FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Project outside course scope

Niels G. W. Serup – `ngws@metanohi.name`

Extending Futhark with a `write` construct

Supervisors: Cosmin Eugen Oancea and Troels Henriksen

24 June 2016

Abstract

Futhark is a small functional programming language which can be compiled into efficient OpenCL code to be run on a GPU. I present a new construct `write` in the Futhark programming language.

I describe how I ended up focusing on designing the `write` construct through porting a variety of benchmarks to Futhark.

I show that:

- `write` can be used in meaningful programs.
- Uses of `write` in programs can be optimised within the existing Futhark compiler infrastructure.
- The Futhark compiler can compile `write`-using ported Futhark benchmarks into fast OpenCL code within a factor three of the original CUDA code speed (and in some cases faster).

Contents

Preface

This report is submitted in fulfillment of a 15 ECTS Masters project outside course scope in Computer Science (*Datalogi*) at the University of Copenhagen, for Niels G. W. Serup.

The original title for this project was "Efficient Histogram Computation on GPGPUs", focusing on adding a "histogram" construct to the Futhark programming language, but was changed during the project to instead focus on extending Futhark with a `write` construct.

1 Introduction

Futhark[10][8][9][1] is a data-parallel, purely functional array programming language that can be compiled into efficient OpenCL code to be run on a GPU. It is designed with interoperability in mind and has, for example, been used as the target of an APL compiler[5][7].

This section introduces the new `write` construct in Futhark.

1.1 Motivation

I spent the first part of the project porting benchmark programs to Futhark. Initially I focused on porting programs from the Accelerate[3][4][11][12] project:

Fluid simulation
: Real-time fluid dynamics simulation[13]. Consists of densities, forces, a linear solver with a stencil, and several wrapper functions utilising the linear solver multiple times for each frame.

Quasicrystals
: A small quasicrystal generation tool.

N-body simulation
: A brute force n-body simulation with each body represented by 10 floats.

For demonstration and testing purposes I have added image generation facilities to two of the benchmarks; see figure 1 for some visual results.

(a) A still frame from the fluid simulation benchmark.

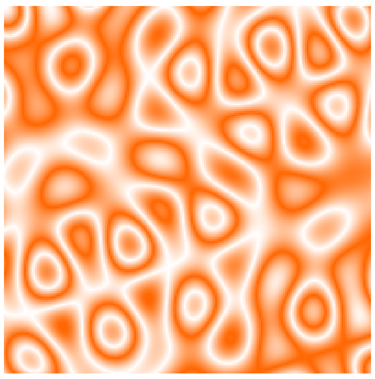
(b) A still frame from the quasicrystal benchmark.

Figure 1: Benchmark outputs

These three benchmarks were unproblematic to port to Futhark. I then went on to try implementing other programs:

- `filter`.
- A basic radix sort.
- A breadth-first search benchmark from the Rodinia[6] benchmark suite.

These three programs turned out to not be efficiently implementable in the then current Futhark language, since they all need a way to write to memory in a non-contigous way: Both programs require a way to, in parallel, write multiple values into an array at multiple indexes, with no further restrictions.

For example, Accelerate's version of radix sort has this final step in each iteration:

```
permute const v (\ix -> index1 (index!ix)) v
```

This creates a new array with multiple values moved to new indexes. Futhark lacked a bulk parallel primitive to do this.

1.2 Write

Consider this construct:

```
write(indexes, values, array)
```

Assume that all arguments are arrays, and that their respective types are the following:

`indexes`
: Array of unsigned 32-bit integers, 32-bit integers being the index type used by Futhark as of this writing.

`values`
: Array of some type `t`, picked by the programmer. The outer lengths of the `indexes` array and the `values` array must be equal.

`array`
: Array of the same type `t`, but with no restriction on the length.

This `write` construct has the following semantics:

```
1.  write(indexes, values, array) =
2.    for each (index, value) in zip(indexes, values):
3.      if index >= 0 && index < outer_length(array):
4.      then array[index] = value
5.      else ignore this (index, value) pair
```

Figure 2: The basic semantics of `write`.

That the `indexes` and `values` arrays must have the same length should be evident from line 2.

As arguably the simplest example of using `write`, see the Futhark program in figure 3.

```
fun [i32, n]
  main([i32, m] indexes,
       [i32, m] values,
       *[i32, n] array) =
  write(indexes, values, array)
```

Figure 3: Simple Futhark program using `write`.

If this program is given these three arrays as input

```
[3,   0,  1]
[99,  7, 32]
[ 0,  1,  2,  3,  4,  5]
```

then the final result would be `[ 7, 32, 2, 99, 4, 5]` since:

- the element at index 3 in the output array is replaced with the corresponding value 99;
- the element at index 0 in the output array is replaced with the corresponding value 7; and
- the element at index 1 in the output array is replaced with the corresponding value 32.

It is important to note that `write` accepts its indexes in any order, and that not all indexes for the entire input/output array range need to be given. This is what makes the construct useful.

To reduce memory copying, the input and output array inhibit the same memory, and all `write` operations occur in-place. This is also why the above example program has an asterisk in the `*[i32, n] array` argument: to ensure that it is unique, i.e. that the Futhark compiler allows an in-place update on the condition that the `array` variable is never used after the `write` operation.

Figure 2 shows that the construct is memory-bound: No calculations are made, so the bulk of the time will be spent accessing the arrays for reading and writing. This is good to know for optimisation purposes: Since memory is the bottleneck, we would like to a) reduce array accesses, and b) merge more calculations into `write` operations.

I have implemented the three programs in question using `write`; see sections 2 and 5.

The implementation has been made part of the official Futhark compiler distribution; see `https://github.com/HIPERFIT/futhark`.

1.3 Related Work

Since `write` can trivially be implemented in an imperative language by writing automatically in-place code, it is more interesting to compare Futhark's new construct with similar constructs in other parallel, functional languages.

The `write` construct appears to be unique to Futhark, but NESL, Accelerate, and Repa all have either `permute` or `backpermute` functions (or both), which are related to but less expressive than `write`.

1.4 Other Work

The major work *not* described in or related to the final project include:

- Implementing support for an `include` header in Futhark programs, thus enabling a Futhark program to be distributed across multiple files; see appendix B for a short description.
- Improving the automatic indentation feature of Futhark's Emacs text editing mode; see `https://github.com/HIPERFIT/futhark/blob/master/tools/futhark-mode.el`

1.5 Roadmap

The description in this section does not cover the final, extended design of the `write` construct. The rest of the report covers these details:

Section 2
Applications. Covers how `write` is used to implement several small algorithms such as radix sort and `filter`.

Section 3
Semantics & Type Checking. Describes the extended construct.

Section 4
Code Generation. Covers the internal representations of the construct and how they are used to generate OpenCL code.

Section 5
Breadth-First Search. Covers how the Rodinia breadth-first search benchmark has been ported to Futhark.

Sections 6 and 7
Optimisations. Goes into most detail with relation to fusion.

Section 8
: Design Phases. Outlines the evolution of the construct and why changes were made from the simple version described in this section.

Section 9
: Evaluation. Compares the Futhark ports to Rodinia's version on several benchmarks.

2 Applications

Once I had the initial design of `write` working, I started implementing programs with it.

2.1 Radix Sort

To get an idea of how to use the `write` construct in a larger setting, consider the least significant digit radix sort algorithm. This is a non-comparative integer sorting algorithm which works by grouping integers by their individual bits, one bit at a time. For simplicity, my implementation only handles 32-bit unsigned integers (type `u32` in Futhark), but could be extended. The algorithm can be summarised as this:

1. Input: An array of n unsigned 32-bit integers.
2. For each bit position in a 32-bit unsigned integer, starting from the least significant digit, do the following:
    (a) Sort the array based on the bit values at the current bit position, e.g. every integer with a 0 bit at the current position is moved to the beginning of the array, and every integer with a 1 bit is moved to the end of the array, while still preserving the original order of integers with the same current bit value.
3. Output: A sorted array of n unsigned 32-bit integers.

See the Futhark program in figure 4 for an example of how to use `write`. A similar approach has been taken by Blelloch[2], although with a `permute` construct.

```
fun [u32, n] main([u32, n] xs) =
  radix_sort(xs)

fun [u32, n] radix_sort([u32, n] xs) =
  loop (xs) = for i < 32 do
    radix_sort_step(xs, i)
  in xs

fun [u32, n] radix_sort_step([u32, n] xs, i32 digit_n) =
  let bits = map(fn i32 (u32 x) => i32((x >> u32(digit_n)) & 1u32), xs)
  let bits_inv = map(fn i32 (i32 b) => 1 - b, bits)
  let ps0 = scan(+, 0, bits_inv)
  let ps0_clean = map(*, zip(bits_inv, ps0))
  let ps1 = scan(+, 0, bits)
  let ps0_offset = reduce(+, 0, bits_inv)
  let ps1_clean = map(+ ps0_offset, ps1)
  let ps1_clean' = map(*, zip(bits, ps1_clean))
  let ps = map(+, zip(ps0_clean, ps1_clean'))
  let ps_actual = map(fn i32 (i32 p) => p - 1, ps)
  in write(ps_actual, xs, copy(xs))
```

Figure 4: Radix sort in Futhark using `write` at the end.

The Futhark program works like this:

1. The entry point function `main` calls the `radix_sort` function.
2. The `radix_sort` function uses the Futhark `loop` construct to loop over each of the 32 bit positions in 32-bit integers.
3. The `radix_sort_step` function sorts the `xs` array according to the current bit position.

Each step is equivalent to the concatenation of two `filter` operations: one that finds all numbers with a 0 bit in the current bit position, and one that finds all numbers with a 1 bit in the currrent bit position.

I will not go into most of the lines of the `radix_sort` function – see the Blelloch article for a mostly similar structure. The important part is the final expression:

```
write(ps_actual, xs, copy(xs))
```

At the point in the execution just before this expression, the `ps_actual` array describes a permutation of the `xs` array: each integer x_i in `ps_actual` is the new position for the value currently located at index i. The `write` then permutes `xs` accordingly.

What is interesting to note here is that if one intends to use `write` as a proxy for permuting an array, i.e. where the second and the third arguments are the same, one must explicitly `copy` the input array, so that the `write` does not simultaneously read from and write to the same array – having to do this is a result of the construct being in-place.

As we will see later on, `write` can also be used in more broad ways, where both its second argument and in-place functionality become useful.

2.2 `filter`

Futhark currently has a `filter` construct, but its implementation is sequential and slow. It is possible to implement a filter function in Futhark's outer language, although the fact that Futhark does not have polymorphic functions or accept anonymous functions as arguments limits how general it can be.

Let `bs` be a `bool` array of k elements. Let `xs` be an array of the same outer dimension, and of some type. The filter works by creating a new array of indexes by scanning with addition on the `bool` array, where `True` is 1 and `False` is 0, nulling indexes with irrelevant values, and using `write` at the end. This functions because the scan ignores all elements where the filter condition does not hold.

For example, take this call (where I have set `xs` to have type `[i32]`):

```
filter(fn bool (i32 x) => bs[x], xs)
```

We give it this data as input:

```
let bs = [True, True, False, True, False]
let xs = [   0,    1,     2,    3,     4]
```

Knowing the semantics of `filter` the result should be

```
[0, 1, 3]
```

Let us see if we reach the same result going through the outlined `write`-using algorithm:

1. First we convert `bs` into ones and zeroes:

   ```
   let flags_i = [1, 1, 0, 1, 0]
   ```

2. We then use a `scan` to get indexes:

   ```
   let is0 = scan(+, 0, flags_i) = [1, 2, 2, 3, 3]
   ```

3. These indexes include duplicate indexes for elements with a 0 in the `flags_i` array. We remove those with a multiplication:

   ```
   let is1 = map(*, zip(is0, flags_i))
           == [1 * 1, 1 * 2, 0 * 2, 1 * 3, 0 * 3]
           == [    1,     2,     0,     3,     0]
   ```

4. These indexes are 1-indexed and not 0-indexed. We subtract 1 from each:

   ```
   let is2 = [0, 1, -1, 2, -1]
   ```

   This has the added bonus of giving elements we want to ignore the index -1, which the `write` construct ignores.

5. We use `write` to get a new array:

   ```
   write([0, 1, -1, 2, -1],
         [0, 1, 2, 3, 4],
         replicate(filter_size, 0)) == [0, 1, 3]
   ```

See the full code in figure 5.

```
fun *[i32] i32_filter([bool, k] bs, [i32, k] ns) =
  let flags = map(fn bool (i32 n) => unsafe bs[n], ns)
  let flags_i = map(fn i32 (bool b) => if b then 1 else 0, flags)
  let is0 = scan(+, 0, flags_i)
  let filter_size = is0[k - 1]
  let is1 = map(fn i32 (i32 i, i32 f) => i * f, zip(is0, flags_i))
  let is2 = map(fn i32 (i32 i) => i - 1, is1)
  in write(is2, ns, replicate(filter_size, 0))
```

Figure 5: A `filter` function using `write`.

2.3 Segmented `scan`

This does not use the `write` construct, but is nevertheless important since it is a useful companion to `write` and is used by several Futhark programs in section 5.

Segmented scan uses two associative operators: An outer one which is used to scan on the source array and a mask array, and an inner one which is used to scan in each segment. This depends on user-supplied operators, which Futhark supports.

See figure 6 for an implementation in the outer Futhark language. The generality of such implementations also suffer from the lack of polymorphism and lambda arguments.

```
fun [i32, k] i32_plus_scan_segm([i32, k] array, [bool, k] mask) =
  let (arrayScanned, maskScanned) =
    unzip(scan(fn (i32, bool) ((i32, bool) arg0, (i32, bool) arg1) =>
                 let (a0, m0) = arg0
                 let (a1, m1) = arg1
                 let a' = if m1 then a1 else a0 + a1
                 let m' = m0 || m1
                 in (a', m'),
               (0, False),
               zip(array, mask)))
  in arrayScanned
```

Figure 6: A segmented `scan` function in Futhark.

3 Semantics & Type Checking

The final `write` construct has been extended to allow for tuples as arguments, resulting in this type:

```
write(
  ([i32, m]_1, ..., [i32, m]_t),
  ([v_1, m], ..., [v_t, m]),
  [v_1, n_1], ..., [v_t, n_t]
) -> [v_1, n_1], ..., [v_t, n_t]
```

The type `[a, m]` means an array with row type `a` and with length m.

The construct has the following semantics:

```
write(
  (indexes_1, ..., indexes_t),
  (values_1, ..., values_t),
  array_1, ..., array_t
) =
for each indexes_k, values_k, array_k:
  for each (index, value) in zip(indexes_k, values_k):
    if index >= 0 && index < length(array_k)
    then array_k[index] = value
    else ignore
return array_1, ..., array_t
```

The construct takes at least three arguments. The first two arguments are t-tuples, but can still also be a one-tuple or just a single element like in the simplified version described in section 1. The first argument is a tuple of arrays of 32-bit signed integers (`i32`), and the second argument is a tuple of arrays with row type `v`. The remaining arguments are t arrays with row type `v` and potentially different lengths than the second argument. The return types are the same as the t final argument arrays.

Note: Internally in the Futhark compiler, an expression can have multiple return values. In the external language, this is automatically represented as tuples unless the programmer takes care to manually de-tuple.

The reasons for choosing this representation is covered in section 8.

The construct works in-place: All changes are written to the input arrays `array_1, ..., array_t`, which are then also returned as the output arrays.

All indexes in each `indexes_k` array must be distinct to avoid indeterministic behavior when executing the memory writes in parallel. It is also invalid for the second and third arguments to in any way overlap (e.g. being the same array, or one of the two arguments being part of the array of the other argument); otherwise the generated OpenCL code might read from and write to the same array memory (but with different indexes) in parallel.

To have a safe implementation of in-place updating, the `write` construct depends on Futhark's uniqueness types, meaning the third argument to a `write` must always be consumable, and not be used more than once. Practically, this means that one must prefix an asterisk on function parameters that are to be used in `write` expressions as input/output arrays to inform the type checker that a certain array is supposed to be unique. If it is not, the compiler will complain.

The Futhark uniqueness types are designed with safe in-place updates in mind. Types prefixed with an asterisk must be unique, which requires passing three compile time checks. Of particular interest to `write` is the fact that if a value is consumed, it (or any of its aliases) cannot be used as a non-unique value anywhere prior to the consumption. This is why the `write` implementation can depend on uniqueness types to ensure no overlap, i.e. no aliasing, between the input/output arrays and the other arguments.

Futhark strives to be as safe a language as possible, meaning the programmer should not be able to enter programs that give indeterministic behaviour or crash improperly at runtime. To combat these issues, the language is strongly typed, and the compiler inserts runtime assertions when it cannot determine at compile time whether some expression is okay to run at a certain place in a program.

As described in section 1 there are several requirements to `write`:

1. Each indexes array must have no duplicate indexes – except for negative values, usually -1, which signal that an `(index, value)` pair should be ignored.
2. An indexes array or values array must not overlap with an input/output array. This is ensured through Futhark's uniqueness system.
3. All indexes arrays and values arrays must have the same outer dimensions.

The latter two can be ensured at compile time or through cheap runtime assertions, but the first cannot. It is possible to add runtime assertions to ensure that all indexes are unique and within bounds, but this would have a cost proportional to the array size and would discourage use of the `write` construct. Instead we have extended the semanics of the `write` construct to allow for ignoring the first requirement, and put the burden of ensuring correctness on the programmer. This means:

1. Bounds checking is done in the write kernel itself, and indexes out of bounds are simply ignored.
2. If an indexes array has duplicate entries, it is the fault of the programmer. We have chosen to let this part of the construct to be unsafe. This is not the *only* unsafe construct in the Futhark outer language: Both `scan` and `reduce` accept user-supplied operators which must be associative for the results to be deterministic, but which can be non-associative if the programmer is not on the lookout.

If all the above conditions hold, it is safe for the generated write OpenCL kernels to engage in concurrent reads and concurrent writes (CRCW).

4 Code Generation

The Futhark compiler can succesfully parse and generate OpenCL code for Futhark programs using `write`. The compiler goes through several phases. For this section to be clear, it is important to distinguish between the representations. There are roughly three different kinds:

- The syntactical construct `write` in the outer language. A programmer uses this in their programs.
- The Haskell data type `Write` in the compiler module `Futhark.Representation.SOACS.SOAC`. This representation is used by the compiler programmer for optimisations.
- The Haskell data type `WriteKernel` in the compiler module `Futhark.Representation.Kernels.Kernel`. This representation is used for generating OpenCL kernels as well as making some optimisations not well-suited for the SOAC representation.

4.1 The `Lambda` Structure

To program fusion within the existing Futhark compiler framework, I have chosen to represent `write` internally using an existing Futhark compiler type, the `Lambda`; see figure 7 and figure 8.

```
data LambdaT lore =
  Lambda { lambdaParams     :: [LParam lore]
         , lambdaBody       :: BodyT lore
         , lambdaReturnType :: [Type]
         }
```

Figure 7: The Lambda data structure of `Futhark.Representation.AST.Syntax` in the Futhark compiler.

```
data SOAC lore =
  ...
  | Write Certificates SubExp (LambdaT lore) [VName] [(SubExp, VName)]
```

Figure 8: The part of the `SOAC` data type defining the internal representation of `write`.

An expression

```
write(
  (indexes_1, ..., indexes_t),
  (values_1, ..., values_t),
  array_1, ..., array_t
)
```

is converted into a `Write` with the following `Lambda`:

```
fn (index_1, ..., index_t, value_1, ..., value_t) =>
  (index_1, ..., index_t, value_1, ..., value_t)
```

This lambda is then applied to to the tuple of arrays

```
(indexes_1, ..., indexes_t, values_1, ..., values_t)
```

to find each index and value.

The point of this structure is to store an easily-modifiable function body to be extended through fusion optimisations. Initially this is the identity function (as can be seen above), but it can be extended later on, for example in the case of fusion optimisations.

5 An Example: Breadth-First Search

I have ported the breadth-first search (or just "BFS") benchmark from Rodinia. This benchmark turns out to be very applicable to the `write` construct. The benchmark was ported from version 3.1 of the Rodinia suite.

The benchmark finds distances – the *cost* – to all nodes from a source node using breadth-first search. The purpose of the benchmark is to efficiently traverse all nodes using a GPU.

Rodinia's represents a graph $G(V, E)$ like the C code in figure 9, except with dynamic allocation and more verbose variable names.

```
struct Node
{
    int starting;
    int no_of_edges;
};

Node graph_nodes[no_of_nodes];
int graph_edges[edge_list_size];

bool graph_mask[no_of_nodes];
bool updating_graph_mask[no_of_nodes];
bool graph_visited[no_of_nodes];
int cost[no_of_nodes];
```

Figure 9: Rodinia's BFS data structure.

The Rodinia benchmark aims to support both an arbitrary number of edges for each node *and* store all edges for all nodes in contiguous memory. This has resulted in the above structure which should be interpreted in the following way:

- A `Node` has `no_of_edges` edges: `graph_edges[starting]`, `graph_edges[starting + 1]`, ..., `graph_edges[starting + no_of_edges - 1]`.
- A `Node` is referenced by its index in the `graph_nodes` array.
- An edge is an index into the `graph_nodes` array.
- `graph_mask`, `updating_graph_mask`, `graph_visited`, and `cost` are central to the algorithm but are not initialised from the dataset.

  `graph_mask`
  : Keeps track of which nodes can be used as start nodes when looking for new paths.

  `updating_graph_mask`
  : Temporary storage between CUDA kernels.

  `graph_visited`
  : Keeps track of which nodes have been visited. When all nodes have been visited, the algorithm terminates.

  `cost`
  : Stores the breadth-first distance from the source node.

Their algorithm structure is outlined in figure 10.

1. Initialise `graph_mask`, `updating_graph_mask`, `graph_visited`, and `cost`.
   - `graph_mask`, `updating_graph_mask`, and `graph_visited` are all initialised to `false` at all indexes.
   - `cost` is initialised to -1 at all indexes.
2. Pick a source node, from which the breadth-first distances to all other nodes are to be found.
3. Set initial values for the source node:
   - `cost[source] = 0;` – the distance between the source node and itself is 0.
   - `graph_mask[source] = true;` – the source node has been assigned a cost and can be used as a path origin.
   - `graph_visited[source] = true;` – the source node has been visited.
4. Do:
   (a) For each node, traverse in parallel all its edges, note which target nodes have been reached, and update their costs.
5. ... while at least one node has had its cost updated. Once all nodes have received a cost through some path, the algorithm terminates.

Figure 10: Rodinia's BFS algorithm.

It is a requirement that there is a path through edges from the source node to all other nodes. Please refer to the Rodinia article for further algorithmic details.

5.1 Parallelism in the Rodinia code

The Rodinia CUDA code invokes two kernels in the do-while loop. The second kernel merely sets some values for each thread. The first kernel has a for loop in the range of the number of edges for a node. This kernel is parallel in the sense that it is run by many threads, but the inner for loop is sequential and can have different dimensions in different threads.

One goal of porting this benchmark to Futhark was to extract parallelism from this irregular inner loop.

5.2 The Futhark Ports

I have developed three distinct Futhark ports, applying different parallelization techniques on each of them. I have chosen to use the same data layout as the Rodinia benchmark.

The Futhark ports have been tested and benchmarked against Rodinia on several datasets; see section 9 for measurements and general evaluation.

The three Futhark ports are:

`bfs_parallel_simple.fut`
: Uses *padding* to ensure equally sized arrays whose accesses can be parallelized.

`bfs_parallel_segmented.fut`
: Uses *segmented operations* to exploit parallelism without padding arrays.

`bfs_parallel_segmented_alternate.fut`
: Does the same as `bfs_parallel_segmented.fut`, but with larger and fewer helper arrays.

These approaches are covered in the following subsections. I have also written a sequential Futhark port, `bfs_sequential.fut`, which I will not discuss (since it is very slow).

5.3 The General Idea

How does the `write` construct apply to the BFS benchmark? Recall that the contiguous way of storing the edges means that a `Node` has the following edges:

`graph_edges[starting]`, `graph_edges[starting + 1]`, . . ., `graph_edges[starting + no_of_edges - 1]`.

The Rodinia algorithm updates the nodes at these edges, so we need a way to do that in Futhark. The simplest way is to, for each node, repeatedly use Futhark's index assignment in a sequential for loop:

```
...
  for starting <= i < no_of_edges do
    let id = graph_edges[i] -- Get a node index.
    -- Do some work with 'id'.
...
```

However, this is not parallel. We can instead use `write`:

```
...
  let edge_indexes = map(+ starting, iota(no_of_edges))
  let node_ids = map(fn i32 (i32 i) => graph_edges[i],
                     edge_indexes)
  in write(node_ids, ..., ...)
...
```

Both code snippets fetches all edges reachable from a node, but the one using `write` does it in parallel.

Unfortunately, it is not this simple. We would like for the `write` code snippet to be run for each node in parallel, but Futhark can only do this if all of the nodes' edge arrays have the same dimensions, since they can then be represented as one contiguous array. In other words, Futhark needs to flatten any nested parallelism. The following subsections outline two different ways of handling this.

5.4 Array Padding

We can use padding to force all subarrays to have the same dimensions. We can apply it like this:

1. Find the node with the maximum number of edges `e_max`.
2. Replace all `iota(no_of_edges)` with `iota(e_max)`. This ensures that all subarrays have the same dimensions.
3. Add an `if` expression to the map body, setting all edge indexes above the local `no_of_edges` to -1. Recall that if an index in a `write` expression is below 0, it is ignored.

The `bfs_parallel_simple.fut` port uses padding. The program structure is outlined in figure 11. Like in the Rodinia code there is an outer do-while loop. The `step` function is run in each iteration. `step` maps the `node_work` function on all active nodes, i.e. nodes which were visited for the first time in the previous iteration. `node_work` then calculates the indexes and costs for each new node reachable through an edge from the current node, setting both index and cost to -1 in case of padded values. This ensures easy parallelism at the cost of more memory usage.

1. While there are still unvisited paths in the graph:
    (a) Find the active node indexes.
    (b) Find the maximum number of edges $e_{\max}$ of any node.
    (c) Find the (index, value) permutation pairs by mapping over each active node:
        i. Calculate the new path costs and their related indexes.
        ii. Pad the arrays so they both have length $e_{\max}$. Pad with the value -1, which will be ignored when used with `write`.
    (d) `reshape` the mapping result into a single-dimensional array of length $e_{\max} \cdot$ number of active nodes.
    (e) `write` the changes in both path costs and graph mask (to keep track of which nodes have been visited).

Figure 11: The structure of `bfs_parallel_simple.fut`.

5.5 Segmented Operations

Another way to represent all node operations in a single array is to use segmented operations. This approach does not pad its subarrays, but instead uses another array to keep track of where subarrays start and end. This *mask array* is then used in conjunction with the edges array and segmented operations. The basic building block is the segmented scan, covered in section 2.

One of the segmented Futhark ports is `bfs_parallel_segmented.fut`. The program structure is outlined in figure 12. The `main` function has been left out because of its similarity to the one in `bfs_parallel_simple.fut`. Instead of having a separate function for node calculations, the calculations are squashed together using masks. The same operations are still executed, but through a segmented scan. This ensures parallelism at the cost of scans (whose implementation in the Futhark compiler is currently relatively expensive).

There is also `bfs_parallel_segmented_alternate.fut`, but that program is mostly the same. Contrary to `bfs_parallel_segmented.fut`, it creates the helper arrays once and then reuses them. The downside is that they are always the largest possible sizes, which is not the case in `bfs_parallel_segmented.fut`, where the sizes vary for each call to `step`.

1. While there are still unvisited paths in the graph:
    (a) Find the active node indexes.
    (b) Find the number of edges for each of the active nodes.
    (c) Create a mask of the edge segments for all nodes using `write`.
    (d) Generate the edge indexes for each segment using segmented scan with addition.
    (e) Do the same for the new path costs, using the same mask.
    (f) `write` the new costs and graph mask on the calculated indexes.

Figure 12: The structure of `bfs_parallel_segmented.fut`.

6 Optimisations

Since `write` is memory-bound, we would like to a) reduce array accesses, and b) merge more calculations into `write` operations. Overall I have attempted two different kinds of optimisations to the `write` construct:

Tweaking the generated kernel code to be more efficient
This does not attempt to reduce memory accesses or add more calculations to the mix, but instead tries to reduce inherent kernel overhead.

Fusing `write` with other Futhark constructs
This *does* attempt to reduce memory accesses and calculate more.

6.1 Tweaking the Kernel Code

The generated Futhark OpenCL kernel for a `write` expression handles only a single indexes-values pairs tuples per thread. This is not the only way to do it. Another way is to use chunked kernels where each thread loops through multiple tuples.

I have prototyped both approaches in CUDA (I find this easier to code by hand than OpenCL). I have simplified a `write` kernel into a kernel that writes to an array in global memory at a given index or given indexes, respectively. See figure 13 for a straightforward kernel and figure 14 for a chunked version using a grid-stride to achieve coalesced memory accesses.

```
__global__ void simple_kernel(
    int* I, int* V, int* A, size_t N_d, size_t K_d) {
  size_t i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < K_d) {
    A[I[i]] = V[i];
  }
}
```

Figure 13: Simple kernel.

```
__global__ void chunked_constants_kernel(
    int* I, int* V, int* A, size_t N_d, size_t K_d, size_t N_chunks_d) {
  size_t i_thread = blockIdx.x * blockDim.x + threadIdx.x;
  size_t n_threads = blockDim.x * gridDim.x; /* use grid-
stride */
  if (i_thread < N_chunks_d) {
    #pragma unroll
    for (size_t i = i_thread, j = 0; j < CHUNK_SIZE; i += n_threads, j++) {
      if (i < K_d) {
        A[I[i]] = V[i];
      }
    }
  }
}
```

Figure 14: Chunked kernel.

The argument for using the chunked version is that fewer threads need to be created, since a thread is used for the same task multiple times (but with different virtual thread ids). This should lead to less overhead and faster execution.

I have measured the two versions on different dataset sizes. All datasets consist of random integers. See the results in figure 15.

N	K	Trivial version	Chunk size	Chunked version
10^7	10^6	1017.50	10	1048.40
10^8	10^7	9855.10	10	10083.20
10^8	10^7	9915.00	100	10222.60

Figure 15: A sample of measurements for the two kernel versions in microseconds. N is the outer length of the input/output array, K is the outer length of the indexes and values arrays. All values are averaged over 10 runs.

As can be seen in the results, the chunked version does not appear to be faster. This might be because the random global memory access is the real bottleneck, or it might be because of that particular graphics card (an NVIDIA GeForce

GTX 690). I cannot rule out that some kernel code tweaks will result in faster code, but I have yet to find such tweaks.

6.2 Fusion

Fusion is an approach whereby several array-using constructs are “fused” into a single construct. It is possible to fuse `write` with itself, with `map`, with `iota`, and with `replicate`. Read more about this in the next section.

7 Fusion

This section details which fusion optimisations are used by the Futhark compiler with relation to the `write` construct.

7.1 Map-Write Fusion

Consider these patterns:

1. `write(map(fn (x) => ..., source), values, array)`
2. `write(indexes, map(fn (x) => ..., source), array)`

In each case a `write` takes the result of a `map` as its argument in one of its fusable arguments (i.e. not `array`). Since indexes and values are put into a single list when internalised, the two examples essentially describe the same scenario, so let us focus on case 1.

Case 1 is internalised into the following two SOAC Lambdas:

```
Map: fn (x) => ...
Write: fn (index, value) => (index, value)
```

The Map is applied on the `source` array, while the Write is applied on the result of the `map` and the `values` array.

We can fuse this into a single `write`. Consider this Write Lambda (the first element of the result pair is a function application):

```
fn (map_arg, value) => ((fn (x) => ...)(map_arg), value)
```

For example, if the `...` in the `map` body was `x * 2 + 1`, the Write Lambda would be

```
fn (map_arg, value) => ((fn (x) => x * 2 + 1)(map_arg), value)
```

i.e.

```
fn (map_arg, value) => map_arg * 2 + 1, value)
```

This also extends to `write` calls with multiple indexes and values, and to `map` calls with more than one source array.

This is an optimisation because it reduces the number of intermediate array accesses and moves more calculations into a `write` body.

Requirements: To fully remove the `map` from the program and fuse it into the write, the `write` must consume all `map` outputs and `write` input/output arrays.

As an example, consider this `write` expression:

```
write([1, 3], map(*3, [33, 333]), array)
```

This will end up with this internal representation:

```
fn (index, value) => (index, value * 3)
```

and be mapped on `zip([1, 3], [33, 333])`.

As an example of what *cannot* result in valid map-write fusion, consider this expression:

```
let indexes' = map(fn i32 (i32 i) => unsafe array[i], indexes)
in write(indexes', values, array)
```

If the `map` were to be fused into the `write`, the `write` would risk reading from and writing to `array` at the same time.

7.2 Write-Write Fusion

Consider this code snippet:

```
write(indexesA, valuesA, arrayA)
write(indexesB, valuesB, arrayB)
```

These are two independent `write` calls. They can be fused into a single `write`:

```
write((indexesA, indexesB), (valuesA, valuesB), arrayA, arrayB)
```

By itself this is not a huge optimisation. It means less overhead from starting kernels, but it still has the same number of global memory accesses. However, write-write fusion might enable more map-write fusion in the case of the same map result being used as input in both writes.

This extends to an arbitrary number of `write` calls.

Requirements: To ensure that no output from one `write` is used as input to another `write` once fused, the dependencies must be horisontal, i.e. no producer-consumer relationship can be present.

Here is an example of two `write` expressions which cannot be fused:

```
let array' = write(indexes1, values1, array)
in write(indexes2, values2, array')
```

This exhibits a producer-consumer relationship, as the output of the first `write` is used as input to the second `write`. If this was fused into a single `write`, both `(index, value)` pairs would target the same piece of memory in the same kernel thread.

7.3 Replicate Removal

This can also be called fusion, but we will call it removal to differentiate it from SOAC-SOAC fusion.

Consider this expression:

```
write(indexes, replicate(k, val), array)
```

This expression sets all elements in `array` at `indexes` to `val`. However, it is inefficient to create an entire array from the `replicate` expression when it can instead me merged into the Write Lambda:

```
fn (index) => (index, val)
```

This Lambda still depends on being called with the `indexes` array, but that is the only input it needs; instead of reading from two arrays, it now only needs to read from a single array, since `val` is always the same.

For example, this program can have a `replicate` removed:

```
write([0, 2, 3], replicate(3, 99.9), array)
```

Internally this will be represented as this:

```
fn (index) => (index, 99.9)
```

7.4 Iota Removal

Consider this expression:

```
write(iota(k), values, array)
```

We would like to not have `iota(k)` be an actual array, but instead be made part of the `write` call to avoid the array overhead.

This is not doable with the SOAC `Write` representation since there is no way to encode the increasing index values $0, 1, \ldots, k-1$. We instead focus on performing the optimisation on the kernel representation of `write`: `WriteKernel`. This data structure also represents the `write` as a Lambda, but additionally takes a thread index in addition to the arguments from the Lambda in the `Write` SOAC.

The `iota(k)` expression can be mapped to this thread index variable. The above `write` expression is turned into this Lambda:

```
fn (thread_index, value) => (thread_index, value)
```

This is an optimisation because the thread index does not come from an array but instead from an OpenCL call, thus reducing the number of accesses to global memory.

This extends to more than single-element tuples.

As an extreme example, consider this:

```
write(iota(k), map(+4, iota(k)), array)
```

Internally the `map(+4, iota(k))` expression will actually just be an `Iota` expression, since the internal `Iota` is more powerful than the external one. This transforms into this body:

```
fn (thread_index) => (thread_index, thread_index + 4)
```

We have eliminated all array accesses, except for writing the results into `array`.

8 Design Phases

Throughout this project the design of the `write` expression has gone through several phases. This section outlines the changes and reasons.

8.1 Initial Design

Initially `write` would only accept single arrays as arguments, like this type:

```
write(
  [i32, m],
  [v, m],
  [v, n]
) -> [v, n]
```

Figure 16: Initial type.

The semantics are the same as in section 1. This design was chosen because it was simple.

8.2 Allowing Tuples as Arguments

Once the compiler could generate working code using the initial design, I extended the construct to take tuples as arguments, resulting in this type:

```
write(
  ([i32, m]_1, ..., [i32, m]_t),
  ([v, m]_1, ..., [v, m]_t),
  ([v, n]_1, ..., [v, n]_t)
) -> [v, n]_1, ..., [v, n]_t
```

Figure 17: Extended type.

This is very similar to the final design in the outer language, but is limited to three arguments: the indexes tuple, the values tuple, and the input/output arrays tuple.

Internally this was represented as a SOAC with a data field for each array tuple.

This change allows for better optimisations. Since the internal representation of `write` also supports tupled argumants, multiple `write` expressions can in some cases be merged together into a single `write`. This process, which we call write-write fusion, was covered in section 7.

8.3 Using a `Lambda` in the Internal Representation

To *actually* enable fusion within the existing Futhark compiler framework, I changed the internal representation – for both `Write` and `WriteKernel` – to use a Futhark `Lambda` data type to represent parts of a `write`. This did not change the construct in the outer language.

Since fusion is applied equally on both the indexes and values arrays, they were merged into a single list of fusable arrays. The list of input/output arrays was still stored by itself.

8.4 Not Using Tuples for the Input/Output Arrays

The final type of `write` is, as described in section 1, this:

```
write(
  ([i32, m]_1, ..., [i32, m]_t),
  ([v, m]_1, ..., [v, m]_t),
  [v, n_1]_1, ..., [v, n_t]_t
) -> [v, n_1]_1, ..., [v, n_t]_t
```

Constrast this to the type in figure 17 in which all input/output arrays must have the same dimensions, not just the same types. This is because Futhark puts restrictions on which values can be inside tuples: Specifically, all arrays in a tuple must have the same dimensions, since the memory is actually an array of element tuples. To not get affected by this existing Futhark core design choice, the final `write` construct "flattens" the third tuple argument.

The major internal change is that the list of input/output arrays now becomes a list of pairs of input/output arrays and their lengths, since every input/output array can now have its own length.

9 Evaluation

This section focuses on the testing and evaluation of Rodinia's Breadth-First Search benchmark. I have also tested radix sort, filter, and segmented scan, but I will not discuss those here.

All Futhark programs wer run on an NVIDIA GeForce GTX 690 (on an AMD Opteron 6274) using the `futhark-opencl` compiler from June 9. Refer to appendix C for details on obtaining the correct version of the compiler.

9.1 Testing

I have tested the correctness of my implementation on several datasets with input-output pairs:

4096nodes
: By a Rodinia random graph generation tool. 4096 nodes. Each node has below 10 edges.

512nodes_high_edge_variance.in
: By my program. 512 nodes. Each node has between 5 and 200 edges. Generated at random.

Note that Futhark and Rodinia use different data formats for both input and output, so I have had to create a couple of scripts to convert to and from these formats (refer to appendix C for details on this).

The tests show that all Futhark ports have the same output as the Rodinia benchmark for both datasets, both with Futhark's sequential C backend and Futhark's parallel OpenCL backend. Considering the random nature of both datasets, and the different cases they cover, I assume that the Futhark ports are correct.

9.2 Measurements

I have benchmarked the different Futhark ports on four datasets:

1Mnodes (shorthand: 1M)
: By a Rodinia random graph generation tool. 1 million nodes. Each node has below 10 edges. Distributed along with Rodinia 3.1.

high_edge_variance_100K (shorthand: H100K)
: By my program. Generated at random. 100,000 nodes. Each node has between 5 and 200 edges. The goal is to have a high edge variance, which might throw off Rodinia's implementation by providing a larger degree of irregularity than in Rodinia's 1M benchmark.

few_nodes_many_edges_1000n_1000e (shorthand: E1K)
: By my program. Generated at random. 1000 nodes. Each node has 1000 edges. The goal is to have many edges, thereby requiring an implementation to also extract parallelism from them. This might throw off Rodinia's implementation by not giving it enough node parallelism to exploit, since Rodinia handles all edges in a sequential loop.

higher_edge_variance_5K (shorthand: HH5K)
: By my program. Generated at random. 5000 nodes. Each node has between 5 and 5000 edges. The goal is the same as H100K, except more extreme.

I think these datasets provide an okay spread.

Rodinia also provides the `4096nodes` and `65536nodes` datasets, but they are fast in all implementations and will not be covered in this section because of their small sizes. Refer to appendix A for all measurements.

Each port has been benchmarked in four ways for each dataset:

Clean (shorthand: C)
: With no write-specific optimisations enabled.

Fusion (shorthand: F)
: With write-write fusion and map-write fusion enabled.

Elimination (shorthand: E)
: With iota elimination and replicate elimination enabled. Also a kind of fusion, but called elimination to distinguish it from the main fusion.

All (shorthand: A)
: With both fusion and elimination enabled.

As such, I have benchmarked one Rodinia version and twelve (3 ports $\times$ 4 optimisation ways) Futhark versions. The Rodinia measurements are averaged over 10 runs. The Futhark measurements are averaged over the default number of runs in the `futhark-bench` tool.

See figure 18 for the Rodinia measurements, and figures 19, 20, and 21 for the Futhark measurements.

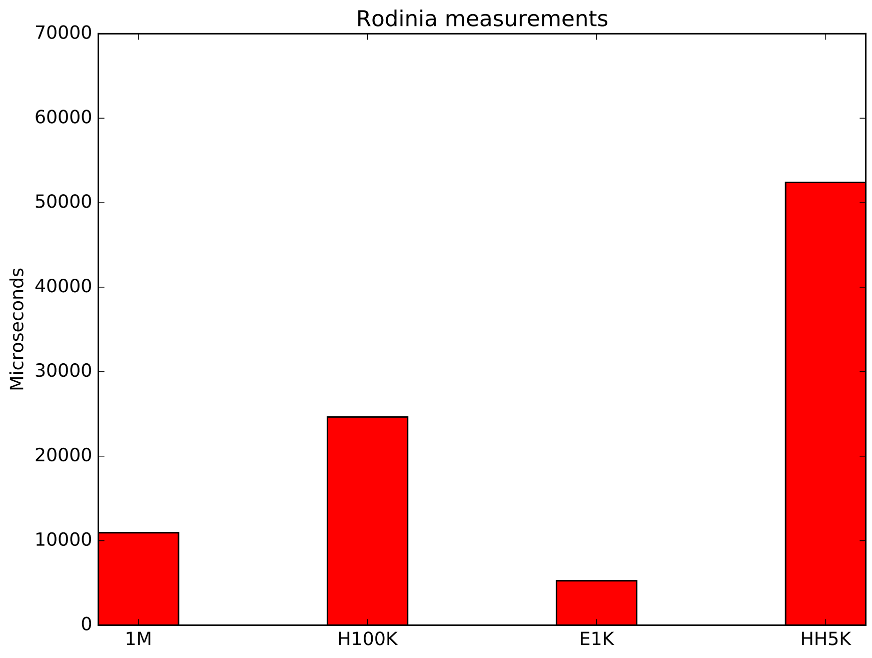

Figure 18: Rodinia measurements.

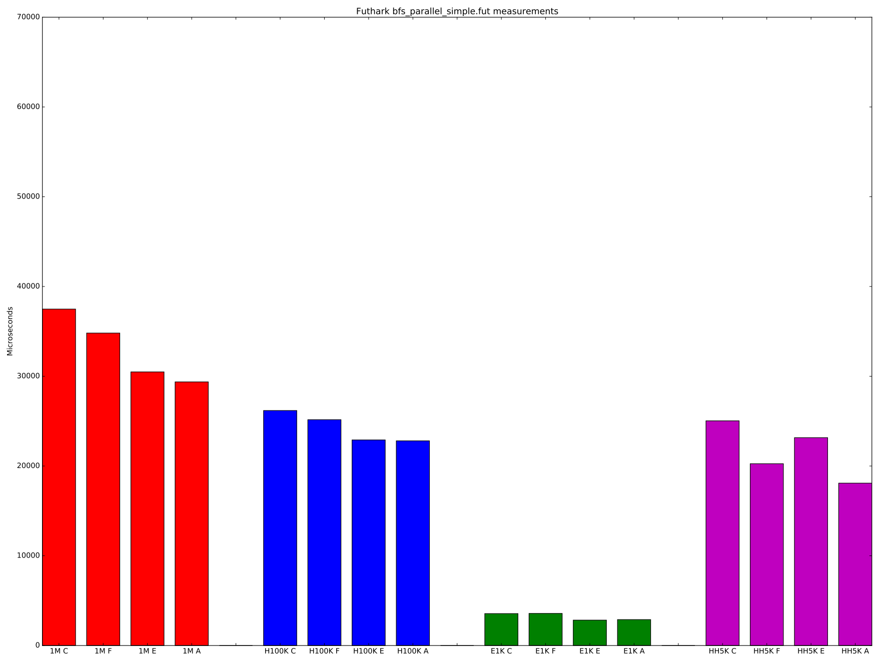

Figure 19: Futhark measurements for `bfs_parallel_simple.fut`.

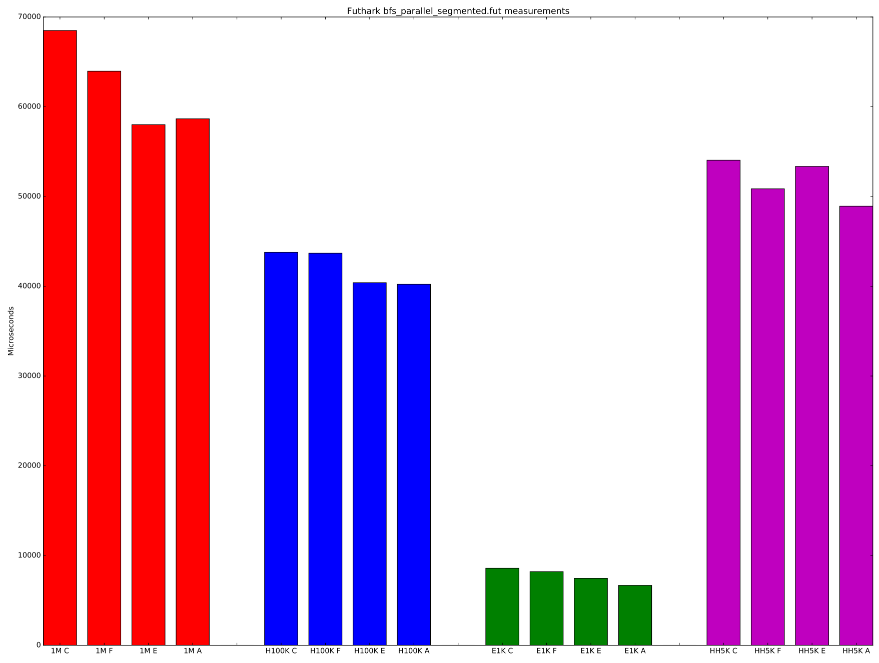

Figure 20: Futhark measurements for `bfs_parallel_segmented.fut`.

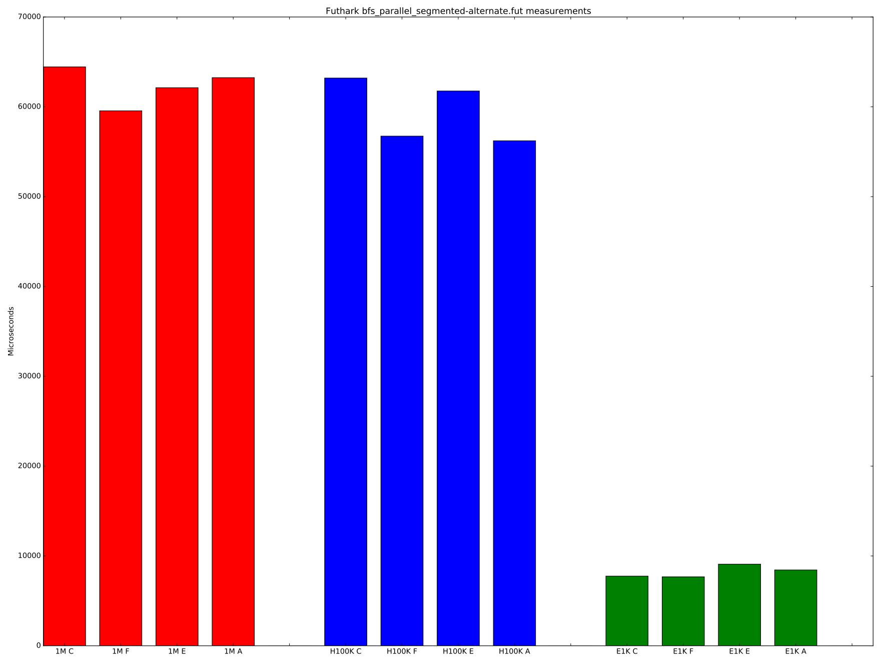

Figure 21: Futhark measurements for `bfs_parallel_segmented_alternate.fut`. There are no bars for the HH5K dataset because the program ran out of memory and did not find a result.

9.3 Discussion

In all cases the `bfs_parallel_simple.fut` port (the one using padding) is the fastest. For the datasets 1M, H100K, and HH5K, the port is fastest with all optimisations enabled. For the E1K dataset, enabling all optimisations gives a small penalty compared to just enabling the elimination optimisations – 2889.50 microseconds compared to 2831.10 microseconds. I have chosen to consider this difference neglible and declare `bfs_parallel_simple.fut` with all optimisations enabled the preferrable way.

The E1K dataset is smaller than the other datasets. It would have been nice to have it be larger, but I had problems with the generated Futhark programs crashing on larger sizes, so I stuck with 1000 nodes and 1000 edges.

I have made other discoveries:

- `bfs_parallel_segmented.fut` is always faster than `bfs_parallel_segmented_alternate.fut` on the four benchmarks. It should be noted, however, that the alternate version is actually the fastest of the two on the smaller 4096nodes and 65536nodes benchmarks.

- `bfs_parallel_segmented_alternate.fut` on the 1M benchmark is on average between 1-4 milliseconds slower with all optimisations enabled than with just one of the groups of optimisations enabled.

I have plotted the performances of `bfs_parallel_simple.fut` and Rodinia together in figure 22.

The datasets generated by my graph generator were partly designed to expose unparallelised code in Rodinia and highlight parallelised code in Futhark. This appears to have paid off, since Futhark is faster than Rodinia in two of my benchmarks and very close in another one.

Rodinia is a factor of three faster than futhark on the 1M benchmark. This benchmark is admittedly the least contrived one of the four. It shows that Futhark is fast, but not quite as fast as Rodinia for a general case.

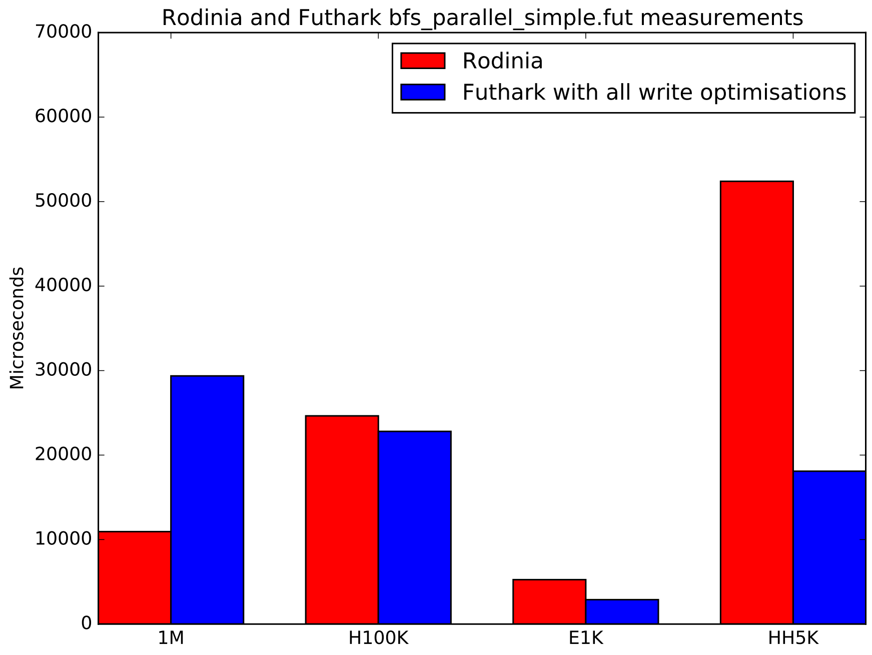

Figure 22: Rodinia compared with `bfs_parallel_simple.fut` and all `write` optimisations enabled.

10 Future Work

The `write` construct has been succesfully implemented, but it can be improved.

- Currently, it is possible to implement `filter` in the outer language using `write`, but this does not make for reusable code, since there is no polymorphism. It would be great if someone made a proper `filter` using `write`, either implementing it directly in the compiler as a transformation, or making a library using Futhark's new structures, or a combination.

- Revisit the BFS ports with segmentation in the future. If the Futhark compiler infrastructure has changed at that point, the ports might be faster or slower, since they depend a lot on how especially scan is implemented.
- Find more `write`-applicable benchmarks and port them to Futhark.

Bibliography

[1] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. Finpar: A parallel financial benchmark. *ACM Trans. Archit. Code Optim.*, 13(2):18:1–18:27, June 2016.

[2] Guy E. Blelloch. Prefix sums and their applications. *Technical Report CMU-CS-90-190*, 1993.

[3] Manuel M T Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming*. ACM, January 2011.

[4] Robert Clifton-Everest, Trevor L McDonell, Manuel M T Chakravarty, and Gabriele Keller. Embedding Foreign Code. In *PADL '14: The 16th International Symposium on Practical Aspects of Declarative Languages*, LNCS. Springer-Verlag, January 2014.

[5] Martin Elsman and Martin Dybdal. Compiling a subset of apl into a typed intermediate language. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 101:101–101:106, New York, NY, USA, 2014. ACM.

[6] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. Apl on gpus: A tail from the past, scribbled in futhark. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, FHPC 2016, pages 38–43, New York, NY, USA, 2016. ACM.

[8] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. Size slicing: A hybrid approach to size inference in futhark. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 31–42, New York, NY, USA, 2014. ACM.

[9] Troels Henriksen and Cosmin E. Oancea. Bounds checking: An instance of hybrid analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 88:88–88:94, New York, NY, USA, 2014. ACM.

[10] Troels Henriksen and Cosmin Eugen Oancea. A T2 Graph-reduction Approach to Fusion. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 47–58, New York, NY, USA, 2013. ACM.

[11] Trevor L McDonell, Manuel M T Chakravarty, Vinod Grover, and Ryan R Newton. Type-safe Runtime Code Generation: Accelerate to LLVM. In *Haskell '15: The 8th ACM SIGPLAN Symposium on Haskell*, pages 201–212. ACM, September 2015.

[12] Trevor L McDonell, Manuel M T Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP '13: The 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2013.

[13] Jos Stam. Real-time fluid dynamics for games. `http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf`, 2003.

A Measurements

This section contains the detailed measurements for the Breadth-First Search benchmark.

Dataset	Rodinia
4096nodes	414.30
65536nodes	875.90
1Mnodes	10942.00
high_edge_variance_100K	24641.20
few_nodes_many_edges_1000n_1000e	5257.40
higher_edge_variance_5K	52400.60

Figure 23: Rodinia measurements.

Dataset	clean	fusion	elimination	all
4096nodes	6253.10	6049.70	5315.80	4358.80
65536nodes	12421.00	13125.70	10739.30	10500.60
1Mnodes	37483.00	34810.90	30481.20	29369.20
high_edge_variance_100K	26180.10	25159.90	22903.90	22803.80
few_nodes_many_edges_1000n_1000e	3557.60	3575.20	2831.10	2889.50
higher_edge_variance_5K	25033.40	20254.10	23157.60	18091.90

Figure 24: Futhark measurements for `bfs_simple.fut`.

Dataset	clean	fusion	elimination	all
4096nodes	13102.30	10214.90	9347.90	10974.30
65536nodes	28692.30	22539.20	20887.40	20010.50
1Mnodes	68491.80	63959.20	58004.70	58657.00
high_edge_variance_100K	43784.90	43683.00	40397.40	40223.20
few_nodes_many_edges_1000n_1000e	8584.40	8208.50	7462.40	6672.80
higher_edge_variance_5K	54042.80	50856.60	53348.90	48918.70

Figure 25: Futhark measurements for `bfs_segmented.fut`.

Dataset	clean	fusion	elimination	all
4096nodes	3810.50	3135.20	3581.80	3401.70
65536nodes	9226.30	8703.20	9333.90	10394.30
1Mnodes	64441.90	59553.20	62124.50	63244.20
high_edge_variance_100K	63200.10	56735.90	61757.90	56216.90
few_nodes_many_edges_1000n_1000e	7754.30	7672.90	9084.60	8439.40
higher_edge_variance_5K	OOM	OOM	OOM	OOM

Figure 26: Futhark measurements for `bfs_segmented_alternate.fut`. “OOM” means that the program ran out of memory before finding a result.

B Adding an Include Header

I have extended the Futhark outer language with a new statement:

```
include module
```

The above will include all functions from whatever `module` is and make them available in the current Futhark program. All include headers must be at the top of the Futhark file, before any function declarations.

Currently, Futhark can only include files. You can include a file into your main Futhark program like this:

```
include other_file
```

The `.fut` extension is implied, so the above will include the file `other_file.fut`.

You can also include files from subdirectories:

```
include path.to.a.file
```

The above will include the file `path/to/a/file.fut`.

This extension added support for headers in general.

C Program Code

Futhark is a rapidly changing programming language. As of this writing, the outer language has received a new array annotations syntax, making much of my code prior to this change non-functional. For that reason, I refer to the `niels-write-bencharking` *git branch* in Futhark's *git repository* for a version of the compiler compatible with my produced Futhark code; see `https://github.com/HIPERFIT/futhark/tree/niels-write-benchmarking/`. This branch contains a compiler from 9 June 2016, as well as code used for benchmarking the BFS ports with different degrees of optimisations enabled.

All benchmarks can be found in the `futhark-benchmarks` git repository on `https://github.com/HIPERFIT/futhark-benchmarks`. However, this version has all benchmarks converted to the new syntax. See `https://github.com/HIPERFIT/futhark-benchmarks/tree/7844d9e71af9aaab8990bf9b1d84aff0c1f361a3` for a directory tree containing the benchmarks in the syntax compatible with the rest of this report.

Here are links to my ports:

BFS
`https://github.com/HIPERFIT/futhark-benchmarks/tree/7844d9e71af9aaab8990bf9b1d84aff0c1f361a3/rodinia/bfs` – also includes scripts for generating the large datasets and converting to and from the Rodinia and Futhark dataset formats.

Fluid simulation
`https://github.com/HIPERFIT/futhark-benchmarks/tree/7844d9e71af9aaab8990bf9b1d84aff0c1f361a3/accelerate/fluid` – also includes a GUI.

Quasicrystals
`https://github.com/HIPERFIT/futhark-benchmarks/tree/7844d9e71af9aaab8990bf9b1d84aff0c1f361a3/accelerate/crystal`

N-body simulation
`https://github.com/HIPERFIT/futhark-benchmarks/tree/7844d9e71af9aaab8990bf9b1d84aff0c1f361a3/accelerate/nbody`