Tackling GPU Memory Size Limitations

By

AFTON NOELLE GEIL
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

John D. Owens, Chair

Nina Amenta

Kent Wilken

Committee in Charge

2023

To my parents, Kathy and Dan, my brother, Eric,
and my lifelong friends, Autumn and Madison.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

ABSTRACT

Tackling GPU Memory Size Limitations

GPUs are now in widespread use for many non-graphics applications, like machine learning, scientific computations, and computer vision, but many challenges remain for achieving their full potential in many areas. Some algorithms and data structure operations, originally developed with sequential CPU architectures in mind, appear to be inherently serial in nature, and require new methods to adapt them to take advantage of the many-core GPU architecture. This dissertation describes methods for utilizing this massive parallelism to solve problems on large datasets while also grappling with the limitations on GPU memory size.

First, we present an approach to maximum clique enumeration (finding all maximum cliques in a graph) on the GPU via an iterative breadth-first traversal of the search tree. In order to achieve high performance on the GPU, our implementation aims to maximize available parallelism and minimize divergence between threads. The biggest challenge for a breadth-first implementation is the memory required to store all of the intermediate clique candidates. To mitigate this issue, we employ a variety of strategies to prune away non-maximum candidates and present a thorough examination of the performance and memory benefits of each of these options. We also explore a windowing strategy as a middle-ground between breadth-first and depth-first approaches, and investigate the resulting trade-off between parallel efficiency and memory usage. Our results demonstrate that when we are able to manage the memory requirements, our approach achieves high throughput for large graphs indicating this approach is a good choice for GPU performance. We demonstrate an average speedup of 1.9x over previous parallel work, and obtain our best performance on graphs with low average degree.

Finally, we present our GPU implementation of the quotient filter, a compact data structure designed to implement approximate membership queries. The quotient filter is similar to the more well-known Bloom filter; however, in addition to set insertion and membership queries, the quotient filter also supports deletions and merging filters without requiring rehashing of the data set. Furthermore, the quotient filter can be extended to include counters without increasing the memory footprint. We implement two types of quotient filters on the GPU: the standard

quotient filter and the rank-and-select-based quotient filter. We describe the parallelization of all filter operations, including a comparison of the four different methods we devised for parallelizing quotient filter construction. In solving this problem, we found that we needed an operation similar to a parallel scan, but for non-associative operators. One outcome of this work is a variety of methods for computing parallel scan-type operations on a non-associative operator.

For membership queries, we achieve a throughput of up to 1.13 billion items/second for the rank-and-select-based quotient filter: a speedup of 3x over the BloomGPU filter. Our fastest filter build method achieves a speedup of 2.1–3.1x over BloomGPU, with a peak throughput of 621 million items/second, and a rate of 516 million items/second for a 70% full filter. However, we find that our filters do not perform incremental updates as fast as the BloomGPU filter. For a batch of 2 million items, we perform incremental inserts at a rate of 81 million items/second – a 2.5x slowdown compared to BloomGPU's throughput of 201 million items/second. The quotient filter's memory footprint is comparable to that of a Bloom filter.

ACKNOWLEDGMENTS

Throughout my many, *many* years working to produce this dissertation, I received support from so many wonderful people, without whom this would have been impossible, or at the very least much less pleasant.

First and foremost, I would like to thank my advisor, John Owens, for his support, guidance, and patience throughout my graduate career. I am deeply grateful and still baffled that you believed in my ability to do this work, even when I entered graduate school barely knowing what a GPU was. Thank you for giving me the freedom to work on whatever problems interested me, even when I had a hard time figuring out what those were. Your wisdom and enthusiasm have been invaluable. Keep being awesome.

Many thanks to Martin Farach-Colton for helping me understand the theory behind quotient filters, contributing some of the most clever ideas in that work, and teaching me the importance of being "technically correct". I am thankful to have had the joy of working on my first paper with Yangzihao Wang, a brilliant and thoughtful human being. Thank you for allowing me to be part of the early days of Gunrock's life. I am also grateful to Serban Porumbescu for his advice and ideas throughout the last few years of my graduate career, and for his many late hours going over my writing with a scalpel. I would also like to thank Ryan Rossi for sharing his time and knowledge with me as I was getting started on the maximum clique work. Our talks were so helpful in getting me started thinking about the problem in general and beginning to formulate my ideas for approaching a GPU implementation.

Thank you to my dissertation committee members, Nina Amenta and Kent Wilken, for your valuable time, feedback, and general kindness. And thanks to the additional members of my qualifying exam committee, Venkatesh Akella and Soheil Ghiasi, for your ideas that also helped guide this work. Thank you also to my coworkers from my summer internships at Intel, who helped me to expand my knowledge and skills early on. Thanks especially to my mentor, Aaron Kunze, for your patience, wisdom, and good conversation.

Over the years, I have had technical and moral support from my many talented labmates. Thank you all for your help with debugging, brainstorming, and sharing in moments of joy and commiserating in times of disappointment. I know the Owens lab will be strong for many years

to come.

This work would not have been possible without financial support from many funding sources. I am especially grateful to the National Science Foundation for entrusting me with a Graduate Research Fellowship. And thank you to the taxpayers for supporting public universities and research funding so that I could do this work.

Of course, I would like to thank my parents, Dan Geil and Kathy Geil, who always told me I was a genius (doubtful) and could do anything I wanted, even when they didn't understand why I wanted it. To my brother, Eric Geil, thank you for your support and friendship, and also for being the one with a Real Job, so our parents could take comfort in the knowledge that at least one of us can survive in society. To my friends Autumn Losey-Bailor and Madison Binkle, who I also consider family, I am so thankful for the ample emotional support you have provided throughout my life, especially during my time in grad school. Thank you, Autumn, for being such a reliably understanding and comforting source of support and for helping me with encouragement and accountability during some of the toughest times. To Madison, thank you for being an amazing cheerleader, sounding board, and companion for many adventures.

I would like to thank Jacob Roche for his support and companionship throughout the majority of my time in grad school. And to George Michael, the absolute best boy, thanks for all the cuddles. I could not have asked for better people to survive the pandemic with than y'all.

Thank you to all of my friends from grad school and before. I won't try to name everyone here, but I'd hate to imagine what my life would be without you all. Thanks to the 915 crew for creating a welcoming environment and helping to make grad school life fun. I am also grateful for my time in Sensual Daydreams Rocky Horror shadow cast and all of my amazing castmates. It was a joy to perform with you all, and I learned so much about myself through those experiences, which I will surely treasure forever. And to my dear Hellmouth friends: enjoy. x

Finally, as I complete this dissertation in the middle of a historic strike across the University of California system, I would like to send thanks and solidarity to my fellow UAW strikers fighting to ensure that future academic researchers can feel secure while they continue to do important work generating knowledge and solving problems.

Chapter 1

Introduction

As we quantify more aspects of our world, the amount of data generated each day has grown exponentially and led to many new discoveries and enabled a variety of valuable services. Relational data describes connections between people, places, and/or things, and is typically represented as a graph. Graph analytics can help us understand biological and chemical mechanisms, identify close communities in social networks, or improve transportation networks. One useful graph feature is cliques – sets of fully connected vertices. The maximum clique(s) of a graph are the largest group(s) of fully connected data points. Finding maximum cliques is a combinatorial problem, and can be challenging even for relatively small datasets, and the difficulty only increases as the datasets increase in size. Adding to the challenge, many datasets are updated frequently, possibly resulting in a change in the maximum clique(s).

As datasets across many applications continue to grow, it has become essential to process large quantities of data in parallel. Graphics processing units (GPUs) are now used for many general purpose computations, far beyond their original use in graphics. The GPU architecture provides opportunities for massive parallelism, with hundreds of thousands of threads active at once. They also achieve significantly better performance per Watt, thereby reducing the cost and carbon footprint for solving large computational problems.

In order to maximize these benefits, we should tailor our implementations to the GPU architecture, which often requires a different approach than an implementation of the same problem on a CPU. We must break the problem down into many pieces that can be solved in parallel. For problems with a large amount of independent data-parallel work available this can be fairly

straightforward, but often it is more challenging, and may not be possible for every problem. In addition to formulating the problem in a parallel manner, there are other techniques that can help to optimize GPU performance, including avoiding thread divergence, workload balancing, coalescing memory accesses, minimizing communication costs, and staying within the GPU memory size limit. Though we consider each of these factors throughout this dissertation, we found the fixed memory size to be the most significant challenge and the main focus of our work.

GPUs are paired with their own dedicated memory located on the graphics card, which provides higher bandwidth than CPU memory; however, the GPU memory size is limited to whatever the manufacturer chooses, while CPU memory can be easily expanded. For example, NVIDIA's H100 graphics card comes with 80 GB of memory, while CPU memory can be expanded to terabytes if needed. In some instances, the dataset itself may be too large to fit in GPU memory. In our prior work implementing Twitter's Who to Follow recommendation system on the GPU, we achieved speedups of up to 1000x over a CPU implementation; however, the practical usefulness of our implementation was limited by the fact that only 75% of vertices and 50% of edges from the complete 2009 Twitter follow graph could fit into GPU memory [12]. While out-of-core implementations are an option, the performance cost is generally quite high, and since the main goal of using a GPU is to improve performance, this is not usually a feasible option.

In this dissertation, we show how the limited GPU memory size creates challenges for our parallel formulation of maximum clique enumeration, and experiment with a variety of methods to reduce the memory usage of our implementation while maintaining sufficient parallel work to properly utilize the GPU. We then describe our implementation of quotient filters, a type of memory-efficient data structure which can help to to reduce memory use for some applications.

The key contributions of this work include:

- A breadth-first implementation of maximum clique enumeration on the GPU, which achieves high throughput, but due to memory limitations and the combinatorial nature of the problem, cannot always solve larger and/or more dense datasets.

- A fast, accurate heuristic to solve for the maximum clique size of a graph, which can be

used to reduce the memory usage of the exact enumeration algorithm or to avoid the exact computation altogether in instances where an approximate solution is acceptable.

- GPU implementations of two types of quotient filters: standard quotient filters and rank-and-select-based quotient filters, including bulk build, delete, and merging operations.
- Three techniques for implementing a parallel scan operation for a non-associative operator with a saturation condition.

Chapter 2

Maximum Clique Enumeration on the GPU

2.1 Introduction

The maximum clique(s) of a graph is the largest group(s) of fully connected vertices. As one of Karp's 21 NP-complete problems [18], the maximum clique problem is among the most studied combinatorial problems in graph theory. While this problem has been widely studied from a theoretical point of view [5], it can also be a useful tool for many real world graph applications. Cliques have applications in social network analysis [42], identity resolution [40], network compression [32], computer vision [15], analysis of financial networks [4], and modeling metabolomic networks [19]. Approximate measures are often used in practice, because it is assumed that solving the exact solution is not feasible. However, for many real-world datasets it is possible to solve for the maximum clique(s) reasonably quickly.

The most common approach to finding maximum cliques is a depth-first branch and bound algorithm, in which a new vertex is added to the clique-in-progress at each level of the search tree, and bounds on the best possible solution for each branch are computed at every branch point and compared to the current best clique found so far to determine which vertex to add next. When one branch has been fully explored, if a new largest clique has been found, the lower bound is updated, and the search backtracks to the last unexplored branch to continue the search. Backtracking algorithms like these are notoriously difficult to implement efficiently on GPUs. When Jenkins et al. implemented the closely-related maximal clique enumeration

problem on the GPU, they found that they could not achieve more than a modest speedup over a single-threaded CPU implementation due to challenges with high divergence, workload imbalance, and irregular memory access patterns [17].

Here we focus primarily on maximum clique enumeration — finding *every* clique of the maximum size in a graph. Although most previous work has focused on finding just one of the maximum cliques, we believe that solving for all maximum cliques is more broadly useful. We highlight the following contributions:

- A breadth-first search approach to *maximum clique enumeration* on the GPU.
- A variety of techniques to reduce memory via pruning, including different heuristics and traversal orderings.
- A data structure designed specifically for efficiently expanding many lists of vertices in parallel to track candidate cliques.
- A *windowed search* scheme for finding a single, maximum clique when memory constraints prevent enumeration, which allows us to explore a middle-ground between a depth-first and breadth-first search and the trade-offs between memory usage and available parallel work.
- A parallel heuristic which manages to find a clique of maximum size for 97% of the datasets in our test set before we even begin running the exact algorithm.

2.2 Background

Given a graph $G = (V, E)$, a *clique*, $C \subseteq V$, is a subset of vertices such that every vertex in C is connected to every other vertex in C via an edge, i.e., the subgraph induced by C is a *complete subgraph* of G. The maximum clique(s), C_ω, are the clique(s) with the largest cardinality. The size of the maximum clique is also known as the *clique number* of a graph, denoted as $\omega(G)$. Different applications may have use for the clique number on its own, the clique number and multiplicity, the list of the vertices belonging to one of the maximum cliques, or the members of all cliques of size ω. In some instances, an approximation of the clique number and/or members

of a large (but not necessarily maximum) clique may suffice. In this work, we seek to enumerate all maximum clique(s) of a graph, encompassing all of these possible applications.

2.2.1 The Search Tree

Since our aim is to find the *exact* maximum clique(s) of a given graph, we must use a systematic approach to consider all possible combinations of vertices in order to guarantee that we have found the largest set(s) of fully connected vertices. This problem is often solved using branch and bound algorithms, with the goal of swiftly eliminating most of these combinations via discerning choices of bounds and traversal order of the search tree. The basic branching algorithm is as follows: begin with an empty clique set, C, and a set of candidate vertices, P, which initially includes all vertices in G. Then, following some ordering scheme, select a vertex $v \in P$ to add to C, and filter out vertices in P not connected to v. Next, select another vertex remaining in P, filter again, and repeat until P is empty, then note this clique and its size. Backtrack to the previous decision point, select a different vertex from the candidate set, and continue on, maintaining a record of the largest clique found so far, until all combinations have been exhausted. In the complete branch and bound algorithm, this search tree traversal is pruned by applying bounds at each branch point to reduce the number of unfruitful branches that are explored before returning the solution.

2.2.2 Bounding the Search

Most implementations use three bounds in pruning the search space: (1) a lower bound on the maximum clique size, (2) an upper bound on the largest clique a vertex belongs to, (3) an upper bound on the largest clique within each set of vertices.

2.2.2.1 Setting an Initial Lower Bound

The size of the largest clique found so far serves as the lower bound on the maximum clique size; however, a heuristic can be used to find a lower bound before beginning the search, in order to preprune the candidate list. Due to the computational complexity of the maximum clique problem, there is a substantial body of previous work on a wide variety of heuristics, which aim to avoid paying the cost of computing an exact solution. When using a heuristic as the first step in solving for an exact solution, a better heuristic leads to better pruning; however, a high-quality

heuristic is also likely to require a lot of work, with the absolute best heuristic approaching the amount of work required for finding the exact solution. Thus, selecting a heuristic involves a trade-off between preprocessing work and work within the exact computation.

2.2.2.2 Pruning Individual Vertices

If we have an upper bound on the largest possible clique a vertex can belong to, then we can compare this against the largest clique found so far and determine whether or not the vertex could be a member of a larger clique. If not, we can ignore this vertex entirely. A simple upper bound for a vertex is its degree plus one. However, we can obtain a tighter bound using the concept of k-cores. A *k-core* of a graph is a vertex-induced subgraph in which all vertices have degree at least k [35]. The largest value of k for which a vertex is a member of a k-core is its *core number*. The largest clique a vertex could be a member of is its core number plus one. We compare the effectiveness of pruning using vertex degrees and core numbers.

2.2.2.3 Finding Upper Bounds for Sets of Vertices

As we traverse the search tree, we use an upper bound on the largest clique contained within the candidate set, P, to determine whether to continue to explore the branch or prune it. The most straightforward upper bound is $|C| + |P|$, the size of the current clique set plus the size of the candidate set. Alternatively, we can find a tighter upper bound using other metrics, such as vertex coloring.

2.2.3 GPU-Specific Considerations

When designing algorithms for GPUs, we must tailor our implementations to their unique architecture to achieve high performance. GPUs are optimized for high throughput, while CPUs are optimized for low latency. Because we have thousands of threads available for computation on a single GPU, we care less about work efficiency and more about maximizing available parallelism and how to best split this work up between threads. Ideally, work is distributed in a balanced way to take full advantage of the compute available. We should also avoid divergence between threads' execution paths, particularly threads within the same 32-thread grouping, known as a *warp* in the CUDA programming model. Threads in the same warp run in lockstep, so when some threads take a different execution path, the others are idling.

As described in Chapter 2.2.1, the most common method for traversing the search tree is a

depth-first approach with backtracking; however, these types of algorithms map poorly onto the massive parallelism of GPUs, due to a lack of available parallel work, high divergence, and imbalanced workloads [17]. If we choose a depth-first algorithm, we could traverse the search tree in a fine-grained thread-parallel or coarse-grained warp-parallel fashion. Both options present challenges for an efficient GPU implementation. In a fine-grained thread-parallel traversal, each thread is assigned its own subtree to search independently. Because the depth of subtrees is irregular and unpredictable, this leads to high divergence and an unbalanced workload. For a coarse-grained warp-parallel traversal, threads in each warp traverse the search tree as a group and work cooperatively to compute the new candidates and bounds at each branch point. Although this avoids the high divergence of the fine-grained traversal, it reduces the amount of parallel work available and does not provide enough work for all threads when the size of the candidate list is less than warp-sized.

Another GPU optimization to keep in mind is that in order to maximize memory bandwidth, we should use coalesced memory accesses whenever possible – that is, we want neighboring threads to access values stored in a contiguous chunk of memory. Again, due to the irregular nature of the search tree, the length of candidate lists is highly variable, making it difficult to arrange coalesced memory accesses. Finally, GPU RAM size is limited, and to avoid the additional communication costs associated with out-of-core implementations, we aim to keep overall data use small enough to fit into GPU memory.

2.2.4 Breadth-First Strategy

As the basis of our implementation we chose a breadth-first exploration of the search tree to maximize the available parallelism, minimize divergence, and improve load balancing. In a breadth-first traversal, we take all branches at each level before moving deeper into the tree. When performing the search sequentially, this is not ideal, because the maximum cliques are found at the deepest leaves of the tree; however, the massively parallel nature of GPUs allows us to explore many of these branches simultaneously instead. Though it will likely require more work overall because we are not updating the lower bound throughout the computation, we can utilize the many available threads, so we hope this allows us to finish the entire search more quickly.

Although a breadth-first approach maximizes the available parallelism, the space required to store all cliques and candidates at once is a limitation of this approach. For a depth-first search, when we reach the end of a path, if the solution found is not a new maximum, the clique and its associated data are discarded. In a parallel breadth-first search, all branches are taken at once, so we need enough memory to store all k-cliques at each level of the tree, which may be impractical, particularly for large or dense graphs. In our work, we investigate ways to overcome these memory constraints via pruning and some deviations from the typical breadth-first traversal.

2.3 Related Work

There have been some previous parallelizations of the maximum clique and maximal clique enumeration algorithms, with most targeting multi-threaded or distributed CPU systems, though there have also been a few GPU implementations.

2.3.0.1 Parallel CPU Maximal Clique Enumeration

Both Schmidt et al. [34] and Du et al. [10] implement parallelizations of the Bron-Kerbosch maximal clique enumeration algorithm for shared memory systems, where all workers have a copy of the full graph. Each worker is assigned a vertex from the graph and traverses the search tree to find all maximal cliques that contain that vertex. Cheng et al. [6] also implement a version of parallel Bron-Kerbosch, and their work is aimed at problems on distributed memory systems with large input graphs and limited memory. In their implementation, one master node computes partitions of the graph and assigns these subgraphs to the other workers, which then compute the maximal cliques in the subgraph using the Bron-Kerbosch algorithm.

In their work on a BFS-style approach, Zhang et al. include a parallel implementation [45] where a task scheduler divides all k-cliques evenly amongst the threads at each iteration, and between iterations (after all threads have finished computing new $(k+1)$-cliques), the threads synchronize, to preserve ordering of the output maximal cliques by increasing size. This implementation is very memory-intensive, because it requires all threads to have a copy of the graph, stored as an adjacency matrix, and to maintain the lists of all intermediate cliques.

2.3.0.2 Parallel CPU Maximum Clique

The maximum clique implementations by Rossi et al. [32] and McCreesh et al. [23] both implement a branch and bound algorithm. They use a global work queue, and split the search tree up at the first level of branching to populate the queue (i.e. one task = searching the subtree initialized with a single-vertex clique). In addition to parallelizing the exact algorithm, Rossi also computes the initial greedy heuristic in parallel. Both Rossi and McCreesh find that they sometimes achieve superlinear speedups because threads working simultaneously are able to find new maximum cliques and prune the search tree more effectively.

Xiang et al. [44] implement a branch and bound algorithm for the maximum clique problem using the MapReduce framework. Their algorithm has two stages: (1) partition the graph to distribute subgraphs amongst processors; (2) run standard branch and bound search in each subgraph. This helps them to achieve better load balancing, but with a significant overhead cost.

2.3.0.3 GPU Maximal Clique Enumeration

The first GPU maximal clique enumeration implementation appears to be the work by Jenkins et al. [17]. They implement the basic Bron-Kerbosch algorithm, with the goal of evaluating how well-suited backtracking algorithms are for GPU architectures. They use both coarse- and fine-grained parallelism, by assigning one subtree of the traversal per warp, then having threads within the warp work cooperatively to determine the best branching strategy. The maximal cliques are written to a pre-allocated buffer that is periodically flushed to the CPU. During this flushing step, the CPU also performs basic load balancing, redistributing the work from warps with large stacks to those with short ones. Jenkins concludes that the irregular memory access patterns for this type of algorithm limit the GPU performance to a 2x speedup over a single-threaded CPU implementation.

Henry [14] also implements the Bron-Kerbosch algorithm on the GPU for use in content-based image retrieval. They process one small graph (representing a pair of images) per block, and assign one node of the search tree per thread. They find that their biggest constraint was the number of new nodes each thread can generate, though they did achieve speedups over a serial CPU implementation.

Lessley et al.'s work [20] is the only BFS-style clique implementation on the GPU we have found to date. Their implementation uses only data-parallel primitives (such as scan, map, reduce, scatter, etc.) for every step of the algorithm. The key to their algorithm is that they find cliques to combine using a dynamic hash table. During each iteration, they construct a hash table containing all k-cliques, by hashing the indices of the last $(k-1)$ vertices of the clique. They then compare cliques with matching hashes to determine whether or not they can be combined into a $(k+1)$-clique. The biggest issue for this work is the large memory requirement for storing the intermediate, non-maximal cliques, which significantly limits the size of the problems they can solve on a single GPU.

Wei et al. [43] explore maximal clique enumeration by transforming a parallel version of the recursive Bron-Kerbosch algorithm with degeneracy into an iterative version amenable to the GPU. They reduce memory through use of a CSR-like (compressed sparse row) adjacency matrix graph representation and obtain an upper bound on the maximum number of maximal cliques using Moon and Moser's theorem [25].

While our work shares commonalities with some of this previous work on maximal clique enumeration, particularly the breadth-first approach by Lessley et al., there is one key point of simplification that we can exploit when solving for maximum cliques, rather than all maximal cliques: pruning. Our goal is to complement the large amount of available parallel work generated by a breadth-first traversal of the search tree with sufficient pruning to avoid running out of memory.

2.3.0.4 GPU Maximum Clique

VanCompernolle et al. [38] implement a version of San Segundo's BBMC [33] maximum clique algorithm on the GPU. They parallelize the search tree traversal at the first level of branching and assign one subtree to each block. Threads within the block perform the bitwise parallel computations for coloring the subtree and filtering out fruitless branches. Each block traverses its subtree recursively, until the search space is exhausted. There are no load balancing mechanisms, so each block must traverse its entire subtree, with no assistance from other blocks. This implementation suffers from memory limitations, due to the stack requirements for recursion and the memory-intensive adjacency matrix representation required for the bit-wise parallel op-

erations. These memory constraints limit the applicability of their implementation to graphs with fewer than 1500 vertices. To the best of our knowledge, this is the only exact maximum clique implementation on the GPU. By contrast, we chose to perform a fine-grained parallel iterative traversal of the search tree, which allows us to avoid the stack memory requirements and load balancing issues that arise from a recursive implementation. We also use a CSR data structure for storing the graph, which is much more compact than an adjacency matrix for most real world sparse datasets. However, we do still run into challenges with memory usage arising from our choice of a breadth-first traversal.

There have also been at least two maximum clique heuristics implemented on GPUs. Cruz et al. [8] use a neural network to compute a maximum clique heuristic, which they deem to be poorly-suited to the GPU architecture. Nogueira et al. [28] devise a new local search heuristic for the maximum weight clique problem, which utilizes two new vertex neighborhood concepts and a tabu list. In their hybrid CPU-GPU implementation, they assign each GPU thread the work of computing the potential move cost for one vertex in each iteration, and use a parallel reduction to determine the optimal move from all candidates. Their GPU implementation delivers a 12x speedup over the sequential version of the same heuristic. In our work, we aim to use a more lightweight heuristic to find a lower bound and reduce the work remaining for the exact computation, while Nogueira et al. instead seek to find a high-quality approximation of the maximum weight clique size for applications that do not require an exact solution.

2.4 Implementation

We find the maximum cliques by performing a breadth-first traversal of the search tree via an iterative process. In each iteration, we launch one thread per candidate vertex across all of the candidate lists in the current level. Each thread adds its vertex to its the clique set and generates the list of candidates for the next level of the search. We wait until all threads have finished, then repeat the process for the next level of the search tree.

The steps of our implementation are as follows: (1) (optionally) compute the vertex k-core decomposition of the graph, (2) find an initial lower bound maximum clique via a greedy heuristic, (3) form the initial lists of 2-cliques/candidates, (4) perform the iterative process described

above, adding vertices to the clique lists and generating new candidate lists for the next iteration. Each of these steps is performed in parallel on the GPU. In this section, we describe the details of each of these operations, as well as a modified version, in which we explore only a subset of the candidates at a time, which we refer to as a windowed breadth-first search.

2.4.1 External Libraries

Our k-core implementation relies on the Gunrock GPU graph library [41]. The Gunrock library enables users to write GPU implementations of graph algorithms at a higher level of abstraction. In Gunrock, computations are implemented through operations on *frontiers* of vertices or edges. In addition to compute operations, Gunrock uses the traversal operations *advance*, which generates a new frontier from the neighbors of the current frontier, and *filter*, which creates a new frontier that is a subset of the current frontier. The Gunrock library generates the CUDA kernels needed for each operation and performs the necessary load balancing for us. We also use Gunrock's graph loader in preprocessing to convert the input dataset into a compressed sparse row format (CSR) graph data structure, which we store in GPU global memory to utilize throughout the rest of the computation. We also make use of NVIDIA's CUB library for its optimized scan, reduce, select, and sort operations [24].

2.4.2 Vertex k-Core Decomposition

A vertex k-core decomposition is the computation of the core numbers for all vertices in the graph. As described in Chapter 2.2.2.2, a vertex's core number is a measure of its membership in subgraphs of highly-connected vertices. We experiment with using vertices' core numbers in our initial heuristic, for vertex pruning bounds, and in choosing the order of traversal for our windowed search. We expect that using core numbers will improve the effectiveness of pruning and the accuracy of the lower bound from the heuristic, though at the cost of additional preprocessing time. We offer analysis of this trade-off between precompute time and improvements in pruning effectiveness in Chapter 2.5.2.4.

As shown in Algorithm 1, our implementation begins with all vertices in the graph in the frontier and $k = 0$. We check the degrees of the vertices, and for any vertices with degree less than or equal to k, set their core number to k, and mark them as deleted. We then we

advance to the deleted vertices' neighbors, and decrement their degrees. We then repeat the filter operation, checking if any of these vertices' degrees have been reduced below k, and mark them to be deleted. We repeat this advance and filter cycle until there are no vertices remaining in the frontier. We then increment k and repeat this process until all vertices' core numbers have been found. This code is available as one of the app examples in the Gunrock library: https://github.com/gunrock/gunrock/tree/dev/gunrock/app/kcore.

2.4.3 Heuristic

The next step is to establish a lower bound on the maximum clique size via a heuristic. As described in Chapter 2.2.2.1, the choice of heuristic involves a trade-off of work between preprocessing and the exact algorithm. We selected a greedy heuristic rather than a more complicated heuristic because we are aiming to minimize preprocessing time, and we expect that the GPU can handle a large amount of work in the exact algorithm stage. We have two different implementations of the greedy heuristic: (1) the *single run version* in which we run the greedy algorithm once and use the GPU threads to filter the vertex list in parallel and (2) the *multi-run version* where we run many instances of the greedy algorithm in parallel on the GPU. For both versions, we provide an option to use either the vertex degrees or core numbers for determining the greedy ordering.

2.4.3.1 Single Run Heuristic

The greedy heuristic is as follows: start with a list of all vertices and pick the vertex with the highest degree (or core number) to add to the clique-in-progress, then remove any vertices not connected to this vertex. From the remaining vertices, add the vertex with the highest degree (or core number) to the clique and once again filter out any vertices not connected to this vertex. Repeat until no vertices remain in the list. The size of the clique found this way serves as a lower bound on the maximum clique size.

In our GPU implementation of this heuristic, we first create a list of all vertices in the graph and use the GPU to sort the vertices descending degree (or core number) order. We pull the first vertex, v_0, out of the candidate list and filter the vertex list on the GPU using a parallel select operation, removing any vertices which are not neighbors of v_0. Then we pick the next vertex from the filtered candidate list, and filter the list again. This process repeats until there are no

vertices remaining in the list. The number of iterations of this greedy algorithm is the lower bound clique size, $\overline{\omega}$.

2.4.3.2 Multi-Run Heuristic

For the multi-run greedy heuristic, we use the same greedy algorithm as the single-run heuristic, except we run many instances of it in parallel, each with a different starting vertex. The implementation makes use of a variety of data-parallel operations that are well-suited to the GPU. The details are shown in Algorithm 2. As in the single run version, we begin with a list of all vertices sorted by decreasing degree/core number, and also a list of the vertices' degrees/core numbers, which we use to select the next vertex to add to each of the cliques-in-progress in each iteration. We select the number instances of the heuristic we would like to run, $h \leq |V|$, and use the h vertices with the highest degree/core numbers as the seed vertices for each of the runs. We perform a few setup steps, creating segmented arrays containing all of the neighbor vertices and their degrees/core numbers for each of the seeds. Then we begin to iterate. First, we find the vertex in each segment with the highest degree/core number using a segmented maximum operation. We use one thread per segment to check whether each of the other vertices in the segment is connected to this vertex and flag vertices to keep. Next, we use a select operation to filter the vertex and degree/core number arrays, removing vertices that are not connected. Then we remove empty segments with one more select operation and update the segment indices via a scan operation. We then iterate until there are no candidate vertices remaining in any of the segments. As in the single run version, the number of iterations is the lower bound on the maximum clique size, but in this case it represents the largest clique found across all h parallel runs of the greedy heuristic. We expect that using the best of multiple runs will result in a better lower bound and, therefore, better pruning.

2.4.4 Clique List Data Structure

An important consideration for our breadth-first parallel implementation is how to store all of the cliques and candidate lists. In parallel in each iteration, we are creating a new candidate list for each of the current candidate vertices across all candidate lists. The size of each of these new candidate lists can vary widely between cliques and between iterations of the algorithm, making it impossible to preallocate the appropriate amount of memory. As mentioned in Chapter 2.2.4,

limited memory size is a significant concern for the breadth-first implementation, so we would also like to store cliques and candidate vertices as compactly as possible and avoid storing any duplicate information.

Criteria The goal is to build a minimally-sized data structure that supports the following operations: (1) add a variable number of total items in each iteration, (2) track which clique each of the newly-added candidate vertices belongs to, and (3) delete data for cliques that have been pruned. Parallel operations take place with one thread per candidate vertex, so we would like to store all candidates in a contiguous block of memory in order to achieve coalesced accesses.

Our Solution The data structure we chose, which we call a *clique list*, is essentially a linked list wherein each node of the list contains a pair of arrays, `vertexID` and `sublistID`. Figure 2.1 shows the clique list for an example graph. Each node in the clique list contains all the necessary data for one iteration of the search. `vertexID` contains the candidate vertices for that level, and `sublistID` contains the index in the previous clique list node where the last vertex added to the clique is stored. Essentially, the `sublistID` is a pointer into the previous clique list node's `vertexID` array. The `sublistID` array allows us to identify which vertices belong to the same candidate list and, at the end of the computation, to read out all of the vertices in the maximum clique(s). The first node of the clique list is different from the others. Because there is no need for indices into a previous node's vertex array, we combine the data for the first two levels of the search tree into one node by using `sublistID` to store the vertex IDs for the first level of the tree. Each node in the clique list also stores the number of candidate vertices in that level (the size of the arrays) and the clique size, k, represented by the level.

Discussion This data structure allows us to simultaneously expand all cliques in each iteration, allocating memory as needed, and to track which vertices belong to each clique. Iterating through the linked list to read out the clique vertex sets is cumbersome, but within each iteration of the search, we only need to know the current candidate vertices in order to check their connections and generate the candidate list for the next iteration; therefore, we only need to access values in previous nodes of the linked list at the end of the computation to read out the members of the maximum clique(s). We avoid storing duplicate information because each of

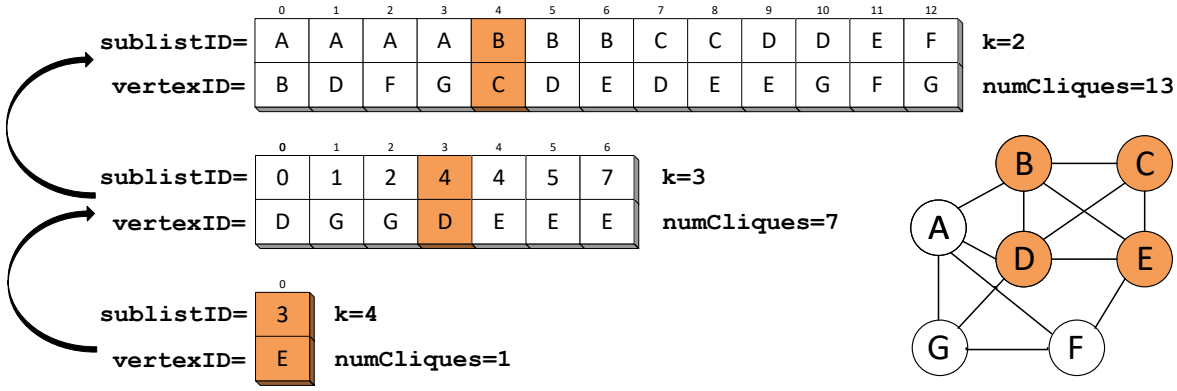

Figure 2.1. An example graph and the clique list structure used to find its maximum clique. To illustrate how the clique list structure works, let's walk through how to read out the maximum clique, represented by the circled blocks. We start at the head of the list, which is the most-recently added node, for $k = 4$. There is only one clique of size 4, so this graph only has one maximum clique. $\texttt{vertexID}_4[0] = E$ means vertex E is in the clique, and $\texttt{sublistID}_4[0] = 3$, so we follow the `previous` pointer to the clique list node for $k = 3$ and find $\texttt{vertexID}_3[3] = D$. Now we have the clique set $C = \{E, D\}$, and we use the $\texttt{sublistID}_3[3] = 4$ as a pointer into the clique list node for $k = 2$, where both $\texttt{vertexID}_2[4] = C$ and $\texttt{sublistID}_2[4] = B$ represent vertices in the clique. Therefore, the maximum clique of this graph is $C = \{E, D, B, C\}$.

the k-cliques generated from the same $(k-1)$-clique point back to their shared ancestor in the previous clique list node. This structure also allows for coalesced accesses, because neighboring threads read from consecutive values in the `vertexID` and `sublistID` arrays. One drawback of this data structure is that it is very difficult to delete data for cliques that have been ruled out as candidates for the maximum clique, because `sublistID` values would need to be updated in all nodes of the clique list structure. We could not find another data structure that met our other criteria and allowed for simpler deletions, so we accept this downside and do not delete any data for eliminated cliques.

2.4.5 Setup: Forming the 2-Clique List

Before running the breadth-first search, we must set up the first node of the clique list containing all of the 2-cliques. This is essentially a list of the edges in the graph stored as an array of source vertices and an array of destination vertices. Algorithm 3 shows the details of how we create the 2-clique list on the GPU using data-parallel operations. The main steps are: (1) one thread per vertex determines the number of neighbors in its sublist; (2) prune sublists shorter than lower

bound clique size determined in heuristic; (3) a scan operation to determine start indices for sublists and amount of memory to allocate for 2-clique list; and (4) one thread per unpruned sublist outputs the vertices for that sublist.

In order to avoid storing duplicate cliques, we use only one of the two directed edges that represent each undirected edge in the original graph, known as an *orientation* of the graph. This orientation step allows us to avoid wasting work and memory space throughout the rest of the computation. We do not explicitly modify the graph, but instead select the desired edges when forming this initial clique list node. From each reciprocal edge pair, we keep the edge where the source vertex has lower degree (line 5). Orienting the graph by degree improves pruning over orientation by index, because vertices with lower degree have shorter adjacency lists. Selecting these vertices as the source vertices should result in initial sublists that are shorter on average and thus a greater proportion of sublists will be smaller than $\bar{\omega}$. We examine the effects of this decision on runtime and memory usage in Chapter 2.5.3.1.

In addition to pruning entire sublists, we also pre-prune individual vertices by comparing their degree/core number to the lower bound found in the heuristic. After perfoming all pruning, it is possible that there may be only one sublist remaining, representing the clique discovered by the heuristic. In this case, we skip the full exact algorithm because we have already found the singular maximum clique (line 35). At this point, if we were only interested in finding one of the (possibly many) maximum cliques, we could skip the exact algorithm in the event that there are no sublists longer than $\bar{\omega} - 1$.

The final preprocessing operation we perform during this stage is to sort vertices by degree within their candidate lists (line 40). Without this sort, vertices will be in the same order as they are stored in the adjacency lists, that is, sorted in order of increasing index values. By sorting candidate vertices according their degrees, we hope to improve pruning, because the vertices near the beginning of the candidate lists are assigned more edge lookups than vertices later in the list. This means lookups for missing edges are moved to earlier iterations, enabling us to prune them earlier in the search. Additionally, placing the low degree vertices at the beginning of the candidate lists means a greater fraction of edge lookups are in shorter adjacency lists, which reduces the average lookup speed. In Chapter 2.5.3.2, we analyze the effect of this operation on

memory usage and overall runtime.

2.4.6 Breadth-First Maximum Clique

Once we have the first node of the clique list, we can begin the iterative breadth-first search, detailed in Algorithm 4. This process is as follows: one thread for each vertex in the clique list checks whether it is connected to each of the vertices that follow it in its sublist, and tallies the number of successful edge lookups. Each successful lookup represents a $(k+1)$-clique. The length of this new sublist is compared to $\bar{\omega}$ to determine whether it should be pruned before returning the count. Next, we use a scan operation to find the start indices for the new sublists and amount of memory to allocate for the $(k+1)$-clique list. If there are no new cliques, we have reached the end of the search, and we use the current clique list node to read out the vertices in the maximum clique(s). Otherwise, we assign one thread per vertex to output the candidate vertices for its new sublist.

This breadth-first approach provides the ability to easily launch a different number of threads in each iteration to match the number of candidate vertices formed in the previous iteration. By changing the number of threads in each iteration to match the number of cliques, we avoid the load imbalance of having each thread traverse multiple levels of the search tree, which would result in both many dead-end threads and threads with vast search trees to explore. The largest portion of the computation in each iteration is the edge checks, each of which consists of a binary search (lines 5 and 19) on the candidate vertex's adjacency list within the CSR. Unfortunately, these memory accesses will not be coalesced, because neighboring threads are responsible for different candidate vertices. However, individual threads may receive some benefit from caching, because all of their reads will be in the same part of the graph data structure.

2.4.7 Windowed Search

As mentioned in Chapter 2.2.4, one of the challenges for a breath-first implementation of maximum clique is the large memory requirement for storing all candidates simultaneously. Particularly for graph datasets that are large and/or dense, there may be more candidate cliques than can fit in GPU memory, even after pruning. For these instances, we consider an approach for solving for only one of the maximum cliques, rather than enumerating all maximum cliques.

We implement a windowed variation on the breadth-first search, wherein we split up the initial list of 2-cliques and run our breadth-first maximum clique algorithm on one subset (window) of candidates at a time. Although a fully depth-first search provides little parallelism and creates too much divergence and workload imbalance between threads to perform well on a GPU, we hope that modifying the search to be less broad can offer a balance between parallelism and memory requirements.

Implementation We want to ensure that the window boundary is between sublists, since candidate vertices use the information for all vertices that follow them in their sublist. When selecting the window tail, we use the GPU threads to quickly read a chunk of `sublistID` values and check if their index is the end of a sublist, and if so, write their index to a global variable using an atomic minimum operation. This gives us the end of a sublist closest to the nominal end of the window. When we have completed the breadth-first search of one window, we update the lower bound if a new largest clique has been found, find the tail for the new window, and repeat until we have finished all windows. With windowing, we have the ability to choose an ordering for the search, as other depth-first implementations do. We experiment with sorting the source vertices in the 2-clique list by their degrees or core numbers and describe our findings in Chapter 2.5.4.

Discussion It is still possible that the combinations from a relatively small set of 2-cliques can lead to a very large list of candidates for larger cliques. The choice of window size is important, because we want to provide enough work to keep the GPU busy, but keep the clique list small enough to stay within memory bounds. We expect graphs with higher average degree to work best with a smaller window of the 2-clique list, because the number of candidates will probably increase more quickly with each iteration. We hope that exploring high-degree neighborhoods first (by sorting the source vertices by degree/core number) will increase the probability of finding the maximum clique earlier in the search and improve pruning for the remainder of the search. We test a variety of window sizes and traversal orderings and describe the trade-offs we find in Chapter 2.5.4.

2.5 Results

Methodology We evaluate our maximum clique implementation on the 58 largest real-world datasets (all datasets with $|E| > 10$k)[1] evaluated in Rossi et al.'s paper [32], downloaded from the Network Repository [31]. These include social, web, road, biological, technological, and collaboration networks ranging in size from 10k to 106M edges. We use Rossi et al.'s Parallel Maximum Clique (PMC) [32] as our main comparison; however, we note that their implementation only finds one of the maximum cliques — they do not find *all* cliques of that size. We randomize the vertex indices, to avoid any bias from the ordering of the original datasets that could affect the comparisons for sorting by index and degree. We also preprocess the datasets (before forming the CSR data structure) to ensure all graphs are undirected and contain no loops. We run all GPU and CPU experiments on a Linux workstation with a 2.8GHz 24-Core AMD EPYC 7402 CPU and 512GB of main memory and an NVIDIA Tesla A100 GPU with 40 GB of on-board memory. Our code is compiled with CUDA 11.6. For both the overall throughput results and comparison with PMC in Chapter 2.5.1, we report the results from the fastest configuration (for our implementation: the best combination of heuristic, window size, and other preprocessing; for PMC: the best number of threads) for each dataset. Reported runtimes for our implementation and PMC represent the average of 5 runs, and do not include the time to load the graph dataset onto the GPU, but do include the heuristic runtime and other preprocessing.

Key Takeaways:

- Our implementation performs best for larger graphs with low average degree.
- We achieve significant speedups over PMC on low-degree graphs, while PMC tends to be faster for high-degree graphs.
- For some graphs with high average degree and other hard to prune graphs, our implementation runs out of memory for storing candidate cliques.

[1] Our implementation is OOM for two datasets (friendster and flickr), so they do not appear in performance data, but are included in Tables 2.1 and 2.2.

- Breaking the search up into smaller windows does enable us to solve more hard to prune graphs, but at a significant performance cost.
- Better pruning does not dependably improve runtimes, so thus we prioritize minimizing preprocessing time and pruning just well enough to avoid running out of memory.
- The overall best heuristic is the multi-run degree-based heuristic. Smaller graphs perform best with a simple heuristic, while the hardest-to-prune graphs perform best with the multi-run core number heuristic.
- We save time and memory by orienting the graph by degree and sorting candidate vertices by degree.

2.5.1 Overall Performance

Performance vs Average Degree **The most consistent factor determining the performance of our implementation is the average degree of the graph.** As shown in Figure 2.2, the number of edges processed per second decreases as the average vertex degree increases. There are a few factors at play here. First, graphs with higher degree are harder to prune because many of the vertices' degrees (or core numbers) will be larger than the heuristic lower bound. Therefore, candidates stick around for more iterations of the exact algorithm, requiring more work. Second, vertices in high degree graphs have larger adjacency lists, which corresponds to longer sublists in our algorithm. The workload for a thread in each iteration is dependent on the length of its sublist and its position within the sublist. With longer sublists, each iteration of the main loop has a longer runtime, and there is greater divergence and poorer load balancing amongst threads. Third, because each edge lookup requires a binary search, larger adjacency lists increase the work for each of these operations.

Performance vs Graph Size **We achieve higher throughput as the graph size increases; however, the challenge is to avoid running out of memory (OOM) while solving these larger graphs, while also maintaining enough work to keep the GPU busy throughout the entire computation.** Figure 2.3 demonstrates that the runtime per edge decreases as the number of edges increases. This indicates that the GPU is able to handle these additional edges,

and we may be able benefit from still greater efficiencies when scaling up to larger (but still low degree) graphs. However, when solving these larger graphs, particularly if they do not also have a very low average degree, there may be too many intermediate candidate cliques to fit in GPU memory. The solution to this problem is to improve pruning, but over-pruning leaves the GPU under-utilized. The clique count distribution for web-wikipedia2009 shown in Figure 2.4 demonstrates how this balance plays out on a single dataset with different levels of pruning. Without pruning, or with poor pruning, the number of mid-sized candidate cliques explodes and the cliques do not fit in GPU memory. Additionally, earlier and later iterations typically have many fewer candidates than the peak at the mid-range values of k, and there is not enough available work to saturate the GPU. If pruning is very effective, we may succeed in pruning almost all candidates not in the maximum clique, leaving little work in any iteration of the main loop, as seen with the multi-run heuristics for web-wikipedia2009. This is a challenge inherent to this application, because pruning these mid-range values is not possible for all graphs, and we are limited in the peak of this distribution by the GPU memory. So the key to optimal GPU performance is keeping the peak low enough to stay in GPU memory, while still leaving enough work in the early and late iterations to fill the GPU.

Comparison with Previous Work **We find that our implementation outperforms PMC for low degree graphs, while PMC is faster for high degree graphs**, as shown in Figure 2.5. In general, the performance of their implementation is more dependent on the number of edges in the graph than the average vertex degree, while ours has the opposite trend. Therefore, **our implementation is more performance-scalable, except for the memory requirements of our breadth-first implementation. Adding the windowing option to our implementation does help to mitigate the memory requirements of the breadth-first implementation, but it comes at a performance cost.** As you can see at the bottom of Figure 2.5, for the handful of graphs where only the windowed version is successful, PMC is significantly faster. Dividing the problem up into small enough windows to keep the memory requirements manageable tends to reduce the amount of parallel work so much that we cannot take advantage of the parallel power of the GPU.

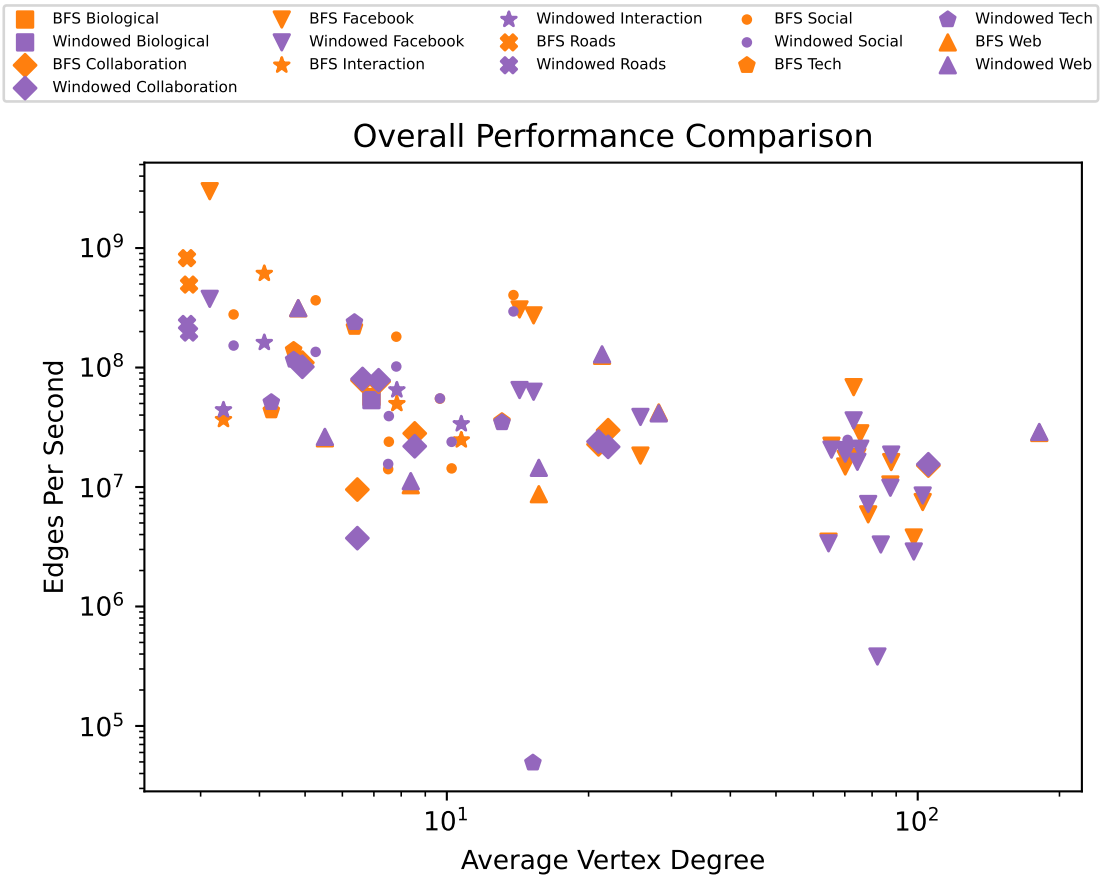

Figure 2.2. Throughput (including all preprocessing) versus average degree for fastest configuration on each dataset for the basic breadth-first version and version with windowing. For both the regular breadth-first and windowed versions, performance is inversely correlated with average vertex degree.

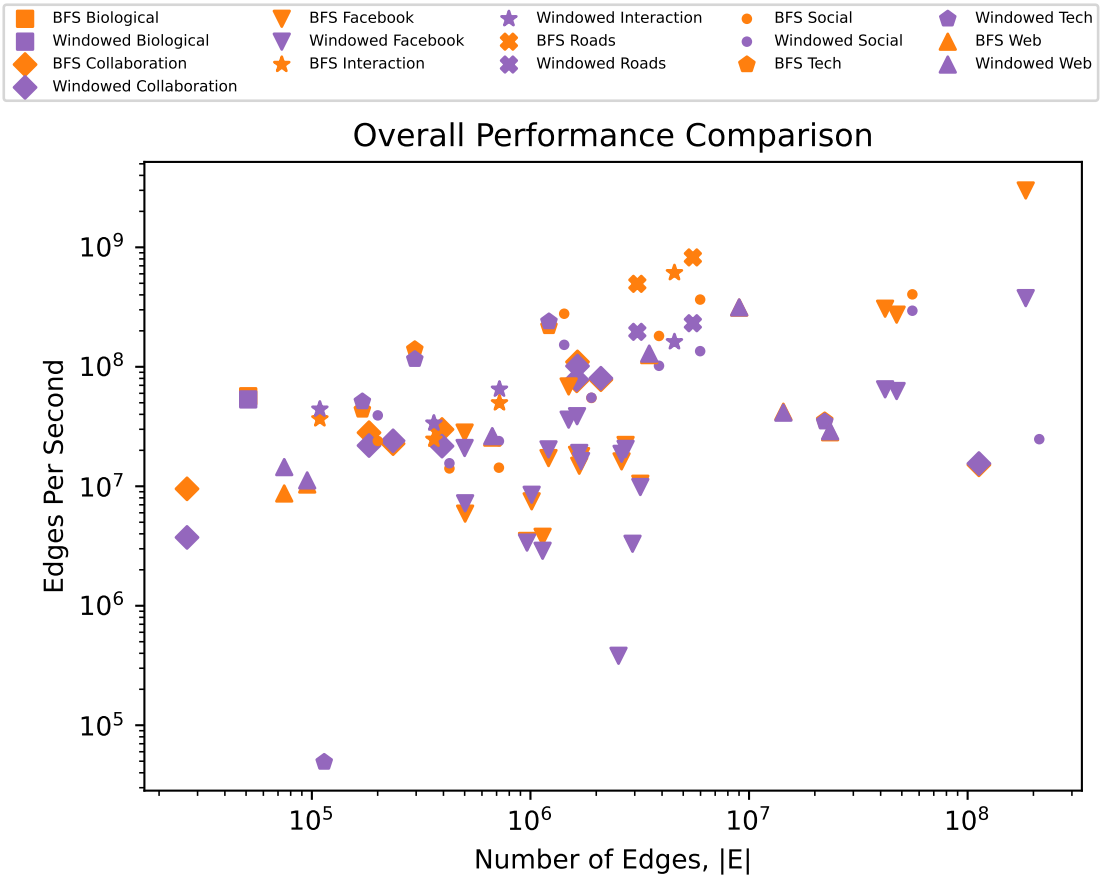

Figure 2.3. Throughput (including all preprocessing) versus number of edges for fastest configuration on each dataset for the basic breadth-first version and version with windowing. For both the regular breadth-first and windowed versions, throughput is higher for larger graphs.

2.5.2 Heuristics

For small graphs and/or graphs with low average degree, it is sometimes possible to run the exact maximum clique computation without computing an initial lower bound. However, we find that without a heuristic lower bound allowing us to prune the search, we typically run out of memory when attempting to solve larger and/or higher degree datasets. As described in Chapter 2.4.3, we implemented four different versions of greedy heuristics. In these experiments, the multi-run heuristics use all vertices in the graph as seeds, i.e., $h = |V|$. We analyze the comparative effectiveness of the heuristics by comparing them along three metrics. (1) *Accuracy*: how close is the estimated lower bound to the true maximum clique size? (2) *Pruning*: how much memory do we save when pruning using these lower bounds, and is this pruning sufficient to

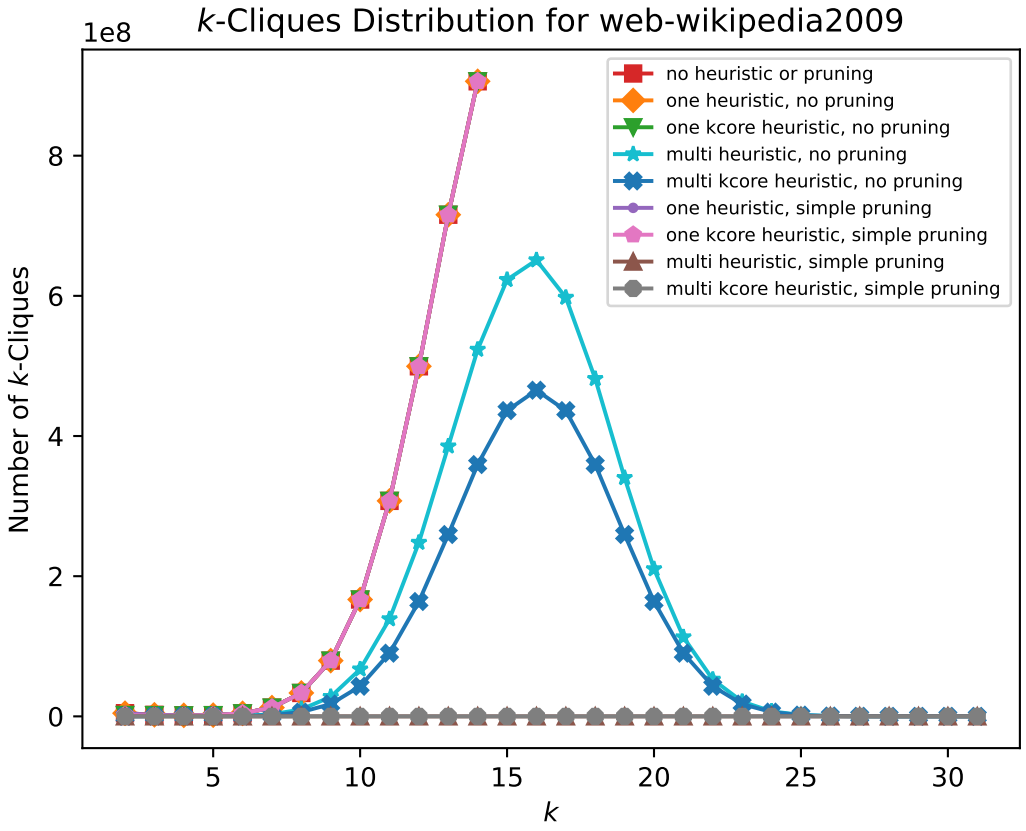

Figure 2.4. k-clique counts in each iteration of the regular breadth-first maximum clique computation for web-wikipedia2009, using each of the heuristics, with and without intermediate pruning of the candidate lists. With insufficient pruning, our implementation runs out of memory, while the highest-quality pruning leaves little work remaining in each iteration of the exact algorithm.

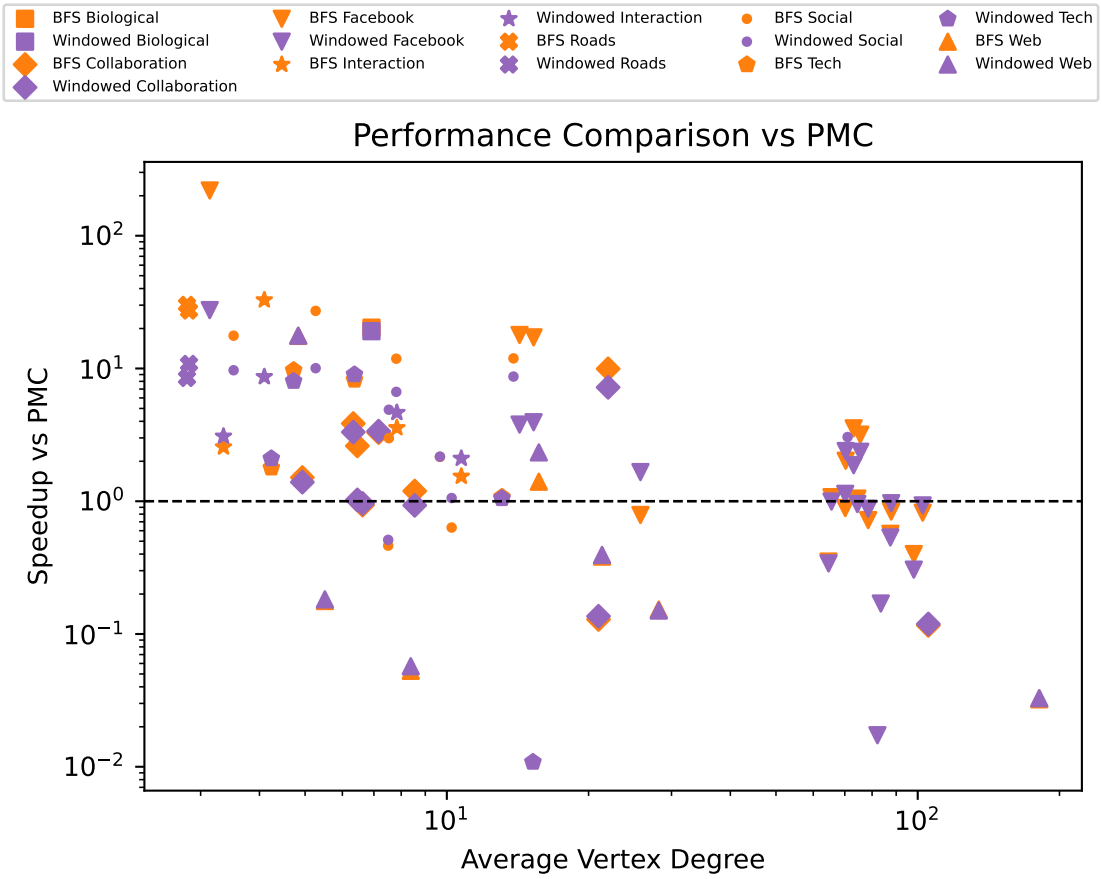

Figure 2.5. Speedup over Rossi PMC for the fastest configurations of our regular breadth-first and windowed implementations. Our implementation performs relatively better on datasets with lower average degree.

avoid running out of memory? (3) *Speedup*: does this pruning result in an overall speedup in solving for the maximum clique, or is the additional preprocessing time greater than the time saved in the exact computation?

2.5.2.1 Accuracy

Of our four heuristic options (single-run degree, single-run core number, multi-run degree, and multi-run core number) the multi-run versions provide much better lower bounds than the single-run options. Figure 2.6 shows the accuracy of the lower bounds found by each of our heuristics and their runtimes and how they vary across datasets. Table 2.1 summarizes the mean error in the heuristic clique size across all datasets for each of our heuristics and the

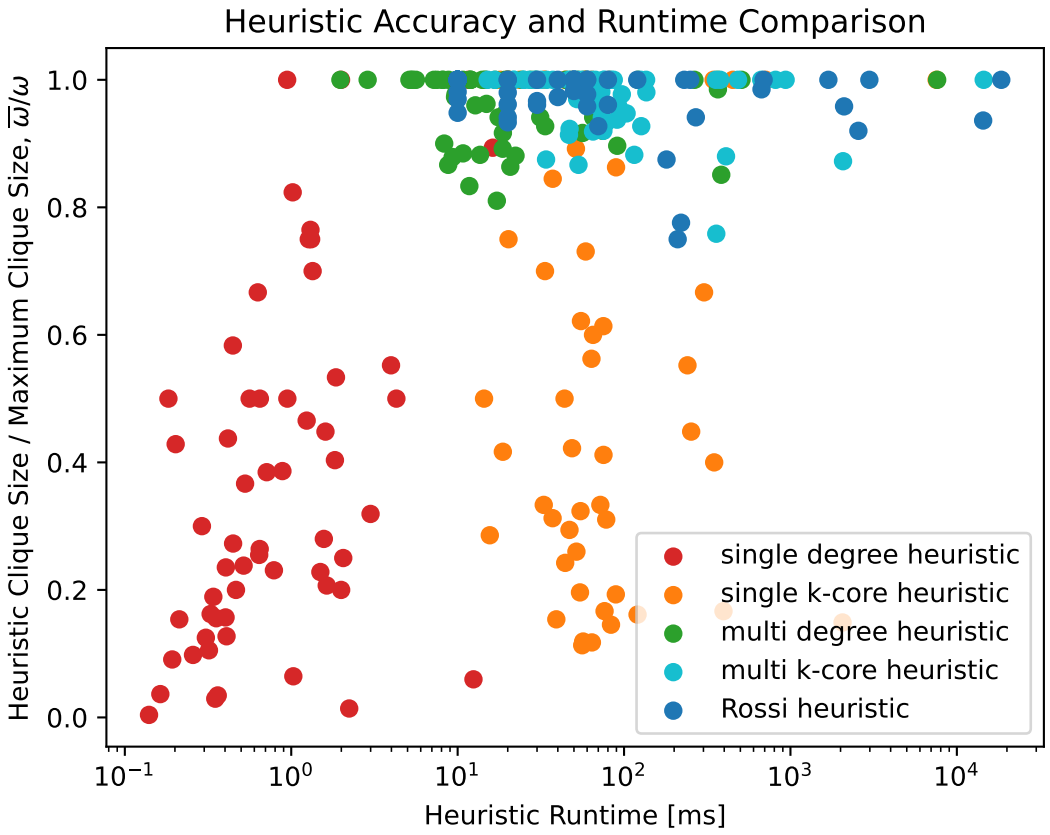

Figure 2.6. Comparison of accuracy and runtime of each of our heuristics and the heuristic used in PMC. The multi-run heuristics have much better accuracy than the single-run heuristics. The ideal heuristic would be at the top left of this diagram, but we find that a higher quality heuristic typically requires a longer runtime.

heuristic used in Rossi et al.'s PMC. For the single-run versions, using core numbers does usually improve the the lower bound significantly, but it comes at the cost of a much longer runtime. The multi-run degree and multi-run core number heuristics result in similar accuracy, with the degree version finding a larger clique for some datasets and the core number version finding a larger clique for others, and for many datasets both versions succeed in finding a clique of the maximum size. The multi-run degree and single-run core number heuristics have similar runtimes, but the multi-run degree heuristic has higher accuracy. Overall, the lower bounds from our multi-run heuristics are comparable to those of the heuristic used in PMC, which uses a similar algorithm to that of our multi-run core number heuristic.

Table 2.1. Heuristics error comparison. Error is the difference between the size of the clique found by the heuristic and the true maximum clique size.

Heuristic	Mean Error
Single-run degree	63.3%
Single-run core number	40.6%
Multi-run degree	3.9%
Multi-run core number	3.0%
Best multi-run	2.1%
Rossi PMC	2.5%

2.5.2.2 Pruning Quality

We find that the multi-run heuristics provide the best pruning and allow us to solve more datasets without running out of memory; however, graphs where the average degree is close to or larger than the maximum clique size are difficult to prune, even with an accurate lower bound. Figure 2.7 shows that the quality of the lower bound is the main determining factor in achieving high levels of pruning. The multi-run heuristics achieve both the highest accuracy and the largest fraction of candidate cliques pruned. However, there are some datasets where even an accurate lower bound does not allow us to prune the candidates very aggressively. Figure 2.8 shows that when the lower bound is not significantly larger than the average degree, pruning tends to be less effective. This makes sense, because all of the upper bounds used in pruning are related to degree. Candidate lists are pruned based on their length, and the lengths of the initial candidate lists are determined by the lengths of the vertices' adjacency lists. Vertices are pre-pruned based on their degree (or core number, which is typically correlated with degree). In instances where the heuristic finds one of the maximum cliques, there is no way to increase pruning by improving the heuristic.

In general, we do see an increase in pruning when using the more complex heuristics with higher accuracies and longer runtimes, as shown in Figure 2.9. The increase in runtime for the versions using core numbers over the the equivalent (single- or multi-run) degree versions is entirely due to the compute time for the vertex k-core decomposition. Most importantly, we do

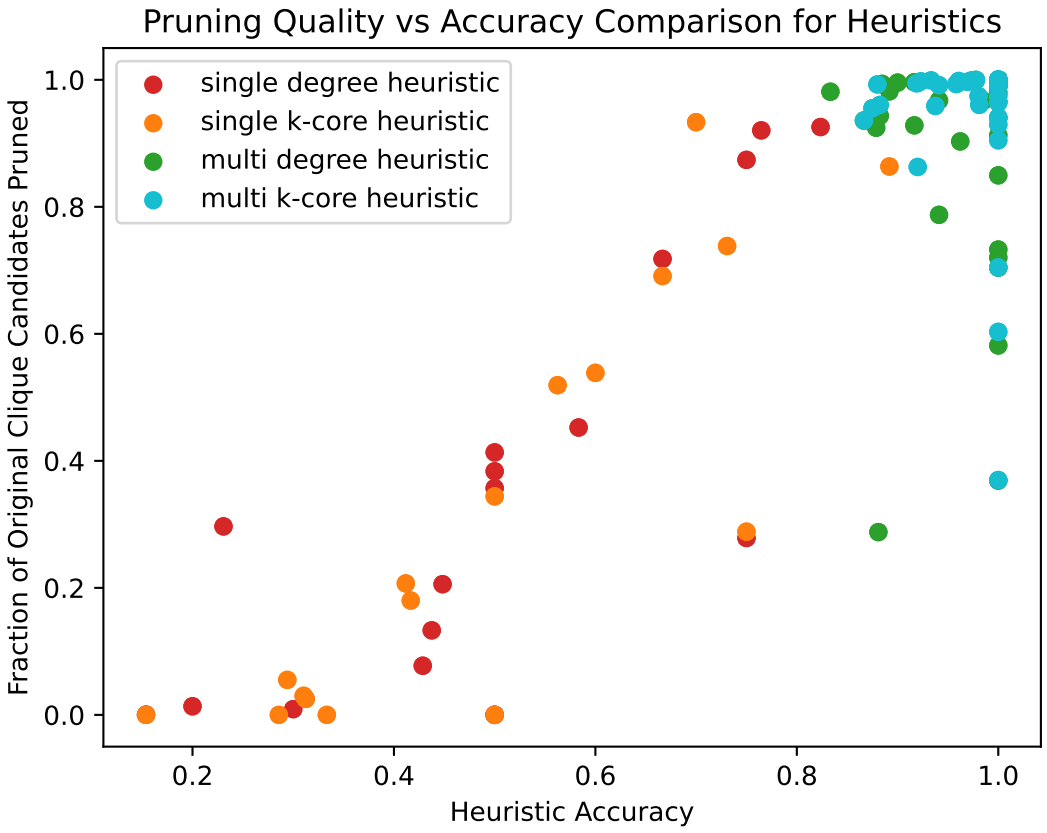

Figure 2.7. The fraction of original candidate cliques pruned by each of our heuristics is typically correlated with the accuracy of the heuristic. The number of cliques used as the baseline is either the total number of candidates found with no pruning, or, when there are too many cliques to store in GPU memory without pruning, we set the baseline at the maximum number of cliques that fit in GPU memory.

find that the improvement in pruning from more complex heuristics enables us to solve more datasets without running out of memory, as shown in Table 2.2, though these more complex heuristics do typically have longer runtimes, as can be seen in Figure 2.17. If increased pruning also results in a faster runtime for the exact algorithm, this presents a trade-off between pre-processing and main algorithm runtime. However, because our breadth-first algorithm runs in parallel on the GPU for a fixed number of iterations, pruning the candidate lists past a certain point will not significantly improve the runtime of the exact algorithm, because we are not using the full capacity of the GPU.

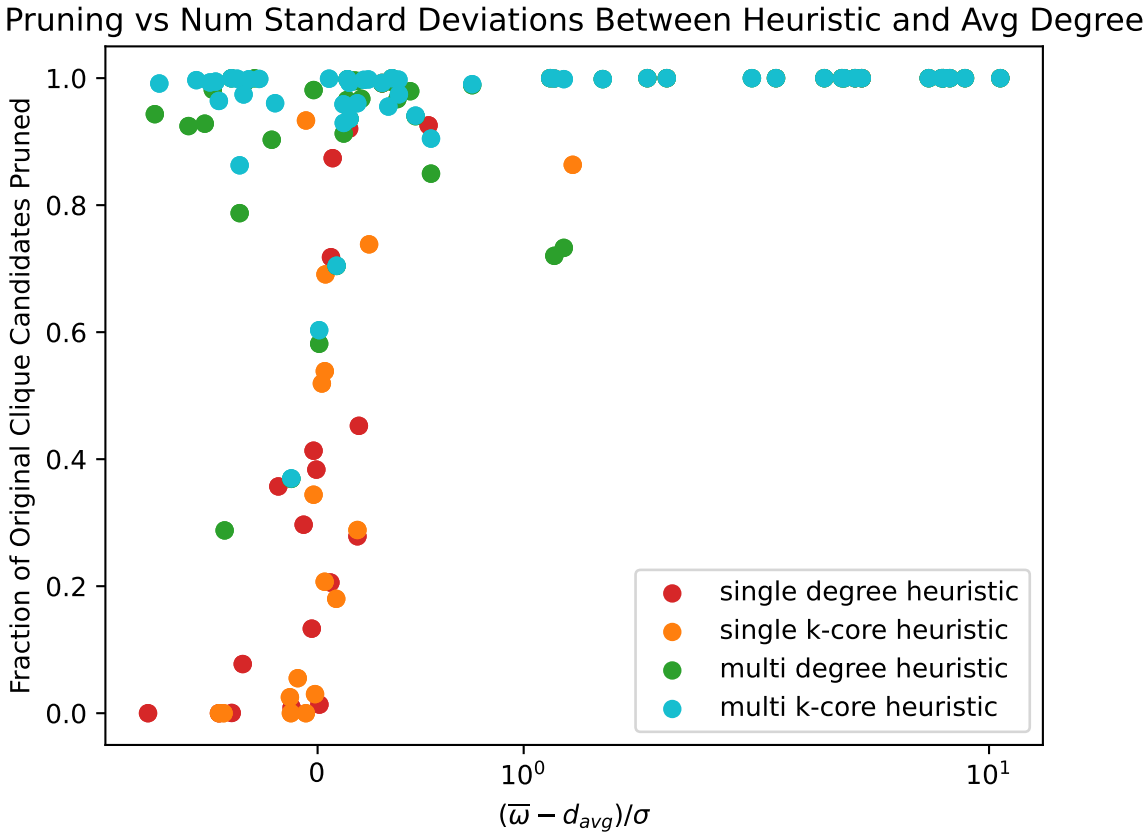

Figure 2.8. Fraction of original candidate cliques pruned versus number of standard deviations difference between heuristic lower bound and average vertex degree. Pruning is not very effective when the lower bound is not significantly higher than the average degree.

Table 2.2. Number of graphs solvable (out of 58 total) using full breadth-first maximum clique with lower bounds from different heuristics.

Heuristic	Solved Graphs	OOM
None	19	67.2%
Single-run degree	21	63.8%
Single-run core number	35	39.7%
Multi-run degree	50	16.0%
Multi-run core number	52	10.3%

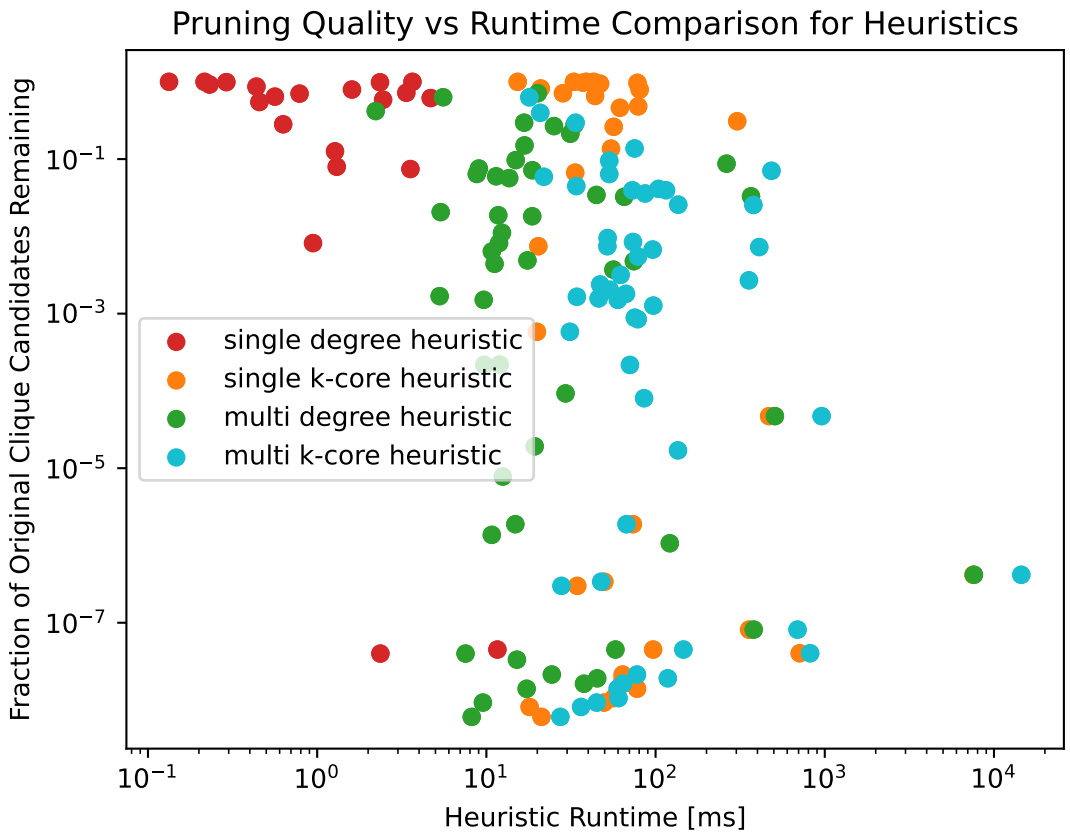

Figure 2.9. Fraction of original candidate cliques remaining after pruning vs heuristic runtime for each of our heuristics. Lower is better. The heuristics that provide the best pruning typically have longer runtimes.

2.5.2.3 Speedup

The central purpose of the heuristic in our implementation is to avoid running out of memory for storing candidate cliques when running the exact algorithm; however, pruning candidates also reduces the work required, which may result in an improvement in runtime, even when not required to stay within GPU memory. Table 2.3 shows speedups for the breadth-first version when switching from less complex heuristics to the more complex ones (order of simplest to most complex: none, single-run degree, single-run core number, multi-run degree, multi-run core number). Because for many datasets, our implementation runs out of memory when we run it with no heuristic or one of the less accurate heuristics, we must use four different baselines in our comparison. The values listed in each row are the geometric mean speedups across

Table 2.3. Geometric mean overall speedups comparison for each of our heuristics. Speedup numbers represent the overall performance improvement from using the heuristic listed in the column value over the baseline listed on the left of each row.

Baseline	Single Deg	Single Core	Multi Deg	Multi Core
None	$1.0x$	$0.4x$	$1.1x$	$0.4x$
Single Degree	—	$0.2x$	$0.3x$	$0.1x$
Single Core	—	—	$2.9x$	$1.1x$
Multi Degree	—	—	—	$0.9x$

all the datasets that require that baseline heuristic in order to complete the maximum clique computation without running out of memory. I.e., the datasets represented in the "None" row of Table 2.3 correspond to those counted in the "None" row of Table 2.2. Figures 2.10, 2.11, 2.12, and 2.13 show these speedups for each of the individual datasets. Here again, these are separated into four figures due to the need for different baselines for different datasets. In Figures 2.10 and 2.12, datasets are listed in order of decreasing maximum clique size from top to bottom. In Figures 2.11 and 2.13, they are listed in order of decreasing average degree.

No Heuristic Baseline **For graphs that can be solved without using a heuristic, it is typically fastest to to skip the heuristic altogether, unless the maximum clique is large.** Figure 2.10 shows speedups for datasets that can be solved with no heuristic. When the maximum clique size is small, usually the best option is no heuristic or a single run of the degree-based heuristic. For graphs with larger maximum cliques, the increase in pruning with the multi-run heuristics does usually provide an overall speedup. This is because the work saved in each iteration adds up over many iterations of the exact algorithm.

Single Run Degree Heuristic Baseline **Although the single-run degree-based heuristic has the shortest runtime, it does not generally provide a good enough lower bound to meet memory constraints, which is the main purpose of the heuristic in our implementation.** Figure 2.11 shows the two datasets in our test set, ca-HepPh and ca-GrQc, that run out of memory with no heuristic, but have sufficient pruning with the lower bound from a single run of the degree-based greedy heuristic. For these datasets, all of our heuristics succeed at finding the maximum clique, and are able to pre-prune all other candidates and skip the exact computation.

Single Run Core Number Heuristic Baseline **For these graphs, using the multi-run degree heuristic achieves the best performance.** In Figure 2.12, with the baseline configuration using a single run of the core number-based heuristic, we see the opposite trend from Figure 2.10 – the multi-run heuristics provide a large speedup when the maximum clique is smaller, and when the maximum clique is larger, the single-run core number and multi-run degree heuristics achieve similar performance. There are two reasons for this. First, these particular datasets have large maximum cliques that are considerably larger than the next largest clique, so as long as the heuristic succeeds in finding (one of) the maximum clique(s), the search is easy to prune. All three heuristics succeed at finding a clique of size ω for these graphs and achieve a high level of pruning. Secondly, each iteration of the multi-run heuristic requires more work than that of the single-run heuristic, because we are effectively running $|V|$ parallel instances of the same algorithm. The number of iterations of the heuristic is equal to the size of the clique found by the heuristic, so for datasets like ca-hollywood-2009, which has $\omega = 2209$, the multi-run degree-based heuristic has a similar runtime to that of the single-run core number heuristic, which spends most of its runtime in the k-core computation. Figure 2.16 shows that the gap between the runtime of the multi-run degree and single-run core number heuristics tends to shrink as the maximum clique size increases. Overall, the multi-run degree heuristic is faster for 11 out of 14 datasets, and so is generally superior to a single run of the core number heuristic.

Multi-Run Degree Heuristic Baseline **For graphs that require lower bounds from the multi-run heuristics to provide enough pruning to avoid running out of memory, about half run faster with the degree heuristic and half run faster with the core number heuristic.** As shown in Figure 2.13, these are almost all Facebook datasets, which tend to have average degree higher than their maximum clique size, and are therefore hard to prune. This makes the high accuracy of the multi-run heuristics essential, and the additional accuracy and tighter vertex pruning upper bounds from the core numbers more likely to be beneficial. In particular, we found that the core-number-based heuristic tended to be faster for graphs with higher average degree.

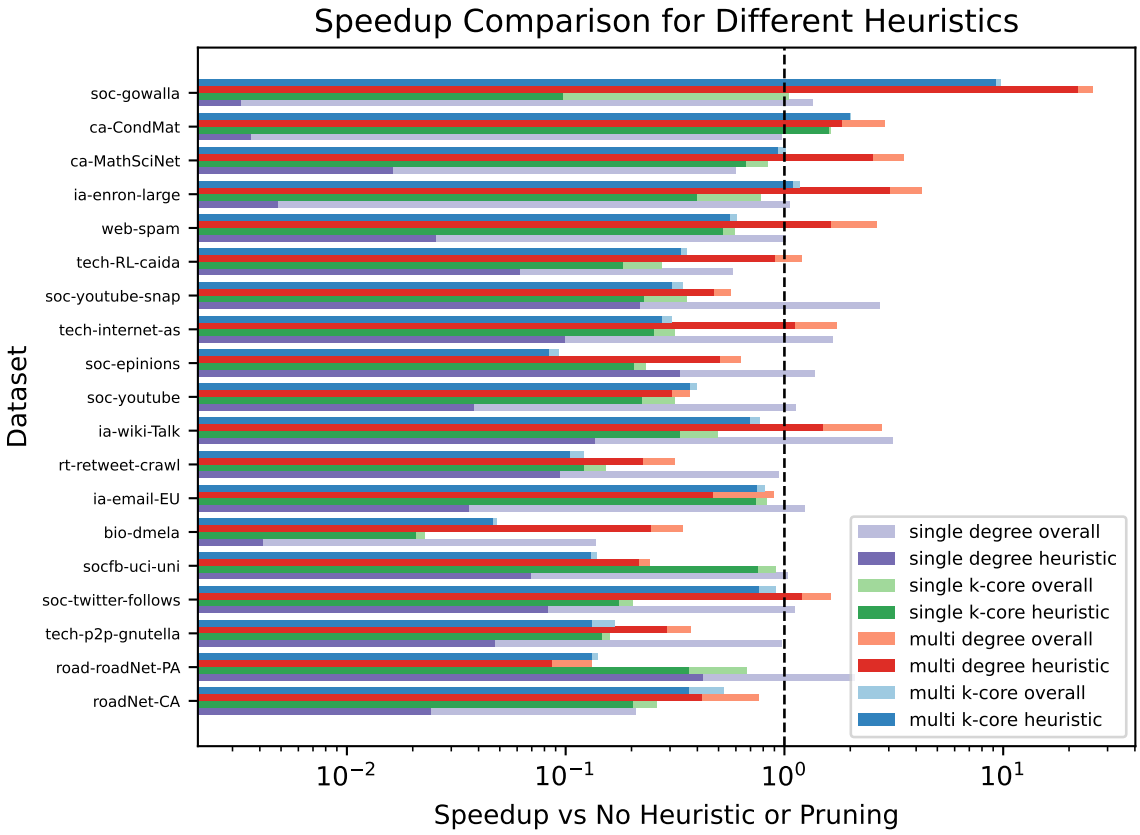

Figure 2.10. Overall speedups for each of our heuristics over baseline using no heuristic. Datasets are sorted from top to bottom in order of decreasing maximum clique size. The darker "heuristic" bars represent the fraction of the total runtime spent on the heuristic computation. For graphs with a small maximum clique size, if the dataset can run without pruning, then the best performance is often achieved by skipping the heuristic altogether, or using a single run of the degree-based heuristic. For graphs with a larger maximum clique size, we can improve the runtime by using a more accurate heuristic, even when no pruning is required to avoid OOM.

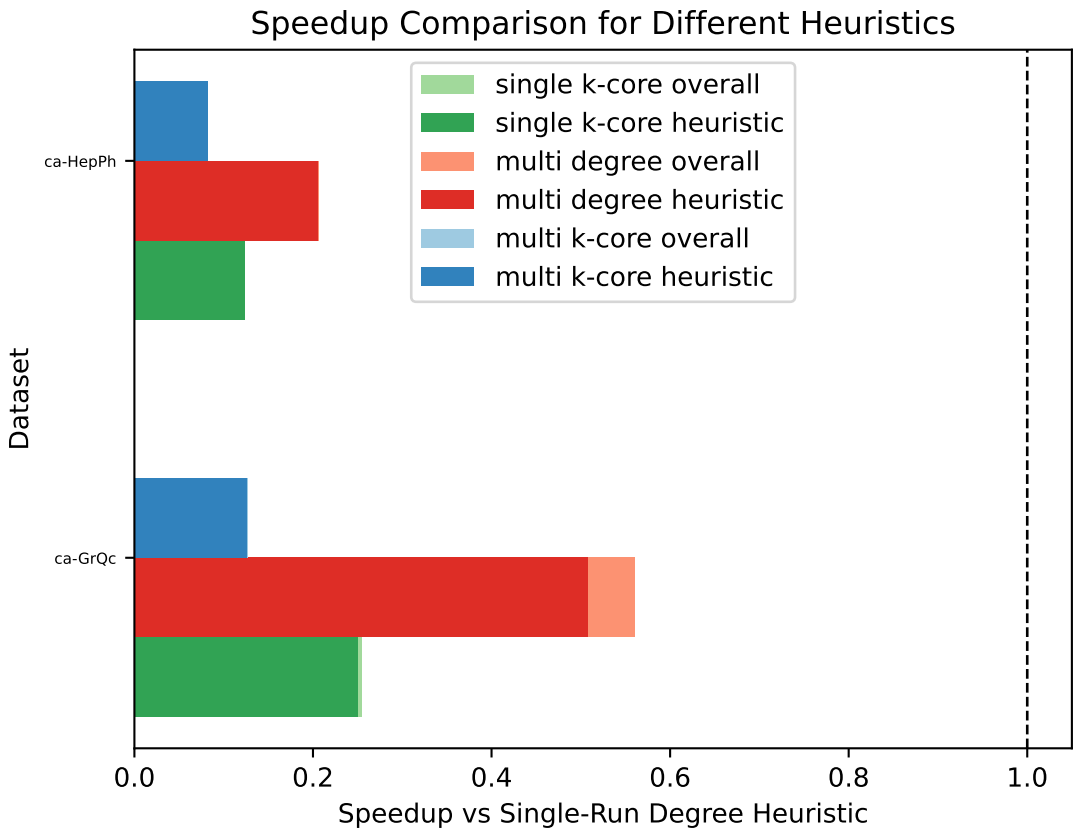

Figure 2.11. Overall speedups for each of our heuristics over baseline using a single run of the degree-based greedy heuristic. The darker "heuristic" bars represent the fraction of the total runtime spent on the heuristic computation. ca-HepPh and ca-GrQc are the only datasets that are OOM with no heuristic, but not with a lower bound from the single-run degree-based heuristic. Although this heuristic is fast, it is much less accurate than the others.

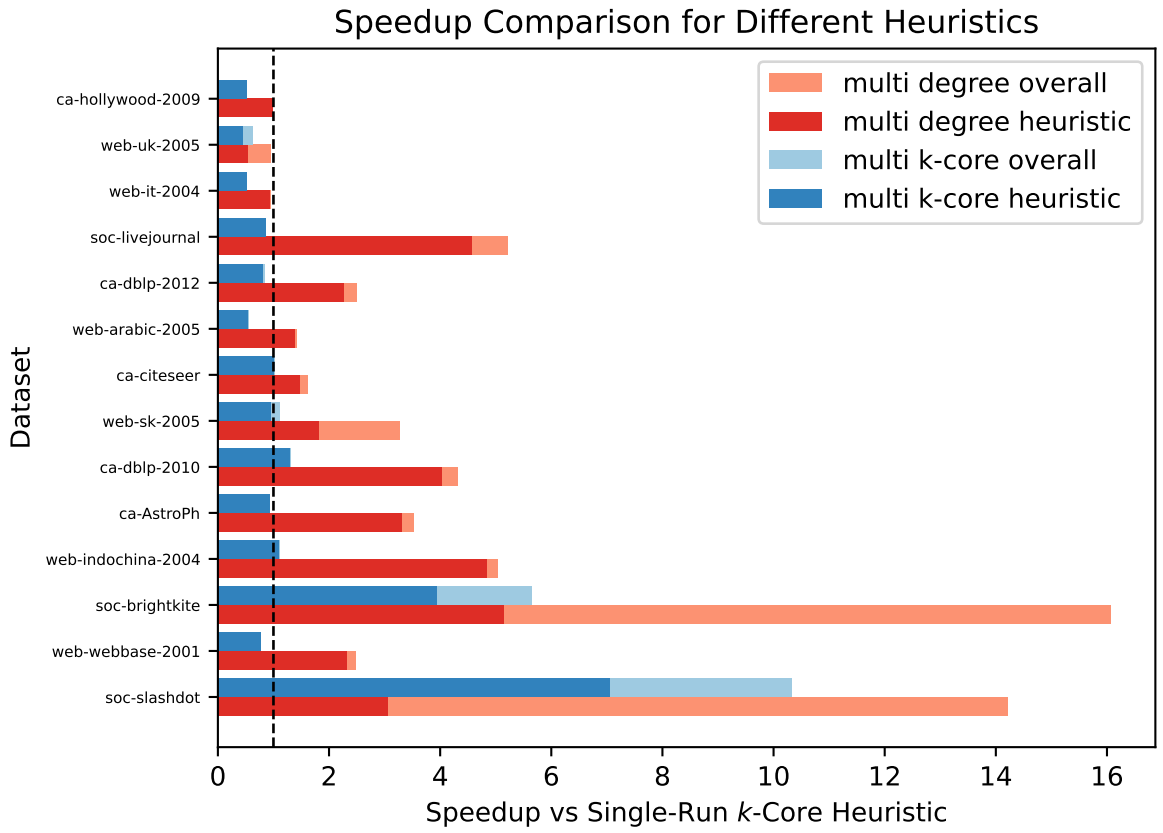

Figure 2.12. Overall speedups for the multi-run heuristics over baseline using a single run of the core-number-based heuristic. Datasets are sorted from top to bottom in order of decreasing maximum clique size. The darker "heuristic" bars represent the fraction of the total runtime spent on the heuristic computation. The multi-run heuristics provide better performance than a single run of the core-number-based heuristic for graphs with large maximum cliques. Additionally, the multi-run degree heuristic is almost always faster than the single-run corre-number-based heuristic.

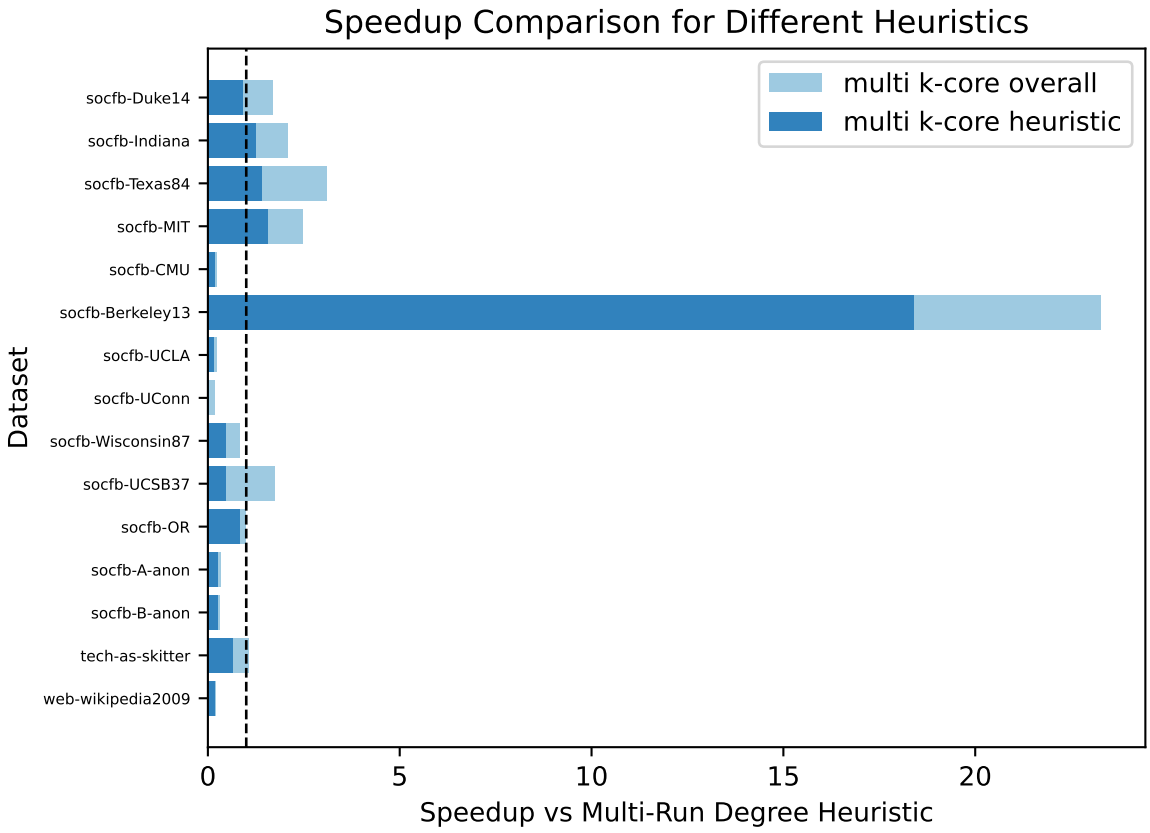

Figure 2.13. Overall speedups using $|V|$ runs of the core number heuristic over baseline using $|V|$ runs of the degree heuristic. Datasets are sorted from top to bottom in order of decreasing average degree. The darker "heuristic" bars represent the fraction of the total runtime spent on the heuristic computation. The multi-run degree heuristic is typically fastest, but the multi-run core-number heuristic has the advantage for graphs with higher average degree, because of the slight improvement in accuracy and the tighter bound for pruning using core numbers rather than degrees.

2.5.2.4 k-Core Computation

In most cases, the improvements in the heuristic and pruning are not worth the cost of computing the k-core vertex decomposition. As can be seen in Figure 2.17, the k-core vertex decomposition increases the heuristic runtime significantly. It is possible that choosing a different k-core implementation, could help to reduce the cost of this operation. However, since the core numbers only increase the accuracy of the lower bound by an average of 0.9% for the multi-run implementation, even a fast k-core computation is not likely to yield a large improvement in overall runtime for most datasets. Aside from increasing the accuracy of the heuristic, the core numbers provide a tighter bound for pre-pruning candidate vertices. Since all vertices core numbers are less than or equal to their degree, using core numbers can improve both the upper and lower bounds. However, in practice, we find that the improved heuristic lower bound accounts for most of the improvement in pruning, because this bound is applied in each iteration of the exact algorithm, while the vertex bounds are only helpful for pruning the initial clique list.

2.5.2.5 Number of Runs for Multi-Run Heuristics

Maximum accuracy for the multi-run heuristics is reached once at least 40% of vertices are used as seeds, while the best overall performance is achieved using only around 1% of vertices as seeds. Figures 2.14 and 2.15 show the accuracy and speedup results from our experiments using different numbers of seeds for the multi-run heuristic. We find that for both the degree and core number-based heuristics the accuracy increases rapidly as we increase the number of seeds from 0.1% up to around 5-10% of vertices, then begins to level off, with only small improvements in the lower bound after that. As we would expect, we find the largest cliques when using the highest-degree (or core number) vertices as seeds, and running the greedy heuristic with more of the lowest-degree vertices as seeds does not reveal previously-undiscovered larger cliques. We find that increasing the number of runs up to around 1% of vertices improves overall performance, but after this point, the pruning improvements from small increases in the lower bound do not outweigh the additional work in the heuristic step. This is particularly true for the degree heuristic, where the number of runs of the heuristic has a much larger proportional effect than for the core number heuristic, which dedicates a significant fraction of the overall runtime

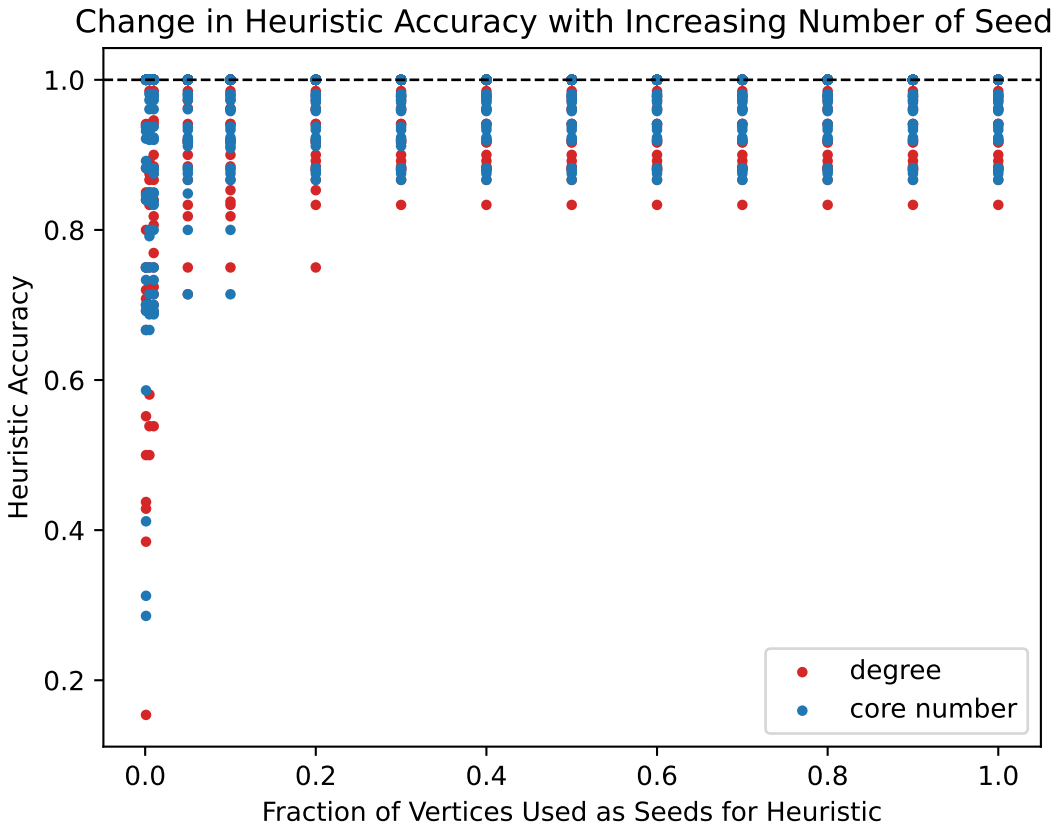

Figure 2.14. Comparison of accuracy for multi-run heuristics using different numbers of seed vertices. Accuracy generally increases with the number of seeds, but after the 30% highest degree vertices, no new, larger cliques are found from running the heuristic with more lower-degree vertices. For most graphs, using only the 5-10% of vertices with the highest degree will achieve the same accuracy as using all vertices as seeds.

to computing the core numbers, regardless of the number of seeds used in the heuristic.

2.5.2.6 Recommendations for Selecting a Heuristic

The best performance is usually achieved by using the simplest heuristic that provides enough pruning to stay withing memory limits. The best default choice for an unknown dataset is the multi-run degree heuristic. These analyses reveal a complicated picture for determining which heuristic will provide the fastest runtime. As a general rule, the fastest runtime is typically achieved by using the simplest heuristic for which the pruning is sufficient to avoid running out of memory. For graphs with fewer edges and/or lower degree, likely no

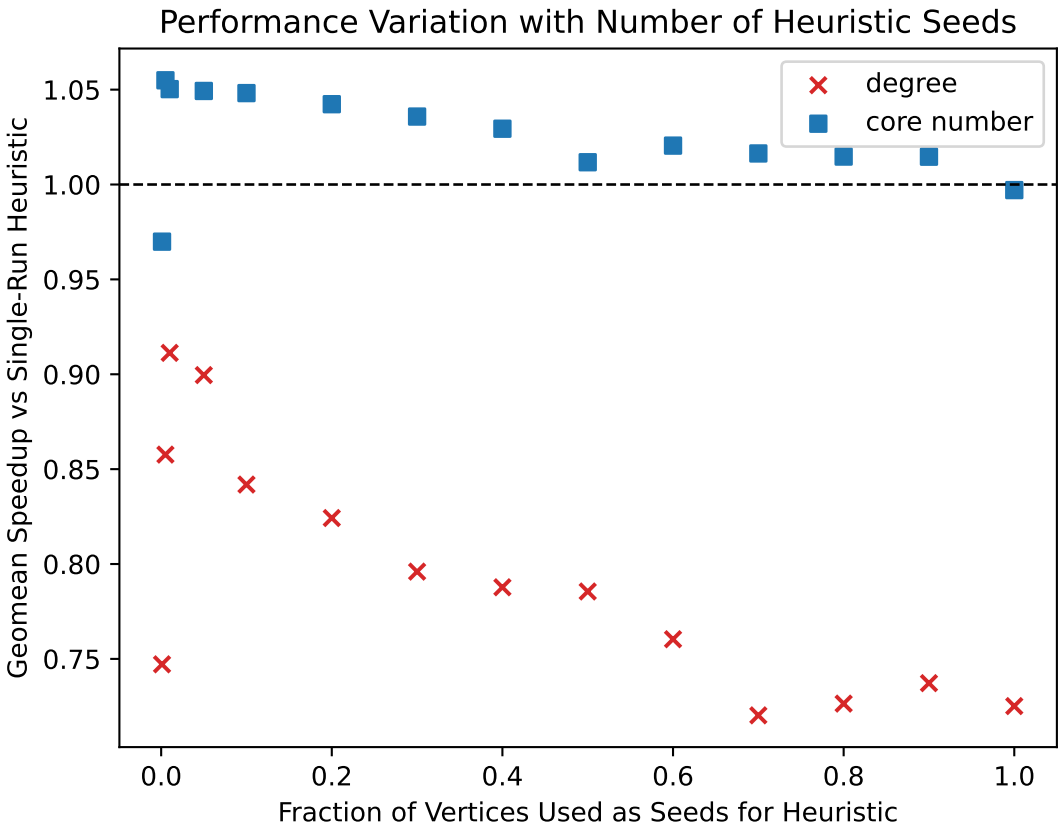

Figure 2.15. Overall speedup over single-run heuristic for multi-run heuristics using different numbers of seed vertices. For the core number heuristics, speedup is multi-run core number heuristic over single-run core number heuristic, and the same for the degree heuristics. Because the k-core decomposition takes a large fraction of the runtime, the overall speedups are higher for the core-number-based heuristics than the degree-based heuristics. For both, we see that, for datasets in that do not require multi-run heuristics to avoid OOM (where we have a valid baseline), performance is best when about 5% of vertices are used as seeds, because this offers a good balance of adequate pruning and reduced preprocessing time.

heuristic will be needed, while larger and higher degree graphs will benefit from the multi-run heuristics. Figures 2.16, 2.17, and 2.18 show the relationship between heuristic runtime and maximum clique size, number of edges, and average degree. We see that heuristic runtime typically increases as the number of edges and/or maximum clique size increases, but not with increasing average degree. This further supports the conclusion that a more complex heuristic is likely to be beneficial for graphs with high average degree. As described in Chapter 2.5.2.4, the k-core computation adds considerable runtime, and a relatively small improvement in accuracy. Therefore, the best combination of accuracy and runtime is the multi-run degree heuristic, and without any further knowledge about the dataset, we find this heuristic to be the best to start with, forgoing the k-core computation. Then only if the run is out of memory with this heuristic, would we recommend trying the multi-run core number version instead.

2.5.3 Other Preprocessing Options

2.5.3.1 Orienting Graph By Degree

As described in Chapter 2.4.5, we chose to orient the graph by degree when forming the initial 2-clique list, which allows us to avoid storing duplicate cliques. In this section we compare orientation by degree versus orientation by index and the effect on memory use and runtime.

Pruning **For most datasets, using degree orientation allows us to prune a large fraction of the candidate cliques that were still unpruned with index orientation**, as shown in Figure 2.19. This is because vertices with lower degree have shorter adjacency lists, so selecting these vertices as the source vertices means that their initial candidate lists are shorter on average, so we can usually prune more of them by comparing the candidate list length to the lower bound on the maximum clique size from the heuristic. This reduces our chances of running out of memory, and can also improve the runtime due to the reduced workload for the main loop.

Speedup **On average, degree orientation does provide an overall speedup, but for many graphs the increase in pruning has little effect on the overall runtime.** Figure 2.20 shows the speedups we achieve from using degree orientation over index orientation. Here the improvement is not as consistent as with pruning. Some datasets receive a sizable speedup from this one small change, particularly some datasets with more edges. However, some datasets with large relative increases in pruning do not receive speedups. This is because some datasets have very

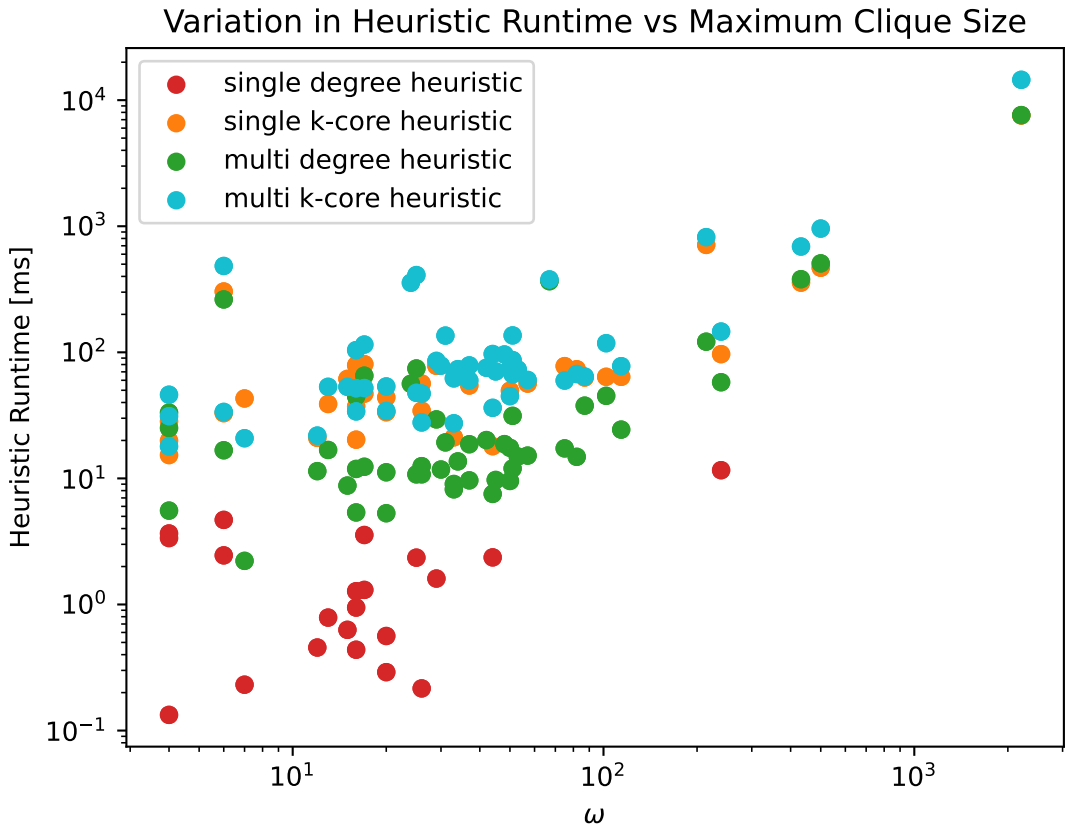

Figure 2.16. Runtime for the heuristic only versus the maximum clique size. The runtime of each of our heuristics increases with the maximum clique size.

good pruning with index orientation, and the improvement from degree orientation does not help because the workload is already too small to keep the GPU busy. For these datasets, often the majority of the runtime is spent in preprocessing, so additional pruning to speed up the main loop does not have much effect. Degree orientation does have a small runtime cost over index orientation, because it requires an additional memory access for each vertex when forming the 2-clique list in order to check the vertex's degree. Still, switching to degree orientation provides a geometric mean speedup of $1.2x$ over index orientation across all datasets, so it is clearly the better option.

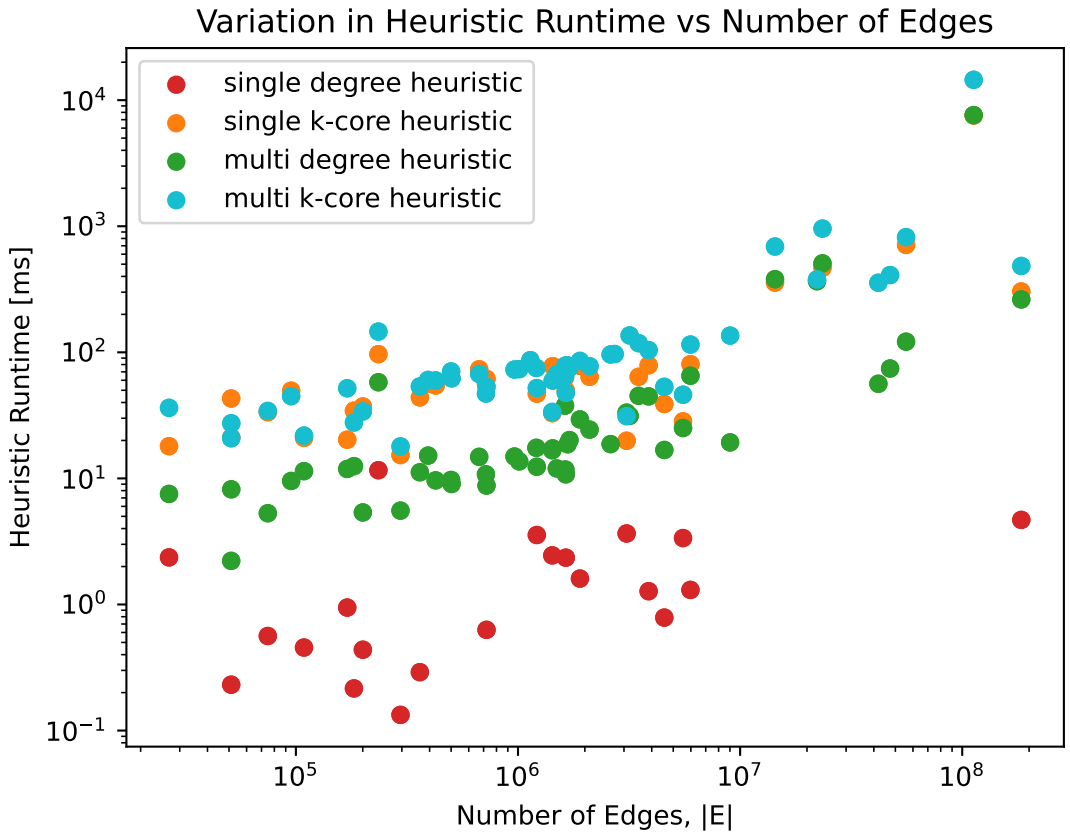

Figure 2.17. Runtime for the heuristic only versus the number of edges in the graph. The runtime of each of our heuristics increases with the number of edges in the graph.

2.5.3.2 Sorting Vertices within Candidate Lists by Degree

Another optional preprocessing operation is sorting vertices by degree within their candidate lists. As described in Chapter 2.4.5, sorting vertices in increasing order should lead to better pruning and lower latency for edge lookups. In this section, we compare memory use and runtime for sorting by degree versus the default ordering by index.

Pruning **We find that sorting vertices by degree within candidate lists improves pruning significantly for many graphs, particularly ones with high variation in degree**, as shown in Figure 2.21. We expected that graphs with more variation in vertex degree should see a bigger pruning increase, since the pruning benefits come from reducing the degree of vertices at the beginning of candidate lists. We do find that the datasets that achieve the largest pruning

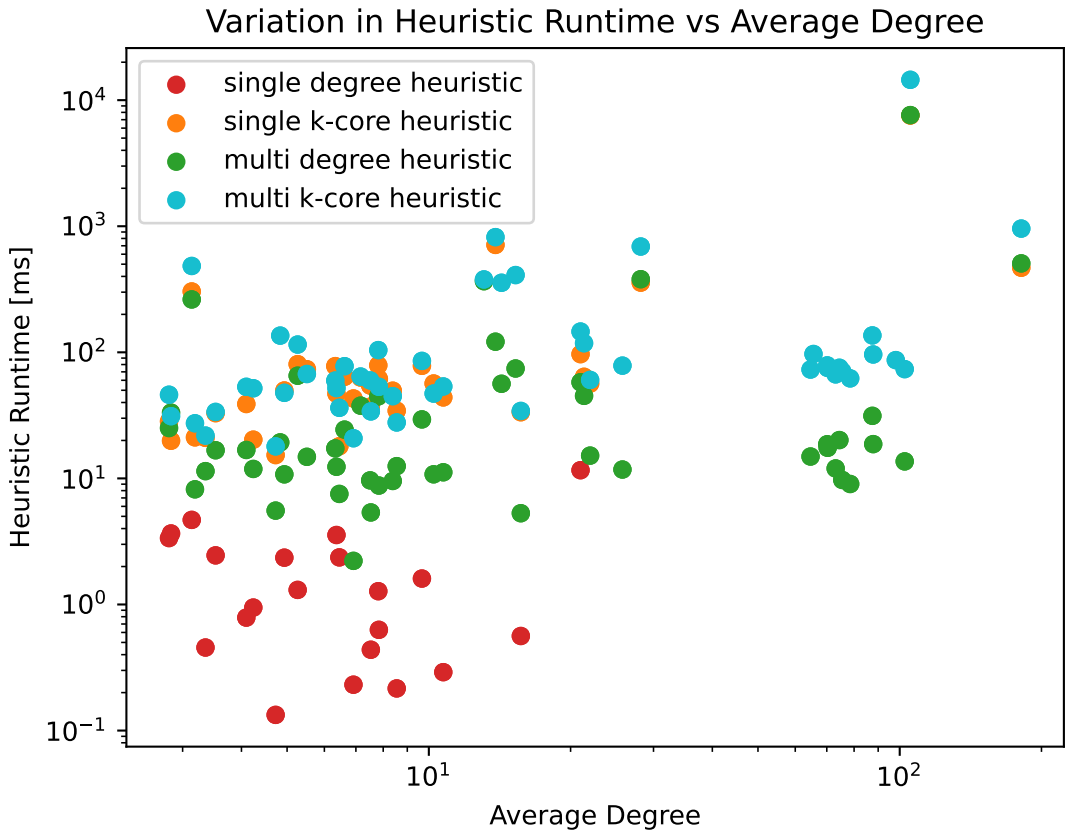

Figure 2.18. Runtime for the heuristic only versus average vertex degree. The runtimes for our heuristics are not correlated with the average degree of the graph.

improvements have higher standard deviation in vertex degree, though not all graphs with high degree variation receive a big improvement. For some of these datasets, no further pruning improvements were possible, since we are already able to prune all vertices except those in the maximum clique. Others only receive modest pruning improvements because they are hard to prune graphs, as described in Chapter 2.5.2.3, where the majority of vertices have higher degree than the maximum clique and are challenging to prune before later iterations of the main loop.

Speedup **As with orientation by degree, we find that many datasets with large relative increases in pruning do not achieve large speedups from sorting vertices by degree, but for most graphs it is worthwhile.** The datasets with the largest speedups are some of the Facebook datasets with the highest standard deviation in degree, as shown in Figure 2.22. For a few of

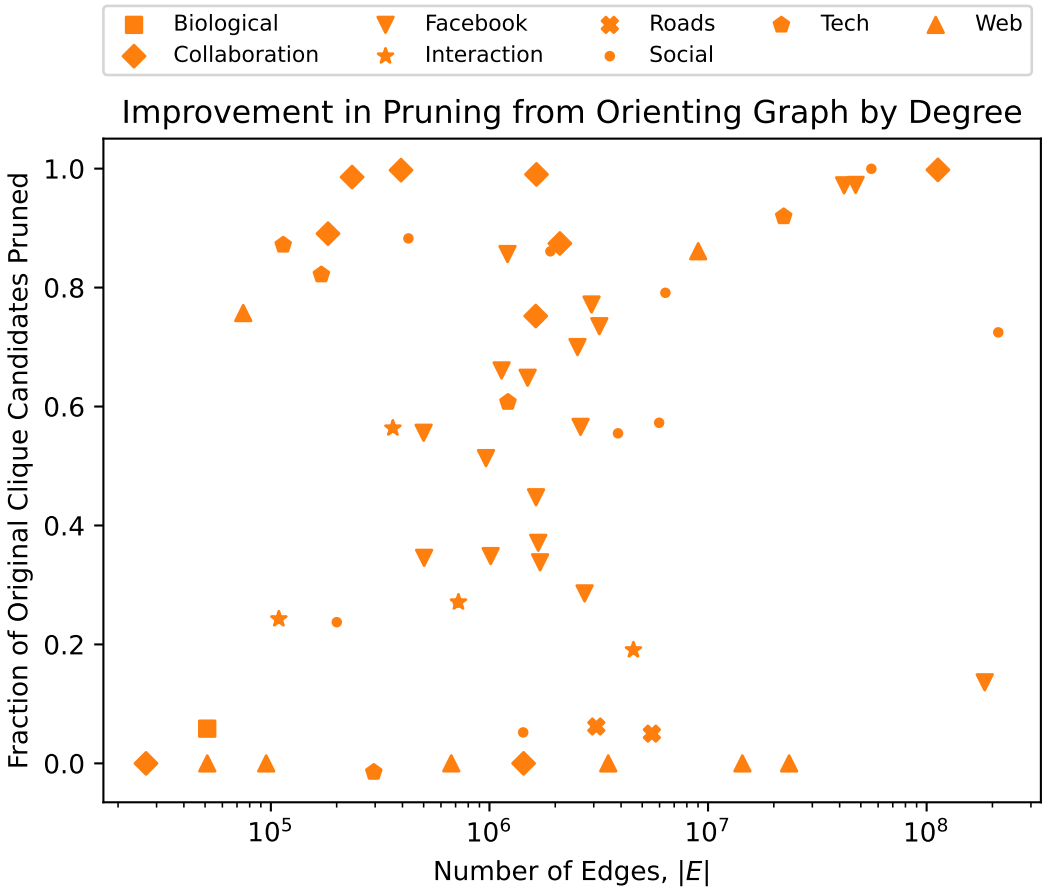

Figure 2.19. Fraction of cliques that are unpruned when using index orientation, but are pruned when using degree orientation. Higher points indicate better pruning. Data is for full breadth-first maximum clique using our multi-run degree-based greedy heuristic. For most datasets, we see a significant improvement in pruning when switching to degree orientation, rather than index orientation.

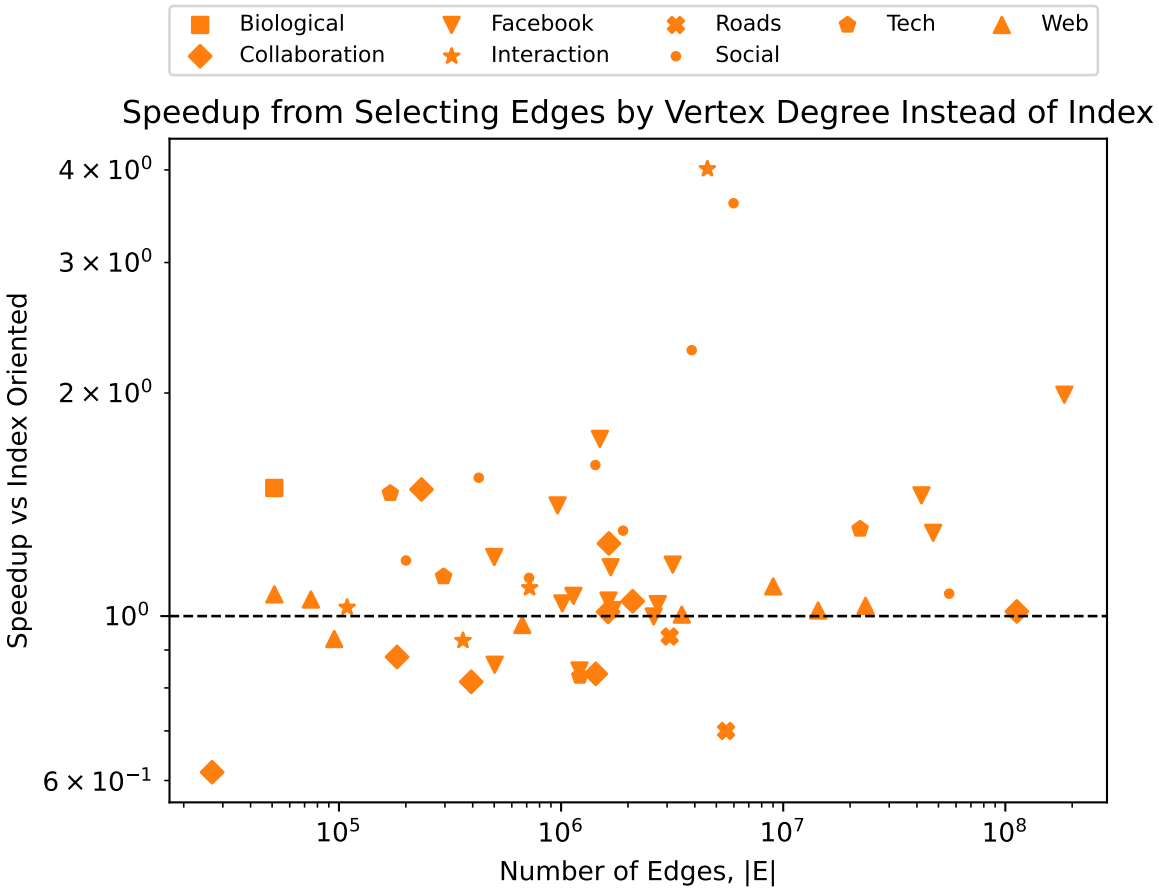

Figure 2.20. Overall speedup from using degree orientation over index orientation. Data is for full breadth-first maximum clique using our multi-run degree-based greedy heuristic. Many datasets with large increases in pruning (see Figure 2.19) do not see similar improvements in runtime, but degree orientation does provide an overall speedup for most graphs.

these graphs, this simple operation returns a speedup of around $10x$. However, many datasets that have a large relative improvement in pruning do not receive overall speedups, or only very small speedups. One reason is that there is an additional preprocessing runtime cost to sort the vertices. Although this is an efficient operation on the GPU, if the reduction in runtime for the main loop is small, it may not be worth it. Secondly, as we found for graph orientation by degree in Chapter 2.5.3.1, for some graphs the candidate lists are already pruned smaller than the width of the GPU and further pruning has little runtime benefit. Overall, sorting vertices by degree within their candidate lists results in a geometric mean runtime speedup of $1.5x$ and is likely worthwhile for graphs with standard deviation in degree ≥ 20.

2.5.4 Windowing

Another option in our implementation is to break up the search into smaller windows of candidates and fully explore one window before moving onto the next. Our goal with windowing was to reduce the number of candidates that need to be stored simultaneously, thereby reducing the memory requirements and allowing us to find the maximum clique for more datasets without running out of memory. We can see from the overall performance results in Figures 2.2 and 2.3 that using windowing generally decreases throughput, as we would expect when reducing the available parallel work. In this section, we look at how the choice of window size creates a trade-off between this runtime increase and memory use reduction. We also test whether we can achieve any benefits from altering the order of the search by sorting the source vertices in the 2-clique list by their degrees.

2.5.4.1 Memory Use and Pruning

Windowing reduces the memory requirements by an average of 85–94%. The smaller the window, the less memory required to store candidates. We also find that searching the neighborhoods of more highly connected vertices first requires more memory than searching less connected vertices first or a random ordering. For the regular breadth-first implementation, all candidates are stored until the search is complete, so memory use is only reduced by improving pruning. With windowing, memory use and pruning are no longer equivalent. Memory requirements are determined by the largest clique list subtree generated from a single window, which is affected by both pruning quality and window size. Pruning is affected

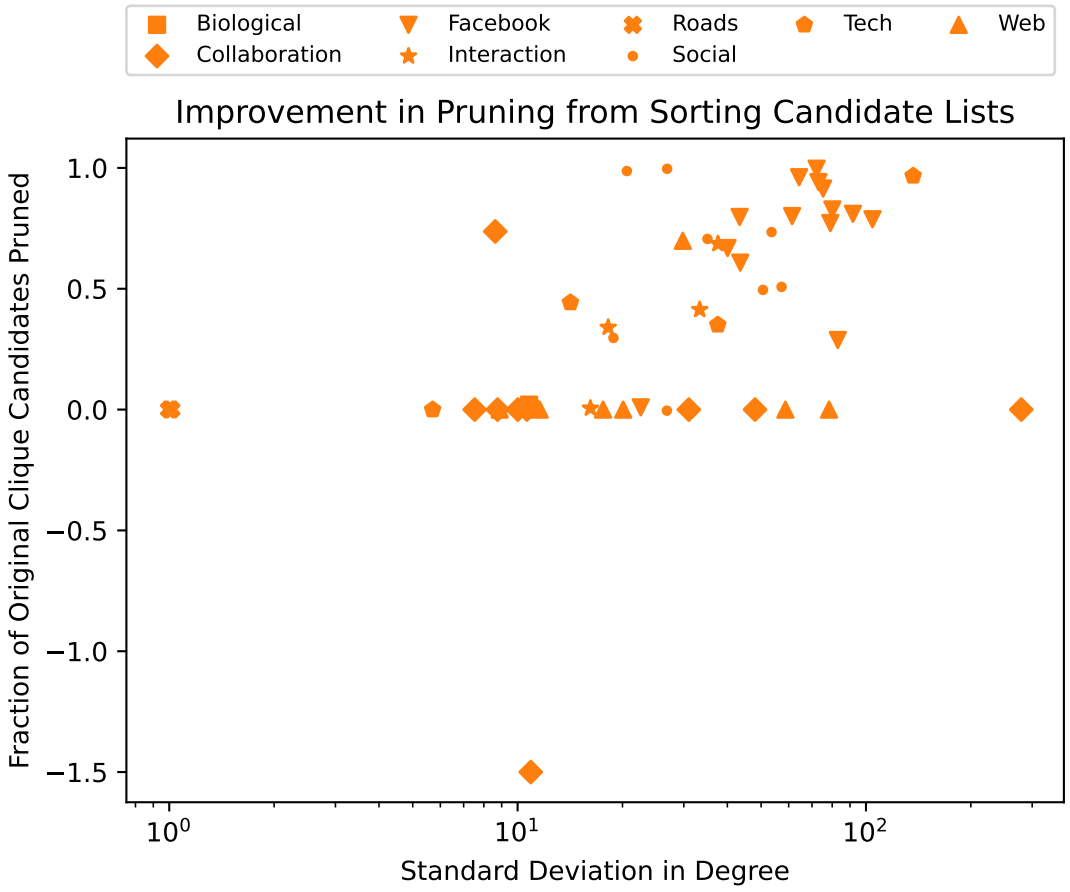

Figure 2.21. Fraction of cliques that are unpruned when using index ordering within candidate lists, but are pruned when candidates are sorted from low to high degree. Data is for full breadth-first maximum clique using our multi-run degree-based greedy heuristic. For graphs with higher variation in degree, sorting candidate lists by degree improves pruning. The dataset showing an increase in the number of candidate cliques is ca-condMat, which achieves extremely high levels of pruning for index or degree ordering.

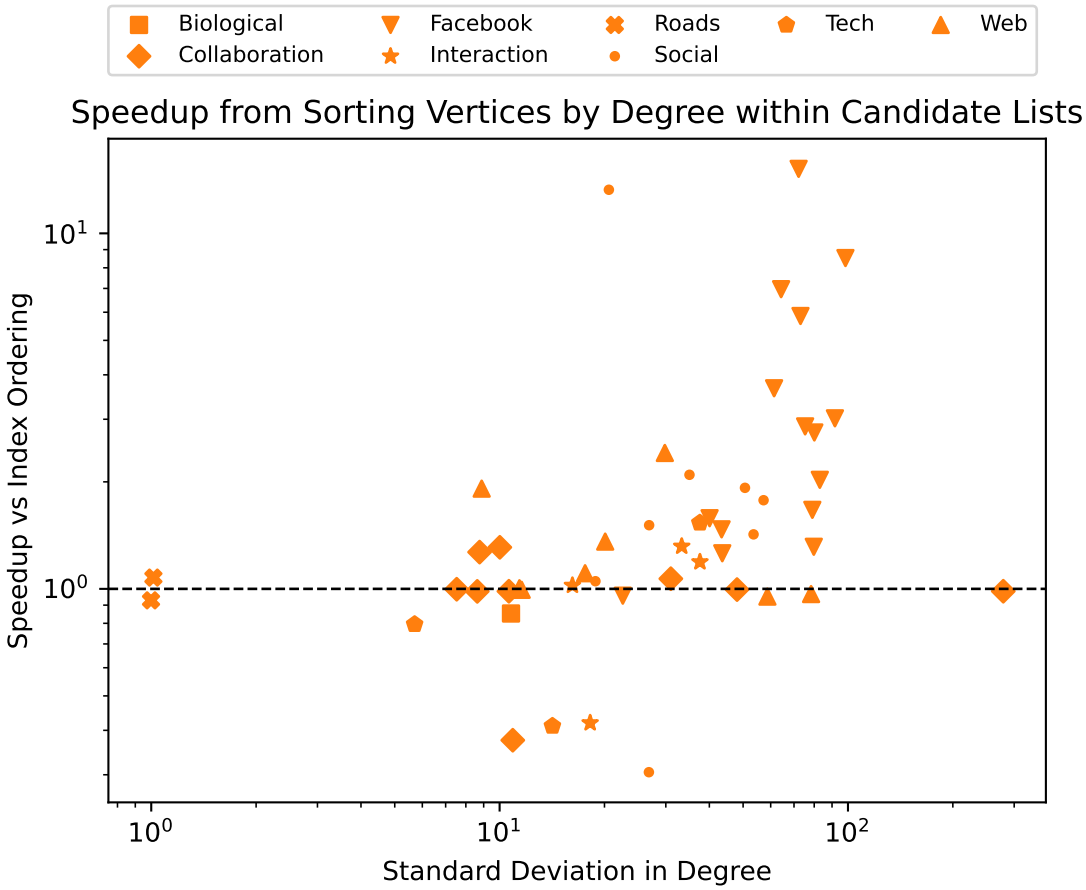

Figure 2.22. Overall speedup from sorting vertices within candidate lists by degree orientation over leaving the vertices sorted by index. Data is for full breadth-first maximum clique using our multi-run degree-based greedy heuristic. Graphs with high deviation in degree tend to see greater speedups than graphs with less variation in degree.

by all factors discussed in previous sections, and can also be improved when a new best clique is found, which increases the lower bound and improves pruning for later windows. Pruning results in less work, but does not necessarily have a large effect on the peak memory use, because for the windowed version, we only need to store the clique lists generated from one window (as well as that of the best clique so far).

Figure 2.23 shows the reduction in memory usage and Figure 2.24 shows the improvement in pruning over the regular breadth-first implementation. Unsurprisingly, smaller window sizes provide larger memory savings and also a small improvement in pruning. We find that by using windowing, we are able to solve 4 more (for a total of 56 out of the 58 datasets) of the graph datasets that run out of memory when running the full breath-first implementation. Sorting source vertices in descending degree order, thereby searching more highly connected vertices' neighborhoods first, uses more memory and achieves less pruning than searching in order of ascending degree or (randomized) index order. We might expect that prioritizing highly connected vertices would improve memory usage and pruning because the maximum clique(s) are more likely to contain these high-degree vertices, but we are also orienting the graph by degree, so larger cliques are more likely to be in low-degree vertices' candidate lists than they would be with index-based orientation. However, we also find that sorting the source vertices in ascending order does not significantly improve pruning or peak memory usage over random order, suggesting that it is generally challenging to predict which sublist(s) the largest clique(s) are located in.

2.5.4.2 Runtime

The smaller the window, the longer the runtime. Changing the traversal order does not have a significant effect on runtime. Figure 2.25 shows the effect of window size and sorting source vertices on runtime. We see an increase in runtime as the window size shrinks, which is to be expected, because we run the main loop on each window sequentially, so as the number of windows increases, so does the runtime. Additionally, depending on the number of candidates generated in the search, smaller windows may not provide enough parallelism to keep the GPU filled with work. Sorting source vertices does not have a consistent effect on runtime. This suggests that the performance is limited by the lack of available parallelism, since reducing the

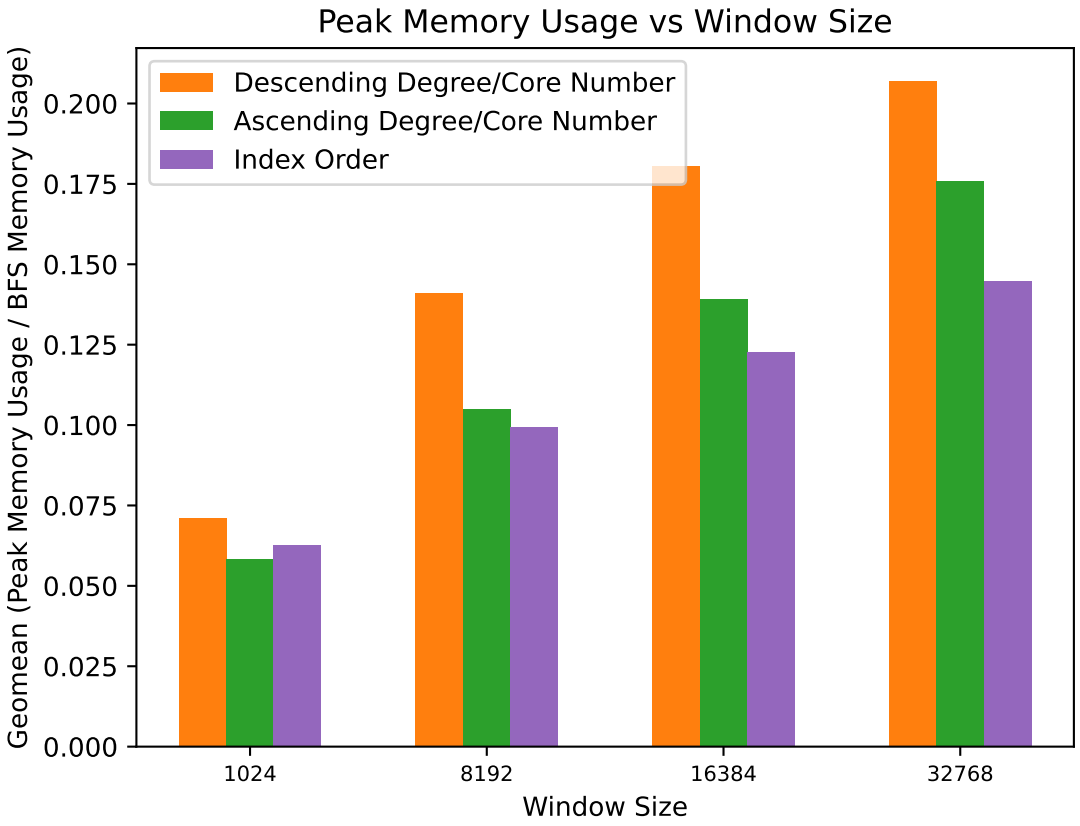

Figure 2.23. Memory usage for windowed computation compared to full breadth-first maximum clique. Uses multi-run degree-based greedy heuristic, orientation by degree, and candidates sorted by degree within candidate lists. Smaller window sizes provide a much smaller peak memory usage.

amount of work is not affecting the runtime. Overall, we conclude that there is no memory or performance argument for changing the order of search from a randomized order; however, depending on the default ordering of the graph dataset, it may be worthwhile to try sorting vertices in ascending degree order if needed to avoid running out of memory.

2.5.4.3 Recursive Windowing

Although we only implement windowing on the first level of the search, it is possible to perform windowing in later stages and/or multiple times during the search (i.e. exploring a subset of the candidates generated from a subset of an earlier set of candidates) to further reduce memory requirements. The results shown in this section indicate that this would be an effective strategy

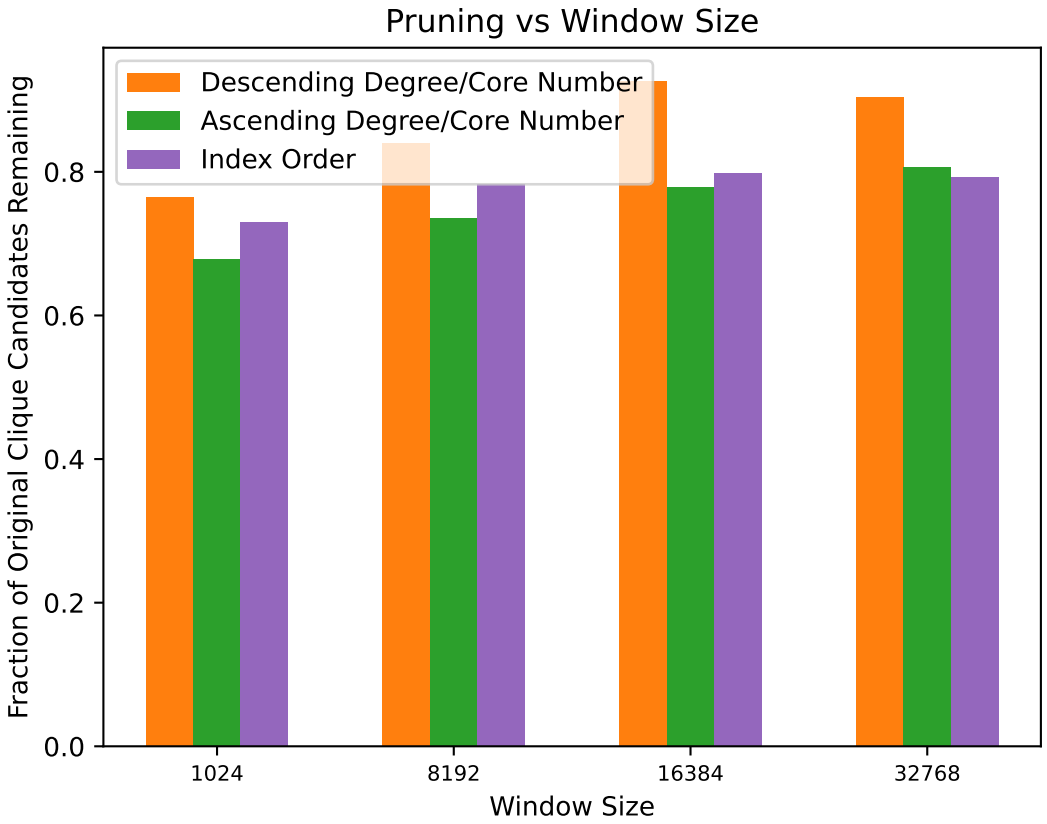

Figure 2.24. Fraction of cliques found in full breadth-first maximum clique that are still unpruned with windowing. Uses multi-run degree-based greedy heuristic, orientation by degree, and candidates sorted by degree within candidate lists. Windowing provides moderate improvements in pruning, in the event that cliques discovered when solving earlier windows are larger than the clique found by the heuristic.

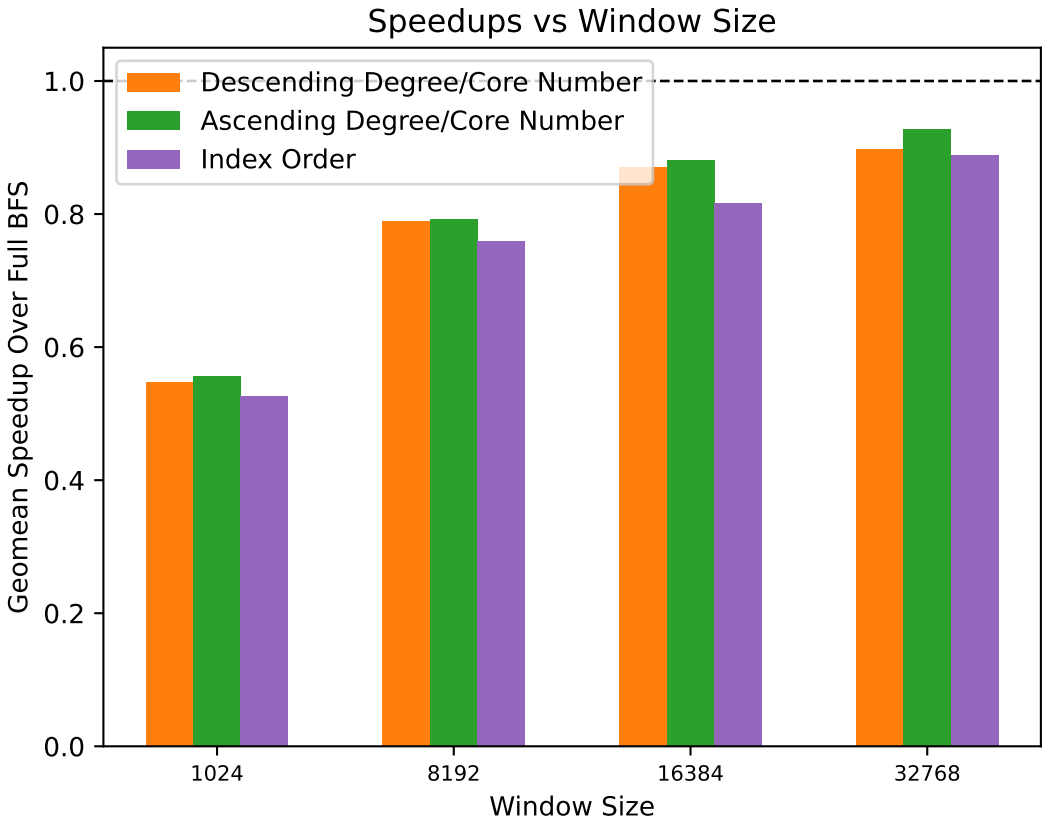

Figure 2.25. Speedup for windowed version over full breadth-first maximum clique. Uses multi-run degree-based greedy heuristic, orientation by degree, and candidates sorted by degree within candidate lists. Using a smaller window size typically results in a longer runtime, and overall, the windowed version has a longer runtime than the full breadth-first version, when we do not run out of memory for storing candidate cliques.

for reducing memory usage, but that the performance cost will also be quite high. Multiple windows could be explored simultaneously using different blocks in order to increase parallelism, but because the number of candidates generated by a window is unpredictable, managing the memory resources is challenging. Increasing windowing also moves the implementation further towards a regular depth-first implementation, which, as discussed in Chapter 2.2.3, is not ideal for GPU performance.

2.6 Conclusions

Although there are a variety of strategies for improving pruning, we find that the fastest configuration is typically one that uses the least (fastest) preprocessing while simultaneously managing to avoid OOM. The goal is to choose a pruning strategy that is ”good enough” and not expend any further effort on pruning after that. This indicates that the breadth-first strategy was a good choice for the GPU, because indeed, the GPU is happy with lots of work, even if it could be “easily” eliminated. However, BFS is not ideal for this problem in general, because (1) memory limits are easily reached with a combinatorial problem like this, and (2) the search will never finish early because the depth itself is the value we are solving for.

In this chapter, we explored many algorithmic techniques for reducing the memory footprint of an application. Another approach to reducing memory use is by utilizing space-efficient data structures. In the next chapter, we describe our GPU implementations of two types of quotient filters, which are probabilistic data structures that are used to perform membership queries while using only a fraction of the memory footprint of an exact data structure.

2.7 Pseudocode

Algorithm 1 k-core vertex decomposition

```
1: function kCoreDecomposition(G = (V, E))
2:     F ← V                                   ▷ begin with all vertices in frontier
3:     D ← {}                                  ▷ vertices removed
4:     k ← 0
5:     D_k ← {}
6:     while |D| < |V| do
7:         while F ≠ ∅ do
8:             for all f_i ∈ F do              ▷ filter to select all vertices with degree < k
9:                 if degree(f_i) ≤ k then
10:                    kcores_i ← k
11:                    D_new ∪ f_i
12:            F ← n_i ∈ (N(D_new) − D)
13:            for all f_i ∈ F do              ▷ advance to neighbors and reduce their degrees
14:                AtomicDECR(degree(f_i))
15:            D ∪ D_new
16:        INCR(k)
17:    return kcores
```

Algorithm 2 Multi-run greedy heuristic

function GETNEIGHBORCOUNTS(G, *vertices*, *neighborCounts*)
  neighborCounts[`threadID`] $\leftarrow |N($*vertices*[`threadID`]$)|$

function SETUPNEIGHBORTHRESHOLDS(G, *vertices*, *neighborCounts*, *vertexThresholds*, *indices*, *neighbors*, *neighborThresholds*)
  offset $\leftarrow$ *indices*[`threadID`]
  count $\leftarrow 0$
  for $u \in N(v)$ **do**
    neighbors[*offset* + *count*] $\leftarrow u$
    neighborThresholds[*offset* + *count*] $\leftarrow$ *vertexThresholds*[u]
    INCR(*count*)

function CHECKCONNECTIONS(G, *neighbors*, *indices*, *maxIndices*, *flags*, *connectedCounts*)
  $v \leftarrow$ *neighbors*[*maxIndices*[`threadID`]]
  currentIndex $\leftarrow$ *indices*[`threadID`]
  segmentEnd $\leftarrow$ *indices*[`threadID` + 1]
  count $\leftarrow 0$
  while *currentIndex* $<$ *segmentEnd* **do**
    $u \leftarrow$ *neighbors*[*currentIndex*]
    if $u \in N(v)$ **then**
      flags[*currentIndex*] $\leftarrow$ TRUE
      INCR(*count*)
    else
      flags[*currentIndex*] $\leftarrow$ FALSE
    INCR(*currentIndex*)
  connectedCounts[`threadID`] $=$ *count*

function MULTIRUNGREEDYHEURISTIC(G, *vertices*, *vertexThresholds*, h)
  ▷ *vertices* sorted in order of descending degree or core number
  for all *vertices* **do**
    GETNEIGHBORCOUNTS(G, *vertices*, *neighborCounts*)
  indices $\leftarrow$CUBSCAN(*neighborCounts*)
  for all *vertices* **do**
    SETUPNEIGHBORTHRESHOLDS(G, *vertices*, *neighborCounts*, *vertexThresholds*, *indices*, *neighbors*, *neighborThresholds*)
  numSegments $\leftarrow h$
  $\bar{\omega} \leftarrow 1$
  while *numSegments* > 0 **do**
    maxIndices $\leftarrow$CUBSEGMENTEDMAX(*neighborThresholds*)
    for all segments **do**
      CHECKCONNECTIONS(G, *neighbors*, *indices*, *maxIndices*, *flags*, *connectedCounts*)
    (*neighbors*, *numCandidates*) $\leftarrow$CUBSELECT(*neighbors*, *flags*)
    (*neighborThresholds*, *numCandidates*) $\leftarrow$CUBSELECT(*neighborThresholds*, *flags*)
    if *numCandidates* $= 0$ **then**
      break
    (*nonzeroCounts*, *numSegments*) $\leftarrow$CUBSELECTIF(*connectedCounts*) ▷ keep values > 0
    indices $\leftarrow$CUBSCAN(*nonzeroCounts*)
    INCR($\bar{\omega}$)
  return $\bar{\omega}$

Algorithm 3 2-clique list set-up

function COUNTTWOCLIQUES(G, $\bar{\omega}$, *filterThresholds*, *sublistLengths*, *flags*)
  $v \leftarrow$ `threadID`
  candidates $\leftarrow 0$
  for $u \in N(v)$ **do**
    if $|N(v)| < |N(u)|$ or ($|N(v)| = |N(u)|$ and $v < u$) **then** ▷ orientation by degree
      if *filterThresholds*$[u] \geq \bar{\omega} - 1$ **then** ▷ prune vertices by degree/core number
        INCR(*candidates*)
  if *count* $\geq \bar{\omega} - 1$ **then** ▷ prune sublists by length
    sublistLengths[`threadID`] $\leftarrow$ *candidates*
    flags[`threadID`] $\leftarrow$ TRUE
  else
    sublistLengths[`threadID`] $\leftarrow 0$
    flags[`threadID`] $\leftarrow$ FALSE
  return

function OUTPUTTWOCLIQUES(G, $\bar{\omega}$, *filterThresholds*, *offsets*, $cliqueList_2$)
  $v \leftarrow$ `threadID`
  cliqueOffset $\leftarrow$ *offsets*[`threadID`]
  count $\leftarrow 0$
  for $u \in N(v)$ **do**
    if $|N(v)| < |N(u)|$ or ($|N(v)| = |N(u)|$ and $v < u$) **then** ▷ orientation by degree
      if *filterThresholds*$[u] \geq \bar{\omega} - 1$ **then** ▷ prune vertices by degree/core number
        $sublistID_2$[*cliqueOffset* + *count*] $\leftarrow v$
        $vertexID_2$[*cliqueOffset* + *count*] $\leftarrow u$
        INCR(*count*)
  return

function SETUPTWOCLIQUES(G, $\bar{\omega}$, *filterThresholds*, *skipMain*)
  ▷ optional preprocessing for windowed version: sort vertices in order of ascending or descending degree or core number
  for all $v \in G$ **do**
    COUNTTWOCLIQUES(G, $\bar{\omega}$, *filterThresholds*, *sublistLengths*, *flags*)
  ▷ prune flagged sublists shorter than $\bar{\omega}$
  (*vertices*, *numSublists*) $\leftarrow$CUBSELECT(*vertices*, *flags*)
  (*cliqueCounts*, *numSublists*) $\leftarrow$CUBSELECT(*sublistLengths*, *flags*)
  offsets $\leftarrow$CUBSCAN(*sublistLengths*)
  $cliqueCount_2 \leftarrow$ *indices*[*numSublists*]
  if *numSublists* $= 1$ and $cliqueCount_2 = \bar{\omega} - 1$ **then**
    skipMain $\leftarrow$ TRUE ▷ maximum clique was found by heuristic
  for all remaining vertices **do**
    OUTPUTTWOCLIQUES(G, $\bar{\omega}$, *filterThresholds*, *offsets*, $cliqueList_2$)
  ▷ optional: sort vertices by degree within candidate lists
  vertexID $\leftarrow$CUBSEGMENTEDSORTPAIRS(*vertexDegrees*, *vertexID*)
  return $cliqueList_2$

Algorithm 4 Breadth-first maximum clique enumeration

function COUNTCLIQUES(G, $cliqueList_k$, $\bar{\omega}$, *counts*)
  $i \leftarrow$ `threadID` $+ 1$
  connected $\leftarrow 0$
  while *sublistID*$_k$[`threadID`] $=$ *sublistID*$_k[i]$ **do**
    if *vertexID*$_k[i] \in N($*vertexID*$_k$[`threadID`]$)$ **then**
      INCR(*connected*)
    INCR(i)
  if *connected*+k$<\bar{\omega}$ **then** ▷ pruning by sublist length
    connected $\leftarrow 0$
  counts[`threadID`] $=$ *connected*
  return

function OUTPUTNEWCLIQUES(G, *cliqueList*$_k$, *offsets*, *cliqueList*$_{k+1}$)
  $i \leftarrow$ `threadID` $+ 1$
  cliqueOffset $\leftarrow$ *offsets*[`threadID`]
  if *cliqueOffset* $=$ *offsets*[`threadID` $+ 1$] **then**
    return
  count $\leftarrow 0$
  while *sublistID*$_k$[`threadID`] $=$ *sublistID*$_k[i]$ **do**
    if *vertexID*$_k[i] \in N($*vertexID*$_k$[`threadID`]$)$ **then**
      vertexID$_{k+1}$[*cliqueOffset* $+$ *count*] $\leftarrow$ *vertexID*$_k[i]$
      sublistID$_{k+1}$[*cliqueOffset* $+$ *count*] $\leftarrow$ `threadID`
      INCR(*count*)
    INCR(i)
  return

function MAXCLIQUES(G, $\bar{\omega}$, *cliqueList*$_k$, *cliqueCount*$_k$)
  $k \leftarrow 2$
  while *cliqueCount*$_k > 1$ **do**
    for all candidates in *cliqueList*$_k$ **do**
      COUNTCLIQUES(G, *cliqueList*$_k$, *counts*)
    offsets $\leftarrow$CUBSCAN(*counts*)
    cliqueCount$_{k+1} \leftarrow$ *offsets*[*cliqueCount*$_k - 1$]
    if *cliqueCount*$_{k+1} = 0$ **then**
      break
    for all candidates in *cliqueList*$_k$ **do**
      OUTPUTNEWCLIQUES(G, *cliqueList*$_k$, *offsets*, *cliqueList*$_{k+1}$)
    if *cliqueCount*$_{k+1} = \bar{\omega} - k + 1$ **then**
      break ▷ maximum clique was found by heuristic
    INCR(k)
  return *cliqueList*$_k$

Chapter 3

Quotient Filters: Approximate Membership Queries on the GPU

3.1 Introduction

In this work, we focus on an *approximate membership query* (AMQ) data structure for the GPU. AMQs, such as Bloom filters [3], are probabilistic data structures that support lookup and update operations on a set S of keys. The chief advantage of AMQs lies in their space efficiency: they use much less space than traditional dictionaries like hash tables. This advantage is particularly important on GPUs, because even today's most powerful GPUs have a relatively small memory (e.g., NVIDIA's Tesla P100 has 16 GB of DRAM). Since many databases, networks, and file systems benefit from the quick filtering of negative queries (often to avoid costly disk or network accesses), AMQs have found wide use. Such applications are emerging research areas on GPUs [9, 36, 37].

This space advantage comes with a trade-off: in an AMQ, membership queries are only approximate. For a key $k \in S$, LOOKUP(k) returns "present," but for $k \notin S$, LOOKUP(k) can also return "present," with probability at most ϵ, where ϵ is a tunable false-positive rate. An AMQ storing n items with a false positive rate ϵ requires at least $n \log(1/\epsilon)$ bits, and AMQs exist that achieve this bound, up to low order terms. So the introduction of a false positive rate allows the AMQ to use many fewer bits than an error-free data structure.

Bloom filters (BF) are the most well-known AMQ. A Bloom filter represents a set with a bit array. To insert a value, the filter hashes the value using k hash functions whose outputs each

correspond to a location in the bit array and sets the bit at each of these locations. To perform a lookup on a value, the BF computes the k hashes and checks whether the bits at all of the corresponding locations are set.

The Bloom filter is straightforward to implement, but has three significant shortcomings: it achieves poor data locality, it does not support delete operations, and it is still a multiplicative factor bigger than the optimal bound noted above. We implement an alternative to the BF: the quotient filter (QF). The quotient filter [2] is designed to maintain locality of data, and beyond supporting all the functionality of the BF, it also supports deletions and the merging of filters. Additionally, the QF can be extended to include counters [29]. We implement two versions of the QF on the GPU, the standard quotient filter (SQF) and the rank-and-select-based quotient filter (RSQF), and compare their relative strengths and weaknesses on this massively parallel architecture. Prior to our work, complete QF implementations have been limited to the CPU.

We describe new algorithms for parallel inserts into SQFs and RSQFs. We also investigate techniques for parallelizing a bulk build of the filter, when a significant portion of the full dataset is available at the outset. We find that this involves implementing a parallel scan with a non-associative operator, and we present implementations of three distinct approaches to this problem. We show that our GPU SQF achieves significantly faster lookups, has faster bulk build times, and uses significantly less memory than BloomGPU. In addition to enabling new applications with increased functionality, our GPU quotient filters can be used as a drop-in replacement for a Bloom filter in any of their existing GPU applications [16, 21, 22, 26, 27, 39].

3.2 Related Work

Prior work on AMQs for the GPU concentrates on Bloom filters. Much of this work has focused solely on using the GPU to accelerate lookup queries, using the CPU for filter construction and updates [21, 22, 26, 27, 39]; however, Costa et al. [7] and Iacob et al. [16] do implement both the filter build and queries on the GPU. Costa et al.'s implementation was open-sourced, so we chose to use their filter as our primary reference for comparison. Their BloomGPU filter parallelizes queries in a straightforward way, by assigning one insert or lookup operation to each thread.

There have been two previous parallel quotient filter implementations on CPUs. Dutta et al. [11] implement a parallel version of their streaming quotient filter, an AMQ designed for removing duplicates from streams of data. Pandey et al. [29] also implement a multithreaded version of their counting quotient filter, which uses the same structure as their rank-and-select-based quotient filter, described in Chapter 3.3.2. Their implementation depends on per-thread locking that does not scale to the parallelism of a GPU.

3.3 The Quotient Filter

This section describes the standard quotient filter and rank-and-select-based quotient filter and algorithms for serial operations on these data structures.

3.3.1 Standard Quotient Filters

The *quotient filter* [2], which we refer to in this paper as the *standard quotient filter* (SQF), is an AMQ that represents a set S from a universe U by a set of fingerprints. Let $f : U \to [2^p]$ be a hash function that hashes elements of U into p-bit strings. Let $F = f(S) = \{f(x)|x \in S\}$ be the set of hash values of the elements of S. To perform an operation LOOKUP(x), the filter checks whether $f(x) \in F$. The QF stores these fingerprints losslessly. Therefore, all false positives arise from collisions in the hash function, where $f(q) \in F$ for a query $q \notin S$.

To store the set F, divide each of the p-bit hash values into its upper and lower bits. The *quotient*, $f_q(x)$, is comprised of the q high order bits, and the *remainder*, $f_r(x)$, is comprised of the $r = p - q$ low order bits. A QF can be thought of as a hash table with chaining, where the quotients are the hash values and the remainders are the values stored in the table, as shown in the top of Figure 3.1. To insert a fingerprint $f(x)$ into the filter, store the remainder, $f_r(x)$ in the $f_q(x)$-th bucket. Although only r bits per item are stored, this scheme allows the complete fingerprint to be recovered by recombining the remainder value and the bucket number.

An SQF consists of an array A of length 2^q, as in the bottom of Figure 3.1, where each slot contains $r + 3$ bits: the remainder plus 3 metadata bits. To insert an item x into the filter, store $f_r(x)$ in slot $A[f_q(x)]$. If there is already an item in this slot, another item in the filter has the same quotient value. Quotient filters deal with these collisions using linear probing. Thus the remainder for a fingerprint may not always in the *canonical slot*, $A[f_q(x)]$, but it can be found

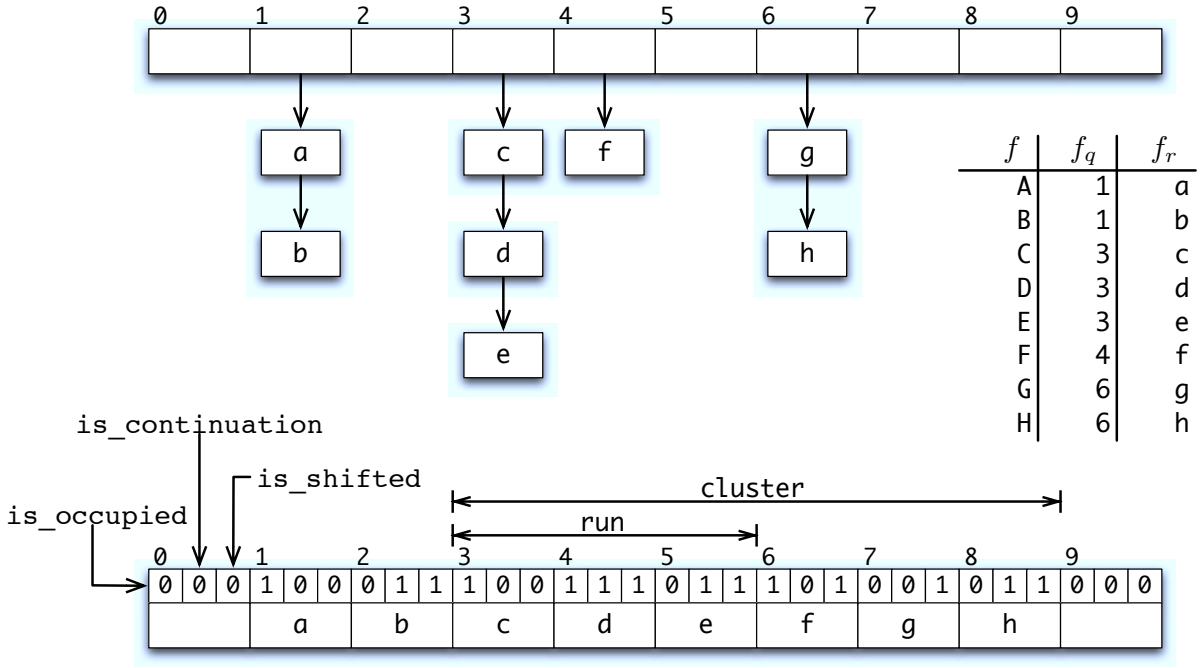

Figure 3.1. An example quotient filter (bottom) with 10 slots, and its representation as a hash table with chaining (top). The filter stores the set of fingerprints $\{A - H\}$. The remainder, f_r, of a fingerprint f is stored in the bucket specified by its quotient, f_q. The quotient filter stores the contents of each bucket in contiguous slots, shifting elements as necessary and using three metadata bits to enable decoding.

nearby. The QF linear-probing algorithm maintains three invariants: (1) Remainders may only be shifted forward (to the right) of their canonical slot. (2) All remainders are stored in sorted order such that if $f(x) < f(y)$, $f_r(x)$ will be stored in a slot before $f_r(y)$. (3) There are no empty slots between an item and its canonical slot. These invariants guarantee that items with the same quotient will be stored in sorted order in contiguous slots, which we call a *run*. A *cluster* is a series of runs with no empty slots between them.

A lookup, insert, or delete operation requires a sequential search within a portion of the filter. Starting at the canonical slot, search to the left to find the beginning of the cluster, then search to the right to find the item's run. The SQF encodes the information needed to determine which run each remainder belongs to using three metadata bits per slot: `is_occupied`, `is_continuation`, and `is_shifted`. Operations maintain good locality, because all reads and writes are in the region around the canonical slot. The performance of all SQF operations is largely dependent on the time spent searching backwards and forwards through the clus-

ters, which is determined by the cluster length. Bender et al. [2] prove that cluster lengths are bounded by a constant in expectation and logarithmically with high probability. Therefore, serial quotient filter operations finish in expected constant time.

As previously mentioned, a QF has a non-zero probability of false positives, meaning a membership query will occasionally return "present" for an item that is not in the set. False positives happen when two keys hash to the same fingerprint—a hard collision. However, as Bender et al. [2] demonstrate, the probability of a hard collision is 2^{-r}; therefore, increasing or decreasing the number of bits in the remainder gives a trade-off between query accuracy and memory usage.

3.3.2 Rank-and-Select-Based Quotient Filters

Pandey et al. [29] designed the RSQF to improve upon the SQF by increasing lookup performance at high load factors and reducing the number of metadata bits.[1] Their filter stores the remainders using the same slot locations and order as the SQF, but it uses a different metadata scheme for locating items within the filter. Figure 3.2 shows the basic structure of the RSQF. The RSQF stores two metadata bits for each remainder slot: `occupieds` and `runEnds`. These bits are stored in separate bit arrays, rather than within the remainder slots themselves, and are accessed via `rank()` and `select()` bit vector operations. To find a run, compute the rank of its `occupied` bit, then select the `runEnd` bit of the same rank. To maintain locality of these operations, the filter is divided into blocks of 64 slots, each with an 8-bit `offset` value to track any overflows from previous blocks. The work required to locate a run is independent of the fill fraction, which means lookup performance does not decrease much as the filter fills up. However, inserts do still require a search to locate the next empty slot and to move items around, so insert performance does decrease with fill fraction, just as in the SQF.

3.4 GPU Standard Quotient Filter Operations

We now describe the GPU implementation of membership queries (lookups), insertions, deletions, and merges. We also devise three parallel methods to construct a quotient filter from a list

[1] They also extend RSQF functionality by storing compact counters in the remainder slots. We chose not to include counters in our GPU implementations in order to focus on how the fundamental differences in the AMQs affect the parallelism we can extract from these data structures.

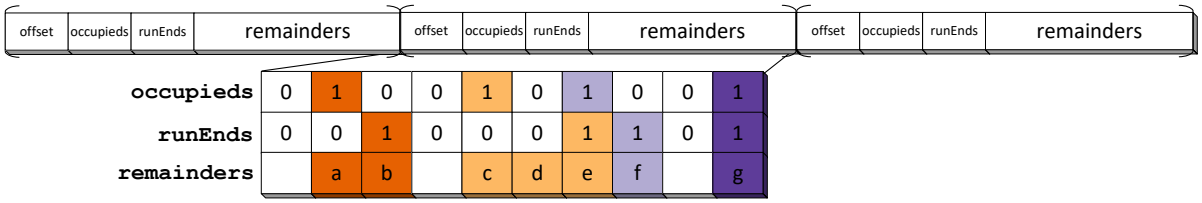

Figure 3.2. Example rank-and-select-based quotient filter with three blocks of ten slots per block. The `occupied` and `runEnds` bit arrays are used to locate items in the filter, and each block has an `offset` value to account for any overflow from previous blocks. The example block (bottom) is shown with the metadata bit arrays oriented above the remainder values to illustrate how these bits are used to determine which run each remainder belongs to. This block has four non-empty runs (denoted by the different colored blocks) with one to three items in each run.

of elements and consider the advantages of each.

The QF stores hashed keys. An important feature of hash values is that they are uniformly distributed, no matter the input, which has pros and cons. On the negative side, uniformity undermines memory locality. On the positive side, uniform distributions favor load balance. Finally, uniformity makes hashes easier to test, since all workloads yield the same behavior, as long as keys are not repeated.

3.4.1 Lookups

To maximize locality between neighboring threads, we first hash and sort the input values. We then assign one membership query per thread and perform a sequential lookup. Pseudocode is shown in Algorithm 5. Performing lookup operations in parallel does not require collision avoidance, because lookups do not modify the QF. Varying cluster lengths results in divergence between threads within warps. However, cluster lengths are small, and therefore each lookup operation will take constant time in expectation and logarithmic time with high probability.

3.4.2 Supercluster Inserts

Assigning one insert per thread can lead to race conditions if different threads try to modify the same slot at the same time. Therefore, we must determine a set of inserts that we can safely perform in parallel. To do this, we identify independent regions of the quotient filter, which we call *superclusters*; we only perform one insert per supercluster at a time. We define a

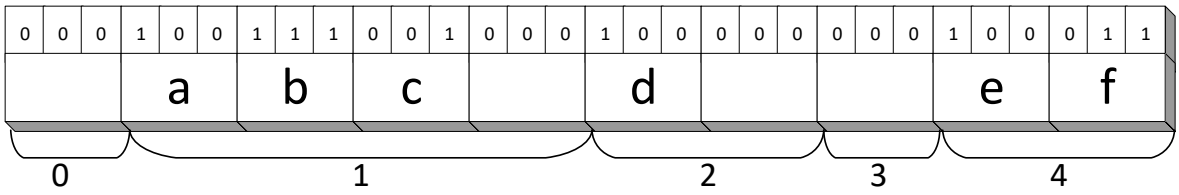

Figure 3.3. Example quotient filter figure with corresponding supercluster labels. Superclusters represent independent regions of the filter, where we may perform inserts in parallel without incurring data races.

supercluster as a region ending with an empty slot. This empty slot allows us to insert a single new element without shifting any elements into another supercluster's space. See Figure 3.3.

In parallel, we mark each slot whose preceding slot is empty with a 1. Then we use the CUB library `DeviceScan` primitive to perform a prefix sum of these bits and label each slot with its supercluster number. The items in the insert queue then bid for exclusive access to their supercluster. We then insert these items, remove them from the queue, and repeat the process until all items have been inserted. Pseudocode is shown in Algorithm 7.

The parallelism of this method is significantly constrained by the number of superclusters in the filter, and as the filter gets fuller, there are fewer superclusters. Additionally, like the lookup method, this insert implementation suffers from warp divergence and lack of memory reuse between threads. However, because the input values are hashed, the distribution of items between superclusters should be roughly uniform, resulting in good load balancing.

3.4.3 Bulk Build

Consider inserting a batch of items into an empty QF. We will do so by computing every item's final location in parallel and then scattering them to their locations.

We begin by computing the fingerprints, sorting them, and splitting them into quotient and remainder values. To compute the location of each item, recall from Chapter 3.3.1 that an element is located either in its canonical slot (shift = 0) or shifted to the right (shift $>$ 0). Figure 3.4 illustrates how items from runs with lower quotient values can shift items in later runs. The shift amount for the first element in a run is the shift amount for the first element in the previous run plus the number of items in the previous run minus the distance between the

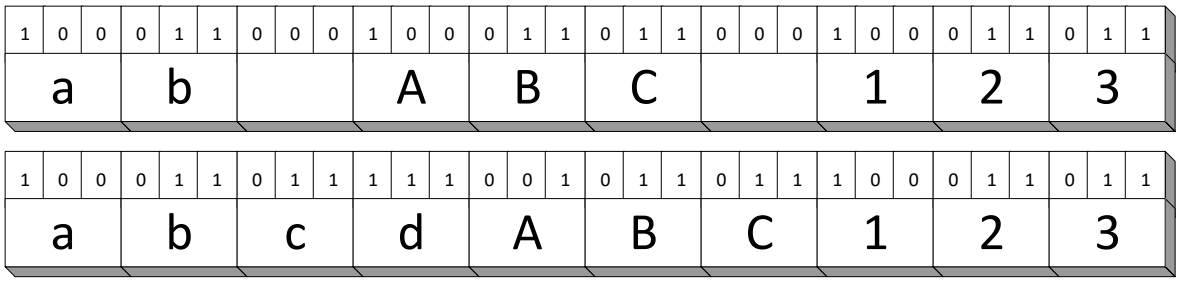

Figure 3.4. Example quotient filter arrays showing the interdependence of item locations in different runs. ABC is a run of elements with $quotient = 3$. With the addition of c and d to slot 0, ABC must be shifted right one slot.

canonical slots of the two runs. Essentially, we keep a running total of underflow/overflow.

Astute parallel programmers might immediately think "prefix-sum". But recall the resulting shift value must be non-negative, so we must saturate the sum at each step so that the resulting shift never goes below zero. Alternatively, when directly computing the location of elements (as opposed to computing their shift values), we could consider a prefix-sum operator of $\max(\textit{value}_{(i-1)} + 1, \textit{value}_i)$. Neither of these operations is associative, thus we cannot use any existing GPU methods that implement prefix-sum, all of which require associative operators. However, both of these operators have an important property that allows us to extract parallelism: certain inputs, including those we see in quotient filter construction, break the dependency chain between output items. At each point where the saturation to zero happens in the prefix sum formulation, or where the $\max(\textit{value}_{(i-1)} + 1, \textit{value}_i)$ operator outputs $\textit{value}_i$, the contribution from the scan of all preceding values to the items that follow is zero. In this way, the problem can be thought of as a segmented scan in which the segment divisions are initially unknown, and we can parallelize over segments.

We explored three methods for bulk QF builds on the GPU, each of which approaches the non-associative scan problem differently:

—*Parallel merging* (Chapter 3.4.3.1) begins with one segment for each unique quotient in the dataset, then iteratively merges pairs of segments together, checking the saturation condition as each pair merges and only sending the output of the scan from the left segment as the input to the right if the saturation condition is not met.

—*Sequential shifting* (Chapter 3.4.3.2) applies the operation to every pair of neighboring runs in each iteration, checking the saturation condition, and iterating until the scan has been carried through the end of the longest independent cluster.

—*Segmented layouts* (Chapter 3.4.3.3) assumes that every segment of $\log(n)$ items is independent and computes the scan serially within these segments. In each iteration, each segment sends the partial scan for its last item to become the initial value for the segment to its right. Because the quotient values are the result of a hash function, this process converges after a small number of iterations.

3.4.3.1 Bulk Build Via Parallel Merging

This first implementation of bulk build uses an iterative merging process, which finishes after $\mathcal{O}(q) = \mathcal{O}(\log(n))$ iterations, or more precisely, $\log(\textit{number\_used\_quotients})$ iterations. First, we compute the items' unshifted locations with a segmented scan. Next, we label the segments in parallel by checking $\textit{quotient}[\textit{idx}] \neq \textit{quotient}[\textit{idx}-1]$ then performing a prefix sum. Initially, items will only be grouped with the other items in their run, as shown in iteration 0 of Figure 3.5, but these segments will grow as we run our merging algorithm.

Each iteration of the merging algorithm, shown in Figure 3.5, consists of two steps. First, for all pairs of segments, we compare the last element in the left-hand segment and the first element in the right-hand segment and compute the overflow or underflow. Second, for all elements, we compute and apply the shift using the overflow/underflow for the segment. We account for any empty filter slots and prevent extraneous shifting by storing negative shift values in a credits array. The pseudocode for this build method is shown in Algorithm 8.

3.4.3.2 Bulk Build Via Sequential Shifting of Runs

In our second method, shown in Figure 3.6, we compute unshifted locations and label the segments, just as we did for the parallel merging bulk build. We then shift the filter elements iteratively; however, instead of combining the runs into larger segments, we launch one thread per run in each round to determine whether the run needs to be shifted to avoid overlap with the previous run. Threads set a global flag each time they perform a shift, which we check (then clear) after each iteration to determine whether or not to launch the kernel again. When a kernel finishes without shifting any elements, the algorithm is finished. Pseudocode is in Algorithm 9.

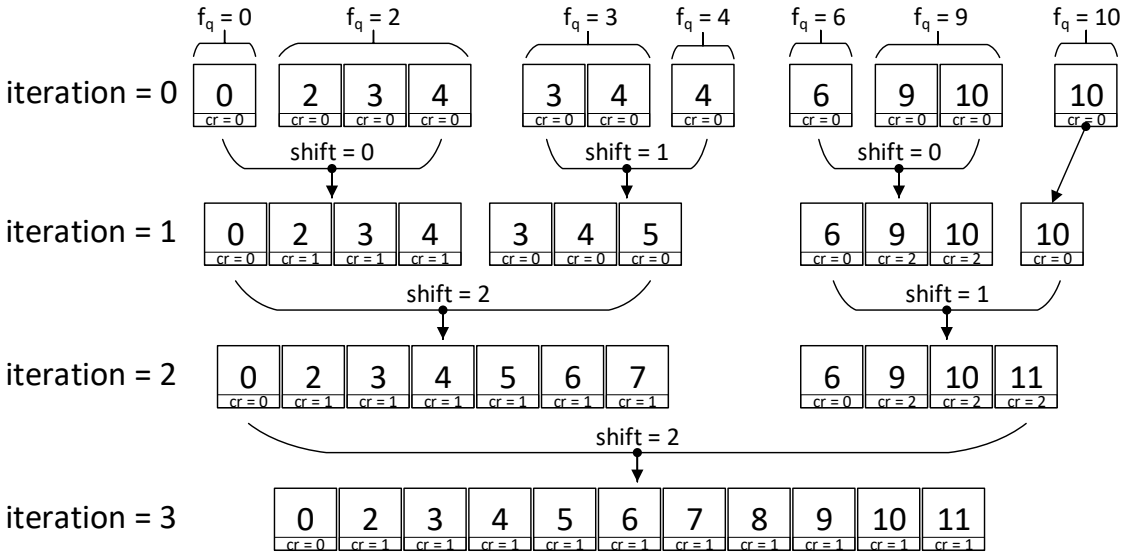

Figure 3.5. Diagram of our parallel merging bulk build algorithm. At the start, items are grouped into segments according to their quotient value (canonical slot), f_q, then in each iteration, neighboring segments are pairwise-merged and any necessary shifts are applied. The large number in each box indicates the slot the associated remainder value will occupy in the final filter construction, and the number in the lower part of the box (`cr`) denotes any credits from empty slots preceding the current slot.

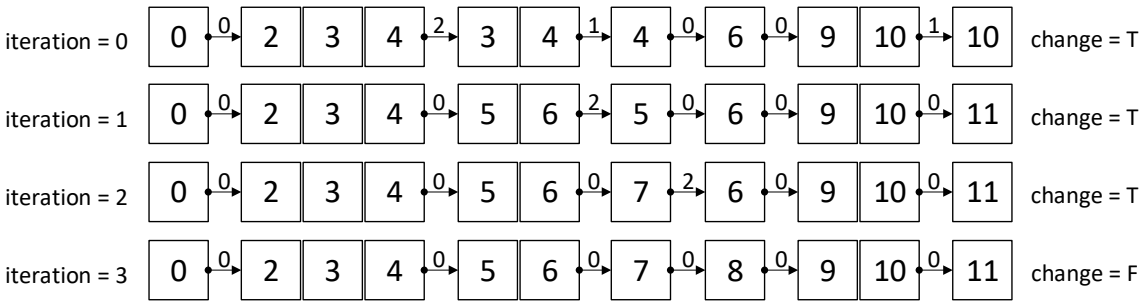

Figure 3.6. Diagram of our bulk build method using sequential shifting of runs. In each iteration, we launch one thread per run to check for overlap between its run and the previous run and shift its run if necessary. As in Figure 3.5, the values in the boxes are the slots the items will occupy in the quotient filter. The values above the arrows indicate the amount the next run must be shifted to avoid overlap. When all of these shift values are 0, the process stops.

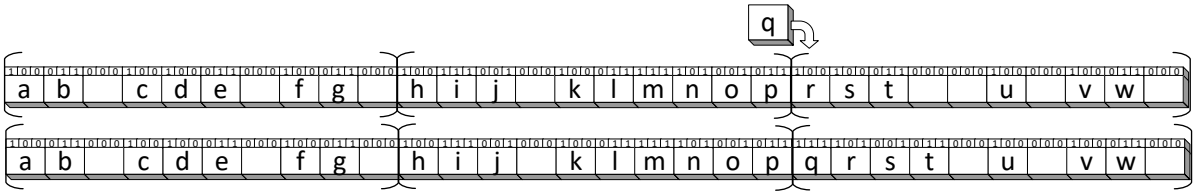

Figure 3.7. Diagram of segmented layouts bulk build for a small example quotient filter. This filter contains three segments, and the layout of each segment is computed in parallel, then checked for overflows. In the first iteration, item q is an overflow item, and gets bumped into the next segment in the second iteration.

3.4.3.3 Segmented Layouts

Our third bulk build method computes shifts in segments of the filter itself, exploiting the fact that the input is hashed, and therefore, items are distributed approximately evenly throughout the filter. For this method, we partition the quotient filter into segments of length $\log(\textit{numSlots}) = \log(2^q) = q$. Each segment has all items whose quotients fall within the segment and an initial shift value for the segment. We launch one thread per segment to lay out all of the items in its segment given the initial shift and output an overflow value. These overflow values are then passed as initial shifts for the next segment, as shown in Figure 3.7, and the process repeats until no new overflows are generated. Because the quotient values are the result of a hash function, this process converges after a small number of iterations. Pseudocode is in Algorithm 10.

Comparison of Build Methods We have now devised four different ways (including supercluster inserts) to construct a QF from scratch. We evaluate these methods experimentally in Chapter 3.7.3, but intuitively, when is each one most appropriate? When a filter is empty, there are many available superclusters, so supercluster inserts should work well. The sequential shifting method would also be likely to work better for filters that are less full, because clusters will be shorter, leading to fewer shifts, and therefore fewer iterations. The parallel merging build requires a constant number of iterations, so it will be the most efficient for building very full filters. The segmented layouts build is likely to perform better for emptier filters, but overall we would expect its running time to increase only moderately as the fill fraction increases. Because the segment length is always q, the segmented layout method also requires the same amount of work in each round, independent of the filter fullness.

We now compare the parallel complexity of these bulk build algorithms. For the parallel merging and sequential shifting builds, preprocessing involves a sort and two prefix sums. For parallel merging, the merging process continues for $\mathcal{O}(\log(n))$ iterations, where each iteration uses a constant number of steps. So the makespan of the parallel merging build is $\mathcal{O}(sort(n) + \log(n))$.

For the sequential shifting build, the shifting process continues until the shifts have been carried through all clusters. This means the number of iterations is bounded by the number of runs in the longest cluster. As Bender et al. show [2], the largest cluster in a QF has size $k = (1+\epsilon)\frac{\ln(n)}{\alpha - \ln(\alpha) - 1}$ with high probability. For a reasonable QF fill fraction, we can approximate this as $\mathcal{O}(\log(n))$. Within each iteration, there are a constant number of steps. Therefore, the makespan of the sequential shifting build is $\mathcal{O}(sort(n) + \log(n))$ with high probability.

Preprocessing for the segmented layouts build only requires a sort. The layout operation itself requires a sequential iteration over the $q = \mathcal{O}(\log(n))$ slots in the segment. In the worst case, the number of iterations is bounded by the maximum shift for any one item in the filter. This shift is bounded by the maximum cluster length, which is $\mathcal{O}(\log(n))$ with high probability, so the complexity of this build method is $\mathcal{O}(sort(n) + \log^2(n))$ with high probability.

Generalization to Other Non-Associative Operators The strategies we used to approach this problem could be used to compute a scan on other non-associative operators – in particular, operators that include a saturation condition. However, the performance of the sequential shifting and segmented layouts methods relies on the ability to break the chain of dependencies. Without independent segments, the performance of these algorithms will be the same as that of a serial algorithm. Fortunately, the QF provides us with ample independent segments, because hashing distributes items uniformly across the entire space of the filter. On the other hand, the parallel merging algorithm requires an operator that is associative between the breaks in the dependency chain. It does not extract parallelism from these discontinuities, but rather from the fact that these specific operators are associative for all items between the saturation locations.

3.4.4 Supercluster Deletes

Deletes use an algorithm similar to the supercluster inserts described in Chapter 3.4.2. We divide the filter into independent supercluster regions similar to those used in the insert method,

but with a modification: we require that the first slot in a supercluster be occupied and unshifted. This means the slot is actually the head of a cluster. We know that the value in this slot, and any shifted slots to its immediate right, will not be affected by any deletes to the left because the item is in its canonical slot. This also prevents the supercluster from being comprised of only empty slots (with no items to delete). Similar to supercluster inserts, we perform a bidding process to choose which items to delete while avoiding collisions. We then delete one item at a time per supercluster, shifting items left and modifying metadata bits as needed. When all threads have finished, we remove successfully deleted items from the queue and repeat. The pseudocode for this operation is shown in Algorithm 11.

3.4.5 Merging Filters

Merging two QFs allows us to use one of our bulk build operations to add a new batch of items to the filter. This is a rebuild of the filter, but without the need to access or rehash the items already stored in the filter. Merging is helpful for some filter applications, e.g., combining datasets stored by different nodes in a distributed system. The first step in merging two QFs is to extract the original fingerprint values. We do this in parallel by assigning one thread to each slot, using the metadata bits to determine its quotient. We scatter these fingerprints to an array, and compact out all of the empty slots. We now have two sorted arrays: one for each of the original filters. We merge these arrays using the GPU merge path algorithm by Green et al. [13]. This leaves us with one sorted array of fingerprints, which we can now input to one of our three bulk build algorithms to construct the QF. Pseudocode is in Algorithm 12.

3.5 GPU Rank-and-Select QF Operations

In this section, we describe the algorithms we devised for querying and modifying the rank-and-select-based quotient filter (RSQF) on the GPU.

3.5.1 Lookups

For the RSQF, we parallelize lookups for the GPU in a similar fashion as we do for the SQF: hash and sort all inputs, then assign one query per thread and have each thread perform the same operation as in the serial case. The pseudocode for RSQF lookups is Algorithm 6. Again, because lookup operations do not modify the data structure, this simple parallelization works

without the addition of any collision avoidance schemes. The sorting step increases memory locality between threads, as in our SQF lookups, but here the benefit is much greater, because metadata values are shared across all slots in an RSQF block, rather than scattered amongst the remainder values.

3.5.2 Inserts

For RSQF inserts, our general strategy was to parallelize over the blocks of the filter and perform inserts in batches. Because each block has an offset value to account for any spillover from previous blocks, the block holds all of the necessary information to insert an item in any of the 64 slots within the block, assuming it does not overflow to the next block. However, some items, particularly when the filter reaches higher fill fractions, will need to overflow to the next block. To deal with inserts that overflow into other blocks while still avoiding race conditions, we allow threads to modify more blocks as the filter gets fuller. To accomplish this, we partition the filter into insert regions, and assign one thread to each region. This is similar to the partitioning for multi-threaded inserts devised by Pandey [29], but because the GPU has many more threads than a CPU, our implementation uses smaller regions than theirs. Initially, regions are one block, and as more items have been inserted, the regions increase in size (and, consequently, decrease in number). In our implementation, we increase the region size by one block every 16 iterations.

The entire insert operation proceeds as follows: Before inserting a batch of items, we hash the inputs, then sort the fingerprints, and divide them into queues based on their block number. We then launch one CUDA thread per region to perform the inserts, using the same basic algorithm as the CPU implementation described in 3.3.2. If an insert operation would require a thread to modify a block that is not in its assigned region (in the case of overflow to the next block), the operation halts. When an insert is completed or halted, the thread sets a flag if it still has items in its queue, to indicate that the insert kernel should be launched again. The pseudocode for this algorithm is shown in Algorithm 14.

The maximum amount of parallelism we can extract using this method is constrained by the number of blocks in the filter. Work balancing is dependent on the distribution of the data: if many of the items hash to the same region of the filter, the threads for other regions will finish

their inserts and have no more work to do while the busy thread is still working. Because the input data is hashed, assuming a good hash function, the most likely cause for an unbalanced workload is repeated items.

3.5.3 Bulk Build, Deletes, Merging Filters

We did not implement bulk build, delete, or merge operations for the RSQF, but these operations can be performed using a straightforward extension of the methods we used in the SQF algorithms. The final slot numbers for all items are the same for the SQF and RSQF, so we can use the algorithms described in Chapter 3.4.3 to compute the locations of all items, with slight modifications in writing the values to the filter to account for the different memory layout and associated metadata values. Merges can also be implemented analogously to the operation in Chapter 3.4.5 by assigning one thread per slot to extract fingerprints into an array, then merging the arrays and rebuilding the new filter. Finally, just as superclusters can be used in both SQF inserts and deletes, our RSQF insert strategy of breaking the filter up by into small regions and assigning one region to each thread can also be applied to RSQF deletes.

3.6 Design Decisions and Trade-Offs

In this section we justify some of the many design decisions we made in adapting the SQF and RSQF to the GPU.

Remainders Divisible by 8 and `char` Containers Quotient filters are designed to be flexible in the number of bits stored per item in order to allow the programmer to choose the best trade-off between memory and error for their application. However, this variability means that, because the elements are stored contiguously in memory, a single slot may be split between bytes (and even cache lines). For arbitrary values of r, this opens up the possibility of memory conflicts, even when we ensure threads are modifying different slots. To simplify this issue, we chose to only use SQFs where the number of bits per slot is divisible by 8. This gives us fewer options in the trade-off between size and false positive rate, but it simplifies our filter operations and increases the amount of parallelism we can extract from the problem. Similarly, we chose to store each SQF slot in one or more `chars`, rather than fitting one or more remainders into a single `int`, so that we are able to write to two neighboring slots without worrying about write

hazards.

Duplicates For datasets with many reoccurring values, deduplication may be essential to speed up membership queries and prevent the filter from over-filling. For the bulk build algorithms, because all items are being inserted at the same time, deduplication is not an inherent part of the algorithm, as it is for incremental inserts; therefore, we give an option for deduplication to be switched off or on, to allow flexibility for different applications.

One Query Per Thread We chose to use only one item per thread for each QF lookup query. An alternative approach would be to launch one query per warp and have threads search cooperatively within clusters to locate the items. However, this would not be very efficient because: (1) the average cluster length is constant, as described in Chapter 3.3.1, and (2) threads must also keep track of metadata values to compute the canonical slot for each value, which would be much more complicated to resolve cooperatively.

RSQF Insert Region Size For inserts into an RSQF, we had a few different options for the granularity. We could have identified the smallest independent regions in the RSQF, similar to the superclusters in the QF. However, the blocked structure of the RSQF means that operations within the same block could lead to race conditions when modifying the block-wide `occupieds`, `runEnds`, and `offset` values. Alternatively, we could have based the size of the regions on the filter's fill fraction, to account for the increased likelihood of interblock overflows as the fill fraction increases. Both of these alternatives would also require a system of tracking which items-to-be-inserted belong to each of the variable-sized regions. We decided to take the simpler approach, where we could perform an initial sort and create stable queues of items to be inserted into each block.

Number of Iterations Between Each Extension of Insert Regions We decided to increase the size of insert regions every 16 iterations based on empirical evidence from our experiments. This seemed to be the interval that worked best for building a filter from scratch. For incremental updates, the most efficient interval would likely vary based on the fill fraction, which would require additional tuning and work to track the fill fraction.

3.7 Results

We evaluate our GPU SQF and RSQF implementations using synthetic datasets of 32-bit keys, generated using the Mersenne Twister pseudorandom number generator. Because the values are hashed before any QF operations are performed, the distribution of the input data should not affect performance, and random data should be sufficient to estimate AMQ performance in real-world applications.

In all experiments, we used QFs with $q = 23 \rightarrow 2^{23}$ slots and $r = 5$, or an error rate of $\epsilon \approx 0.03125$. We compare our GPU SQF and RSQF with a variety of other AMQs: Costa et al.'s BloomGPU filter [7], Pandey et al.'s multithreaded counting quotient filter (CQF) [29], Arash Partow's Bloom filter [30], and our own CPU SQF implementation, which uses the serial operations described in Chapter 3.3. We also modified the BloomGPU filter to create a version ("BloomGPU 1-bit") using only one bit per element of the filter bit array, rather than an entire byte. This required using atomic bitwise operations, which were not yet supported by CUDA at the time that Costa created BloomGPU. The invariant in our comparisons is false-positive rate. Because we also inserted the same number of items for all data structures and the BF error rate increases as more bits are set, we increase the BF size as the comparable QF fills up in order to maintain a similar false positive rate in both filters. To achieve a balance between error rate and accuracy, we chose to use 5 hash functions in all BFs.

All GPU experiments were run on a Linux workstation with 2 $\times$ 2.53 GHZ 4-core Intel E5630 Xeon CPUs, 12 GB of main memory, and two GPUs: an NVIDIA Tesla K40c with 12 GB on-board memory and an NVIDIA GeForce GTX 1080 with 8 GB of on-board memory. We ran all experiments on each GPU separately and have noted any significant differences in performance results between the two architectures in our discussion. All source files were compiled with CUDA 8.0. CPU experiments were run on a Linux workstation with one 3.7 GHz 4-core Intel E3-1280V5 CPU. We chose to use a different workstation for our CPU experiments to give a fair comparison to Pandey's multithreaded CQF, which utilizes recently-added x86 instructions to speed up the `rank()` and `select()` operations [29]. We used 4 threads for all multithreaded CQF experiments.

Summary of Results Overall, we find that our SQF outperforms BloomGPU on lookups and initial filter construction, while the BF achieves higher throughput for incremental insert operations. The RSQF achieves a 2–3x speedup on lookups compared to BloomGPU, but has lower incremental insert throughput than the SQF. We find that the BloomGPU 1-bit modification has only a modest effect on the filter performance, so we use this version as our primary reference for comparison. We also discover that the segmented layouts method is generally our fastest QF construction method.

3.7.1 Lookups

Figures 3.8 and 3.9 show the performance difference for membership queries using BloomGPU and our GPU QFs on the Tesla K40c and GTX 1080, respectively. For a fill fraction $\alpha < 0.8$, our SQF achieves higher throughput than BloomGPU. Both the SQF and RSQF show an initial increase in throughput as the fill fraction increases. This is because at the lower fill fractions, the batch size is not yet big enough to fill the entire GPU. SQF query throughput is highly dependent on fill fraction, because each lookup requires reading an entire cluster of elements, and as the filter gets full, the average cluster length increases. Bender et al. recommend that a QF remain $\leq 75\%$ full for this reason. Our RSQF, on the other hand, maintains a similar throughput across all fill fractions, because the `rank` and `select` operations require the same amount of compute across all fill fractions. The only linear searching the RSQF performs in a lookup is within the item's run, which, at high fill fractions, is much smaller than a cluster. As a result of this property, RSQF lookup throughput is higher than the standard QF for $\alpha \geq 0.5$. BloomGPU filter throughput also remains constant because the filter performs the same number of reads per query for all fill fractions.

Comparing Figures 3.8 and 3.9, we see that all filters' performance improves on the GeForce GTX 1080 with the newer microarchitecture and increased memory bandwidth. One notable difference is that the BloomGPU 1-bit achieves very high throughputs at low fill fractions on the GTX 1080. This is likely because the smallest Bloom filters fit into the larger cache in the GTX 1080, and we resize the Bloom filters to maintain equivalent false positive rates.

All GPU filters show a large performance increase as the batch size grows (Figure 3.10). This illustrates the importance of providing sufficient work to keep all GPU compute units

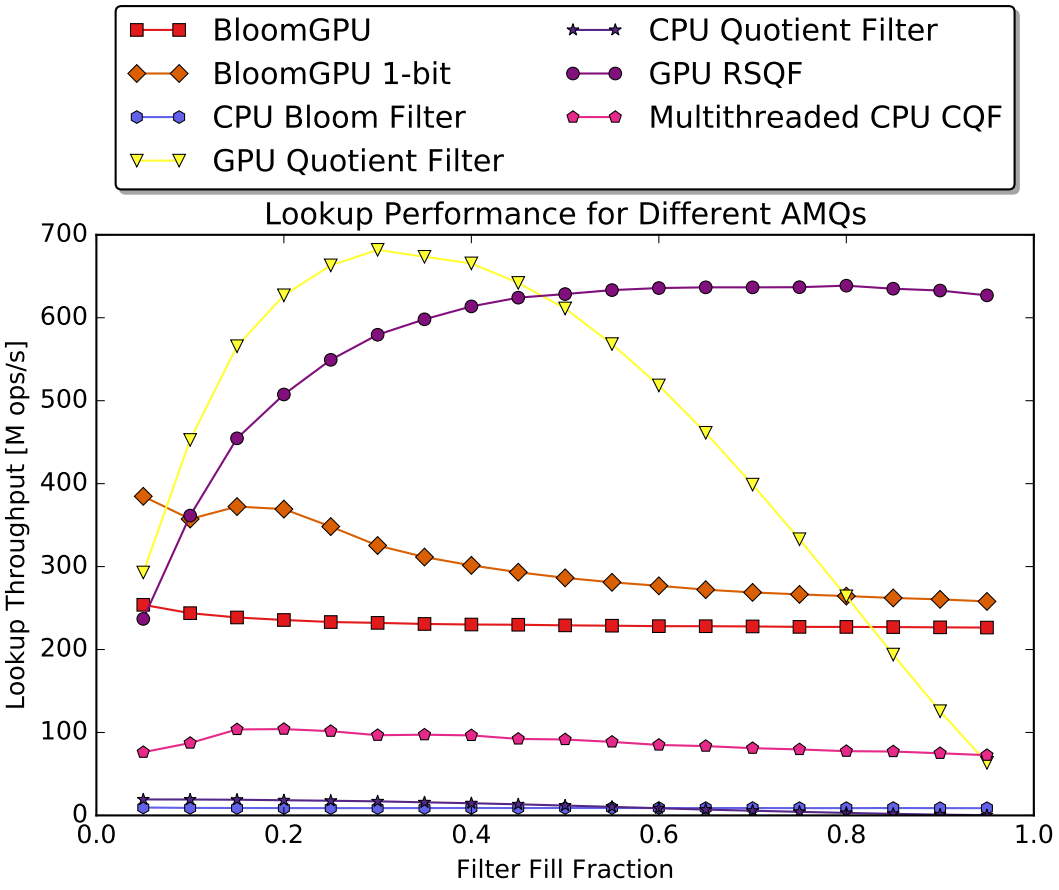

Figure 3.8. Lookup performance on NVIDIA Tesla K40c for different AMQs with varying fill rates. The batch size is all items in the filter. Our quotient filters achieve higher throughput than BloomGPU or the CPU filters. For higher filter fill fractions, the RSQF achieves better performance than the SQF.

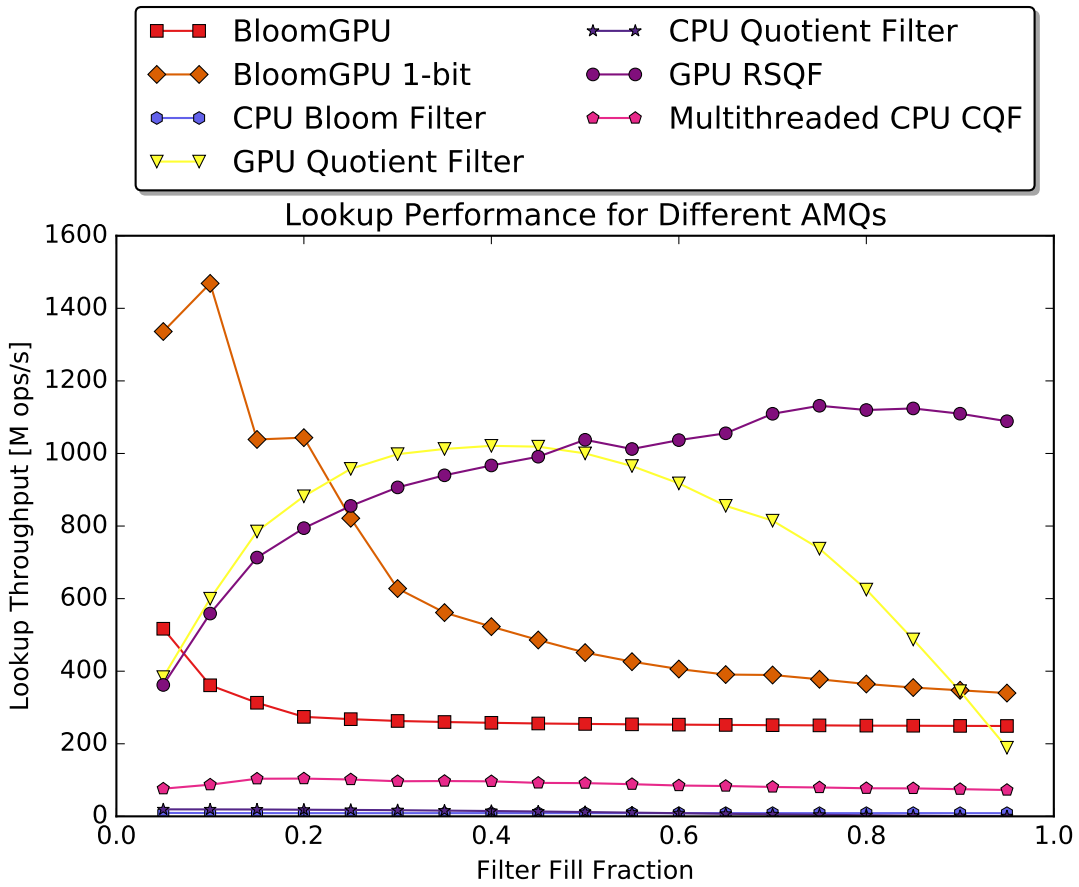

Figure 3.9. Lookup performance on NVIDIA GeForce GTX 1080 for different AMQs with varying fill rates. The batch size is all items in the filter. Our quotient filters achieve higher throughput than BloomGPU or the CPU filters. For higher filter fill fractions, the RSQF achieves better performance than the SQF. Our filters also receive a significant boost in lookup performance with the newer architecture.

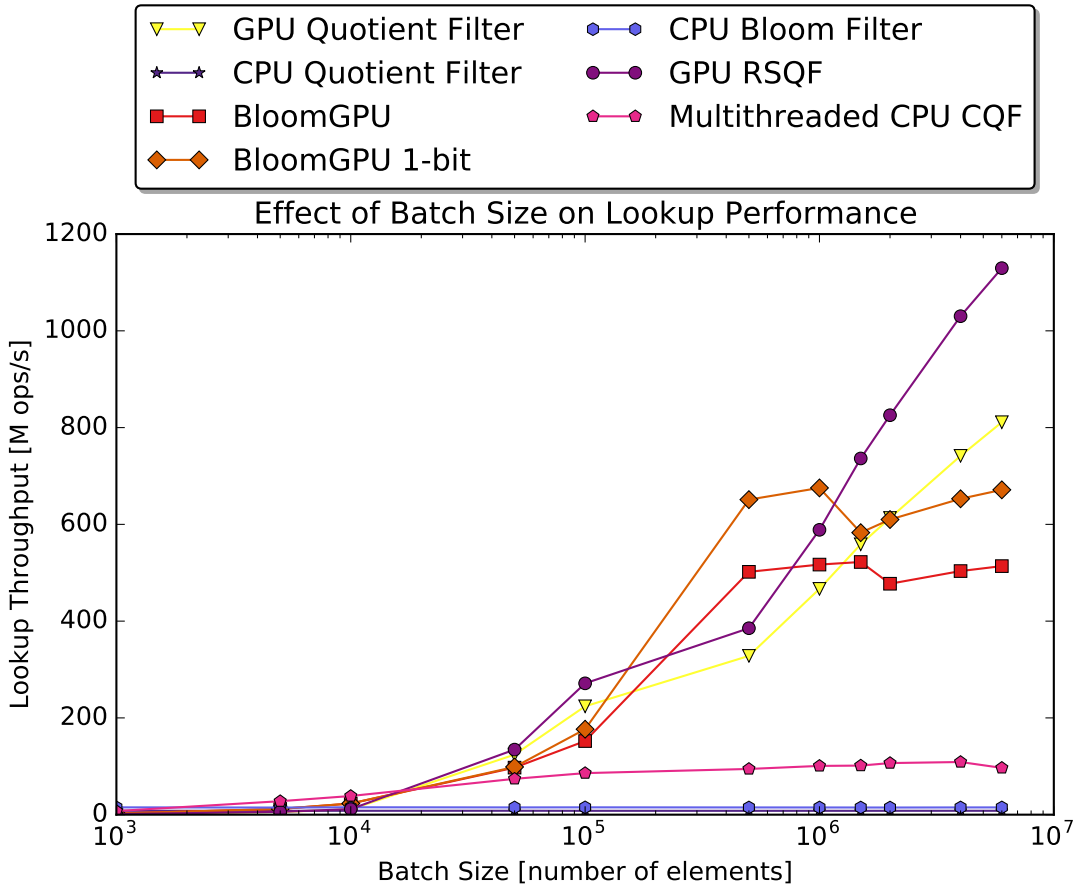

Figure 3.10. Lookup performance on NVIDIA GeForce GTX 1080 for different AMQs with varying batch sizes, with an initial fill fraction of $\alpha = 0.7$. All GPU filters perform better with larger batch sizes.

busy. At around 10^6 items per batch, BloomGPU throughput levels out. At this point, the performance is memory-bound for the BF, but not the QF, due to the greater locality of the QF operations.

3.7.2 Inserts and Deletes

Figure 3.11 shows the change in insert and delete throughput for a constant batch size (100000 items) as a function of filter fullness. Performance for supercluster inserts and deletes decreases as the filter fills and the number of superclusters decreases. Also, the latency for each operation increases as clusters grow and the GPU must search through a longer section of the filter to locate the correct slot. This reinforces the rule of thumb of maintaining a filter fullness of

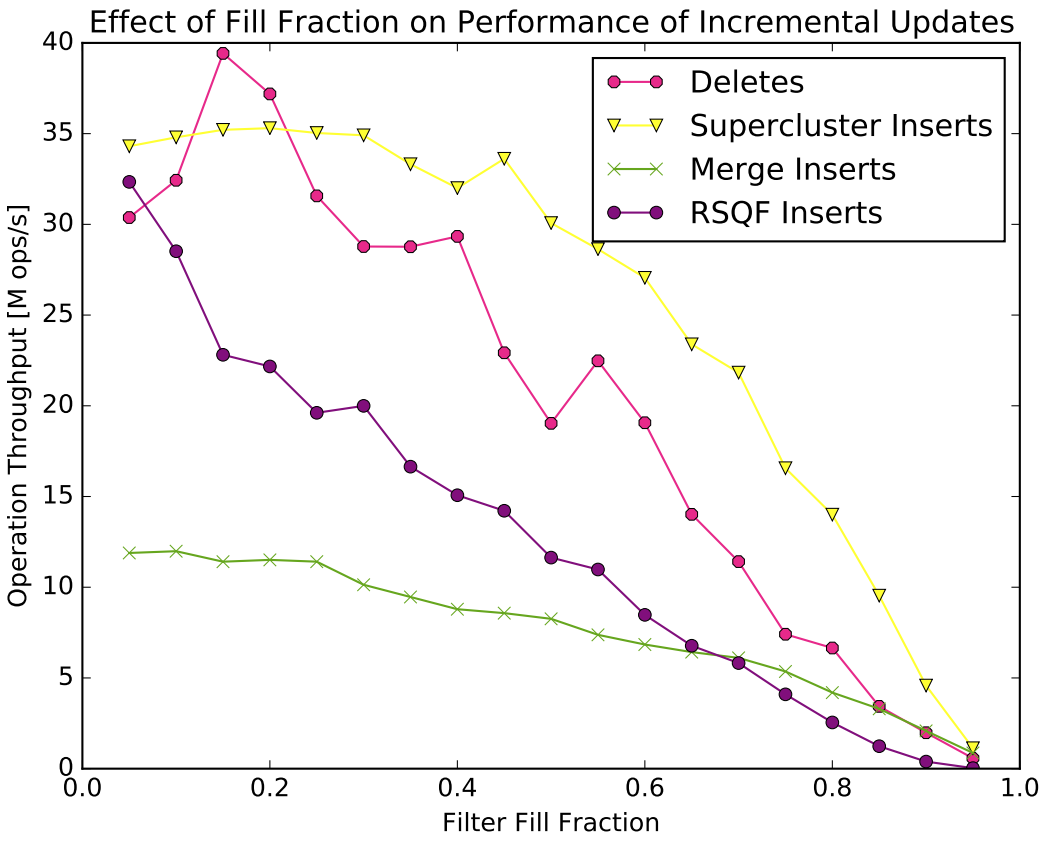

Figure 3.11. Insert and delete throughputs on the NVIDIA GeForce GTX 1080 for the GPU quotient filters using a constant batch size of 100000. Throughput for all operations decreases as the filter fills.

$\leq 75\%$.

We find that our RSQF inserts achieve lower throughput than supercluster inserts, and that RSQF insert throughput decreases as the filter gets fuller. This is because RSQF inserts shift items to make room for new ones, so much of the compute time is spent searching for empty slots and rearranging items, and as the filter fills, these empty slots become more difficult to locate. Figure 3.11 also shows a performance comparison for supercluster inserts versus the merge-and-rebuild (merge inserts) approach. For filters below $\approx 80\%$ full, supercluster inserts have a 2x speedup over rebuilding, and even at 95% full, supercluster inserts still achieve a slightly higher throughput.

All AMQs show a performance increase as the batch size grows (Figure 3.12); however,

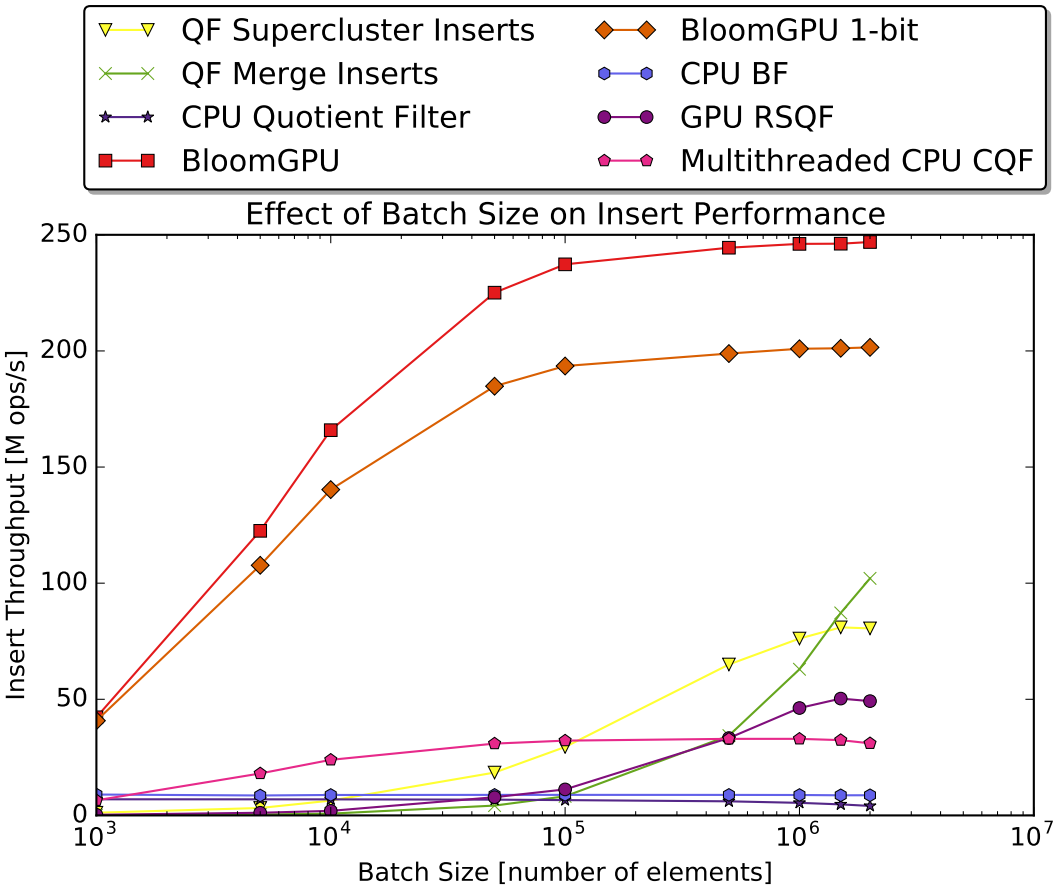

Figure 3.12. Insert performance on the NVIDIA GeForce GTX 1080 for different AMQs with varying batch sizes. The Bloom filters achieve better insert performance, due to the simplicity and lack of interdependence between operations. For very large batch sizes over 2 million items, merging and rebuilding the filter is faster than performing incremental insert operations.

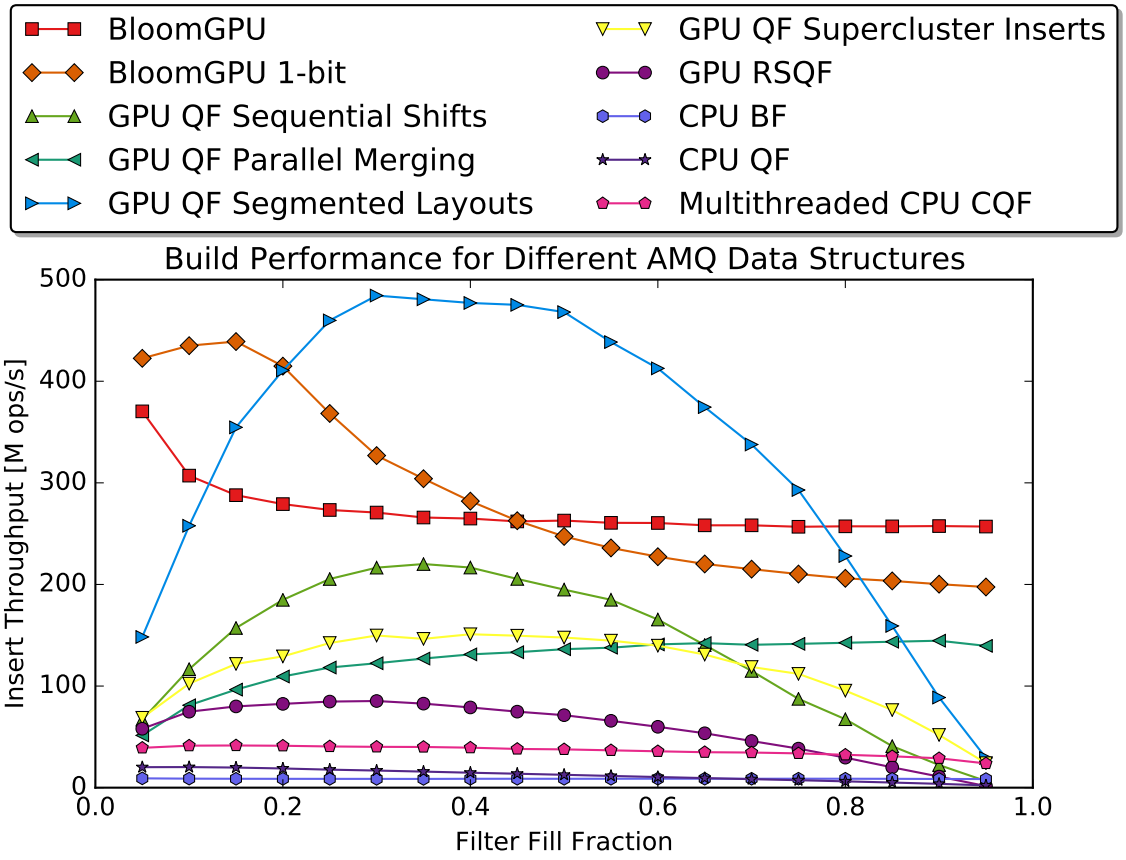

Figure 3.13. Filter build performance on NVIDIA Tesla K40c for different AMQ data structures with varying fill rates. Overall, the segmented layouts method performs best for most fill fractions, though the parallel merging method is better for very full filters.

both the overall performance and performance improvement are much lower for the QFs. This is likely because the available parallelism is restricted to one insert per supercluster for the SQF, and one insert per block region for the RSQF. We can also see that for smaller batch sizes, supercluster inserts are faster than merge inserts, but for batch sizes of ≥ 2 million items, it is actually faster to extract the quotients and rebuild the filter with the new values.

3.7.3 Comparing Filter Build Methods

Figures 3.13 and 3.14 show the build throughput for all AMQs on the Tesla K40c and GTX 1080. As with lookups, the BloomGPU 1-bit inserts achieve high throughputs at low fill fractions, likely because these smaller filters fit in the GTX 1080's larger L2 cache.

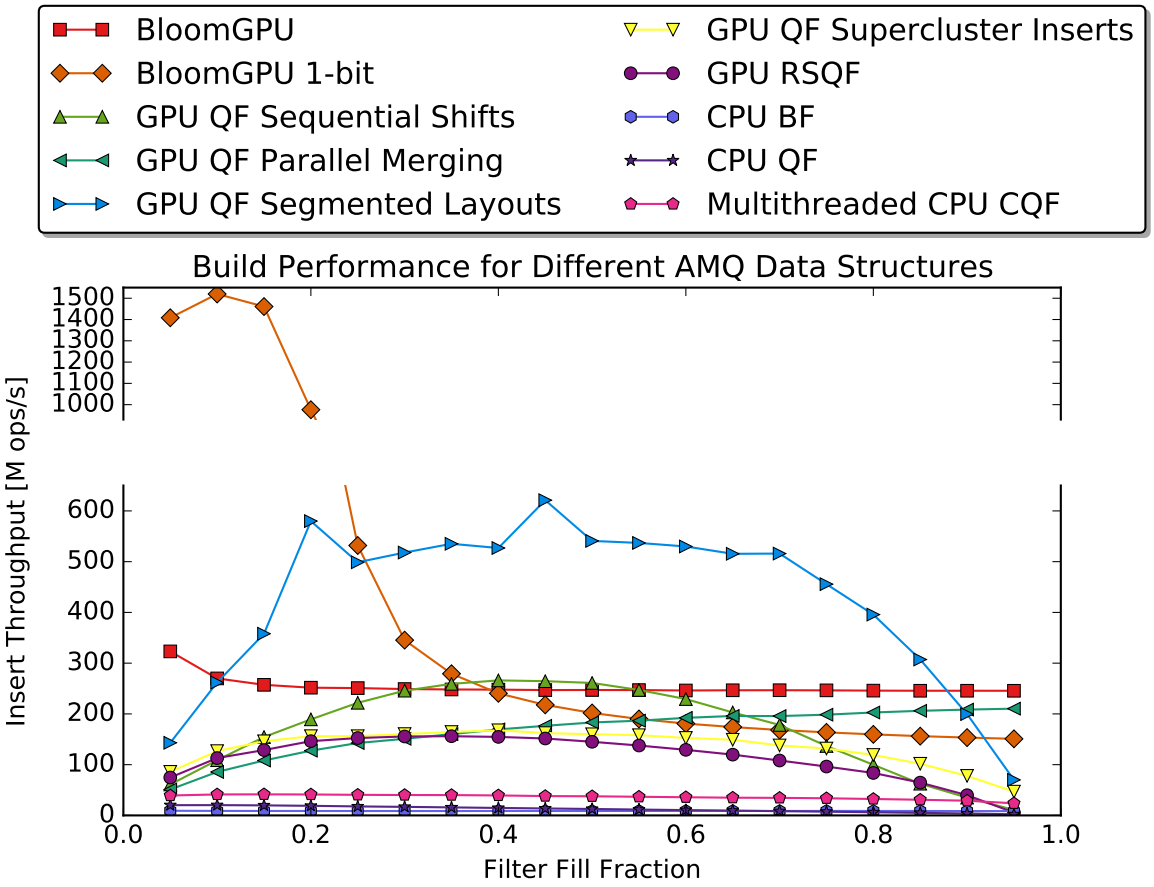

Figure 3.14. Filter build performance on NVIDIA GeForce GTX 1080 for different AMQ data structures with varying fill rates. Comparative performance is similar to the results on the Tesla K40c (Figure 3.13), but the 1-bit Bloom filter sees a large increase in throughput for very small filters because these fit into the larger cache of the GTX 1080.

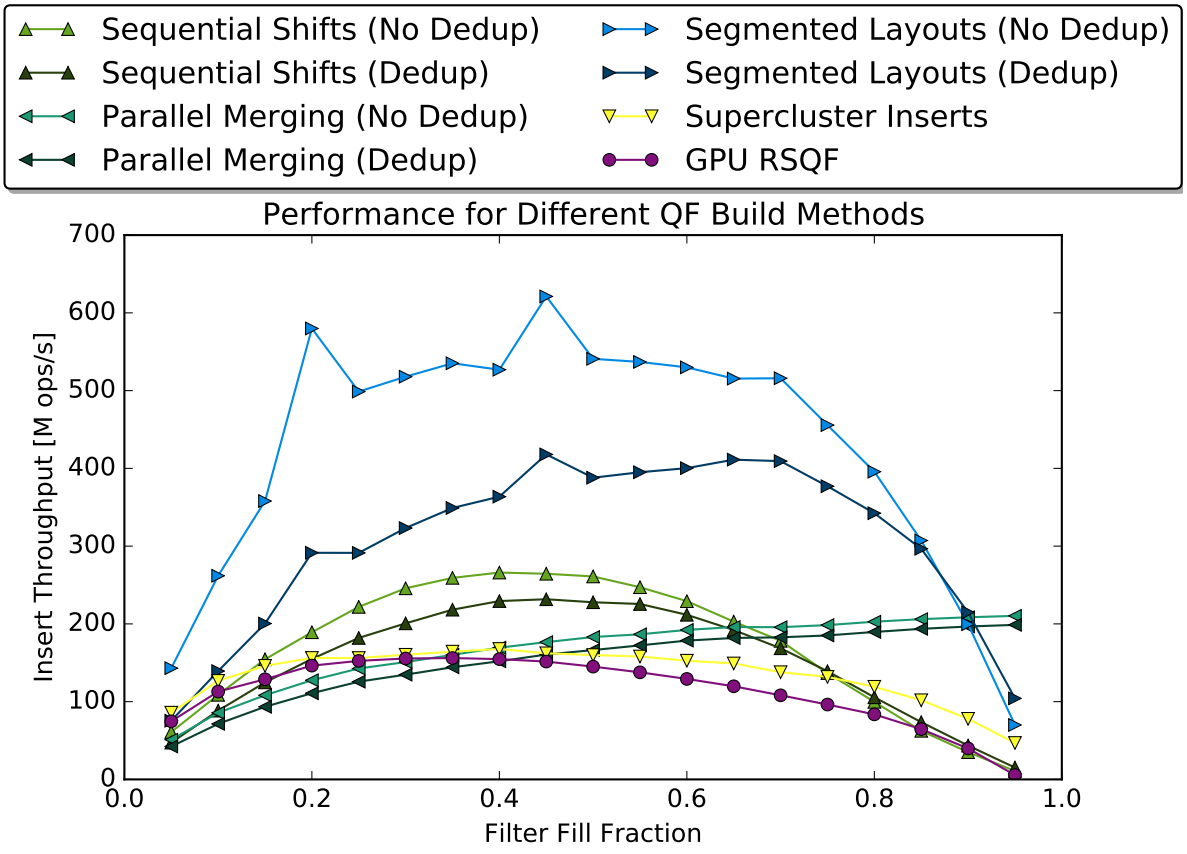

Figure 3.15. GPU quotient filter build performance on NVIDIA GeForce GTX 1080 for all build methods at varying fill rates, with and without a deduplication step. Deduplication does have a modest performance cost when constructing the filter, but may be worthwhile to reduce the size and/or fullness of the resulting filter.

As with lookups, the original 1-byte BloomGPU insert performance does not vary with changing fill fraction; however, the 1-bit implementation does show a steady decrease in throughput as the fill fraction increases. For fill fractions $\alpha > 0.4$, the 1-byte version achieves higher throughput than the 1-bit version. This is likely due to the computational cost of atomic operations required for 1-bit Bloom filter inserts. All QF build methods have an initial increase in performance before throughput either decreases or levels off. This increase is probably because there is not enough work to fill the GPU for very low fill fractions. After this initial ramping up, we see different behavior for each of the build methods:

- Throughput for the parallel merging build increases monotonically with fill rate, because the number of iterations for this method is dependent only on the number of quotients used. This is largely independent of the fill fraction, so the computation required per item decreases as the total number of items increases.

- The performance of the sequential shifting build increases until the filter is about 50% full, then begins to fall off. This is because as the filter gets full, the number of shifts, and therefore, the number of iterations, will increase. Additionally, the shift operation performed by each thread is also serial, so as the quotients' runs get longer, the latency of each operation increases.

- The segmented layouts build method is the fastest for all $\alpha \leq 0.85$. This method achieves a peak throughput at around 40% full, then performance decreases steadily as the filter gets fuller and more iterations are required for convergence. Even at the ideal maximum QF capacity of $\alpha = 0.75$, this build method still achieves higher throughput than BloomGPU.

- Deduplication does generally cause a moderate decrease in throughput (Figure 3.15). Interestingly, in the sequential shifting method, deduplication is costly for low filter fullness, but becomes insignificant to overall throughput as the filter fills and compute time is dominated by the many iterations required to perform all of the shifts.

Table 3.1. AMQ data structure memory use

	Bytes/Item			
Error Rate	Standard QF	RSQF	BloomGPU	Bloom 1-bit
0.03	1.3	0.94	7.7	0.96
0.0001	2.7	2.0	20.4	2.6
5×10^{-7}	4.0	3.0	45.2	5.7

3.7.4 Memory Use

Table 3.1 shows memory usage for all GPU AMQs. The RSQF has a smaller memory footprint than the SQF because it can be filled up to 95% full without compromising lookup performance, while the SQF should be sized to be 75% full. We note two limitations of our QF implementation with respect to memory usage: (1) Our SQF does not support a more fine-grained selection of false positive rates because we require slot sizes to be divisible into complete bytes, as described in Chapter 3.6. (2) Our bulk build methods allocate additional (temporary) memory to calculate element positions within the filter. This means that we cannot bulk-build a filter on the GPU that will fill a majority of the GPU on-board memory. For these filters, we would need to perform incremental inserts in smaller batches to construct the filter without running out of memory. However, real-world use cases would require additional free memory in order to read in batches of items for lookups anyway.

3.8 Conclusions

For inserts alone, the simplicity of modifications and resulting high level of parallelism available for BFs outweighs the locality benefits of the QF. In contrast, this locality does lead to better GPU performance for QF lookups, where memory conflicts are not an issue and parallelism is not constrained.

Recomputing dynamic independent regions between each round of updates (supercluster inserts) leads to higher throughput vs. parallelizing updates over fixed-sized regions (RSQF inserts). Although computing the supercluster locations requires additional work each round, it guarantees a priori that inserts in those regions will succeed. For RSQF inserts, the fixed-

sized regions we use are not guaranteed to be conflict-free and therefore require a strategy for handling overflows. By contrast, in order to achieve a high level of parallelism while avoiding conflicts for the supercluster inserts, we only allow our SQF to use remainders that fit in full bytes, which limits the number of available remainder sizes available. In the RSQF, the blocking structure divides the filter into segments that align with word boundaries, so we need not restrict the RSQF size and corresponding false positive rate.

To parallelize bulk QF construction, we needed to perform a parallel scan operation on a non-associative operator. In our three bulk build implementations, we leverage the fact that this operator has a saturation condition, and extract parallelism from breaks in the dependency chain.

Finally, the GPU RSQF performance could be improved if NVIDIA added support for a bit-manipulation operation equivalent to the PDEP operation available on the Intel Haswell architecture. This gives a significant performance boost on the CPU and would likely have a similar benefit for the GPU.

The code for our quotient filters is available at `https://github.com/owensgroup/GPUQuotientFilters`.

3.9 Pseudocode

Algorithm 5 SQF membership queries

```
1:  function FINDRUNSTART(S, f_q)
2:      ▷ find beginning of cluster
3:      b ← f_q
4:      while is_shifted(S[b]) do
5:          DECR(b)
6:      ▷ walk forward to find the run for f_q
7:      s ← b
8:      while b ≠ f_q do
9:          repeat
10:             INCR(s)                                   ▷ skip current run
11:         until !is_continuation(S[s])
12:         repeat
13:             INCR(b)                                   ▷ count number of runs
14:         until is_occupied(S[b])
15:     return s

16: function LOOKUP(S, inputs)
17:     ▷ preprocessing: hash and sort inputs
18:     for all inputs do
19:         f_q ← ⌊f[threadID]/2^r⌋
20:         f_r ← f[threadID]%2^r
21:         if !is_occupied(S[f_q]) then
22:             return FALSE
23:         s ← FINDRUNSTART(S, f_q)
24:         ▷ search slots in the run for f_r
25:         repeat
26:             if S[s] = f_r then
27:                 return TRUE
28:             INCR(s)
29:         until !is_continuation(S[s])
30:         return FALSE
```

Algorithm 6 RSQF membership queries

```
1: function LOOKUP(R, inputs)
2:   ▷ preprocessing: hash and sort inputs
3:   for all inputs do
4:     f_q ← ⌊f[threadID]/2^r⌋
5:     f_r ← f[threadID]%2^r
6:     b ← f_q/SLOTS_PER_BLOCK
7:     slot ← f_q%SLOTS_PER_BLOCK
8:     if R[b][slot].occ = 0 then
9:       return FALSE
10:    r ← RANK(R[b].occ, slot)
11:    end ← SELECT(R[b].run, r)
12:    while end = NULL do
13:      ▷ run end is in next block
14:      r ← r− POPCOUNT(R[b].run)
15:      INCR(b)
16:      end ← SELECT(R[b].run, r)
17:    s ← end
18:    repeat
19:      if R[b][s].rem = f_r then
20:        return TRUE
21:      if R[b][s].rem < f_r then
22:        return FALSE
23:      DECR(s)
24:    until s < f_q ∨ R[b][s].run = 1
25:  return FALSE
```

Algorithm 7 SQF inserts

```
1: function LocateSuperclusters(S)
2:     ▷ mark supercluster starts by checking for empty slots
3:     if IsEmpty(S[threadID − 1]) then
4:         return 1
5:     else
6:         return 0

7: function Bidding(S, inputs, labels)
8:     ▷ one thread per input bids for supercluster
9:     (f_q, f_r) ← HashAndQuotient(inputs[threadID])
10:    sc ← labels[f_q]
11:    return winners[sc] ← threadID

12: function InsertItems(S, inputs, winners)
13:    ▷ each thread performs its own sequential insert operation
14:    (f_q, f_r) ← HashAndQuotient(inputs[threadID])
15:    if IsEmpty(S[f_q]) then
16:        SetElement(S, f_q, f_r)
17:        return f_q
18:    s ← FindRunStart(S, f_q)                                  ▷ from Algorithm 5.1
19:    if is_occupied(S[f_q]) then
20:        ▷ search through run for item
21:        repeat
22:            if S[s] = f_r then
23:                SetElement(S, s, f_r)
24:                return s
25:            else if S[s] > f_r then
26:                break
27:            INCR(s)
28:        until !is_continuation(S[s])
29:    ▷ insert item at location s; move items right as needed
30:    InsertHere(S, s, f_r)
31:    return s

32: function Insert(S, inputs)
33:    repeat
34:        for all SQF slots do
35:            flags ←LocateSuperclusters(S)
36:        for all SQF slots do
37:            labels ←CUBScan(flags)
38:        for all remaining inputs do
39:            winners ←Bidding(S, inputs, labels)
40:        for all superclusters do
41:            slots ←InsertItems(S, inputs, winners)
42:        for all remaining inputs do
43:            ▷ compact out inserted values
44:            inputs ←CUBSelect(inputs, winners)
45:    until length(inputs) = 0
46:    return slots
```

Algorithm 8 Parallel merging bulk build

```
1: function CalcOffsets(slot, label, credit, offset, carry)
2:     seg ← label[threadID]
3:     ▷ compute offsets at odd-numbered segment heads:
4:     if seg ≠ label[threadID − 1] ∧ seg%2 = 1 then
5:         offset[seg] ← slot[threadID − 1] − slot[threadID] + 1
6:         carry[seg] ← credit[threadID − 1]
7:     return

8: function ShiftItems(offset, carry, slot, label, credit)
9:     seg ← labels[threadID]
10:    overlap ← offset[seg] − credit[threadID]
11:    empties ← 0
12:    if overlap > 0 then                                          ▷ shift item
13:        slot[threadID] ← slot[threadID] + overlap
14:        empties ← 0
15:    else                                                         ▷ track any empty slots
16:        empties ← empties − overlap
17:    credit[threadID] ← empties + carry[seg]
18:    ▷ merge segments
19:    label[threadID] ← label[threadID]/2
20:    return

21: function ParallelMergeBuild(S, inputs)
22:    ▷ preprocessing: hash, sort, compute unshifted locations, label segments
23:    for i ← 0, ⌈log2(segments)⌉ do
24:        for all inputs do
25:            CalcOffsets(slot, label, credit, offset, carry)
26:        for all inputs do
27:            ShiftItems(offset, carry, slot, label, credit)
28:    ▷ post-processing: write remainders and metadata
29:    return
```

Algorithm 9 Sequential shifting bulk build

```
1:  function SHIFTSEGMENTS(starts, slots, change)
2:      index ← starts[threadID]
3:      shift ← slots[index − 1] − slots[index] + 1
4:      if shift > 0 then
5:          length ← starts[threadID + 1] − index
6:          for i ← 0, length do
7:              slots[index + i] ← slots[index + i] + shift
8:          change ← 1
9:      return

10: function SEQUENTIALSHIFTBUILD(S, inputs)
11:     ▷ preprocessing: hash, sort, compute unshifted locations & segment starts
12:     change ← 1
13:     while change = 1 do
14:         change ← 0
15:         for all segments do
16:             SHIFTSEGMENTS(starts, locations, change)
17:     ▷ post-processing: write remainders and metadata
18:     return
```

Algorithm 10 Segmented layouts bulk build

```
1: function LAYOUT(f_q, start, shift, overflow, change)
2:     first ← start[threadID]
3:     last ← start[threadID + 1] − 1
4:     n ← last − first + 1
5:     if n ≤ 0 then
6:         ▷ segment is empty
7:         overflow[threadID] ← 0
8:         return
9:     ▷ track the furthest right element in the segment
10:    max ← threadID ∗ q + shift[threadID − 1]
11:    for i ← first, last do
12:        if f_q[i] > max then
13:            max ← f_q[i]
14:        INCR(max)
15:    ▷ check for overflow and changes from last iteration
16:    end ← ((threadID + 1) ∗ q) − 1
17:    extra ← (max − 1) − end
18:    if extra > 0 then
19:        overflow[threadID] ← extra
20:        if extra > shift[threadID] then
21:            change ← 1
22:    else
23:        overflow[threadID] ← 0
24:    return

25: function SEGMENTEDLAYOUTSBUILD(S, inputs)
26:    ▷ preprocessing: hash, sort, compute segment starts
27:    change ← 1
28:    while change = 1 do
29:        change ← 0
30:        shift ← overflow
31:        for all segments do
32:            LAYOUT(f_q, start, shift, overflow, change)
33:    ▷ post-processing: write remainders and metadata
34:    return
```

Algorithm 11 SQF deletes

```
1: function LocateDeleteSuperclusters(S)
2:   ▷ superclusters for deletes are regular clusters
3:   if !IsEmpty(S[threadID])∧!is_shifted(S[threadID]) then
4:     return 1
5:   else
6:     return 0

7: function DeleteItems(S, inputs, winners)
8:   ▷ each thread performs sequential delete operation
9:   (f_q, f_r) ← HashAndQuotient(inputs[threadID])
10:  if !is_occupied(S[f_q]) then
11:    return
12:  s ← FindRunStart(S, f_q)                                   ▷ from Algorithm 5.1
13:  repeat
14:    if S[s] = f_r then
15:      break
16:    else if S[s] > f_r then
17:      return
18:    INCR(s)
19:  until !is_continuation(S[s])
20:  if S[s]! = f_r then
21:    return
22:  ▷ s now points to item to be deleted
23:  ▷ delete item; move other items over as needed
24:  DeleteItemHere(S, s)
25:  return

26: function Delete(S, inputs)
27:  repeat
28:    for all SQF slots do
29:      flags ←LocateDeleteSuperclusters(S)
30:    for all SQF slots do
31:      labels ←CUBScan(flags)
32:    for all remaining inputs do
33:      ▷ from Algorithm 7.7
34:      winners ←Bidding(S, inputs, labels)
35:    for all superclusters do
36:      DeleteItems(S, inputs, winners)
37:    for all remaining inputs do
38:      ▷ compact out inserted values
39:      inputs ←CUBSelect(inputs, winners)
40:  until length(inputs) = 0
41:  return
```

Algorithm 12 Merging filters

```
1: function ExtractFingerprints(Q, empty)
2:   if IsEmpty(Q[threadID]) then
3:     empty[threadID] ← TRUE
4:     return
5:   if !is_shifted(Q[threadID]) then
6:     ▷ item is beginning of cluster
7:     return (threadID << r) ∨ Q[threadID]
8:   ▷ for shifted items, find beginning of cluster
9:   b ← threadID
10:  repeat
11:    INCR(b)
12:  until !is_shifted(Q[b])
13:  ▷ step through cluster, counting runs
14:  s ← b
15:  while s ≤ threadID do
16:    repeat
17:      INCR(s)
18:    until !is_continuation(Q[s])
19:    if s > threadID then
20:    repeat
21:      INCR(b)
22:    until !is_occupied(Q[b])
23:  return (b << r) ∨ Q[threadID]

24: function MergeFilters(Q_1, Q_2)
25:  for all QF slots do
26:    f_1 ← ExtractFingerprints(Q_1, empty_1)
27:  for all QF slots do
28:    f_2 ← ExtractFingerprints(Q_2, empty_2)
29:  for all QF slots do
30:    ThrustRemoveIf(f_1, empty_1)
31:  for all QF slots do
32:    ThrustRemoveIf(f_2, empty_2)
33:  for all extracted values do
34:    f_combined ← MGPUMerge(f_1, f_2)
35:  ▷ rebuild new filter
36:  SegmentedLayoutsBuild(Q_new, f_combined)
37:  return
```

Algorithm 13 RSQF insert kernel

```
1: function INSERTINTOREGIONS(R, starts, nexts, f_q, f_r, size)
2:   first ← threadID * size
3:   last ← first + size − 1
4:   value ← nexts[first]
5:   block ← first
6:   while value = NULL ∧ block < last do
7:     ▷ insert queue for block is empty - check next one
8:     INCR(block)
9:     value ← nexts[block]
10:  if value = NULL then
11:    return                                              ▷ no items in queue
12:  home ← f_q[value]%SLOTS_PER_BLOCK
13:  r ← RANK(R[block].occ, home)
14:  end ← SELECT(R[block].run, r)
15:  while end = NULL do
16:    ▷ run end is in next block
17:    r ← r− POPCOUNT(R[block].run)
18:    INCR(block)
19:    end ← SELECT(R[block].run, r)
20:  if block > last then
21:    return TRUE                                         ▷ item is in next region
22:  if end < home then
23:    ▷ slot is empty; insert item here
24:    INSERTHERE(R, end, f_r[value])
25:    INCR(nexts[block])
26:    return TRUE
27:  else
28:    ▷ search through filter for first empty slot
29:    INCR(end)
30:    s ← FINDFIRSTUNUSEDSLOT(R, block, end)
31:    if block > last then
32:      return TRUE                                       ▷ out of region
33:    while s > end do
34:      ▷ move items over until we get back to item's run
35:      R[block][s].rem ← R[block][s − 1].rem
36:      R[block][s].rem ← R[block][s − 1].rem
37:      DECR(s)
38:    ▷ find correct slot in run
39:    repeat
40:      if R[block][s − 1].rem ≤ f_r[value] then
41:        INSERTHERE(R, s, f_r[value])
42:        INCR(nexts[block])
43:        return TRUE
44:      DECR(s)
45:    until s < home ∨ R[block][s].run = 1
46:  INSERTHERE(R, s, f_r[value])
47:  INCR(nexts[block])
48:  return TRUE
```

Algorithm 14 RSQF inserts

```
1: function FindFirstUnusedSlot(R, block, slot)
2:     r ← Rank(R[block].occ, slot)
3:     s ← Select(R[block].run, r)
4:     while slot ≤ s do
5:         if s = NULL then
6:             INCR(block)
7:             s ← R[block].offset + 1
8:         slot ← s + 1
9:         r ← Rank(R[block].occ, slot)
10:        s ← Select(R[block].run, r)
11:    return slot

12: function FindBlockStartIndices(f_q, starts)
13:    block ← f_q[threadID]/SLOTS_PER_BLOCK
14:    previous ← f_q[threadID − 1]/SLOTS_PER_BLOCK
15:    if block ≠ previous then
16:        starts[block] ← threadID

17: function Insert(R, inputs)
18:    ▷ preprocessing: hash, sort, quotienting
19:    for all inputs do
20:        FindBlockStartIndices(f_q, starts)
21:        iterations ← 1
22:        size ← 1
23:    while more do
24:        more ← FALSE
25:        nregions ← nblocks/size
26:        for all regions do
27:            more ← InsertIntoRegions(R, starts, nexts, f_q, f_r, size)
28:        INCR(iterations)
29:        size ← iterations/16 + 1
30:    return
```

Chapter 4

Conclusions

Our work demonstrates that while the massively parallel architecture of GPUs is effective for performing analysis of large datasets, memory size limitations can pose a challenge for many of these problems. The three main strategies we use to mitigate this challenge are:

- *Approximation*: For maximum clique, heuristics use much less memory than an exact computation in order to find cliques that are large, but not guaranteed to be maximum. Introducing a small false positive rate enables quotient filters to use only a small fraction of the space required for an exact membership query.

- *Preprocessing*: For quotient filters, we deduplicate the dataset before constructing the filter. We also hash the keys into fingerprints which reduces memory use because we can store much of the fingerprint implicitly. Our maximum clique implementation pre-prunes both vertices and entire candidate lists. We also implicitly orient the graph by degree, which eliminates half of the edges (and many many potential combinations of edges), without sacrificing the accuracy of the solution. Additionally, we increase the probability of pruning even more candidates in earlier stages by sorting vertices in order of increasing degree within their sublists.

- *Breaking the Problem into Smaller Pieces*: In our maximum clique implementation, when we found our other memory reducing strategies were insufficient, we tried a windowed approach. By solving smaller subproblems one at a time, we were able to solve more datasets without running out of memory, but at a significant runtime cost.

Next, I propose a variety of future directions for research based on our work.

4.1 Future Work: Quotient Filter

4.1.1 Parallel Operations on Many Small Quotient Filters

Our quotient filters are a good fit for queries on a single large dataset, but another useful application space may be a scenario where there are many smaller datasets, like perhaps, many lists of cliques that one would like to perform membership queries on. This would require different parallelization strategies, and would likely make many of the operations we perform in bulk better-suited to a single thread or a warp working cooperatively. In this scenario, the filters could fit in shared memory, which would further improve throughput.

4.1.2 More Choices for False Positive Rate

One of the biggest limitations of our GPU standard quotient filter is the requirement that the slots be divisible by 8. This reduces the tunability of the filter in that it greatly reduces the number of options for the false positive rate, so future work devising an algorithm for parallel inserts that allows for more filter sizes while still maintaining enough parallel work would be valuable. Possibly a new type of quotient filter structure could be designed to address this problem and allow this data structure to be used in a wider range of GPU applications.

4.1.3 Coarse-Grained Parallel Inserts for RSQF

For the rank-and-select-based quotient filter, the blocked structure reduces the amount of available parallelism, because all slots within one block share their metadata. We may be able to improve performance by instead performing operations in a coarse-grained warp-parallel fashion, assigning groups of threads to perform all insert operations in one block cooperatively. Because threads within a warp can easily communicate, they can work together on tasks and reliably avoid race conditions. The filter block size of 64 also fits well with the CUDA warp size of 32 threads.

4.2 Future Work: Maximum Clique Enumeration

4.2.1 Windowing Improvements

4.2.1.1 Save Only Largest Cliques' Information

We could alter the version with windowing to perform a full enumeration while also further reducing memory usage by reading out and storing the "best-cliques-so-far" and freeing the rest of the clique list when a new biggest clique is discovered. This is a fairly slow operation on our clique list data structure, so this would once again be a trade-off of performance for reduced memory usage (and finding all maximum cliques).

4.2.1.2 Dynamic Windowing

As mentioned in Chapter 2.5.4.3, another possible way to reduce memory usage in our maximum clique implementation would be to implement windowing in a more flexible way. Rather than using a fixed window size for only the first level of the clique list, we could try to dynamically determine if and when to start windowing and what window size to use. We could use the full breadth-first search to expand the amount of available work until we have plenty of work to fill the GPU, then use windowing to explore the candidate lists in a more depth-first way to avoid running out of memory. We could also try to predict how much the clique lists are likely to expand, perhaps using the average degree or average sublist length or the increase in the number of candidate cliques between the previous two iterations. We could then select the window size based on the estimated increase in the clique list size in the next iteration.

Of course, predicting the number of cliques in the next iteration is challenging. We cannot even guarantee whether the number will increase or decrease from the previous iteration; however, we do find that the clique counts usually follow one of two trends: a bell curve or an initial peak followed by a steady decrease. Using this information, we can choose to either not use windowing or select a larger window size once the total number of cliques begins to decrease, increasing the available parallel work.

4.2.1.3 Coarse-Grained Parallel Windowing

Another possible expansion of windowing would be to assign blocks of threads to explore their own windows independently. Although this seems like a way to utilize the GPU more efficiently, it would be very complicated to implement. Additionally, with multiple windows of different

sizes and windowing on different levels, tracking which candidates have been explored would be cumbersome. We would likely need to implement a scheduling system to manage the work queue, which is a significant challenge itself. Also, because each of the blocks would be allocating and freeing variable amounts of memory in their own clique lists at different times, it would be even harder to avoid running out of memory.

4.2.2 Multi-GPU Implementation

To handle larger datasets, including graphs in which the dataset itself may require most of the GPU memory, we could consider a multi-GPU implementation. There are many additional complications to consider when coordinating multiple GPUs, but a starting point for a multi-GPU implementation could be to do a form of windowing. Each GPU could be assigned its own window of the clique list, which would be much larger than the window sizes used in our implementation, but would function in the same way. The search in each window can proceed independently, and when all GPUs have finished, they can compare their largest cliques and determine which clique(s) are the largest overall.

4.2.3 Heterogeneous GPU + CPU Implementation

We found that there is often not sufficient work to keep the GPU busy in the earlier and/or later iterations of the exact maximum clique enumeration algorithm. For these iterations, we could instead perform the work on the CPU, taking advantage of the lower latency to complete this work more quickly. One drawback of this approach is data transfer time from the GPU to CPU and vice versa. However, because this would only be used for instances where the clique list is small, the amount of data transferred should be small. This strategy would be most likely to be beneficial for the final iterations, when we know the number of cliques will continue to decrease in each iteration, as described in Chapter 4.2.1.2.

4.2.4 Clique List in Shared Memory

Our maximum clique implementation does not utilize shared memory at all. Shared memory is shared by threads in a block, and allows for much faster accesses than global memory; however, using shared memory will only improve performance if the data is reused multiple times, because of the initial cost of moving the data from global to shared memory. Because threads

read all values in their assigned vertex's sublist, we may be able to achieve some performance benefit by pulling the relevant sublists into shared memory before beginning the edge checks. This technique should provide at least a small speedup, but the biggest contribution to memory traffic in the `countCliques` and `outputNewCliques` kernels is the edge lookups in the graph data structure, and because each thread is accessing a different vertex's adjacency list, there is no opportunity for data reuse in those operations.

4.2.5 Pruning Improvements

4.2.5.1 Improving Lower Bound

Although our heuristic achieves high accuracy for most datasets, we could further improve pruning for harder-to-solve graphs by using a modified version of our exact algorithm. We could run the exact computation once, but rather than building the full clique list, we could free the memory from earlier clique lists, only maintaining the list of candidates for the current value of k. This would greatly reduce the total memory required and may allow us to avoid running out of memory. When this reduced memory computation has finished, we will know the size and number of maximum cliques and one vertex from each clique (the last-added vertex for each clique). We can use this information to rerun the full enumeration algorithm, but now we have a perfect lower bound and we could also prune any vertices not connected to the vertices in the final clique list node. Although this strategy sounds very promising, it is less practically useful than it appears, because for many of the challenging graphs the peak clique list size for just one iteration is still larger than the available GPU memory. Still, there would certainly be a subset of graphs for which this strategy would be an effective solution for avoiding running out of memory.

4.2.5.2 Better Sublist Pruning

Our implementation prunes sublists in each iteration based on their length, that is, if the sublist length plus k is less than the length of the largest clique found so far then we do not output the sublist to be explored in the next iteration. We could instead use a stronger bound to prune sublists, reducing the total memory use. As mentioned in Chapter 2.2.2.3, one option for a tighter bound is vertex coloring. If a graph, G, can be colored with k colors, then $\omega(G) \leq k$. Similarly, a coloring of the set P gives an upper bound on the size of the largest clique

in the subgraph induced by P. Finding an optimal coloring of P at every branch point is computationally expensive, so we would want to use an approximate coloring. And of course we should choose a GPU-friendly implementation, which can be efficiently parallelized over many subsets of vertices. This more complex pruning step presents yet another trade-off between precompute time and pruning effectiveness.

4.2.5.3 Explicitly Prune the Graph

Another way to save memory and possibly speed up the exact algorithm at the same time is to explicitly prune the graph during preprocessing. This would reduce the memory used by the graph data structure itself, and should also speed up edge lookups, because vertices' adjacency lists will be shorter. The vast majority of memory is used by the clique list structure, so reducing memory used by the graph will likely only have a marginal effect, but pruning the graph would also reduce vertices' degrees, further improving vertex pruning. The challenge for this will be in modifying the graph data structure. It is not possible to efficiently delete vertices from a CSR data structure, so this would likely require a full rebuild of the data structure, which is very time-consuming. Therefore, we would likely want to consider using a different graph data structure that is easier to update.

4.2.6 Data Structures

4.2.6.1 Faster Edge Lookups

Since the key operation in finding cliques is performing edge lookups, a data structure that allows for faster lookups could significantly improve the performance of our maximum clique implementation. Some previous work has utilized an adjacency matrix structure to enable fast set intersection operations; however, most real world datasets are large and sparse, and the adjacency matrix representation is not space-efficient for these datasets. Another possible way to speed up edge lookups might be to use an approximate membership query data structure like the quotient filter to quickly filter out most of the failed lookups. This would require representing each vertex's adjacency list as a small filter, and rather than parallelizing many operations on one large filter, we would instead be performing operations on many small filters. A downside of this approach is that although AMQ data structures are small, they would use additional memory in an application where memory space is already limited. We could however

use the QFs to replace the complete graph, then run the full breadth-first maximum clique algorithm. This would give us a set of *approximate* maximum cliques. Because quotient filters will only return false positives, not false negatives, we can guarantee that no larger cliques were missed, but some of the reported ”cliques” may not be true cliques. We could then verify these cliques by checking whether the edges are truly present in the full graph. This post-processing should be minimal compared to the number of edge lookups in all iterations of the complete algorithm; however, it would involve the inefficient process of reading cliques from the clique list, possibly multiple times, if the approximate maximum clique(s) are found to be invalid.

4.2.6.2 Delete Clique Information

Another useful improvement would be to either (1) devise an efficient method for deleting cliques from our clique list data structure or (2) implement a different method for storing clique information that allows for simpler deletes. For (1), we may be able to draw from our experience in performing bulk rebuilds and batch deletes on quotient filters to implement similar operations on the clique list. However, because the data in each node of the clique list is dependent on data in other nodes, this is likely to be come with even more complications than we found when implementing parallel quotient filter operations. As for option (2), we chose our data structure to minimize the memory footprint and keep relevant data for each iteration close together (as explained in Chapter 2.4.4), sacrificing mutability and ease of reading out the members of each clique, but a data structure with fewer downsides may be achievable.

4.2.7 Applicability to Other Problems

4.2.7.1 Maximum Clique Size Only

For some applications, it may be useful to know just the maximum clique size, ω. For example, it could be used as an upper bound for memory allocation in another graph computation or for comparing properties of different graphs. In this case, we could use the reduced-memory implementation proposed in Chapter 4.2.5.1 to find only the maximum clique size. This would use less memory than the full enumeration computation, and provide an exact maximum clique size, rather than an estimate that could be found via a heuristic.

4.2.7.2 General Clique Counting or Enumeration

Counting k-cliques is another operation that is useful in graph analysis [1]. Our implementation can be used as is to enumerate (or, more simply, to count) all cliques in the graph by simply skipping the heuristic computation. Alternatively, we can find all cliques above a certain size using an input lower bound instead of a lower bound from the heuristic.

4.2.7.3 Streaming Maximum Clique

Another related, but more complex, problem is formulation of maximum clique as a streaming problem. In the streaming context, a graph is being modified, with vertices and/or edges added or deleted, and the goal is to track whether and how these changes affect the maximum clique(s) of the graph. We could use the information in the clique list structure to find vertices that are members of the maximum clique(s) and also all smaller cliques, but the information in the structure becomes invalid once updates have been made. Here again, techniques for performing updates in the clique list structure and/or a more easily updated structure would be useful. Quotient filters may also be useful here. We could store the maximum clique(s), and maybe also some of the near-maximum cliques, in quotient filters. This would enable fast membership queries of newly-added or newly-deleted edges to filter out updates that will not affect the maximum clique(s). The flexibility to delete and merge quotient filters may also be useful as edges are deleted and added and cliques merge into one another.

REFERENCES

[1] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. Parallel k-clique counting on GPUs. In *Proceedings of the 36th ACM International Conference on Supercomputing*, volume 21 of *ICS '22*, pages 1–14. Association for Computing Machinery, June 2022. doi: 10.1145/3524059.3532382.

[2] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, August 2012. doi: 10.14778/2350229.2350275.

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970. doi: 10.1145/362686.362692.

[4] Vladimir Boginski, Sergiy Butenko, and Panos M. Pardalos. Statistical analysis of financial networks. *Computational Statistics and Data Analysis*, 48(2):431–443, February 2005. doi: 10.1016/j.csda.2004.02.004.

[5] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. The maximum clique problem. In *Handbook of Combinatorial Optimization*, pages 1–74. Kluwer Academic Publishers, 1999. doi: 10.1007/978-1-4757-3023-4_1.

[6] James Cheng, Linhong Zhu, Yiping Ke, and Shumo Chu. Fast algorithms for maximal clique enumeration with limited memory. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1240–1248. ACM, August 2012. doi: 10.1145/2339530.2339724.

[7] Lauro B. Costa, Samer Al-Kiswany, and Matei Ripeanu. GPU support for batch oriented workloads. In *IEEE 28th International Performance Computing and Communications Conference*, IPCCC 2009, pages 231–238, December 2009. doi: 10.1109/PCCC.2009.5403809.

[8] R. Cruz, N. López, and C. Trefftz. Parallelizing a heuristic for the maximum clique problem on GPUs and clusters of workstations. In *IEEE International Conference on Electro-Information Technology*, EIT 2013, pages 1–6, May 2013. doi: 10.1109/EIT.2013.6632645.

[9] Feras Daoud, Amir Watad, and Mark Silberstein. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, pages 6:1–6:8, June 2016. doi: 10.1145/2931088.2931091.

[10] Nan Du, Bin Wu, Liutong Xu, Bai Wang, and Pei Xin. *Parallel Algorithm for Enumerating Maximal Cliques in Complex Network*, pages 207–221. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-540-88067-7_12.

[11] Sourav Dutta, Ankur Narang, and Suman K. Bera. Streaming quotient filter: A near optimal approximate duplicate detection approach for data streams. *Proceedings of the VLDB Endowment*, 6(8):589–600, June 2013. doi: 10.14778/2536354.2536359.

[12] Afton Geil, Yangzihao Wang, and John D. Owens. WTF, GPU! Computing Twitter's who-to-follow on the GPU. In *Proceedings of the Second ACM Conference on Online Social Networks*, COSN '14, pages 63–68, October 2014. doi: 10.1145/2660460.2660481. URL `http://escholarship.org/uc/item/5xq3q8k0`.

[13] Oded Green, Robert McColl, and David A. Bader. GPU merge path: A GPU merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 331–340, June 2012. doi: 10.1145/2304576.2304621.

[14] Christopher Henry. A parallel GPU solution to the maximal clique enumeration problem for CBIR. In *GPU Technology Conference (GTC 2014)*, March 2014. URL `https://on-demand.gputechconf.com/gtc/2014/presentations/S4510-maximal-clique-enumeration-problem-cbir.pdf`.

[15] Christopher J. Henry and Sheela Ramanna. Maximal clique enumeration in finding near neighbourhoods. *Transactions on Rough Sets XVI*, pages 103–124, 2013. doi: 10.1007/978-3-642-36505-8_7.

[16] Alexandru Iacob, Lucian Itu, Lucian Sasu, Florin Moldoveanu, and Constantin Suciu. GPU accelerated information retrieval using Bloom filters. In *2015 19th International Conference on System Theory, Control and Computing*, ICSTCC 2015, pages 872–876, October 2015. doi: 10.1109/ICSTCC.2015.7321404.

[17] John Jenkins, Isha Arkatkar, John D. Owens, Alok Choudhary, and Nagiza F. Samatova. Lessons learned from exploring the backtracking paradigm on the GPU. In *Euro-Par 2011 Parallel Processing*, pages 425–437. Springer Berlin Heidelberg, August 2011. doi: 10.1007/978-3-642-23397-5_42.

[18] Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, pages 85–103. Springer US, March 1972. doi: 10.1007/978-1-4684-2001-2_9.

[19] Frank Kose, Wolfram Weckwerth, Thomas Linke, and Oliver Fiehn. Visualizing plant metabolomic correlation networks using clique metabolite matrices. *Bioinformatics*, 17 (12):1198–1208, December 2001. doi: 10.1093/bioinformatics/17.12.1198.

[20] Brenton Lessley, Talita Perciano, Manish Mathai, Hank Childs, and E. Wes Bethel. Maximal clique enumeration with data-parallel primitives. In *IEEE 7th Symposium on Large Data Analysis and Visualization*, LDAV '17, pages 16–25, Oct 2017. doi: 10.1109/LDAV.2017.8231847.

[21] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI. *BMC Bioinformatics*, 12(1):85, March 2011. doi: 10.1186/1471-2105-12-85.

[22] Lin Ma, Roger D. Chamberlain, Jeremy D. Buhler, and Mark A. Franklin. Bloom filter performance on graphics engines. In *2011 International Conference on Parallel Processing*, ICPP 2011, pages 522–531, September 2011. doi: 10.1109/ICPP.2011.27.

[23] Ciaran McCreesh and Patrick Prosser. Multi-threading a state-of-the-art maximum clique algorithm. *Algorithms*, 6(4):618–635, October 2013. doi: 10.3390/a6040618.

[24] Duane Merrill. CUDA UnBound (CUB) library, 2015–2022. `https://nvlabs.github.io/cub/`.

[25] John W. Moon and Leo Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3(1): 23–28, March 1965. doi: 10.1007/bf02760024.

[26] Iulian Moraru and David G. Andersen. Exact pattern matching with feed-forward Bloom filters. *J. Exp. Algorithmics*, 17:3.4:3.1–3.4:3.18, September 2012. doi: 10.1145/2133803.2330085.

[27] Shuai Mu, Xinya Zhang, Nairen Zhang, Jiaxin Lu, Yangdong Steve Deng, and Shu Zhang. IP routing processing with graphic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 93–98, March 2010. doi: 10.1109/DATE.2010.5457229.

[28] Bruno Nogueira and Rian G.S. Pinheiro. A CPU-GPU local search heuristic for the maximum weight clique problem on massive graphs. *Computers & Operations Research*, 90: 232–248, February 2018. doi: 10.1016/j.cor.2017.09.023.

[29] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 775–787, May 2017. doi: 10.1145/3035918.3035963.

[30] Arash Partow. C++ Bloom filter library. URL `http://www.partow.net/programming/bloomfilter/index.html`.

[31] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 4292–4293, January 2015. doi: 10.1609/aaai.v29i1.9277.

[32] Ryan A. Rossi, David F. Gleich, and Assefaw Hadish Gebremedhin. Parallel maximum clique algorithms with applications to network analysis. *SIAM J. Scientific Computing*, 37 (5):C589–C616, December 2015. doi: 10.1137/14100018X.

[33] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2): 571–581, February 2011. doi: 10.1016/j.cor.2010.07.019.

[34] 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, April 2009. doi: 10.1016/j.jpdc.2009.01.003.

[35] Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3): 269–287, September 1983. doi: 10.1016/0378-8733(83)90028-X.

[36] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498, March 2013. doi: 10.1145/2451116.2451169.

[37] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proceedings of the VLDB Endowment*, 6 (14):1930–1941, September 2013. doi: 10.14778/2556549.2556574.

[38] Matthew VanCompernolle, Lee Barford, and Frederick Harris. Maximum clique solver using bitsets on GPUs. In *Information Technology: New Generations*, pages 327–337. Springer International Publishing, March 2016. doi: 10.1007/978-3-319-32467-8_30.

[39] A. B. Vavrenyuk, N. P. Vasilyev, V. V. Makarov, K. A. Matyukhin, M. M. Rovnyagin, and A. A. Skitev. Modified Bloom filter for high performance hybrid NoSQL systems. *Life Science Journal*, 11(7s):457–461, 2014. doi: 10.7537/marslsj1107s14.98.

[40] Jiannan Wang, Tim Kraska, Michael J. Franklin, and Jianhua Feng. CrowdER: Crowdsourcing entity resolution. *Proc. VLDB Endow.*, 5(11):1483–1494, July 2012. doi: 10.14778/2350229.2350263.

[41] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. Gunrock: GPU graph analytics. *ACM Trans. Parallel Comput.*, 4(1):3:1–3:49, August 2017. doi: 10.1145/3108140.

[42] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*, volume 8. Cambridge University Press, 1994. doi: 10.1017/CBO9780511815478.

[43] Yi-Wen Wei, Wei-Mei Chen, and Hsin-Hung Tsai. Accelerating the Bron-Kerbosch algorithm for maximal clique enumeration using GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 32(9):2352–2366, March 2021. doi: 10.1109/TPDS.2021.3067053.

[44] Jingen Xiang, Cong Guo, and A. Aboulnaga. Scalable maximum clique computation using MapReduce. In *29th IEEE International Conference on Data Engineering*, ICDE 2013, pages 74–85, April 2013. doi: 10.1109/ICDE.2013.6544815.

[45] Yun Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova. Genome-scale computational approaches to memory-intensive applications in systems biology. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, SC '05, page 12, Nov 2005. doi: 10.1109/SC.2005.29.