Hash Maps in a Functional Array Language

William Henrich Due[1][0009−0003−8195−4462],
Martin Elsman[2][0000−0002−6061−5993], and
Troels Henriksen[3][0000−0002−1195−9722]

University of Copenhagen, Copenhagen, Denmark
`{widu,mael}@di.ku.dk, athas@sigkill.dk`

Abstract. We present a systematic derivation of a data-parallel implementation of two-level, static and collision-free hash maps, by giving a functional formulation of the Fredman et al. construction, and then flattening it. We discuss the challenges of providing a flexible, polymorphic, and abstract interface to hash maps in a functional array language, with particular attention paid to the problem of dynamically sized keys, which we address by associating each hash map with an arbitrary context. The algorithm is implemented in Futhark, and we characterise GPU execution performance on simple benchmark problems, where we find that our hash maps outperform conventional tree/search-based approaches. Furthermore, our implementation is compared against the state-of-the-art cuCollections library, which is significantly faster for hash map construction, and to a lesser degree for lookups. We show that the performance difference is partially due to low-level code generation limitations in the Futhark compiler, but more significantly due to the data-parallel programming vocabulary not providing the constructs necessary to express the equivalent of the algorithms used by cuCollections. We end by reflecting on the extent to which the functional array language programming model could, or should, be extended to address these weaknesses.

Keywords: functional programming, parallel programming, hash table, GPU

1 Introduction

Hash maps are a commonly used data structure for storing key-value pairs. They are also well suited for data-parallel programming, just like arrays. As an example, you can map over its values, or reduce the values to a single value.

A *functional array language* is a functional language with support for data-parallel operations on arrays, typically intended for execution on parallel hardware. When compared to general-purpose functional languages, functional array languages are often very restricted. In particular, they usually have limited support for recursive data structures and functions.

Despite their general prevalence, hash maps are not commonly seen in functional array languages. Although often used to implement certain constructs in APL dialects, in those cases the hash maps are part of the interpreter (typically written in C), and not implemented in the language itself. In this paper we describe how to implement a parallel hash map in a functional array language in a way that is asymptotically efficient, and also performs reasonably in practice. We describe how such a hash map can be derived from a functional algorithm. Concretely, we demonstrate how to implement the two-level hash set construction by Fredman et al. in a functional array language. The key challenge is to adapt the original algorithm, which uses recursion and irregular arrays[1], to a data-parallel programming model that supports neither. Furthermore, we demonstrate how to interact with hash maps using an SML-style module system, in such a way that it is possible to use the hash maps with irregular datatypes as keys.

Our specific contributions are as follows:

– We demonstrate how the two-level hash map construction of Fredman et al [9] can be expressed in a data-parallel vocabulary, which requires nontrivial segmented operations.
– We show how hash maps can be exposed in a modular and generic way in a data-parallel language, using an ML-style module system, including how to handle complicated key types such as strings.
– We perform a performance comparison demonstrating that data-parallel hash maps running on GPUs outperform data structures based on binary search on both integer and string keys, but that the achieved performance still falls short of what is possible in CUDA.

This paper is structured as follows:

– Section 2 describes a functional algorithm for the two-level hash set construction by Fredman et al. and how it can be flattened to a data-parallel algorithm. The description will use SML-like pseudo code to describe the algorithm with common array operations, which can be parallelised.
– Section 3 describes an interface for hash maps suitable for a functional array language. Interfaces are written in Futhark's ML-style module system.
– Section 4 describes the benchmarks used to compare the performance of our hash map implementation against existing alternatives in array languages written in Futhark and the cuCollections library.

2 Two-Level Hash Maps

There are many ways of implementing hash maps, but only a few are suitable for a data-parallel functional array language. A problem that has to be solved during construction of a hash map is accounting for collisions. One could imagine using a chained hash map where we have two key-value pairs to be inserted at the same index in the hash map in parallel due to a hash collision. In a low-level parallel language, such as CUDA, we could use atomic operations to insert them

[1] An irregular multidimensional array is one where the elements can differ in size, e.g., `[[1,2], [3]]` is irregular.

into the hash map in some order depending on hardware specifics. In a functional setting such a strategy is problematic, because it requires nondeterministic and impure operations. One of the possible solutions could be sorting the keys by their hashes and then inserting them in order into the hash map. Strategies based on sorting, however, are inefficient and should be avoided as the performance will drop to the performance of binary search trees or Eytzinger trees [18].

A solution to the collision problem is to use the two-level hash set construction by Fredman et al. [9] (FKS). This construction computes a hash function that is collision-free, such that we do not have to consider the insertion order of the keys. To use the algorithm efficiently on real world hardware, we need the ability to compute histograms [12] using atomic operations. These are used to efficiently determine hash collisions.

2.1 Prerequisites

This article assumes a basic familiarity with data-parallel programming and functional programming. We will use a data-parallel vocabulary that is found in Figure 1 to describe the algorithm. This is a common vocabulary found in array languages with some specialized functions.

`fst`	$: (\alpha, \beta) \to \alpha$	`or`	$: [n]\mathbf{bool} \to \mathbf{bool}$
`snd`	$: (\alpha, \beta) \to \beta$	`segor`	$: [m]\mathbf{int} \to [n]\mathbf{bool} \to [n]\mathbf{bool}$
`map`	$: (\alpha \to \beta) \to [n]\alpha \to [n]\beta$	`rep`	$: \mathbf{int} \to \alpha \to [n]\alpha$
`iota`	$: \mathbf{int} \to [n]\mathbf{int}$	`scatter`	$: [m]\alpha \to [n]\mathbf{int} \to [n]\alpha \to [m]\alpha$
`zip`	$: [n]\alpha \to [n]\beta \to [n](\alpha, \beta)$	`sort`	$: [n](\mathbf{int}, \alpha) \to [n]\alpha$
`unzip`	$: [n](\alpha, \beta) \to ([n]\alpha, [n]\beta)$	`groupby`	$: (m : \mathbf{int}) \to [n]\mathbf{int} \to [n]\alpha \to [m][]\alpha$
`partition`	$: (\alpha \to \mathbf{bool}) \to [n]\alpha \to ([m]\alpha, [k]\alpha)$	`random`	$: \mathbf{unit} \to [c]\mathbf{int}$
`filter`	$: (\alpha \to \mathbf{bool}) \to [n]\alpha \to [m]\alpha$	`hash`	$: [c]\mathbf{int} \to \alpha \to \mathbf{int}$
`presum`	$: [n]\mathbf{int} \to [n]\mathbf{int}$	`hist`	$: (n : \mathbf{int}) \to [m]\mathbf{int} \to [m]\mathbf{int} \to [n]\mathbf{int}$
`sum`	$: [n]\mathbf{int} \to \mathbf{int}$	`repiota`	$: [n]\mathbf{int} \to [m]\mathbf{int}$

Fig. 1: The data-parallel vocabulary used for explaining the construction of two-level hash maps.

The size of the arrays is denoted by n, m, and k and is used to denote that the arrays have different sizes for a given function while c is a constant size. The first projection of tuples is denoted by `fst` and the second by `snd`. Applying a function on every element in an array is denoted by `map`. Producing an array of integers from 0 to $n-1$ is done by `iota` n. Constructing an array of tuples from two arrays is done by `zip` and `unzip` is the inverse to $\lambda(a, b) \to$ `zip` $a\, b$. For partitioning an array into two arrays based on a predicate we use `partition`, the array in the first position of the result is for elements where the predicate holds true and the second is for elements where it holds false. Filtering every element in an array that does not satisfy a predicate is denoted by `filter`. The operation for computing a prefix sum of an array of integers is denoted by `presum`. The operation for computing the sum of an array of integers is denoted by `sum`. The operation for computing the logical or over an array of booleans is `or`. A segmented scan using the logical or operation can be computed using `segor` (i.e., a segmented-`or`). An expression $\texttt{rep}\ n\ a$ results in an array where the element a is replicated n times. The `scatter` function takes an array to be written to, an array of indices, and an array of values to write at the given indices. Sorting is denoted by `sort`, which sorts the array by the first element and discards it (e.g., $\texttt{sort}\ [(2, x), (1, y)] = [y, x]$). To group an array of elements by an array of indices is denoted by `groupby`, which produces an irregular array of arrays (e.g., $\texttt{groupby}\ 4\ [2, 0, 2]\ [x, y, z] = [[y], [], [x, z], []]$). It is important to note the implementation of `groupby` in a data-parallel language would be based on a integer sort. The function `random` produces an array, of fixed size c, containing c random integers. This function must have the property that returned values form a distribution such that when given as constants to `hash` they form a universal family of hash functions [5] [23, p. 2]. The basic idea is to have the ability to generate random constants using `random` to be used in `hash` such that it is possible to generate random hash function. We treat the `random` function as impure, but it is readily implemented in a functional language through conventional state passing mechanisms. Computing histograms is denoted by $\texttt{hist}\ n\ is\ vs$, here n is the number of bins (initialized to 0), is specifies what bucket a given value in vs belongs to (i.e., like `scatter` but allows for duplicate indices where their values are summed). A replicated iota [8] is denoted by `repiota`, which is like `iota` but where each integer in the argument array determines the number of times an element is repeated (e.g., $\texttt{repiota}\ [2, 3, 1] = [0, 0, 1, 1, 1, 2]$).

2.2 The Construction

A *hash set construction* for a non-empty finite set of keys K (taken from a universe U), with cardinality $|K| = n$, and parameterised over a constant c, is a tuple

$$(\mathit{const}, \mathit{consts}) : ([c]\mathbf{int}, [n][c]\mathbf{int})$$

which defines a hash function $\texttt{hash}_1 : U \to \mathbf{int}$ by the hash function $\texttt{hash}_2 : U \to \mathbf{int}$ as follows:

$$\begin{aligned}
&\texttt{hash}_1\ k = \mathit{offsets}[\texttt{hash}_2\ k] + (\texttt{hash}\ \mathit{consts}[\texttt{hash}_2\ k]\ k \ \texttt{mod}\ \mathit{shape}[\texttt{hash}_2\ k])\\
&\texttt{hash}_2\ k = (\texttt{hash}\ \mathit{const}\ k)\ \texttt{mod}\ n\\
&\mathit{shape} = (\texttt{map}\ (\lambda \mathit{size} \to \mathit{size}^2) \circ \texttt{hist}\ n\ (\texttt{map}\ \texttt{hash}_2\ K))\ (\texttt{rep}\ n\ 1)\\
&\mathit{offsets} = \texttt{presum}\ \mathit{shape}
\end{aligned}$$

We further say that such a hash set construction is *well-formed* and *collision-free* if the following properties hold:

1. $\{\texttt{hash}_1\ k \mid k \in K\} \subseteq \{0, \ldots, s - 1\}$, where $s = \texttt{sum}\ \mathit{shape}$

2. $|\{\texttt{hash}_1\ k \mid k \in K\}| = |K|$

The goal now is, given a non-empty finite set of keys, to find a collision-free and well-formed hash set construction. Given such a construction, it is then possible to create an array *arr* of size s and place every key *key* at index $\texttt{hash}_1\ k$ and fill remaining indices with any key $k \in K$. To assert if for some $a \in U$ (the universe), we also have $a \in K$, we simply test if $arr[\texttt{min}\ (s-1)\ (\texttt{hash}_1\ a)] = a$ holds true. Such a hash set can be extended to a hash map by storing the value in an accompanying array where $\texttt{hash}_1\ a$ is used for indexing.

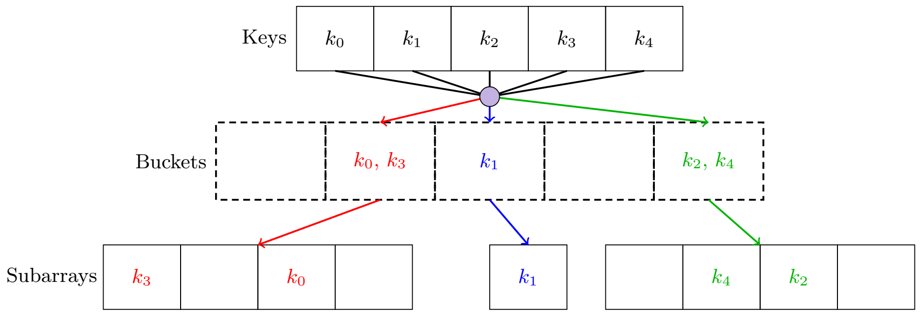

Fig. 2: Visualization of the two-level hash set when using $\texttt{hash}_1$. Initially all keys are hashed with $\texttt{hash}_2$. Then each key lands into a bucket where *offsets*, *shape* and *consts* can be accessed using the index of the bucket (i.e the hash returned by $\texttt{hash}_2$). Use the buckets hash function constant from *consts* and subarray size from *shape* to hash a key into its subarrays size. Lastly add the offset from *offsets* to the get keys index in the flat array. Here the flat array is the subarrays concatenated together.

2.3 Functional Construction

The data-parallel variant of the two-level hash set construction can be derived from a functional variant, which we now present. This variant comes from a formulation of the construction described in the FKS paper [9].

We first define a function `hashes` for computing the hashes of an array of keys and a function `collision` for detecting if the hashes of a set of constants collide:

$$\textbf{def}\ \texttt{hashes}\ (\mathit{const} : [c]\textbf{int})\ (\mathit{keys} : [m]\alpha) : [m]\textbf{int} =$$
$$\texttt{map}\ ((\texttt{mod}\ m^2) \circ \texttt{hash}\ \mathit{const})\ \mathit{keys}$$

$$\textbf{def}\ \texttt{collision}\ (\mathit{hashes} : [m]\textbf{int}) : \textbf{bool} =$$
$$(\texttt{or} \circ \texttt{map}\ (> 1) \circ \texttt{hist}\ m^2\ \mathit{hashes})\ (\texttt{rep}\ m\ 1)$$

The top-level and the lower-level of the FKS construction are defined by the two functions $\texttt{make}_1$ and $\texttt{make}_2$, as follows:

```
def make_2 (keys : [m]α) : [c]int =
  let const = random ()
  let hs = hashes const keys
  in if collision hs then make_2 keys else const

def make_1 (keys : [n]α) : ([c]int, [n][c]int) =
  let const = random ()
  let hashes = map ((mod n) ∘ hash const) keys
  let consts = (map make_2 ∘ groupby n hashes) keys
  in (const, consts)
```

At top-level (i.e., the $\texttt{make}_1$ function), the first step is to pick an array of random constants for a hash function. This array is used to compute the *hashes* of *keys*. Afterwards every key is grouped by its hash and then the $\texttt{make}_2$ function is called on each subarray of keys such that a collision-free hash set for every subarray can be found. We consider every hash set made by $\texttt{make}_2$ to be the second level of the hash set. The function $\texttt{make}_2$ continues to generate constants *const* for a hash function until it finds constants that lead to a collision-free hash function.

We observe now that in the functional variant, the function $\texttt{make}_2$ is mapped over an irregular array of keys, which indicates that the variant is not immediately data-parallel, although it may provide a basis for a task-parallel variant. To construct a data-parallel variant of the FKS construction, we shall set out to perform a flattening-transformation of the functional variant of the algorithm.

Analysis Almost every line of $\texttt{make}_1$ does $O(n)$ work except for the mapping of $\texttt{make}_2$ over the grouped keys which creates *consts*. Grouping can be done in $O(n)$ work and $\texttt{make}_2$ does $O(n_i^2)$ work where n_i is the size of the subarray. The sum of every subarray of size n_i^2 has the expected asymptotic value of $O(n)$ and for any application of $\texttt{make}_2$ on a subarray it is expected to do at most 2 trials to find a collision free hash function (i.e., expected $O(1)$ recursive calls) [1]. The total work done by mapping $\texttt{make}_2$ over every subarray has the expected work of $O(n)$ and so does the total work done by constructing the hash set.

2.4 Flattened Construction

Now we will start by flattening the functional formulation for constructing a two-level hashset. This flattened version will be a step in the derivation of the final formulation where sorting will be avoided.

We first have to concern ourselves with $\texttt{make}_1$. It operates on an irregular array of keys due to the use of `groupby`. The way we represent an irregular array is by associating a flat array with a *shape* array where the elements represent the size of a subarray in the flat array. This representation allows us to implement the `groupby` operation using a stable sorting algorithm, to sort the keys by their hashes, and using a histogram to compute the shape. The second task is to replace the call to $\texttt{make}_2$, which maps over an irregular array, with a call to a *lifted* version of $\texttt{make}_2$, called $\texttt{segmake}_2$, which operates on a flattened array of keys along with the shape of the subarrays. Using this strategy, we can derive the following flattened version of $\texttt{make}_1$:

```
def $\texttt{make}_1$ $(keys : [n]\alpha) : ([c]\mathbf{int}, [n][c]\mathbf{int}) =$
  let $const = \texttt{random}\ ()$
  let $hashes = \texttt{map}\ ((\texttt{mod}\ n) \circ \texttt{hash}\ const)\ keys$
  let $sorted\_keys = (\texttt{sort} \circ \texttt{zip}\ hashes)\ keys$
  let $shape = \texttt{hist}\ n\ hashes\ (\texttt{rep}\ n\ 1)$
  let $consts = \texttt{segmake}_2\ shape\ sorted\_keys$
  in $(const, consts)$
```

To lift $\texttt{make}_2$, the idea is to lift `random`, `hashes`, and `collision` individually and then use these lifted functions to define $\texttt{segmake}_2$. For the following lifted functions, we will ignore an edge case where a subarray is of size 0 as such cases can be handled later in $\texttt{segmake}_2$ by filtering out zero-size subarrays. To lift `random` we need to generate a constant for every subarray.

```
def segrandom $(shape : [m]\mathbf{int}) : [m][c]\mathbf{int} =$
  $(\texttt{map}\ \texttt{random} \circ \texttt{rep}\ m)\ ()$
```

Now for `hashes`, we want to know the hash of every key but since this is the lifted version, the hash is located at an offset that depends on what subarray the key belongs to. So to lift `hashes`, we need to know the offsets of the flattened array of squared subarray sizes. These offsets can be determined by taking the prefix-sum of the square of every size in the shape array. Afterwards, the index of what subarray a key belongs to is needed so the offset, constants, and size of the subarray can be retrieved. Here we use `repiota` to produce an array of indices that associates every key with its given subarray. From these building blocks we can define the lifted version of `hashes`:

```
def seghashes $(shape : [m]\mathbf{int})\ (consts : [m][c]\mathbf{int})\ (keys : [n]\alpha) : [m]\mathbf{int} =$
  let $offsets = (\texttt{presum} \circ \texttt{map}\ (\lambda size \to size^2))\ shape$
  in $(\texttt{map}\ (\lambda(k, j) \to$
    let $(offset, const, size) = (offsets[j], consts[j], shape[j])$
    in $offset + (\texttt{hash}\ const\ k)\ \texttt{mod}\ size^2) \circ \texttt{zip}\ keys)\ (\texttt{repiota}\ shape)$
```

As one can see, the lifted version of `hashes` computes offsets and the keys subarray indices first. Then the keys are zipped with their subarray index and the hash is computed for every key by retrieving the offset, constants, and size of the subarray it belongs to.

Finally, the `collision` function is lifted, which can be done by computing the squared shape so every hash is an index into a flattened array. Then, using a histogram operation, the number of collisions for every subarray can be computed, and the segmented-or can be used to determine if any subarray has a collision. Here is the lifted version of the `collision` function:

```
def segcollisions $(shape : [m]\mathbf{int})$ $(hashes : [n]\mathbf{int}) : [m]\mathbf{bool} =$
  let $shape\_squared = \texttt{map}(\lambda size \to size^2)\ shape$
  let $flat\_size = \texttt{sum}\ shape\_squared$
  let $counts = \texttt{hist}\ flat\_size\ hashes\ (\texttt{rep}\ n\ 1)$
  in $(\texttt{segor}\ shape\_squared \circ \texttt{map}\ (> 1))\ counts$
```

The next problem is how to translate the last conditional expression of $\texttt{make}_2$ into a lifted version. The goal is to stop working on all subarrays where the constants for the given hash function leads to zero collisions and then return its constants. The problem here is that the subarrays may finish out of order. To solve this issue, every subarray size in shape and resulting constant for a hash function is associated with an index. This index can then be used to reorder the constants at the end. For this part, a lifted version of the results and new keys that will be passed along to the recursive call will be given. This is done by partitioning the indices of the subarrays based on whether they have collisions or not. The subarrays that are done can then be used to produce the resulting constants and the remaining subarrays shape are worked on in the recursive call. Furthermore, using the collisions array the keys that will be worked on in the recursive call can be determined. This leads to the following two functions:

```
def segresult $(is\_shape : [m](\mathbf{int}, \mathbf{int}))$ $(consts : [m][c]\mathbf{int})$
    $(collisions : [m]\mathbf{bool}) : ([](\mathbf{int}, \mathbf{int}), [](\mathbf{int}, [c]\mathbf{int})) =$
  let $is = \texttt{map}\ \texttt{fst}\ is\_shape$
  let $(not\_done, done) = \texttt{partition}\ (\lambda i \to collisions[i])\ (\texttt{iota}\ m)$
  let $is\_consts' = \texttt{map}\ (\lambda i \to (is[i], consts[i]))\ done$
  let $is\_shape' = \texttt{map}\ (\lambda i \to is\_shape[i])\ not\_done$
  in $(is\_shape', is\_consts')$
```

```
def segkeys $(shape : [m]\mathbf{int})$ $(collisions : [m]\mathbf{bool})$ $(keys : [n]\alpha) : []\alpha =$
  let $collision\_keys = (\texttt{map}\ (\lambda i \to collisions[i]) \circ \texttt{repiota})\ shape$
  in $(\texttt{map}\ \texttt{snd} \circ \texttt{filter}\ \texttt{fst} \circ \texttt{zip}\ collision\_keys)\ keys$
```

Now we almost have all the building blocks to define the lifted version of $\texttt{make}_2$. We still need an auxiliary function that will give back an unordered array of constants from the subarrays that are done. This function is called $\texttt{segmake}'_2$

and simply applies the lifted functions in order:

```
def segmake'_2 (is_shape : [m](int, int)) (keys : [n]α) : [](int, [c]int) =
  if n = 0 then [] else
  let shape = map snd is_shape
  let consts = segrandom shape
  let hashes = seghashes shape consts keys
  let collisions = segcollisions shape hashes
  let (is_shape', is_consts') = segresult is_shape consts collisions
  let keys' = segkeys shape collisions keys
  in is_consts' ++ segmake'_2 is_shape' keys'
```

It is important to note that the recursive function can be written in a tail-recursive manner by using an accumulator for the constants or just using a while-loop construct if the language supports it. This fact is helpful since this function will map well to a GPU.

Now we can finally define the lifted version of $\texttt{make}_2$ by removing empty subarrays, applying $\texttt{segmake}'_2$, and scattering the resulting constants into an array to obtain the constants in the correct order. Here we achieve the final function $\texttt{segmake}_2$:

```
def segmake_2 (shape : [n]int) (keys : [n]α) : [n][c]int =
  let is_shape = (filter ((≠ 0) ∘ snd) ∘ zip (iota n)) shape
  let (is, consts) = (unzip ∘ segmake'_2 is_shape) skeys
  in scatter (rep n (random ())) is consts
```

It is important to note is we give all empty subarrays the same random constant since any constant will be collision-free for an empty subarray.

2.5 Sortless Construction

Now that we have a flattened variation of constructing the constants needed for a two-level hash set the goal is now to avoid sorting. The idea to avoid sorting is to associate every key with the index of the subarray it belongs to. Using this index we can determine what subarray a key belongs to instead of sorting and relying on `repiota`. To achieve this we will start off by modifying $\texttt{make}_1$ by simply removing the sorting step:

```
def make_1 (keys : [n]α) : ([c]int, [n][c]int) =
  let const = random ()
  let hashes = map ((mod n) ∘ hash const) keys
  let shape = hist n hashes (rep n 1)
  let consts = segmake_2 shape keys hashes
  in (const, consts)
```

Now we have to modify $\texttt{segmake}_2$ to work without sorted keys. To avoid sorting every key is associated with the offset of the subarray it belongs to by using the *hashes* array. This results in the following modified version of $\texttt{segmake}_2$:

```
def $\texttt{segmake}_2$ $(shape : [n]\textbf{int})$ $(keys : [n]\alpha)$ $(hashes : [n]\textbf{int}) : [n][c]\textbf{int}$ =
  let $offsets = (\texttt{presum} \circ \texttt{map}\ (\neq 0))\ shape$
  let $sub\_offsets = \texttt{map}\ (\lambda i \to offsets[i])\ hashes$
  let $js\_keys = \texttt{zip}\ sub\_offsets\ keys$
  let $is\_shape = (\texttt{filter}\ ((\neq 0) \circ \texttt{snd}) \circ \texttt{zip}\ (\texttt{iota}\ n))\ shape$
  let $(is, consts) = (\texttt{unzip} \circ \texttt{segmake}'_2\ js\_keys)\ is\_shape$
  in $\texttt{scatter}\ (\texttt{rep}\ n\ (\texttt{random}\ ()))\ is\ consts$
```

Here *sub_offsets* is every key's subarray offset according to its hash and then *js_keys* is an array of tuples where every key is associated with its subarray offset.

The next step is to modify `seghashes` since we no longer can rely on the keys being in order according to their subarray. This is done by letting `seghashes` use the subarray index associated with every key to compute its hash instead of using the subarray indices created by `repiota`. Doing this leads to the following modified version of `seghashes`:

```
def seghashes $(shape : [m]\textbf{int})$ $(consts : [m][c]\textbf{int})$
    $(js\_keys : [n](\textbf{int}, \alpha)) : [n]\textbf{int}$ =
  let $offsets = (\texttt{presum} \circ \texttt{map}(\lambda size \to size^2))\ shape$
  in map $(\lambda(j, key) \to$
    let $(offset, const, size) = (offsets[j], consts[j], shape[j])$
    in $offset + (\texttt{hash}\ const\ key)\ \texttt{mod}\ size^2)\ js\_keys$
```

The next problem is $\texttt{segmake}'_2$ since now whenever keys and subarrays are removed the offset associated with the key is no longer valid since the number of subarrays have changed. This can be resolved by finding the keys new offset after removing the done subarrays and then update the key's offset to be the new offset. And so we arrive to the following modified version of `segmake`$'_2$:

```
def segmake'_2 $(\mathit{js\_keys} : [n](\mathbf{int}, \alpha))$
        $(\mathit{is\_shape} : [m](\mathbf{int}, \mathbf{int})) : [](\mathbf{int}, [c]\mathbf{int}) =$
    if $n = 0$ then $[]$ else
    let $\mathit{shape}$ = map snd $\mathit{is\_shape}$
    let $\mathit{consts}$ = segrandom $\mathit{shape}$
    let $\mathit{hashes}$ = seghashes $\mathit{shape}\ \mathit{consts}\ \mathit{keys}$
    let $\mathit{collisions}$ = segcollisions $\mathit{shape}\ \mathit{hashes}$
    let $\mathit{key\_offsets}$ = presum $\mathit{collisions}$
    let $(\mathit{is\_shape}', \mathit{is\_consts}')$ = segresult $\mathit{is\_shape}\ \mathit{consts}\ \mathit{collisions}$
    let $\mathit{js\_keys}'$ =
        (map $(\lambda(j, \mathit{key}) \rightarrow (\mathit{key\_offsets}[j], \mathit{key}))$
        $\circ$ filter $(\lambda(j, \mathit{key}) \rightarrow \mathit{collisions}[j]))$ $\mathit{js\_keys}$
    in $\mathit{is\_consts}'$ ++ segmake'_2 $\mathit{js\_keys}'\ \mathit{is\_shape}'$
```

Here the offsets are found by taking a prefix-sum of the *collisions* array to find the key offsets after removing the done subarrays.

Analysis Once again almost every line of code in `make`$_1$ is $O(n)$ since every line of code does at most $O(n)$ work besides `segmake`$_2$. It ends up doing the expected work of $O(n)$ since it works on every subarray of size n_i^2 and it is expected to work on each subarray at most 2 times. And all the work done internally by `segmake`$_2$ is using functions which do at most the expected work of $O(n)$. So in total the algorithm has the expected work of $O(n)$ hence it is work efficient.

For the span of the algorithm we see a segmented-or over the flattened array which has an expected span of $O(\log n)$. No other operation has a worse span, this means that the expected span of the algorithm is $O(\log n)$.

3 Interface

In this section we will describe our abstract programming interface for key-value maps in a functional array language. The goal is to define an interface that can allow keys of any type, as long a universal hash function and a total ordering exists for the type, and values of any type. We will use the concrete syntax of Futhark, including its ML-derived module system [7], but we will assume no familiarity and describe the concepts as necessary. For a primer on the Standard ML module system, see [24], although note that Futhark diverges somewhat in syntax and nomenclature. The interface is condensed compared to our actual implementation, in order to avoid cluttering the central ideas.

3.1 Irregularly Sized Keys

One major technical challenge is how to handle keys of irregular size (e.g., strings), which is not directly supported in Futhark. The problem is that Futhark does not support *irregular arrays* (sometimes called *jagged arrays*), meaning arrays where elements can have different sizes. When strings are modeled as arrays of characters, this prevents the key array from being representable, unless all the keys happen to be strings of the same length—which is an unlikely case. While our implementation is in Futhark, the requirement for regular arrays is a common limitation among functional array languages, such as Accelerate [6] and SaC [22], and is ultimately rooted in fundamental issues the efficient representation of arrays in memory. The NESL language [4] does support irregular arrays (through a flattening transformation), but the resulting representation, when done completely automatically, tends to be rather inefficient. APL [14] supports a form of irregular arrays by "boxing" the elements, essentially turning them into scalars, but the resulting in-memory representation becomes an array of pointers, which can lead to poor locality. Most of these languages are targeted at numerical problems, where the inability to represent irregular arrays is less of a problem, but it does pose a challenge when we wish to represent something as basic as an array of strings.

One common solution, and the one we will use, is to represent a string as a pair of an *offset* and a *length* into some other array of characters, which we call the *context* (the choice of this term will be made clear shortly). The idea is that the context essentially consists of the concatenation of all strings of interest, and we slice it as necessary to obtain substrings. The context is not a *string pool*, as we do not assume that a string is *uniquely* determined by its offset and length - it may well be that two strings with different offsets actually correspond to sequences of equivalent characters in the context.

In some languages, it would be possible for a string to contain a reference to the context from which its characters are sliced. This is not possible in Futhark, or most similar value-oriented array languages, as they do not support references or other such pointer-like types, and a reference to the context would imply a copy. Instead, the coupling between a string and its context is implicit, and for all functions defined on strings—such as ordering or hashing—the caller must pass the context as well. Passing the wrong context by accident is indeed a potential source of errors, and one that Futhark's type system is not sufficiently strong to avoid. In our implementation of key-value maps, we mandate that all keys stored in the map must use the same context (which we will also store in the map itself), but when querying membership, the "needle" need not be.

3.2 The Map Interface

The abstract interface for our key-value maps, expressed as a module type (*signature* in Standard ML), is shown in Section 3.2. A module type specifies types, values, and functions that must be provided by an implementation of the interface. In the module type, all types are abstract, but an implementation will often *refine* some of these types to be concrete.

The map module type comprises three abstract types: key denotes type of keys (which an implementation will refine to be a specific type, e.g., i32), the ctx is the context used for comparison and hash functions, and map [n] v is the type of a map containing n key-value mappings to values of type v. The [n] is a *size parameter* [3], which is used to document how the sizes of function arguments relate to function outputs, although it is only of minor importance to the present work. The map type is declared with the keyword type~, which is Futhark notation for an abstract type that may internally contain arrays of unspecified size—this implies that we may not construct arrays with elements of type map, as this would essentially result in irregular arrays.

The map module type also specifies three API functions: from_array constructs a map from an array of key/value-pairs, from_array_nodup does the same but assumes that no duplicate keys exist (saving us deduplication during construction), and lookup retrieves the value corresponding to a specified key, using an option type to handle the case where no such key exists in the map.

```
module type map = {
  type key
  type ctx
  type~ map [n] 'v
  val from_array [u] 'v: ctx -> [u](key, v) -> ?[n].map [n] v
  val from_array_nodup [n] 'v: ctx -> [n](key,v) -> map [n] v
  val lookup [n] 'v: ctx -> key -> map [n] v -> opt v
}
```

Fig. 3: The map module type, which specifies the abstract types and functions of the key-value API. This is a minimal API that elides many convenient functions that are present in our full implementation, but are not interesting from a data-parallel perspective.

3.3 The Key Interface

The map interface can be implemented in many ways, but an implementation is always specialised for a given key type, as the map type itself is not polymorphic in the keys, only in the values. Clearly it is not desirable to reimplement the interface from scratch whenever we wish to use a new key type. We solve this with *parameterised modules* (*functors* in Standard ML), which can be seen as module-level functions that are applied at compile time to form new modules. Essentially, we write a parameterised module that, given a module that provides a key type and necessary operations on keys, constructs a module that implements the map module type for the supplied key type.

We describe our requirements for keys in a module type key, shown in Figure 4. Three types are specified: ctx is the context as discussed above, key is the key type itself, and finally `uint` is the type of hashes. We also require three value definitions: `c` is the number of constants passed to the hash function as discussed in Section 2.1, `hash` is a function for hashing a value of type `key`, and `(<=)` is the comparison operator. Both `hash` and `(<=)` are provided a context. For the latter, two contexts are in fact used: one for each value, which is crucial when performing lookups, as the "needle" key is likely to use a different context than "haystack" keys.

Figure 4 also shows a binding of a module `i32key` that implements the `key` module type where all types have been made concrete. The implementation has been elided, but it is based on a universal hash function. Note that the context type has been refined to the unit type, as no context is needed for integer keys.

```
module type key = {
  type ctx
  type key
  type uint
  val c: i64
  val hash: ctx -> [c]uint -> key -> uint
  val (<=): (ctx, key) -> (ctx, key) -> bool
}

module  u8key: key with ctx = () with key = u8
                                  with uint = u64 = ...

module i32key: key with ctx = () with key = i32
                                  with uint = u64 = ...
```

Fig. 4: The `key` module type, which must be implemented for any type used as a key, as well as a binding of modules `u8key` and `i32key` that declare they implements a refinement of `key` with all abstract types made concrete.

Array Slices as Keys We can now define a representation of array slices that implement the `key` module type. In Figure 5 we first define a module type `slice` that specifies the rather simple interface for array slices: constructing slices from offset and length, decomposing a slice into offset and length, and finally applying the slice to a concrete array, returning another array of some unknown size `k`. The type `slice a` denotes a slice of an array with element type `a`—in practice, `a` is a phantom type. The implementation of the module type (as a module also called `slice`) is straightforward and also shown in Figure 5.

The `mk_slice_key` parameterised module accepts a `key` module `E` and produces a `key` module for keys of type `slice E.key`, and context `[]E.key`. Importantly, the element type itself must not require any context. We elide the implementation for brevity, but note that the implementation of a universal hash function for sequences is somewhat complicated, although unrelated to data-parallelism and outside the scope of this paper.

```
module type slice = {
  type slice 'a
  val mk 'a: (i: i64) -> (n: i64) -> slice a
  val unmk 'a: slice a -> (i64, i64)
  val get [n] 'a: slice a -> [n]a -> ?[k].[k]a
}

module slice : slice = {
  type slice 'a = {i: i64, n: i64}
  def mk i n = {i, n}
  def unmk {i, n} = (i, n)
  def get {i, n} xs = xs[i:i + n]
}

module mk_slice_key (E: key with ctx = () with uint = u64)
  : key with ctx = []E.key
        with key = slice.slice E.key
        with uint = u64 = {
  ...
}
```

Fig. 5: The `slice` module type and module, as well as a parameterised module `mk_slice_key` that constructs a key module for slices of arrays of some type, given a module that implements `key` for the element type, with a unit context.

3.4 Constructing Maps

Now that we have established the `key` and `map` abstractions, we can defined parameterised modules that produce implementations of `map`. Figure 6 sketches an implementation that represents the mapping as a sorted array of keys and a corresponding array of values. Construction then requires sorting (and possibly deduplicating) the provided key/value-pairs, and `lookup` is implemented as a binary search. This is a fairly simple implementation, assuming that one has access to an efficient sorting function. Note that with our definition of the `key` module type, radix sorting is not possible (because we provide no way to decompose into bits), so instead our implementation uses merge sort. As a locality optimisation, instead of storing the keys in sorted order, we can use an Eytzinger layout that essentially stores each level of the corresponding binary tree that is implicitly formed when performing the binary search [25].

We show the implementation of `lookup` in order to demonstrate how contexts are passed around. We assume the existence of a function `binary_search`

`: (v -> v -> bool) -> []v -> v -> i64` that given a comparison operator performs a binary search for the index of a provided needle in a sorted array, and returns `-1` in case the needle is not found. It is crucial that `lookup` provides the right context to the `K.<=` operator. In particular, the `x` parameter is drawn from the arraymap itself, while `y` is the needle. This means the former must be paired with the `ctx` from the map, and the latter with the `ctx` provided by the caller of `lookup`. Unfortunately these are both of type `ctx`, so mixing them up will not cause a type error—we will instead obtain wrong results at runtime, and somewhat insidiously, the issue will only occur for keys that actually make use of the contexts, and not for the common case of, for instance, integer keys.

```
module mk_arraymap (K: key)
    : map with key = K.key with ctx = K.ctx = {
  type key = K.key
  type~ ctx = K.ctx
  type~ map [n] 'v = {ctx: ctx, keys: [n]key, vals: [n]v}

  def lookup [n] 'v (ctx: ctx) (k: key) (m: map [n] v) =
    let cmp x y =  (m.ctx, x) K.<= (ctx, y)
    in match binary_search cmp m.keys k
       case -1 -> #none
       case i -> #some m.vals[i]

  ...
}
```

Fig. 6: Outline of the `mk_arraymap` parameterised module, which given an implementation of `key` constructs an implementation of `map`. The value definitions have been elided for brevity.

Of course, given the topic of the present paper, an implementation based on binary search is not satisfactory. We therefore also define `mk_hashmap`, shown in Figure 7, which is based on two-level hash maps following the approach of Section 2. Note the complete equivalence with the interface in Figure 6.

While the definition of these modules is somewhat intricate, applying the building blocks is straightforward, as shown in Figure 8. The resulting modules can be used without knowledge of module-level programming, and it is straightforward to provide predefined modules for common key types in a library. Some of the generality in the underlying design does leak through in undesirable ways. For example, the type of `i32hashmap.lookup` is `() -> i32 -> i32hashmap.map [n] v -> opt v`. The `()` parameter is the context, which is the unit type for this module. It is not difficult to write wrapper definitions that remove the need for passing a unit value to `lookup`, but it is somewhat tedious, and requires replicating the entire `map` module type—which is significantly larger

```
module mk_hashmap (K: key)
    : map with key = K.key with ctx = K.ctx = {
  type~ map [n] 'v =
      ?[f][m].
      { ctx: K.ctx
      , key_values: [n](K.key, v)
      , offsets: [f]i64
      , level_one_consts: [K.c]uint
      , level_two_consts: [m][K.c]uint
      , level_two: [n][3]i64
      , rng: rng
      }
  ...
}
```

Fig. 7: Outline of the `mk_hashmap` parameterised module, which given an implementation of `key` constructs an implementation of `map` based on two-level hash maps as discussed in Section 2. We assume the presence of a type `rng` that is a the state of a random number generator. The sizes `f` and `m` are internally bound and not visible in the external type.

in a practically useful library than the subset we show in Section 3.2, although the cost is entirely on the implementer of the module, not on the user.

```
module i32arraymap = mk_arraymap i32key
module i32hashmap = mk_hashmap i32key
module strkey = mk_slice_key u8key
module stringhashmap = mk_hashmap strkey
```

Fig. 8: Modules for various implementations of the `map` module type, where we consider strings to be simply arrays of unsigned 8-bit integers.

4 Benchmarks

We have implemented two-level hash maps in Futhark, which is a functional array language that can generate usually quite efficient GPU code [19,13,10]. To evaluate the performance of our hash maps, we have constructed a set of benchmarks. They are all based around inserting n keys of two types (64-bit integers and strings of 5–25 characters), with the values always being 32-bit integers. The keys are uniformly distributed and generated such that there are no duplicates. For various implementations of key-value maps, we benchmark the time it takes to perform the following operations:

1. *Construction* of the map (e.g., building a hash map or sorting an array).

2. *Lookup* of the value associated with every key in the map.
3. *Membership* testing of every key—this differs from lookup in that the value is not retrieved.

We benchmark the following implementations:

1. Futhark, using two-level hash maps.
2. Futhark, using binary search.
3. Futhark, using binary search with an Eytzinger layout.
4. cuCollections [20], an implementation of hash maps in CUDA C++, which is to our knowledge the state of the art in hash maps on GPUs. The static hash map in cuCollections is not based on two-level hash maps, but rather uses open addressing with linear probing.

We run our benchmarks on an NVIDIA A100 GPU, using Futhark's CUDA backend, and without disabling bounds checking (which adds about 5% overhead [11]). The results are shown in Table 1. The most obvious result is that cuCollections is vastly faster than Futhark in all cases; ranging from nearly $10\times$ speedup in the construction of the hash table, to $1.8\times$ for membership testing. We will return to this in Section 4.1. The results suggest that even in a data-parallel language and for integral keys, hash maps bring meaningful speedup compared to comparison/tree-based techniques. Because we assume unique keys, constructing the hash maps is not even slower than sorting key/value arrays, although this would not be the case if we had to perform deduplication of keys.

	64-bit integer keys ($n = 10^7$)			**String keys** ($n = 10^7$)		
	Construction	*Lookup*	*Membership*	*Construction*	*Lookup*	*Membership*
Futhark (hash maps)	18.3	3.3	1.6	33.2	4.3	2.8
Futhark (binary search)	40.9	6.2	5.8	83.0	5.7	5.8
Futhark (Eytzinger)	42.3	4.3	2.4	85.3	5.3	5.3
cuCollections	2.7	1.1	0.9	2.7	1.3	1.2

Table 1: Benchmark results. All runtimes are in milliseconds.

4.1 Future Work

It is clear that the hash maps implemented in cuCollections significantly outperform the ones we have implemented in Futhark. This is not surprising—cuCollections is developed by GPU experts with a track record of research into GPU hash maps [15,17], and using a low-level language that allows tuning and exploitation of hardware properties. It is unlikely that an implementation in a high level and hardware-agnostic language can match this. Yet it may still be possible to narrow the gap, and this may improve high-level implementations of *other* algorithms, which may not already have been carefully implemented in low-level languages by experts.

The fastest GPU hash map implementations are open addressing hash maps, implemented as concurrent data structures, with multiple threads simultaneously inserting elements [17,16], with synchronisation done through atomic operations.

While open addressing hash maps can be implemented in Futhark (and we have done so as an experiment), we find that the efficient implementations that are possible in CUDA cannot be expressed using the available data-parallel vocabulary, and they perform significantly worse than two-level hash maps.

The core problem is that although hash maps provide a deterministic interface, their efficient (concurrent) implementation is *internally* nondeterministic—for instance, when two threads compete for the same slots in an open addressing hash map, it is nondeterministic who gets which slot, although the choice is not externally visible. When implementing a data structure in a deterministic data-parallel language, such as Futhark and most other array languages, we are not at liberty to locally exploit nondeterminism and impurity the way we can do in CUDA. Of course, the flip side is also that the language prevents us from accidentally introducing nondeterminism—the race-freedom of cuCollections is due to careful implementation and testing, rather than a guaranteed property of CUDA itself. However, it does seem functional array languages are missing a mechanism for locally exploiting nondeterministic operations, which can be encapsulated in such a way that the nondeterminism can be localised, but it is not clear what such a mechanism should look like. The Dex language uses an effect system to allow local mutation of an array shared between multiple iterations of a parallel loop [21], although the only effect allowed without hindering parallelism is incrementing an array element. This is not sufficient for our needs, and indeed the restriction to increments is what provides determinism.

Another point of view is that the language should provide hash maps as a built-in type, which can then be implemented with low-level code in the language implementation itself. The arrays in an array language are implemented this way, so it is certainly conceivable that hash maps could be provided similarly—and some APL dialects do indeed provide such built-in data structures. While pragmatic, we do consider this approach less compelling from a research perspective, and certainly less ambitious from an implementation perspective.

5 Related Work

There are several implementations of hash maps meant for GPUs. The warpcore and warpdrive libraries [17,16] are written in CUDA and also has an implementation of open addressing hash maps. They do a similar parallel probing scheme like cuCollections such that every thread in a warp works together to find a key to achieve coalesced access patterns. The warpcore library also contains a bucket list hash table, where each bucket contains a linked list of key-value pairs.

The consolidation of the aforementioned libraries are cuCollections [20]. The library contains a fast implementation of open addressing hash maps and is used for benchmarking in this paper.

BGHT [2] has static hash maps which both uses bucketed cuckoo hash tables and iceberg hashing. These hash tables are implemented in CUDA and are designed to be used in a concurrent setting.

The aforementioned libraries are all written in CUDA, but there also exists an implementation of a hash map[2] written in the high-level functional array language NESL [4]. It is an open addressing hash map with quadratic probing, what they roughly do is try to insert every key in parallel. All keys that lead to a collision because the slot is already occupied, try recursively to insert the keys into the next slot using quadratic probing. And if the slot is empty then select one of the keys to insert into that slot, the remaining are then inserted in the recursive call using the probing scheme. Such an implementation could lead to nondeterminism depending on the implementation but you could also use some rule to select which key to insert into the empty slot, like we do for our open addressing hash map written in futhark. The reason for this not being benchmarked against in the present paper is that no compiler is available for NESL that can generate code for our GPUs.

6 Conclusions

We have shown in detail how to derive a data-parallel construction of two-level hash maps by flattening a functional formulation. The resulting algorithm is work efficient, meaning it has an expected work of $O(n)$ and an expected span of $O(\log n)$.

We describe how such key/value data structures can be provided in a generic and abstract way, demonstrated by an implementation in the functional array language Futhark. Our approach allows for irregular keys, such as strings, by representing them as an offset and length into a context array of characters, although the interface does allow mismanagement of said context that is not detected statically.

Our performance evaluation shows that hash maps performs well compared to a binary search based implementations, but is much slower than state of the art GPU hash maps implemented in CUDA. The performance differences are partly due to the CUDA implementations being carefully tuned to take advantage of specific hardware properties, but also due to the CUDA implementation using a more efficient data structure—open addressing hash tables—which are difficult to implement efficiently in our data-parallel vocabulary. Furthermore, our two-level hash maps use more memory than open addressing and will most likely lead to two cache misses per lookup. Unlike open addressing which will most likely only lead to one cache miss per lookup. Finally we have briefly discussed the problems of implementing efficient concurrent data structures in functional array languages, for which the central challenge seems to be allowing some forms of controlled nondeterminism. While the two-level hash map construction can be expressed using the data parallel vocabulary, it is inherently less efficient than constructing an open addressing hash table. However, the efficient implementation of the latter requires nondeterministic operations that lie outside of the available high level vocabulary.

[2] The implementation can be found at https://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/code/nesl/nesl/examples/hash-table.nesl.

References

1. Abrahamsen, M., Thorup, M.: Static dictionary using two-level hashing (2020), course notes for NDAB18001U Randomised Algorithms for Data Analysis (RAD), 2022
2. Awad, M.A., Ashkiani, S., Porumbescu, S.D., Farach-Colton, M., Owens, J.D.: Analyzing and implementing GPU hash tables. In: SIAM Symposium on Algorithmic Principles of Computer Systems. pp. 33–50. APOCS23 (Jan 2023). https://doi.org/10.1137/1.9781611977578.ch3, https://escholarship.org/uc/item/6cb1q6rz
3. Bailly, L., Henriksen, T., Elsman, M.: Shape-constrained array programming with size-dependent types. In: Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing. pp. 29–41. FHPNC 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3609024.3609412, https://doi.org/10.1145/3609024.3609412
4. Blelloch, G.E.: NESL: A nested data-parallel language (version 3.1). Citeseer (1995)
5. Carter, J., Wegman, M.N.: Universal classes of hash functions. Journal of Computer and System Sciences **18**(2), 143–154 (1979). https://doi.org/https://doi.org/10.1016/0022-0000(79)90044-8, https://www.sciencedirect.com/science/article/pii/0022000079900448
6. Chakravarty, M.M., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating Haskell array codes with multicore GPUs. In: Proceedings of the sixth workshop on Declarative aspects of multicore programming. pp. 3–14 (2011)
7. Elsman, M., Henriksen, T., Annenkov, D., Oancea, C.E.: Static interpretation of higher-order modules in Futhark: Functional GPU programming in the large. Proc. ACM Program. Lang. **2**(ICFP), 97:1–97:30 (Jul 2018). https://doi.org/10.1145/3236792, http://doi.acm.org/10.1145/3236792
8. Elsman, M., Henriksen, T., Serup, N.G.W.: Data-parallel flattening by expansion. In: Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming. pp. 14–24. ARRAY 2019, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3315454.3329955, http://doi.acm.org/10.1145/3315454.3329955
9. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with 0(1) worst case access time. J. ACM **31**(3), 538–544 (Jun 1984). https://doi.org/10.1145/828.1884, https://doi.org/10.1145/828.1884
10. Henriksen, T.: Design and Implementation of the Futhark Programming Language. Ph.D. thesis, University of Copenhagen, Universitetsparken 5, 2100 KÃ¸benhavn (11 2017)
11. Henriksen, T.: Bounds checking on GPU. International Journal of Parallel Programming (03 2021). https://doi.org/10.1007/s10766-021-00703-4
12. Henriksen, T., Hellfritzsch, S., Sadayappan, P., Oancea, C.: Compiling generalized histograms for GPU. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '20, IEEE Press (2020)
13. Henriksen, T., Thorøe, F., Elsman, M., Oancea, C.: Incremental flattening for nested data parallelism. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. pp. 53–67. PPoPP '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3293883.3295707, http://doi.acm.org/10.1145/3293883.3295707

14. Iverson, K.E.: A programming language. In: Proceedings of the May 1-3, 1962, spring joint computer conference. pp. 345–351 (1962)
15. Juenger, D., Iskos, N., Wang, Y., Hemstad, J., Hundt, C., Sakharnykh, N.: Maximizing performance with massively parallel hash maps on GPUs. https://developer.nvidia.com/blog/maximizing-performance-with-massively-parallel-hash-maps-on-gpus/ (2023), accessed: 2025-05-21
16. Jünger, D., Hundt, C., Schmidt, B.: WarpDrive: Massively parallel hashing on multi-GPU nodes. In: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018. pp. 441–450. IEEE Computer Society (2018). https://doi.org/10.1109/IPDPS.2018.00054, https://doi.org/10.1109/IPDPS.2018.00054
17. Jünger, D., Kobus, R., Müller, A., Hundt, C., Xu, K., Liu, W., Schmidt, B.: Warpcore: A library for fast hash tables on GPUs. In: 27th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2020, Pune, India, December 16-19, 2020. pp. 11–20. IEEE (2020). https://doi.org/10.1109/HiPC50609.2020.00015, https://doi.org/10.1109/HiPC50609.2020.00015
18. Khuong, P., Morin, P.: Array layouts for comparison-based searching. CoRR **abs/1509.05053** (2015), http://arxiv.org/abs/1509.05053
19. Munksgaard, P., Henriksen, T., Sadayappan, P., Oancea, C.: Memory optimizations in an array language. In: 2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC). pp. 424–438. IEEE Computer Society, Los Alamitos, CA, USA (nov 2022), https://doi.ieeecomputersociety.org/
20. NVIDIA: cucollections. https://github.com/NVIDIA/cuCollections (2020)
21. Paszke, A., Johnson, D.D., Duvenaud, D., Vytiniotis, D., Radul, A., Johnson, M.J., Ragan-Kelley, J., Maclaurin, D.: Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming. Proc. ACM Program. Lang. **5**(ICFP) (Aug 2021). https://doi.org/10.1145/3473593, https://doi.org/10.1145/3473593
22. Scholz, S.B.: Single assignment c: efficient support for high-level array operations in a functional setting. Journal of Functional Programming **13**(6), 1005–1059 (2003). https://doi.org/10.1017/S0956796802004458
23. Thorup, M.: High speed hashing for integers and strings. CoRR **abs/1504.06804** (2015), http://arxiv.org/abs/1504.06804
24. Tofte, M.: Essentials of Standard ML modules. In: International School on Advanced Functional Programming, pp. 208–229. Springer (1996)
25. Williams, J.: Algorithm 232: Heapsort. Commun. ACM **7**(6), 347–348 (May 2025). https://doi.org/10.1145/512274.3734138, https://doi.org/10.1145/512274.3734138