* work that contains data dependencies, synchronization, high latency cache misses, etc [95]. To make this work perfectly parallelizable, the program must be refactored, or redesigned, to expose more *explicit* parallelism that can increase the speedup (S_P). Brent [16] shows that this explicit parallelism should first seek to minimize the *span* of the program, which is the longest chain of tasks that must be executed sequentially in order. Defining T_1 as the total serial work and T_∞ as the span, *Brent's Lemma* relates the work and span as $T_P \leq (T_1 - T_\infty)/P + T_\infty$. This lemma reveals that the perfectly parallelizable work $T_1 - T_\infty$ is scalable with P, while the imperfectly parallelizable span takes time T_∞ regardless of P and is the limiting factor of the scalability of T_P.

A common factor affecting imperfectly parallelizable work and scalability is memory dependencies between parallel (or concurrent) tasks. For example, in a *race condition*, tasks contend for exclusive write access to a single memory location and must *synchronize* their reads to ensure correctness [95]. While some dependencies can be refactored into a perfectly parallelizable form, others still require synchronization (e.g., locks and mutexes) or hardware *atomic* primitives to prevent non-deterministic output. The key to enabling scalability in this scenario is to avoid high contention at any given memory location and prevent *blocking* of tasks, whereby tasks remains idle (sometimes deadlocked) until they can access a lock resource. To enable lock-free progress of work among tasks, fine-grained atomic primitives are commonly used to efficiently check and increment values at memory locations [52, 31]. For example, the *compare-and-swap* (CAS) primitive atomically compares the value read at a location to an expected value. If the values are equal, then a new value is set at the location; otherwise, the value doesn't change.

Moreover, programs that have a high ratio of memory accesses to arithmetic computations can incur significant *memory latency*, which is the number of clock or instruction cycles needed to complete a single memory access [121]. During this latency period, processors should perform a sufficient amount of parallel work to *hide* the latency and avoid being idle. Given the *bandwidth*, or instructions completed per cycle, of each processor, *Little's Law* specifies the number of parallel instructions needed to hide latency as the bandwidth multiplied by latency [88]. While emerging many-core and massively-threaded architectures provide more available parallelism and higher bandwidth rates, the memory latency rate remains stagnant due to physical limitations [95]. Thus, to exploit this greater throughput and *instruction-level parallelism* (ILP), a program should ideally be decomposed into fine-grained units of computation that perform parallelizable work (*fine-grained parallelism*).

Furthermore, the increase in available parallelism provided by emerging architectures also enables larger workloads and data to be processed in parallel [95, 54].

Gustafson [47] noted that as a problem size grows, the amount of parallel work increases much faster than the amount of serial work. Thus, a speedup can be achieved by decreasing the serial fraction of the total work. By explicitly parallelizing fine-grained computations that operate on this data, scalable *data-parallelism* can be attained, whereby a single instruction is performed over multiple data elements (SIMD) in parallel (e.g., via a vector instruction), as opposed to over a single scalar data values (SISD). This differs from *task-parallelism*, in which multiple tasks of a program conduct multiple instructions in parallel over the same data elements (MIMD) [121]. Task-parallelism only permits a constant speedup and induces *coarse-grained parallelism*, whereby all tasks work in parallel but an individual task could still be executing serial work. By performing inner fine-grained parallelism within outer course-grained parallel tasks, a *nested parallelism* is attained [12]. Many recursive and segmented problems (e.g., quicksort and closest pair) can often be refactored into nested-parallel versions [11]. Flynn [39] introduces SIMD, SISD, and MIMD as part of a taxonomy of computer instruction set architectures.

2.2 Data Parallel Primitives

The redesign of serial algorithms for scalable data-parallelism offers platform portability, as increases in processing units and data are accompanied by unrestricted increases in speedup. *Data-parallel primitives* (DPPs) provide a way to explicitly design and program an algorithm for this scalable, platform-portable data-parallelism. DPPs are highly-optimized *building blocks* that are combined together to compose a larger algorithm. The traditional design of an algorithm is thus refactored in terms of DPPs. By providing highly-optimized implementations of each DPP for each platform architecture, an algorithm composed of DPPs can be executed efficiently across multiple platforms. This use of DPPs eliminates the combinatorial (cross-product) programming issue of having to implement a different version of the algorithm for each different architecture.

The early work on DPPs was set forth by Blelloch [11], who proposed a *scan vector model* for parallel computing. In this model, a vector-RAM (V-RAM) machine architecture is composed of a vector memory and a parallel vector processor. The processor executes vector instructions, or *primitives*, that operate on one or more arbitrarily-long vectors of atomic data elements, which are stored in the vector memory. This is equivalent to having as many independent, parallel processors as there are data elements to be processed. Each primitive is classified as either *scan* or *segmented* (per-segment parallel instruction), and must possess a parallel, or *step*, time complexity of $O(\log n)$ and a serial, or *element*, time complexity of $O(n)$, in terms of n data elements; the element complexity is the time needed to simulate the primitive on a serial random access machine (RAM). Several canonical primitives are then introduced and used as building blocks to refactor a variety of data structures and algorithms into data-parallel forms.

The following are examples of DPPs that are commonly-used as building blocks to construct data-parallel algorithms:

- *Map*: Applies an operation on all elements of the input array, storing the result in an output array of the same size, at the same index;
- *Reduce*: Applies an aggregate binary operation (e.g., summation or maximum) on all elements of an input array, yielding a single output value. *ReduceByKey* is a variation that performs segmented *Reduce* on the input array based on unique key, yielding an output value for each key;
- *Gather*: Given an input array of values, reads values into an output array according to an array of indices;
- *Scan*: Calculates partial aggregates, or a prefix sum, for all values in an input array and stores them in an output array of the same size;

- *Scatter*: Writes each value of an input data array into an index in an output array, as specified in the array of indices;
- *Compact*: Applies a unary predicate (e.g., if an input element is greater than zero) on all values in an input array, filtering out all the values which do not satisfy the predicate. Only the remaining elements are copied into an output array of an equal or smaller size;
- *SortByKey*: conducts an in-place segmented *Sort* on the input array, with segments based on a key or unique data value in the input array;
- *Unique*: Ignores duplicate values which are adjacent to each other, copying only unique values from the input array to the output array of the same or lesser size; and
- *Zip*: Binds two arrays of the same size into an output array of pairs, with the first and second components of a pair equal to array values at a given index.

Several other DPPs exist, each meeting the required step and element complexities specified by Blelloch [11]. Cross-platform implementations of a wide variety of DPPs form the basis of several notable open-source libraries.

The Many-Core Visualization Toolkit (VTK-m) [109] is a platform-portable library that provides a growing set of DPPs and DPP-based algorithms [148]. With a single code base, back-end code generation and runtime support are provided for use on GPUs and CPUs. Currently, each GPU-based DPP is a modified variant from the Nvidia CUDA Thrust library [117], and each CPU-based DPP is adopted from the Intel Thread Building Blocks (TBB) library [53]. VTK-m also provides the flexibility to implement DPPs for new architectures or parallel programming languages as they become available. A single VTK-m algorithm code base can be executed on one of several devices at runtime. The choice of device is either specified at compile-time by the user, or automatically selected by VTK-m. VTK-m, Thrust, and TBB all employ a generic programming model that provides C++ Standard Template Library (STL)-like interfaces to DPPs and algorithms [125]. Templated arrays form the primitive data structures over which elements are parallelized and operated on by DPPs. Many of these array types provide additional functionality on top of underlying vector iterators that are inspired by those in the Boost Iterator Library [13].

The CUDA Data Parallel Primitives Library (CUDPP) [27] is a library of fundamental DPPs and algorithms written in Nvidia CUDA C [115] and designed for high-performance execution on CUDA-compatible GPUs. Each DPP and algorithm incorporated into the library is considered best-in-class and typically published in peer-reviewed literature (e.g., radix sort [100, 6], mergesort [128, 29], and cuckoo hashing [2, 3]). Thus, its data-parallel implementations are constantly updated to reflect the state-of-the-art.

2.3 General-Purpose GPU Computing

A *graphical processing unit* (GPU) is a special-purpose architecture that is designed specifically for high-throughput, data-parallel computations that possess a high arithmetic intensity—the ratio of arithmetic operations to memory operations [121]. Traditionally used and hard-wired for accelerating computer graphics and image processing calculations, modern GPUs contain many times more execution cores and available instruction-level parallelism (ILP) than a CPU of comparable size [115]. This inherent ILP is provided by a group of processors, each of which performs SIMD-like instructions over thousands of independent, parallel threads. These *stream processors* operate on sets of data, or *streams*, that require similar computation and exhibit the following characteristics [60]:

- *High Arithmetic Intensity*: High number of arithmetic instructions per memory instruction. The stream processing should be largely compute-bound as opposed to memory bandwidth-bound.
- *High Data-Parallelism*: At each time step, a single instruction can be applied to a large number of streams, and each stream is not dependent on the results of other streams.
- *High Locality of Reference*: As many streams as possible in a set should align their memory accesses to the same segment of memory, minimizing the number of memory transactions to service the streams.

General-purpose GPU (GPGPU) computing leverages the massively-parallel hardware capabilities of the GPU for solving general-purpose problems that are traditionally computed on the CPU (i.e., non-graphics-related calculations). These problems should feature large data sets that can be processed in parallel and satisfy the characteristics of stream processing outlined above. Accordingly, algorithms for solving these problems should be redesigned and optimized for the data-parallel GPU architecture, which has significantly different hardware features and performance goals than a modern CPU architecture [114].

Modern GPGPUs with dedicated memory are most-commonly packaged as discrete, programmable devices that can be added onto the motherboard of a compute system and programmed to configure and execute parallel functions [121]. The primary market leaders in the design of discrete GPGPUs are Nvidia and Advanced Micro Devices (AMD), with their GeForce and Radeon family of generational devices, respectively. Developed by Nvidia, the CUDA parallel programming library provides an interface to design algorithms for execution on an Nvidia GPU and configure hardware elements [115]. For the remainder of this survey, all references to a GPU will be with respect to a modern Nvidia CUDA-enabled GPU, as it is used prevalently in most of the GPU hashing studies.

The following subsections review important features of the GPU architecture and discuss criteria for optimal GPU performance.

2.3.1 SIMT Architecture. A GPU is designed specifically for *Single-Instruction, Multiple Threads* (SIMT) execution, which is a combination of SIMD and simultaneous multi-threading (SMT) execution that was introduced by Nvidia in 2006 as part of the Tesla micro-architecture [116]. On the host CPU, a program, or *kernel* function, is written in CUDA C and invoked for execution on the GPU. The kernel is executed *N* times in parallel by *N* different CUDA threads, which are dispatched as equally-sized *thread blocks*. The total number of threads is equal to the number of thread blocks times the number of threads per block, both of which are user-defined in the kernel. Thread blocks are required to be independent and can be scheduled in any order to be executed in parallel on one of several independent *streaming multi-processors* (SMs). The number of blocks is typically based on the number of data elements being processed by the kernel or the number of available SMs [115]. Since each SM has limited memory resources available for resident thread blocks, there is a limit to the number of threads per block—typically 1024 threads. Given these memory constraints, all SMs may be occupied at once and some thread blocks will be left inactive. As thread blocks terminate, a dedicated GPU scheduling unit launches new thread blocks onto the vacant SMs.

Each SM chip contains hundreds of ALU (arithmetic logic unit) and SFU (special function unit) compute cores and an interconnection network that provides *k*-way access to any of the *k* partitions of off-chip, high-bandwidth global DRAM memory. Memory requests first query a global L2 cache and then only proceed to global memory upon a cache miss. Additionally, a read-only texture memory space is provided to cache global memory data and enable fast loads. On-chip thread management and scheduling units pack each thread block on the SM into one or more smaller logical processing groups known as *warps*—typically 32 threads per warp; these warps compose a *cooperative thread array* (CTA). The thread manager ensures that each CTA is allocated sufficient shared memory space and per-thread registers (user-specified in kernel program). This on-chip shared memory is designed to be low-latency near the compute cores and can be programmed to serve as L1 cache or different ratios thereof (newer generations now include these as separate memory spaces) [48].

Finally, each time an instruction is issued, the SM instruction scheduler selects a warp that is ready to execute the next SIMT scalar (register-based) instruction, which is executed independently and in parallel by each active thread in the warp. In particular, the scheduler applies an active mask to the warp to ensure that only active threads issue the instruction; individual threads in a warp may be inactive due to independent branching in the program. A synchronization barrier detects when all threads (and warps) of a CTA have exited and then frees the warp resources and informs the scheduler that these warps are now ready to process new instructions, much like context switching on the CPU. Unlike a CPU, the SM does not perform any branch prediction or speculative execution (e.g., prefetching memory) among warp threads [121].

SIMT execution is similar to SIMD, but differs in that SIMT applies one instruction to multiple independent warp threads in parallel, instead of to multiple data lanes. In SIMT, scalar instructions control individual threads, whereas in SIMD, vector instructions control the entire set of data lanes. This detachment from the vector-based processing enables threads of a warp to conduct a form of SMT execution, where each thread behaves more like a heavier-weight CPU thread [121]. Each thread has its own set of registers, addressable memory requests, and control flow. Warp threads may take divergent paths to complete an instruction (e.g., via conditional statements) and contribute to starvation as faster-completing threads wait for the slower threads to finish.

The two-level GPU hierarchy of warps within SMs offers massive nested parallelism over data [121]. At the outer, SM level of granularity, coarse-grained parallelism is attained by distributing thread blocks onto independent, parallel SMs for execution. Then at the inner, warp level of granularity, fine-grained data and thread parallelism is achieved via the SIMT execution of an instruction among parallel warp threads, each of which operates on an individual data element. The massive data-parallelism and available compute cores are provided specifically for high-throughput, arithmetically-intense tasks with large amounts of data to be independently processed. If a high-latency memory load is made, then it is expected that the remaining warps and processors will simultaneously perform sufficient work to hide this latency; otherwise, hardware resources remain unused and yield a lower aggregate throughput [146]. The GPU design trades-off lower memory latency and larger cache sizes (such as on a CPU) for increased instruction throughput via the massive parallel multi-threading [121].

This architecture description is based on the Nvidia Maxwell micro-architecture, which was released in 2015 [48]. While certain quantities of components (e.g., SMs, compute cores, memory sizes, and thread block sizes) change with each new generational release of the Nvidia GPU, the general architectural design and execution model remain constant [115]. The CUDA C Programming Guide [115] and Nvidia PTX ISA documentation [116] contain further details on the GPU architecture, execution and memory models, and CUDA programming.

2.3.2 Optimal Performance Criteria. The following performance strategies are critical for maximizing utilization, memory throughput, and instruction throughput on the GPU [114].

Sufficient parallelism: Sufficient instruction-level and thread-level parallelism should be attained to fully hide arithmetic and memory latencies. According to Little's Law, the number of parallel instructions needed to hide a latency (number of cycles needed to perform an instruction) is roughly the latency times the throughput (number of instructions performed per cycle) [88]. During this latency period, threads that are dependent on the output data of other currently-executing threads in a warp (or thread block) are stalled. Thus, this latency can be hidden either by having these threads simultaneously perform additional, non-dependent SIMT instructions in parallel (instruction-level parallelism), or by increasing the number of concurrently running warps and warp threads (thread-level parallelism) [146].

Since each SM has limited memory resources for threads, the number of concurrent warps possible on an SM is a function of several configurable components: allocated shared memory, number of registers per thread, and number of threads per thread block [115]. Based on these parameters, the number of parallel thread blocks and warps on an SM can be calculated and used to compute the *occupancy*, or ratio of the number of active warps to the maximum number of warps. In terms of Little's Law, sufficient parallel work can be exploited with either a high occupancy or low occupancy, depending on the amount of work per thread. Based on the specific demands for SM resources, such as shared memory or register usage, by the kernel program, the number of available warps will vary accordingly. Higher occupancy, usually past 50 percent, does not always translate into improved performance [114]. For example, a lower occupancy kernel will have more registers available per thread than a higher occupancy kernel, allowing low-latency access to local variables and minimizing register spilling into high-latency local memory.

Memory coalescing: When a warp executes an instruction that accesses global memory, it *coalesces* the memory accesses of the threads within the warp into one or more memory transactions, or cache lines, depending on the size of the word accessed by each thread and the spatial coherency of the requested memory addresses. To minimize transactions and maximize memory throughput, threads within a warp should coherently access memory addresses that fit within the same cache line or transaction. Otherwise, memory divergence occurs and multiple lines of memory are fetched, each containing many unused words. In the worst case alignment, each of the 32 warp threads accesses successive memory addresses that are multiples of the cache line size, prompting 32 successive load transactions [114].

The shared memory available to each thread block can help coalesce or eliminate redundant accesses to global memory [121]. The threads of the block (and associated warp) can share their data and coordinate memory accesses to save significant global memory bandwidth. However, it also can act as a constraint on SM occupancy—particularly limiting the number of available registers per thread and warps—and is prone to *bank conflicts*, which occur when two or more threads in a warp access an address in the same bank, or partition, of shared memory [115]. Since an SM only contains one hardware bus to each bank, multiple requests to a bank must be serialized. Thus, optimal use of shared memory necessitates that warp threads arrange their accesses to different banks [115]. Finally, the read-only texture memory of an SM can be used by a warp to perform fast, non-coalesced lookups of cached global memory, usually in smaller transaction widths.

Control flow: Control flow instructions (e.g., if, switch, do, for, while) can significantly affect instruction throughput by causing threads of the same warp to diverge and follow different execution paths, or branches. Optimal control flow is realized when all the threads within a warp follow the same execution path [114]. This scenario enables SIMD-like processing, whereby all threads complete an instruction simultaneously in lock-step. During *branch divergence* in a warp, the different executions paths, or branches, must be serialized, increasing the total number of instructions executed for the warp. Additionally, the use of atomics and synchronization primitives can also require additional serialized instructions and thread starvation within a warp, particularly during high contention for updating a particular memory location [138].

2.4 Index-Based Searching

Let $U = \{i\}_{0 \leq i < u}$ be the universe for some arbitrary positive integer u. Then let $S \subset U$ be an unordered array of $n = |S|$ elements, or *keys*, belonging to U. The *search problem* seeks an answer to the *query*: "Is key k a member of S?" If $k \in S$, then we return its corresponding *value*, which is either k itself or a different value. A *data structure* is built or constructed over S to efficiently facilitate the searching operation. The data structure is implementation-specific and can be as simple as a sorted (ordered) variant of the original array, a hash table, or a tree-based partitioning of the elements.

A generalization of the search task is the *dictionary problem*, which seeks to both modify and query key-value pairs (k,v) in S. A canonical dictionary data structure supports $insert(k,v)$, $delete(k,v)$, $query(k)$, $range(k_1,k_2)$ (returns $\{k|k_1 \leq k \leq k_2\}$), and $count(k_1,k_2)$ (returns $|range(k_1,k_2)|$). To support these operations, the dictionary must be dynamic and accommodate incremental or batch updates after construction; this contrasts to a static data structure, which either does not support updates after a one-time build or must be rebuilt after each update. In multi-threaded environments, these structures must also provide concurrency and ensure correctness among mixed, parallel operations that may access the same elements simultaneously.

An extensive body of work has embarked on the redesign of data structures for construction and general computation on the GPU [119]. Within the context of searching, these *acceleration* structures include sorted arrays [2, 129, 3, 57, 79, 81, 7] and linked lists [152], hash tables (see section 2.5), spatial-partitioning trees (e.g., *k*-d trees [156, 62, 150], octrees [155, 62], bounding volume hierarchies (BVH) [76, 62], R-trees [91], and binary indexing trees [65, 131]), spatial-partitioning grids (e.g., uniform [71, 59, 43] and two-level [58]), skiplists [111], and queues (e.g., binary heap priority [51] and FIFO [20, 133]).

Due to significant architectural differences between the CPU and GPU, search structures cannot simply be "ported" from the CPU to the GPU and maintain optimal performance. On the CPU, these structures can be designed to fit within larger caches, perform recursion, and employ heavier-weight synchronization or hardware atomics. However, during queries, the occurrence of varying paths of pointers (pointer chasing) and dependencies between different phases or levels of the structure can limit the parallel throughput on the GPU. If the pointers are scattered randomly in memory, then memory accesses may be uncoalesced and induce additional global memory transactions (cache line loads). These attributes are particularly important to real-time, interactive applications, such as surface reconstruction and rendering, that make frequent updates and queries to the acceleration structure.

Spatial-partitioning or tree-based search structures are particularly vulnerable to these portable performance issues. For example, a canonical method of searching within a spatial domain involves explicitly computing a bounding box over the domain and then recursively subdividing it into smaller and smaller regions, or cells. Each cell contains a subset of elements, such as points coordinates or integers within an interval. This subdivision hierarchy can then be represented by a grid (e.g., uniform and two-level) or tree (e.g., *k*-d tree, octree, or bounding volume hierarchy) data structure that conducts a query operation by traversing a path through the hierarchy until the queried element is found. While these search structures are designed for fast, highly-parallel usage, they typically do not exhibit fast reconstruction rates due to complex spatial hierarchies, and may contain deep tree structures that are conducive to thread branch divergence during parallel query traversals.

For searching an unordered array of elements on the GPU, two canonical, spatially-linear data structures exist: the sorted array and the hash table. Both of these data structures are amenable to data-parallel design patterns [7] and, thus, avoid most of the portable-performance design challenges faced by tree-based search structures. In this dissertation, we focus exclusively on data-parallel sorting- and hashing-based search techniques. A background on both of these approaches is presented as follows.

2.4.1 Searching Via Sorting. Given a set of *n* unordered elements, a canonical searching approach is to first sort the elements in ascending order and then conduct a binary or *k*-nary search for the query element. This search requires a logarithmic number of comparisons in the worst-case, but is not as amenable to caching as consecutive comparisons are not spatially close in memory for large *n*. Moreover, on the GPU, an ordered query pattern by threads in a warp can enable memory coalescing during comparisons.

The current version of the CUDA Thrust library [117] provides fast and high-throughput data-parallel implementations of mergesort [128] and radix sort [100] for arrays of custom (e.g., comparator function) or numerical (i.e., integers and floats) data types, respectively. Similarly, the latest version of the CUDPP library [27] includes best-in-class data-parallel algorithms for mergesort [128, 29] and radix sort [100, 6], each of which are adapted from published work. Singh et al. [136] survey and compare the large body of recent GPU-based sorting techniques.

A few studies have investigated various factors that affect the performance of data-parallel sort methods within the context of searching [2, 3, 81]. Kaldewey and Blas introduce a GPU-based p-ary search that first uses p parallel threads to locate a query key within one of p larger segments of a sorted array, and then iteratively repeats the procedure over p smaller segments within the larger segment. This search achieves high memory throughput and is amenable to memory coalescing among the threads [57]. Moreover, the algorithm was also ported to the CPU to leverage the SIMD vector instructions in a fashion similar to the k-ary search introduced by Schlegel et al. [129]. However, the fixed vector width restricts the degree of parallelism and value of p, which is significantly higher on the GPU.

Inserting or deleting elements into a sorted array is generally not supported and requires inefficient approaches such as appending/removing new elements and re-sorting the larger/smaller array, or first sorting the batch of new insertions and then merging them into the existing sorted array. Ashkiani et al. [7] present these approaches and the resulting performance for a dynamic sorting-based dictionary data structure, along with setting forth the current challenges of designing dynamic data structures on the GPU.

2.4.2 Searching Via Hashing. Instead of maintaining elements in sorted order and performing a logarithmic number of lookups per query, *hash tables* compactly reorganize the elements such that only a constant number of direct, random-access lookups are needed on average [26]. More formally, given a universe U of possible keys and an unordered set $S \subseteq U$ of n keys (not necessarily distinct), a *hash function*, $h : U \mapsto H$, maps the keys from S to the range $H = \{j\}_{0 \leq j < m}$ for some arbitrary positive integer $m \geq n$. Defining a memory space over this range of size m specifies a hash table, into which keys are inserted and queried. Thus, the hash table is addressable by the hash function. During an insertion or query operation for a key q, the hash function computes an address $h(q) = r$ into H. If the location $H[r]$ is empty, then q is either inserted into $H[r]$ (for an insertion) or does not exist in H (for a query). If $H[r]$ contains the key q (for a query), then either q or an associated *value* of q is returned[1], indicating success. Otherwise, if multiple distinct keys $q' \neq q$ are hashed to the same address $h(q') = r$, then a situation known as a hash *collision* occurs. These collisions are typically resolved via *separate chaining* (i.e., employing linked lists to store multiple keys at a single address) or *open-addressing* (e.g., when an address is occupied, then store the key at the next empty address).

The occurrence of collisions deteriorates the query performance, as each of the collided keys must be iteratively inspected and compared against the query key. According to the birthday paradox, with a discrete uniform distribution hash function that outputs a value between 1 and 365 for any key, the probability that two random keys hash to the same address in a hash table of size 23 is 50 percent [140]. More generally, for n hash values and a table size of m, the probability $p(n,m)$ of a collision is

$$p(n,m) = \begin{cases} 1 - \prod_{k=1}^{n-1}\left(1 - \frac{k}{m}\right) & n \leq m \\ 1 & n > m \end{cases}$$

$$\approx 1 - \left(\frac{m-1}{m}\right)^{\frac{n(n-1)}{2}}.$$

Thus, for a large number of keys (n) and small hash table (m), hash collisions are inevitable.

[1] In practice, the values should be easily stored and accessible within an auxiliary array or via a custom arrangement within the hash table.

In order to minimize collisions, an initial approach is to use a good quality hash function that is both efficient to compute and distributes keys as evenly as possible throughout the hash table [26]. One such family of functions are randomly-generated, parameterized functions of the form $h(k) = (a \cdot k + b) \bmod p \bmod |H|$, where p is a large prime number and a and b are randomly-generated constants that bias h from outputting duplicate values [3]. However, the effectiveness of h also depends on the hash table size, $|H|$. If $|H|$ is too small, then not even the best of hash functions can avoid an increase in collisions. Given the table size, the *load factor* α of the table is defined as $\alpha = n/|H|$, or the percentage of occupied addresses in the hash table, which $|H|$ is typically larger than n. If new keys are inserted into the table and α reaches a maximum threshold, then typically the table is allocated to a larger size and all the keys are *rehashed* into the table.

To avoid collision resolution altogether, a *perfect* hash function can be constructed to hash keys into a hash table without collisions. Each key is mapped to a distinct address in the table. However, composing such a perfect hashing scheme is known to be difficult in general [78]. The probability of attaining a perfect hash for n keys in a large table of size m ($m \gg n$) is defined as

$$\begin{aligned} p(n,m) &= (1) \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdots \left(1 - \frac{n-1}{m}\right) \\ &\approx e^{\frac{-n^2}{2m}}, \end{aligned}$$

which is very small for a large n or small m.

In practice, a perfect hash function can be described as an imperfect hash function that is then iteratively corrected into a perfect form. One approach to doing this is to construct one or more auxiliary lookup tables that perturb the hash table addresses of collided keys into non-colliding addresses. These tables are typically significantly more compact than the hash table. Another foundational approach, introduced by Fredman et al. [41], is the use of a multi-level hash table that hashes keys into smaller and smaller buckets—each with a separate hash function—until each key is addressed to a bucket of its own, yielding a collision-free, perfect hash table with constant worst-case lookup time. Lefebvre and Hoppe [78] introduce a perfect spatial hashing (PSH) approach that is also the first GPU-specific perfect hashing approach. In PSH, a minimal perfect hash function and table are constructed over a sparse set of multi-dimensional spatial data, while simultaneously ensuring locality of reference and coherence among hashed points. Thus, on the GPU, spatially-close points are queried coherently, in parallel, by threads within the same warp. In order to maximize memory coalescing among these threads, points are also coherently accessed within the hash table, as opposed to via a random access pattern, which can necessitate multiple memory load instructions.

A hash table is *static* if it does not support modification after being constructed; that is, the table is only constructed to handle query operations. Thus, a static hash table also does not support mixed operations and the initial batch of insertions used to construct the table (bulk build) must be completed before the batch of query operations. A hash table that can be updated, or mutated, via insertion and deletion operations post-construction is considered *dynamic*. Denoting the query, insert, and delete operations as q, i, and d, respectively, the *operation distribution* $\Gamma = (q,i,d), q+i+d = 1$ specifies the percentage of each operation that are conducted concurrently in a hashing workload [5]. For example, $\Gamma = (0.7, 0.15, 0.15)$ represents a query-heavy workload that performs 70% queries and 30% updates. Additionally, the percentage q can be split into queried keys that exist in the hash table and those that do not. Often, queries for non-existent keys can present worst-case scenarios for many hash techniques, as a maximum number of searches are conducted until failure [5].

As general data structures, hash tables do not place any special emphasis on the key access patterns over time [25]. However, the patterns that appear in various real-world applications do possess observable structure. For example, geometric tasks may query spatially-close keys in a sequential or coherent pattern, and database tasks may query certain subsets of keys more frequently than others, whereby the hash table serves as a working set or most-recently-used (MRU) table for cache-like accesses [3, 126, 25]. Moreover, dynamic hash tables do not place special emphasis on the mixture Γ, or pattern, of query and update operations. However, execution time performance may be better or worse for some hashing techniques, depending on the specific Γ, such as query-heavy for key-value stores [153] or update-heavy for real-time, interactive spatial hash tables [78, 2, 44, 112].

Finally, hash tables offer compact storage for sparse spatial data that contains repeated elements or empty elements that don't need to be computed. For example, instead of storing an entire, mostly-empty voxelized 3D grid, the non-empty voxels can be hashed into a dense hash table [78]. Then, every voxel can be queried to determine whether it should be rendered or not, returning a negative result for the empty voxels. Furthermore, a hash table does not have to be one-dimensional. Instead, the data structure can consist of multiple hash tables or *bucketed* partitions that are each addressed by a different hash function.

While collision resolution is straightforward to implement in a serial CPU setting, it does not easily translate to a parallel setting, particularly on massively-threaded, data-parallel GPU architectures. GPU-based hashing presents several notable challenges:

– Hashing is a memory-bound problem that is not as amenable to the compute-bound and limited-caching design of the GPU, which hides memory latencies via a large arithmetic throughput.

- The random-access nature of hashing can lead to disparate writes and reads by parallel-cooperating threads on the GPU, which performs best when memory accesses are coalesced or spatially coherent.
- The limited memory available on a GPU puts restrictions on the maximum hash table size and number of tables that can reside on device.
- Collision resolution schemes handle varying numbers of keys that are hashed and chained to the same address (separate chaining), or varying numbers of attempts to place a new, collided key into an empty table location (open-addressing). This variance causes some insert and query operations to require more work than others. On a GPU, threads work in groups to execute the same operation on keys in a data-parallel fashion. Thus, a performance bottleneck arises when faster, non-colliding threads wait for slower, colliding threads to finish. Moreover, some threads may insert colliding keys that are unable to find an empty table location, leading to failure during construction of the table.

Searching via the construction and usage of a hash table on the GPU has recently received a breadth of new research, with a variety of different parallel designs and applications, ranging from collision detection to surface rendering to nearest neighbor approximation. The next section covers these GPU-based parallel hashing techniques.

2.5 Data-Parallel Hashing Techniques

This section reviews existing parallel hashing techniques that employ one of the following two forms of collision-handling: open-addressing and separate chaining. Both forms of collision-handling motivate the need for data-parallel hashing approaches for optimal performance, particularly on GPU devices. Moreover, open-addressing is the foundation upon which this dissertation work and contributed techniques are based. A complete survey of data-parallel hashing techniques and additional collision-handling methods can be found in my area exam paper [80]; only a subset of this survey is presented in this section.

2.5.1 Open-addressing Probing.

In *open-addressing*, a key is inserted into the hash table by *probing*, or searching, through alternate table locations—the *probe sequence*—until a location is found to place the element [26]. The determination of where to place the element varies by probing scheme: some schemes probe for the first unused location (*empty slot*), whereas others *evict* the currently-residing key at the probe location (i.e., a collision) and swap in the new key. Each probe location is specified by a hash function unique to the probing scheme. Thus, some probe sequences may be more compact or greater in length than others, depending on the probing method. For a query operation, the locations of the probe sequence are computed and followed to search for the queried key in the table.

Each probing method trades-off different measures of performance, particularly with respect to GPU-based hashing. A critical influence on performance is the load factor, which is the percentage of occupied locations in the hash table (subsection 2.4.2). As the load factor increases towards 100 percent, the number of probes needed to insert or query a key increases greatly. Once the table becomes full, probing sequences may continue indefinitely, unless bounded, and lead to insertion failure and possibly a hashing *restart*, whereby the hash table is reconstructed with different hash functions and parameters. Moreover, for threads within a warp on the GPU, variability in the number of probes per thread can induce branch divergence and inefficient SIMD parallelism, as all the threads will need to wait for the worst-case number of probes to execute the next instruction.

The following subsections review research on open-addressing probing for hashing, distinguishing each study by its general probing scheme: linear probing, cuckoo hashing, double hashing, multi-level or bucketized probing, and robin hood hashing.

2.5.1.1 Linear Probing-based Hashing. *Linear probing* is the most basic method of open-addressing. In this method, a key k first hashes to location $h(k)$ in the hash table. Then, if the location is already occupied, k linearly searches locations $h(k)+1, h(k)+2, \ldots$ etc. until an empty slot (insertion) or k itself (query) is found. If $h(k)$ is empty, then k is inserted immediately, without probing; otherwise, a worst-case $O(n)$ probes will need to be made to locate k or an empty slot, where n is the size of the hash table. While simple in design, linear probing suffers from *primary clustering*, whereby a cluster, or contiguous block, of locations following $h(k)$ are occupied by keys, reducing nearby empty slots. This occurs because colliding keys at $h(k)$ each successively probe for the next available empty slot after $h(k)$ and insert themselves into it. An improved variant of linear probing is *quadratic probing*, which replaces the linear probe sequence starting at $h(k)$ with successive values of an arbitrary quadratic polynomial: $h(k) + 1^2, h(k) + 2^2, \ldots$ etc. This avoids primary clustering, but also introduces a *secondary clustering* effect as a result. For a more than half-full table, both of these probing methods can incur a long probe sequence to find an empty slot, possibly resulting in failure during an insert.

Bordawekar [14] develops an open-addressing approach based on multi-level bounded linear probing, where the hash table has multiple levels to reduce the number of lookups during linear probing. In the first level hash table, each key hashes to a location $h_1(k)$ and then looks for an empty location, via linear probing, within a bounded probe region $P_1 = [h_1(k), h_1(k) + (j-1)]$, where j is the size of the region. If an empty location is not found, then the key must be inserted into the second-level hash table, which is accomplished by hashing to location $h_2(k)$ and linear probing within another, yet larger, probe region P_2. This procedure continues for each level, until an empty location is found. In this work, only 2-level and 3-level hash tables are considered; thus, a thread must perform bounded probing on a key for at most three rounds, before declaring failure. To query a key, a thread completes the same hashing and probing procedure. In a data-parallel fashion, each thread within a warp is assigned a key from the bounded probe region and compares this key with the query key, using warp-level voting to communicate success or failure. This continues across warps, for each hash table level.

The initial design goal of this multi-level approach was to bound and reduce the average number of probes per insertion and query, while enabling memory coalescing and cache line coherency among threads (or lanes) within a warp. By using a small, constant number of hash tables and functions, the load factor could be increased beyond the 70 percent of Alcantara et al.'s cuckoo hashing (subsection 2.5.1.2), without sacrificing performance. However, experimental results reveal that this approach, with both two and three levels (and hash functions), does not perform as fast as cuckoo hashing for the largest batches of key-value pairs (hundreds of millions); for smaller batches, the multi-level approaches are the best performers. This finding is particularly noticeable for querying the keys, suggesting that improved probing and memory coalescing are likely not achieved. Additional details are needed to ascertain whether the ordering of the keys—spatial or random—affect this multi-level approach, or specific reasons why the expected warp-level memory coalescing is not being realized.

2.5.1.2 Cuckoo-based Hashing. In *cuckoo hashing*, each key is assigned two locations in the hash table, as specified by primary and secondary hash functions [120]. When inserting a new key, its first location is probed with the primary function and the contents of the location are inspected. If the slot is empty, then the key is inserted and the probe sequence ends. Otherwise, a collided key already occupies the slot and the cuckoo eviction procedure begins. First, the occupying key is evicted and hashed to the location specified by its secondary function, where its contents are probed as before.

This *eviction chain* continues until either the evicted key is successfully inserted or a maximum chain length is reached. If the eviction is successful, then the new key is finally inserted at its primary location (first probe). Numerous follow-up studies to this canonical approach have introduced cuckoo hashing approaches with more than two hash functions (probes) per key, a separate hash table for each hash function, and other optimizations for concurrent, mixed operations (e.g., simultaneous inserts and queries) on the GPU. These studies are surveyed in this subsection.

Alcantara et al. [2] introduce a data-parallel, dynamic hashing technique based on perfect hashing and cuckoo hashing that supports both hash table construction and querying at real-time, interactive rates. The querying performance of this technique is compared against that of the perfect hashing technique of Lefebvre and Hoppe [78] and a standard data-parallel sort plus binary search approach. In this work, a two-phase hashing routine is conducted to insert and query elements, with the goal of maximizing shared-memory usage during cuckoo hashing.

First, elements are hashed into bucket regions within the hash table, following the perfect hashing approach of Fredman et al. [41]. The maximum occupancy of each bucket is the number of threads in a thread block (e.g., 512), such that the entire bucket can fit within shared memory. The hash function aims to coherently map elements into buckets such that:

- Each bucket, on average, maintains a load factor of 80%, and
- Spatially-nearby elements are located within the same bucket, enabling coalescing of memory among threads during queries.

If more than 512 elements hash to a given bucket, then a new hash function is generated and this phase is repeated. Then, within each bucket, cuckoo hashing is performed to insert or query an element, using $i = 3$ different hash functions h_i (i.e., the multiple choices), each corresponding to a sub-table T_i. During construction, each element simultaneously hashes to its location h_1 in T_1, in a winner-takes-all fashion. If multiple threads hash to the same location, then the winning thread (i.e., the last thread to write) remains and the other threads proceed to hash into location h_2 in T_2. This continues for T_3, after which any remaining unplaced elements cycle back to the beginning and hash into h_1 in T_1 again. At this point, if a collision occurs at h_1, then the current residing element is evicted and added to the batch of unplaced elements. This cuckoo hashing procedure continues until all elements are successfully placed into a sub-table T_i or a maximum number of cycles have occurred.

An observation of this construction routine is that restarts can occur during both phases if either a bucket overflows or the cuckoo hashing reaches the maximum number of cycles within a bucket. While this reconstruction may be viewed as a disadvantage of probing techniques in general, the authors maintain that the occurrence of these restarts are reasonable in practice and fast to compute on massively-parallel GPU architectures. Moreover, this technique makes extensive use of thread atomics to increment and check values in both global and shared memory. While only a fixed number of atomic operations are made each phase, they are still serialized and must handle varying levels of thread contention, both of which are known to degrade performance.

After construction, a query operation is performed by hashing once into a bucket, and then making at most $d = 3$ hashing probes to locate the element within one of the sub-tables T_i of the bucket. Insertions and queries are all conducted in a data-parallel, SIMD fashion. Since each thread warp assigned to a bucket has its own dedicated block of shared memory, the probing and shuffling of elements in the cuckoo hashing can be performed faster locally, as opposed to accessing global memory.

Experimental results for this technique reveal the following:

- For querying elements (voxels in a 3D grid) in a randomized order, this hashing approach outperforms the perfect hashing approach of Lefebvre et al. [78] and the data-parallel binary search of radix-sorted elements of Satish et al. [128], particularly above 5 million elements. After this point, the binary searches used in both methods do not scale and become time-prohibitive.
- For querying in a sequential order, the data-parallel binary search demonstrates better performance than this hashing technique, due to more favorable thread branch divergence and memory coalescing among the sorted elements.
- Constructing the hash table of elements in this approach is comparably-fast to radix sorting the elements, with noticeable slowdowns due to more non-coalesced write operations. Moreover, for large numbers of insertions, both approaches are magnitudes faster than constructing the perfect spatial hash table of [78], which is initially built on the CPU, rather than the GPU (onto which the table is copied for subsequent querying).

Alcantara et al. [3] improved upon their original work [2] by introducing a generalized parallel variant of cuckoo hashing that can vary in the number of hash functions, hash table size, and maximum length of a probe-and-eviction sequence. In [2], the authors hypothesized that parallel cuckoo hashing within GPU global memory would encounter performance bottlenecks due to the shuffling of elements each iteration and use of global synchronization primitives; thus, shared memory was used extensively in the two-level hashing scheme. However, in this follow-up work, a single-level hash table is constructed entirely in global memory and addressed directly with the cuckoo hash functions, without the first-level bucket hash. The cuckoo hashing dynamics remain the same, except that the probing and evicting of elements occurs over the entire global memory hash table, as opposed to the shared-memory buckets of the two-level approach.

To construct a hash table of N elements, approximately N threads will operate in SIMD parallel fashion to place their elements into empty slots in the global table. A given thread block will not complete until all of its threads have successfully placed their elements; a smaller block size helps minimize the completion time, as the block will likely contain fewer threads with long eviction chains.

The construction (insertion) and query performance of the single-level hash approach is compared against that of Merril's radix sort plus binary search [100] and the authors' previous two-level cuckoo hashing approach. Experimental results reveal the following:

- *Insertions*. For large numbers of insertions, radix sort [100] becomes increasingly faster than both hashing methods, with a much higher throughput of insertions-per-second. For the same size hash table, the single-level hash table is constructed significantly faster than the two-level table, due to shorter eviction chains on average, over all insertion input sizes (the two-level table uses a fixed $1.42N$ space, while the single-level table is variable-sized). Radix sort achieves an upper bound of 775 million memory accesses (read and write) per second, while the single-level hashing only attains 670 million accesses per second. This higher throughput by radix sort is due to its more-localized memory access patterns that enable excellent memory coalescing among threads sharing a memory-bound instruction (up to 70% of the theoretical maximum pin bandwidth on the tested Nvidia GTX 480 GPU, versus 6% of the single-level hashing).
- *Queries: Binary Search vs. Hashing*. For random, unordered queries, binary search probing of the radix sorted elements is much slower than cuckoo hash probing of the elements. This arises from uncoalesced global memory reads and branch divergence for many of the threads, which use the maximum $O(logN)$ probes. Both

cuckoo hashing approaches lookup elements in a worst-case constant number of probes and, thus, perform significantly better than binary searching, despite these probes being largely uncoalesced.

- *Queries: Two-Level vs. Single-Level.* When all queried elements exist in the hash table, the single-level cuckoo hashing makes a smaller average number of probes per query than the two-level approach, leading to faster completion times. However, when a large percentage of the queried elements do not exist in the hash table, the two-level hashing needs fewer worst-case probes before declaring the element as not found. This is because the single-level hashing uses four hash functions, or probes, to lookup an element, whereas the two-level hashing only uses three functions. By setting the number of hash functions to three in the single-level hashing, the authors observe comparable querying performance between the two approaches.

A notable performance observation in this work is that only randomized queries are considered. The authors indicate, as a limitation of their work, that if the elements to be queried are instead ordered (sorted), then binary searching the radix-sorted elements should yield improved thread branch divergence and memory coalescing. This work has since been incorporated into the CUDPP library [27] as a best-in-class GPU hash table data structure.

2.5.1.3 Double Hashing. *Double hashing* first hashes a key k to location $h(k)$ in the hash table and then, if the location is already occupied, computes another independent hash $h'(k)$ that defines the step size to the next probing location [26]. Thus, the second probe location is $h(k) + i \cdot h'(k)$, where i is the current i-th probe in the probe sequence. This hashing and probing continues until an empty slot (insertion) or k itself (query) is found. Similar to linear and quadratic probing, if $h(k)$ is empty, then k is inserted immediately, without probing. The choice of the second hash function has a large impact on performance, as it dictates the locality of consecutive probes and, thus, the opportunity for memory coalescing among threads on the GPU.

Khorasani et al. [64] introduce a *stadium hashing* (*Stash*) technique that builds and stores the hash table in out-of-core host memory, and resolves insert collisions via double hashing until an empty slot is found. In GPU global memory, a compact auxiliary *ticket-board* data structure is maintained to grant read and write accesses to the hash table. For each hash table location, the ticket board maintains a *ticket*, which consists of a single *availability bit* and small number of optional *info bits*. The availability bit indicates whether the location is empty (set to 1) or occupied by a key (set to 0), while the info bits are a small generated signature of the key to help identify the key prior to accessing its value. Within individual thread warps, a shared-memory, *collaborative lanes* (*clStash*) load-balancing scheme is used to ensure that, during insertions, all threads are kept busy, preventing starvation by unsuccessful threads.

Stadium hashing is meant to address three limitations of previous GPU parallel hashing techniques, specifically in regard to the cuckoo hashing approach of Alcantara et al. [3]:

1. Support for concurrent, mixed insert and query operations. Without proper synchronization, cuckoo hashing encounters a race condition whereby a query probe fails at a location because a concurrently-inserted key hashes to the location and evicts the queried key, yielding a false negative lookup. Stadium hashing avoids this issue by using eviction-free double hashing and granting atomic access to a location via a ticket board ticket with the availability bit set to 1.
2. Reduce host-to-device memory requests for large hash table sizes. In cuckoo hashing, CAS atomics are used to retrieve the content of a memory location,

compare the content with a given value, and swap in a new value, if necessary. When a hash table is stored in host memory, the large number of parallel retrieval requests from thousands of GPU threads will turn the hashing into a PCIe bandwidth-bound problem and degrade performance. Stadium hashing uses the GPU ticket board data structure to minimize retrieval requests to the host memory hash table.

3. Efficient use of SIMD hardware. During a cuckoo hashing operation, a thread failing to insert or query a key can cause starvation among the other threads in the thread warp, as they all perform the same instruction in lock-step. Thus, if the other threads complete their operation early, then they will remain idle and contribute to work imbalance. Stadium hashing uses the clStash load-balancing routine to maintain a warp-wide, shared memory store of pending operations that early-completing threads can claim to remain busy.

For an out-of-core hash table, the ticket-board with larger ticket sizes (more info bits per key) helps improve the number of operations per second by reducing the number of expensive host memory accesses over the PCIe bus. This improvement is especially significant for unnecessary queries of elements which do not actually reside in the host hash table. In this case, the PCIe bandwidth from GPU to CPU memory is the primary performance bottleneck. However, when the hash table resides in GPU memory, the underutilization of SIMD thread warps becomes the primary bottleneck on performance for low load factors (fewer collisions). The efficiency of warps is shown to improve by using the collaborative lanes clStash scheme in combination with the Stash hashing.

Regarding the experiments and findings in this work, the cuckoo hashing approach of [3] is specifically designed for hash table construction and querying within GPU memory. Thus, the use of this technique in out-of-core memory should not necessarily be expected to perform optimally, and should be kept in mind when comparing against the out-of-core performance of stadium hashing.

2.5.1.4 Robin Hood-based Hashing. *Robin Hood* hashing [21] is an open-addressing technique that resolves hash collisions based on the *age* of the collided keys. The age of a key is the length of the probe sequence, $h^1(k), h^2(k), \ldots$, needed to insert the key into an empty slot in the hash table. During a collision at a probe location, the key with the youngest age is evicted and the older key inserted into that location. The evicted key is then robin hood hashed again until it is placed in a new empty location, incrementing its age along the new probe sequence. The idea of this approach is to prevent long probe sequences by favoring keys that are difficult to place. Even in a full table with high load factor, this eviction policy guarantees an expected maximum age of $\Theta(\log n)$ for an insert or query key. However, the worst-case maximum age M may still be higher and worse than the maximum probe sequence length of cuckoo hashing, prompting a table reconstruction in some cases. These maximum M probes will be required during queries for empty keys (those which do not exist in the hash table), unless they are detected and rejected early.

Garcia et al. [44] introduce a data-parallel robin hood hashing scheme that maintains coherency among thread memory accesses in the hash table. Neighboring threads in a warp are assigned neighboring keys to insert or query from a spatial domain (e.g., pixels in an image or voxels in a volume). By specifying a coherent hash function, both keys will be hashed near each other in the hash table and the threads can access memory in a coalesced fashion, i.e., as part of the same memory transaction. Thus, the sequence of probes for groups of threads will likely also be conducted in a coherent manner, as nearby keys of a young age are evicted and replaced by nearby keys of an older age.

The insertion and query performance of this techniqie is evaluated on both randomly- and spatially-ordered key sets from a 2D image and 3D volume. For all load factor settings, the existence of coherence in the keys and probe sequences results in significant improvements in construction and querying performance (millions of keys processed per second), as compared to the use of randomly-ordered keys. Moreover, coherent hashing achieves greater throughput than the cuckoo hashing of Alcantara et al. [3], which employs four hash functions for a maximum probe sequence of length four. For load factors above 0.7, coherent hashing maintains superior performance without failure (hash table reconstruction) during insertions, whereas the cuckoo hashing exhibits an increase in failures.

In absence of coherence in the access patterns, coherent hashing brings little to no benefit compared to the random access robin hood and cuckoo hashing. Thus, this approach is of particular use for applications with spatial coherence in the data. In one of the spatially-coherent experiments, the task is to insert a sparse subset of pixels from an image (e.g., all the non-white pixels) into the hash table, and then query every pixel to reconstruct the image. Since only non-white pixels are hashed, there will be empty queries for the white pixel keys. Spatial and coherent ordering of keys is attained by applying either a linear row-major, Morton, or bit-reversal function to the spatial location of elements; a non-coherent, randomized order is attained by shuffling the keys.

Coherent hashing has some notable design characteristics that can affect GPU performance. First, upon completing an insert or query operation, a thread sits idle until all threads in its warp have finished as well. The number of threads per warp (192 in this work) and amount of branch divergence due to incoherent ordering of keys are primary factors affecting the warp load balancing. Second, while inserting a key, the hash table is reconstructed if the age, or probe sequence length, of the key exceeds a threshold maximum (15 in this work). Moreover, the hash table is not fully dynamic and is designed to process queries after an initial build phase. Thus, if new keys are inserted after the build phase, then the table is reconstructed entirely, with a larger table size or load factor if necessary.

2.5.2 Separate Chaining. *Separate chaining* is a classic collision resolution technique that uses a linked list or node-based data structure to store multiple collided keys at a single hash table entry. Each hash table entry contains a pointer, or memory address, to a head node of a linked list, or *chain*. Each node in the linked list consists of a key, associated value (optional), and a pointer to the next node in the list, if any. If a single key hashes to an entry, then the linked list consists of a single node with a null pointer to the non-existent next node. Otherwise, if multiple keys collide and hash to the same location, then the linked list forms a *chain* of these keys, each represented by a separate node in the list. During a query operation, a key hashes to an entry in the table and then iterates through each of the nodes of the chain referenced at the entry, searching for a matching key. This iterative search is similar in nature to open-addressing linear probing (refer to subsection 2.5.1.1), where a key hashes to an initial table entry and then probes each subsequent entry until a matching key is found. Both techniques can result in degenerate, worst-case queries that require a non-constant number of probes.

Unlike separate chaining, linear probing is prone to primary clustering of collided keys and performs *lazy deletion* of keys that renders unoccupied table entries heavily fragmented and may require re-hashing or compaction. However, linear probing is more amenable to caching, as probes are conducted within a contiguous block of memory, instead of over the scattered memory of linked list nodes.

Moreover, separate chaining is a form of *open hashing* in which keys and values are stored in allocated linked lists outside of the hash table and then referenced by head node pointers that are stored inside the table. Contrarily, open addressing collision resolution follows *closed hashing*, whereby each hashed key (and value) is inserted directly into the hash table.

In the context of parallel hashing, separate chaining must synchronize collisions during key insertions to ensure that the linked list data structures are properly allocated and constructed. Moreover, a dynamic memory allocation scheme must ensure that concurrent threads conducting insert operations properly synchronize their requests for new available blocks of memory. Similar design challenges exist for the deletion of keys, and the simultaneous execution of queries by threads must avoid reader-writer race conditions that result in faulty memory accesses to incorrect or deallocated nodes (keys).

A large body of research has investigated *concurrent* hash tables for separate chaining on multi- and many-core CPU systems [46, 101, 122, 134, 45]. Each of these hash tables is designed to support dynamic[2] updates and resizing with lock-based methods (e.g., mutexes or spin-locks) or lock-free (non-blocking) hardware atomics, such as compare-and-swap (CAS). Since the majority of these hash tables are linked list-based data structures, they are not designed for scalable, high-performance on massively-parallel GPU architectures. In particular, when ported to the GPU, the performance of these approaches may degrade due to several reasons:

– Lock-based methods induce substantial thread contention during blocking operations for shared resources and are not scalable with increasing numbers of concurrent threads. This contention creates starvation for blocked threads and warp under-utilization, since each thread must wait for its other warp threads to finish acquiring and releasing the lock. Moreover, lock-free hardware atomic

[2] Some implementations are aware of future insertions at compile-time and preallocate sufficiently-large additional memory. These hash tables are *semi-dynamic* since they do not dynamically allocate new memory at runtime for unknown insertions.

primitives prevent deadlock, but still neglect the sensitivity of GPUs to global memory accesses and thread branch divergence.

- Lack of coalescing among memory accesses due to the scattering of linked list node pointers in memory and random addressing of keys by threads within the same warp, which can lead to additional global memory transactions (cache line loads).
- Dynamic memory management and pointer chasing required for the linked lists on the GPU is challenging for traditional CPU-based synchronization schemes, due to the massive thread parallelism. This performance challenge is similarly observed in pointer-heavy spatial search tree structures that are ported to the GPU.

The following studies introduce GPU-based separate chaining hashing approaches that attempt to address these performance challenges.

Moazeni and Sarrafzadeh [105] and Misra and Chaudhuri [103] deploy some of the earliest lock-free, separate chaining-based hash tables on a GPU architecture. Using CUDA atomic CAS operations (atomicCAS and atomicInc), both approaches support batches of concurrent query and insert operations, with only [103] also supporting delete operations. [105] achieves a significant execution time speedup for queries over counterpart lock-based and OpenMP-based CPU implementations. However, the lock-free table only attains significantly higher throughput (operations per second) than the OpenMP implementation for query-heavy batches (80% queries and 20% inserts). Additionally, this work does not focus on larger, scalable batch sizes and provides little analysis regarding thread- or warp-level performance. [103] demonstrates that a GPU lock-free hash table leverages a much higher degree of concurrency and throughput than a CPU implementation for both query-heavy and update-heavy workload batches. This performance increase is largely due to spreading the thread contention and atomic comparisons over multiple different hash locations, as threads work in SIMT data-parallel fashion to conduct mixed operations at random locations.

Stuart and Owens [138] and newer versions of the Nvidia CUDA C library [115] both introduce new efficient synchronization and atomic primitives (e.g., warp-level and share memory atomics) for CUDA-compatible GPUs. These primitives likely satisfy the inefficiencies of atomics for pointer-based data structures cited in Misra and Chaudhuri [103].

Ashkiani et al. [5] propose a dynamic *slab hash* table on the GPU that is built upon an array of linked-lists, or *slab lists*, each of which represent a chain of one or more *slabs*, or memory units, that store collided keys. Each slab of memory is roughly the size of a warp memory transaction width (128 bytes), or the number of warp threads (32) times the size of a key (4 bytes). Thus, each warp is aligned to perform operations over the keys stored in a single slab. As part of a novel *work-cooperative work sharing* (WCWS) strategy, each warp maintains a *work queue* that stores all the arbitrary operations assigned to the different threads in the warp. In a round-robin fashion, each batch of the same operation type in the queue is fully and cooperatively executed by the threads. For a given operation type, all threads perform a warp-wide ballot instruction to denote the *active* threads that were assigned this operation. For each active thread, the entire warp cooperates to execute the active thread's operation and its corresponding key. If the operation is a query for a key q, then the warp hashes q to a slab list $b_i = h(q)$ in the slab hash table H. The first warp-sized slab, b_{i0}, of the slab list at $H[b_i]$ is loaded from global memory via a single memory transaction. This slab memory unit contains the same number of keys as threads in the warp. So, each warp thread then compares its corresponding key k with the query key q. If any thread has $k = q$, then a successful result is returned. Otherwise, the warp follows a *next* pointer stored in b_{i0} to load the next connected slab b_{i1}, in which q is cooperatively searched again. This search ends when either q is found or the last slab in b_i has been searched.

An insert operation proceeds similarly, except the threads search for an empty slab spot into which a new key can be atomically inserted. If no empty spot is found in any of the slabs of the slab list, then a new slab must be atomically and dynamically allocated (since other warps may also be trying to allocate). This allocation is efficiently performed via a novel warp-synchronous *SlabAlloc* allocator (see [5] for further details).

This warp-cooperative approach differs from previous GPU separate chaining in which the threads of a warp execute a SIMT query or update operation for different keys, each of which likely require a random, uncoalesced memory access. WCWS ensures memory coalescing for each operation by perfectly aligning the threads of a warp with the keys of a slab, both of the same size. Thus, the same block of cache-aligned global memory is loaded in a single transaction for every operation by the warp, exposing increased throughput (millions of operations per second). Moreover, while being inserted, keys are always stored at contiguous addresses within a slab memory unit. This contrasts with traditional linked list storage in which keys are inserted as new nodes at random memory locations.

The performance of the dynamic slab hash table is compared to the static cuckoo hash table of Alcantara et al. [2]—which must be rebuilt upon updates—and the semi-dynamic lock-free hash table of Misra and Chaudhuri [103]. For static bulk builds, cuckoo hashing consistently achieves a higher throughput of insertions per second, while slab hashing achieves higher query throughput only when the average number of slabs per slab list is less than 1 (i.e., approximately a single "node" list). Over all configurations, cuckoo hashing attains the better query throughput. In the best case scenario, it only makes a single atomic comparison for an insertion and a single random memory access for a query; contrarily, in the best case, slab hashing requires both a memory access (to load a slab) and an atomic comparison for an insertion. For dynamic updates, slab hashing significantly outperforms cuckoo hashing, in terms of execution time, as the number of inserted keys increases. This is due to the rebuilding of the static cuckoo hash table each time a new batch is inserted. Additionally, slab hashing significantly outperforms lock-free hashing across different distributions of mixture operations and increasing numbers of slab lists (i.e., the size of the hash table).

2.6 Conclusion

This chapter provides a comprehensive background and survey on the major topics of this dissertation: data-parallelism, index-based searching, and index-based search techniques for diverse many-core systems. This background material serves as a primer for the upcoming chapters, each of which help inform the dissertation question regarding the best index-based search techniques for visualization and analysis algorithms on diverse many-core systems.

CHAPTER III

DATA-PARALLEL SEARCHING FOR DUPLICATE ELEMENTS

Searching for duplicate elements arises in multiple scientific visualization and data analysis contexts, such as identifying the overlapping interior regions of a mesh solid or detecting the integers that occur more than once in an array. There are two canonical index-based approaches for identifying these duplicate elements:

1. Sort all elements and search for identical neighbors.
2. Hash all elements and search for collisions.

In this chapter, we introduce a new hashing-based technique and propose a new method that uses an existing sorting-based technique. Both of these contributions are designed entirely in terms of DPP for platform-portable data parallelism.

The work of this chapter is adopted primarily from a collaborative paper that was published at a conference and composed by myself, Roba Binyahib, Robert Maynard, and Hank Childs [79]. As lead author, I developed and implemented the contributed techniques, and wrote the majority of the text contained in this chapter. Hank Childs provided valuable guidance towards the motivation and application of this work. Robert Maynard contributed significant feedback and insight into the data-parallel design patterns used in our techniques. Roba Binyahib helped conduct the experimentation of our techniques and contributed to the writing of the experimental overview text, which is contained in Chapter V.

3.1 Sorting-Based Algorithm

This approach uses sorting to identify duplicate elements. First, elements are placed in an array and sorted. Then, the array can be searched for duplicates in consecutive entries. The sorting operation requires a way of comparing two elements (i.e., a “less-than" test); e.g., for integer vectors, we order the integers within a vector, and then compare the integers with the lowest index, proceeding to subsequent indices in cases of ties.

Algorithm 1: Pseudocode for the sorting-based technique for identifying duplicate elements of a comparable type T. N is the total number of elements, U is the number of unique elements, and L is the number of elements with a frequency count of 1.

```
1  template <typename T>
2  /*Input*/
3  Array: T elements[$N$]
4  /*Output*/
5  Array: T nonDuplicates[$L$]
6  Array: T uniqueElements[$L \leq U \leq N$]
7  Array: int elementCounts[$L \leq U \leq N$]
8  /*Local Object*/
9  ArrayConstant: int ones[$N$]
10 elements←Sort(elements);
11 if count duplicates then
12     (uniqueElements, elementCounts)← ReduceByKey(elements, ones);
13     if remove elements with count > 1 then
14         nonDuplicates←CopyIf(uniqueElements, elementCounts);
15         return nonDuplicates;
16     end
17 else
18     uniqueElements← Unique(elements);
19 end
20 return (uniqueElements, elementCounts);
```

The pseudocode for our sorting-based algorithm is presented in Algorithm 1. First, the array of elements is data-parallel sorted in ascending order (line 10) so that all duplicate elements will be contiguous to each other within the array. If a frequency count of the number of duplicates per unique element is needed, then a ReduceByKey DPP (line 12) can be applied to the sorted array of elements. All elements with a count greater than 1 are considered duplicate. If only non-duplicate elements need to be returned from the algorithm, then a CopyIf, or compaction, DPP can be applied to the array of unique elements (line 14); the resulting array of non-duplicates is returned as output (line 15). Finally, if frequency counts of duplicates are not necessary, then a Unique DPP can simply be applied to the sorted array of elements (line 18)). The resulting array of unique elements is returned as output, along with the frequency count array (line 20); for the case that does not require frequency counts, the count array will just be returned in its original form.

3.2 Hashing-Based Algorithm

Collisions are a key aspect of hashing. Typically, these collisions are handled with either chaining (i.e., employing linked lists to store multiple entries at a single address) or open addressing (i.e., when an address is occupied, storing an entry at the next open address). While these strategies are straight-forward to implement in a serial setting, they do not directly translate to a data-parallel setting. For example, in a GPU setting where each thread is executing the same program, the variable number of operations resulting from chaining or open-addressing can lead to divergence (while non-collided threads wait for a collided thread to finish), and thus a performance bottleneck. Additionally, if multiple threads map to the same hash entry at the same time, then the behavior may be erratic, unless atomic operations are employed.

To address the problem of collisions in a data-parallel setting, we devise a modified hashing scheme that uses multiple iterations. In our scheme, no care is taken to detect collisions, making atomic operations unnecessary. Instead, every element is written directly to the hash table, possibly overwriting previously-hashed elements. The final hash table will then contain the winners of this “last one in" approach. However, our next step is to check, for each element, whether it was actually placed in the hash table. If so, the element is included for calculations during that iteration. If not, then the element is saved for future iterations. All elements are eventually processed, with the number of iterations equal to the maximum number of elements hashed to a single index. Throughout this iterative process, a binary indicator array keeps track of whether or not an element has been declared a duplicate element, as a result of a hash collision with another equal-valued element. The final output indicator array contains 1 values for non-duplicate elements and 0 values for duplicate elements. For search tasks that seek to remove all duplicate elements (i.e., those with a binary value of 0), the indicator array can then be used in conjunction with a CopyIf DPP to copy only the non-duplicate elements to a new, smaller output array.

In terms of hashing specifics, our hash function takes an element as input and produces an unsigned integer as output. For example, for triangular faces of a mesh cell, the hash function maps the three point indices of a face to an unsigned integer as output. This integer value, modulo the size of the hash table, is the hash index for the element. The hash function is important, as good function choices help minimize collisions, while poor choices create more collisions and, thus, more iterations. We experimented with multiple hash functions and used the best performing, FNV-1a, for our study (see Chapters V and IX for details).

3.2.1 Algorithm Details. The pseudocode for our DPP-based hashing algorithm is listed in Algorithm 2, and the following subsections complement this pseudocode with descriptions. We refer to this hashing-based algorithm as Hashing throughout this chapter.

The algorithm begins by computing an unsigned integer hash value for each element using a custom Map DPP that we denote ComputeHash (line 17). After this, the algorithm applies an iterative process to identify duplicate elements (lines 18–29). Within an iteration, some elements will not be successfully placed into the hash table because a collision with another element will displace it. These “lost" elements are identified and

Algorithm 2: Pseudocode for the hashing-based technique for identifying duplicate elements of a comparable type T. N is the total number of input elements, A is the number of active elements, and L is the number of non-duplicate elements. The constant c is a multiplier, or load factor, for the hash table size.

```
/*Input*/
template <typename T>
Array: T elements[$N$]
/*Output*/
Array: T nonDuplicates[$L$]
Array: int isNonDup[$N$]
/*Local Objects*/
Array: int hashes[$N$], elementIds[$N$], winningElementIds[$A$], hashTable[$cA$],
  isActive[$N$]
Array: T winningElements[$A$], activeElements[$A$]
/*Initialize variables and allocate arrays*/
$F = |\text{elements}|$;
$A \leftarrow F$;
hashTable← $\vec{\mathbf{0}}$;
elementIds← <0,...,$A-1$>;
isActive← $\vec{\mathbf{1}}$;
isNonDup← $\vec{\mathbf{1}}$;
hashes←ComputeHash(elements);
while $A > 0$ do
    hashTable←Scatter(hashes, elementIds, isActive, hashTable);
    winningElementIds←Gather(hashes, hashTable);
    winningElements←Gather(winningElementIds, elements);
    (isActive, isNonDup)← CheckForMatches(winningElements, elements,
      winningElementIds, isActive, isNonDup);
    elementIds←CopyIf(elementIds, isActive);
    isActive←CopyIf(isActive, isActive);
    $A$ ←Reduce(isActive); // new sum of active elements
    hashTable←Shrink(hashTable); // new smaller size $cA$
    activeElements←Gather(elementIds, elements);
    hashes←ComputeHash(activeElements);
end
if remove duplicate elements then // isNonDup[i] = 0
    nonDuplicates←CopyIf(elements, isNonDup);
    return nonDuplicates;
end
return isNonDup;
```

considered again in subsequent iterations. The algorithm terminates when every element has been considered, meaning that they had all been successfully placed into the hash table and classified as either a duplicate or non-duplicate. We refer to the set of elements that still need to be considered as “active elements," and the algorithm begins with every element as an active face (line 15).

The specifics of an iteration are as follows:

1. Write the unique Ids of all active elements into the hash table, via a Scatter DPP (line 19). The hash table destinations of the scatter are the hash indices, and it is this scatter process that results in collisions. The results of this operation are non-deterministic: elements with the same hash index displace (overwrite) each other, meaning only a subset of the active elements actually remain in the hash table at the end of the scatter.
2. For each hash index, retrieve the last element Id that was written into that hash table index, via a Gather DPP (line 20). Each of these elements (and their Ids) is denoted as a “winning" element, since it won the last-one-in “hash fight" among other possible colliding elements at the same hash index.
3. With the winning element Ids as reverse indices, gather the elements corresponding to the Ids, using another Gather DPP (line 21).
4. Detect whether any of the winning elements are duplicate elements, via a custom Map DPP that we denote CheckForMatches (lines 22). Each active element compares itself to the winning element residing at its hash location. Given the possibility of hashing collisions, multiple active elements may be compared to the winning element at a given location. If an active element and winning element are equal but have different Ids, then the elements are considered duplicates; however, if the elements are not equal, then a hashing collision has occurred at

this hash table location. If the elements are equal and share the same Ids, then the active element resides in the hash table (i.e, it is a winning key) and is considered non-duplicate unless it satisfies the duplicate element criteria with another active element (different Id). After being designated as either duplicate or non-duplicate, an active element becomes “inactive."

5. Using a CopyIf DPP (lines 23–24), compact the element Ids array to only those Ids that are still active, and correspondingly compact the isActive array.
6. Using a Reduce DPP, sum the number of 1-values in the isActive array (line 25). The resulting summation is the updated number of active elements remaining in the search.
7. Shrink the hash table to a constant load-factor c of the new number of active elements (line 26). Since the table is non-persistent, or refreshed, from iteration to iteration, the portion of the table that is removed is no longer needed. This change in hash table size will affect the output of the hash function (specifically, the modulus value) and the element hash indices.
8. Using the updated element Ids, gather the new set of active elements (line 27), and rehash them into the now-smaller hash table (line 28).

This iterative process continues until every element becomes inactive and has been considered as either duplicate or non-duplicate. If the maximum number of distinct elements (different values and Ids) that hash to a given hash table location is K, then at most K iterations will be performed. Since an active element that “collides" with a winning element does not immediately become inactive, it must wait to become the winning element (i.e., reside in the hash table) before becoming inactive and considered a duplicate or non-duplicate element. If $K-1$ other elements hash to the same location, then up to K iterations may be necessary for this element to become a winning element.

Finally, if the search task requires all of the duplicate elements to be removed from the final output array of elements, then a CopyIf DPP is applied to the array (line 31); a duplicate element has a binary value of 0 in the isNonDup array. The resulting array of non-duplicate elements is returned as output (line 32). However, if duplicate elements do not need to be removed, then the isNonDup array is returned as output (line 34). Note that our sorting-based algorithm should be used for search tasks that require counting the frequency of duplicates for each unique element, or returning the set of unique elements.

3.3 Dissertation Question

This chapter proposes a novel index-based search technique, HashFight, for identifying duplicate elements in an array. This technique is based on hashing and performs collision-handling in an iterative fashion without the use of hardware atomics or locking mechanisms. We also present a sorting-based technique for identifying duplicate elements. Both techniques are designed in terms of DPPs, which offers platform-portability across many-core systems. The hashing-based technique will be expanded into a general-purpose hash table data structure in Chapter IV, and both the sorting- and hashing-based techniques will be applied to a data-intensive visualization algorithm in Chapters V and IX. The experimental findings from this visualization application will help answer the dissertation question regarding the best index-based search techniques for visualization and analysis algorithms on diverse many-core systems.

CHAPTER IV

HASHFIGHT: DATA-PARALLEL HASH TABLE

In this chapter, we expand upon the collision-handling technique of Chapter III and introduce a new hash table data structure, which we refer to as *HashFight*. The operations of HashFight are composed entirely of DPPs and scale to billions of key-value pairs on both older and modern GPU and CPU platforms (single node, on-device memory), using only a single code base. Similar to the hashing-based technique of Chapter III, HashFight does not use any locking mechanisms or hardware atomics (e.g., compare-and-swap) to synchronize hash collisions that occur at the same hash table address. Instead, the occurrence of race conditions and “last one in" writes by parallel threads are fundamental features of our technique. Moreover, HashFight maintains a competitive peak memory footprint compared to best-in-class GPU implementations, and is not dependent on power-of-two table sizes, which are common in CPU-based implementations.

The content of this chapter is adopted primarily from a collaborative journal manuscript composed by myself, Shaomeng Li, and Hank Childs. As lead author, I designed and implemented the algorithms, and wrote the manuscript text. Shaomeng Li contributed significantly to the design and analysis of the corresponding experiments, which are discussed in Chapter VI as applied to unsigned integer hashing. Hank Childs provided valuable guidance and feedback towards the motivation and contributions of the work, and edited the manuscript text.

4.1 Design

The following section introduces our HashFight hash table and collision-handling approach. In this approach, keys are inserted into and queried from a multi-level hash table via an iterative scatter and gather routine that is based off the hashing routine of

Chapter III. During hash collisions at table locations, no explicit synchronization or hardware atomics are used. Instead, all colliding keys are inserted into the location in a winner-takes-all fashion, with the winner of the hash fight being the last key written into the location. Then, after all hash fights have completed, each thread checks its hash table location to see its key is currently residing in the table; that is, whether the key was the winner of its hash fight. All winning keys are marked as inactive and the remaining non-winning keys proceed to the next round, or iteration, where they'll hash fight again, but into a smaller smaller hash table. The size of the hash table at each iteration is equal to the number of active keys times the pre-specified hash table load factor. As in the first iteration, all active keys attempt to insert themselves into the subtable, and then check to see whether they've won the fight into the table or need to remain active for another iteration of hash fighting.

The hash fighting routine continues until a specified number, or threshold, of keys have become inactive and are successfully placed into a location in one of the subtables. After this threshold, the remaining active key-value pairs are sorted and then contiguously inserted into a buffer region at the end of the hash table. Since the hash table size decreases each iteration, new sets of colliding keys may arise at hash locations, leading to a variable number of iterations necessary to insert all keys. In all of our experimental configurations, particularly those with a small hash table load factor, the number of iterations rarely exceeds 6 and never reaches 10 (note that only the first quarter or so of the total iterations account for most of the overall runtime).

For querying keys within the hash subtables, hash fighting is performed in a similar fashion to the insertion phase. Each iteration, all active keys hash to their location in the current subtable and then read the key-value pair residing in that location. If the residing key is equal to the query key, then the value is returned and the query key is set to inactive. If not equal, then the key remains active and performs a query again within the subtable of the subsequent iteration. Since each table location is initially populated with an “empty" key-value pair prior to the insertion phase, the residing key will remain empty if no input pairs were hashed and inserted into that location. For this latter case, a query key immediately returns an empty value and is set to inactive. As in the insertion phase, the querying continues until a minimum threshold of active keys is reached, after which the remaining active keys each binary search for their query matches within the ending buffer region of the hash table. This region contains the keys from the insertion phase that were sorted and contiguously inserted.

HashFight is particularly designed and tested for static hashing in which key-value pairs are first inserted into a table, followed by subsequent querying, without inter-mixed modifications (e.g., deleting pairs or changing the values of keys). If entries in the hash table must be deleted or new entries inserted, then the hash table must be reconstructed.

Due to its data-parallel processing, HashFight is best-suited to perform insertions and queries in large batches of keys, with each key being assigned to one of many threads. Thus, the execution of HashFight on small, or even individual, workloads of keys can result in sub-optimal performance, as the overhead time of the DPP kernel invocations exceeds the time needed to perform the actual hash table operation.

Listings 1 through 5 provide algorithm pseudocode for the HashFight insertion and query phases, along with the scatter and gather subroutines that compose the majority of the overall computation. These phases are discussed in more detail as follows.

4.1.1 Insertion Phase.

In the `Insert` function, keys are hashed to random locations in the hash table and then inserted as pairs with their accompanying values (Listing 1). Given an array of 32-bit unsigned integer keys and an array of 32-bit unsigned integer values (the arguments in lines 1 and 2), each value is appended to

```
void Insert(const uint32 *keys,
            const uint32 *vals,
            HashTable &ht,
            uint32 numKeys)
{
  //All keys start active
  uint32 activeKeys = numKeys;
  uint8 *isActive[numKeys] = {1};
  //Hash into subtables
  uint32 tableStart = 0;
  uint32 tableSize =
    activeKeys*ht.loadFactor;

  //Hash until a lower limit
  //of active keys is reached.
  while (activeKeys > HASHING_LIMIT)
  {
    ht.subTableSizes.push_back(tableSize);
    //Active keys hash into subtable
    Fight(keys,vals, ht.entries,
          isActive, tableStart,
          tableSize);
    //Active keys check if they won.
    //If a winner, a key is deactived.
    CheckWinner(keys, ht.entries,
                isActive, tableStart,
                tableSize);
    tableStart = tableSize;
    activeKeys = Reduce(isActive);
    tableSize = activeKeys*ht.loadFactor;
  }
  //Sort remaining active keys and
  //insert them into end of table.
  uint32 *tempKeys, tempVals;
  CopyIf(keys, isActive, tempKeys);
  CopyIf(vals, isActive, tempVals);
  SortByKey(tempKeys, tempVals);
  CopyToTable(tempKeys, tempVals,
              ht.entries, tableStart,
              HASHING_LIMIT);
}
```

Listing 1 Pseudocode for the HashFight Insert function.

its corresponding key to form a 64-bit unsigned integer pair. The hash table structure (line 3) is pre-allocated sufficient memory to store these pairs, and each location in the table is initialized to an “empty" key-value pair `UINT_MAX « 32`. Initially, all keys are considered “active" (line 7) and marked as such with an `isActive` array of bit indicators (line 8).

Next, the hash fighting routine begins, consisting of multiple iterations (line 17). Each iteration is assigned a separate contiguous partition, or sub-table, of the larger hash table, into which the currently-active keys are hashed and inserted. This sub-table is specified by a start location (line 11) and a size (line 12), the latter of which is equal to the number of active keys times a pre-specified hash table load factor. As the number of active keys decreases each iteration, the sub-table sizes decrease proportionally.

Given a sub-table, the active keys proceed to insert, or fight, themselves into the sub-table.

In the `Fight` kernel (Listing 2), a thread is assigned to each key (line 10), and only threads with active keys (line 12) perform computation. Each thread computes the hash value of its assigned key (line 18) and takes the modulo of the sub-table size to determine the write location (line 19). We use a randomly-generated, non-cryptographic hash function that is a variant of the MurmurHash, which is efficient to compute and maintains strong hash properties to minimize collisions between 32-bit unsigned integers. Then, the threads concurrently write their key-value entries into the sub-table (line 22). During this scatter process, no locking mechanism or hardware atomics are used to synchronize simultaneous writes. Instead, race conditions are a fundamental and non-detrimental feature of handling hash collisions. Multiple threads may contend for the same location and overwrite each other (one after the other), but the final thread to write its pair into the location is declared the “winner" of the hash fight.

```
void Fight(const uint32 *keys,
           const uint32 *vals,
           const uint8 *isActive,
           uint64 *entries,
           uint32 tableStart,
           uint32 tableSize)
{
  //Thread index
  uint32 tid = getGlobalIndex();

  if (isActive[tid])
  {
    uint32 key = keys[tid];
    uint32 value = vals[tid];
    uint64 entry =
      ((uint64)(key) << 32)+value;
    uint32 hash = Hash(key);
    hash = (hash%tableSize)+tableStart;
    //Non-atomic write
    entries[hash] = entry;
  }
}
```

Listing 2 Pseudocode for the HashFight Fight function.

After all threads have finished hash fighting their keys, they return to the Insert function and proceed to determine whether they have “won" the fight, or successfully inserted the keys into the sub-table (Listing 1, line 26). In the `CheckWinner` kernel (Listing 3), each thread re-computes the hash location of its assigned (and active) key (line 13) and reads the currently-residing, or “winning,” key-value pair at the location in the sub-table (line 14). If the thread’s key is equal to the winning key (line 17), then the thread won the hash fight and marks the key as inactive (line 18). Otherwise, the key remains active and another thread will attempt to insert the key again in the next iteration of the insertion phase.

```
void CheckWinner(const uint32 *keys,
                 const uint64 *entries,
                 uint8 *isActive,
                 uint32 tableStart,
                 uint32 tableSize)
{
  //Thread index
  uint32 tid = getGlobalIndex();

  if (isActive[tid])
  {
    uint32 hash = Hash(keys[tid]);
    hash = (hash%tableSize)+tableStart;
    uint64 entry = entries[hash];
    uint32 winningKey =
      (uint32)(entry >> 32);
    if (winningKey == keys[tid])
      isActive[tid] = 0;
  }
}
```

Listing 3 Pseudocode for the HashFight CheckWinner function.

Finally, the threads return to the insertion function and the local function variables are updated for the next iteration (lines 30 through 33), including a Reduce data-parallel operation that counts and updates the number of active keys (or set bits).

The hash fighting continues until a minimum number of active keys remain; in this work, we used a minimum of 1 million keys (line 17). After this point, all the active key-value pairs are sorted in ascending order by key (lines 38 through 41) and then contiguously written into a new sub-table, or buffer region (line 42). This ending sort and write procedure is meant to be faster and simpler to perform than hash fighting a small number of active keys into smaller-sized sub-tables. With a small number of active keys, new collisions arise and induce extra hash fight iterations and overhead of kernel invocations.

```
 1 void Query(const uint32 *queryKeys,
 2            const HashTable &ht,
 3            uint32 *queryVals,
 4            uint32 numKeys)
 5 {
 6   //All query keys start active
 7   uint32 activeKeys = numKeys;
 8   uint8 *isActive[numKeys] = {1};
 9   //Hash into subtables
10   int iter = 0;
11   int numTables = ht.subTableSizes.size();
12   uint32 tableStart = 0;
13   uint32 tableSize = ht.subTableSizes[0];
14
15   //Query each subtable
16   while (iter < numTables)
17   {
18     //Probe the sub-table for the
19     //active query keys.
20     Probe(queryKeys, ht.entries,
21           queryVals, isActive,
22           tableStart, tableSize);
23
24     tableStart = tableSize;
25     tableSize = ht.subTableSizes[iter++];
26   }
27   //Binary search the end of hash table
28   //for remaining active query keys.
29   BinarySearch(queryKeys, queryVals,
30                isActive, ht.entries,
31                tableStart, HASHING_LIMIT);
32 }
```

Listing 4 Pseudocode for the HashFight Query function.

4.1.2 Query Phase. In the `Query` function (Listing 4), a batch of input query keys (line 1) is looked-up within the multiple sub-tables of a hash table (line 2), which was previously constructed in the insertion phase (Listing 1). The result of this function is an output array of values (line 3) corresponding to the query keys. Since the query keys are independent from the keys inserted into the hash table, a certain percentage

```
 1 void Probe(const uint32 *keys,
 2            const uint64 *entries,
 3            uint32 *vals,
 4            uint8 *isActive,
 5            uint32 tableStart,
 6            uint32 tableSize)
 7 {
 8   //Thread index
 9   uint32 tid = getGlobalIndex();
10 
11   if (isActive[tid])
12   {
13     uint32 queryKey = keys[tid];
14     uint32 hash = Hash(queryKey);
15     hash = (hash%tableSize)+tableStart;
16     uint64 entry = entries[hash];
17     uint32 residingKey =
18       (uint32)(entry >> 32);
19 
20     bool isEqual = queryKey==residingKey;
21     bool isEmpty = residingKey==UINT_MAX;
22     //If keys match or residing key is
23     //empty, then deactivate query key.
24     if (isEqual || isEmpty)
25       isActive[tid] = 0;
26     //Query is successful, so return
27     //the residing value.
28     if (isEqual)
29       vals[tid] = (uint32)entry;
30   }
31 }
```

Listing 5 Pseudocode for the HashFight Probe function.

of the query keys may not exist within any of the sub-tables, prompting an “empty," or failed, query value to be returned. Initially, all query values are set to an empty value of `UINT_MAX`, which is the largest 32-bit unsigned integer value.

Similar to the `Insert` function, all input query keys are marked as active (lines 7 and 8) and local parameters are initialized to record the start index and size of a sub-table (lines 11 through 14). Then, `numTables` iterations of querying are conducted, with each iteration searching for active query keys within a different sub-table (line 18). A data-parallel Probe kernel is invoked (line 23) to look-up, or probe, the query keys within the sub-table. Within this kernel (Listing 5), each query key is assigned to a thread (line 9), which computes the hash table location of the key (if active) and reads the residing key-value pair at that location (lines 14 through 16). If the query key is equal to residing key, then the residing value is returned as the query value and the query key is marked inactive (lines 26 and 31). If the table location contains the empty key-value pair, then no key was ever hashed to that location, and the query key is marked inactive (line 25). Since the query value is set to empty by default, a query value does not need to be returned. Finally, if the residing key is neither empty nor equal to the query key, then the query key *may* exist within another sub-table and, thus, remains active.

After all probing has completed, the threads return to the Query function, and local parameter values are updated for the next iteration of querying (lines 27 and 28). Once all sub-tables have been searched via hashing, any remaining active query keys are binary-searched (line 33) within the sorted buffer region of keys that were inserted at the end of the hash table during the insertion phase (Listing 1, line 42). This binary search is performed in data-parallel fashion by threads assigned to the remaining query keys. As in the Probe kernel, if the query key is not found, then an empty query value is returned; otherwise, the residing value in the table is returned.

4.1.3 Peak Memory Footprint. Assuming 4-byte (32-bit) keys and values, and 8-byte (64-bit) hash table entries, the peak CPU memory allocation (in bytes) required for both the insertion and querying phases is approximated by the following equation:

$$mem_{cpu} = (9 + 12.6f)|K| + 8|Q| + 8|B|, \tag{4.1}$$

where f is the hash table load factor, $|K|$ is the number of input keys, $|Q|$ is the number of query keys, and $|B| << |K|$ is the number of active keys that are sorted and copied directly into the buffer region B at the end of the hash table (Listing 1, line 40).

On GPU devices, the memory allocated on-device is released following the completion of each phase and kernel, except for the hash table. Since copies of the keys and values are inserted into the hash table as pairs, the original arrays can be released from GPU memory following the insertion phase. Then, during the querying phase, the query keys and values arrays must be transferred into GPU memory. Due to the `isActive`, `tempKeys`, and `tempVals` arrays (Listing 1), the insertion phase induces the peak GPU memory usage (in bytes):

$$mem_{gpu} = (9 + 12.6f)|K| + 8|B|, \tag{4.2}$$

which is similar to mem_{cpu} minus the allocation for the query keys and values arrays.

4.2 Dissertation Question

This chapter proposes a novel general-purpose hash table data structure that is based upon the HashFight collision-handling technique introduced in Chapter III. This hash table supports batch insertions and queries of key-value pairs, and performs these operations using the iterative, atomics-free HashFight approach. All operations and index-based search techniques are designed in terms of DPP, which enables the hash table to be constructed and queried on diverse many-core platforms with a single implementation. In Chapter VI, we will assess the performance of the our hash table on large sets of unsigned integer key-value pairs, comparing the resulting insertion and query throughput against that of benchmark implementations from leading parallel libraries. The experimental findings from this data analysis application will help inform the dissertation question regarding the best index-based search techniques for visualization and analysis algorithms on diverse many-core systems.

Part II

Applications

In this part of the dissertation, we apply the DPP design patterns and index-based search techniques of Part I to a collection of problems in the scientific visualization and data analysis domains. In particular, we introduce new DPP-based algorithms for the following four applications:

1. External facelist calculation (EFC)
2. Hashing integer key-value pairs
3. Maximal clique enumeration (MCE)
4. Graph-based image segmentation

Each of these applications are data-intensive and require non-trivial reformulations into a DPP-form.

CHAPTER V

EXTERNAL FACELIST CALCULATION

This chapter applies the duplicate element search techniques of Chapter III to the scientific visualization task of external facelist calculation (EFC). The following content is adopted primarily from a collaborative paper that was published at a conference and composed by myself, Roba Binyahib, Robert Maynard, and Hank Childs. As lead author, I developed the contributed techniques, wrote the majority of the manuscript text, and conducted a significant portion of the experiments CPU and GPU experiments. Roba Binyahib helped me conduct the experiments and contributed to the writing of the experimental overview text. Hank Childs acquired the datasets for the experiments and helped review the text. Robert Maynard helped interpret the experimental results and ensure accuracy of the algorithm implementations.

The remainder of this chapter proceeds as follows. Sections 1 and 2 offer a high-level overview of EFC, with a description of the design challenges facing DPP-based EFC. Section 3 provides a thorough background of EFC techniques and related work. Section 4 documents the modifications needed to perform duplicate element searching for EFC, using the techniques of Chapter III. Section 5 reviews our experimental configurations and implementation details. Section 6 presents the results of a set of CPU and GPU experiments for our DPP-based algorithms. Section 7 compares the performance of our DPP-based algorithms to that of existing serial implementations for EFC. Finally, Section 8 summarizes our findings for the dissertation question.

5.1 External Facelist Calculation

Scientific visualization algorithms vary regarding the topology of their input and output meshes. When working with three-dimensional volumes as input, algorithms such as isosurfacing and slicing produce outputs (typically triangles and quadrilaterals) that can be rendered via traditional surface rendering techniques, e.g., rasterization via OpenGL. Algorithms such as volume rendering operate directly on three-dimensional volumes, and use a combination of color and transparency to produce images that represent data both on the exterior of the volume and in the interior of the volume. However, some scientific visualization algorithms take three-dimensional volumes as input and produce three-dimensional volumes as output. While these three-dimensional volume outputs could be rendered with volume rendering or serve as inputs to other algorithms such as isosurfacing, users often want direct renderings of these algorithms' outputs using surface rendering. With this work, we consider this latter case, and consider the approach where geometric primitives are extracted from a volumetric unstructured mesh, in order to use traditional surface rendering techniques.

Given, for example, an unstructured mesh of N connected tetrahedrons to render, a naïve solution would be to extract the four faces that bound each tetrahedron, and render the corresponding $4 \times N$ triangles. This naïve solution would be straight-forward to implement and would fit well with existing rendering approaches. However, many of the $4 \times N$ triangles this algorithm would produce are contained within the interior of the volume, and thus not useful. The primary downside to the naïve approach, then, is efficiency. For a data set with N tetrahedrons, only $O(N^{\frac{2}{3}})$ of the faces would actually lie on the exterior, meaning the large majority of the $4 \times N$ faces produced are unwanted, taking up memory to store and slowing down rendering times. If N was one million, then the expected number of external faces would be approximately 10,000, where the naïve algorithm would calculate four million faces, i.e., 400X too many. A second downside to these triangles is that they can create rendering artifacts. If the faces are rendered using transparency, then internal faces become visible, which is typically not the effect the user wants when they opt to use surface rendering on a three dimensional volume.

A better algorithm, then, is to produce only the faces that lie on the exterior of the mesh, so called external facelist calculation (EFC). EFC is a mainstay in scientific visualization packages, specifically to handle the case of rendering the exteriors of three-dimensional volumes via surface-rendering techniques.

5.2 Combination and Challenges

The challenges with EFC and DPP are two-fold. One, serial EFC is traditionally done with hashing, which is non-trivial to implement with DPP. As a result, we needed to construct new, hashing-inspired algorithms that sidestep the problems with traditional hashing within DPP. And, although DPP has been shown to be efficient with more traditional scientific visualization algorithms that iterate over cells or pixels, EFC is essentially a search problem, and so it is unclear if DPP will perform well. On this front, we demonstrate that DPP does indeed perform well and again does provide good performance on this class of scientific visualization problem.

The contribution of this chapter, then, is to illuminate the best techniques to execute EFC with DPP. We adopt the sorting- and hashing-based search algorithms of Chapter III to introduce two novel DPP-based algorithms for performing EFC. We then conduct a performance study that assesses the runtime performance of our algorithms across multiple data sets and architectures. Our findings demonstrate that our hashing-inspired algorithm is the best approach for EFC on multi-core and many-core architectures using DPP.

5.3 Related Work

EFC comes up surprisingly often in scientific visualization. For example, many engineering applications, such as bridge and building design, use the external faces of their model as their default visualization, often to look at displacements. Further, clipping and interval volumes are also commonly used with external facelist calculation. In these algorithms, a filter removes a portion of the volume (based on either spatial selection or data selection); if no further operations are performed then EFC is needed to view the clipped region. As a final example, some material interface reconstructions approaches, like that by Meredith et al. [99], take three-dimensional volumes and create multiple three-dimensional volumes, each one corresponding to a pure material. In this case, when users remove one or more materials, EFC is needed to view the material boundaries.

While not an active area of research, implementations of EFC can be found on the internet, for example with VTK's `vtkUnstructuredGridGeometryFilter` [147] and VisIt's `avtFacelistFilter` [145]. The basic idea behind these filters is to count how many times a face is encountered. If it is encountered twice, then it is internal, since the face is incident to two different cells, and so it is between them. If a face is encountered a single time, then it is external, since there is no neighboring cell to provide a second abutment.

In both implementations readily available on the internet, the “face count" is calculated through hashing. That is, in the first phase, every face is hashed (with the hash index derived from the point indices that define the face) into a large hash table. Then, in the second phase, the hash table is traversed. If a face was hashed into a hash table index two times, then it is internal and discarded. But if it was hashed into a hash table index only a single time, then it is external, and the face is added to the output.

Niessner et al. [112] employ a DPP-based hashing approach on the GPU to perform scalable, real-time 3D scene reconstruction of live captures from a depth camera. This work uses a speed-efficient hash table data structure to insert, retrieve, and delete voxel blocks, each of which store sensor data for a uniform subregion of the perceived world. While the hashing routines are data-parallel, they, however, depend on atomic operations to avoid race conditions and collisions that can arise when inserting hash entries in parallel. Our proposed DPP-based hashing algorithms for EFC do not depend on this restriction and resolve collisions via an iterative, non-blocking process. Moreover, the work of [112] does not apply data-parallel hashing to the EFC task, which is the focus of our work.

5.4 Algorithms

This section modifies the sorting- and hashing-based algorithms of Chapter III to operate on the triangular faces of tetrahedral cells in an unstructured mesh. In Chapter III, these algorithms for searching for duplicate elements are defined over elements of an arbitrary comparable type. In this chapter, duplicate element searching is performed over three-dimensional point vectors, each of which store the indices of the three points of a triangular cell face. The input to each algorithm is an array of cell faces that may contain multiple pairs of duplicate faces (i.e., abutting cells have faces that overlap each other). Our strategy is to first identify all duplicate, or internal, faces via modified variants of the duplicate element searching algorithms, and then remove all of the internal faces. The resulting compacted array contains only the non-duplicate, or external, faces, which are returned as output.

The EFC algorithms remain the same as those in Chapter III, except for the following differences:

- The comparison operator for equality of faces consists of two parts. First, the three point Ids of a three-dimensional face point vector are compared in order with those of another point vector. Second, the unique integer face Ids of the two faces are compared for equality. A decision is then made as to whether the faces are distinct or duplicates.
- The hash function in the hashing-based technique maps the three point Ids of a face into a single unsigned integer value. This value, modulo the size of the hash table,

becomes the hash table index into which the face is hashed. However, the point vector of the face itself is not written to this index. Instead, the unique integer Id of the face is written into the table, while the point vector is stored in a separate auxiliary array. Thus, each face has an associated key-value pair, where the unique Id is the key and the point vector is the value. During a pairwise comparison between faces, the Id is used to gather the point vector from the auxiliary array, and then both the Id and vector are used in the comparison. We experimented with multiple hash functions and used the best performing, FNV-1a, for this study; refer to Chapter IX for a performance comparison of different hash functions.

Note that we refer to duplicate elements as duplicate faces within the context of EFC, since the elements under consideration are cell faces. The reader is referred to Chapter III for the pseudocode of our DPP-based algorithms for duplicate element searching.

5.5 Experiment Overview

In this section, we discuss our experimental setup and the different configurations that are tested.

5.5.1 Factors.

This study varies the following four factors:

1. Data set: Since the choice of data set may affect algorithm performance, we varied the data over both size and layout.
2. Hardware platform: In order to evaluate platform-portable performance, we test our implementation on both a CPU and GPU platform. For the CPU, we also consider the effect of concurrency on runtime performance, by varying the number of hardware cores.
3. Algorithm implementation: We assess the variation in performance for two different DPP-based algorithms for EFC.

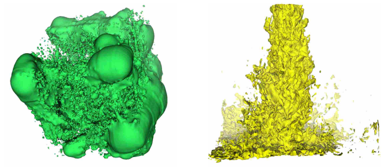


Figure 1. Visualizations of two of the 3D data sets used in the EFC experiments. The Enzo-10M data set is on the left and Nek-50M is on the right.

4. Hash table size: For the hashing-based algorithms, we vary the starting size of the hash table, and observe its effect on performance.

5.5.2 Software Implementation. Both of our EFC algorithms are implemented using the VTK-m toolkit. With VTK-m, a developer chooses DPP to employ for algorithm design and then customizes them with functors of C++-compliant code. This code is then used to create platform-specific code for platforms of interest, such as CUDA code for NVIDIA GPUs and Threading Building Blocks (TBB) code for Intel CPUs. In our experiments, both the TBB and CUDA configurations of VTK-m are compiled with the gcc compiler, and the VTK-m index integer (vtkm::Id) size was set to 32 bits.

5.5.3 Configurations. Next, we vary the four factors over a sequence of five phases, resulting in 420 total test configurations. The number of options per factor is as follows:

- Data set (6 options)
- Hardware architecture (7 options)
- Algorithm (2 options)
- Hash table size (5 options)

These configurations are discussed in the following subsections. Section 5.4 provides a discussion on the two EFC algorithms and their implementation.

5.5.3.1 Data Sets. We applied our test cases to six data sets, four of which were derived from two primary data sets. Figure 1 contains renderings for these two data sets.

- Enzo-10M: A cosmology data set from the Enzo [118] simulation code. The data set was originally on a 128^3 rectilinear grid, but was mapped to a 10.2M tetrahedral grid. The data set contains approximately 20M unique faces, of which 194K are external.
- Enzo-80M: An 83.9M tetrahedron version of Enzo-10 M, with approximately 166M unique faces, of which 780.3K are external.
- Nek-50M: An unstructured mesh that contains 50M tetrahedrons from a Nek5000 thermal hydraulics simulation [38]. The data set contains approximately 100M unique faces, of which 550K are external.
- Re-Enzo-10M, Re-Enzo-80M, Re-Nek-50M: Versions of our previous three data sets where the point lists were randomized. Especially for the Enzo data sets, the regular layout of the data leads to cache coherency — by randomizing the point list, each tetrahedron touches more memory. Specifically, each individual tetrahedron in the mesh occupies the same spatial location as its non-randomized predecessor, but the four points that define the tetrahedron no longer occupy consecutive or nearby points in the point list.

Finally, while we reference the data sources, we note the only important aspect for evaluating EFC performance is the mesh and mesh connectivity.

5.5.3.2 Hardware Platforms. We ran our tests on the following two platforms:

1. CPU: A 16-core machine running 2 nodes, each with a 2.60 GHz Intel Xeon(R) E5-2650 v2 CPU with 8 cores and 16 threads. Each CPU has a base frequency of 2.6

GHz, memory bandwidth of 59.7 GB/s, and 64GB shared memory. We also vary the number of cores: 1, 2, 4, 8, 12 and 16. Each concurrency uses the Intel TBB library for multi-core data-parallelism.

2. GPU: An NVIDIA Tesla K40 Accelerator with 2880 processor cores, 12 GB memory, and 288 GB/sec memory bandwidth. Each core has a base frequency of 745 MHz, while the GDDR5 memory runs at a base frequency of 3 GHz. All GPU experiments use NVIDIA CUDA V6.5.

5.5.3.3 Hash Table Size. For our hashing-based EFC algorithm, we assess the runtime performance as the hash table size changes. The table size is set at various multiples of the total number of faces in the data set. In this study, we considered five options: 0.5X, 1X, 2X, 4X, and 8X.

The 0.5X option underscores the difference between regular hashing and our hashing variant. With regular hashing and a chaining approach, the size of the hash table must be at least as large as the number of elements to hash, and preferably much larger. With our variant, the table size can be decreased, with the only penalty being that there will be more iterations, as the maximum number of faces hashed to a single index will (on average) increase proportionally. In this way, the memory allocated to the hash table can be reduced, but at the cost of increased execution time.

5.6 Results

Our study consists of five phases. The first phase examines one case in depth (“base case") and the remaining four phases each examine the impact of different parameters on performance: hash table size, architecture, data sets, and concurrency. In this section, we present and analyze the results of these different phases. We refer to the sorting-based EFC algorithm as Sorting and the hashing-based algorithm as Hashing.

Time	Sorting	Hashing
Main Computation	0.5	0.6
Total Time	0.9	0.9

Table 1. Comparison of overall CPU execution times (sec) for the Sorting and Hashing EFC algorithms. Total Time represents the time for both the Main Computation and Initialization.

Phase	CPU Time
Initialization	0.2
Sort	0.3
ReduceByKey	0.2
CopyIf	0.02
Overhead	0.2
Total time	0.9

Table 2. Individual CPU phase times (sec) for the Sorting EFC algorithm.

5.6.1 Phase 1: Base Case.

Our base case assesses the performance of Sorting and Hashing with the following configuration of factors:

Configuration: (CPU, 16 cores, Enzo-10M, hash table factor 2 for Hashing) × 2 algorithms.

For each algorithm, we measured the total execution time, along with the sub-times for the primary data-parallel operations and routines. Additionally, we measured the overhead time for memory allocations and de-allocations. The results of the Sorting and Hashing CPU-based experiments are presented in Tables 1 through 3.

As seen in Tables 1 and 2, Sorting completed the experiment in 0.9 seconds, with the Sort and Reduction operations—the main computation—accounting for 56% of the total time. From Tables 1 and 3, Hashing performed comparably with Sorting, with the main hashing operations contributing 67% of the 0.9-second total runtime.

5.6.2 Phase 2: Hash Table Size.

In this phase, we study the effect of the hash table size on the performance of the Hashing algorithm, using the following set of factors:

Phase	CPU Time
Initialization	0.2
Scatter	0.1
CheckForMatches	0.3
CopyIf	0.1
ComputeHash	0.05
Overhead	0.1
Total time	0.9

Table 3. Individual CPU phase times (sec) for select DPP operations of the Hashing EFC algorithm.

Multiplier	0.5X	1X	2X	4X	8X
Total Time	1.0	0.9	0.9	0.8	0.8

Table 4. CPU execution time (sec) of the Hashing EFC algorithm as a function of the hash table size multiplier.

Time	Sorting	Hashing
Main Computation	0.5	0.2
Total Time	0.7	0.4

Table 5. GPU execution time (sec) for the main computation of the Sorting and Hashing EFC algorithms.

Configuration: (CPU, 16 cores, Enzo-10M) × 5 different hash table proportions.

For each multiplier c, the initial hash table size is computed as $c * F$, where F is the total number of non-unique faces ($F \approx 40$ million for the Enzo-10M data set).

The results in Table 4 show that Hashing is only modestly affected by the hash table multiplier. This primarily occurs because the hash values are recomputed each iteration, leading to a slower decrease in collisions, which results in a more-stable number of hashing iterations. Moreover, in memory-poor environments, lower multipliers can be used with only a modest slowdown in execution time.

5.6.3 Phase 3: Architecture.

In this phase, we assess the performance of the algorithms on a GPU architecture with the Enzo-10M data set.

Configuration: (GPU, Enzo-10M) × 2 algorithms.

Phase	GPU Time
Initialization	0.1
Sort	0.5
ReduceByKey	4.0e-02
CopyIf	4.5e-03
Overhead	0.1
Total time	0.7

Table 6. Individual GPU phase times (sec) for the Sorting EFC algorithm.

Phase	GPU Time
GetFacePoints	0.1
Scatter	0.1
CheckForMatches	0.1
StreamCompact	4.1e-02
ComputeFaceHash	8.0e-03
Overhead	0.1
Total time	0.4

Table 7. Individual GPU phase times (sec) for the Hashing EFC algorithm.

From Table 5, we observe that Hashing achieves a faster runtime than Sorting. Moreover, Hashing devotes only half of its total execution time to main computation, which, for hashing, is the cumulative time spent in the hashing while-loop. Contrarily, Sorting spends more than 70% of its total runtime on the CUDA Thrust Sort operation, which, along with the ReduceByKey operation, comprises the main computation; see Table 6 for GPU sub-times of the Sorting algorithm. Table 7 shows that the Scatter and CheckForMatches DPP account for at least half of the work for Hashing. This contrasts slightly from the equivalent CPU findings of Phase 1. The results of Table 7 indicate that the GPU significantly reduced the runtime of these DPP operations.

5.6.4 Phase 4: Data Sets. This phase explores the effects of data set, by looking at six different data sets, which vary over data size and memory locality. The study also varies platform (CPU and GPU) and algorithm (Sorting and Hashing).

Configuration: (CPU, 16 cores, GPU) × 6 data sets × 2 algorithms.

Data set	CPU		GPU	
	Sorting	Hashing	Sorting	Hashing
Enzo-10M	0.9	0.9	0.7	0.4
Nek-50M	4.3	4.3	3.3	2.1
Enzo-80M	7.4	7.3	7.4	7.3
Re-Enzo-10M	1.2	0.9	1.0	0.4
Re-Nek-50M	5.5	4.5	5.3	2.2
Re-Enzo-80M	9.2	7.7	10.1	6.5

Table 8. CPU and GPU execution times (sec) for different EFC data set/algorithm pairs.

Table 8 displays the execution times on the CPU platform using 16 cores. These results show that Sorting is affected by the locality of the cells within a mesh, as evident from the increase in runtime between the pairs of regular and restructured data sets. Further corroborating this observation, Table 8 also shows that Sorting realizes a nearly 1.5–time speedup in total runtime when presented with the restructured version of a data set on the GPU architecture. Contrarily, Hashing maintains stable execution times regardless of the cell locality in data sets.

With respect to execution time on both the CPU and GPU, Hashing consistently achieves comparable CPU performance to Sorting for the regular data sets and significantly better CPU and GPU performance for the restructured data sets. These findings indicate that Hashing is superior for GPU-based execution and for data sets with poor memory locality (both CPU and GPU).

5.6.5 Phase 5: Concurrency. In this phase, we investigate the CPU runtime performance of both Sorting and Hashing using different numbers of hardware cores with the base case Enzo-10M data set and its corresponding Re-Enzo-10M data set.

Configuration: (CPU) × 6 different concurrency levels × 2 data sets × 2 algorithms.

Tables 9 and 10 show that, although Sorting performs better than Hashing on configurations of 8 cores or fewer, Hashing provides stable performance regardless of

Method	1	2	4	8	12	16
Sorting	8.0	4.3	2.3	1.7	1.1	0.9
Hashing	10.8	5.6	3.9	1.9	1.1	0.9

Table 9. Impact of the number of CPU cores on the execution time (sec) for the Sorting and Hashing EFC algorithms using the Enzo-10M data set.

Method	1	2	4	8	12	16
Sorting	9.6	5.1	2.9	1.9	1.3	1.1
Hashing	11.2	5.8	3.1	1.9	1.1	0.9

Table 10. Impact of the number of CPU cores on the execution time (sec) for the Sorting and Hashing EFC algorithms using the Re-Enzo-10M data set.

memory locality; this confirms our findings from the previous phases. Additionally, the results indicate that with 1 CPU core, there is nearly a 10-time increase in runtime over the 16-core experiment, for both Sorting and Hashing. This observation demonstrates clear parallelism; however, the speedup is sub-linear.

A review of Hashing over both Enzo-10M datasets indicates that only the initialization, *ComputeHash*, and *CheckForMatches* DPP operations achieve near-linear speedup from 1 core to 16 cores. The remaining operations (e.g., *Scatter* and *CopyIf*) achieve sub-linear speedup, contributing to the overall sub-linear speedup. For a majority of the individual operations, the smallest runtime speedup from a doubling of the hardware cores occurs in the switch from 8 to 16 cores. These findings suggest that, on up to 8 cores (a single CPU node), scalable parallelism is achieved, whereas from 8 to 16 cores (two CPU nodes with shared memory) parallelism does not scale optimally, possibly due to hardware and multi-threading limitations.

5.7 Comparing to Existing Serial Implementations

In Section 5.6.5, Hashing demonstrated a nearly 10-time increase in runtime over the base 16-core configuration, when executed on 1 CPU core. This single-core experiment simulates a serial execution of Hashing and motivates a comparison with the serial EFC implementations of community scientific visualization packages.

Data set	VTK	VisIt	Hashing
Enzo-10M	6.2	1.4	10.8
Nek-50M	33.1	5.2	60.9
Enzo-80M	51.7	9.1	102.6
Re-Enzo-10M	9.9	2.1	8.2
Re-Nek-50M	59.1	10.3	41.5
Re-Enzo-80M	84.4	17.7	109.1

Table 11. Single-core (serial) CPU execution time (sec) for different EFC data set/algorithm pairs.

This section compares the runtime of serial Hashing (1-core) with that of the VTK `vtkUnstructuredGridGeometryFilter` and VisIt `avtFacelistFilter`, both of which are serial, single-threaded algorithms for EFC.

In Table 11, we observe that the VisIt algorithm outperforms both the VTK and Hashing algorithms on all of the data sets from Section 5.6.4, while Hashing performs comparably with the VTK implementation. The overall weak performance of Hashing is to be expected, since the DPP-based implementation is optimized for use in parallel environments. When compiled in VTK-m serial mode, the DPP functions are resolved into backend, sequential loop operations that iterate through large arrays without the benefit of multi-threading. Thus, Hashing is neither optimized nor designed for 1-core execution. In particular, it introduces extra instructions to resolve hash collisions that are unnecessary in this setting. Contrarily, both VisIt and VTK are optimized specifically for single-core, non-parallel environments, leading to better runtimes than Hashing on the majority of the datasets. However, in a parallel setting, both Sorting and Hashing achieve better runtime performance than the serial algorithms

5.8 Conclusion

This chapter contributes two novel DPP-based algorithms for the EFC visualization application, both of which use index-based search techniques. Both algorithms—one based on sorting and the other based on hashing—are designed in terms of DPPs and tested on both CPU and GPU devices, demonstrating platform-portability across many-core systems. These algorithms are thought to be the first shared-memory, parallel algorithms for EFC.

Within each algorithm, the search task involves identifying duplicate, overlapping cell faces, which accounts for the majority of the computational work in the algorithm. In particular, the hashing-based technique is a specialization of the HashFight technique for duplicate element searching that was introduced in Chapter III. For the EFC application, the HashFight approach demonstrates superior CPU and GPU runtime performance compared to the sorting-based approach, particularly on larger and more-complex 3D unstructured mesh datasets. Moreover, the both approaches realize improved CPU runtime performance as concurrency increases. Based on these findings, we believe that our hashing-based solution is the best index-based search technique for this particular EFC visualization algorithm on diverse many-core systems. These findings inform our dissertation question and will be further synthesized in Chapter X.

CHAPTER VI

HASHING INTEGER KEY-VALUE PAIRS

This chapter applies the HashFight hash table from Chapter IV to the analysis task of hashing and querying large sets of unsigned integer key-value pairs. We compare the insertion and query performance of HashFight to that of state-of-the-art CPU- and GPU-based hash table implementations from open-source parallel computing libraries.

The content presented here is adopted primarily from a collaborative journal publication composed by myself, Samuel Li, and Hank Childs. As lead author, I wrote the majority of the text contained in this chapter, and collaborated with Samuel Li to conduct and analyze the CPU and GPU experiments. Hank Childs provided valuable feedback on the experimental design and findings, and final presentation of the manuscript.

The remainder of this chapter proceeds as follows. Section 1 summarizes current parallel-hashing research relevant to this study, including a review of our comparator hash tables. Section 2 presents an overview of the suite of hashing experiments and configurations. Section 3 documents and analyzes the results of these experiments. Section 4 summarizes our findings for the dissertation question. Refer to Chapter IV for a detailed account and review of the HashFight operations and collision-handling routine.

6.1 Background

The following section summarizes the state-of-the-art hash tables and techniques that we compare against in this study.

6.1.1 Parallel Hashing. Since the emergence of multi- and many-core CPUs and general-purpose computing on GPUs, a large body of research has investigated the design of parallel hashing techniques [80]. In this parallel setting, multiple threads each simultaneously perform one or more hash table operations, with one operation per assigned key. Typically, operations are performed in batches to maintain work balance among threads and saturate available parallelism. Further, for dynamic hash tables, each batch may consist of mixed operations in any order, defining a distribution of insertions, updates, queries, and deletions; static hash tables are built with an initial batch of insertions, followed by a batch of queries. Each concurrent operation at a hash table location must be synchronized to handle hash collisions and prevent reader-writer race conditions. For instance, queries may attempt to read recently-deleted keys or access a location before a desired key is inserted, resulting in a false failure. Moreover, if multiple concurrent insertions prompt the hash table to reach maximum occupancy, then only one request should be granted access to extend the capacity of the table, creating a need for synchronized memory allocation.

6.1.1.1 CPU-based Techniques. Single-node, CPU-based approaches have primarily focused on the design of dynamic, concurrent hash tables within shared memory [46, 101, 122, 134, 45]. These tables synchronize concurrent operations with either lock-based methods (e.g., mutexes or spin-locks) or lock-free hardware atomics (e.g., compare-and-swap (CAS)). Many of these tables are implemented as linked list data structures to support extensibility (or resizing) and separate-chaining collision-resolution. This form of collision handling requires careful synchronization of colliding key-value pairs during concurrent insertions, as each new key at a table location needs to have a new node allocated and appended to the linked list. For lock-based tables, a performance bottleneck may arise when there is high contention at any given hash table location. For example, when expanding a linked list to include several new colliding keys (insertions), concurrent queries remain blocked until access is granted to probe the list.

Notable open-source library implementations of CPU-based concurrent hash tables are provided within the Intel Thread Building Blocks (TBB) library and the Microsoft Parallel Patterns Library (PPL) [53, 102]. Both libraries include a *concurrent* *unordered map* hash table that supports lock-free (non-blocking) insertions, queries, and updates, using an underlying linked list, with CAS atomics over nodes [101, 134]. These unordered maps are extensible and hash key-value pairs into buckets, or segments of the linked list, similar to the unordered map provided by the C++ Standard Library. However, both maps do not support concurrent-safe deletions and the hash table size is expected to be a power of 2, which may affect the choice of hash function used.

6.1.1.2 GPU-based Techniques. Single-node, GPU-based hashing techniques have become an active area of research, spurred by the massive available thread- and instruction-level parallelism offered on modern GPU architectures [80]. GPUs are specifically designed for *data-parallel* processing, whereby a single instruction is performed over multiple data elements (SIMD) in parallel, such as via a vector instruction[1]. This contrasts with performing a single instruction over a single scalar data value (SISD), which is common on CPU architectures. In particular, the execution and threading model of the GPU presents various design challenges for parallel hashing:

- Threads works in small logical processing groups, or *warps*, to execute the same instruction over different data elements in parallel. For a memory load or store instruction, each thread makes its own addressable memory request and the warp services as many memory transactions as necessary to satisfy all requests. The greater the number of transactions, the longer the latency to complete the work of the warp. Thus, warp throughput is optimized when threads coalesce their memory requests within the same memory transaction or cache line.
- Threads also possess their own control flow and may take divergent branches or paths to complete a warp instruction, such as a conditional statement, atomic

[1] NVIDIA Tesla-generation compute GPUs employ single-instruction, multiple threads (SIMT) execution, which is a combination of SIMD and simultaneous multi-threading (SMT) execution.

operation, or spin-lock mutex. If one branch requires more instructions to complete than another branch, then the faster-completing threads will wait idle for the slower threads to finish, causing starvation and serialization of the control flow. Thus, warp throughput is optimized when threads follow the same branch.

- CPU-based hashing leverages large on-chip caching and shared (global) memory to service random-access memory requests quickly. On the GPU, the translation-lookaside buffer (TLB) and data caches are limited in size, which can induce more cache misses and expensive global memory transactions. Moreover, the limited global memory on the GPU restricts the maximum size of the hash table and input data that can fit on-device.

In traditional data-parallel hashing, threads in a warp are each assigned one or more key-value pairs and together execute the same operation, such as an insertion or query. Modern NVIDIA GPUs specify 32 threads per warp and multiple warps fit within a thread block (maximum 1024 threads), which contains register and shared memory space for the resident threads. Multiple blocks are dispatched in parallel on one of several streaming multiprocessors (SMs) that contain hundreds of compute and warp-scheduling units. The set of SMs can dispatch a large number of threads in parallel overall, creating an ideal platform for large amounts of key-value pairs to be hashed and queried in a data-parallel fashion. However, since hashing is a memory-bound problem designed for random-access memory insertions and queries, the occurrence of cache misses and uncoalesced memory requests among warp threads is inevitable. With the limited-caching design of the GPU, these requests are likely to result in high-latency global memory loads and stores. Depending on the number of hash collisions and collision-handling approach, warp branch divergence may arise when faster, non-colliding threads immediately insert their pairs and wait for the slower, colliding threads to finish.

Chapter II surveyed and categorized a large body of research on data-parallel GPU hashing techniques, and identified performance challenges affecting the design of such techniques. Due to these challenges, many of the most-successful GPU hashing techniques maintain a static, on-device hash table with open-addressing collision-handling. These tables are efficient to construct and use fine-grained, hardware atomic primitives to synchronize table accesses and modifications.

The best-in-class open-source library implementation of GPU parallel hashing is based on cuckoo hashing and is packaged within the CUDA Data Parallel Primitives Library (CUDPP) [27], which contains top-performing algorithms and data structures written in NVIDIA CUDA. Introduced by Alcantara et al. [3], this general-purpose, cuckoo hash table supports a variable number of hash functions, hash table size, and maximum length of a probing sequence. The hash table resides in global memory and is constructed in parallel by threads each inserting key-value pairs into locations specified by the cuckoo hash functions. Insertions and evictions are synchronized using CAS atomic primitives, and each thread manages the re-insertion of any key-value pair that it evicts along the eviction chain, until a pair is finally placed in an empty table location. Since this is a static hash table, queries are performed after insertions and, thus, do not require an atomic primitive for each probe. If new key-value pairs need to be inserted into the table post-construction, then the table is constructed again. Also, the table is reconstructed if a thread exceeds it's maximum eviction, or probe, chain length during the insertion phase.

The CUDA Thrust library of data-parallel algorithms and data structures [117] provides fast and high-throughput data-parallel implementations of mergesort [128] and radix sort [100] for arrays of custom or numerical data types, respectively. Additionally, Thrust includes a data-parallel, vectorized binary search primitive to efficiently search within a sorted array. The combination of sorting an array and searching within it has been widely-used as a benchmark for search-based tasks, particularly hashing [3, 6]. As a platform-portable library, Thrust also provides implementations of these algorithms and data structures in TBB and OpenMP for CPU execution.

6.2 Experimental Overview

In this section, we assess the performance of HashFight across several different factors, comparing its performance to that of best-in-class comparator implementations. Our primary measure of hashing performance is the throughput of insertions and queries, which is calculated as the number of key-value pairs inserted or queried per second. The different experimental factors are outlined as follows, each factor consisting of multiple options.

- Algorithm (5 options)
- Platform (3 options)
- Dataset size (29 options)
- Hash table load factor (10 options)
- Query failure rate (10 options)

Since some of the configurations are not compatible together, We do not test the cross product of all configurations. For example, some algorithms cannot be executed on a CPU platform, and several dataset size and load factor combinations would exceed available on-device memory of certain GPU platforms. The details of each factor and configuration are discussed in the following subsections.

6.2.1 Algorithms. We compare the performance of HashFight with that of four different parallel hashing and search-based implementations from well-known open-source libraries:

- **HashFight** (CPU+GPU): Data parallel primitive based implementation with a single code base for both CPU and GPU platforms.
- **Thrust-Sort/Search** (CPU): Quick sort and vectorized binary search implementations written in TBB and contained within the Thrust library.
- **CUDPP** (GPU): Cuckoo hash table implementation written in CUDA and packaged within the CUDPP library (see Section 6.1.1).
- **Thrust-Sort/Search** (GPU): Radix sort and vectorized binary search implementations written in CUDA and provided in the Thrust library (Section 6.1.1).
- **TBB-Map** (CPU): Lock-free, concurrent unordered map hash table written in TBB and packaged within the TBB library of parallel algorithms (see Section 6.1.1).

HashFight is written with the open-source VTK-m library (v1.2), which is C++-11/14 compliant and provides a set of generic data-parallel primitives (e.g., Reduce, Sort, Scan, Copy, and LowerBounds) that can be invoked on both GPU and CPU devices with a single algorithm code base. For NVIDIA GPU execution, primitives from the CUDA Thrust library are invoked; for CPU execution, primitives from the Intel TBB parallel algorithms library are invoked. Thus, HashFight can be mapped to either CUDA- or TBB-compliant code, via the back-end implementations of the primitives inside of VTK-m.

The Thrust Sort and Search comparators are meant to provide a baseline measure of throughput performance for a search-based task such as hashing. The combination of sorting key-value pairs and then querying them via binary search is a canonical alternative to constructing and querying a hash table.

6.2.2 Platforms. To assess the cross-platform performance of HashFight, we conduct experiments on the following two GPU devices and one CPU device, each residing on a single node:

- **K40 GPU**: NVIDIA Tesla K40 accelerator with 11.4 GB on-board memory.
- **V100 GPU**: NVIDIA Tesla V100 accelerator with 32.5 GB on-board memory.
- **Xeon Gold CPU**: 2.3 GHz Intel Xeon Gold 6140 (Skylake generation) CPU with 36 physical cores (72 logical) and 370 GB DDR4 memory.

All CPU code was compiled using GCC with flags for `-O3` optimization and the C++-11 standard. Additionally, the TBB scalable allocator was used for dynamic memory allocation with the TBB concurrent unordered map, which resizes itself as new key-value pairs are inserted. All GPU code was compiled using NVCC with GCC as the host compiler.

6.2.3 Dataset Sizes. In this study we focus on the task of hashing unique, unsigned 32-bit integer key-value pairs into the hash table. Each key and value is randomly-generated by the Mersenne Twister pseudo-random generator, using a state size of 19937 bits (mt19937). Any duplicate keys are removed, and the remaining unique keys are shuffled. To construct a hash table, a batch of *k* unsigned integer keys and *k* corresponding unsigned integer values are provided as input from two separate datasets. To query the hash table, a batch of *k* randomly-generated, unsigned integer keys is provided as input from a separate dataset. These query keys may contain duplicates and are not necessarily equal to any of the keys previously inserted into the table.

Among the four tested GPU and CPU platforms, we generate and experiment with datasets containing between 50 million and 1.45 billion unsigned integers (*k*), in increments of 50 million. This results in 29 different sizes of the number of key-value pairs and query keys. The maximum size that can be executed on a given platform is a function of the maximum on-device memory of the platform and the hash table load factor (see Equations 1 and 2). Thus, holding the load factor constant, on devices with larger available memory, we are able to insert and query larger sets of key-value pairs.

6.2.4 Hash Table Load Factors. We measure the effect of the insertion and query throughput as the hash table load factor, f, is varied between 1.03 and 2.0, inclusive. Overall, 10 different values of f are tested: 1.03, 1.10, 1.15, 1.25, 1.40, 1.50, 1.60, 1.75, 1.90, 2.0. A load factor of 1.03 was selected as the minimum value because the CUDPP cuckoo hash table implementation is only designed and tested for load factors of at least this value. Traditionally, a load factor of 2.0 has served as the conservative upper-bound for constructing a hash table [26]. The smaller load factors reduce the memory footprint of the hash table, but at the expense of an increase in the number of hash collisions.

6.2.5 Query Failure Rates. For a dataset size of k query keys, we randomly-generate 10 different sets of k query keys, each with a different percentage of keys that are not contained within the hash table; that is, “failed" queries that return empty query values. The rate of failure of query keys is varied, in increments of 10 percent, from 0 percent (all query keys exist in the table) up to 90 percent. This failure rate factor is meant to assess the worst-case querying ability of hash table implementations.

6.3 Results

In this section, we present and analyze the findings of our GPU and CPU hashing experiments. For each experiment, we assess the insertion and query throughput of three search-based techniques (HashFight compared to a benchmark hash table and sorting-based technique) as the dataset size, load factor, and query failure rate are varied. Each configuration, or data point, is run 10 trials, with each trial using a different randomly-generated data set of unique, 32-bit unsigned integer keys and values. The result of each configuration is reported as the average throughput of the 10 different runs, with the operation being an insertion or query. For a given configuration, the same 10 data sets are used by each hashing implementation, in order to provide a fair comparison.

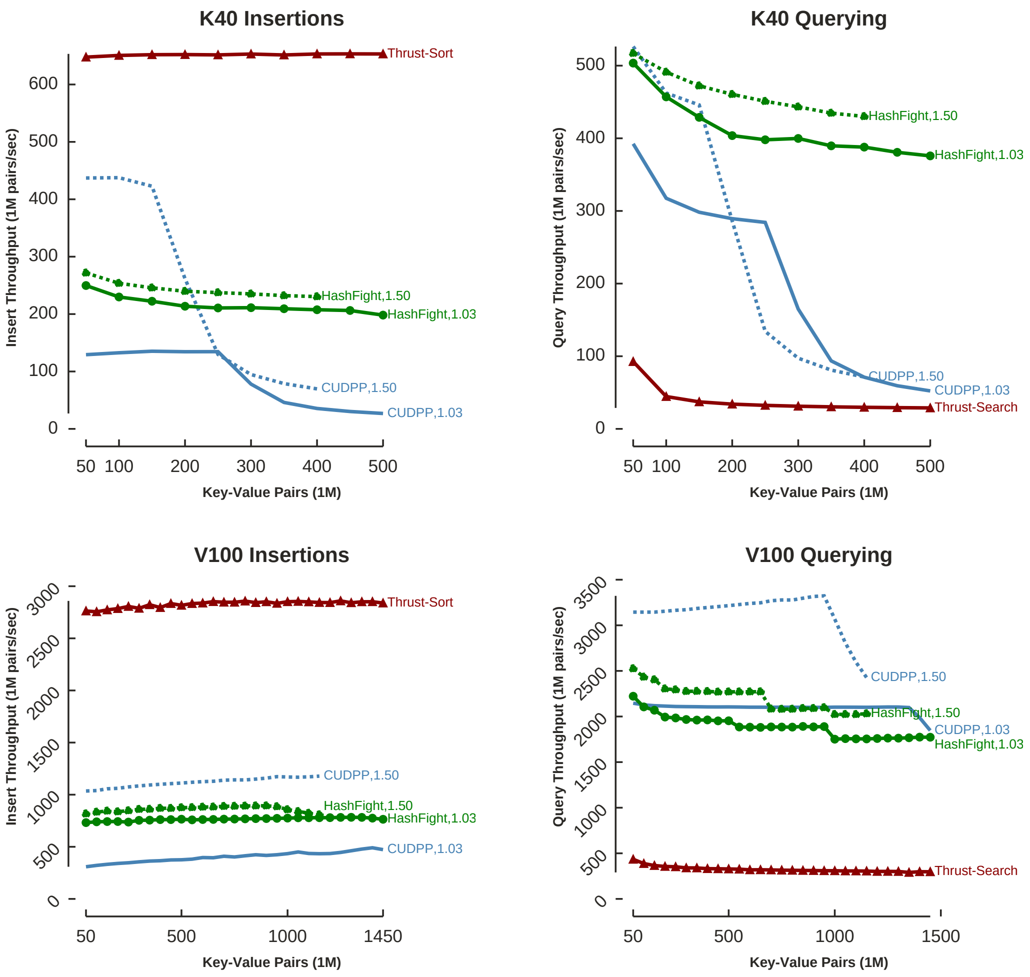


Figure 2. GPU insertion and query throughput as the number of key-value pairs is varied on the K40 and V100 devices. For both HashFight and CUDPP, hash table load factors of 1.03 and 1.50 are presented separately.

6.3.1 GPU Experiments. We conduct three different GPU experiments, each run on both the K40 and V100 NVIDIA GPU devices. The results and analysis of these experiments are as follows.

6.3.1.1 Vary Data Size. Our first GPU experiment assesses the throughput performance of each hash table as the number of key-value pairs is increased and the load factor is held constant. We display results for constant load factors of 1.03 and 1.50, since they reveal the performance of both higher-capacity and lower-capacity hash tables.

Figure 2 displays the results for both the K40 and V100 GPUs, each testing a different maximum number of key-value pairs due to on-device memory limitations. From these results, we observe that HashFight achieves scalable and leading query throughput for the largest numbers of key-value pairs on both GPU devices, while remaining within a factor of 1.5 of the CUDPP cuckoo hash table for all other data sizes. For insertions, both hash table approaches maintain comparable throughput, with HashFight demonstrating more-stable throughput for both of the tested load factors, and CUDPP performing its best for the 1.50 factor. Overall, both HashFight and CUDPP achieve higher throughput for queries than for insertions, which is a desirable property for hash tables that are used primarily as look-up structures, particularly in real-time applications.

With the 1.03 load factor, HashFight attains a consistently-higher insertion and query throughput than CUDPP on both devices, and matches the query throughput of CUDPP for the smallest and largest data set sizes on the V100 device. With the 1.50 load factor, CUDPP sees an increase in query throughput and outperforms HashFight within a factor of 1.5 until 950 million key-value pairs. However, when the number of key-value pairs exceeds 1 billion, CUDPP experiences a drop in throughput and nearly matches the throughput of HashFight at 1.45 billion pairs. This trend can also be observed for CUDPP queries on the K40 device at both 150 million key-value pairs (1.03 load factor) and 250 million pairs (1.50 load factor).

A further analysis reveals that these dropoff points directly coincide with the points at which the total memory of the hash table exceeds the size, or coverage, of the translation lookaside buffer (TLB) of the last-level cache (L3 cache on K40 and L2 cache on V100). According to the micro benchmarking of Jia et al., the V100 and K40 have TLB sizes of approximately 8.2 GB and 2 GB, respectively [55]. On the K40 and V100, these last-level caches use physical memory addresses and so the virtual addresses of thread read and write requests must first be translated into physical addresses via one of the page tables cached in the TLB. Once the pages tables of the TLB are full, TLB misses induce page faults (or swaps) that increase the latency of memory transactions, regardless of whether the memory requests hit or miss the last-level cache. Recently, Karnagel et al. microbenchmarked a suite of modern NVIDIA GPUs and discovered that irregular memory accesses for data sets larger than 2 GB on the K40 result in latency increases, due to inefficient accesses to the L3 TLB [61]. Lai et al. expand upon this finding by modeling a multi-pass scatter and gather scheme that splits a batch of TLB-exceeding memory accesses into smaller chunks of accesses that each fit within the size of the TLB [72].

HashFight draws from these findings and performs the `Fight`, `CheckWinner`, and `Probe` kernels (Listings 2, 3, and 5) in a multi-pass fashion that, as seen in Figure 1, does not suffer a drop in throughput after the TLB size is exceeded. Once the hash table memory size reaches the TLB size, the hash table is logically broken into chunks of locations, each of roughly equal size and smaller than the TLB size. Then, a kernel is invoked for each chunk in order, and only the threads with memory accesses into the current chunk are allowed to insert or query their key. The minimization of TLB page faults more than offsets the overhead of invoking each kernel multiple times per HashFight iteration, resulting in more stable throughput than CUDPP for large data sizes and hash tables. This multi-pass feature maintains platform-portability, since TLB-specific constant values are only used when HashFight is compiled in GPU (CUDA) mode.

On the V100 GPU device, both tables realize a considerable improvement in insertion and query throughput, reaching approximately 1 billion pairs per second for

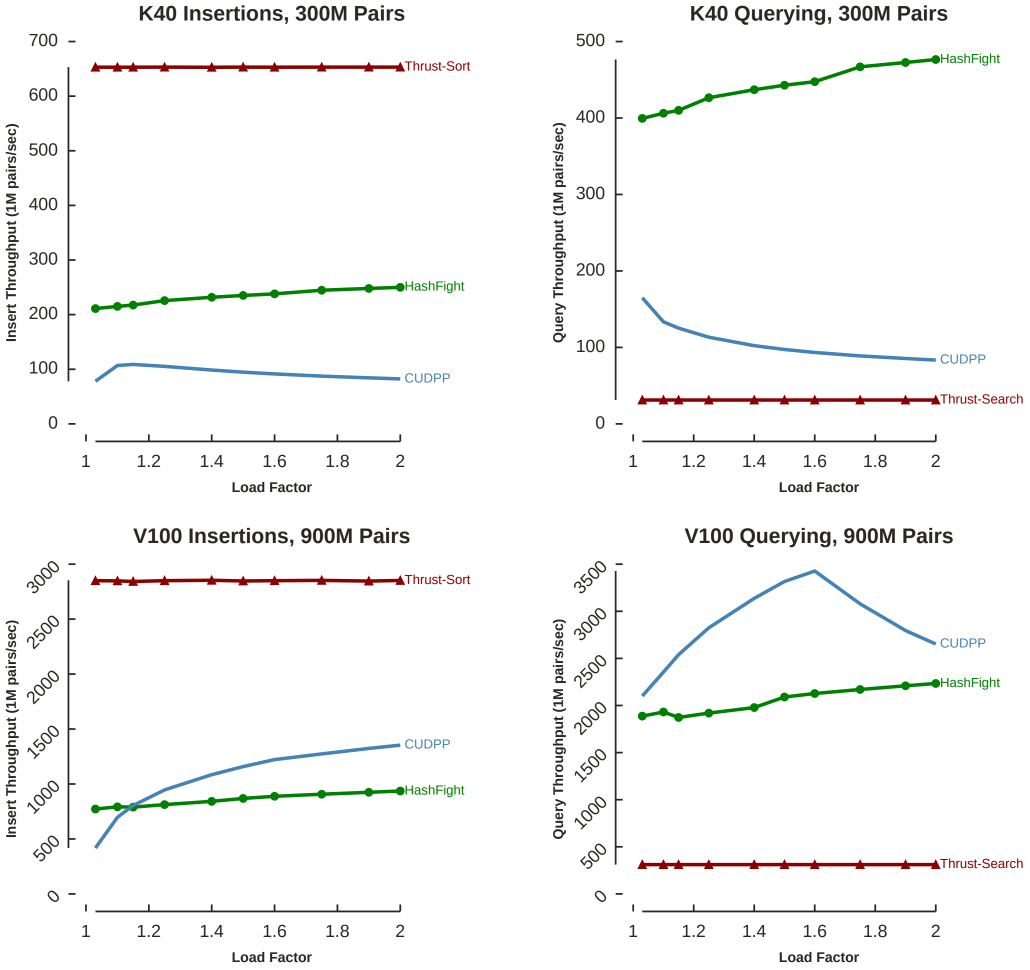


Figure 3. GPU insertion and query throughput as the hash table load factor, or capacity, is varied on the K40 and V100 devices. The number of key-value pairs inserted and queried is set equal to the maximum number that can satisfy on-device memory constraints for all load factors.

insertions and at least 2 billion pairs per second for queries. This increase in throughput can be attributed to several hardware improvements over the older-generation K40:

- Increase in measured global memory bandwidth from 183 GB per second to 736 GB per second. This improvement is particularly beneficial to the random-access nature of hashing, which is highly-dependent on global memory accesses due to cache misses.

– Increase in the L2 instruction and data cache size from 1.57 MB to 6.14 MB.

– Increase in the L2 load bandwidth from 340 GB per second to 2.16 TB per second.

– Decrease in latency, or the number of clock cycles, for global memory atomic instructions, particularly during high thread contention. This improvement is due to the introduction of hardware atomics (as opposed to software emulation) in post-K40 devices.

Each of these factors play a role in the absolute throughput differences observed between the devices, as CUDPP uses CAS atomic operations during insertions and both CUDPP and HashFight resolve a very high percentage of their memory transactions from global memory.

Finally, Figure 1 reveals that, for both K40 and V100, sorting the input key-value pairs with the Thrust radix sort is significantly faster than inserting the pairs into either of the two hash tables. However, this comes with a tradeoff of significantly slower queries via the Thrust binary search, which suffers from uncoalesced and random-access query patterns into the sorted array. The hash tables enable each query to complete within a constant fixed number of uncoalesced probes through the hash table, whereas the binary search must make a logarithmic number of probes in the worst case.

6.3.1.2 Vary Load Factor. Our second GPU experiment measures the performance of the hash table approaches as the load factor is varied between 1.03 and 2.0, and the number of key-value pairs is held constant. We hold 300 million and 900 million pairs constant for the K40 and V100 devices, respectively, as these are the maximum numbers for which all load factors can be tested within on-device memory limits. We also plot the throughput of the Thrust radix sort and binary search for both of the tested number of key-value pairs; since the load factor is hash table-specific, the Thrust performance does not vary.

From Figure 3 we see that HashFight maintains very stable throughput performance across all load factors, never deviating by more than 200 million pairs per second. CUDPP, as an open-addressing method, achieves faster insertions, as the load factor and hash table capacity increase, particularly on the V100 GPU. On the K40 GPU, CUDPP fails to increase it's insertion throughput for 300 million pairs beyond a load factor of 1.10. At this point, the hash table memory usage reaches the maximum L3 TLB capacity of the K40 and, as noted before, CUDPP is unable to further increase it's insertion throughput due to excessive TLB page faults. Also, as seen in Figure 2, HashFight performs comparably or better than CUDPP for the smallest load factors and highest-capacity hash table sizes.

6.3.1.3 Vary Query Failure Rate. Our third and final GPU experiment assesses the query throughput of HashFight and CUDPP as the percentage of failed, or unsuccessful, query keys is varied between 0 and 90 percent, while holding the number of query keys and load factor (1.03 and 1.50) constant.

Figure 4 reveals that CUDPP and HashFight are modestly affected by an increase in unsuccessful queries on both the K40 and V100. On the V100, CUDPP begins with a slightly higher query throughput than HashFight for the 1.50 load factor, but then sees a 43 percent decrease in throughput until it nearly matches the throughput of HashFight at the 90 percent failure rate. A similar pattern appears for the 1.03 load factor. On the K40, both hash table approaches realize slightly larger decreases in query throughput as compared to the V100 runs. For the 90 percent query failure rate and 1.50 load factor, HashFight and CUDPP realize decreases in throughput by 33 and 43 percent, respectively. For the 1.03 load factor, HashFight and CUDPP observe decreases in throughput by 48 and 47 percent, respectively, until the 90 percent failure rate, at which point the throughput of both tables is equivalent.

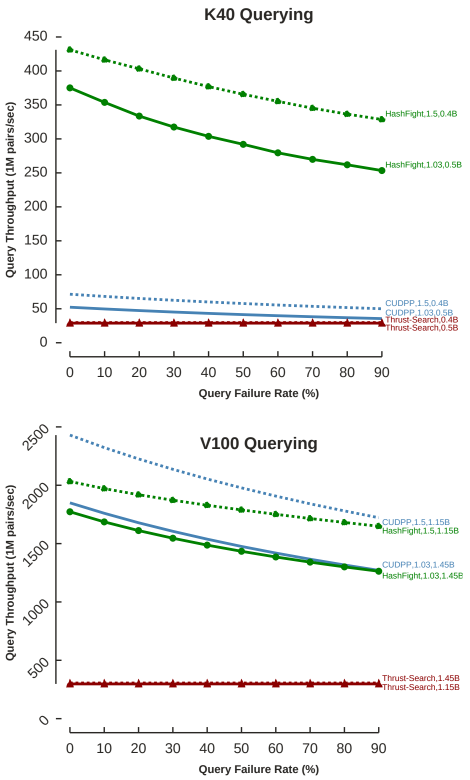


Figure 4. GPU query throughput as the percentage of failed, or unsuccessful, queries is varied on the K40 and V100 devices. For both HashFight and CUDPP, separate plots are presented for hash table load factors of 1.03 and 1.50, which are queried with the maximum number of query keys (millions) permitted within on-device memory limits. The Thrust binary search throughput is plotted for the same number of query keys.

The K40 findings can be largely explained by the TLB caching limit and HashFight's TLB-oblivious design, as observed in the previous experiments, whereas the V100 findings are more indicative of algorthmic properties, such as increase in worst-case probes per thread and memory load transactions per warp. As an optimized cuckoo hash table, CUDPP only requires at most *h* lookup probes per query, where *h* is the number of cuckoo hash functions or possible table locations for a key to be inserted. Since this value is typically small (4 in this study), CUDPP does not have to perform too many extra global memory loads to determine that a query key does not exist within the hash table. HashFight is even less affected by the failed queries, as most threads can determine in the

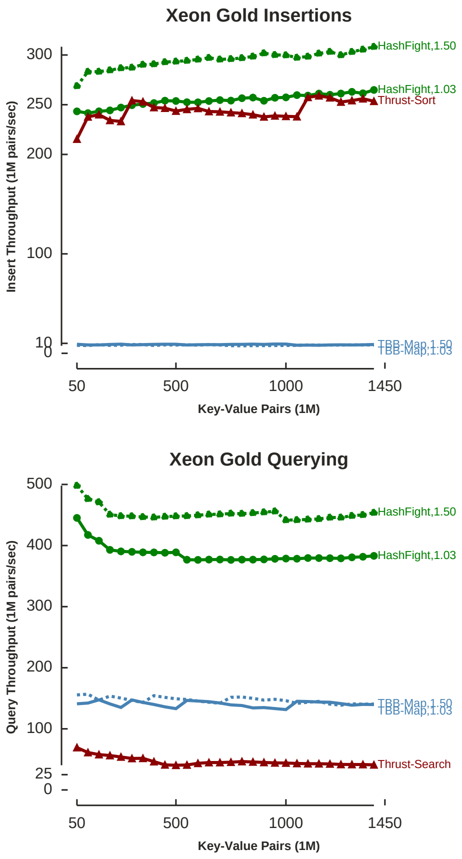


Figure 5. CPU insertion and query throughput as the number of key-value pairs is varied on the Xeon Gold device. For both HashFight and the TBB unordered map, hash table load factors of 1.03 and 1.50 are presented separately.

first iteration whether their active query key resides in the table, and rarely will need to exceed 6 iterations to determine failure.

6.3.2 CPU Experiments. We conduct three different experiments on the Intel Xeon Gold CPU, comparing the throughput of HashFight with that of the TBB concurrent unordered map (TBB-Map) and the Thrust sort and binary search primitives. HashFight and Thrust are compiled and run in TBB mode, without any code changes from the equivalent GPU experiments. The results and analysis of the CPU experiments are presented as follows.

Figure 5 shows that HashFight achieves a significantly higher throughput than the TBB-Map for both insertions and queries across all data sizes. In particular, for the largest batch of key-value pairs, 1.45 billion, and smallest load factor, 1.03, the throughput of HashFight exceeds that of TBB-Map by approximately 30X and 3X for insertions and queries, respectively. From Figure 6, as the load factor, or table capacity, is increased from 1.03 up to 2.0, HashFight continues to increase its insertion and query throughput by 1.3X and 1.4X, respectively. However, TBB-Map maintains relatively the same throughput for insertions (8.5 million pairs/sec) and queries (137.5 million pairs/sec) until a load factor of 1.60. After this point, TBB-Map realizes a noticeable 6.3X decrease in insertion throughput and a 1.35X decrease in query throughput, instead of expected increases. Also, the aggregate runtime of performing a Thrust sort followed by a vectorized binary search is actually faster than that of TBB-Map, yet still considerably slower than the aggregate runtime of HashFight.

The significantly lower insertion throughput of TBB-Map is largely due to the underlying linked list design of the hash table and its construction under large magnitudes of key-value pairs, such as those tested in this experiment. As an extensible hash table, TBB-Map must frequently resize and allocate new memory segments as more and more key-value pairs are inserted in an unordered fashion [134]. Each insertion of a key-value pair has to follow multiple layers of pointer indirection to access a segment bucket, and the insertion must be synchronized via a lock-free atomic primitive, which adds additional overhead above that of HashFight.

Moreover, the slight drop in throughput for TBB-Map above a load factor of 1.50 is a combination of excessive TLB cache thrashing and the use of separate-chaining for collision resolution. A hash table with a load factor of 1.50 requires at least 16GB of memory to insert 1.45 billion pairs. This memory footprint just exceeds the

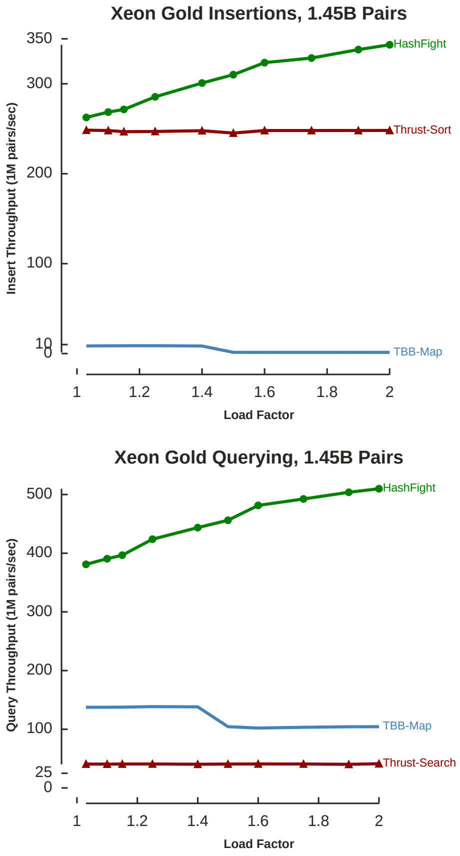


Figure 6. CPU insertion and query throughput as the hash table load factor is varied on the Xeon Gold device.

aggregate coverage of the L1 and L2 TLBs on the Xeon Gold CPU, and, since page entries are of smaller 4KB and 2MB sizes, pages are frequently swapped in and out of a TLB by concurrent threads during random-access insertions. As observed in our GPU experiments, memory latency increases and insert throughput decreases as more hash table address mappings reside outside of the TLB. HashFight obviates this issue by means of its multi-pass gather and scatter procedure, and its use of open-addressing collision resolution, whereby the number of memory accesses typically decreases as the hash table size increases. However, TBB-Map is vulnerable to TLB memory limitations and does not benefit as well from a larger hash table due to multiple layers of pointer indirection and linear probing within buckets. A possible direction towards alleviating the TLB issue is to use huge 1GB page entries. This permits more virtual-to-physical memory mappings in a single TLB access, but requires modifying the system configuration with root permission; due to the latter reason, we were unable to perform this experiment.

In order to validate the results of TBB-Map, we also tested the same CPU experiments with the TBB concurrent hash map, which is based on an underlying array structure as opposed to a linked-list. The concurrent hash map produced nearly identical insertion and query throughput as the concurrent unordered map. Additionally, by conducting multiple trials per configuration (10), we verified that the resulting runtimes were not affected by “cold" threads, whereby some threads are not yet active and need a warmup phase.

Finally, we conducted the third experiment of varying the query failure rate and observed that, unlike in the GPU experiments, the query throughput of HashFight and TBB-Map is only marginally decreased as the rate is increased. Since there are many buckets in the TBB-Map hash table and a relatively light load per bucket, the cost of performing an unnecessary or failed query is non-increasing, contrary to extra probing required by CUDPP cuckoo hashing for failed queries on the GPU.

6.4 Conclusion

This chapter applies the HashFight hash table of Chapter IV to the task of hashing and querying large datasets of unsigned integer key-value pairs. We demonstrate the viability of HashFight as a general-purpose hashing approach via a suite of experiments. HashFight achieves higher insertion and query throughput rates than best-in-class GPU- and CPU-based implementations across a wide variety of experimental configurations. In particular, on different GPU devices, HashFight attains competitive to leading insertion and query throughput as the number of pairs tested grows large towards billions of pairs. On CPU devices, HashFight achieves leading throughput performance across all configurations tested, demonstrating platform-portable performance of the DPP-based design. For both GPU and CPU experiments, the throughput of HashFight is robust to different hash table sizes and query access patterns.

Based on these findings, we believe that our HashFight hash table solution is the best index-based search technique for this particular data analysis application on diverse many-core systems. These findings inform our dissertation question and will be further synthesized in Chapter X.

CHAPTER VII

MAXIMAL CLIQUE ENUMERATION

In this chapter, we assess the viability of DPP-based design patterns for a data- and memory-intensive graph algorithm that consists of index-based search routines. Specifically, we contribute a new data-parallel approach for the task of *maximal clique enumeration* (MCE) within undirected graphs, and demonstrate single-node, platform-portable runtime performance that exceeds that of state-of-the-art MCE approaches for graphs with a high ratio of maximal cliques to total cliques. Our MCE algorithms are used in Chapter VIII as a preprocessing phase for an image processing application, which requires maximal cliques and also aims to support platform-portable execution. The content of this chapter is adopted primarily from a collaborative conference-accepted publication composed by myself, Talita Perciano, Manish Mathai, Wes Bethel, and Hank Childs [83]. As lead author, I designed and implemented the algorithm, and wrote the majority of the manuscript text. Manish Mathai helped me conduct the experiments, write the background section text, and fine-tune our experimental setup. Talita Perciano, Wes Bethel, and Hank Childs all provided very helpful guidance towards motivating the work, analyzing the experimental results, and editing the final manuscript.

The remainder of this chapter proceeds as follows. Section 1 provides a background on MCE and documents related work. Section 2 introduces our DPP-based MCE algorithm. Section 3 reviews our experimental setup. Section 4 presents the results of our suite of MCE experiments. Section 5 summarizes our findings for the dissertation question

7.1 Background and Related Work

This section defines the MCE problem and reviews existing approaches for performing MCE.

7.1.1 Maximal Clique Enumeration.

A *graph* G consists of a set of *vertices* V, some pairs of which are joined to form a set of *edges* E. A subset of vertices $C \subseteq V$ is a *clique*, or *complete subgraph*, if each vertex in C is connected to every other vertex in C via an edge. C is a *maximal clique* if its vertices are not all contained within any other larger clique in G. The size of a clique can range from zero—if there are no edges in G—to the number of vertices in V, if every vertex is connected to every other vertex (i.e., G is a complete graph). The *maximum clique* is the clique of largest size within G, and is itself maximal, since it cannot be contained within any larger-sized clique. The task of finding all maximal cliques in a graph is known as *maximal clique enumeration* (MCE). Figure 7 illustrates a graph with 6 vertices and 9 undirected edges. An application of MCE on this graph would search through 15 total cliques, of which only 3 are maximal.

The maximum number of maximal cliques possible in G is exponential in size; thus, MCE is considered an NP-Hard problem for general graphs in the worst case [106]. However, for certain sparse graph families that are encountered in practice (e.g., bipartite and planar), G typically contains only a polynomial number of cliques, and numerous algorithms have been introduced to efficiently perform MCE on real-world graphs. A brief survey of prior MCE research, including the algorithms we compare against in our study, is provided later in this section.

7.1.2 Related Work.

7.1.2.1 Visualization and Data Parallel Primitives.

While we are considering the DPP approach for a graph algorithm, there have been several similar studies for scientific visualization. In each case, they have studied a specific visualization algorithm: Maynard et al. for thresholding [94], Larsen et al. for ray-tracing [75] and unstructured volume rendering [74], Schroots and Ma for cell-projected volume rendering [132],

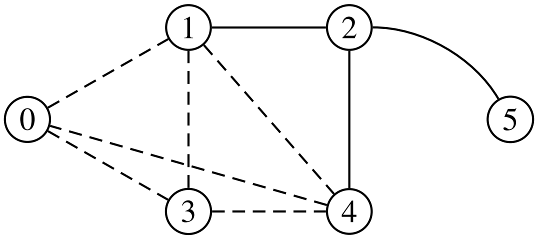


Figure 7. Undirected graph with 6 vertices and 9 edges. This graph consists of 15 total cliques, 3 of which are maximal cliques. The maximal cliques are 2-5, 1-2-4, and 0-1-3-4, the latter of which is the largest-sized clique in the graph. This maximum clique is denoted with dotted edges.

Lessley et al. for external facelist calculation [79], Lo et al. for isosurface generation [89], Widanagamaachchi et al. and Harrison et al. for connected component finding [150, 49], Carr et al. for contour tree computation [19], and Li et al. for wavelet compression [87]. Moreover, several DPP-based algorithms have been introduced for the construction of spatial search structures in the visualization domain (e.g., ray tracing), particularly for real-time use on graphics hardware. These include *k*-d trees, uniform grids, two-level grids, bounding volume hierarchies (BVH), and octrees [89, 156, 62, 59, 58, 71, 76].

Finally, our experiments make use of the VTK-m framework [109], which is the same framework used in several of these scientific visualization studies. VTK-m is effectively the unification of three predecessor visualization libraries—DAX [107], EAVL [98], and PISTON [89]—each of which were constructed on DPP with an aim to achieve portable performance across multiple many-core architectures.

7.1.2.2 Maximal Clique Enumeration. Several studies have introduced algorithms for MCE. These algorithms can be categorized along two dimensions: traversal order of clique enumeration and whether it is serial or parallel.

Serial depth-first MCE uses a backtracking search technique to recursively *expand* partial cliques with candidate vertices until maximal cliques are discovered. This process represents a search forest in which the set of vertices along a path from a root to a child constitutes a clique, and a path from a root to a leaf vertex forms a maximal clique. Upon discovering a maximal clique, the algorithm backtracks to the previous partial clique and branches into a recursive expand operation with another candidate vertex. This approach limits the size of the search space by only exploring search paths that will lead to a maximal clique.

The works in [1, 17] introduce two of the earliest serial backtracking-based algorithms for MCE; the implementation of the algorithm in [17] attained more prominence due to its simplicity and effective performance for most practical graphs. The algorithms proposed in [56, 67, 144, 77, 23, 93] build upon [17] and devise similar depth-first, tree-based search algorithms. Tomita et al. [142] optimize the clique expansion (*pivoting*) strategy of [17] to prune unnecessary subtrees of the search forest, make fewer recursive calls, and demonstrate very fast execution times in practice, as compared to [17, 144, 23, 93]. Eppstein et al. [36, 37] develop a variant of [17] that uses a degeneracy ordering of candidate vertices to order the sequence of recursive calls made at the top-most level of recursion. Then, during the inner levels of recursion, the improved pivoting strategy described in [142] is used to recurse on candidate vertices. [37] also introduces two variants of their algorithm, and propose a memory-efficient version of [142] using adjacency lists. Experimental results indicate that [37] is highly competitive with the memory-optimized [142] on large sparse graphs, and within a small constant factor on other graphs.

Distributed-memory, depth-first MCE research has also been conducted. Du et al. [35] present an approach that assigns each parallel process a disjoint subgraph of vertices and then conducts serial depth-first MCE on a subgraph; the union of outputs from each process represents the complete set of maximal cliques. Schmidt et al. [130] introduce a parallel variant of [17] that improves the process load balancing of [35] via a dynamic work-stealing scheme. In this approach, the search tree is explored in parallel among compute nodes, with unexplored search subtrees dynamically reassigned to underutilized nodes. Lu et al. [90] and Wu et al. [151] both introduce distributed parallel algorithms that first enumerate maximal, duplicate, and non-maximal cliques, then perform a post-processing phase to remove all the duplicate and non-maximal cliques. Dasari et al. [28] expand the work of [37] to a distributed, MapReduce environment, and study the performance impact of various vertex-ordering strategies, using a memory-efficient partial bit adjacency matrix to represent vertex connectivity within a partitioned subgraph. Svendsen et al. [141] present a distributed MCE algorithm that uses an enhanced load balancing scheme based on a carefully chosen ordering of vertices. In experiments with large graphs, this algorithm significantly outperformed the algorithm of [151].

Serial breadth-first MCE iteratively expands all k-cliques into $(k+1)$ cliques, enumerating maximal cliques in increasing order of size. The number of iterations is typically equal to the size of the largest maximal clique. Kose et al. [70] and Zhang et al. [154] introduce algorithms based on this approach. However, due to the large memory requirements of these algorithms, depth-first-based algorithms have attained more prevalence in recent MCE studies [130, 141].

Shared-memory breadth-first MCE on a single node has not been actively researched to the best of our knowledge. In this study, we introduce a breadth-first approach that is designed in terms of data-parallel primitives. These primitives enable MCE to be conducted in a massively-parallel fashion on shared-memory architectures, including GPU accelerators, which are designed to perform this data-parallel computation. We compare the performance of our algorithm against that of Tomita et al. [142], Eppstein et al. [37] and Svendsen et al. [141]. These studies provide suitable benchmark comparisons because they each introduce the leading MCE implementations in their respective categories: Tomita et al. and Eppstein et al. for serial depth-first MCE and Svendsen et al. for distributed-memory, depth-first MCE.

7.2 Algorithm

This section presents our new DPP-based MCE algorithm, which consists of an initialization procedure followed by the main computational algorithm. The goal of the initialization procedure is to represent the graph data in a compact format that fits within shared memory. The main computational algorithm enumerates all of the maximal cliques within this graph. The implementation of this algorithm is available online [149], for reference and reproducibility.

7.2.1 Initialization. In this phase, we construct a compact graph data structure that consists of the following four component vectors:

- I: List of vertex Ids. The contents of the list are the lower-value vertex Ids of each edge;
- C: List containing the number of edges per vertex in I;
- E: Segmented list in which each segment corresponds to a vertex v in I, and each vertex Id within a segment corresponds to an edge of v. The length of a segment is equal to the number of edges incident to its vertex;
- V: List of indices into the edge list, E, for each vertex in I. Each index specifies the start of the vertex's segment of edges.

This data structure is known as a *v-graph* [11] and it is constructed using only data-parallel operations. The compressed form of the *v-graph* in turn enables efficient data-parallel operations for our MCE algorithms.

We construct the *v-graph* as follows. Refer to algorithm 3 for pseudocode of these steps and Figure 8 for an illustration of the input and output.

(0,1)	(0,1)	(0,1)	
(0,3)	(0,3)	(0,1)	
(1,0)	(0,1)	(0,3)	(0,1)
(1,2)	(1,2)	(0,4)	(0,3)
(1,3)	(1,3)	(1,2)	(0,4)
(1,4)	(1,4)	(1,3)	(1,2)
(2,4)	(2,4)	(1,4)	(1,3)
(2,5)	(2,5)	(1,4)	(1,4)
(4,0)	(0,4)	(2,4)	(2,4)
(4,1)	(1,4)	(2,5)	(2,5)
(4,3)	(3,4)	(3,4)	(3,4)

$$I = [0\ 1\ 2\ 3]$$
$$C = [3\ 3\ 2\ 1]$$
$$V = [0\ 3\ 6\ 8]$$
$$E = [\underbrace{1\ 3\ 4}_{v_0}\ \underbrace{2\ 3\ 4}_{v_1}\ \underbrace{4\ 5}_{v_2}\ \underbrace{4}_{v_3}]$$

Input ReorderSortUnique   Output: v-graph

Figure 8. Initialization process to obtain a v-graph representation of an undirected graph. Starting with an unordered set of (possibly) directed edges, we first reorder the two vertices in each edge to ascending Id order. Second, all edges are sorted in ascending order. Third, all duplicate edges are removed, leaving unique undirected edges. These edges are then further processed to construct the output v-graph.

1. **Reorder**: Accept either an undirected or directed graph file as input; if the graph is directed, then it will be converted into an undirected form. We re-order an edge (b,a) to (a,b) if $b > a$. This maintains the ascending vertex order that is needed in our algorithms;
2. **Sort**: Invoke a data-parallel *Sort* primitive to arrange all edge pairs in ascending order (line 9 of algorithm 3). The input edges in Figure 8 provide an example of this sorted order;
3. **Unique**: Call the *Unique* data-parallel primitive to remove all duplicate edges (line 10 of algorithm 3). This step is necessary for directed graphs, which may contain bi-directional edges (a,b) and (b,a);
4. **Unzip**: Use the *Unzip* data-parallel primitive to separate the edge pairs (a_i, e_i) into two arrays, A and E, such that all of the first-index vertices, a_i, are in A and all of

Algorithm 3: Pseudocode for the construction of the v-graph data structure used in our MCE algorithm. This graph structure consists of vertex Ids (I), segmented edges (E), per-vertex indices into the edge list (V), and per-vertex edge counts (C). M is the number of input edges, N_{edges} is the number of output edges, and N_{verts} is the number of output vertices.

```
1  /*Input*/
2  Array: int edgesIn[$M$]
3  /*Output*/
4  Array: int C[$N_{verts}$], E[$N_{edges}$], I[$N_{verts}$], V[$N_{verts}$]
5  /*Local Objects*/
6  Array: int edgesOrdered[$M$], edgesSorted[$M$], edgesUndirected[$N_{edges}$],
       A[$N_{edges}$]
7  Int: $N_{edges}$, $N_{verts}$
8  edgesOrdered←Reorder(edgesIn)
9  edgesSorted←Sort(edgesOrdered)
10 edgesUndirected←Unique(edgesSorted)
11 A, E←Unzip(edgesUndirected)
12 $N_{edges} \leftarrow |E|$
13 C, I←ReduceByKey(A, $\vec{\mathbf{1}}$)
14 $N_{verts} \leftarrow |I|$
15 V←ExclusiveScan(C)
16 //Continue with Algorithm 4 after returning.
17 return (C, E, I, V)
```

the second-index vertices, e_i, are in E (line 11 of algorithm 3). For example, using the edges from Figure 8, we can create the following A and E arrays:

$$\begin{bmatrix} 0\,0\,0\,1\,1\,1\,2\,2\,3 \\ 1\,3\,4\,2\,3\,4\,4\,5\,4 \end{bmatrix} \xrightarrow{Unzip} \begin{matrix} A: \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 3 \end{bmatrix} \\ \\ E: \begin{bmatrix} 1 & 3 & 4 & 2 & 3 & 4 & 4 & 5 & 4 \end{bmatrix} \end{matrix}$$

The array E represents the edge list in our *v-graph* structure;

5. **Reduce**: Use the *ReduceByKey* data-parallel primitive to compute the edge count for each vertex (line 13 of algorithm 3). Using the arrays A and E from step 3, this operation counts the number of adjacent edges from E that are associated with each unique vertex in A. The resulting output arrays represent the lists I and C in our

v-graph structure:

$$I: \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}$$

$$C: \begin{bmatrix} 3 & 3 & 2 & 1 \end{bmatrix}$$

6. **Scan**: Run the *ExclusiveScan* data-parallel operation on the edge counts array, C, to obtain indices into the edge list, E, for each entry in I (line 15 of algorithm 3). This list of indices represents the list V in our *v-graph* (see Figure 8). In our running example, vertex 0 has 3 edges and vertex 1 has 3 edges, representing index segments 0-2 and 3-5 in E, respectively. Thus, vertex 0 and vertex 1 will have index values of 0 and 3, respectively:

$$C : [3\,3\,2\,1] \xrightarrow{ExclusiveScan} V : [0\,3\,6\,8]$$

7.2.2 Hashing-Based Algorithm.

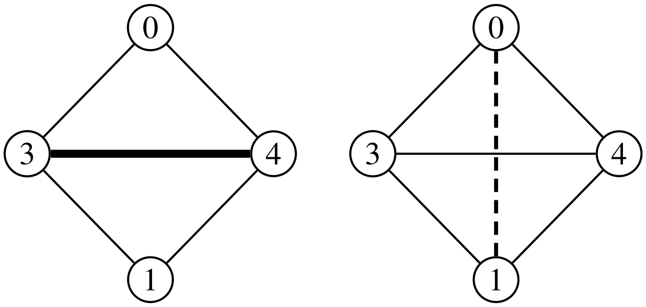
We now describe our hashing-based algorithm to perform maximal clique enumeration, which comprises the main computational work. This algorithm takes the *v-graph* from the initialization phase as input. In the following subsections we provide an overview of the algorithm, along with a more detailed, step-by-step account of the primary data-parallel operations.

7.2.2.1 Algorithm Overview.

We perform MCE via a bottom-up scheme that uses multiple iterations, each consisting of a sequence of data-parallel operations. During the first iteration, all 2-cliques(edges) are expanded into zero or more 3-cliques and then tested for maximality. During the second iteration, all of these new 3-cliques are expanded into zero or more 4-cliques and then tested for maximality, so on and so forth until there are no newly-expanded cliques. The number of iterations is equal to the size of the *maximum* clique, which itself is *maximal* and cannot be expanded into a larger clique. Figure 9 presents the progression of clique expansion for the example graph in Figure 7.

2-cliques:	0-1	0-3	0-4	1-2	1-3	1-4	2-4	**2-5**	3-4
3-cliques:	0-1-3	0-1-4	0-3-4	**1-2-4**	1-3-4				
4-cliques:	**0-1-3-4**								

Figure 9. Clique expansion process for an example undirected graph. In the first iteration, only 2-cliques (edge pairs) are considered. Then, these cliques are expanded into larger 3-cliques. The 4-clique in the final iteration cannot be expanded further since it is maximal; this clique also cannot be expanded further because it is the maximum-sized clique. All maximal cliques are denoted in boxes with bold font.



(a) 3-cliques: 0-3-4, 1-3-4 (b) 4-clique: 0-1-3-4

Figure 10. Example of clique expansion. As shown in (a), the set of four vertices, 0-1-3-4, is composed of two 3-cliques, 0-3-4 and 1-3-4. Both of these cliques share a common 2-clique, 3-4, which is highlighted in bold. If $(0,1)$ is an edge in the graph (dotted line), as shown in (b), then 0-1-3-4 is a 4-clique.

During this process, we assess whether a given *k*-clique is a subset of one or more larger $(k+1)$-cliques. If so, then the *k*-clique is marked as non-maximal and the new $(k+1)$-cliques are stored for the next iteration; otherwise, the *k*-clique is marked as maximal and discarded from further computation.

In order to determine whether a *k*-clique is contained within a larger clique, we use a hashing scheme that searches through a hash table of cliques for another *k*-clique with the same hash value. These matching cliques share common vertices and can be merged into a larger clique if certain criteria are met. Thus, hashing is an important element to our algorithm. Figure 10 illustrates the clique merging process between two different *k*-cliques.

7.2.2.2 Algorithm Details.

Within each iteration, our MCE algorithm consists of three phases: dynamic hash table construction, clique expansion, and clique maximality testing. These phases are conducted in a sequence and invoke only data-parallel operations; refer to algorithm 4 for pseudocode of these phases and operations. In the following description, the 3-cliques from Figure 9 are used as a running example. We start from iteration 3 with the linear array

$$cliques = [\text{0-1-4 } \text{ 0-3-4 } \text{ 1-2-4 } \text{ 1-3-4 } \text{ 0-1-3}].$$

of length $(k=3) \times (numCliques = 5) = 15$.

Dynamic Hash Table Construction: Our algorithm uses both sorting and hashing as integral index-based search components. We discuss the following operations that are used to construct a non-persistent hash table into which the cliques are hashed and queried.

First, each clique is hashed to an integer (line 21 of algorithm 4). This is done using the FNV-1a hash function [40], h, and taking the result modulo the number of cliques. Further, only the clique's last $k-1$ vertex indices are hashed. Only the last $(k-1)$ vertices are hashed because we just need to search (via a hash table) for matching $(k-1)$-cliques to form a new $(k+1)$-clique. For example, cliques 0-3-4 and 1-3-4 both hash their last two vertices to a common index, i.e., $h(\text{3-4})$, and can combine to form 0-1-3-4, since leading vertices 0 and 1 are connected (see Figure 10).

Next, we allocate an array, *hashTable*, of the same size as *cliques*, into which the cliques will be rearranged (permuted) in order of hash value. In our example, there are 5 cliques, each with an Id in $cliqueIds = [0\ 1\ 2\ 3\ 4]$. After applying the hash operation, these cliques have the hash values, $hashes = [1\ 0\ 4\ 0\ 1]$. Sorting *cliqueIds* in order of hash value (*hashes*), we obtain $sortedIds = [1\ 3\ 0\ 4\ 2]$ and $sortedHashes = [0\ 0\ 1\ 1\ 4]$ (line 22 of algorithm 4). A *ReduceByKey* operation (line 24 of algorithm 4) computes a count for each unique hash value in *hashes*: $unique = [0\ 1\ 4]$ and $count = [2\ 2\ 1]$. A series of *Scan*, *Map*, and *Scatter* primitives are then employed on *cliqueIds* and *counts* to construct offset index arrays (of length *numCliques*) into *hashTable*, denoted as *cliqueStarts* and *chainStarts*, respectively (lines 26 and 27 algorithm 4). Permuting *cliques* by *sortedIds* (line 23 of algorithm 4), we obtain

$$\begin{aligned}
hashTable &= [\underbrace{\text{0-3-4}\ \ \text{1-3-4}}_{Chain_0}\ \underbrace{\text{0-1-4}\ \ \text{0-1-3}}_{Chain_1}\ \underbrace{\text{1-2-4}}_{Chain_2}] \\
tableIndex &= [\underbrace{0\ 1\ 2}_{Clique_0}\ \underbrace{3\ 4\ 5}_{Clique_1}\ \underbrace{6\ 7\ 8}_{Clique_2}\ \underbrace{9\ 10\ 11}_{Clique_3}\underbrace{12\ 13\ 14}_{Clique_4}] \\
cliqueIds &= [0\ 1\ 2\ 3\ 4] \\
chainIds &= [0\ 0\ 1\ 1\ 2] \\
cliqueStarts &= [0\ 3\ 6\ 9\ 12] \\
chainStarts &= [0\ 0\ 6\ 6\ 12],
\end{aligned}$$

with three *chains* of contiguous cliques, each sharing the same hash value. Since the cliques within a chain are not necessarily in sorted order, the chain must be *probed* sequentially using a constant number of lookups to find the clique of interest.

This probing is also necessary since different cliques may possess the same hash value, resulting in *collisions* in the chain. For instance, cliques 0-1-3 and 0-1-4 both hash to index $h(\text{1-3}) = h(\text{1-4}) = 1$, creating a collision in $Chain_1$. Thus, the hash function is important, as good function choices help minimize collisions, while poor choices create more collisions and, thus, more sequential lookups.

This combined sorting- and hashing-based search technique effectively simulates a hash table, since colliding cliques with the same hash value are grouped, or bucketed, together and traditional open-addressing querying is supported via linear probing within a bucket.

Clique Expansion: Next, a two-step routine is performed to identify and retrieve all valid $(k+1)$-cliques for each k-clique in $hashTable$. The first step focuses on determining the sizes of output arrays and the second step focuses on allocating and populating these arrays.

In the first step, a *Map* primitive computes and returns the number of $(k+1)$-cliques into which a k-clique can be expanded (line 29 of algorithm 4). The first step works as follows. For a given k-clique, i, at $cliqueStarts[i]$, we locate its chain at $chainStarts[i]$ and iterate through the chain, searching for another k-clique, j, with (a) a larger leading vertex Id and (b) the same ending $(k-1)$ vertices; these two criteria are needed to generate a larger clique and avoid duplicates (see Theorem A.0.1 and Theorem A.0.3). For each matching clique j in the chain, we perform a binary search over the adjacent edges of i in the v-graph edge list E to determine whether

Algorithm 4: Pseudocode for our DPP-based MCE algorithm.

```
1  /*Input from Algorithm 3*/
2  v-graph: int C[$N_{verts}$], E[$N_{edges}$], I[$N_{verts}$], V[$N_{verts}$]
3  /*Output*/
4  Array: int[$N_{maximal} \times N_{maximalVerts}$]: maxCliques
5  /*Local Objects*/
6  int: $N_{cliques}, N_{newCliques}, N_{chains}, N_{maximal}, N_{maximalVerts}, iter$
7  Array: int[$N_{cliques}$]: cliqueStarts, cliqueSizes, cliqueIds, sortedCliqueIds,
     newCliqueCounts, scanNew, writeLocations
8  Array: float[$N_{cliques}$]: hashes, sortedHashes
9  Array: int[$N_{newCliques}$]: newCliqueStarts
10 Array: int[$N_{cliques} \times (iter+1)$]: newCliques
11 Array: int[$N_{cliques} \times iter$]: cliques, isMaximal, hashTable
12 Array: int[$N_{newCliques} \times (iter-1)$]: repCliqueIds, repCliqueStarts, localIndices,
     vertToOmit
13 Array: int[$N_{chains}$]: uniqueHashes, chainStarts, chainSizes, scanChainSizes
14 $iter \leftarrow 2$
15 cliques←Get2-Cliques(E)
16 $N_{cliques} \leftarrow N_{edges}$
```

```
while $N_{cliques} > 0$ do
    cliqueStarts$\leftarrow [iter \times i], 0 \leq i < N_{cliques}$;
    cliqueSizes$\leftarrow [iter_i], 0 \leq i < N_{cliques}$;
    cliqueIds$\leftarrow [i], 0 \leq i < N_{cliques}$;
    hashes$\leftarrow$ComputeHash(cliques, cliqueStarts, cliqueSizes);
    sortedHashes, sortedIds $\leftarrow$SortByKey(hashes, cliqueIds);
    hashTable$\leftarrow$Permute(sortedIds, cliques);
    uniqueHashes, chainSizes $\leftarrow$ReduceByKey(sortedHashes, $\vec{\mathbf{1}}$);
    $N_{chains} \leftarrow |$uniqueHashes$|$;
    scanChainSizes$\leftarrow$ScanExclusive(chainSizes);
    chainStarts$\leftarrow [$scanChainSizes$[i] \times iter]$;
    isMaximal$\leftarrow \vec{\mathbf{1}}$;
    newCliqueCounts, isMaximal $\leftarrow$FindCliques(v-graph, $iter$, cliqueStarts,
      chainStarts, chainSizes, hashTable, isMaximal);
    $N_{newCliques}$, scanNew$\leftarrow$ScanExclusive(newCliqueCounts);
    writeLocations$\leftarrow$Multiply(scanNew, $iter + 1$);
    newCliques$\leftarrow \vec{\mathbf{0}}$;
    newCliques$\leftarrow$GetCliques(v-graph, $iter$, writeLocations, chainStarts,
      chainSizes, hashTable, newCliques);
    repCliqueIds$\leftarrow [i_0 \ldots i_{iter-2}], 0 \leq i < N_{newCliques}$;
    newCliqueStarts$\leftarrow [iter \times i], 0 \leq i < N_{newCliques}$;
    repCliqueStarts$\leftarrow$Gather(repCliqueIds, newCliqueStarts);
    localIndices, vertToOmit$\leftarrow$Modulus($iter - 1$, repCliqueIds);
    isMaximal$\leftarrow$TestForMaximal(repCliqueIds, repCliqueStarts, $iter - 1$,
      localIndices, vertToOmit, chainStarts, chainSizes, hashTable, isMaximal,
      newCliques);
    maxCliques = maxCliques + Compact(hashTable, isMaximal, IsIntValue(1));
    $N_{cliques} \leftarrow N_{newCliques}$;
    $iter \leftarrow iter + 1$;
end
return (maxCliques)
```

the leading vertices of i and j are connected. If so, then, by Theorem A.0.1, cliques i and j can be expanded into a larger $(k+1)$-clique consisting of the two leading vertices and the shared $(k-1)$ vertices, in ascending order. The total number of expanded cliques for i is returned. In our example, this routine returns a counts array, *newCliqueCounts* $= [1\ 0\ 0\ 0\ 0]$, indicating that only clique 0-3-4 could be expanded into a new 4-clique; Figure 10 illustrates the generation of this 4-clique.

In the second step, an inclusive *Scan* primitive is invoked on *newCliqueCounts* to compute the sum of the clique counts, *numNewCliques* (line 30 of algorithm 4). The second step works as follows. This sum is used to allocate a new array, *cliques*, of size *numNewcliques* $\cdot (k+1)$ to store all of the $(k+1)$-cliques, along with a new offset index array with increments of $(k+1)$. With these arrays, we invoke a data-parallel *Map* operation that is identical to the *Map* operation of the first step, except that, upon discovery of a new $(k+1)$-clique, we write the clique out to its location in *cliques* (using the offset array), instead of incrementing a *newCliques* counter (line 32 of algorithm 4). For the running example, the new *cliques* array consists of the single 4-clique, 0-1-3-4.

Clique Maximality Test: Finally, we assess whether each k-clique is maximal or not. Prior to **Clique Expansion**, a bit array, *isMaximal*, of length *numCliques*, is initialized with all 1s. During the first step of **Clique Expansion**, if a clique i merged with one or more cliques j, then they all are encompassed by a larger clique and are not maximal; thus, we set $isMaximal[i] = isMaximal[j] = 0$, for all j. Since each $(k+1)$-clique includes $(k+1)$ distinct k-cliques—two of which are the ones that formed the $(k+1)$-clique—we must ensure that the remaining $k-1$ k-cliques are marked as non-maximal with a value of 0 in *isMaximal*. In our example, the 4-clique 0-1-3-4 is composed of 4 different 3-cliques: 1-3-4, 0-3-4, 0-1-4, and 0-1-3. The first two were already marked as non-maximal, but the remaining two are non-maximal as well, and need to be marked as so in this phase. Our approach for marking these remaining cliques as non-maximal is as follows.

First, we use a custom modulus map operator (line 34 of algorithm 4) to construct, in parallel, an array of length *numNewCliques* $\times (k-1)$, with $(k-1)$ local indices per new $(k+1)$-clique: $[2_0 \ldots k_0 \ldots 2_{numCliques-1} \ldots k_{numCliques-1}]$. Then, we parallelize over this index array via a *Map* operation (line 35 of algorithm 4) that, given an index $2 \leq i \leq k$ and corresponding clique $0 \leq t \leq numCliques - 1$, determines whether the k-clique formed by omitting vertex $t[i]$ is maximal or not. If the k-clique is discovered in $hashTable$ (using the same hashing approach as in **Clique Expansion**), then it is marked as 0 in $isMaximal$. A *Compact* primitive then removes all k-cliques in $hashTable$ that have $isMaximal =$ 0, leaving only the maximal cliques, which are appended in an auxiliary array (line 36 of algorithm 4).

The algorithm terminates when $numNewCliques = 0$ (line 17 of algorithm 4). The generated $cliques$ array of new $(k+1)$-cliques becomes the starting array (line 37 of algorithm 4) for the next iteration (line 38 of algorithm 4), if the termination condition is not met.

7.3 Experimental Overview

We assess the performance of our MCE algorithm in two phases, using a collection of benchmark input graphs and both CPU and GPU systems. In the first phase, we run our algorithm–denoted as $Hashing$—on a CPU platform and compare its performance with three state-of-the-art MCE algorithms—*Tomita* [142], $Eppstein$ [37], and *Svendsen* [141]. In the second phase, we evaluate portable performance by testing $Hashing$ on a GPU platform and comparing the runtime performance with that of the CPU platform, using a common set of benchmark graphs. The following subsections describe our software implementation, hardware platforms, and input graph datasets.

7.3.1 Software Implementation.

Both of our MCE algorithms are implemented using the platform-portable VTK-m toolkit [149], which supports fine-grained concurrency for data analysis and scientific visualization algorithms. With VTK-m, a developer chooses data parallel primitives to employ, and then customizes

Graph	Collection	V	E	Max_{size}	$Cliques_{max}$	$Cliques_{all}$	Max_{ratio}
amazon0601	Stanford	403,394	3,387,388	11	1,023,572	18,043,744	0.06
cit-Patents	Stanford	3,774,768	16,518,947	11	14,787,032	36,180,638	0.41
email-Enron	Stanford	36,692	183,831	20	226,859	107,218,609	< 0.01
loc-Gowalla	Stanford	196,591	950,327	29	1,212,679	1,732,143,035	$\ll 0.01$
soc-wiki-Vote	Stanford	7,115	103,689	17	459,002	41,792,503	0.01
roadNet-CA	Stanford	1,965,206	2,766,607	4	2,537,996	2,887,325	0.88
brock200-2	DIMACS	200	9,876	12	431,586	6,292,399	0.07
hamming6-4	DIMACS	64	704	4	464	1,904	0.24
MANNa9	DIMACS	45	918	16	590,887	160,252,675	< 0.01
p_hat300-1	DIMACS	300	10,933	8	58,176	367,022	0.16
UG100k.003	DIMACS	100,000	14,997,901	4	10,589,956	19,506,096	0.54

Table 12. Statistics for a subset of the test graphs used in this MCE study. Graphs are either from the Stanford Large Network Dataset Collection [137] or the DIMACS Challenge data set [32]. V is the number of graph vertices, E is the number of edges, Max_{size} is the size of the largest clique, $Cliques_{max}$ is the number of maximal cliques, $Cliques_{all}$ is the total number of cliques, and Max_{ratio} is the ratio of $Cliques_{max}$ to $Cliques_{all}$.

those primitives with functors of C++-compliant code. This code is then used to create architecture-specific code for architectures of interest, i.e., CUDA code for NVIDIA GPUs and Threading Building Blocks (TBB) code for Intel CPUs. Thus, by refactoring an algorithm to be composed of VTK-m data-parallel primitives, it only needs to be written once to work efficiently on multiple platforms. In our experiments, the TBB configuration of VTK-m was compiled using the gcc compiler, the CUDA configuration using the nvcc compiler, and the VTK-m index integer (vtkm::Id) size was set to 64 bits. The implementation of this algorithm is available online [149], for reference and reproducibility.

7.3.2 Test Platforms. We conducted our experiments on the following two CPU and GPU platforms:

- CPU: A 16-core machine running 2 nodes, each with a 3.2 GHz Intel Xeon(R) E5-2667v3 CPU with 8 cores. This machine contains 256GB DDR4 RAM memory.

All the CPU experiments use the Intel TBB multi-threading library for many-core parallelism.

- GPU: An NVIDIA Tesla K40 Accelerator with 2880 processor cores, 12 GB memory, and 288 GB/sec memory bandwidth. Each core has a base frequency of 745 MHz, while the GDDR5 memory runs at a base frequency of 3 GHz. All GPU experiments use NVIDIA CUDA V6.5.

7.3.3 Test Data Sets. We applied our algorithm to a selected set of benchmark and real-world graphs from the DIMACS Challenge [32] and Stanford Large Network Dataset collections [137]. Table 12 lists a subset of these test graphs, along with their statistics pertaining to topology and clique enumeration. For each graph, we specify the number of vertices (V), edges (E), maximum clique size (Max_{size}), number of maximal cliques ($Cliques_{max}$), number of total cliques ($Cliques_{all}$), and ratio of maximal cliques to total cliques (Max_{ratio}). The DIMACS Challenge data set includes a variety of benchmark instances of randomly-generated and topologically-challenging graphs, ranging in size and connectivity. The Stanford Large Network Data Collection contains a broad array of real-world directed and undirected graphs from social networks, web graphs, road networks, and autonomous systems, to name a few.

7.4 Results

In this section, we present the results of our set of MCE experiments, which consists of two phases: CPU and GPU.

7.4.1 Phase 1: CPU. This phase assesses the performance of our *Hashing* algorithm on a CPU architecture with the set of graphs listed in Table 13 and Table 14.

For each graph in Table 13, the total runtime (in seconds) of *Hashing* is compared with that of *Tomita* and *Eppstein*, two serial algorithms that have demonstrated state-of-the-art performance for MCE. The set of graphs used for comparison was adopted

Graph	Tomita	Eppstein	Hashing
amazon0601	**	3.59	**1.69**
cit-Patents	**	28.56	**3.27**
email-EuAll	**	1.25	2.24
email-Enron	31.96	0.90	17.91
roadNet-CA	**	2.00	**0.27**
roadNet-PA	**	1.09	**0.16**
roadNet-TX	**	1.35	**0.19**
brock200-2	0.55	1.22	0.71
hamming6-4	< 0.01	< 0.01	< **0.01**
johnson8-4-4	0.13	0.24	0.50
johnson16-2-4	5.97	12.17	**5.10**
MANNa9	0.44	0.53	27.74
p_hat300-1	0.07	0.15	**0.07**
soc-wiki-Vote	0.96	1.14	6.14
keller4	5.98	11.53	7.22

Table 13. Total CPU execution times (sec) for our *Hashing* MCE algorithm, as compared to the serial *Tomita* and *Eppstein* algorithms, over a set of common test graphs. Results with double asterisk symbols indicate that the graph could not be processed due to memory limitations. Results in bold indicate that *Hashing* achieved the fastest execution time for that particular graph.

Graph	*Svendsen*	*Hashing*
cit-Patents	109	**3.27**
loc-Gowalla	112	545.25
UG100k.003	353	**5.39**
UG1k.30	129	**11.10**

Table 14. Total CPU execution times (sec) for our *Hashing* MCE algorithm, as compared to the distributed-memory *Svendsen* algorithm, over a set of common test graphs. Results in bold indicate that *Hashing* achieved the fastest execution time for that particular graph.

from the paper of Eppstein et. al [37], which compared the CPU results of three newly-introduced MCE algorithms with that of the *Tomita* algorithm. In this phase, we report the best total runtime among these three algorithms as *Eppstein*. Moreover, we only test on those graphs from [37] that are contained with the DIMACS Challenge and Stanford Large Network Data collections. Among these graphs, 9 were omitted from the comparison because our *Hashing* algorithm exceeded available shared memory on our single-node CPU system (approximately 256GB). Each of these graphs has a very large number of non-maximal cliques relative to maximal cliques. Thus, most of these non-maximal cliques are progressively expanded and passed on to the next iteration of our algorithm, increasing the computational workload and storage requirements. Reducing our memory needs and formalizing the graph properties that lead to a high memory consumption by our algorithm will be investigated in future work.

From Table 13, we observe that *Hashing* performed comparably or better on more than half—8 out of 15—of the test graphs. Using the graph statistics from Table 12, it is apparent that our algorithm performs best on graphs with a high ratio of maximal cliques to total cliques, Max_{ratio}. This is due to the fact that, upon identification, maximal cliques are discarded from further computation in our algorithm. So, the larger the number of maximal cliques, the smaller the amount of computation and memory accesses that will need to be performed. $Tomita$ and $Eppstein$ do not perform as well on these types of graphs due to the extra sequential recursive branching and storage of intermediary cliques that is needed to discover a large number of maximal cliques. From Table 13 we see that $Tomita$ exceeded the available shared memory of its CPU system (approximately 3GB) for the majority of the graphs on which we possess the faster runtime.

Next, we compare *Hashing* to the CPU-based distributed-memory MCE algorithm of Svendsen et al. [141], which we refer to as *Svendsen*. We use the set of 12 test graphs from [141], 10 of which are from the Stanford Large Network Data Collection and 2 of which are from the DIMACS Challenge collection. As can be seen in Table 14, we attain significantly better total runtimes for 3 of the graphs. Each of these graphs have high values of Max_{ratio}, corroborating the findings from the CPU experiment of Table 13. For the remaining 9 graphs, one completed in a very slow runtime (loc-Gowalla) and 8 exceeded available shared memory. We do not report the graphs that failed to finish

Graph	*Tomita-Eppstein*	*Hashing*-CPU	*Hashing*-GPU
amazon0601	3.59	1.69	**0.86**
email-Enron	0.90	17.91	15.56
email-EuAll	1.25	2.24	1.52
roadNet-CA	2.00	0.27	**0.17**
roadNet-PA	1.09	0.16	**0.11**
roadNet-TX	1.35	0.19	**0.13**
brock200-2	0.55	0.71	**0.45**
p_hat300-1	0.07	0.07	0.09
soc-wiki-Vote	0.96	6.14	4.78

Table 15. Total GPU execution times (sec) for our *Hashing* MCE algorithm over a set of test graphs. For comparison, the best execution time between *Tomita* and *Eppstein* is listed, along with the CPU execution time of *Hashing*. Results in bold indicate that *Hashing*-GPU attained the fastest execution time for that particular graph.

processing due to insufficient memory; each of these graphs have low values of Max_{ratio}. The loc-Gowalla graph just fits within available device memory, but possesses a low Max_{ratio} (see Table 12), leading to the significantly slower runtime than *Svendsen*.

7.4.2 Phase 2: GPU. Next, we demonstrate and assess the portable performance of *Hashing* by running it on a GPU architecture, using the graphs from Table 15. Each GPU time is compared to both the *Hashing* CPU time and the best time between the *Tomita* and *Eppstein* algorithms. From Table 15 we observe that, for 8 of the 9 graphs, *Hashing* GPU achieves a speedup over the CPU. Further, for 5 of these 8 graphs, *Hashing* GPU performs better than both *Hashing* CPU and *Tomita*/*Eppstein*. These speedups demonstrate the ability of a GPU architecture to utilize the highly-parallel design of our algorithm, which consists of many fine-grained and compute-heavy data-parallel operations. Moreover, this experiment demonstrates the portable performance of our algorithm, as we achieved improved execution times without having to write custom, optimized GPU functions within our algorithm; the same high-level algorithm was used for both the CPU and GPU experiments.

7.5 Conclusion

This chapter contributes a novel DPP-based algorithm for the MCE graph analysis application. As a major component of this algorithm, we develop an index-based search solution to a sub-routine that finds pairs of $(k-1)$-cliques to merge into larger $(k+1)$-cliques. Our solution for this routine involves a combined sorting- and hashing-based technique that simulates a non-persistent hash table for looking up matching $(k-1)$-cliques. While this technique is not one of the hashing-based techniques introduced in Chapters III and IV, we discovered that it was necessary in order to obtain best performance for the iterative merging routine, as neither sorting nor hashing can solve the routine alone. Sorting the cliques into buckets in order of hash function value avoids building a new hash table of cliques each algorithm iteration, which, for hundreds of millions of integer values, was shown in Chapter VI to be a slower operation than sorting. The organization of cliques into hash table-like buckets then enables a direct query lookup into a bucket for a matching clique entry.

Our MCE algorithm is designed entirely in terms of DPP and tested on both CPU and GPU devices, demonstrating platform-portability across many-core systems. Compared to state-of-the-art MCE implementations, our algorithm achieves competitive to leading runtime performance on large real-world benchmark graphs with a high ratio of maximal cliques to total cliques. Based on these findings, we believe that our combined sorting- and hashing-based solution is the best index-based search technique for this particular MCE analysis algorithm on diverse many-core systems. These findings inform our dissertation question and will be further synthesized in Chapter X.

CHAPTER VIII

GRAPH-BASED IMAGE SEGMENTATION

In this chapter, we assess the viability of DPP-based design patterns for a graph-based image processing task that consists of index-based search routines, including our DPP-based MCE algorithm from Chapter VII. In particular, we develop a new data-parallel algorithm, DPP-PMRF, for performing image segmentation using Markov Random Fields (MRFs), which are a form of probabilistic graphical model (PGM). For a collection of geological image data, we demonstrate single-node, platform-portable image segmentation runtime performance that exceeds that of a state-of-the-art reference algorithm.

The content of this chapter is adopted primarily from a collaborative conference-accepted publication composed by myself, Talita Perciano, Colleen Heinemann, David Camp, Wes Bethel, and Hank Childs [82]. As lead author, I designed and implemented our algorithm and wrote a significant portion of the manuscript text. Colleen Heinemann conducted experiments for the reference algorithm, generated plots for the experimental results, and contributed to the text for the experimental overview. Talita Perciano and Wes Bethel contributed text related to the motivation, background, image data descriptions, and concluding remarks. Talita also performed the verification tests for our segmentation output and composed the section text related to this. David Camp and Hank Childs both provided valuable guidance towards motivating the work and designing our experiments.

The remainder of this chapter proceeds as follows. Section 1 provides a background and survey of related work on MRF-based image segmentation. Section 2 introduces our DPP-PMRF algorithm. Section 3 reviews our experimental setup and presents the results and analysis of the experiments. Section 4 summarizes our findings for the dissertation question

8.1 Background

In this section, we first provide a background on image segmentation and its existing graph-based approaches using MRFs. Then, we summarize related work towards data-parallelism and platform-portable performance in graph-based methods.

8.1.1 MRF-based Image Segmentation.

Image segmentation is a compute-intensive task, and is a key component of multi-stage scientific analysis pipelines, particularly those that work with large-scale image-based data obtained by experiments and advanced instruments, such as the X-ray imaging devices located at the Advanced Light Source at the Lawrence Berkeley National Lab. As such instruments continually update in spatial and spectral resolution, there is an increasing need for high-throughput processing of large collections of 2D and 3D image data for use in time-critical activities such as experiment optimization and tuning [10]. Our work here is motivated by the need for image analysis tools that perform well on modern platforms, and that are expected to be portable to next-generation hardware.

The process of segmenting an image involves separating various phases or components from the picture using photometric information and/or relationships between pixels/regions representing a scene. This essential step in an image analysis pipeline has been given great attention recently when studying experimental data [123]. There are several different types of image segmentation algorithms, which can be divided into categories such as: threshold-based, region-based, edge-based, clustering-based, graph-based and learning-based techniques. Of these, the graph- and learning-based methods tend to present the highest accuracy, but also the highest computational cost.

Graph-based methods are well-suited for image segmentation tasks due to their ability to use contextual information contained in the image, i.e., relationships among pixels and/or regions. The probabilistic graphical model (PGM) known as a

Markov Random Field (MRF) [86] is an example of one such method. MRFs represent discrete data by modeling neighborhood relationships, thereby consolidating structure representation for image analysis [85].

An image segmentation problem can be formulated using an MRF model on a graph G, where the segmented image is obtained through an optimization process to find the best labeling of the graph. The graph $G(V,E)$ is constructed from an input image, where V is the set of nodes and E is the set of edges. Each node V_i represents a region (set of pixels) and two nodes, V_i and V_j, are connected by an edge if their corresponding regions share a boundary.

In an MRF model, the optimization process uses a global energy function to find the best solution to a similarity problem, such as the best pixel space partition. This energy function consists of a data term and a smoothness term. For image segmentation, we use the mean of the intensity values of a region as the data term. The smoothness term takes into account the similarity between regions. The goal is to find the best labeling for the regions, so that the similarity between two regions with the same labels is optimal for all pixels [92].

Given an image represented by $\mathbf{y} = (y_1, \ldots, y_N)$, where each y_i is a region, we want a configuration of labels $\mathbf{x} = (x_1, \ldots, x_N)$ where $x_i \in L$ and L is the set of all possible labels, $L = \{0, 1, 2, \ldots, M\}$. The MAP criterion [86] states that one wants to find a labeling $\mathbf{x}^*$ that satisfies $\mathbf{x}^* = \underset{x}{\operatorname{argmax}}\{P(\mathbf{y}|\mathbf{x},\Theta)P(\mathbf{x})\}$, which can be rewritten in terms of the energies [86] as $\mathbf{x}^* = \underset{x}{\operatorname{argmin}}\{U(\mathbf{y}|\mathbf{x},\Theta) + U(\mathbf{x})\}$ (please refer to [124] for details regarding the prior and likelihood energies used in our approach).

Despite their high accuracy, MRF optimization algorithms have high computational complexity (NP-hard). Strategies for overcoming the complexity, such as graph-cut techniques, are often restricted to specific types of models (first-order

MRFs) [69] and energy functions (regular or submodular) [69]. In order to circumvent such drawbacks, recent works [96, 97] have proposed theoretical foundations for distributed parameter estimation in MRF. These approaches make use of a composite likelihood, which enables parallel solutions to sub problems. Under general conditions on the composite likelihood factorizations, the distributed estimators are proven to be consistent. The Linear and Parallel (LAP) [104] algorithm parallelizes naturally over cliques and, for graphs of bounded degree, its complexity is linear in the number of cliques. It is fully parallel and, for log-linear models, it is also data efficient. It requires only the local statistics of the data, i.e., considering only pixel values of local neighborhoods, to estimate parameters.

Perciano *et al.* [124] describe a graph-based model, referred to as Parallel Markov Random Fields (PMRF), which exploits MRFs to segment images. Both the optimization and parameter estimation processes are parallelized using the LAP method. In the work we present here, we use an OpenMP-based PMRF implementation as the "reference implementation," and reformulate this method using DPPs. We study the viability of using DPPs as an alternative way to formulate an implementation to this challenging graph-based optimization problem, and compare shared-memory scalability of the DPP and reference implementation.

8.1.2 Performance and Portability in Graph-based Methods.

The idea of formulating algorithms as sequences of highly optimized kernels, or motifs, is not new: this approach has formed the basis for nearly all numerical library and high performance simulation work going back almost 40 years, to the early implementations of LINPACK [34]. Over the years, several different highly optimized and parallel-capable linear algebra libraries have emerged, which serve as the basis for constructing a diverse collection of scientific computing applications. Such libraries include ScaLAPACK [24], BLASFEO (Basic Linear Algebra Subroutines for Embedded Optimization) [42] and MAGMA (Matrix Algebra on GPU and Multicore Architectures) [143], to name a few.

The concept of using combinations of highly optimized building blocks has served as guiding design principle for many works focusing on high performance graph processing tools. The Boost Graph Library (BGL) [135] is a seminal implementation of data structures and methods for operating on graphs. The Multi-thread Graph Library (MTGL) [9] adapts and focuses BGL design principles for use on multithreaded architectures, where latencies associated with irregular memory access are accommodated by increasing the thread count to fully utilize memory bandwidth. More recent works, such as CombBLAS [18] and GraphBLAS [63], provide the means to implement graph-based algorithms as sequences of linear algebra operations, with special attention to the irregular access patterns of sparse vector and matrix operations, and on distributed-memory platforms. GraphMat [139] provides the means to write vertex programs and map them to generalized sparse matrix vector multiplication operations that are highly optimized for multi-core processors. The STAPL parallel graph library [50] focuses more on the data structures and infrastructure for supporting distributed computations that implement computational patterns (e.g., map-reduce) for user-written graph algorithms.

Like many of these previous works, we are also examining the concept of platform portable graph algorithm construction using optimized building blocks. Compared to these previous works, our focus is narrower in terms of graph algorithm (MRF optimization) and building block (DPP). An open question, which is outside the scope of this work, is whether or not the MRF optimization problem can be recast in a way that leverages platform-portable and parallel implementations such as GraphBLAS [18, 63], which accelerates graph operations by recasting computations as sparse linear algebra problems. Unlike many graph problems, the MRF optimization problem here is not a

Algorithm 5: Pseudocode of the benchmark Parallel MRF (PMRF) algorithm for performing graph-based image segmentation using Markov Random Fields.

```
Require: Original image, oversegmentation, number of output labels
Ensure: Segmented image and estimated parameters
Initialize parameters and labels randomly
Create graph from oversegmentation
Calculate and initialize $k$-neighborhoods from graph
for each EM iteration do
    for each neighborhood of the subgraph do
        Compute MAP estimation
        Update parameters
    end
end
Update labels
return (Segment labels, Label Parameters)
```

sparse-data problem: as part of the problem setup, the graphical model is represented internally, in the reference implementation, in dense array form, and then the energy optimization computations are performed on densely packed arrays. Our DPP-PMRF implementation recasts these dense-memory computations using DPPs, which are highly amenable to vectorization. A focal point of this study is to better understand the performance comparison and characterization between a reference implementation and one derived from DPPs.

8.2 Algorithm Design

This section introduces our new DPP-based PMRF image segmentation algorithm, which we refer to as DPP-PMRF. We first review the foundational PMRF approach upon which our work is based, and then present our reformulation of this approach using DPP.

8.2.1 Parallel MRF. The parallel MRF algorithm (PMRF) proposed by Perciano et al. [124] is shown in Algorithm 5. It consists of a one-time initialization phase, followed by a compute-intensive, primary parameter estimation optimization phase. The output is a segmented image.

The goal of the initialization phase is the construction of an undirected graph of pixel regions. The graph is built based on an oversegmented version of the original input image. An oversegmentation is a partition of the image into non-overlapping regions (superpixels), each with statistically similar grayscale intensities among member pixels [113]. The partitioning of the image we are using in this work is irregular, i.e. the non-overlapping regions can have different sizes and shapes. Each vertex V of the graph represents a region in the oversegmented image (i.e., a spatially connected region of pixels having similar intensity), and each edge E indicates spatial adjacency between regions. Given the irregular nature of the oversegmentation, the topological structure of the graph varies accordingly.

Next, in the main computational phase, we define an MRF model over the set of vertices, which includes an energy function representing contextual information of the image. In particular, this model specifies a probability distribution over the k-neighborhoods of the graph. Each k-neighborhood consists of the vertices of a maximal clique, along with all neighbor vertices that are within k edges (or hops) from any of the clique vertices; in this study, we use $k = 1$. Using OpenMP, the PMRF algorithm performs energy function optimization, in parallel, over the neighborhoods, each of which is stored as a single row in a *ragged array*. This optimization consists of an iterative invocation of the expectation-maximization (EM) algorithm, which performs parameter estimation using the maximum *a posteriori* (MAP) inference algorithm [68]. The goal of the optimization routine is to converge on the most-likely (minimum-energy) assignment of labels for the vertices in the graph; the mapping of the vertex labels back to pixels yields the output image segmentation.

Our proposed DPP-based algorithm overcomes several important problems encountered in the PMRF implementation such as non-parallelized steps of the algorithm (e.g., partitioning of the graph and MAP estimation computation), and platform portability. In particular, the OpenMP design of the PMRF conducts outer-parallelism over the MRF neighborhoods, but does not perform inner-parallelism of the optimization phase for each neighborhood (e.g., the energy function computations and parameter updates). Thus, the ability to attain fine-grained concurrency and greater parallelism is limited by the non-parallel computations within each outer-parallel optimization task. Finally, for the construction of MRF neighborhoods, our new method makes use of our DPP-based maximal clique enumeration (MCE) algorithm from Chapter VII.

8.2.2 DPP Formulation of PMRF. We now describe our DPP-based PMRF algorithm (DPP-PMRF) to perform image segmentation. Our algorithm redesigns PMRF in terms of DPPs to realize outer-level parallelism over MRF neighborhoods, and inner-level parallelism within the optimization routine for the vertices of each neighborhood. This data-parallel approach consists of an initialization phase followed by the main MRF optimization phase; refer to Algorithm 6 for the primary data-parallel steps.

8.2.2.1 Initialization. In this initial phase, we first construct an undirected graph *G* representing the connectivity among oversegmented pixel regions in the input image; refer to Section 8.2.1; Then, we enumerate all of the maximal cliques within *G*, yielding a set of complete subgraphs that form the basis of the MRF neighborhood structure.

Our initialization procedure is similar to that of the reference PMRF, but differs in the following ways. First, all of our initialization operations and algorithms are designed in terms of DPP, exposing high levels of data-parallelism throughout the entire image segmentation pipeline. Second, we represent *G* in a compressed, sparse row (CSR) format that fits compactly within shared memory; see Chapter VII for details on the DPP construction of this graph format.

Algorithm 6: Pseudocode of our DPP-based DPP-PMRF algorithm for performing graph-based image segmentation using Markov Random Fields.

```
Require: Original image, oversegmentation, number of output labels
Ensure: Segmented image and estimated parameters
Create graph from oversegmentation in parallel
Enumerate maximal cliques of graph in parallel
Initialize parameters and labels randomly
Construct $k$-neighborhoods from maximal cliques in parallel
Replicate neighborhoods by label in parallel
for each EM iteration do
    Gather replicated parameters and labels in parallel
    for each vertex of each neighborhood do
        MAP estimation computed in parallel
        Compute MAP estimation
    end
end
Update labels and parameters in parallel
return (Segment labels, Label Parameters)
```

8.2.2.2 Optimization. Given the graph G and its set of maximal cliques from the initialization, we proceed to the optimization phase, which consists of the following two primary data-parallel tasks: construction of neighborhoods over the maximal cliques and EM parameter estimation, the latter of which comprises the main computational work in this phase. Prior to constructing the neighborhoods, the mean and standard deviation parameters, μ and σ, of each label are randomly initialized to values between 0 and 255, representing the 8-bit grayscale intensity spectrum; in this study we focus on binary image segmentation with two labels of 0 and 1. Additionally, the label for each vertex of G is randomly initialized to either 0 or 1.

Construction of Neighborhoods: In the PMRF algorithm, 1-neighborhoods are serially constructed from maximal cliques during the initialization process of the algorithm. Our approach constructs the 1-neighborhoods before the parameter estimation phase and consists of the following data-parallel steps that operate on individual vertices, as opposed to entire maximal cliques, exposing more inner, fine-grained parallelism.

1. **Find Neighbors**: Invoke a data-parallel *Map* primitive to obtain, for each vertex, a count of the number of neighbors that are within 1 edge from the vertex and not a member of the vertex's maximal clique.
2. **Count Neighbors**: Call a *Scan* primitive to add the neighbor counts, the sum of which is used to allocate a neighborhoods array.
3. **Get Neighbors**: In a second pass to a *Map* primitive, populate the neighborhoods array with the neighbors, parallelizing over vertices as before.
4. **Remove Duplicate Neighbors**: Since multiple vertices within the same maximal clique may output common 1-neighbors in the neighborhoods array, successively invoke *SortByKey* and *Unique* primitives to remove the duplicate neighbors. The *SortByKey* primitive contiguously arranges vertices in the array in ascending order of their vertex Id and clique Id pairs. Then, the *Unique* primitive removes these duplicate, adjacent vertices, leaving a final neighborhoods array in which each set of neighbors is arranged in order of vertex Id.

EM Parameter Estimation: We formulate the EM parameter estimation via the following data-parallel steps.

1. **Replicate Neighborhoods By Label**: Next, each neighborhood is replicated for each of the two class output labels. With a sequence of *Map*, *Scan*, and *Gather* DPPs, we obtain a set of expanded indexing arrays, each of size $2 \times |hoods|$. The *testLabel* array indicates which replication of the neighborhood a given element belongs to; e.g., vertex element 2 belongs to the first copy of its neighborhood, denoted by a 0 label. The *hoodId* array gives the Id of the neighborhood to which

element belongs, and the *oldIndex* array contains back-indices into the original *hoods* array, for each replicated element.

$$hoods = [0\ 1\ 2\ 5\ 1\ 3\ 4]$$

$$testLabel = [0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1]$$

$$oldIndex = [0\ 1\ 2\ 3\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 4\ 5\ 6]$$

$$hoodId = [0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1]$$

$$repHoods = [\underbrace{0\ 1\ 2\ 5}_{\substack{Hood_0\\ Label_0}}\ \underbrace{0\ 1\ 2\ 5}_{\substack{Hood_0\\ Label_1}}\ 1\ 3\ 4\ 1\ 3\ 4]$$

The replication of the *hoods* array, *repHoods*, is not allocated in memory, but is simulated on-the-fly with a memory-free *Gather* DPP using *oldIndex*.

2. **For each EM iteration** *i*:

   – **Compute Energy Function**: Using the array of back-indices (*oldIndex*) into the neighborhoods array (*hoods*), we invoke a set of *Gather* DPP to create replicated data arrays of size $2 \times |hoods|$:

   $$vertLabel = [1\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1]$$

   $$vertMu = [40\ 20\ 55\ 25\ 40\ 20\ 55\ 25\ 20\ 65\ 35\ 20\ 65\ 35]$$

   $$labelMu = [30\ 30\ 60\ 30\ 30\ 30\ 60\ 30\ 30\ 60\ 30\ 30\ 60\ 30]$$

   We then invoke a *Map* DPP to compute an energy function value for each of the replicated neighborhood vertices. This operation parallelizes over the data arrays and calculates, for each vertex of a neighborhood, the energy, or deviation, between its actual grayscale intensity value (*vertMu*) and that of the label mean parameter (*labelMu*).

   – **Compute Minimum Vertex and Label Energies**: Within the array of computed energy function values, each vertex of a given neighborhood is associated with two values, one for each of the labels. In order to determine the minimum energy value between these labels, we invoke a *SortByKey* DPP, which makes each pair of energy values contiguous in memory. Then, we call consecutive *ReduceByKey*$\langle$*Min*$\rangle$ DPP on the sorted energy values to obtain the minimum energy value for each vertex:

- **Compute Neighborhood Energy Sums**: Given the minimum energies values, we call a *ReduceByKey*$\langle$*Add*$\rangle$ DPP to compute the sum of the values for each neighborhood.
- **MAP Convergence Check**: We maintain an array that stores the energy sum of each neighborhood at the end of every EM iteration. Using a *Map* DPP, we measure the amount of change in neighborhood energy sums from the previous L iterations ($L = 3$ in this study), and mark a neighborhood as *converged* if this change falls below a constant threshold of 1.0×10^{-4}. Once all neighborhoods have converged—assessed via a *Scan* DPP primitive—we end the EM optimization.

3. **Update Output Labels**: Invoke a *Scatter* DPP to write the minimum-energy label of each neighborhood vertex to its corresponding location in the global vertex label array.
4. **Update Parameters**: Use a sequence of *Map*, *ReduceByKey*, *Gather*, and *Scatter* DPP, to update the parameters of each label (μ and σ) as a function of a) the intensity values of the vertices assigned to the labels (i.e., minimum energy labels), and b) the sum of the per-label and per-vertex energy function values.
5. **EM Convergence Check**: We maintain an array that stores, for each EM iteration, the total sum of the neighborhood energy value sums after the final MAP iteration. Calling a *Scan* DPP on these neighborhood sums yields this total EM sum. Similar

to the MAP convergence check, we assess, via a *Map* DPP, the variation in EM sums over the previous L iterations.

During our experimentation, we find that most invocations of the EM optimization converge within 20 iterations; thus, we use that number of iterations in this study. Finally, we return the estimated parameters and assignment of labels to vertices as output. These labels can be mapped back to pixel regions of the vertices to produce the final segmented image.

8.3 Results

The experimental results in this section serve to answer two primary questions. First, in Section 8.3.2, we examine the question of correctness: is the new DPP-PMRF algorithm producing correct results? Second, in Section 8.3.3, we are interested in understanding how well the DPP-PMRF implementation performs on different modern CPU and GPU platforms: does DPP-PMRF demonstrate platform-portable performance? Because these experiments examine different questions, each uses a different methodology, which we present in conjunction with the experiment results. Section 8.3.1 describes the source datasets and the computational platforms that we use in both sets of experiments.

8.3.1 Source Data, Reference Implementation, and Computational Platforms.

8.3.1.1 Datasets. We test the DPP-PMRF implementation using two types of image-based datasets: one is synthetic and the other is output from a scientific experiment. The former is used to verify the accuracy of the proposed algorithm against a known benchmark that offers a ground-truth basis of comparison. The latter shows how DPP-PMRF performs on a real-world problem.

Synthetic data. We selected the synthetic dataset from the 3D benchmark made available by the Network Generation Comparison Forum (NGCF) [1]. The NGCF datasets are a global, recognized standard to support the study of 3D tomographic data of porous media. The datasets provided are binary representations of a 3D porous media. For the purposes of this analysis, we corrupted the original stack by noise (salt-and-pepper) and additive Gaussian with $\sigma = 100$. Additionally, we also simulate ringing artifacts [123] into the sample to closer resemble real-world results. For the segmentation algorithm analysis, the corrupted data serves as the "original data" and the binary stack as the ground-truth. A full synthetic dataset is 268 MB in size, and consists of 512 image slices of dimensions 512×512. The chosen dataset emulates a very porous fossiliferous outcrop carbonate, namely Mt. Gambier limestone from South Australia. Because of the more homogeneous characteristic of this dataset, its related graph contains a larger number of smaller-sized neighborhoods.

Experimental data. This dataset contains cross-sections of a geological sample and conveys information regarding the x-ray attenuation and density of the scanned material as a gray scale value. This data was generated by the Lawrence Berkeley National Laboratory Advanced Light Source X-ray beamline 8.3.2 [2] [33]. The scanned samples are pre-processed using a separate software that provides reconstruction of the parallel beam projection data into a 3 GB stack of 500 image slices with dimensions of 1813×1830. This dataset contains a very different and more complex set of structures to be segmented. Consequently, compared to the synthetic data, this experimental data leads to a denser graph with many more neighborhoods of higher complexity.

[1] `http://people.physics.anu.edu.au/~aps110/network_comparison`

[2] `microct.lbl.gov`

8.3.1.2 Hardware Platforms. Our verification and performance tests were run on two different multi-core platforms maintained by the National Energy Research Scientific Computing Center (NERSC). For each platform, all tests were run on a single node (among many available). Specifications for these two platforms are as follows:

1. `Cori.nersc.gov` (KNL): Cray XC40 system with a partition of 9,688 nodes, each containing a single-socket 68-core 1.4 GHz Intel Xeon Phi 7250 (Knights Landing (KNL)) processor and 96 GB DDR4 2400 GHz memory. With hyper-threading, each node contains a total of 272 logical cores (4 hyper-threads per core)[3].
2. `Edison.nersc.gov` (Edison): Cray XC30 system comprised of 5586 nodes, each containing two 12-core 2.4 GHz Intel Ivy Bridge processors (two sockets) and 64 GB DDR3 1866 MHz memory. With hyper-threading, each node contains a total of 48 logical cores (24 logical cores per socket of a node)[4]

The intention of running on the KNL and Edison systems is to create an opportunity for revealing architecture-specific performance characteristics.

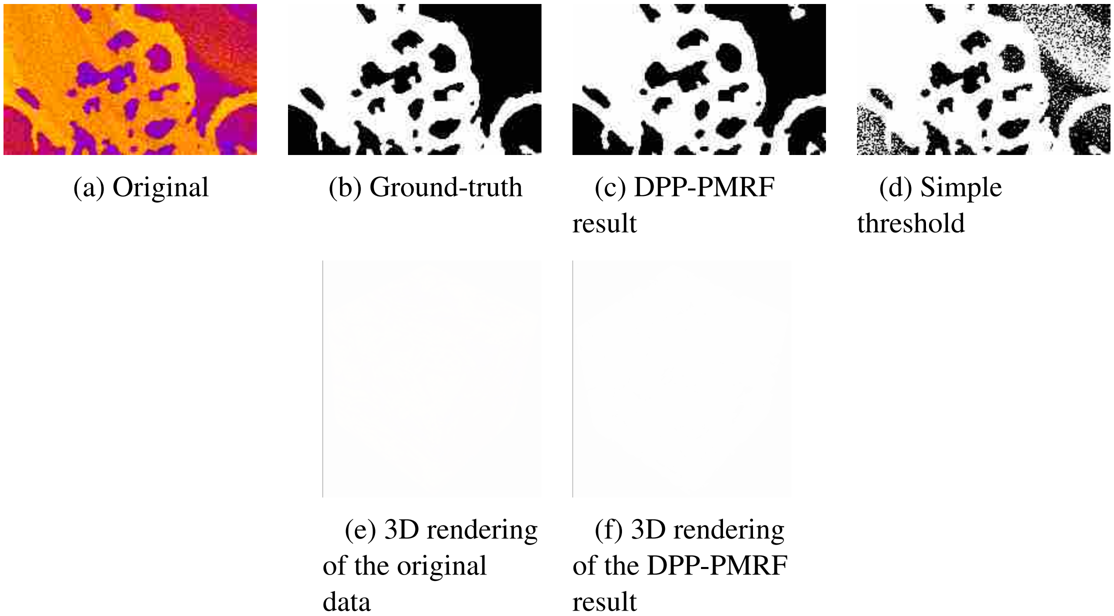
Performance tests were also conducted on a general-purpose GPU platform:

1. `K40`: NVIDIA Tesla K40 Accelerator with 2880 processor cores, 12 GB memory, and 288 GB/sec memory bandwidth. Each core has a base frequency of 745 MHz, while the GDDR5 memory runs at a base frequency of 3 GHz.

For both the experimental and synthetic image datasets, the peak memory usage of DPP-PMRF is well within the maximum available memory of the tested CPU (between

[3] Cori configuration page: `http://www.nersc.gov/users/computational-systems/cori/configuration/`

[4] Edison configuration page: `http://www.nersc.gov/users/computational-systems/edison/configuration/`



(a) Original

(b) Ground-truth

(c) DPP-PMRF result

(d) Simple threshold

(e) 3D rendering of the original data

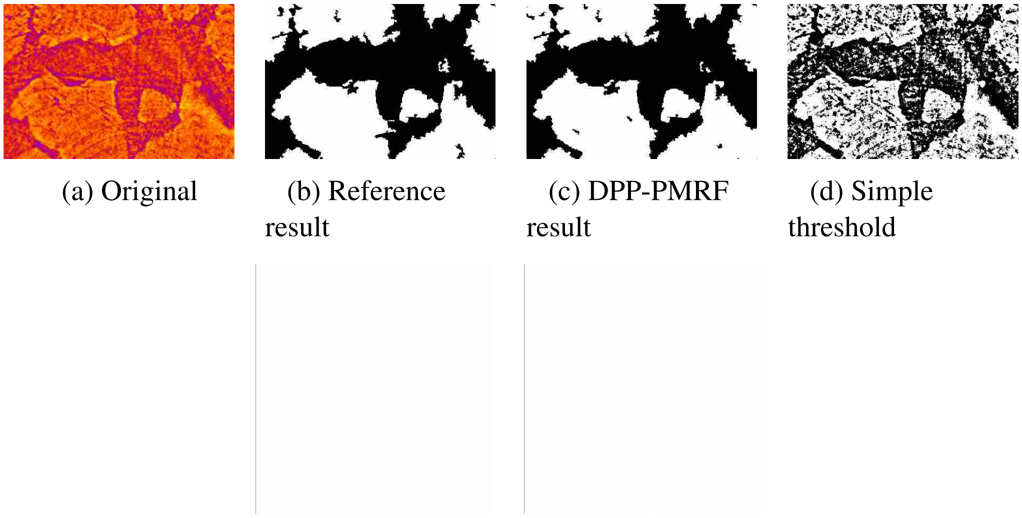
(f) 3D rendering of the DPP-PMRF result

Figure 11. Results of applying DPP-PMRF image segmentation to the synthetic image dataset. (a) Region of interest from the noisy data; (b) Ground-truth; (c) Result obtained by DPP-PMRF; (d) Result obtained by using a simple threshold; (e) 3D rendering of the original noisy image dataset; (f) 3D rendering of the result obtained by DPP-PMRF.

64 GB and 96 GB) and GPU (12 GB) platforms. The execution of DPP-PMRF results in a maximum memory footprint of between 300 MB to 2 GB for the experimental images and between 100 MB and 400 MB for the synthetic images.

8.3.1.3 Software Environment. Our DPP-PMRF algorithm is implemented using the platform-portable VTK-m toolkit [148]. With VTK-m, a developer chooses the DPPs to employ, and then customizes those primitives with functors of C++-compliant code. This code then invokes back-end, architecture-specific code for the architecture of execution (enabled at compile-time), e.g., CUDA Thrust code for NVIDIA GPUs and Threading Building Blocks (TBB) code for Intel CPUs.

In our CPU-based experiments, VTK-m was compiled with TBB enabled (version 17.0.2.174) using the following C++11 compilers: GNU GCC 7.1.0 on Edison and Intel ICC 18.0.1 on KNL. In our GPU-based experiments, VTK-m was compiled with CUDA



(a) Original

(b) Reference result

(c) DPP-PMRF result

(d) Simple threshold

(e) 3D rendering of the original data

(f) 3D rendering of the DPP-PMRF result

Figure 12. Results of applying DPP-PMRF image segmentation to the experimental image dataset. (a) Region of interest from the original data; (b) Reference result; (c) Result obtained by DPP-PMRF; (d) Result obtained by using a simple threshold; (e) 3D rendering of the original image dataset; (f) 3D rendering of the result obtained by DPP-PMRF.

enabled (version 8.0.61) using the NVIDIA CUDA NVCC compiler. For all experiments, version 1.2.0 of VTK-m was used.

With TBB enabled in VTK-m, each invocation of a DPP executes the underlying TBB parallel algorithm implementation for the primitive. The basic input to each of these parallel algorithms is a linear array of data elements, a functor specifying the DPP operation, and a *task* size that sets the number of contiguous elements a single thread can operate on. A partitioning unit invokes threads to recursively split, or divides in half, the array into smaller and smaller *chunks*. During a split of a chunk, the splitting thread remains assigned to the left segment, while another *ready* thread is assigned to the right segment. When a thread obtains a chunk the size of a task, it executes the DPP functor operation on the elements of the chunk and writes the results into an output array. Then, the thread is ready to be scheduled, or re-assigned, to another chunk that needs to be split further. This work-stealing scheduling procedure is designed to improve load balancing among parallel threads, while minimizing cache misses and cross-thread communication.

8.3.1.4 Reference Implementation of PMRF. In this study, we compare the performance and correctness of the new DPP-PMRF implementation with the PMRF reference implementation developed with OpenMP 4.5, which is described in Section 8.2.1. We take advantage of OpenMP loop parallelism constructs to achieve outer-parallelism over MRF neighborhoods, and make use of OpenMP's dynamic scheduling algorithm in the performance studies (see Section 8.3.3.3 for details).

The OpenMP-based implementation was built with the following C++11 compilers (same as for the DPP-based version): GNU GCC 7.1.0 on Edison and Intel ICC 18.0.1 on KNL.

8.3.2 Verification of Correctness. The following subsections present a set of tests aimed at verifying that DPP-PMRF computes the correct, ground-truth image segmentation output.

8.3.2.1 Methodology: Evaluation Metrics. In order to determine the precision of the segmentation results we use the metrics $precision = \frac{TP}{TP+FP}$, $recall = \frac{TP}{TP+FN}$, and $accuracy = \frac{TP+TN}{TP+TN+FP+FN}$, where TP stands for True Positives, TN for True Negatives, FP for False Positives, and FN for False Negatives.

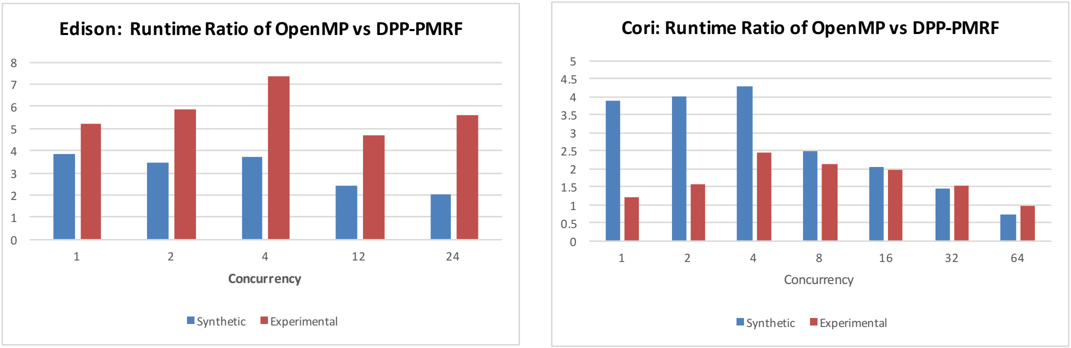
In addition, we also use the porosity (ratio between void space and total volume), or $\rho = \frac{V_v}{V_t}$,, where V_v is the volume of the void space and V_t is the total volume of the void space and solid material combined.

8.3.2.2 Verification Results. Figure 11 shows the results of applying DPP-PMRF to the synthetic data. Figure 11(a) presents a 2D region of interest from the corrupted data, Figures 11(b-d) show the ground-truth, the result from DPP-PMRF and the result using a simple threshold, respectively, and Figures 11(e-f) shows the 3D renderings of both the corrupted data and the DPP-PMRF result. We observe a high similarity between the DPP-PMRF result and the ground-truth, indicating a high precision when compared to the simple threshold result. For this synthetic dataset, the verification metrics obtained are a precision of 99.3%, a recall of 98.3%, and an accuracy of 98.6%.

Following the same methodology, we present the results using the experimental dataset in Figure 12. Figures 12(a-d) shows regions of interest from the original data, the result from the reference implementation, the DPP-PMRF result, and a simple threshold result, respectively. Much like the results using the synthetic data, we observe a close similarity between the DPP-PMRF result and the reference result. The differences observed between the results are usually among the very small regions in the image, where small variations of labeling of the graph could lead to the same minimum energy value. For this dataset the verification metrics obtained are a precision of 97.2%, a recall of 95.2% and an accuracy of 96.8%.

8.3.3 Performance and Scalability Studies. The next subsections present a set of studies aimed at verifying the performance, scalability, and platform-portability of DPP-PMRF, as compared to a OpenMP-parallel reference implementation and serial baseline.

8.3.3.1 Methodology. The objectives for our performance study are as follows. First, we are interested in comparing the absolute runtime performance of the OpenMP and DPP-PMRF shared-memory parallel implementations on modern multi-core CPU platforms. Second, we wish to compare and contrast their scalability characteristics, and do so using a strong-scaling study, where we hold the problem size constant and increase concurrency. Finally, we assess the platform-portable performance of DPP-PMRF by executing the algorithm on a general-purpose GPU platform and comparing the runtime



(a) Edison: Ivy Bridge

(b) Cori: Knights Landing (KNL)

Figure 13. Comparison of absolute image segmentation runtimes of the DPP and OpenMP implementations at varying concurrency, on both platforms, and both sample datasets. The horizontal axis is concurrency level, or the number of physical cores used on a single node. Each bar represents the ratio of runtimes of the DPP-PMRF to the OpenMP code. The vertical axis measures how much faster the DPP-PMRF code is than the OpenMP code for a given dataset, on a given platform, and a given concurrency. A bar height of 1.0 means both codes have the same runtime; a bar height of 2.0 means the DPP code ran in half the time of the OpenMP code.

performance to a serial (non-parallel) baseline and the CPU execution from the strong-scaling study.

To obtain elapsed runtime, we run these two implementations in a way where we iterate over 2D images of each 3D volume (synthetic data, experimental data). We report a single runtime number, which is the average of elapsed runtime for each 2D image in the 3D volume. The runtime takes into account only the optimization process of the algorithm as this is the portion of the algorithm that is most computationally intensive.

From runtime measurements, we report results using charts that show time vs. concurrency, which are often known as “speedup” charts. Moreland and Oldfield [108] suggest that such traditional performance metrics may not be effective for large-scale studies for problem sizes that cannot fit on a single node. Since our problems all fit within a single node, and are of modest scale, we are showing traditional speedup charts.

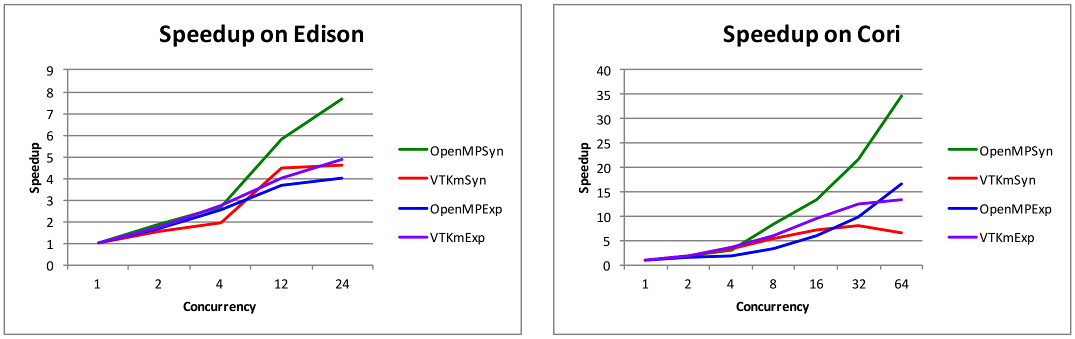
Speedup is defined as $S(n,p) = \frac{T^*(n)}{T(n,p)}$ where *T(n,p)* is the time it takes to run the parallel algorithm on *p* processes with an input size of *n*, and *T*(n)* is the time for the best serial algorithm on the same input.

8.3.3.2 CPU Runtime Comparison: OpenMP vs. DPP-PMRF.

The first performance study question we examine is a comparison of runtimes between the OpenMP and DPP-PMRF implementations. We executed both codes at varying levels of concurrency on the Cori and Edison CPU platforms, using the two different datasets as input. Each concurrency level represents the number of physical cores used within a single node. Hyper-threading was active in each experiment, resulting in more logical (virtual) cores than physical cores being utilized per node (see Section 8.3.1.2 for hardware configuration details). The runtimes for this battery of tests are presented in Fig. 13 in a way that is intended to show the degree to which DPP-PMRF is faster, or slower, than the OpenMP version.

In Fig. 13, each bar is computed as the quotient of the OpenMP runtime and the DPP runtime. A bar height of 1.0 means both codes have the same runtime; a bar height of 2.0 means DPP-PMRF ran in half the time of the OpenMP code. A bar height of less than 1.0 would mean that the OpenMP code ran faster than DPP-PMRF. These results reveal that DPP-PMRF significantly outperforms the OpenMP code, by amounts ranging from $2X$ to $7X$, depending upon the platform and concurrency level.

The primary factor leading to this significant performance difference is the fact that the DPP formulation makes better use of the memory hierarchy. Whereas the OpenMP code operates in parallel over rows of a ragged array, DPP-PMRF recasts the problem as a series of atomic data parallel operations. To do so, it creates 1D arrays, which are then partitioned at runtime across a thread pool. Such a formulation is much more amenable to vectorization, and results in significantly more uniform and predictable



(a) Edison: Ivy Bridge

(b) Cori: Knights Landing (KNL)

Figure 14. Image segmentation runtime speedup of synthetic and experimental image datasets on the Edison and Cori CPU platforms.

memory access patterns. The size of these partitions, or chunks, is determined by TBB in a way to best match the cache characteristics and available parallelism of the underlying platform (see Section 8.3.1.3 for details). In contrast, the OpenMP code "chunk size" is the size of the given graph neighborhood being processed. There is a significant performance difference that results when using a consistent and well chosen "blocking factor," which results in better use of locality, in both space and time [30]. Our results are consistent with previous literature, which suggest one key factor to high performance on contemporary architectures is through code vectorization ([84]).

At higher levels of concurrency on the KNL platform, we see a significant performance decrease in DPP-PMRF, which is unexpected. At 64 cores, the runtime for DPP-PMRF actually *increases* compared to the runtime for 32 cores. Examination of detailed per-DPP timing indicates the runtime for two specific data parallel primitives, *SortByKey* and *ReduceByKey*, are the specific operations whose runtime increases going from 32 to 64 cores. These data parallel primitives rely on an underlying vendor implementation in VTK-m with TBB as the back-end. Further investigation is needed to better understand why these underlying vendor implementations decrease in performance going from 32 to 64 cores.

8.3.3.3 Strong Scaling Results. The second performance study question we examine is the degree to which the OpenMP and DPP-PMRF implementations speed up with increasing concurrency. Holding the problem size fixed, we vary the concurrency on each platform, for each of the two image datasets. Concurrency ranges from 1 to N, where N is the maximum number of cores on a node of each of the two platforms.

Results from this study are shown in Fig. 14, which show speedup curves for both implementations, on both platforms, at varying concurrency. Looking at these results, the discussion that follows centers around two main themes. First, how well are these codes scaling, and what might be the limits to scalability? Second, how do scaling characteristics change with increasing concurrency, platform, and dataset?

In terms of how well these codes are scaling, the ideal rate of speedup would be equal to the number of cores: speedup of 2 on 2 cores, speedup of 4 on 4 cores, and so forth. The first observation is the both codes exhibit less than ideal scaling. The OpenMP code shows the best scaling on the synthetic dataset on both platforms, even though its absolute runtime is less than DPP-PMRF (except for one configuration, at 64 cores on the KNL platform). The reasons for why these codes exhibit less than ideal scaling differ for each of the codes.

The OpenMP code, which uses loop-level parallelism over the neighborhoods of the graph, has as critical section that serializes access by all threads. This critical section is associated with a thread writing its results into an output buffer: each thread is updating a row of a ragged array. We encountered what appears to be unexpected behavior with the C++ compiler on both platforms in which the output results were incorrect, unless this operation was serialized (see Section 8.3.1.4 for compilation details). Future work will focus on eliminating this serial section of the code to improve scalability.

The DPP-PMRF code, which is a sequence of data parallel operations, depends upon an underlying vendor-provided implementation of key methods. In these studies, an analysis of runtime results looking at individual runtime for each of the DPP methods (detail not shown in Fig. 14), indicates that two specific DPP operations are limited in their scalability. These two operations, a *SortByKey* and *ReduceByKey*, exhibit a maximum of about 5*X* speedup going from 1 to 24 cores on Edison, and about 11*X* speedup going from 1 to 64 cores on Cori. As a result, the vendor-supplied implementation of the underlying DPP is in this case the limit to scalability. We have observed in other studies looking at scalability of methods that use these same DPP on GPUs [81], that the vendor-provided DPP implementation does not exhibit the same limit to scalability. In that study, the sort was being performed on arrays of integers. In the present study, we are sorting pairs of integers, which results in greater amounts of memory access and movement, more integer comparisons, as well as additional overhead to set up the arrays and work buffers for both those methods.

On Edison, both codes show a tail-off in speedup going from 12 to 24 cores. A significant difference between these platforms is processor and memory speed: Edison has faster processors and slower memory; Cori has slower processors and faster memory. The tail-off on Edison, for both codes, is most likely due to increasing memory pressure, as more cores are issuing an increasing number of memory operations.

For the OpenMP code on both platforms, we see a noticeable difference in speedup curves for each of the two different datasets. On both platforms, the OpenMP code scales better for the synthetic dataset. Since the algorithm performance is a function of the complexity of the underlying data, specifically neighborhood size, we observe that these two datasets have vastly different demographics of neighborhood complexity (not shown due to space limitations). In brief, the synthetic dataset has a larger number

Platform / Dataset	Experimental	Synthetic
Serial CPU	284.51	44.63
DPP-PMRF CPU	22.77	7.09
DPP-PMRF GPU	6.55	1.71
Speedup-CPU	13X	7X
Speedup-GPU	44X	27X

Table 16. GPU image segmentation runtimes (sec) for DPP-PMRF over the experimental and synthetic image datasets, as compared to both serial and parallel CPU executions of DPP-PMRF on the KNL platform. The GPU speedup for a dataset is the serial CPU runtime divided by the DPP-PMRF GPU runtime. The CPU speedup is the DPP-PMRF CPU runtime divided by the DPP-PMRF GPU runtime.

of smaller-sized neighborhoods and the histogram indicates bell-shaped distribution. In contrast, the experimental dataset has many more neighborhoods of higher complexity, and the distribution is very irregular. Because the OpenMP code parallelizes over individual neighborhoods, it is not possible to construct a workload distribution that attempts to create groups of neighborhoods that result in an even distribution of work across threads. We are relying on OpenMP's dynamic scheduling to achieve good load balance in the presence of an irregular workload distribution. In this case, the result is the more irregular workload results in lower level of speedup for the OpenMP code on both platforms. In contrast, the DPP code reformulates this problem in a way that avoids this limitation.

8.3.3.4 Platform Portability: GPU Results. The final performance study question we examine is an assessment of the platform-portable performance of DPP-PMRF. We ran the algorithm on an NVIDIA Tesla K40 GPU accelerator, using the experimental and synthetic image datasets as input. The average GPU runtime for each dataset is compared to the average KNL CPU runtimes of both a serial (single core, hyper-threading disabled) execution of DPP-PMRF and the parallel execution of DPP-PMRF at maximum concurrency (68 cores, hyper-threading enabled; see Section 8.3.3.3).

From Table 16 we observe that, for both of the image datasets, DPP-PMRF achieves a significant speedup on the GPU over the serial version, with a maximum speedup of 44X on the experimental images. Further, for both datasets, DPP-PMRF attains greater runtime performance on the GPU (maximum speedup of 13X on the experimental images), as compared to its execution on the KNL CPU platform. These speedups demonstrate the ability of a GPU architecture to utilize the highly-parallel design of our algorithm, which consists of many fine-grained and compute-heavy data-parallel operations. Moreover, this experiment demonstrates the portable performance of DPP-PMRF, as we achieved improved runtimes without having to write custom, optimized NVIDIA CUDA GPU code within our algorithm; the same high-level algorithm was used for both the CPU and GPU experiments.

8.4 Conclusion

This chapter contributes a new DPP-based algorithm, DPP-PMRF, for a graph-based image segmentation application, using Markov Random Fields (MRF). DPP-PMRF is comprised of an iterative optimization routine that performs several aggegrate (e.g., summation and maximum) calculations to arrive at the most-probable segment labels for each vertex in an image graph. This optimization routine operates on possibly-overlapping subgraphs and, thus, distinct vertices may appear multiple times within the optimization. We deploy a sorting- and reduction-based search technique to aggregate the values of duplicate vertices and use these aggregate values to inform the parameters for the next iteration of the optimization. We found this technique to be the more-appropriate than a hashing-based technique for this particular analysis algorithm and optimization routine, since the performance of sorting and reducing basic numerical values each iteration exceeded that of building and probing a hash table structure.

DPP-PMRF is designed entirely in terms of DPP and tested on both CPU and GPU devices, demonstrating platform-portability across many-core systems. Compared to a state-of-the-art, CPU-based reference implementation, DPP-PMRF achieves competitive to leading segmentation runtime performance on large stacks of 2D real-world geological images. Moreover, DPP-PMRF attained a runtime speedup on a GPU device, as compared to that on a CPU device. Based on these findings, we believe that our sorting- and reduction-based solution is the best index-based search technique for this particular graph-based image analysis algorithm on diverse many-core systems. These findings inform our dissertation question and will be further synthesized in Chapter X.

Part III

Best Practices

In this part of the dissertation, we discover, synthesize, and recommend a set of best practices for attaining best platform-portable performance with the index-based search techniques of Parts I and II.

CHAPTER IX

BEST PRACTICES FOR DUPLICATE ELEMENT SEARCHING

This chapter presents best practices for the duplicate element index-based search techniques of Chapters III, using EFC as the experimental application (V). We discovered that the performance of these techniques can vary unexpectedly with certain combinations of hash function, architecture, and data set. In this chapter, we investigate several different test configurations, in an effort to better understand anomalous behavior. Using various metrics, we are able to understand the causes of unusual performance and specify recommendations for choices that will perform well over a variety of configurations.

The content of this chapter is primarily adopted from a collaborative conference publication completed by myself, Kenneth Moreland, Matthew Larsen, and Hank Childs [81]. As lead author of this publication, I contributed the majority of the software development, experimental analysis, and paper writing. Kenneth Moreland and Matthew Larsen helped me conduct experiments on different compute platforms, formulate the final experimental design, generate plots, and analyze the experimental results. Hank Childs provided valuable guidance towards the motivation of the work and editing the final submission.

9.1 Introduction

Searching for duplicate elements comes up in multiple scientific visualization contexts, most notably in EFC (see Chapter V). There are two main index-based approaches for identifying duplicates: (1) sorting all elements and looking for identical neighbors, and (2) hashing all elements and looking for collisions. Data-parallel variants of these approaches were introduced in Chapter III and then used in Chapter V for the EFC task. However, the performance of the algorithm can vary unexpectedly with certain combinations of hash function, architecture, and data set. In this chapter, we investigate several different test configurations, in an effort to better understand anomalous behavior. Using various metrics, we are able to understand the causes of unusual performance. We believe the contributions of this chapter are two-fold. First, we contribute a better understanding of platform portable algorithms for identifying duplicates and their pitfalls, and specify recommendations for choices that will perform well over a variety of configurations. Second, we believe the result is useful in identifying potential performance issues with DPP algorithms.

9.2 Experiment Overview

To better understand the behavior of EFC with respect to algorithm design choices, particularly for that of hash functions, we conducted experiments that varied three factors:

- Algorithm (7 options)
- Hardware architecture (3 options)
- Data set (34 options)

We did run the cross-product of tests (714 = 7 × 3 × 34), but our results section presents the relevant subset that capture the underlying behavior.

9.2.1 Algorithm.

We studied three types of algorithms, which we refer to as **SortyById**, **Sort**, and **Hash-Fight** in this chapter (note that Hash-Fight is not the same as our HashFight hash table of Chapters IV and VI, but refers to the “hash-fighting" approach used in our hashing-based EFC). Sort and Hash-Fight each need to be coupled with a hashing function. We considered three different hashing functions: **XOR**, **FNV1a**, and **Morton**. In total, we considered seven algorithms: Sort+FNV1a, Sort+XOR, Sort+Morton, Hash-Fight+FNV1a, Hash-Fight+XOR, Hash-Fight+Morton, and SortByID. We provide a review of the three algorithms and three hash functions in the following subsections. Refer to Chapters III and V for additional details regarding SortById and Hash-Fight.

9.2.1.1 Algorithms. **SortById**: The idea behind this approach is to use sorting to identify duplicate faces. First, faces are placed in an array and sorted. Each face is identified by its indices. The sorting operation requires a way of comparing two faces (i.e., a "less-than" test); we order the vertices within a face, and then compare the vertices with the lowest index, proceeding to the next indices in cases of ties. The array can then be searched for duplicates in consecutive entries. Faces that repeat in consecutive entries are internal, and the rest are external.

SortById is likely not optimal, in that it requires storage for each index in the face (e.g., three locations for each point in a triangular face of a tetrahedron), resulting in a penalty for sorting extra memory. This motivates the next approach, which is to use hashing functions to reduce the amount of memory for each face in the sort.

Sort: We denote the algorithm that modifies SortById to sort hash values rather than indices. For each face, the three vertex indices are hashed, and the resulting integer value is used to represent the face. However, this creates additional work. The presence of collisions forces us to add a step to the algorithm that verifies whether matching hash values actually belong to the same face. In this study, we explore the tradeoff between sorting multiple values per face versus resolving collisions. Further, the specific choice of hash function may affect performance, and we explore this issue as well.

Hash-Fight: Traditionally, hash collisions are handled via a chaining or open-addressing approach. While these approaches are straight-forward to implement in a serial setting, they do not directly translate to a parallel setting. For example, if multiple threads on a GPU map to the same hash entry at the same time, then the behavior may be non-deterministic, unless atomics are employed.

In Chapter III, hash collisions are addressed in a data-parallel setting via a hashing scheme that uses multiple iterations. In this scheme, which we denote as Hash-Fight for this chapter, no care is taken to detect collisions or prevent race conditions via atomics. Instead, every face is simultaneously written to the hash table, possibly overwriting previously-hashed faces. The final hash table will then contain the winners of this “last one in" approach. However, our next step is to check, for each face, whether it was actually placed in the hash table. If so, the face is included for calculations during that iteration. If not, then the face is saved for future iterations. All faces are eventually processed, with the number of iterations equal to the maximum number of faces hashed to a single index.

9.2.1.2 Hash Function. The following hash functions are considered in our study, each returning a hash value in the form of a 32-bit unsigned integer.

XOR: The XOR hashing function is a very simple bitwise exclusive-or operation on all the indices of a face. The XOR hash is very fast to compute but makes little effort to avoid collisions.

FNV1a: FNV1a hashing [40] starts with an offset value (2166136261 for this study) and then iteratively multiplies the current hash by a prime number (16777619 for this study) and performs an exclusive-or with the vertex index. The addition of the offset and prime number pseudo-randomize the hash values, which helps reduce collisions. However, FNV1a takes longer to compute because of its additional calculations.

Morton: The Morton z-order function maps a multi-dimensional point coordinate to a single Morton code value. It does this by interleaving and combining the binary representations of the coordinate values, while preserving the spatial locality of the points [110]. In this study, we compute a separate Morton code for each of the three vertex indices of a triangular cell face, and then add the three Morton codes together to form a hash value.

9.2.2 Hardware Architecture. We ran our tests on the following three architectures:

1. Haswell: Dual Intel Xeon Haswell E5-2698 v3, each with 16 cores running at 2.30 GHz and 2 SMT hardware threads and a total of 512 GB of DDR4 memory.
2. Knights Landing: An Intel Knights Landing Self Hosting Xeon Phi CPU with 72 cores, running in quadrant cluster mode and flat memory node. Each core has 4 threads and runs at a base clock frequency of 1.5 GHz. This processor also maintains 16 KB of on-package MCDRAM memory and 96 GB of DDR4 memory.
3. Tesla: An NVIDIA Tesla P100 Accelerator with 3,584 processor cores, 16 GB memory, and 732 GB/sec memory bandwidth. Each core has a base frequency of 1,328 MHz, and a boost clock frequency of 1480 MHz.

9.2.3 Data set. For all of our experiments, we used variants of an unstructured tetrahedral mesh derived by tetrahedralizing a uniform grid. We considered 17 different data set sizes ranging from 4 million to 667 million cells, which translates to a range of 16 million to 2.6 billion faces. Additionally, we considered two types of mesh connectivity indexing schemes:

1. Regular indexing: mesh connectivity was left unaltered. Mesh coordinates that are spatially close were generally located nearby in memory.
2. Randomized indexing: mesh connectivity was randomized, i.e., mesh coordinates were scattered randomly in memory.

We considered randomized indexing to study the behavior of the different algorithms and hashing functions with different types of memory access patterns. We viewed these two patterns as ends of a spectrum (coherent access versus incoherent access), and we believe real world data sets will fall within these two extremes. Between the 17 data set sizes and two indexing schemes, there were 34 total data sets used in the study.

To obtain our randomized indexing, we randomized the location of the vertices and cells in the regular data sets and adjusted the cell indices to match the new locations of the vertices. The randomized topology has the following two effects. First, indices are no longer near one another. For example, a tetrahedral cell in the regular topology might have indices 14, 21, 16 and 25, while, with the randomized topology, the same cell might have indices 512, 1, 73, and 1024. Second, accessing the vertices of a cell will exhibit poor memory access patterns since each point vertex is likely on a different cache line.

9.3 Results

Our study contains three phases, each of which assesses the impact of different factors on performance: hash function, architecture, and data set regularity. In this section, we present and analyze the results of these phases.

9.3.1 Phase 1: Hash Functions. This phase examines the choice of hash function, and considers the performance over our 7 combinations of algorithm type and hash function. We also vary the data set size, specifically looking at all 17 regularly-indexed data sets. The only architecture considered in this phase is the Intel Haswell CPU, although we vary the architecture in subsequent phases.

We first analyze the performance of the Hash-Fight algorithm. From Figure 15, we observe that Hash-Fight+FNV1a is consistently the fastest algorithm across the entire range of data set sizes. Hash-Fight+Morton is a close second, up until large data set sizes. When the data set size jumps from 530 million cells to 670 million cells, its execution time more than doubles. This jump in execution time is partially attributed to an increase in the number of hash-fight iterations, from 18 to 24. This increase in iterations is the

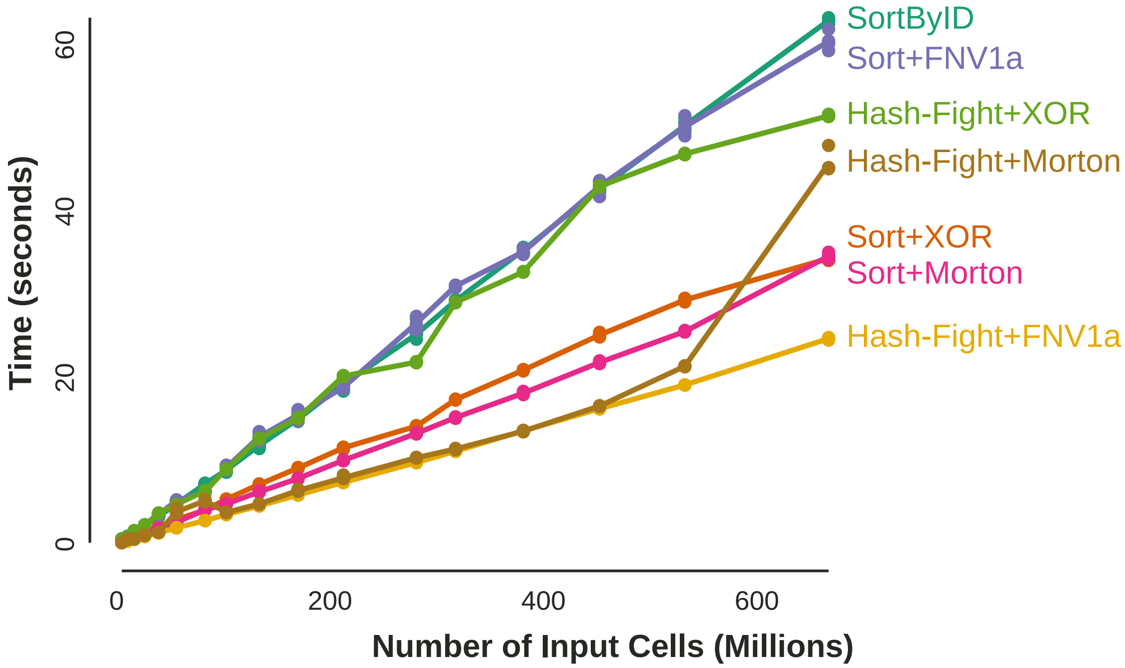


Figure 15. EFC runtime comparison of all algorithm types and all hash functions on Intel's Haswell architecture with regularly-indexed 3D data sets. 10 trials were conducted per algorithm for each data set, and each trial is represented by a dot. The trendlines plot the average times of these trials.

result of the number of hash collisions tripling from 619 million to 1.9 billion, as seen in Figure 16 (Morton regular plot). We believe that the large increase in hash collisions occurred because the spatial locality of the Morton z-order curve deteriorates as the cell count significantly increases, due to the additive property of the Morton hash function. This increase in collisions was also seen with Sort+Morton.

Additionally, Hash-Fight coupled with the XOR hash function tends to perform poorly compared to the couplings with FNV1a and Morton. From Figure 16 (XOR regular plot), Hash-Fight+XOR consistently yields the largest number of hash collisions, by a wide margin, among the Hash-Fight algorithms. Overall, the trends for collisions in Figure 2 closely match the trends for Hash-Fight execution times in Figure 1.

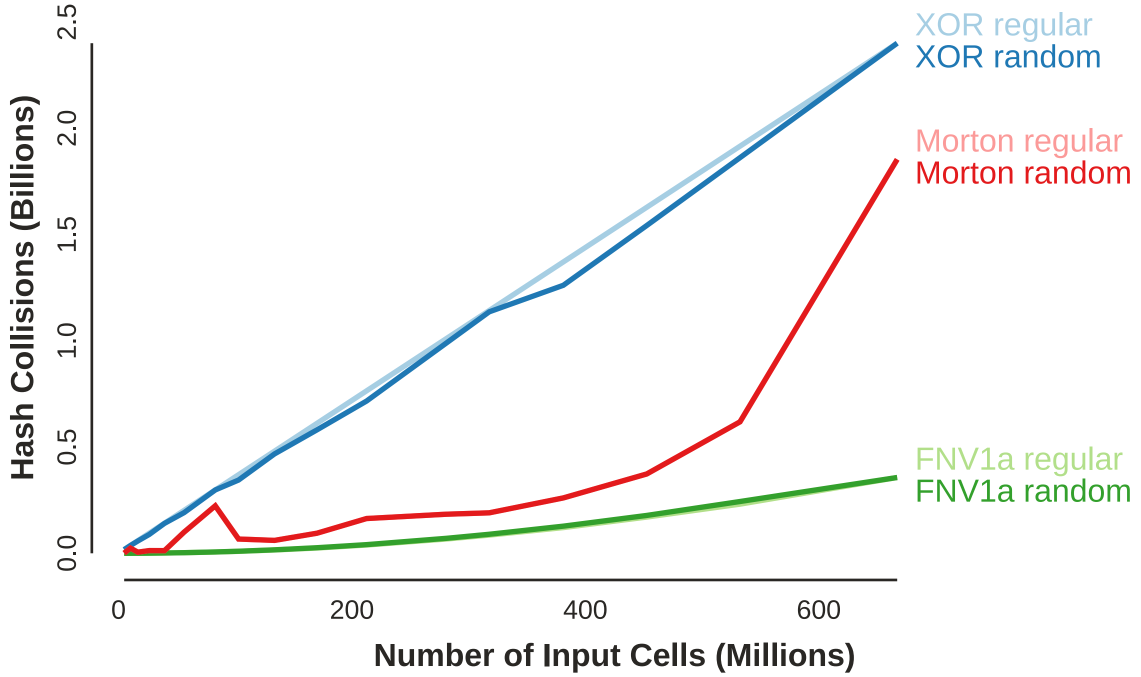


Figure 16. A comparison of the number of collisions produced by the 3 hash functions considered in this study, over both regular and random indexed 3D data sets for EFC. The Morton and FNV1a functions have little variation in collisions between data set type and, thus, have overlapping plots.

Operation	SortBy	Sort	Sort	Sort
	Id	XOR	FNV1a	Morton
prepInputFaces	0.88	0.86	0.86	0.87
hashFaces	2.05	1.03	1.04	1.13
hashSortReduce	25.37	9.41	23.59	10.82
findInternalFaces	0.17	5.72	4.89	1.89
prepOutputFaces	0.56	0.29	0.46	0.41
Total Time	29.03	17.31	30.84	15.12

Table 17. Intel Haswell CPU runtimes (sec) for each of the primary data-parallel operations of the Sort algorithms for EFC. Each time is the average of the 10 trials conducted on the 400^3 grid data set with approximately 318 million input cells.

We now consider the performance of the Sort algorithm. Although Hash-Fight generally performs better with FNV1a and worse with XOR, the opposite outcome tends to occur with Sort.

From Figure 15, we observe that Sort+XOR begins to significantly outperform Sort+FNV1a as the number of input cells increases. Although the XOR hash function produces many collisions, which increases the time to find internal faces, it also results in a markedly faster sort time (see the hashSortReduce row in Table 17). We discovered that the XOR hash function happens to create hash values that are already close to being in sorted order for these data set structures. The parallel sort algorithm, which comes from the TBB library [127], is a parallel version of quicksort, and this algorithm runs much faster on data that is pre-sorted or close to being sorted. The benefits from the faster sorting outweigh the extra time needed to resolve collisions.

Finally, the performance of SortById confirms our hypothesis from Section 9.2.1.1 that the cost of sorting multiple values per face (3 points per triangular face) is larger than the cost of resolving collisions from hashing the face. As Table 17 indicates, SortById consumes most of its total runtime ordering the 3 face points (hashFaces operation) and then sorting and reducing the faces (hashSortReduce operation).

9.3.2 Phase 2: Architectures. In this phase, we conduct the same set of experiments from Phase 1 on two additional architectures—Intel Knights Landing (KNL) CPU and Nvidia Tesla P100 GPU—and compare the results to that of the Intel Haswell CPU experiments from Phase 1. Figures 17 and 18 present the results of the KNL and Tesla experiments, respectively. Note that, due to machine memory restrictions, only the 11 datasets up to 213 million cells (350^3 grid) are considered for KNL, and only the 5 datasets up to 40 million cells (200^3 grid) are considered for Tesla.

Figures 17 and 18 reveal that the Sort and Hash-Fight algorithm types almost always run the fastest when coupled with the FNV1a or Morton hash functions on both the KNL and Tesla. Additionally, Hash-Fight+FNV1a and Hash-Fight+Morton are consistently the highest-performing algorithms for both architectures. From Phase 1, we

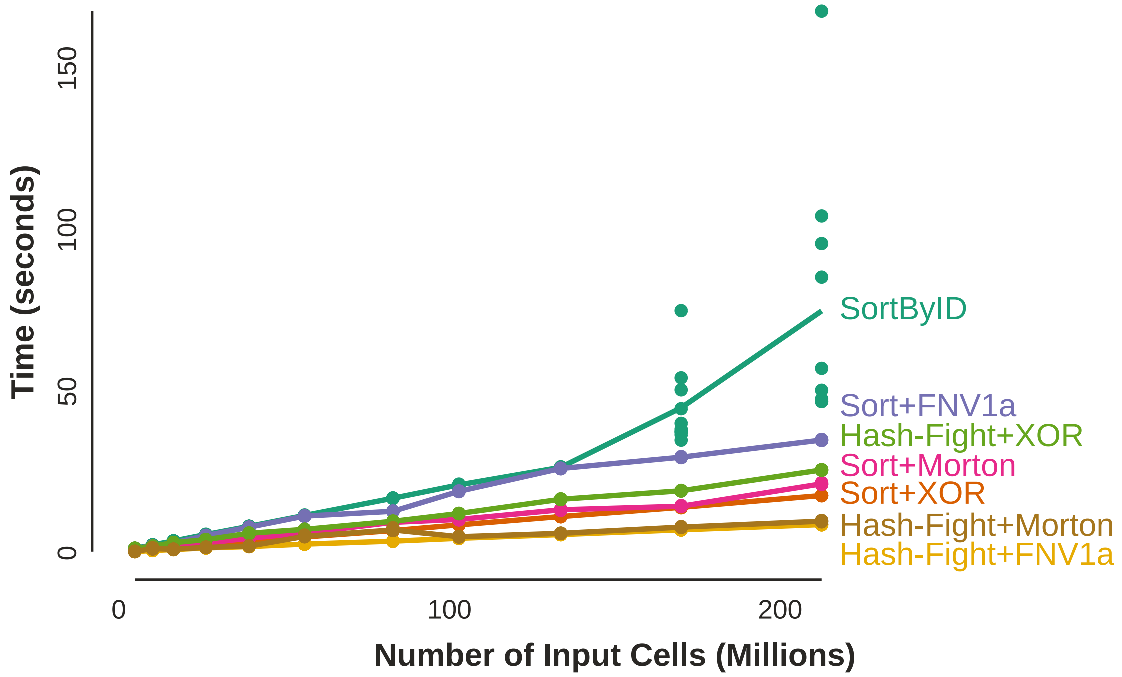


Figure 17. EFC runtime comparison of all algorithms and all hashing functions on Intel's Knights Landing architecture with regularly indexed 3D data sets. 10 trials were conducted per algorithm for each data set, and each trial is represented by a dot. The trendlines plot the average times of these trials.

discovered that the performance of an algorithm deteriorated for larger numbers of input cells with the Morton function and Haswell architecture. The analysis revealed that this lower performance was due to an increase in hash collisions. On the KNL and Tesla, this increase in collisions still occurs, but does not negatively affect the performance as it did on Haswell, at least for the size of data sets that fit within these architectures' memory. This finding may also be due to architecturally-specific traits in the memory hierarchies (e.g., NUMA, cache infrastructure, etc.); we will investigate this further in future work.

Similar to the findings of Phase 1 on Haswell, the Sort algorithm type performs very well on KNL when coupled with the XOR function. However, Sort+XOR is the worst-performing algorithm, along with SortById, on the Tesla architecture. This reversal in performance of Sort+XOR on the Tesla can be attributed to the sub-routine that finds

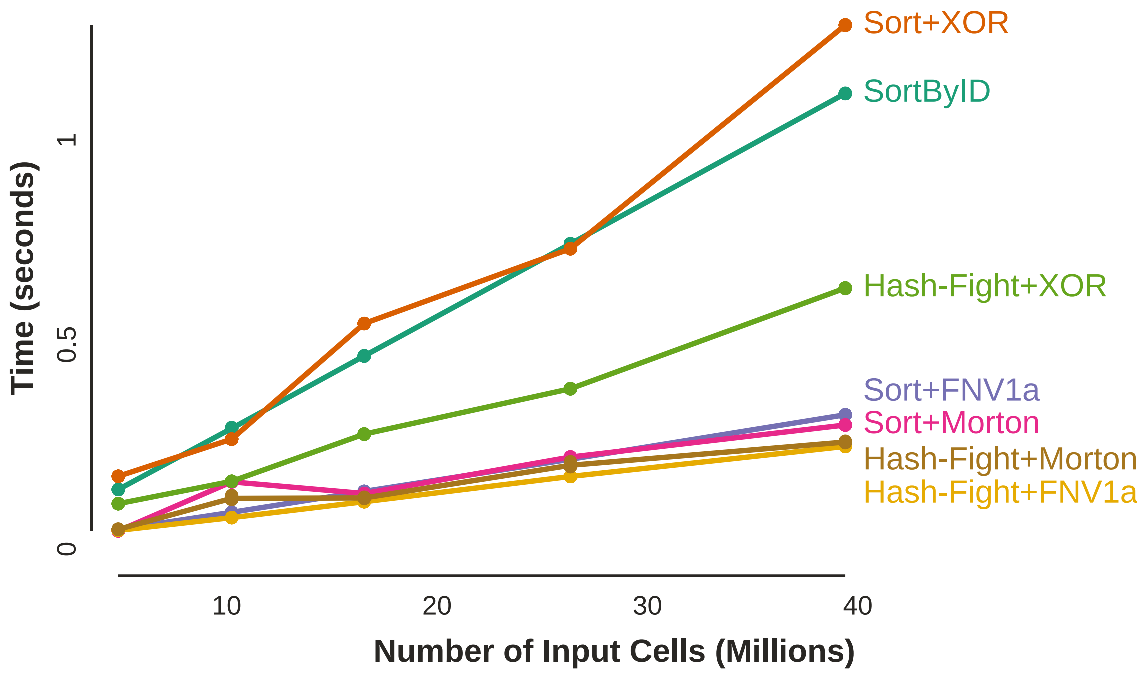


Figure 18. EFC runtime comparison of all algorithms and all hashing functions on NVIDIA's Tesla P100 GPU with regularly indexed 3D data sets. 10 trials were conducted per algorithm for each data set, and each trial is represented by a dot. The trendlines plot the average times of these trials.

all internal faces. From Table 18, we observe that XOR, FNV1a, and Morton all take approximately the same amount of time (0.15 seconds) to sort and reduce the hash values, but differ greatly in the time needed to find internal faces, with XOR requiring almost a whole second longer to complete. This is because XOR yielded a substantially larger number of hash collisions, which generated more neighbor searches to resolve collisions and find the internal faces. This matches the finding from Phase 1. However, unlike in Phase 1, the XOR algorithm did not yield a significantly faster sort time to compensate for the increase in time caused by the added collisions. This is because the GPU's parallel sort algorithm, which comes from the Thrust library [8, 117], is based on radix sorting, which is both faster in general and much less sensitive to initial ordering than quicksort.

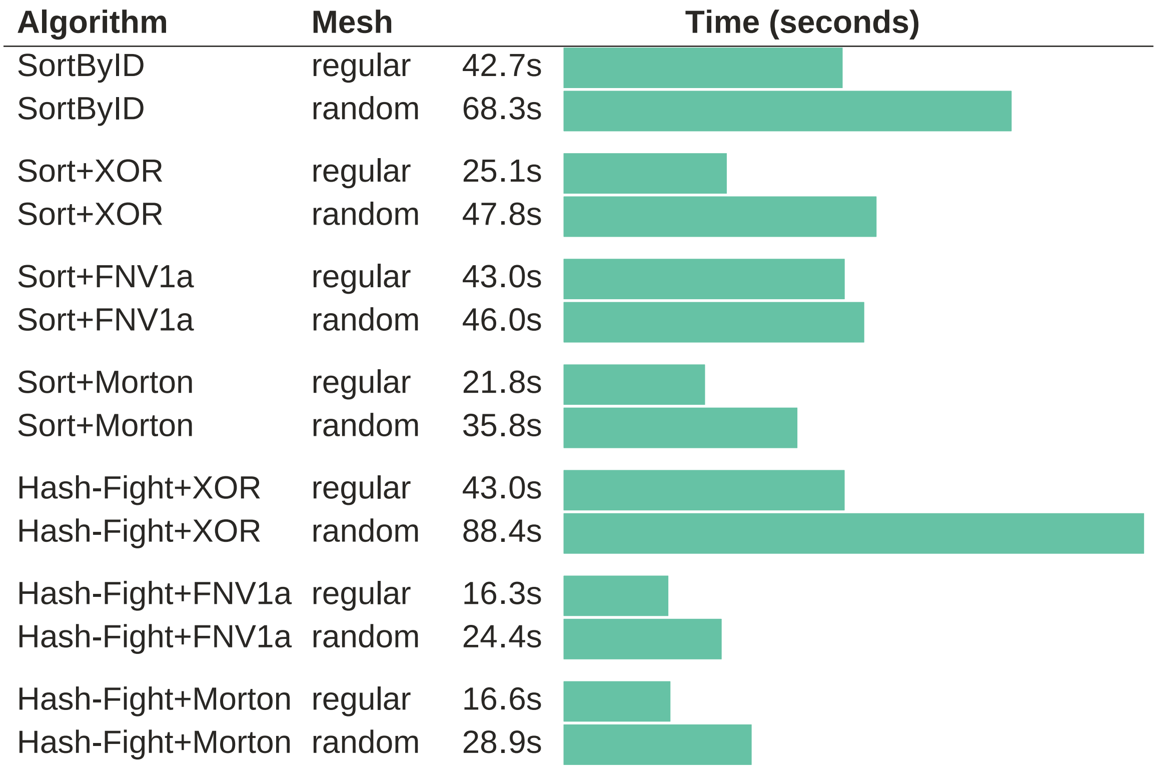
Operation	Sort XOR	Sort FNV1a	Sort Morton
prepInputFaces	0.02	0.02	0.02
hashFaces	0.01	0.01	0.01
hashSortReduce	0.13	0.15	0.15
findInternalFaces	1.09	0.13	0.11
prepOutputFaces	0.03	0.02	0.01
Total Time	1.28	0.33	0.30

Table 18. EFC runtimes (sec) for each of the primary data-parallel operations of the Sort algoritms on an Nvidia Tesla P100 GPU. Each time is the average of the 10 trials conducted on the 200^3 grid data set with approximately 40 million input cells.

Contrarily, Sort+FNV1a and Sort+Morton perform very well on the Tesla because the majority of the work involves sorting operations, which are very suitable for massive thread and data-parallelism. Using the CUDA Thrust radix sort, the runtime needed to sort the larger arrays of unique hash values for Sort+FNV1a and Sort+Morton was significantly faster than that of Intel TBB quicksort, which is used in our Haswell and KNL experiments. Replacing the Thrust radix sort with a known-to-be slower Thrust merge sort revealed the same sorting pattern from Phase 1, in which Sort+FNV1A takes longer to perform the hash value sorting than Sort+XOR and Sort+Morton. This indicates that the choice of the sort algorithm matters a lot, with the radix sort being faster in general.

9.3.3 Phase 3: Irregular Data Sets. In this phase of the study, we evaluate the performance of the algorithms when using data sets with randomized, irregular topologies. Figures 19, 20, and 21 compare algorithm runtimes of regular and randomized topologies on the Haswell, KNL, and Tesla architectures, respectively.

The XOR and FVN1a hash functions are both based on creating an index-based hash and, for CPU architectures, SortById, Sort+XOR, and Hash-Fight+XOR pay significant penalties with the randomized topology. For SortById and Sort+XOR, the initial positions of the keys are much closer to their sorted positions with the regular

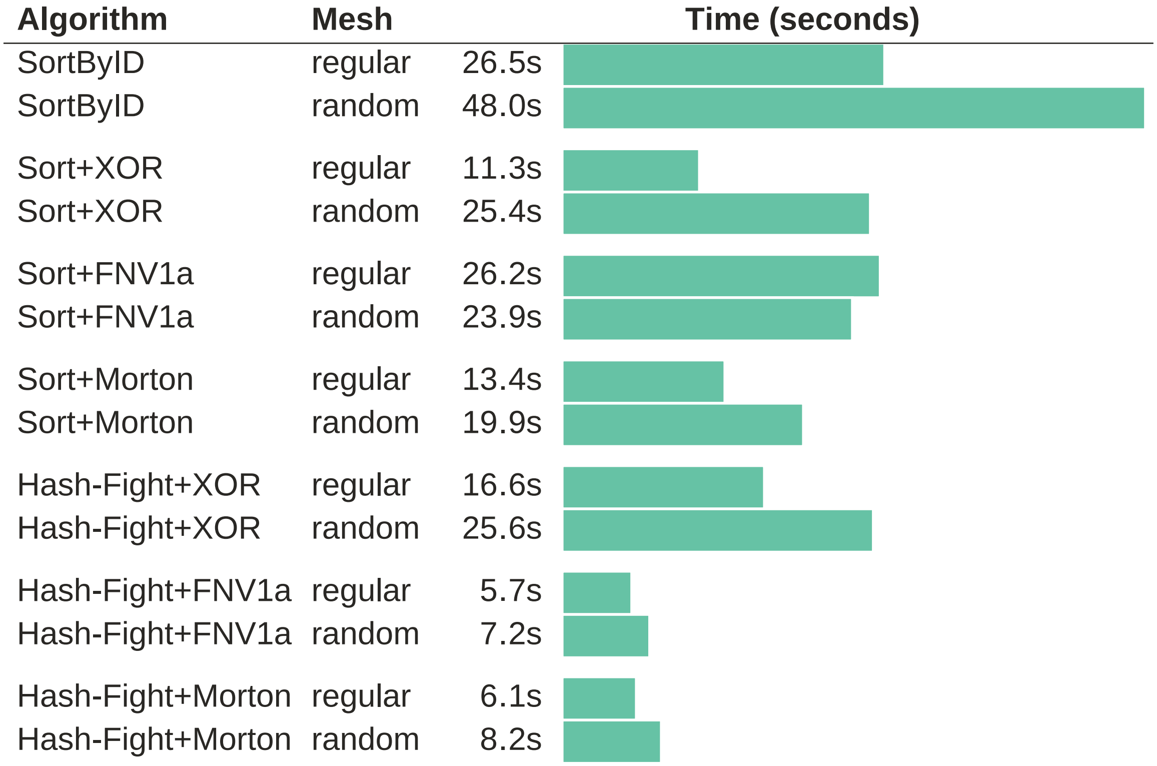


Algorithm	Mesh	Time (seconds)
SortByID	regular	42.7s
SortByID	random	68.3s
Sort+XOR	regular	25.1s
Sort+XOR	random	47.8s
Sort+FNV1a	regular	43.0s
Sort+FNV1a	random	46.0s
Sort+Morton	regular	21.8s
Sort+Morton	random	35.8s
Hash-Fight+XOR	regular	43.0s
Hash-Fight+XOR	random	88.4s
Hash-Fight+FNV1a	regular	16.3s
Hash-Fight+FNV1a	random	24.4s
Hash-Fight+Morton	regular	16.6s
Hash-Fight+Morton	random	28.9s

Figure 19. A comparison of the EFC runtime performance between regular indexing and randomized indices on Haswell with 3D data sets of 453 million tetrahedral cells (450^3 grid).

topologies than with the randomized versions. Thus, the sorting algorithm has to perform more work, leading to increased randomized topology runtimes. On the GPU, the VTK-m thrust back-end uses an optimized radix sort when keys are single 32-bit values and a merge sort for all other value types. Consequently, Sort+XOR with a randomized topology pays far less of a penalty as compared to the CPU version, while SortById actually pays a higher penalty. Conversely, the hashing properties of FNV1a distribute keys evenly with both regular and randomized topologies, leading to better performance on the KNL architecture using the randomized topology.

As seen in Figure 16 (XOR random plot), XOR has the largest number of collisions since it is a poor hash function, and the number of collisions increases linearly



Algorithm	Mesh	Time (seconds)
SortByID	regular	26.5s
SortByID	random	48.0s
Sort+XOR	regular	11.3s
Sort+XOR	random	25.4s
Sort+FNV1a	regular	26.2s
Sort+FNV1a	random	23.9s
Sort+Morton	regular	13.4s
Sort+Morton	random	19.9s
Hash-Fight+XOR	regular	16.6s
Hash-Fight+XOR	random	25.6s
Hash-Fight+FNV1a	regular	5.7s
Hash-Fight+FNV1a	random	7.2s
Hash-Fight+Morton	regular	6.1s
Hash-Fight+Morton	random	8.2s

Figure 20. A comparison of the EFC runtime performance between regular indexing and randomized indices on KNL with 3D data sets of 134 million tetrahedral cells (300^3 grid).

with the size of the data set. FNV1a and Morton perform much better as the data set size increases, as the increase in the number of duplicates is less than linear. With Morton, the number of collisions increased significantly for the largest data set. For all three hash functions, there is no significant difference in the number of collisions between the regular and random meshes.

9.4 Conclusion

We summarize our findings and best practices by phase. From Phase 1, we conclude the following:

– Use the Hash-Fight+FNV1a algorithm for consistently-optimal performance for regularly-indexed data sets.

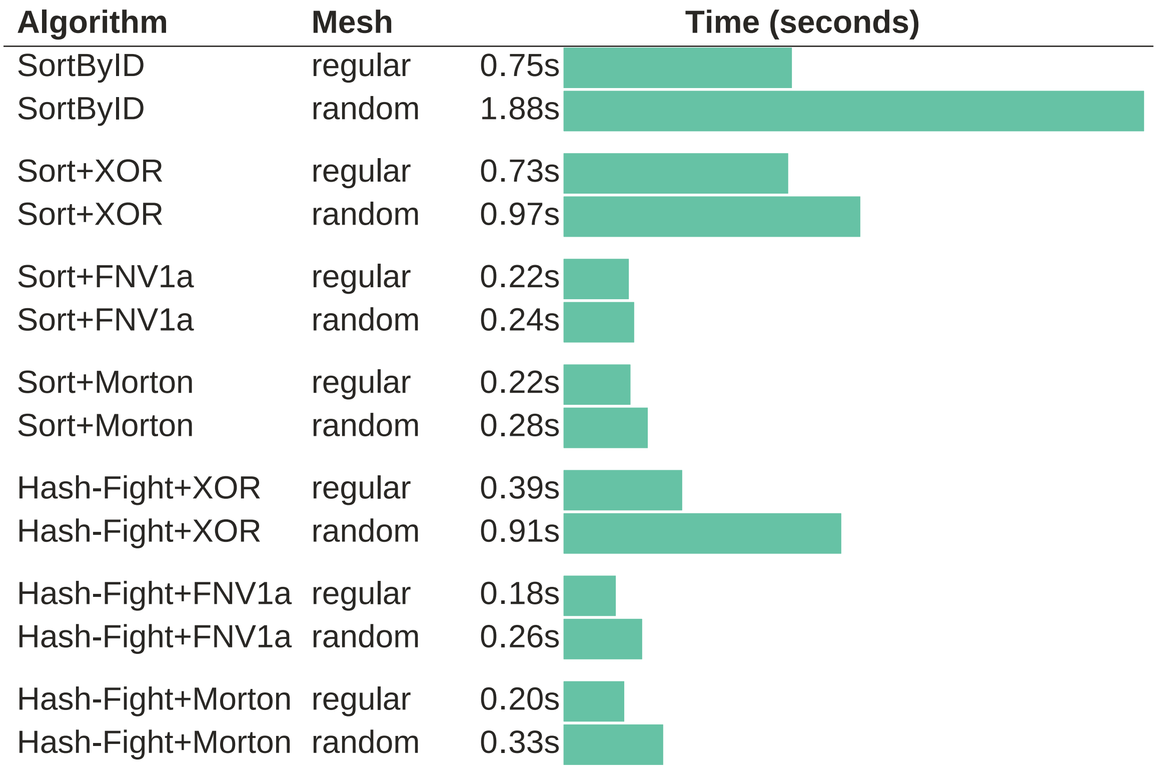


Figure 21. A comparison of the EFC runtime performance between regular indexing and randomized indices on Tesla with 3D data sets of 26 million tetrahedral cells (175^3 grid).

- Avoid the Morton hash function for large data set sizes, as it does not demonstrate robustness to hash collisions.

From Phase 2, we conclude the following:

- Use the Hash-Fight+FNV1a algorithm for optimal portable performance across varying architectures.
- Avoid the use of Sort+XOR on a GPU architecture; instead, use either Sort+FNV1a or Sort+Morton, in combination with the CUDA Thrust radix sort.

From Phase 3, we conclude the following:

- Both Sort and Hash-Fight perform best with the FNV1a hash function, which is robust to both regular and randomized mesh topologies across multiple architectures.
- Radix sort performs best overall, while quicksort performs poorly with heavily shuffled input. Further, the best-performing hash functions produce heavily shuffled input.

Overall, we believe these findings inform best practices for searching for duplicate elements with the context of DPP. The Hash-Fight+FNV1a algorithm consistently performed as a top choice in all configurations; all other algorithms suffered slowdowns in at least some configurations. The fact that the Hash-Fight configurations often beat the more traditional sorting-based algorithm is particularly interesting considering that it intentionally invokes write after write hazards. Although writing to the same or nearby memory locations from multiple threads can degrade cache performance, Hash-Fight still performs efficiently.

Overall, we believe that the best practices identified in this chapter help further address the dissertation question of the best index-based search techniques for visualization algorithms, as they pertain to duplicate element searching and EFC.

CHAPTER X

SUMMARY, FINDINGS & RECOMMENDATIONS

The overarching goal of this dissertation work is to investigate and address the dissertation question: *What are the best index-based search techniques for visualization and analysis applications on diverse many-core systems?* This concluding chapter addresses the dissertation question by synthesizing the findings and best practices of the previous chapters and offering recommendations for further research. This synthesis is provided in the form of a decision tree analysis that reflects lessons learned from studying the visualization and analysis applications of Part II. This decision tree helps guide the selection of a most-suitable index-based search technique for a given algorithm that requires a search subroutine. The choice of technique is determined by different properties related to the algorithm and search routine. This decision tree is meant to be adaptable as additional search-based visualization and analysis algorithms, techniques, parallel libraries, and many-core systems are identified or become available.

The remainder of this chapter proceeds as follows. Section 1 summarizes the contents of each dissertation chapter. Section 2 synthesizes the findings of this dissertation with guidance on selecting an index-based search technique for best platform-portable algorithm performance. Section 3 concludes the dissertation with recommendations for future work.

10.1 Summary

This dissertation consisted of nine chapters, each based primarily on previously-published, or under-submission, original research by myself and various co-authors. The content of each chapter is summarized as follows.

Chapter 1 introduced and motivated the dissertation question, as well outlining the contents of the dissertation. Chapter 2 reviewed fundamental concepts and related work for the different topical areas of this dissertation, including index-based searching, DPP-based algorithm design, and platform-portable performance. Chapter 3 introduced new DPP-based techniques for duplicate element searching. Chapter 4 presented a new DPP-based hash table and collision-resolution technique that is inspired from the hashing-based technique in Chapter 3. Chapter 5 applied the techniques of Chapter 3 to the EFC scientific visualization task, which identifies all duplicate mesh cell faces and then removes them to yield only the external, or non-duplicate faces. Chapter 6 applied the hash table technique of Chapter 4 to the task of hashing and querying hundreds of millions to billions of unsigned integer values. Chapter 7 used DPP-based sorting and hashing techniques for the search of maximal cliques within large real-world undirected graphs. Chapter 8 used DPP-based sorting and reduction techniques for the computation of the most-probable labels to assign to vertices within a graph-based image segmentation task. Finally, Chapter 9 presented a collection of best usage practices for duplicate element searching, as applied to EFC.

10.2 Synthesis of Findings

This section synthesizes the findings of this dissertation to provide insight into the dissertation question. Figures 20 and 21 present this synthesis in the form of a decision tree flowchart that guides the selection of a search technique for use within a DPP-based algorithm. We enumerate this step-by-step selection process as follows.

10.2.1 Decision: Search Routine. Given an algorithm that is to be designed in terms of DPP, the first question that must be answered is whether the algorithm contains any inherent searching routines or operations (see Figure 20). If searching is not needed, then the algorithm can be designed in terms of non-search DPP, without the search techniques introduced in this dissertation. If searching is necessary, then we proceed to the next decision, which attempts to specialize the type of search routine.

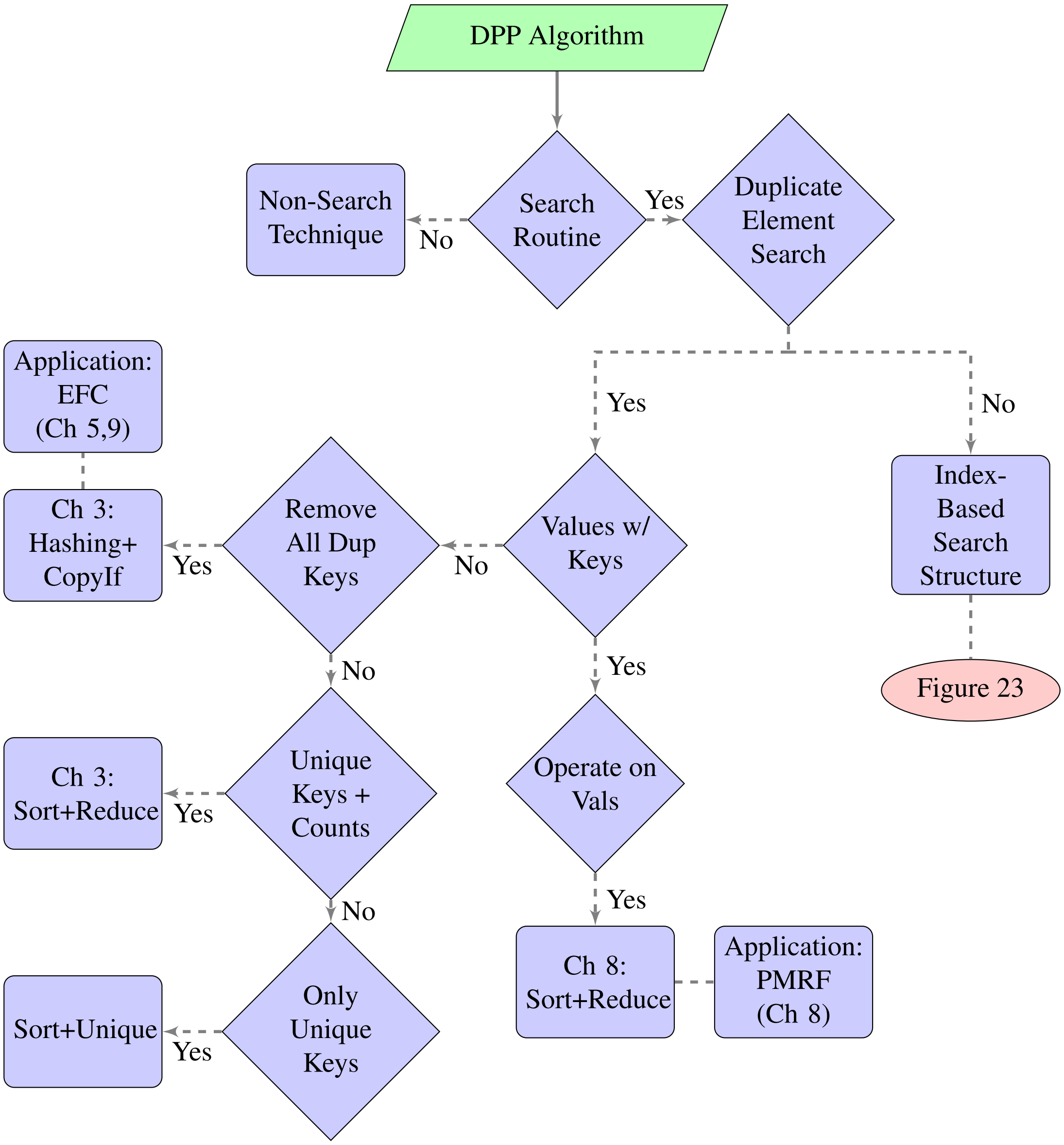


Figure 22. Decision tree flowchart that guides the selection of the most-suitable index-based search technique for a given DPP-based algorithm. This figure contains the full decision sub-tree for duplicate element search techniques.

10.2.2 Decision: Duplicate Element Search. An important decision regarding the choice of search technique is whether or not the search routine involves a search for duplicate elements. If the search routine can be recast as a search for duplicate elements of key-value pairs, then the techniques of either Chapter 3 or Chapter 8 can be used for achieving platform-portable search performance.

In particular, if the goal of the search routine is to only detect and indicate duplicate keys (e.g., point coordinates or unique integer IDs), without concern for the corresponding values, then either the sorting- or hashing-based techniques of Chapter 3 should be used. If the search routine requires all duplicates to be removed, leaving only keys with a frequency of one, then the hash-based technique should be used. In this technique, a binary indicator array is used to mark whether a key is a duplicate or not. The duplicate pairs can be removed by applying a CopyIf DPP over the indicator and key-value pair arrays. Chapters 5 and 9 provide application-specific guidance towards implementing this technique, with respect to the EFC scientific visualization problem, which is only concerned with non-duplicate keys. Otherwise, if all unique keys must be returned (e.g., to emulate a set data structure), then the key-value pairs should first be sorted in ascending order of key, followed by the application of the Unique DPP to yield only the unique keys. Likewise, if the search routine requires a frequency count for each unique key (i.e., a key that appears only once), then the key-value pairs should be sorted in ascending order of key, followed by the application of a ReduceByKey DPP. This latter technique follows the sorting-based approach that is introduced in Chapter 3 and applied in Chapters 5 and 9.

Contrarily, if the goal of the search routine is to operate over the values of duplicate keys, then the technique from Chapter 8 should be used. In Chapter 8, the key-value pairs are sorted in ascending order of key using a SortByKey DPP to group all duplicate, or equal, keys together. Then, an aggregate operator, such as summation, is applied over the values of groups of duplicate keys using a ReduceByKey DPP, resulting in a single aggregate value for each unique key (group of duplicate keys).

If the search routine does not involve duplicate element detection, but is instead a standard look-up routine for arbitrary key-value pairs (duplicate or not), then the use of a index-based data structure for storing and querying the pairs should be considered. The choice of a suitable index-based data structure is discussed as follows and is illustrated in Figure 21.

10.2.3 Decision: Index-Based Data Structure. The first decision towards selecting an index-based data structure is whether or not the search routine involves dynamically inserting new key-value pairs and/or deleting pairs. If dynamic insert and delete operations are necessary, then the SlabHash dynamic hash table from Chapter 2 should be used; a review of SlabHash is provided in Chapter 2 as part of the background on previous data-parallel hashing techniques. If key-value pairs only need to be inserted into a search structure in a single batch, without subsequent deletions, then a static, open-addressing-based hash table should be considered. This choice of using a static hash table depends on whether the general pattern of key queries is expected to be ordered, or nearly-sorted with respect to key. If an ordered query pattern is expected, then the findings of Chapter 5 suggest that better performance may be achieved by first sorting the key-value pairs by key (SortByKey DPP), and then subsequently querying for keys via a data-parallel binary search. However, if random or unordered query patterns are expected, then a static hash table should be used, as it provides efficient platform-portable performance for random-access queries. The selection of a static hash table is discussed in the following subsection.

10.2.4 Decision: Static Hash Table. A static hash table can exist either as a persistent data structure of size fN ($1 < f \leq 2$) that is pre-allocated and constructed only once with a batch of N key-value pair insertions, or as a temporary, non-persistent look-up structure of size N that is re-constructed for a new batch of N pairs as needed

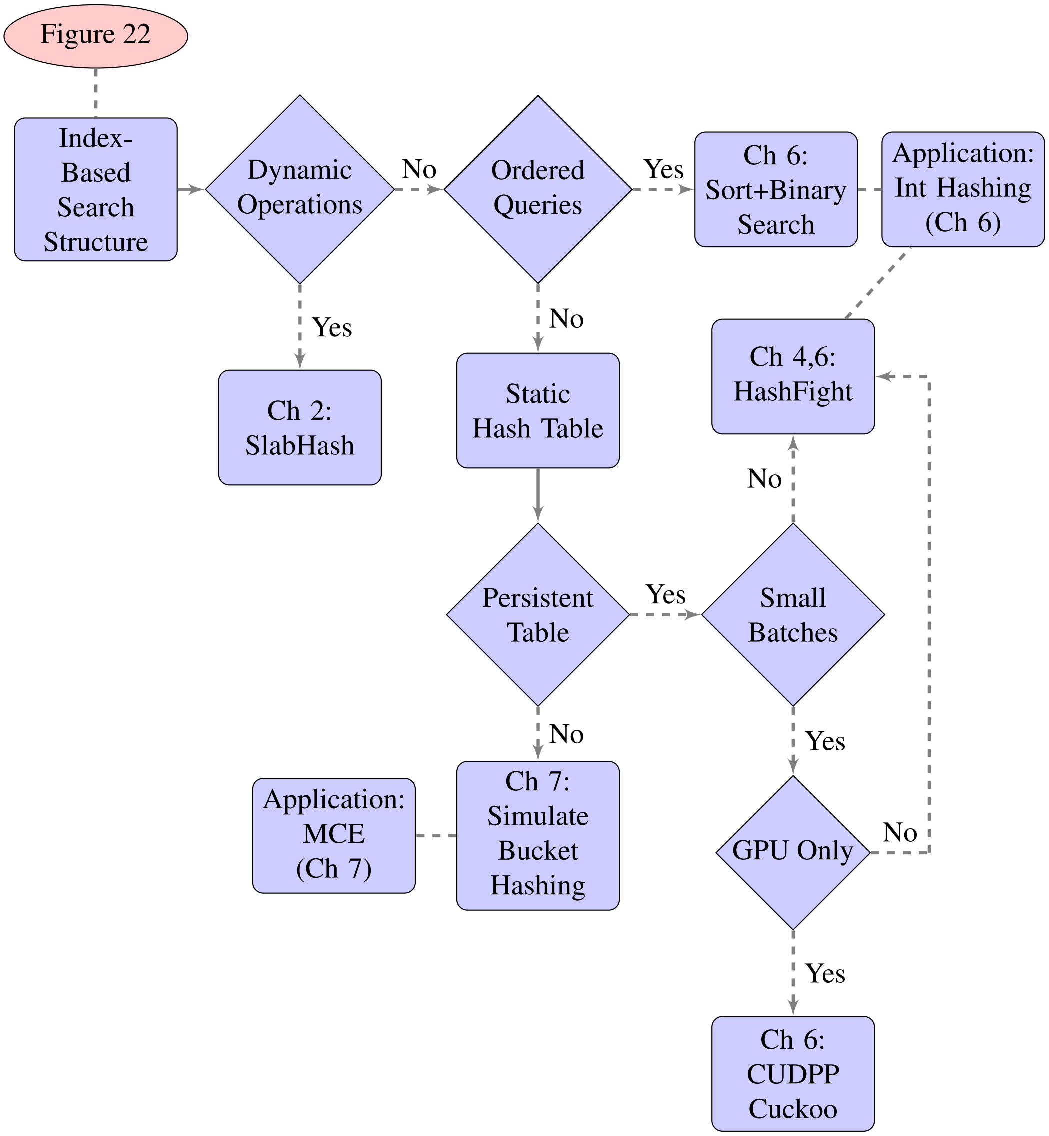


Figure 23. Continuation of the decision tree flowchart. This figure contains the decision sub-tree for the selection of the most-suitable index-based search structure, such as a hash table.

(this temporary structure differs from a dynamic hash table in that it does not support deletions or individual insertions and queries). If the search routine consists of multiple iterations, each of which can leverage a non-persistent look-up structure, then the sorting- and hashing-based technique for clique merging in Chapter VII should be used to perform queries. This technique simulates a hash table, since colliding keys with the same hash value are grouped, or bucketed, together and traditional querying is supported via linear probing within a bucket.

However, if the search routine requires a persistent static hash table, then either the HashFight or CUDPP cuckoo hashing hash tables of Chapter 6 should be used, depending on the platforms of execution, expected batch size of insertions and queries, and the existence of non-unique keys. The CUDPP hash table is designed exclusively for GPU execution and should be used when the batch sizes are less than hundreds of millions. Above this size, TLB caching and global on-device memory limitations deteriorate the performance of CUDPP, as compared to that of HashFight. For platform-portable execution on both CPUs and GPUs, and large batch sizes in the hundreds of millions to billions of pairs, the HashFight table should be used. As presented in Chapter 4 and applied in Chapter 6, HashFight demonstrates robustness to GPU memory bottlenecks and achieves leading insertion and query throughput on both CPU and GPU platforms.

10.3 Recommendations for Future Study

This concluding section presents our recommendations for future study that would build upon and further substantiate the work presented in this dissertation. These recommendations are enumerated as follows:

1. Future studies should reassess the performance of our index-based search techniques on emerging compute platforms and data-parallel coding libraries. Each technique is implemented with the open-source, platform-portable VTK-m library. As this library continues to mature, additional back-end code support for new accelerators (e.g., FPGA or new GPUs) and DPP-based libraries will be added to maintain platform-portability across algorithms.

2. The usage of 64-bit unsigned integer keys and values is prevalent and should be made available for use in the HashFight hash table, with subsequent experimentation against 64-bit variants of the CUDPP cuckoo hash table or any emerging state-of-the-art hash tables. In its current form, HashFight supports 32-bit unsigned keys and values.
3. Future studies should analyze whether the performance of our duplicate element index-based search techniques is affected by the percentage of duplicate elements in a dataset. Some datasets may have a large number of duplicates to detect—with a large number of hash collisions—while other datasets may only have a small number duplicates.
4. Additional research could focus on collecting and analyzing hardware performance counters to gain a deeper understanding of the memory utilization characteristics of our index-based search techniques.
5. Future studies should assess the performance of HashFight with spatial point-coordinate keys and the Morton curve hash function, which seeks to map spatially-close points to similar-valued unsigned integer values. In this case, the hash values would be nearby indices in the hash table, and could offer coalesced memory loads when queried in an ordered fashion.
6. The adaptable decision tree should be periodically revisited to re-evaluate the performance of our index-based search techniques on emerging compute platforms, visualization and analysis applications, and data-parallel coding libraries, as they become available. This will assure the usability of our analysis and corroborate the “future-proof” platform-portable performance that is offered by DPP-based algorithm design.

APPENDIX

LEMMAS AND THEOREMS

The following lemmas and theorem relate to properties that are used within Chapter VII for the design of a DPP-based algorithm for maximal clique enumeration (MCE).

Lemma A.0.1. *For $k \geq 2$, a $(k+1)$-clique is comprised of two k-cliques that both share $(k-1)$ vertices.*

Proof. Please refer to [70]. Figure 10 demonstrates this property by creating a 4-clique, 0-1-3-4, from two 3-cliques, 0-3-4 and 1-3-4, both of which share the 2-clique 3-4. Effectively, once two k-cliques with matching (and trailing) $(k-1)$ vertices are found, we only need to test whether the leading vertices are connected; if so, the two k-cliques can be merged into a new $(k+1)$-clique. □

Lemma A.0.2. *An expanded $(k+1)$-clique maintains the ascending vertex order.*

Proof. In our algorithm, a k-clique is only matched with another k-clique that that a higher leading vertex Id. Both of the leading vertices of these two k-cliques have lower Ids and are distinct from the vertices of the matching $(k-1)$-clique. By induction, this $(k-1)$-clique must also be in ascending vertex order. Thus, the expanded $(k+1)$-clique must possess an ascending vertex Id order. □

Theorem A.0.3. *During iteration k, there are no duplicate k-cliques.*

Proof. We will prove by induction on the size k.

- Base case of $k = 2$. The starting set of 2-cliques are the edges from the *v-graph* edge list, all of which are unique, since duplicate edges were removed in the initialization routine.

- Induction hypothesis. There are no duplicate k-cliques in iteration k.
- Inductive step. No duplicate $(k+1)$-cliques exist in iteration $k+1$. In the previous iteration k, a $(k+1)$-clique was produced via a merging between two k-cliques that shared a trailing $(k-1)$-clique (see Theorem A.0.1). If duplicate copies of this $(k+1)$-clique existed, then duplicate pairs of the k-cliques must have also existed, since any two k-cliques can only produce one new $(k+1)$-clique in our algorithm (see proof of Theorem A.0.2). However, by the induction hypothesis, there are no duplicate k-cliques. Thus, by contradiction, a $(k+1)$-clique cannot have duplicates.

□

REFERENCES CITED

[1] E. A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM Journal on Computing*, 2(1):1–6, 1973.

[2] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the GPU. In *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pages 154:1–154:9, New York, NY, USA, 2009. ACM.

[3] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Chapter 4 - Building an efficient hash table on the GPU. In Wen-mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 39 – 53. Morgan Kaufmann, Boston, 2012.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[5] S. Ashkiani, M. Farach-Colton, and J. D. Owens. A Dynamic Hash Table for the GPU. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium*, IPDPS '18, pages 419–429, May 2018.

[6] Saman Ashkiani, Andrew 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 '16, pages 12:1–12:13, 2016.

[7] Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D. Owens. GPU LSM: A dynamic dictionary data structure for the GPU. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium*, IPDPS '18, pages 430–440, May 2018.

[8] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. In W.-M. Hwu, editor, *GPU Computing Gems*, pages 359–371. Elsevier/Morgan Kaufmann, 2011.

[9] Jonathan W. Berry, Bruce Hendrickson, Simon Kahan, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–14, 2007.

[10] E. Wes Bethel, Martin Greenwald, Kerstin Kleese van Dam, Manish Parashar, Stefan M. Wild, and H. Steven Wiley. Management, Analysis, and Visualization of Experimental and Observational Data – The Convergence of Data and Computing. In *Proceedings of the 2016 IEEE 12th International Conference on eScience*, Baltimore, MD, USA, October 2016.

[11] Guy E Blelloch. *Vector models for data-parallel computing*, volume 75. MIT press Cambridge, 1990.

[12] R. Blikberg and T. Sørevik. Load balancing and OpenMP implementation of nested parallelism. *Parallel Computing*, 31(10):984 – 998, 2005. OpenMP.

[13] Boost C++ Libraries. Boost.Iterator Library, 2003. `http://www.boost.org/doc/libs/1_65_1/libs/iterator/doc/index.html`.

[14] Rajesh Bordawekar. Evaluation of parallel hashing techniques. In *GPU Technology Conference*, March 2014.

[15] Fabiano C. Botelho and Nivio Ziviani. External perfect hashing for very large key sets. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 653–662, New York, NY, USA, 2007. ACM.

[16] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[17] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, September 1973.

[18] Aydin Buluç and John R. Gilbert. The Combinatorial BLAS: design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[19] Hamish A. Carr, Gunther H. Weber, Christopher M. Sewell, and James P. Ahrens. Parallel peak pruning for scalable SMP contour tree computation. In *6th IEEE Symposium on Large Data Analysis and Visualization, LDAV 2016, Baltimore, MD, USA, October 23-28, 2016*, pages 75–84, 2016.

[20] Daniel Cederman, Bapi Chatterjee, and Philippas Tsigas. Understanding the performance of concurrent data structures on graphics processors. In *Proceedings of the 18th International Conference on European Parallel Processing*, Euro-Par 2012, pages 883–894, August 2012.

[21] Pedro Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, Canada, 1986.

[22] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos. Design and evaluation of parallel hashing over large-scale data. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2014.

[23] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, February 1985.

[24] J. Choi, J. Demmel, Inderjit S. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers – design issues and performance. *Computer Physics Communications*, 97, aug 1996.

[25] Z. Choudhury, S. Purini, and S. R. Krishna. A hybrid CPU+GPU working-set dictionary. In *2016 15th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 56–63, July 2016.

[26] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[27] CUDA Data Parallel Primitives Library. `http://cudpp.github.io`, November 2017.

[28] Naga Shailaja Dasari, Desh Ranjan, and Zubair Mohammad. Maximal clique enumeration for large graphs on hadoop framework. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pages 21–30, New York, NY, USA, 2014. ACM.

[29] Andrew Davidson, David Tarjan, Michael Garland, and John D. Owens. Efficient parallel merge sort for fixed and variable length keys. In *Innovative Parallel Computing*, page 9, May 2012.

[30] Peter J. Denning. The Locality Principle. *Commun. ACM*, 48(7):19–24, July 2005.

[31] David Dice, Danny Hendler, and Ilya Mirsky. Lightweight contention management for efficient compare-and-swap operations. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 595–606, Berlin, Heidelberg, 2013. Springer-Verlag.

[32] DIMACS - The Maximum Cliques Problem. `http://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark`, June 2017.

[33] J. Donatelli et al. Camera: The center for advanced mathematics for energy research applications. *Synchrotron Radiation News*, 28(2):4–9, 2015.

[34] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[35] N. Du, B. Wu, L. Xu, B. Wang, and X. Pei. A parallel algorithm for enumerating all maximal cliques in complex network. In *Sixth IEEE International Conference on Data Mining - Workshops (ICDMW'06)*, pages 320–324, Dec 2006.

[36] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Algorithms and Computation: 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part I*, pages 403–414. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[37] David Eppstein and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. In Panos M. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms: 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*, pages 364–375. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[38] Paul F. Fischer, James W. Lottes, and Stefan G. Kerkemeier. nek5000 Web page, 2008. http://nek5000.mcs.anl.gov.

[39] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[40] Glenn Fowler, Landon Curt Noll, Kiem-Phong Vo, Donald Eastlake, and Tony Hansen. The FNV non-cryptographic hash algorithm. Technical report, Network Working Group, 2017. `https://tools.ietf.org/html/draft-eastlake-fnv-13`.

[41] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, June 1984.

[42] Gianluca Frison, Dimitris Kouzoupis, Andrea Zanelli, and Moritz Diehl. BLASFEO: basic linear algebra subroutines for embedded optimization. *CoRR*, abs/1704.02457, 2017.

[43] Heng Gao, Jie Tang, and Gangshan Wu. Parallel surface reconstruction on GPU. In *Proceedings of the 7th International Conference on Internet Multimedia Computing and Service*, ICIMCS '15, pages 54:1–54:5, New York, NY, USA, 2015. ACM.

[44] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent parallel hashing. *ACM Trans. Graph.*, 30(6):161:1–161:8, December 2011.

[45] E. L. Goodman, D. J. Haglin, C. Scherrer, D. Chavarría-Miranda, J. Mogill, and J. Feo. Hashing strategies for the Cray XMT. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pages 1–8, April 2010.

[46] Michael Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 260–269, New York, NY, USA, 2002. ACM.

[47] John L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, May 1988.

[48] Mark Harris. Maxwell: The Most Advanced CUDA GPU Ever Made. `https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/`, September 2014.

[49] C. Harrison, H. Childs, and K. P. Gaither. Data-parallel mesh connected components labeling and analysis. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '11, pages 131–140, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association.

[50] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. The stapl parallel graph library. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing: 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers*, pages 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[51] X. He, D. Agarwal, and S. K. Prasad. Design and implementation of a parallel priority queue on many-core architectures. In *2012 19th International Conference on High Performance Computing*, pages 1–10, Dec 2012.

[52] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[53] Intel Corporation. Intel Thread Building Blocks Documentation, January 2019. `https://software.intel.com/en-us/tbb-documentation`.

[54] Jim Jeffers and James Reinders. *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, volume 2. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2015.

[55] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *CoRR*, abs/1804.06826, 2018.

[56] H. C. Johnston. Cliques of a graph-variations on the Bron-Kerbosch algorithm. *International Journal of Computer & Information Sciences*, 5(3):209–238, 1976.

[57] Tim Kaldewey and Andrea Di Blas. Large-scale GPU search. In *GPU Computing Gems Jade Edition*, pages 3–14, 12 2012.

[58] Javor Kalojanov, Markus Billeter, and Philipp Slusallek. Two-level grids for ray tracing on GPUs. *Computer Graphics Forum*, 30(2):307–314, 2011.

[59] Javor Kalojanov and Philipp Slusallek. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 23–28, New York, NY, USA, 2009. ACM.

[60] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, August 2003.

[61] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. Big data causing big (TLB) problems: Taming random memory accesses on the GPU. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, DAMON '17, pages 6:1–6:10, New York, NY, USA, 2017. ACM.

[62] Tero Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In Carsten Dachsbacher, Jacob Munkberg, and Jacopo Pantaleoni, editors, *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, pages 33–37. The Eurographics Association, 2012.

[63] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, José E. Moreira, John D. Owens, Carl Yang, Marcin Zalewski, and Timothy G. Mattson. Mathematical foundations of the GraphBLAS. *CoRR*, abs/1606.05790, 2016.

[64] F. Khorasani, M. E. Belviranli, R. Gupta, and L. N. Bhuyan. Stadium hashing: Scalable and flexible hashing on GPUs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 63–74, Oct 2015.

[65] 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, New York, NY, USA, 2010. ACM.

[66] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[67] Ina Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1):1 – 30, 2001.

[68] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[69] V. Kolmogorov and R. Zabin. What energy functions can be minimized via graph cuts? *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(2):147–159, Feb 2004.

[70] F. Kose, W. Weckwerth, T. Linke, and O. Fiehn. Visualizing plant metabolomic correlation networks using clique-metabolite matrices. *Bioinformatics*, 17(12):1198–1208, 2001.

[71] Ares Lagae and Philip Dutré. Compact, fast and robust grids for ray tracing. In *ACM SIGGRAPH 2008 Talks*, SIGGRAPH '08, pages 20:1–20:1, New York, NY, USA, 2008. ACM.

[72] Zhuohang Lai, Qiong Luo, and Xiaoying Jia. Revisiting multi-pass scatter and gather on GPUs. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 25:1–25:11, New York, NY, USA, 2018. ACM.

[73] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[74] Matthew Larsen, Stephanie Labasan, Paul Navrátil, Jeremy Meredith, and Hank Childs. Volume Rendering Via Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 53–62, Cagliari, Italy, May 2015.

[75] Matthew Larsen, Jeremy Meredith, Paul Navrátil, and Hank Childs. Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium*, pages 279–286, Hangzhou, China, April 2015.

[76] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[77] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, 1980.

[78] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 579–588, New York, NY, USA, 2006. ACM.

[79] Brent Lessley, Roba Binyahib, Robert Maynard, and Hank Childs. External Facelist Calculation with Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 10–20, Groningen, The Netherlands, June 2016.

[80] Brenton Lessley. Data-parallel hashing techniques for GPU architectures. *CoRR*, abs/1807.04345, 2018.

[81] Brenton Lessley, Kenneth Moreland, Matthew Larsen, and Hank Childs. Techniques for Data-Parallel Searching for Duplicate Elements. In *Proceedings of IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 1–5, Phoenix, AZ, October 2017.

[82] Brenton Lessley, Talita Perciano, Colleen Heinemann, David Camp, Hank Childs, and E. Wes Bethel. DPP-PMRF: Rethinking Optimization for a Probabilistic Graphical Model Using Data-Parallel Primitives. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*, Eugene, OR, Under review.

[83] Brenton Lessley, Talita Perciano, Manish Mathai, Hank Childs, and E. Wes Bethel. Maximal Clique Enumeration with Data-Parallel Primitives. In *Proceedings of IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 16–25, Phoenix, AZ, October 2017.

[84] John Levesque and Aaron Vose. *Programming for Hybrid Multi/Many-core MPP Systems*. Chapman & Hall, CRC Computational Science. CRC Press/Francis&Taylor Group, Boca Raton, FL, USA, November 2017. preprint.

[85] O. Lezoray and L. Grady. *Image Processing and Analysis with Graphs: Theory and Practice*. CRC Press, 2012.

[86] S. Z. Li. *Markov Random Field Modeling in Image Analysis*. Springer Publishing Company, 2013.

[87] Shaomeng Li, Nicole Marsaglia, Vincent Chen, Christopher Sewell, John Clyne, and Hank Childs. Achieving Portable Performance For Wavelet Compression Using Data Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 73–81, Barcelona, Spain, June 2017.

[88] John D. C. Little. Little's Law as viewed on its 50th anniversary. *Oper. Res.*, 59(3):536–549, May 2011.

[89] Li-ta Lo, Christopher Sewell, and James P Ahrens. Piston: A portable cross-platform framework for data-parallel visualization operators. In *EGPGV*, pages 11–20, 2012.

[90] L. Lu, Y. Gu, and R. Grossman. dmaximalcliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution. In *2010 IEEE International Conference on Data Mining Workshops*, pages 1320–1327, Dec 2010.

[91] L. Luo, M. D. F. Wong, and L. Leong. Parallel implementation of r-trees on the gpu. In *17th Asia and South Pacific Design Automation Conference*, pages 353–358, Jan 2012.

[92] D. Mahapatra and Y. Sun. Integrating Segmentation Information for Improved MRF-Based Elastic Image Registration. *IEEE Transactions on Image Processing*, 21(1):170–183, Jan 2012.

[93] Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004*, pages 260–272. Springer Berlin Heidelberg, 2004.

[94] Robert Maynard, Kenneth Moreland, Utkarsh Atyachit, Berk Geveci, and Kwan-Liu Ma. Optimizing threshold for extreme scale analysis. In *IS&T/SPIE Electronic Imaging*, pages 86540Y–86540Y. International Society for Optics and Photonics, 2013.

[95] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[96] Z. Meng, D. Wei, A. Wiesel, and A. O. Hero. Distributed learning of gaussian graphical models via marginal likelihoods. In *The Sixteenth International Conference on Artificial Intelligence and Statistics*, pages 39–47, 2013.

[97] Z. Meng, D. Wei, A. Wiesel, and A. O. Hero. Marginal likelihoods for distributed parameter estimation of gaussian graphical models. In *IEEE Transactions on Signal Processing*, volume 62, pages 5425–5438, 2014.

[98] Jeremy S. Meredith, Sean Ahern, Dave Pugmire, and Robert Sisneros. EAVL: The Extreme-scale Analysis and Visualization Library. In Hank Childs, Torsten Kuhlen, and Fabio Marton, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2012.

[99] Jeremy S. Meredith and Hank Childs. Visualization and Analysis-Oriented Reconstruction of Material Interfaces. *Computer Graphics Forum (CGF)*, 29(3):1241–1250, June 2010.

[100] Duane G. Merrill and Andrew S. Grimshaw. Revisiting sorting for GPGPU stream architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 545–546, New York, NY, USA, 2010. ACM.

[101] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[102] Microsoft Corporation. Parallel Patterns Library Documentation, January 2019. `https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=vs-2017`.

[103] Prabhakar Misra and Mainak Chaudhuri. Performance evaluation of concurrent lock-free data structures on GPUs. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, ICPADS '12, pages 53–60, Washington, DC, USA, 2012. IEEE Computer Society.

[104] Y. D. Mizrahi, M. Denil, and N. de Freitas. Linear and parallel learning of markov random fields. In *Proceedings of International Conference on Machine Learning*, volume 32, pages 1–10, 2014.

[105] M. Moazeni and M. Sarrafzadeh. Lock-free hash table on graphics processors. In *2012 Symposium on Application Accelerators in High Performance Computing*, pages 133–136, July 2012.

[106] J. W. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3(1):23–28, 1965.

[107] K. Moreland, U. Ayachit, B. Geveci, and K. L. Ma. Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *2011 IEEE Symposium on Large Data Analysis and Visualization*, pages 97–104, Oct 2011.

[108] K. Moreland and R. Oldfield. Formal metrics for large-scale parallel performance. In *Proceedings of International Supercomputing Conference*, 2015.

[109] Kenneth Moreland, Christopher Sewell, William Usher, Lita Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, Matthew Larsen, Chun-Ming Chen, Robert Maynard, and Berk Geveci. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications (CG&A)*, 36(3):48–58, May/June 2016.

[110] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Ontario, Canada, 1966.

[111] N. Moscovici, N. Cohen, and E. Petrank. A GPU-friendly skiplist algorithm. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 246–259, Sept 2017.

[112] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (TOG)*, 2013.

[113] R. Nock and F. Nielsen. Statistical region merging. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(11):1452–1458, Nov 2004.

[114] Nvidia Corporation. CUDA C Best Practices Guide. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`, September 2017.

[115] Nvidia Corporation. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, September 2017.

[116] Nvidia Corporation. Parallel Thread Execution ISA Version 6.0. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html, September 2017.

[117] Nvidia Corporation. Thrust, January 2018. https://thrust.github.io.

[118] B. W. O'Shea, G. Bryan, J. Bordner, M. L. Norman, T. Abel, R. Harkness, and A. Kritsuk. Introducing Enzo, an AMR Cosmology Application. *ArXiv Astrophysics e-prints*, March 2004.

[119] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[120] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[121] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[122] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[123] T. Perciano, D. Ushizima, H. Krishnan, D. Parkinson, N. Larson, D. M. Pelt, W. Bethel, F. Zok, and J. Sethian. Insight into 3D micro-CT data: exploring segmentation algorithms through performance metrics. *Journal of Synchrotron Radiation*, 24(5):1065–1077, Sep 2017.

[124] T. Perciano, D. M. Ushizima, E. W. Bethel, Y. D. Mizrahi, D. Parkinson, and J. A. Sethian. Reduced-complexity image segmentation under parallel markov random field formulation using graph partitioning. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 1259–1263, Sept 2016.

[125] P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[126] Orestis Polychroniou and Kenneth A. Ross. High throughput heavy hitter aggregation for modern SIMD processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN ’13, pages 6:1–6:6, New York, NY, USA, 2013. ACM.

[127] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O’Reilly, July 2007.

[128] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS ’09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[129] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. K-ary search on modern processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN ’09, pages 52–60, New York, NY, USA, 2009. ACM.

[130] Matthew C. Schmidt, Nagiza F. Samatova, Kevin Thomas, and Byung-Hoon Park. A scalable, parallel algorithm for maximal clique enumeration. *Journal of Parallel and Distributed Computing*, 69(4):417–428, 2009.

[131] J. Schneider and P. Rautek. A versatile and efficient GPU data structure for spatial indexing. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):911–920, Jan 2017.

[132] Hendrik A. Schroots and Kwan-Liu Ma. Volume Rendering with Data Parallel Visualization Frameworks for Emerging High Performance Computing Architectures. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing*, SA ’15, pages 3:1–3:4. ACM, 2015.

[133] Thomas R.W. Scogland and Wu-chun Feng. Design and evaluation of scalable concurrent queues for many-core architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE ’15, pages 63–74, New York, NY, USA, 2015. ACM.

[134] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.

[135] J. Siek, L.Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[136] Dhirendra Pratap Singh, Ishan Joshi, and Jaytrilok Choudhary. Survey of gpu based sorting algorithms. *International Journal of Parallel Programming*, Apr 2017.

[137] Stanford Large Network Dataset Collection. `https://snap.stanford.edu/data/`, June 2017.

[138] Jeff A. Stuart and John D. Owens. Efficient synchronization primitives for GPUs. *CoRR*, abs/1110.4623(1110.4623v1), October 2011.

[139] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.

[140] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday paradox for multi-collisions. In Min Surp Rhee and Byoungcheon Lee, editors, *Information Security and Cryptology – ICISC 2006: 9th International Conference, Busan, Korea, November 30 - December 1, 2006. Proceedings*, pages 29–40. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[141] Michael Svendsen, Arko Provo Mukherjee, and Srikanta Tirthapura. Mining maximal cliques from a large graph using mapreduce: Tackling highly uneven subproblem sizes. *Journal of Parallel and Distributed Computing*, 79:104–114, 2015.

[142] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, October 2006.

[143] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

[144] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[145] VisIt. avtfacelistfilter, April 2016. https://github.com/visit-vis/VisIt/blob/master/avt/Filters/avtFacelistFilter.C.

[146] Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016.

[147] VTK. vtkunstructuredgridgeometryfilter, April 2016. http://www.vtk.org/doc/nightly/html/class/vtkUnstructuredGridGeometryFilter.html.

[148] VTK-m. `https://gitlab.kitware.com/vtk/vtk-m`, November 2017.

[149] VTK-m. `https://gitlab.kitware.com/vtk/vtk-m`, June 2017.

[150] W. Widanagamaachchi, P. T. Bremer, C. Sewell, L. T. Lo, J. Ahrens, and V. Pascuccik. Data-parallel halo finding with variable linking lengths. In *2014 IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 27–34, Nov 2014.

[151] Bin Wu, Shengqi Yang, Haizhou Zhao, and Bai Wang. A distributed algorithm to enumerate all maximal cliques in mapreduce. In *Proceedings of the 2009 Fourth International Conference on Frontier of Computer Science and Technology*, FCST '09, pages 45–51, Washington, DC, USA, 2009. IEEE Computer Society.

[152] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the GPU. In *Proceedings of the 21st Eurographics Conference on Rendering*, EGSR'10, pages 1297–1304, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[153] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.*, 8(11):1226–1237, July 2015.

[154] Yun Zhang, Faisal N. Abu-Khzam, Nicole E. Baldwin, Elissa J. Chesler, Michael A. Langston, and Nagiza F. Samatova. Genome-scale computational approaches to memory-intensive applications in systems biology. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 12–12, Washington, DC, USA, 2005. IEEE Computer Society.

[155] K. Zhou, M. Gong, X. Huang, and B. Guo. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):669–681, May 2011.

[156] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time KD-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 126:1–126:11, New York, NY, USA, 2008. ACM.