GPU-accelerated Perfect Hash Construction: Parallel Implementation of the BDZ Algorithm and Application to Xor Filters

GPU-accelerated Perfect Hash Construction:

Parallel Implementation of the BDZ Algorithm and Application to Xor Filters

Lu Sien Chua

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2023

Abstract

A perfect hash function (PHF) is an injection on a set of n keys S, mapping every key in S to integers in the interval $[0, m-1]$ with no collisions, $m \geq n$. The BDZ algorithm constructs PHFs using peeling processes on random hypergraphs, and is an algorithm suited for key set sizes where the induced hypergraph fits in internal memory. In this work, we exploit the inherent parallelism present in the BDZ algorithm to introduce a GPU-accelerated construction algorithm, and show how it can be applied to a new type of approximate membership query (AMQ) data structure called the xor filter. We compare construction performance with sequential implementations, where our results show discernible improvements in construction time.

Acknowledgements

I would like to thank Professor Michael Mitzenmacher for inspiring the ideas in this thesis when I attended his class CS 222 “Algorithms at the End of the Wire” in fall of 2020, and for his guidance throughout my research journey.

I would also like to express gratitude to Professor Hongming Wang for her dedicated assistance, and valuable feedback throughout the thesis process.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter I.

Introduction

1.1. Overview

A perfect hash function (PHF) is an injection on a set S of n keys, mapping each key in S to integers in the interval $[0, m-1]$ with no collisions[1], $m \geq n$. When $m = n$, the PHF is a bijective, *minimal* perfect hash function (MPHF).

PHFs can be useful ingredients for building space-efficient data structures; we describe an implementation example. When building a hash table for an immutable input collection of key-value pairs, a core issue is the non-negligible likelihood[2] of a hash collision between any two keys. To resolve potential collisions, the keys must be stored along with the values in the hash table. However, if the application only needs to look up keys which are in S, or that false positives[3] can be tolerated, then we can use a PHF on S to implement the hash table, and skip storing the keys altogether. This yields savings in space, which can be significant when the values

[1] A collision occurs when the hash function produces the same hash value for two distinct keys.

[2] By the birthday paradox, which states that the probability p of two people sharing a birthday within a small group is surprisingly high. Only 23 randomly chosen individuals is needed for p to exceed 0.5

[3] Reporting an arbitrary value for a key which is not in the set

take up relatively less space compared to the keys.

Efficient techniques (Majewski et al., 1996; Chazelle et al., 2004; Botelho et al., 2007) have been proposed that find PHFs by *construction*, and these advances have led to interesting new possibilities in applications. For instance, the PHF construction of Botelho et al. (2007), also called the BDZ algorithm, was used to implement a new approximate membership query[4] (AMQ) data structure called the xor filter (Graf & Lemire, 2020). The filter inherits the compact representation and constant-time evaluation properties of the underlying PHF, and in practical tests, outperforms the classical Bloom and cuckoo filters in space and evaluation time. However, the construction time (i.e. the BDZ algorithm runtime) is comparatively slow, empirically observed to be approximately 2x slower than a Bloom filter construction.

1.2. Contribution

In this thesis, we focus on implementing an accelerated BDZ algorithm for execution on modern parallel hardware. Our basis stems from theoretical and practical results in hypergraph peeling processes (Jiang et al., 2013): these indicate that core construction steps of the BDZ algorithm contain untapped parallelism which could yield concrete speedups. At the time of writing, we do not know of a parallel implementation of the BDZ algorithm. We create new parallel algorithms, implement these on GPUs and evaluate performance in applications: the xor filter and a newer variant known as the fuse filter.

[4]An AMQ answers if an element is in a set or not with some false positive rate ϵ

1.3. Thesis outline

Outline. In Chapter 2 we review related work, including foundations to the BDZ algorithm, as well as theoretical and practical results in parallel peeling processes. Chapter 3 introduces the classical algorithm, providing an overview of the BDZ algorithm and a discussion of its inherent parallelism. In Chapter 4, we present our implementation of the new parallel algorithms, and detail use of these algorithms in applications: the xor and fuse filters. In Chapter 5, we describe our experiments and results. Finally, in Chapter 6, we summarize our work and conclude the thesis.

Chapter II.

Background

2.1. Peeling processes

Given a random r-uniform hypergraph[1] as input, a peeling process repeatedly removes vertices with degree less than k, along with edges incident to these vertices. At termination, it returns the k-core of the hypergraph, defined as the largest subgraph where every vertex is of degree at least k. The peeling process is efficient: algorithms using peeling techniques generally have runtimes linear in the size of the hypergraph, and hence seem adequate for handling problems with large input data sizes. In fact, there exists a varied range of peeling-based applications, including low-density parity check-codes (Luby et al., 2001), hash-based sketches (Chazelle et al., 2004; Goodrich & Mitzenmacher, 2011), cuckoo hashing (Pagh & Rodler, 2004) and satisfiability of random Boolean formulae (Broder & Mitzenmacher, 2003; Molloy, 2005).

Data structures and algorithms that use a peeling process often aim to find an

[1]A hypergraph is a generalization of an undirected graph, such that every edge connects $r \geq 2$ vertices. A r-uniform hypergraph is a hypergraph where every edge consists of r distinct vertices.

empty k-core, i.e. to peel away the whole hypergraph. It is known that asymptotically, this condition occurs with high probability if the edge density (number of edges over vertices) c of the hypergraph is below a threshold value $c^*_{k,r}$ (Molloy, 2005); above this threshold, the k-core is non-empty with high probability. Applications frequently employ this fact as part of initial configuration before creating the hypergraph. The particular configuration of $k = 2, r = 3$ and $c < c^*_{2,3} \approx 0.818$ appears to be common (Graf & Lemire, 2020; Goodrich & Mitzenmacher, 2011; Botelho et al., 2007). In later sections where we discuss specific algorithmic examples, we state why this case is used in context.

2.1.1 Peeling order

When a random r-uniform hypergraph has an empty 2-core, it is possible to define a peeling order: it is an ordering of the edges such that each edge is incident to at least one degree-1 vertex in the subgraph as previous edges in the order are peeled away. A peeling order appears particularly useful when a hypergraph is used to model a system of linear equations: a vertex corresponds to a variable, and an edge corresponds to an equation binding variables together. By arranging the system equations according to this order, each equation is guaranteed to have at least one variable which does not appear in any subsequent equation in the order. The system becomes triangular, and would be solvable by backward substitution. The BDZ algorithm, a focus of this thesis, is a known technique for constructing PHFs using a peeling order; we discuss further details in section 2.2.

2.1.2 Parallelism in peeling

Jiang et al. (2013) analyzed parallelism in the peeling process, considering the simple round-based algorithm: in each round, all vertices of degree less than k and their associated edges are removed in parallel from the hypergraph. The authors show that with high probability, when $c < c^*_{k,r}$, only $\frac{1}{\log((k-1)(r-1))} \log\log m + O(1)$ rounds of peeling are required to complete the peeling process (Logarithms are to the base 2 in this thesis unless otherwise specified). This is possible as the analysis reveals that the fraction of vertices remaining after each round falls doubly exponentially. We state the original theorem (Jiang et al., 2013) for clarity:

Theorem 1. Let $k, r \geq 2$ with $k + r \geq 5$, let c be a constant, and let $G^r_{m,cm}$ denote a r-uniform random hypergraph with m vertices and cm edges. With probability $1 - o(1)$, the parallel peeling process for the k-core in $G^r_{m,cm}$ with edge density c terminates after $\frac{1}{\log((k-1)(r-1))} \log\log m + O(1)$ rounds when $c < c^*_{k,r}$.

Theorem 1 translates well into fast practical performance, which Jiang et al. (2013) demonstrates using the algorithmic example of Invertible Bloom Lookup Tables (IBLTs) (Goodrich & Mitzenmacher, 2011). An IBLT — represented as an array — stores a set S of keys, and every key is hashed to r random locations in the array. When multiple keys hash to the same location, the keys are XOR-ed[2]. The IBLT is modeled by a random r-uniform hypergraph, where the set keys correspond to edges, and array cells correspond to vertices. The goal is to recover the set S from the IBLT,

[2]With the XOR trick, the same operation can be used to add or remove a key from the array.

and this is achieved by peeling[3] the hypergraph. The authors implemented in parallel the core steps of IBLT construction (creating the hypergraph) and set recovery (peeling the hypergraph) on the GPU. Encouragingly, they observe speedups of 10-12x and 20x respectively over the sequential algorithm.

2.2. The MWHC construction technique

The work of Majewski et al. (1996) proposed the first known technique that uses a peeling process on a random r-uniform hypergraph to construct a MPHF. For a set of n keys $S \subseteq U$, the technique generates a special, compact array g for computing a function $f : S \to \{0, 1, ..., n-1\}$.

We describe the construction of g based on 3-uniform hypergraphs[4], i.e. $r = 3$. Three random hash functions $h_0, h_1, h_2 : U \to \{0, 1, ..., \lambda n - 1\}$ are chosen, where $\lambda = 1.23$. Then a hypergraph of λn vertices and n edges is created by mapping each key $x \in S$ to the edge $\{h_0(x), h_1(x), h_2(x)\}$. The hypergraph models a linear system: a vertex corresponds to a variable, and an edge corresponds to an equation binding three variables. The goal is to solve the system, i.e. find an array of variables g_i, such that for every key $x \in S$,

$$g_{h_0(x)} + g_{h_1(x)} + g_{h_2(x)} = f(x) \mod n$$

[3]A peeling step proceeds as follows: find a cell containing a single key (we refer to as a "pure" cell), and place the key into the result set. Then remove the key from the r locations of the array.

[4]The construction works for r-uniform hypergraphs, $r \geq 2$, but space usage is minimized at $r = 3$ (Molloy, 2005)

If the hypergraph is peelable, then it is possible to traverse all the edges in *reverse* peeling order. For each edge, the algorithm associates it with a unique integer $a \in \{0, 1, \dots, n-1\}$. The considered edge has one or more *unassigned*[5] vertices; we denote the set of such vertices as $\{v_0, v_1, \dots, v_j\}$. Then the algorithm sets the value zero to $g[v_1], \dots, g[v_j]$, and assigns $g[v_0] = a - \sum_{i=1}^{r} g[v_i] \mod n$.

The algorithm almost always succeeds, and is fast. First, there is a strong probabilistic guarantee that the hypergraph is peelable (and thus lead to a solvable system). With $\lambda = 1.23$, the hypergraph created has an edge-over-vertices ratio (edge density) of $1/1.23 \approx 0.813$ that is slightly below the critical threshold $c^*_{2,3} \approx 0.818$, hence ensuring it is peelable with probability $1 - o(1)$ (Molloy, 2005). Second, peeling processes have runtimes linear in the size of the hypergraph, leading to efficient construction times.

The function $f(x)$ is stored using $\lambda n \lceil \log n \rceil$ bits for the array of variables g_i. Assuming the hash functions h_0, h_1 and h_2 compute in $O(1)$ time, evaluating the function $f(x)$ would also take $O(1)$ time, with three hash computations using the chosen hash functions and three memory probes into the array.

The construction can be easily modified to store a PHF in reduced $O(n)$ space. This is because a peelable hypergraph is also orientable[6]: we can associate (orient) each edge e to a vertex $v \in e$ where v and e were peeled together in a particular peel

[5]These vertices are termed unassigned to indicate they are not yet associated with a value; unassigned vertices are guaranteed to exist due to edge traversal by peeling order, which we described in section 2.1.1

[6]The orientation of a hypergraph $G = (V, E)$ is an injection $f : E \to V$ with $f(e) \in e$ for all $e \in E$.

operation. Since every edge (equation) is associated with a *unique* vertex (variable), it implies every input key can be injectively assigned an integer in $\{0, 1, ..., \lambda n - 1\}$. Chazelle et al. (2004) were first to exploit orientability: their proposed technique[7] stores the function $f : S \rightarrow \{0, 1, ..., r-1\}$, and the goal differs slightly from MWHC in that it tries to find an array of variables g_i, such that for every key $x \in S$,

$$g_{h_0(x)} + g_{h_1(x)} + g_{h_2(x)} = f(x) \mod r$$

The function f stores the index $j \in \{0, 1, ..., r-1\}$ of the hash function h_j where $h_j(x)$ identifies which of the r associated vertices is the one assigned to the edge. This gives rise to a PHF of the form $h_{f(x)}(x) : S \rightarrow [\lambda n]$; the construction can be stored in $\lambda n \lceil \log r \rceil$ bits. Botelho et al. (2007) subsequently proposed the BDZ algorithm, which additionally uses a ranking structure to make the PHF minimal.

2.2.1 Application: AMQ filters

An xor filter, represented as an array g, is designed for queries over an immutable set $S \subseteq U$. The membership test for some key x calculates the fingerprint of the key $F(x)$, then computes three hash function values $h_0(x), h_1(x), h_2(x)$, and finally checks if the expected fingerprint of x equals $F(x)$, i.e.

$$g[h_0(x)] \oplus g[h_1(x)] \oplus g[h_2(x)] = F(x)$$

[7]Interestingly, although the technique is fundamentally the same as MWHC, its presentation in (Chazelle et al., 2004, section 3.3) makes no connection to random hypergraphs and PHFs

An xor filter construction follows the BDZ algorithm, which builds peelable, 3-uniform, random hypergraphs. As described in section 2.1, to ensure the hypergraph is peelable with high probability, it is necessary to configure an edge-density below the critical threshold. In the xor filter context, this is achieved by ensuring the filter is stored in an array with capacity larger than the number of keys n in S; in this case it is set to $1.23n$.

Interestingly, by changing the type of hypergraph used in the BDZ algorithm to what is known as a *fuse* graph (Dietzfelbinger & Walzer, 2019), it is possible to reduce the space overhead from $1.23n$ to $1.12n$. Graf & Lemire (2022) subsequently introduced a variant of the xor filter, called binary fuse filters, which exploits this fact.

2.3. Data parallel primitives

Data parallel primitives (DPPs) were systematically conceptualized in Blelloch (1990). Exploring the power of data-parallel programming, the work defined a small set of parallel primitives which operate on vectors[8] of data, and are used as building blocks for expressing parallel algorithms. Each primitive satisfies two efficiency criteria: First, it must run in $O(n)$ time on a serial random access machine. Second it belongs to class NC^1 (Cook, 1985), i.e. on vectors of length n, the primitive is computable by a Boolean circuit of $O(\log n)$ depth and of polynomial size. Examples of primitive operations include 1. summing the elements of a vector, 2. re-arranging

[8]The term "vector" denotes a one-dimensional array

the elements of a vector by a second vector of indices, and 3. merging elements of two sorted vectors. Essentially, a DPP captures a particular pattern of how input data in a dense array is processed in parallel to produce output.

DPPs have influenced the development of high-level application programming interfaces (APIs). The Thrust (Bell & Hoberock, 2012) library abstracts primitive parallel operations such as *scan* and *reduce* as generic interfaces, while programmers specify the concrete operation ("user-defined functor") to perform. At compile-time, the generic data-parallel algorithm and functor are expanded, then optimized to yield performance. The Many-core Visualization Tool Kit (VTK-m) project (Moreland et al., 2016) — a scientific visualization library — builds its APIs on top of Thrust library methods, and has in turn fueled considerable study of DPPs in parallel visualization applications.

Here we list and describe fundamental primitives which serve as building blocks for some of the the parallel algorithms introduced in this thesis.

Map. For every input element in a vector, map applies a single operation, then stores the computed result in an output vector of the same length at the same index. In Thrust, the `transform` and `for_each` APIs perform the parallel map.

Scan. Scan (Blelloch, 1990) — also known as the all-prefix-sums operation — takes a binary associative operator $\oplus$ with identity i, and an ordered set $[a_0, a_1, \ldots, a_{n-1}]$ of n elements, and returns the ordered set $[i, a_0, (a \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2})]$. There are two versions of scan: The above refers to an exclusive scan, because each element j in the result is the sum of all elements up to but not including j in the input array; for an inclusive scan, all elements including j are summed (Harris et al., 2007).

Uses of scan include 1. stream compaction — an important parallel primitive in many general-purpose applications — and 2. radix sort. The Thrust APIs `copy_if` and `sort_by_key` encapsulate stream compaction and radix sort routines respectively.

Reduce. Reduce is a generalization of summation: it takes an input vector of values and combines all elements of the vector using some binary operator. In (Blelloch, 1990), the binary operators are one of $+$, `maximum`, `minimum`, `or`, or `and`. Implementations such as Thrust allow other user-defined sum operators, but requires the operator to be associative for parallelizability.

Stream compaction. Stream compaction is a filtering operation: Given an input vector v and a predicate p, the operation selects only elements in v for which $p(v_i), i \in [|v|)$ is true, and packs the result into a dense output vector. The output preserves the ordering of input elements (Harris et al., 2007). In known implementations, the operation requires a scan followed by a scatter (Horn, 2005).

Radix sort. Radix sort assumes input keys as d-digit numbers, and sorts keys over iterations, one digit per iteration, from least to most significant. For each of the d passes, the routine performs a stable counting sort: Given the current digit of each key, the number of keys with smaller digits plus the number of keys with same digits but occurring earlier in the input sequence is computed. This yields the *rank*, or the output index where the key should be relocated. Each pass completes with relocation of keys to the computed indices. The final sort order is output when all passes are complete. In known implementations, the inherent parallelism of radix sort is exploited by reducing the counting sort in each iteration to the all-prefix-sums operation (i.e. scan) (Harris et al., 2007; Satish et al., 2009)

Chapter III.

Algorithm Design and Analysis

3.1. The BDZ algorithm

The BDZ algorithm constructs PHFs on key set sizes that fit in internal memory. It is a randomized algorithm that succeeds when a random, acyclic, r-partite[1], r-uniform hypergraph can be generated from the input set of keys $\in S$. We state the definition of an acyclic hypergraph.

Definition 1. An r-graph (i.e. r-uniform hypergraph) is acyclic if and only if some sequence of repeated deletions of edges containing vertices of degree 1 yields a graph with no edges (Czech et al., 1997).

The goal of BDZ is to solve an assignment problem, defined here for $r = 3$: For a 3-uniform hypergraph $G = (V, E)$, $|V| = m$, $|E| = n$, $m > n$, construct an array g so that the following function $f : E \to [0, m-1]$ is a perfect hash function

[1]Botelho et al. (2012) prove that r-partite hypergraphs constructed using random hash functions h_i with codomain $\left[\frac{i|V|}{r}, \frac{(i+1)|V|}{r}\right)$ retain the same peelability thresholds as r-hypergraphs using hash functions with codomain $\left[0, |V|\right)$.

on E:

$$f(e = (v_0, v_1, v_2) \in E) = \begin{cases} v_0, & \text{if } (g[v_0] + g[v_1] + g[v_2]) \mod r = 0 \\ v_1, & \text{if } (g[v_0] + g[v_1] + g[v_2]) \mod r = 1 \\ v_2, & \text{if } (g[v_0] + g[v_1] + g[v_2]) \mod r = 2 \end{cases} \tag{III.1}$$

The algorithm aims to assign to each vertex a value $\in \{0, 1, 2, 3\}$, so that for each edge $e \in E$ where e is a 3-subset of V, the sum of values associated with its three vertices modulo $r = 3$ is a *unique* value in $[0, m - 1]$. This assignment can be found, if G is acyclic. The value 3 is a special value to indicate an unassigned vertex.

BDZ consists of three steps: mapping, assigning and ranking. Only the first two steps are required to output a PHF. When the ranking step is included, the output is a MPHF. We focus on details of the mapping and assigning steps, which are sufficient to solve the assignment problem.

3.1.1 Mapping step

Figure 1 shows the pseudocode of the mapping step. The mapping step requires a set of keys S, and a set $\mathcal{H}$ of hash functions which map U into disjoint partitions[2] of V. BDZ uses an edge-oriented structure to represent hypergraphs: each edge is represented as an array of r vertices, and each vertex v has a list of edges incident to v. Line 3 initializes an empty set of edges E for the hypergraph G_r. Line

[2]From $\mathcal{H}$, the algorithm selects r hash functions h_i with ranges $\left[\frac{i|V|}{r}, \frac{(i+1)|V|}{r}\right)$, $i \in [0, r]$

```
procedure MAPPING($S, \mathcal{H}, G_r, \mathcal{L}$)
    repeat
        $E(G_r) = \emptyset$
        Select $h_0, h_1, \cdots, h_{r-1}$ uniformly at random from $\mathcal{H}$
        for each $x \in S$ do
            $e = \{h_0(x), h_1(x), \ldots, h_{r-1}(x)\}$
            addEdge($G_r, e$)
        end for
        $\mathcal{L}$ = isAcyclic($G_r$)
    until $E(G_r)$ is empty
end procedure
```

Figure 1: The mapping step. Adapted from Botelho et al. (2013)

9 tests if G_r is acyclic: edges which contain vertices of degree 1 are iteratively deleted, and the order of deletion is stored in $\mathcal{L}$. In BDZ, this test is done with an iterative peeling process on G_r, as described in step 2 of figure 2.

1. Initialize an empty queue Q, and traverse G_r. For any edge containing a vertex of degree one, store the edge in Q.
2. While Q is not empty, dequeue an edge e from Q, remove e from G_r, and store e in $\mathcal{L}$. When removing e from G_r, check if any of the associated vertices now become degree one. If true, enqueue the edge which contains the degree one vertex.

Figure 2: Algorithm to test if a r-graph is acyclic. Adapted from Botelho et al. (2013)

The list of edges $\mathcal{L}$, obtained at the end of the peeling process, stores the removed edges in order of removal, i.e. the first edge in $\mathcal{L}$ is the first one removed, the second edge is the second one removed, and so on.

3.1.2 Assigning step

Figure 3 shows the pseudocode of the assigning step. This step requires the list of edges $\mathcal{L}$ as input, and returns an assignment of values to the vertices of G_r,

```
procedure ASSIGNING($G_r, \mathcal{L}, g$)
    for $u = 0$ to $m - 1$ do
        $Visited[u]$ = false
        $g[u] = r$
    end for
    for $i = |\mathcal{L}| - 1$ to $0$ do
        $e = \mathcal{L}[i]$
        $sum = 0$
        for $k = r - 1$ to $0$ do
            if not $Visited[e[k]]$ then
                $Visited[e[k]]$ = true
                $u = e[k]$
                $j = k$                                ▷ $j$ is the index of $u$ in $e$
            else
                $sum$ += $g[e[k]]$
            end if
        end for
        $g[u] = (j - sum) \mod r$
    end for
end procedure
```

Figure 3: The assigning step. Adapted from Botelho et al. (2013)

represented as an array g containing values in the range $[0, r]$.

The algorithm first initializes $g[i] = r$, and a boolean array $Visited[i] = false$ for $i \in \{0, m-1\}$. The value r marks every vertex of G_r as unassigned, and $Visited$ flags which vertices have been visited. The list of edges $\mathcal{L}$ is then traversed *in reverse*[3] from the last peeled edge to the first. For each $e \in \mathcal{L}$, the algorithm finds the first vertex u of e that is not yet visited, and stores in j the index of u in e, where $0 \leq j \leq r-1$. Line 18 performs assignment, setting $g[u] = (j - \sum_{v \in e \wedge Visited[v]=true} g[v])$ mod r. Also, whenever an edge e is considered, the algorithm passes through every vertex u in e, and any u not yet visited is set to $Visited[u] = true$.

[3]The traversal of edges $\in \mathcal{L}$ in reverse order (of removal from G_r) ensures every edge contains at least one vertex that is visited for the first time.

3.2. Analysis

In this section, we analyze the BDZ algorithm for sources of parallelism, specifically in the mapping and assigning steps.

Edge insertion. When a key from the input set S is considered, r hash computations are performed to yield the edge $\{h_0, h_1, \ldots, h_{r-1}\}$, and the edge is added to some hypergraph representation. Since the corresponding edge of each key can be independently computed in memory, edge creation appears to be parallelizable. However, when adding edges to the hypergraph, resource contention is likely to occur; this is because any two edges being added may contain overlapping vertices. A hypergraph representation that stores the associations between edges and vertices will likely be required to handle concurrent access to the same vertices, so that parallel insertions can be supported. We note the idea (of adding hypergraph edges in parallel) is feasible in practice and not new; it was proposed and implemented for execution on a GPU by Jiang et al. (2013) to build the hypergraph associated with an IBLT (this data structure was described previously in section 2.1.2).

Peeling. In the BDZ algorithm, the generated hypergraph is r-partite: random hash functions h_i with codomain $\left[\frac{i|V|}{r}, \frac{(i+1)|V|}{r}\right)$ — instead of $[0, V)$ — are used to create the hypergraph edges $h_0(x), h_1(x), \ldots, h_{r-1}(x)$. This structural difference from uniformly random hypergraphs appears to be negligible, as Botelho et al. (2012) prove that r-partite random hypergraphs share the same peelability thresholds. An advantage of the r-partite property is that by construction, $h_i(x) \neq h_j(x)$ for $i \neq j$, and so degenerate edges (edge joining a vertex to itself) cannot be created. This appears to reduce the number of trials needed to find a peelable hypergraph and in practice, experiments indicate only one trial is needed (Belazzougui et al., 2014).

The acyclicity test in the BDZ algorithm executes a peeling operation that finds the 2-core of the associated hypergraph. This peeling operation is shown to be parallelizable, but the naive round-based algorithm (i.e. in each round, all vertices of degree one and their incident edges are removed in parallel from the hypergraph) is known to negatively affect peeling algorithm correctness. Jiang et al. (2013) point out that in the naive case, since an edge e is associated with r vertices, $r \geq 2$, there can be more than one degree-1 vertex among the r vertices. With each vertex processed in parallel, these degree-1 vertices and their associated edge e would be peeled simultaneously, leading to e being peeled *multiple* times from the hypergraph. To solve this issue, the authors proposed a technique called "parallel peeling with subtables", where the hypergraph is split r partitions, and for every edge e, exactly one of its vertices exists in each partition. Each peeling round is split into sequential iteration through r subrounds, and in each subround i, the ith subtable is processed in parallel. This ensures that every edge e is peeled from the hypergraph at most once: when a degree-1 vertex is found in some partition, the other vertices associated with e are removed from the other partitions. Hence, it seems fortuitous that the hypergraph generated in the BDZ algorithm is also r-partite; it is possible to employ the same technique by Jiang et al. (2013) to achieve a correct, and parallel peeling algorithm for the acyclicity test.

Layers. The round-based parallel peeling algorithm described so far would partition the edge set E into groups of edges that we refer to as *layers* (borrowing the same term from Belazzougui et al. (2014)). A layer is produced for each subround, and all edges which were peeled in the same subround belong to the same layer. When peeling terminates, a sequence of layers is obtained. As noted by Bellazzougui et al (Belazzougui et al., 2014, section 4.2), concatenating this sequence of layers produces a valid peeling order (i.e. $\mathcal{L}$) as required for the assigning step. This is due to the fact that 1. multiple valid peeling orders exist, and 2., the ordering of edges within a layer is irrelevant towards obtaining a valid peeling order.

Assignment. We first state how behaviors in the assigning step (figure 3) ensures correctness: 1. the list of edges $\mathcal{L}$ obtained from the mapping step is traversed in the reverse peeling order, and 2. when the algorithm passes through a vertex u from some edge e being processed, if u is unvisited, then it is marked visited. The first behavior ensures that when some edge is considered, there exists some vertex v of degree-1 at the point when its associated edge was peeled (i.e. v is assignable). The second behavior ensures that when a vertex has been visited, it is never modified again, since only unvisited vertices are considered for assignment. Consider a proposed *parallel* assigning step (figure 4):

We claim that this parallel algorithm ensures the two conditions for correctness described. First, by construction, all edges belonging to a layer were peeled away

1. Assume a list of peeled edges $\mathcal{L}$ ordered layer-wise, i.e. the first layer of edges are the edges peeled in the first subround, the second layer of edges are the edges peeled in the second subround, and so on. Also assume a mapping $\mathcal{M}$ exists such that for every edge e, it is possible to lookup the vertex u of degree-1 which marked the edge for peeling in a particular subround.
2. Iterate through $\mathcal{L}$ one layer at a time, from the *last* layer to the first. In each iteration, process all edges belonging to the layer in parallel. When processing an edge e, query $\mathcal{M}$ to determine the degree-1 vertex u, and assign $g[u] = (j - sum) \bmod r$, where j is the index such that $h_j(e) = u$, and $sum = \sum_{v \in e \wedge v \neq u} g[v]$

Figure 4: Layer-wise parallel assigning

because one degree-1 vertex for each of these edges was found in the same partition of the r-partite hypergraph in a particular peeling subround. This satisfies the first condition. Second, by using the mapping $\mathcal{M}$, the vertex u assigned is the *same*, globally unique vertex which marked the associated edge for peeling in a particular subround. Since u is unique, and we apply the same operation to all edges, the second condition is satisfied for every edge processed. The sequential assigning step lacks this guarantee due to the loop in figure 3, line 9 which finds the first unvisited vertex u for assignment, and so needs an additional $Visited$ boolean array to mark vertices as visited.

Note that when values are gathered for assignment to vertices in parallel, multiple threads may gather values from the same vertex $v, v \neq u$. We claim that this case of concurrent access is safe: since vertex v belongs to a different partition of the hypergraph from u, v will never be assigned in the current and subsequent iterations (i.e. immutable), and hence we do not expect any additional handling for concurrency needed in a parallel implementation.

Chapter IV.

Implementation

In this chapter, we describe a GPU-accelerated construction in the case of $r = 3$, i.e. 3-partite random hypergraphs are used in the construction.

4.1. Data structures

Hypergraph representation. Given a set S of n keys, we initialize a representation of a 3-partite, random hypergraph G of m vertices by storing it as an array of size m, where each cell location corresponds to a unique vertex. Each cell consists of a key field, a counter field, and a layer field. We assume it is possible to find three random hash functions h_0, h_1, h_2 which map elements in U to integers in $[0, \lfloor \frac{m}{3} \rfloor)$, $[\lfloor \frac{m}{3} \rfloor, \lfloor \frac{2m}{3} \rfloor)$, $[\lfloor \frac{2m}{3} \rfloor, m)$ respectively. These hash functions will be used to add or remove edges from the G.

Add and remove operations. The add or remove operation[1] for an edge $e = \{h_0(e), h_1(e), h_2(e)\}$ looks up the cell locations $h_0(e), h_1(e), h_2(e)$, and XORs the key field of the r corresponding cells with e. This XOR technique is used in Belazzougui

[1]With the XOR technique, the add and remove operations are identical in implementation

et al. (2014), and is advantageous for several reasons: to store an incidence list for each vertex (i.e. the edges incident to the vertex), the typical conventional approach is to use linked lists storing incident edges. With the XOR technique, only a single fixed-size key field is required, and therefore translates to space-savings. Furthermore, experiments (Belazzougui et al., 2014) show that for input key sizes where the induced hypergraph fits in available internal memory [2], peeling algorithms using this technique execute substantially faster.

Vertex degree. The degree of each vertex is represented throughout the algorithm execution by the counter field of each cell. An insert or remove operation for an edge e looks up the cell locations $h_0(e), h_1(e), \ldots, h_{r-1}(e)$, and increments or decrements respectively the counter field of the r corresponding cells. We refer to a cell with a counter field value of one as a "pure" cell, borrowing the term from Jiang et al. (2013).

Vertex layer. We count the number of peeling subrounds completed so far in the Mapping step, and since each subround induces a new logical layer (of edges), the count corresponds to the cumulative number of layers produced so far in the execution. When an edge is removed from the hypergraph (during peeling), the count of peeling rounds completed is stored in the layer field of the single cell representing a degree-1 vertex found by the peeling process. The value of the layer field acts as a grouping variable, so that in the Assigning step, the algorithm is able to find all cells with the same value (belonging to the same layer), and execute the assignment operation for

[2]The BDZ algorithm, is designed for computations within internal memory

these cells in parallel.

4.2. GPU-based implementation

Note on notation. The pseudocode we use to describe the parallel algorithms comprise of standard sequential code constructs, except for the PARFOR construct to express parallelism. This construct is based on a draft document by Blelloch et al. (2021). We give an example: in figure 5, all n elements of the array A are incremented by one, and written back to the same location. Assuming an unbounded number of processors, each array element $\in A$ is allocated to a processor, which executes the body of the PARFOR iteration in parallel with all other processors.

```
parfor $i$ from 0 to $|A| - 1$ do
    $A[i] \leftarrow A[i] + 1$
end parfor
```

Figure 5: Pseudocode example using the parfor construct

Overview. From our analysis in the preceding chapter, there are three components in the algorithm which appear to have inherent parallelism: adding edges to the hypergraph, peeling edges from the hypergraph, and assigning each edge to a unique vertex. We describe how these components are implemented for parallel execution, under the practical constraints of what GPUs can achieve.

4.2.1 Adding edges

To add edges to the hypergraph in parallel, as in figure 6, we allocate a GPU thread for every key $x \in S$ to be added. Each GPU thread independently computes

```
1: parfor each $x \in S$ do
2:     $e \leftarrow \{h_0(x), h_1(x), \ldots, h_{r-1}(x)\}$
3:     addEdge$(G, e)$
4: end parfor
```

Figure 6: Adding edges in parallel

the corresponding edge $\{h_0(x), h_1(x), h_2(x)\}$. The `addEdge` operation then XORs the key field of the corresponding cells with x. Since multiple threads may try to update the same cell, we use atomic XOR operations in `addEdge` to handle concurrent updates. Each time the key field is updated in a cell, the degree field of that cell is incremented by one.

We note that the potential amount of resource contention due to parallel operations appears feasible in practice: in the $r = 3$ case, where there are n input keys (edges) and $m = 1.23n$ vertices, on average, about $3/1.23 \approx 2.4$ keys are hashed to the same vertex.

Note on duplicate keys. A caveat here is that our implementation does not support duplicate keys in the input. For example, suppose some key x appears twice in the input, then $x \oplus x = 0$, effectively deleting the edge representing x from the hypergraph. Some preprocessing is therefore needed to remove duplicates prior to running the algorithm.

4.2.2 Peeling edges

Partitions. Due to the non-overlapping ranges of the hash functions used to map every key to an edge, the associated vertices of each edge being added are located in three physical partitions, with exactly one vertex per partition. Peeling proceeds by iterating sequentially over the partitions, and each partition is processed in parallel. A full peeling round therefore consists of three sequential subrounds.

Peeling operations. Throughout the peeling process, we track the cumulative number of subrounds via a global counter . In the ith subround, the ith partition is processed in parallel and the global counter is incremented. We allocate a GPU thread to process each array cell in the partition. Each thread tests if the cell is pure (i.e. degree field value equals 1); if true, then occurrences of the key is XOR-removed from the other two partitions, leaving exactly one occurrence in the current partition. The layer field of the cell containing this only occurrence of the key is updated with the global counter (of the number of subrounds). In all three partitions, the degree fields of the cells corresponding to vertices of the key are decremented by one.

We note an atomic operation is required when removing occurrences of a key from the hypergraph, and when decrementing the degree fields. This is because two threads removing distinct edges x, y from G may try to update the same vertex $h_i(x) = h_i(y)$ for some $0 \leq i \leq r - 1$.

Verification. The peeling algorithm terminates when no pure cells are found in a full round. To verify if the peeling process succeeded (i.e. every edge in G was peeled away), we count the number of cells containing a layer field with a non-zero value and assert if it equals n. If this test fails, the current execution is discarded, and the algorithm restarts from the point when a new set of hash functions h_0, h_1, h_2

is chosen, and a new hypergraph is created. Otherwise, in the success scenario, we have obtained an array where the n cells with *non-zero* layer field value also stores (in the key field) the value of the original input key. Taken together, such cells store the mapping $\mathcal{M}$ described in section 3.2, i.e. the mapping of the edge (key) to the degree-1 vertex which marked the edge for peeling in a particular subround.

Work efficiency. Our implementation of the parallel peeling algorithm will be less work efficient than sequential implementations. This is because every cell is examined in every round to capitalize on an expected large number of pure cells (i.e. cells with degree = 1) in each round. A sequential implementation only needs to scan the whole array once in the first round to get a starting collection of pure cells; remaining keys would be peeled by peeling from the starting collection to uncover more pure cells.

4.2.3 Assigning values to vertices

We create an integer array g of size m (equivalent to the number of vertices in G). Each element of g corresponds one-one with the cells of the array representing G, and g is intended as the output array containing values assigned to vertices.

Iteration in reverse. We iterate from the last peel layer to the first, i.e. a traversal in *reverse* peel layer order. We process G in parallel in each round, allocating a thread to process each array cell. Each thread tests if the cell's layer field matches the current iteration's counter value. If true, the thread obtains the key field of the cell, denoted x, and computes the three hash values $h_0(x), h_1(x), h_2(x)$. This yields the three array locations in G associated with the edge corresponding to x. Exactly one of these locations equals the cell's array location, denoted u. With at most three comparisons, we can determine the index j of the hash function h_j, where $h_j(x) = u$, and assign a value to $g[u]$ such that

$$g[u] = j - \sum_{v \in \{h_0(x), h_1(x), h_2(x)\} \wedge v \neq u} g[v] \mod 3$$

The assigning step is complete when all layers are processed, and we obtain a PHF of the form

$$h_{f(x)}(x) : S \to [0, m)$$

where $f(x) = g[h_0(x)] + g[h_1(x)] + g[h_2(x)] \mod 3$.

Extensibility. The assigning step can be easily extended to support other application use cases, such as xor filters (an approximate membership query filter over immutable sets). In xor filter construction, Graf & Lemire (2020) used the Mapping and Assigning steps of the BDZ algorithm, and introduced a simple modification in the Assigning step to store a different function as follows:

$$g[u] = F(x) \oplus g[v] \oplus g[w]$$

where $u, v, w \in \{h_0(x), h_1(x), h_2(x)\}$, and $v \neq u, w \neq u, v \neq w$, and $F(x)$ is a fingerprint value[3]. The membership test for some key x then requires evaluating

[3]A fingerprint is the result of a hash function h which maps a arbitrarily large input key to a much smaller bit string representation, typically a word with fixed number of bits.

whether

$$g[h_0(x)] \oplus g[h_1(x)] \oplus g[h_2(x)]$$

equals $F(x)$.

Work efficiency. Our implementation of the parallel assignment algorithm will be less work efficient than sequential implementations. Every cell is examined in every round for a matching peel layer, in order to exploit an expected large number assignable vertices (cells) in each layer. In contrast, a sequential implementation might store the list of peeled edges using a stack; popping items from the stack achieves processing of edges in reverse peeling order, and the work required is optimally linear in the number of input keys.

4.3. Using the GPU-based implementation in applications

4.3.1 Xor filter

Xor filter construction is based on the BDZ algorithm, but as Graf & Lemire (2020) observe, the speed of construction is relatively slow. Motivated by this observation, we introduce an xor filter construction that uses our GPU-accelerated BDZ algorithm. We present the pseudocode.

Algorithm 1 Xor filter construction, using GPU-accelerated BDZ

```
Require: set of keys $S$ from universe $U$
Require: fingerprint function $f$
  repeat
      Pick three hash functions $h_0$, $h_1$, $h_2$ at random, independent of $f$
  until MAP($S, h_0, h_1, h_2$) returns success with arrays $H, L$ and integer $z$      ▷ See
  algorithm 2
  $B \leftarrow$ array of size $\lfloor 1.23 \cdot |S| \rfloor + 32$, of type $k$-bit values, uninitialized
  for $i \leftarrow z$ to 1 do
      parfor $j$ from 0 to $|H| - 1$ do
          ASSIGN($i, j, B, H, L, h_0, h_1, h_2, f$)                  ▷ See algorithm 5
      end parfor
  end for
  return $B$ and hash functions $h_0$, $h_1$, $h_2$
```

Algorithm 2 Parallel Mapping, xor filter

```
Require: set of keys $S$
Require: hash functions $h_0$, $h_1$, $h_2$

  function MAP($S, h_0, h_1, h_2$)
      Let $N$ be $1.23 \cdot |S| + 32$
      $H \leftarrow$ array of sets, size $N$
      $L \leftarrow$ array of grouping variables, size $N$          ▷ See usage in algorithm 4
      parfor $x$ in $S$ do
          ADD_TO_SETS($x, H, h_0, h_1, h_2$)                        ▷ See algorithm 3
      end parfor
      $z \leftarrow 1$
      $prev \leftarrow 0$
      while $prev \neq z$ do
          $prev \leftarrow z$
          parfor $i$ from 0 to $N/3 - 1$ do
              REMOVE_FROM_SETS($i, H, L, h_0, h_1, h_2, z$)         ▷ See algorithm 4
          end parfor
          if any element in $H$ updated then $z \leftarrow z + 1$
          parfor $i$ from $N/3$ to $2N/3 - 1$ do
              REMOVE_FROM_SETS($i, H, L, h_0, h_1, h_2, z$)
          end parfor
          if any element in $H$ updated then $z \leftarrow z + 1$
          parfor $i$ from $2N/3$ to $N - 1$ do
              REMOVE_FROM_SETS($i, H, L, h_0, h_1, h_2, z$)
          end parfor
          if any element in $H$ updated then $z \leftarrow z + 1$
      end while
      $count \leftarrow 0$
      parfor $x$ in $H$ do
          if $x$ contains single key then $count \leftarrow count + 1$
      end parfor
      return success with $H$, $L$ and $z$ if $count = |S|$ else failure
  end function
```

Algorithm 3 Add to sets, xor filter

```
Require: key $x$ from input keys $S$
Require: array $H$ of sets
Require: hash functions $h_0$, $h_1$, $h_2$

  procedure ADD_TO_SETS($x, H, h_0, h_1, h_2$)
      append $x$ to $H[h_0(x)]$
      append $x$ to $H[h_1(x)]$
      append $x$ to $H[h_2(x)]$
  end procedure
```

Algorithm 4 Remove from sets, xor filter

```
Require: array $H$ of sets
Require: index $i$ in $H$
Require: array $L$ of grouping variables; $L[i]$ is the logical group assigned to $H[i]$
Require: hash functions $h_0$, $h_1$, $h_2$
Require: integer $z$, a group value assigned to $L[i]$ if $H[i]$ contains single key

  procedure REMOVE_EDGE($i, H, L, h_0, h_1, h_2, z$)
      if $H[i]$ contains single key then
          Let $x$ be the single key value
          $L[i] \leftarrow z$
          for $i$ from 0 to 2 do
              if $H[h_i(x)] \neq x$ then
                  remove $x$ from $H[h_i(x)]$
              end if
          end for
      end if
  end procedure
```

Algorithm 5 Assign, xor filter

```
Require: integer $i$, grouping variable for the current procedure call
Require: array $B$ of $k$-bit values
Require: array $H$ of sets
Require: index $j$ in $H$
Require: array $L$ of integers, corresponding to elements in $H$
Require: hash functions $h_0$, $h_1$, $h_2$
Require: fingerprint function $f$

  procedure ASSIGN($i, j, B, H, L, h_0, h_1, h_2, f$)
      if $L[j] = i$ then
          Let $x$ be the single key value in $H[j]$
          $B[j] \leftarrow 0$
          $B[j] \leftarrow f(x) \oplus B[h_0(x)] \oplus B[h_1(x)] \oplus B[h_2(x)]$
      end if
  end procedure
```

4.3.2 Fuse filter

Structure. The fuse filter (Graf & Lemire, 2022) is a variation on the xor filter. It differs from the xor filter only in the choice of the hash functions used to map input keys to edges in the generated hypergraph. In the xor filter, the three hash functions each map to a distinct partition of the 3-partitioned array representing vertices of the hypergraph. In the fuse filter, the hash function first chooses a starting segment, then picks three distinct locations, one in each of three consecutive segments (a segment in this case is much smaller than a partition in the xor filter). This induces what is known as a *fuse* graph (Dietzfelbinger & Walzer (2019)). Consequently, the array is physically divided into many smaller, disjoint equal-sized segments — typically hundreds of segments. Intuitively, this choice of hash functions increases the locality of a key's hash locations and makes it less likely any key is hashed to locations at extreme ends of the array.

Tradeoffs. Since the generated hypergraph is now a fuse graph, and not a standard 3-uniform random hypergraph, the edge-density threshold for peelability is increased such that for n input keys, the required array size for successful construction with high probability improves to $1.125n$ (Graf & Lemire, 2022) compared to $1.23n$ in the xor filter, i.e. a reduced space overhead. However the peeling process takes a longer time: keys hashed toward the extreme locations tend to be peeled first, while those in the middle survive more rounds of peeling. This is disadvantageous for efficiency in two ways: one, more peeling rounds incur more communication overhead between the CPU and GPU (the GPU only accelerates per-loop parallel processing, while the CPU orchestrates the loop iteration). Two, with edges peeled over more rounds, the number of edges peeled per round must decrease, which reduces per-loop parallel processing efficiency.

Challenges. The change to the number of segments (partitions) in the array presents interesting challenges for a parallel implementation. We present a strategy for a GPU-accelerated implementation, based on use of data-parallel primitives in various processing stages. We provide an outline in table 1, and proceed to describe the use of parallel primitives in the implementation.

	Stages	Parallel primitive/s used
1	Mapping step	Stream compaction
2	Reduction by peel layer	Sorting (radix sort), Reduce
3	Assigning step	n.a.

Table 1: Key steps in fuse filter construction

Stream compaction. Given the segmentation of the array into hundreds of segments, it is inefficient to iterate sequentially by segment (and process each segment in parallel) since the iterations become a significant sequential bottleneck.

We claim the proportion of cells which could be peeled in parallel is equivalent to the xor filter, i.e. ≈ 1 in 3. This follows because for any given segment s, we risk peeling the same key twice[4] only if the algorithm tries to examine (and perform the XOR-remove operation on) cells simultaneously from any two consecutive neighbor segments of s. Hence if we build a partition consisting of every third segment, then

[4]This problem was discussed previously in section 3.2

all cells in this partition can be processed in parallel, just as in the xor filter case. We frame the following problem:

Given a sparse array, select only interesting elements and pack these into a dense output array.

The goal in our case is to obtain three partitions p_0, p_1, p_2 such that the concatenation of these partitions is a copy of the full array (elements not necessarily in the original order). We describe an algorithm to build each partition p_j:

1. With the array as input, treat segments as elements
2. While there are elements to iterate: for each element we apply the predicate $i \bmod 3 = j$, where i denotes the element index
3. Copy the element to p_j only if predicate returns true.

Figure 7: Algorithm to pack interesting elements into a dense array

As it turns out, such problems are amenable to GPU-acceleration with stream compaction, an important parallel primitive in applications such as collision detection and sparse matrix compression (Harris et al., 2007). For implementation, we use the `copy_if` method from CUDA Thrust library, which also preserves the relative order of input elements in the output partitions. This order-preserving property ensures we can work backwards with simple arithmetic to derive the position of the element in the original array.

Note that these partitions become inputs to the peeling algorithm, and a GPU thread is allocated to each cell in the partition (instead of the original array). When a pure cell is found in the partition, we peel the edge from the cell in the partition, and from the cell in the original array so as to maintain same information in both copies of data. When peeling is complete, the partitions are discarded.

MapReduce use-case. In our parallel peeling algorithm, when a pure cell is encountered, the thread updates the layer field of the cell with the current layer value (the current layer corresponds to the number of peeling subrounds so far in the program execution). On successful completion of peeling (i.e. all $|S| = n$ edges were peeled away), n cells in the array contain some non-zero value in the layer field, but these cells are scattered across the array in non-contiguous locations. Suppose we wanted to process cells having the same layer field value in parallel, then the entire array has to be processed, a GPU thread is allocated for every cell in the array, and the thread must test if the layer field value of the cell matches the targeted layer value.

We see a natural use case for the MapReduce paradigm to reduce the computational working set. Informally, MapReduce algorithms take as input a set of key-value pairs, and operations consist of three stages: MAP, SHUFFLE and REDUCE. We can think of the BDZ algorithm's Mapping step as analogous to the MAP stage: the hypergraph peeling algorithm on successful termination, yields a set of key-value pairs $\langle k, v\rangle$ where k is the value of a cell layer field and v is the corresponding cell. For the SHUFFLE stage, we sort the set of key-value pairs by key (i.e. layer) descending (since we want to iterate by reverse peel layer order), and output the sorted array. In the REDUCE stage, for each key k in the sorted array, we return the count of the number of cells that has layer field value equal to k. Note that this is in fact a histogram that shows the frequency count of cells mapped per layer.

In SHUFFLE, we can exploit two features in the data: (1) the number of layers is small[5] in relation to the input size n and (2) peel layers are naturally represented as positive integers. Thus we can make use of faster non-comparative sorting algorithms, i.e. radix sort by peel layer to achieve asymptotic $O(\omega n)$ complexity, where ω denotes number of digits in the largest peel layer. In our implementations of SHUFFLE and REDUCE, we use the CUDA Thrust library methods `sort_by_key` and `reduce_by_key` respectively.

Work-efficient Assigning step. A parallel assigning step that is work-efficient performs work linear in the number of input keys n. To do this, we use (1) the array sorted by descending value of the layer field value, and (2) the histogram of frequency of cells per layer in the following way: construct a processing loop where in every iteration (layer), the collection of contiguous cells in the sorted array to be processed is found by looking up the histogram value, and the sorted array is marked with start and end indices corresponding to the collection size. The CPU only sends to the GPU the specified collection of cells, which the GPU then processes in parallel.

We present the pseudocode of parallel fuse filter construction in the following pages. Names of key methods are suffixed with "v2" to distinguish the names from the previously-introduced xor filter construction.

[5]In our tests, the number of peel layers totaled 1000-1100 for an input size between 10^7 and 10^8 keys.

Algorithm 6 Fuse filter construction, using GPU-accelerated BDZ

```
Require: set of $n$ keys $S$ from universe $U$
Require: fingerprint function $f$
  Let $B \leftarrow$ an array of size approx. $1.125n$ of $k$-bit values, initialized to 0, divided into $2^{\lfloor \log_{3.33} n + 2.25 \rfloor}$ segments
  repeat
      Pick three hash functions $h_0$, $h_1$, $h_2$ at random, independent of $f$ such that $h_0(x)$, $h_1(x)$, $h_2(x)$ occupy locations in three distinct consecutive segments of $B$
  until MAP_V2$(S, h_0, h_1, h_2)$ returns success with array $H$, array $L$ and integer $z$   ▷ See algorithm 7

  Sort $H$ by key, where key$(H[i]) = L[i]$

  Reduce $H$ by key, where key$(H[i]) = L[i]$. Reduce operation counts number of keys in each group of consecutive equal keys. Return array of (key, value) pairs $T$

  Let $d \leftarrow 0$                                        ▷ offset index from start of $V$
  for $i \leftarrow 0$ to $z - 1$ do
      Let $(k, v) \leftarrow T[i]$
      parfor $j \leftarrow d$ to $v - 1$ do
          ASSIGN_V2$(j, B, H, L, h_0, h_1, h_2, f)$            ▷ See algorithm 10
      end parfor
      $d \leftarrow d + v$
  end for
  return $B$ and hash functions $h_0$,$h_1$,$h_2$
```

Algorithm 7 Parallel Mapping, fuse filter

```
Require: set of keys $S$
Require: hash functions $h_0$, $h_1$, $h_2$

  function MAP_V2($S, h_0, h_1, h_2$)
      Let $N$ be $1.125 \cdot |S|$
      $H \leftarrow$ array of sets, size $N$
      $L \leftarrow$ array of grouping variables, size $N$            ▷ See usage in algorithm 9
      parfor $x$ in $S$ do
          ADD_EDGE($x, H, h_0, h_1, h_2$)                              ▷ See algorithm 8
      end parfor
      Let $A_0$, $A_1$, $A_2$ be arrays of sets, each array of size $\approx N/3$, initially empty
      Do stream compaction of $H$ into three arrays: For each element $H[i]$, if $i$
  mod $3 = p$, pack the element into $A_p$
      $z \leftarrow 1$
      $prev \leftarrow 0$
      while $prev \neq z$ do
          $prev \leftarrow z$
          parfor $i$ from 0 to $|A_0| - 1$ do
              REMOVE_EDGE($i, A_0, H, L, h_0, h_1, h_2, z, A_1, A_2$)         ▷ See algorithm 9
          end parfor
          if any element in $H$ updated then $z \leftarrow z + 1$
          parfor $i$ from 0 to $|A_1| - 1$ do
              REMOVE_EDGE($i, A_1, H, L, h_0, h_1, h_2, z, A_0, A_2$)
          end parfor
          if any element in $H$ updated then $z \leftarrow z + 1$
          parfor $i$ from 0 to $|A_2| - 1$ do
              REMOVE_EDGE($i, A_2, H, L, h_0, h_1, h_2, z, A_0, A_1$)
          end parfor
          if any element in $H$ updated then $z \leftarrow z + 1$
      end while
      $count \leftarrow 0$
      parfor $x$ in $H$ do
          if $x$ contains single key then $count \leftarrow count + 1$
      end parfor
      return success with $H$, $L$ and $z$ if $count = |S|$ else failure
  end function
```

Algorithm 8 Add edge, fuse filter

```
Require: key $x$ from input keys $S$
Require: array $H$ of sets
Require: hash functions $h_0$, $h_1$, $h_2$

  procedure ADD_EDGE_V2($x, H, h_0, h_1, h_2$)
      append $x$ to $H[h_0(x)]$
      append $x$ to $H[h_1(x)]$
      append $x$ to $H[h_2(x)]$
  end procedure
```

Algorithm 9 Remove edge, fuse filter

```
Require: arrays $A, A', A''$ containing sets. Derived from $H$ by stream compaction
Require: array $H$ containing sets
Require: index $i$ in $A$
Require: array $L$ of grouping variables. Each element in $L$ and its corresponding
  element in $H$ constitute (key, value) pairs
Require: hash functions $h_0$, $h_1$, $h_2$
Require: integer $z$, a group value

  procedure REMOVE_EDGE_V2($i, A, H, L, h_0, h_1, h_2, z, A', A''$)
      if $A[i]$ contains single key then
          Let $x$ be the single key value
          Let $b$ be the corresponding index in $H$ where $H[b] = x$
          $L[b] \leftarrow z$
          for $j$ from 0 to 2 do
              if $H[h_i(x)] \neq x$ then
                  remove $x$ from $H[h_i(x)]$
              end if
          end for
          Let $u$, $v$ be the two other indexes in $A'$, $A''$ respectively where $x$ is located
          Remove $x$ from $A'[u]$, $A''[v]$                    ▷ Keep data in sync with $H$
          Remove $x$ from $A[i]$
      end if
  end procedure
```

Algorithm 10 Assign, fuse filter

```
Require: integer $i$, group value
Require: array $B$ of $k$-bit values
Require: array $H$ of sets
Require: index $j$ in $H$
Require: array $L$ of grouping variables. Each element in $L$ and its corresponding
  element in $H$ constitute a (key, value) pair
Require: hash functions $h_0$, $h_1$, $h_2$
Require: fingerprint function $f$

  procedure ASSIGN_V2($i, j, B, H, L, h_0, h_1, h_2, f$)
      if $L[j] = i$ then
          Let $x$ be the single key value in $H[j]$
          $B[j] \leftarrow 0$
          $B[j] \leftarrow f(x) \oplus B[h_0(x)] \oplus B[h_1(x)] \oplus B[h_2(x)]$
      end if
  end procedure
```

Chapter V.

Evaluation

In this chapter, we benchmark the construction times of two applications which we modified to use our proposed parallel BDZ algorithms: 1. the xor filter, modified to use the GPU-accelerated BDZ algorithm, and 2. the fuse filter, modified to use the GPU-accelerated BDZ algorithm variant[1]. We compare against the sequential, reference implementations (Graf & Lemire, 2020, 2019) of these filters to gauge the amount of speedup obtained by our parallel implementations.

5.1. Results

Setup. We implemented GPU code with CUDA version 10.1. and ran all implementations on an Amazon EC2 *g3s.xlarge* instance equipped with a Nvidia Tesla M60 GPU having 8GB of device memory. This GPU unfortunately did not support 64-bit xor atomic operations, so we split up 64-bit data fields into two 32-bit fields. While this adds some computational overhead, we do not expect significant

[1]We state for clarity that the BDZ algorithm variant peels a *fuse* hypergraph instead of the standard 3-uniform random hypergraph.

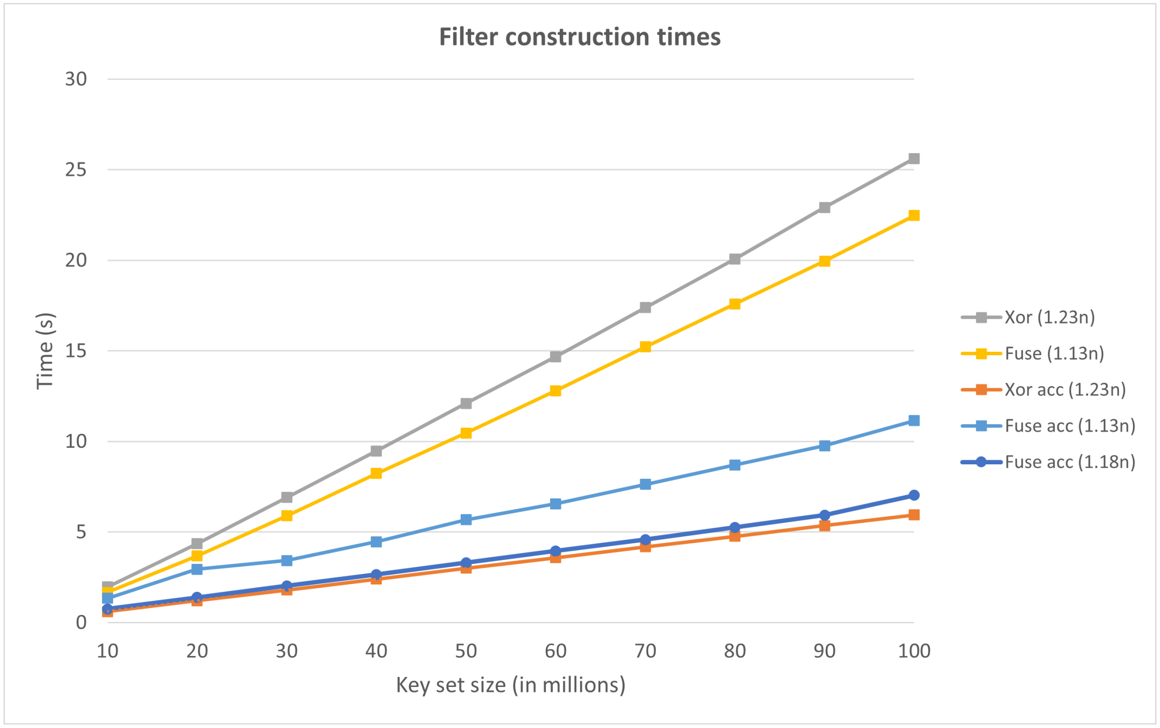

Figure 8: Construction time. Sequential and GPU-accelerated implementations

impact on the results. We used 64-bit integers as input key data type. Key set sizes[2] range between 10^7 to 10^8.

Speedup. We ran the filter constructions and recorded the time taken, averaged over five trials. Figure 8 shows that at the largest input sizes, xor filter construction with GPU acceleration attains speedup exceeding 4x; for fuse filter construction with GPU acceleration, the speedup is between 2x to 3x, depending on configuration (we discuss this point later in this chapter). The Amdahl effect was observed here, i.e. with increasing input size for a fixed number of processors, the maximum possible speedup increases.

[2]We note that fuse filter construction fails for key set sizes in the region of 10^5 keys, hence we set the minimum at 10^7 keys. This (higher) minimum key set size requirement seems to be a known property of fuse filter implementations (Docs.rs, 2020).

Type	**Time(%)**	**Time**	**Calls**	**Avg**	**Min**	**Max**	**Name**
GPU	45.36%	149.77ms	93	1.6104ms	1.4737ms	2.6316ms	assign
activities:	27.66%	91.345ms	1	91.345ms	91.345ms	91.345ms	insertKeys
	9.04%	29.856ms	32	933.00us	487.14us	4.1888ms	peelSet0
	9.04%	29.852ms	32	932.87us	489.06us	4.7560ms	peelSet1
	8.90%	29.380ms	32	918.13us	488.93us	5.3278ms	peelSet2
API calls:	50.71%	330.33ms	**98**	3.3708ms	494.44us	149.18ms	**cudaDeviceSynchronize**
	38.34%	249.77ms	6	41.628ms	21.862us	225.05ms	cudaMallocManaged
	7.04%	45.850ms	**190**	241.32us	6.2190us	25.369ms	**cudaLaunchKernel**
	3.82%	24.869ms	6	4.1449ms	27.889us	16.125ms	cudaFree
	0.05%	347.59us	1	347.59us	347.59us	347.59us	cuDeviceTotalMem
	0.04%	231.66us	96	2.4130us	733ns	74.589us	cuDeviceGetAttribute
	0.00%	27.343us	1	27.343us	27.343us	27.343us	cuDeviceGetName
	0.00%	3.7330us	3	1.2440us	756ns	1.8610us	cuDeviceGetCount
	0.00%	3.7010us	1	3.7010us	3.7010us	3.7010us	cuDeviceGetPCIBusId
	0.00%	2.4800us	2	1.2400us	779ns	1.7010us	cuDeviceGet
	0.00%	874ns	1	874ns	874ns	874ns	cuDeviceGetUuid

Table 2: GPU profiling statistics for xor filter, 10M keys

.

We utilized Nvidia's `nvprof` utility to obtain a breakdown of the running times in the parallel algorithms in Tables 2-5. We provide some explanations of these details. In the "GPU activities" section, the `insertKeys`, `peelSet0`, `peelSet1`, `peelSet2` methods correspond to hypergraph instantiation and peeling activities of the mapping step. The `assign` method corresponds to the assigning step, and is invoked for every *layer* in the peel ordering. We note that:

(i) Fuse filter construction time is affected by larger number of API calls to the GPU compared to its xor filter counterpart. The total number of `cudaDeviceSynchronize`[3] and `cudaLaunchKernel`[4] calls is above 3000, but for the xor filter case, this number is slightly below 300. These numbers stay relatively constant despite key set size

[3]blocking API that waits for the results of GPU computations to complete before resuming CPU code execution

[4]asynchronous API that launches kernel functions in the GPU

Type	**Time(%)**	**Time**	**Calls**	**Avg**	**Min**	**Max**	**Name**
GPU	45.63%	1.52587s	94	16.233ms	14.723ms	27.662ms	assign
activities:	27.73%	927.20ms	1	927.20ms	927.20ms	927.20ms	insertKeys
	8.92%	298.22ms	32	9.3193ms	4.8646ms	49.907ms	peelSet1
	8.91%	297.81ms	32	9.3064ms	4.8500ms	43.827ms	peelSet0
	8.82%	294.82ms	32	9.2132ms	4.8685ms	56.707ms	peelSet2
API calls:	73.82%	3.34405s	**98**	34.123ms	4.8575ms	1.52524s	**cudaDeviceSynchronize**
	12.48%	565.23ms	6	94.204ms	17.123us	235.82ms	cudaMallocManaged
	7.11%	322.23ms	**191**	1.6871ms	6.3290us	262.55ms	**cudaLaunchKernel**
	6.57%	297.72ms	6	49.620ms	34.910us	149.90ms	cudaFree
	0.01%	375.70us	1	375.70us	375.70us	375.70us	cuDeviceTotalMem
	0.01%	364.16us	96	3.7930us	735ns	179.55us	cuDeviceGetAttribute
	0.00%	29.409us	1	29.409us	29.409us	29.409us	cuDeviceGetName
	0.00%	3.7580us	1	3.7580us	3.7580us	3.7580us	cuDeviceGetPCIBusId
	0.00%	3.6810us	3	1.2270us	751ns	1.9680us	cuDeviceGetCount
	0.00%	2.3500us	2	1.1750us	771ns	1.5790us	cuDeviceGet
	0.00%	893ns	1	893ns	893ns	893ns	cuDeviceGetUuid

Table 3: GPU profiling statistics for xor filter, 100M keys

Type	**Time(%)**	**Time**	**Calls**	**Avg**	**Min**	**Max**	**Name**
GPU	67.37%	1.51532s	1053	1.4391ms	1.4012ms	1.9433ms	assign
activities:	9.25%	208.03ms	352	590.98us	474.40us	8.3381ms	peel_set0
	9.24%	207.74ms	352	590.16us	472.38us	9.3225ms	peel_set1
	9.19%	206.63ms	352	587.03us	472.54us	10.132ms	peel_set2
	4.96%	111.66ms	1	111.66ms	111.66ms	111.66ms	insert_keys
API calls:	78.23%	2.21457s	**1058**	2.0932ms	479.04us	1.47226s	**cudaDeviceSynchronize**
	10.56%	299.02ms	9	33.225ms	18.586us	237.00ms	cudaMallocManaged
	9.84%	278.55ms	**2110**	132.01us	5.9250us	35.276ms	**cudaLaunchKernel**
	1.33%	37.775ms	9	4.1973ms	45.092us	13.827ms	cudaFree
	0.01%	366.26us	96	3.8150us	734ns	183.13us	cuDeviceGetAttribute
	0.01%	355.67us	1	355.67us	355.67us	355.67us	cuDeviceTotalMem
	0.00%	60.099us	1	60.099us	60.099us	60.099us	cuDeviceGetName
	0.00%	3.6190us	3	1.2060us	752ns	2.0190us	cuDeviceGetCount
	0.00%	3.5940us	1	3.5940us	3.5940us	3.5940us	cuDeviceGetPCIBusId
	0.00%	2.1930us	2	1.0960us	759ns	1.4340us	cuDeviceGet
	0.00%	889ns	1	889ns	889ns	889ns	cuDeviceGetUuid

Table 4: GPU profiling statistics for fuse filter, 10M keys

Type	**Time(%)**	**Time**	**Calls**	**Avg**	**Min**	**Max**	**Name**
GPU	68.98%	15.7027s	1094	14.353ms	13.999ms	23.593ms	assign
activities:	9.00%	2.04795s	366	5.5955ms	4.7045ms	71.332ms	peel_set0
	8.97%	2.04201s	366	5.5793ms	4.7058ms	79.196ms	peel_set1
	8.92%	2.03022s	366	5.5471ms	4.7023ms	87.973ms	peel_set2
	4.13%	941.23ms	1	941.23ms	941.23ms	941.23ms	insert_keys
API calls:	88.57%	21.7459s	**1100**	19.769ms	4.7107ms	14.6756s	**cudaDeviceSynchronize**
	6.80%	1.67015s	**2193**	761.58us	6.0140us	248.56ms	**cudaLaunchKernel**
	2.90%	712.40ms	9	79.155ms	18.652us	249.73ms	cudaMallocManaged
	1.73%	423.56ms	9	47.062ms	39.729us	138.33ms	cudaFree
	0.00%	369.07us	96	3.8440us	730ns	184.43us	cuDeviceGetAttribute
	0.00%	353.23us	1	353.23us	353.23us	353.23us	cuDeviceTotalMem
	0.00%	35.017us	1	35.017us	35.017us	35.017us	cuDeviceGetName
	0.00%	4.3660us	1	4.3660us	4.3660us	4.3660us	cuDeviceGetPCIBusId
	0.00%	3.8490us	3	1.2830us	802ns	2.1730us	cuDeviceGetCount
	0.00%	2.2420us	2	1.1210us	747ns	1.4950us	cuDeviceGet
	0.00%	897ns	1	897ns	897ns	897ns	cuDeviceGetUuid

Table 5: GPU profiling statistics for fuse filter, 100M keys

Peeling complexity. In every peeling subround, if at least one pure cell was found (therefore marking its associated edge for peeling), we increment the layer count once per subround. The observed number of calls to `assign` (> 1000) in figure 5 for the fuse filter case indicated a large number of subrounds involved in the peeling process (calls to the `peel_set`* methods). We further note in Figure 9 that the number of layers stays remarkably constant with increasing key set size.

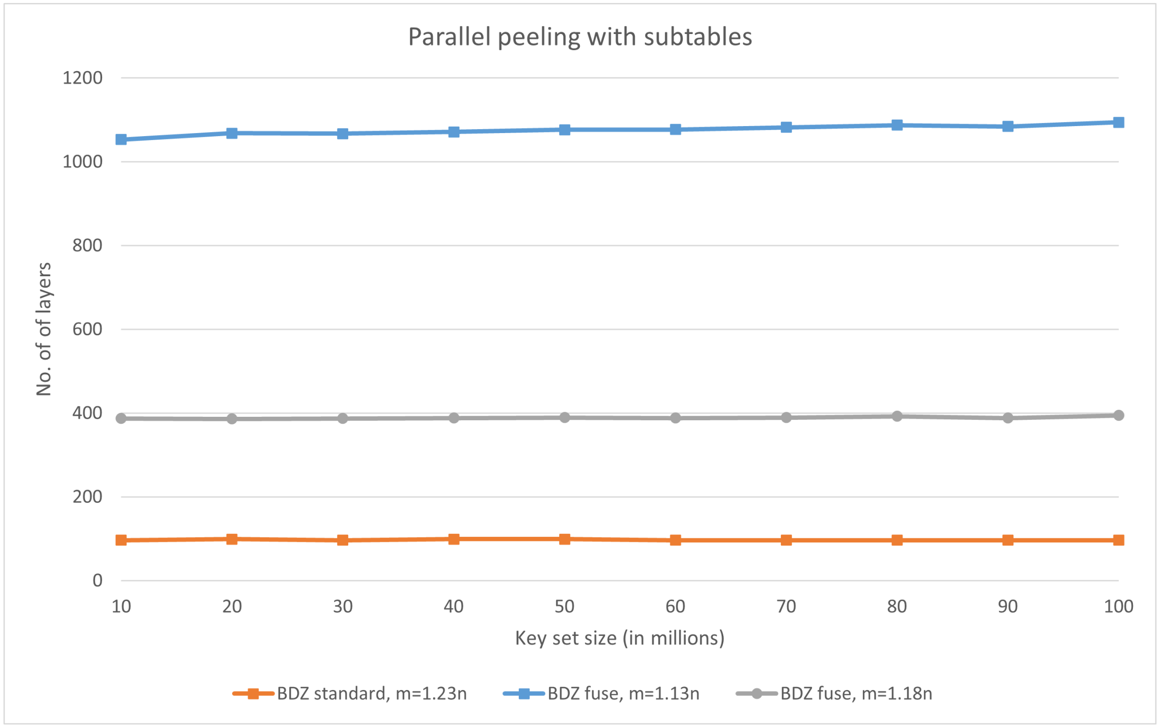

Figure 9: Count of layers generated by parallel peeling

We appeal to the following theorem introduced in Jiang et al. (2013), for our analysis:

Theorem 2. Let $r \geq 3$ and $k \geq 2$. Let $\phi_{r-1} = \lim_{k\to\infty} F_{r-1}^{1/k}(k)$ be the asymptotic growth rate for the Fibonacci sequence of order $r-1$. Let G be a hypergraph over n nodes with cn edges generated according to the following random process. The vertices of G are partitioned into r subsets of equal size, and the edges are generated at random subject to the constraint that each edge contains exactly one vertex from each set. With probability $1 - o(1)$, the peeling process for the k-core in G that uses r subrounds in each round terminates after $\frac{1}{r \log \phi_{r-1} + \log(k-1)} \log\log n + O(1)$ rounds when $c < c^*_{k,r}$.

In the xor filter case ("BDZ standard, $m = 1.23n$" in figure 9), the standard

3-uniform hypergraph constructed in the mapping step follows the random process described in Theorem 1. Given $r = 3$, we have $\phi_{r-1} \approx 1.61$. We peel to an empty 2-core, so $k = 2$. Therefore the number of subrounds is given by $\frac{1}{\log \phi_{r-1}} \log \log n + O(1) \approx 4.36 + O(1)$ for $n = 100$ million keys, and the hidden additive constant for our hypergraph instances is estimated to be between 92-94 for our current configuration $(r = 3, k = 2)$.

Hidden values. While Theorem 1 is not directly applicable to the fuse filter's fuse graph, the results suggests the presence of a hidden additive constant (or slow-growing component) for fuse graphs, with a value exceeding 1000 for our current configuration $(r = 3, number\_of\_segments = 100, k = 2)$. An analysis of the relationship between the magnitude of these constants and the associated hypergraph's edge distribution is beyond the scope of this thesis. The work of Dietzfelbinger & Walzer (2019) provides ample theoretical analysis, and already anticipates that fuse graphs will require more rounds of peeling. What we do observe from these experimental runs are that these values appears to have a substantial impact on inherent parallelism.

Tuning the gap. For the fuse graph, given n vertices and m edges, we observe that if the vertices-to-edges ratio is adjusted from 1.13 to 1.18, then the number of layers induced by the peeling process drops from 1000 to less than 400 (figure 9). This has a positive effect on parallel processing efficiency: by trading off some space savings, we observe the speedup improves from 2x to 3x over sequential implementations.

Chapter VI.

Conclusions

6.1. Summary

In this thesis, we have designed a GPU-accelerated version of the BDZ algorithm, specifically by exploiting inherent parallelism in the Mapping and Assigning steps. We provide an implementation of this parallel algorithm, using it to speed up the construction of an AMQ data structure known as the xor filter. Our results show that the construction time of xor filters can be improved by a factor of at least 3-4x. This helps to mitigate the problem that the typical xor filter construction is twice as slow as a Bloom filter construction.

We have also investigated a BDZ algorithm variant, which peels a *fuse graph* instead of the standard 3-uniform hypergraph. The fuse graph is a new family of hypergraphs proposed by Dietzfelbinger & Walzer (2019). We find that this variant appears to have less inherent parallelism; in our attempt to design a GPU-accelerated algorithm, we found it necessary to employ data-parallel primitives such as stream compaction, sort and reduce to gain some parallel processing efficiency.

We see our parallel implementation of the BDZ algorithm as an experiment with impact beyond the original problem the algorithm set out to solve (i.e. PHF construction). Since the BDZ algorithm is extensible to application contexts such as xor filter construction, performance gains from parallelism easily translate to such applications, as we have shown. In further work, we think it may be worthwhile to explore additional application scenarios where the BDZ algorithm can be extended in unique ways.

Appendix A.

Source code

The source code for this project can be found in the *github repository*.

References

Belazzougui, D., Boldi, P., Ottaviano, G., Venturini, R., & Vigna, S. (2014). Cache-oblivious peeling of random hypergraphs. In *2014 Data Compression Conference* (pp. 352–361).

Bell, N. & Hoberock, J. (2012). Chapter 26 - thrust: A productivity-oriented library for cuda. *Applications of GPU Computing Series*, (pp. 359–371).

Blelloch, G. E. (1990). *Vector Models for Data-Parallel Computing.* MIT Press.

Blelloch, G. E., Dhulipala, L., & Sun, Y. (2021). Introduction to parallel algorithms.

Botelho, F. C., Pagh, R., & Ziviani, N. (2007). Simple and space-efficient minimal perfect hash functions. Algorithms and Data Structures (pp. 139–150).: Springer Berlin Heidelberg.

Botelho, F. C., Pagh, R., & Ziviani, N. (2013). Practical perfect hashing in nearly optimal space. *Information Systems*, 38(1), 108–131.

Botelho, F. C., Wormald, N., & Ziviani, N. (2012). Cores of random r-partite hypergraphs. *Information Processing Letters*, 112(8), 314–319.

Broder, A. & Mitzenmacher, M. (2003). Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1.

Chazelle, B., Kilian, J., Rubinfeld, R., & Tal, A. (2004). The bloomier filter: An efficient data structure for static support lookup tables. *Soda '04*, (pp. 30–39).

Cook, S. A. (1985). A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1), 2–22. International Conference on Foundations of Computation Theory.

Czech, Z. J., Havas, G., & Majewski, B. S. (1997). Perfect hashing. *Theoretical Computer Science*, 182(1), 1–143.

Dietzfelbinger, M. & Walzer, S. (2019). Dense peelable random uniform hypergraphs. In *ESA*.

Docs.rs (2020). xorf::fuse8 - rust. `https://docs.rs/xorf/0.6.0/xorf/struct.Fuse8.html`.

Goodrich, M. & Mitzenmacher, M. (2011). Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2011* (pp. 792–799).

Graf, T. & Lemire, D. (2020). Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics*, 25, 1–16.

Graf, T. M. & Lemire, D. (2019). Header-only xor filter library. `https://github.com/FastFilter/xor_singleheader` commit: a6068bc75665f5bccb1b4f89b8918ecdb50af624.

Graf, T. M. & Lemire, D. (2022). Binary fuse filters: Fast and smaller than xor filters. *arXiv preprint arXiv:2201.01174*.

Harris, M., Sengupta, S., & Owens, J. (2007). Parallel prefix sum (scan) with cuda. *In GPU Gems 3. Addison-Wesley*, 39, 851–.

Horn, D. (2005). Stream reduction operations for gpgpu applications. *GPU Gems*, 2.

Jiang, J., Mitzenmacher, M., & Thaler, J. (2013). Parallel peeling algorithms. *CoRR*, abs/1302.7014.

Luby, M. G. Mitzenmacher, M., Shokrollahi, M. A., & Spielman, D. A. (2001). Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47, 569–584.

Majewski, B. S., Wormald, N. C., Havas, G., & Czech, Z. J. (1996). A family of perfect hashing methods. *The Computer Journal*, 39(6), 547–554.

Molloy, M. (2005). Cores in random hypergraphs and boolean formulas. *Random Structures & Algorithms*, 27(1), 124–135.

Moreland, K., Sewell, C., Usher, W., Lo, L.-t., Meredith, J., Pugmire, D., Kress, J., Schroots, H., Ma, K.-L., Childs, H., Larsen, M., Chen, C.-M., Maynard, R.,

& Geveci, B. (2016). Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE Computer Graphics and Applications*, 36(3), 48–58.

Pagh, R. & Rodler, F. F. (2004). Cuckoo hashing. *Journal of Algorithms*, 51(2), 122–144.

Satish, N., Harris, M., & Garland, M. (2009). Designing efficient sorting algorithms for manycore gpus. *2009 IEEE International Symposium on Parallel and Distributed Processing*, 23, 1–10.