UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's thesis

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

Memory Block Merging in Futhark

Supervisor: Cosmin Eugen Oancea

24 October 2017

Futhark is a purely functional array programming language designed to be compiled to efficient GPU code. It comes with an optimising compiler.

We describe and implement two memory optimisations aimed at reducing the memory footprint and copying overhead of Futhark programs.

We argue that the optimisations function as expected; describe their behaviour on many test programs; and discuss their current limitations.

Finally we show that, compared to the base compiler, these optimisations result in average memory footprint reductions ranging from 0% to 70% on a pre-existing suite of more than 30 Futhark benchmark programs ported from other array programming projects. However, we see regressions in the runtimes of the benchmarks, where we get average runtime reductions (speedups) ranging from -28% to +16%.

Contents

Preface

This thesis is submitted in fulfilment of a 30 ECTS master's thesis in Computer Science (*Datalogi*) at the University of Copenhagen, for Niels G. W. Serup.

(*Front page picture:* Like programs being optimised, hedgehogs will occasionally engage in a spiny merging (*reproduction*) that can produce better offspring (*transformed programs*). *Unlike* hedgehogs performing a merging, we would like programs to have a gestation period (*compilation time*) of less than one month.)

Chapter 1

Introduction

In the realm of array programming, there is a choice between imperative and functional programming:

- Imperative programming maps nicely to hardware: The compiler can translate a "write to this array" instruction into a "write to this memory location" machine instruction, allowing the programmer much control over how memory is used (depending on how low-level the imperative language is).
- On the other hand, functional programming tends to take much of the memory management burden off the programmer, instead leaving more for the compiler to optimise, which is possible because of the typically richer properties of functional languages.

For example, a compiler for a functional language can be made to check whether several expressions can share the same memory; we may be able to write a compiler optimisation that automatically determines if two arrays x and y residing in separate memory locations can be changed to reside in the same memory location without changing the meaning of the program – what we call *memory block merging*. Such an optimisation is not restricted to functional languages, but the flexibility in rearranging code parts inherent in functional code is helpful.

Functional languages typically use either garbage collection or region-based memory management for smart memory allocation and freeing. These approaches are too limiting in our usecase, for several reasons:

- The Futhark compiler targets GPUs, wherein memory allocation and deallocation support is more limited than in ordinary sequential code; for example, you cannot allocate memory from inside GPU kernels. In such a context, both garbage collection and region-based memory mangement would fail to reuse memory buffers within the same kernel. Furthermore, neither of these techniques provides a mechanism for optimising copying overheads.
- A core goal of the Futhark project is to enable interoperability between common programming languages and the Futhark language, which is achieved through foreign function interfaces. Garbage collection would complicate such interoperability solutions, for example when structures span two system heaps.

- *Practicalities:* The existing Futhark compiler infrastructure requires very limited runtime support, consisting mostly of wrappers for memory allocation and freeing, and built-in functions. We do not wish to complicate this design by extending it.

This thesis contributes two memory optimisations for reducing *memory usage* and *data copying*, and shows their applicability on pre-existing benchmark programs. Our main idea is to decouple the *allocation of memory* from the *creation of an array*, allowing allocations to be aggressively hoisted (outside GPU kernels), and allowing memory reuse across multiple arrays. We approach this by lifting the semantics of register allocation and coalescing[23] to work on arrays rather than scalars.

Compared to the base compiler, these optimisations result in average memory footprint reductions ranging from 0% to 70% on a pre-existing suite of more than 30 Futhark benchmark programs ported from other array programming projects. However, we see regressions in the runtimes of the benchmarks, where we get average runtime reductions (speedups) ranging from -28% to +16%.

Roadmap

- In chapter 2 we describe the background of the Futhark programming language, and the basic techniques on which our optimisations are based on.
- In chapter 3 we introduce the Futhark representation in which our optimisations operate.
- In chapter 4 we give an overview of the program analyses and program transformations that are part of our optimisations.
- In chapter 5 we describe our core program transformations for merging memory blocks.
- In chapter 6 we describe in detail what we call *enabling program analyses*: program analyses whose results are necessary for performing our program transformations.
- In chapter 7 we describe in detail what we call *enabling program transformations*: program transformations that are not absolutely necessary for the core transformations to work, but which will make the core transformations function better.
- In chapter 8 we attempt to take a more formal approach to describing the memory block coalescing optimisation covered in chapter 5, though we do not complete the attempt.
- In chapter 9 we list the currently known limitations of our optimisations.
- In chapter 10 we evaluate our algorithms and implementation; qualitatively by analysing the effects of the optimisations on purposely-written pathological test programs and four benchmark programs, and quantitatively by running a large automatic test suite. We also show the improvements and slowdowns we get by enabling our optimisations on more than 30 pre-existing Futhark benchmark programs.
- In chapter 11 we touch upon related work in the areas of register allocation and memory coalescing.

Chapter 2

Background

In this chapter we describe the basics of Futhark, as well as the basic techniques on which our later optimisations are based on.

Futhark

Futhark[14] is a small array programming language designed to be compiled to efficient GPU code, and comes with an optimising compiler. It is a purely functional, data-parallel array programming language with constructs for mapping, reducing, scanning, and other parallel array operations. It also supports non-parallel constructs such as if-then-else, for-loops, and while-loops. The Futhark project was started as part of the HIPERFIT[1] initiative in 2013.

As an example, the program in figure 2.1 computes the factorial number of the input number n of type `i32` (32-bit integer) using Futhark's parallel constructs. It computes the product of the integers $1..n$.

```
1 let main (n: i32): i32 =
2   reduce (*) 1 [1...n]
```

Figure 2.1: A Futhark program for calculating the factorial, using the parallel construct `reduce`.

On the other hand, the program in figure 2.2 computes the factorial number using a inherently sequential loop. It sets the initial value of the accumulator `f` to 1, and then computes `f * i` for every $1 \leq i < n + 1$.

[1] `http://hiperfit.dk/`

```
let main (n: i32): i32 =
  loop f = 1 for 1 <= i < n + 1 do
    f * i
```

Figure 2.2: A Futhark program for calculating the factorial, using the sequential construct `loop` with `for`.

The general look of the Futhark language matches that of other functional languages like SML and Haskell, except with fewer features. Notable differences include:

- All arrays must be *regular*, i.e. all rows of an array must have the same shape. For example, `[[0], [1, 2]]` is illegal, because the first row has length 1, and the second row has length 2 (if this cannot be checked at compile-time, it is checked at runtime).
- In-place updates are supported through Futhark's uniqueness types. For example, you can write `let xs' = xs with [i] <- y`, where `xs` is a $(n+1)$-dimensional array, `y` is a n-dimensional array, and `i` is an integer index, and get in-place updates in the generated code. This works under the uniqueness requirement (simplified) that `xs` is not used after this statement. A shorthand for this expression is `let xs[i] = y`, where `xs` is similar to `xs'` in the longform expression.
- If nothing else is specified, "running a Futhark program" means running the `main` function in the program.
- Comments start with `--` (two dashes).

The abstract syntax is covered in chapter 3; however, note that this is a grammar for a memory block-related intermediate representation, since our optimisations depend on such a representation, and not the source language.

Register Allocation and Coalescing

To determine if an array with some associated memory can be set to use the memory of another array without changing the result of executing the program, we want to (among other things) check if the current memory block and the potential memory block do not interfere. This is analogous to how liveness analysis is used in register allocation to map variables to registers; we lift it to cover arrays instead.

While register allocation may use graph coloring on an interference graph to determine a mapping from variable to register (or spill), we instead want to ultimately find a mapping from array variables to memory blocks.

We use an approach related to *linear scan register allocation*[23] to achieve this. The linear scan algorithm works by iterating over the liveness intervals of variables in a program, and maintaining an active list of register-associated active intervals, i.e. registers holding in-use variables at the current program point. At any given statement, the defined variable is associated with

a register that is not in the active list, and the registers associated with variables that are lastly used in that statement are removed from the active list.

Lifting this register allocation algorithm to work on arrays in Futhark is complicated by several factors:

- Futhark's support of nested bodies: The linear scan algorithm works on assembly-like flat programs and can thus use line numbers to denote locations in a program, but we need to decide on a more powerful representation for Futhark code; more about this in chapter 6.
- Register allocation works on a preset number of registers; while in a Futhark program, the number of seen arrays grows as the pass traverses the program, and our aim is merely to reduce the number of total memory blocks, without any hard limit. This does have the benefit of us not having to handle spilling.
- Scalars, which can be written to registers, cannot alias each other. On the other hand, arrays *can* (through e.g. slices), complicating the analysis.
- There are only small variations in the byte sizes of scalars – typically between 1 and 8 bytes – but arrays can have very different sizes, further complicating the analysis.

Register allocation can also cause the side-effect of eliminating the data copying inherent in `dst := copy(src)`-like instructions by assigning `dst` and `src` to the same register, and simplifying the instruction away. We do the same, but for arrays.

Chapter 3

Explicit Memory Annotations

The Futhark language has different representations in the compiler, suitable for different optimisations. In the so-called EXPLICITMEMORY representation, every array expression is associated with a memory block which was previously explicitly allocated in the program. Since memory block merging is about modifying the memory blocks of variables, we will perform our optimisations in this representation. This chapter first introduces our notation, then presents the grammar of the EXPLICITMEMORY representation, then discusses how delayed arrays are supported by means of index functions, and, finally, demonstrates on several examples how a memory-agnostic program is transformed into its EXPLICITMEMORY form.

Notation

We write $\overline{x}^{(n)} = x_1, \cdots, x_n$ to range over sequences of n objects (and $\overline{x}$ when the size does not matter).

When we write "source language", we refer to the programming language in which a programmer writes Futhark code. This is in contrast to the intermediate representations.

Also, when we write "the statement `x`", we mean the statement where `x` is created. We know this to be precise, since all variable names are unique in the representation.

EXPLICITMEMORY Syntax

We show the grammar of a simplified form of Futhark's EXPLICITMEMORY intermediate representation in figure 3.1, and the memory block index functions in figure 3.2. A program in this representation consists of a list of function definitions. Each function definition consists of a list of parameters and a body of **`let`** bindings (statements) and results.

We use c, f, m, d and x and y to range over constants, function, memory-block, scalar and array names, respectively, and v to range over any variable name.

f	::=	**id**	(Function name)
m	::=	**id**	(Memory name)
d	::=	**id**	(Scalar name)
x, y	::=	**id**	(Array name)
v	::=	$x \mid d \mid \ldots$	(Variable name)
c	::=	**const**	(Constant value)
t_0	::=	`i32` | `f32` | `bool` | ...	(Built-in types)
i	::=	**IxFun**	(Index function)
τ	::=	$t_0 \mid [d]\tau$	(Scalar/array type)
ρ	::=	$t_0 \mid \tau@m\{i\}$	(Types with memory info)
t	::=	$\rho \mid *\rho$	(Nonunique/unique type)
e^{alg}	::=	$c \mid d \mid e^{alg} \odot e^{alg}$	(Scalar expression)
$p^{\exists}$	::=	$v : \rho$	(Existential context)
p	::=	$(v : \rho) \mid \overline{p^{\exists}}(v : \rho)$	(Binding pattern)
P	::=	$\epsilon \mid FP$	(Program: sequence of functions)
F	::=	**let** $f\ \overline{p} : \overline{t} = b_{out}$	(Function definition)
b_{out}	::=	$\overline{v}$	(Tuple result)
	|	**let** $p = f\ \overline{v}$	(Function call)
	|	**let** $m =$ **alloc** e^{alg} **in** b_{out}	(Memory allocation)
	|	**let** $p = k$ **in** b_{out}	(Parallel let binding)
	|	**let** $p = e^{<b_{out}>}$ **in** b_{out}	(Sequential let binding)
b_{in}	::=	$\overline{v}$	(In-kernel tuple result)
	|	**let** $p = e^{<b_{in}>}$ **in** b_{in}	(In-kernel sequential let binding)
k	::=	**ker** $\left(\overline{d^i < d^N}^q\right) \left(\overline{x = y[\overline{d^i}^{s \leq q}]}\right) = b_{in}$	(Kernel map nest)
z	::=	$v \mid$ **copy** x	(Alias/deep copy)
$e^{<b>}$	::=	z	(Variable/deep copy)
	|	e^{alg}	(Scalar expression)
	|	**reshape** $\overline{d}\ x$	(Reshape dimensions)
	|	**rearrange** $\overline{c}\ x$	(Permute dimensions)
	|	$x[\overline{d}]$	(Array index)
	|	x **with** $[\overline{d}] \leftarrow z$	(In-place update)
	|	**scratch** $\overline{d}$	(New array)
	|	`[`i`..<`j`]`	(New array with range contents)
	|	**replicate** dv	(Replicate a value v for d times)
	|	**concat** $\overline{z}$	(Array concatenation)
	|	**if** d **then** b **else** b	(Branch)
	|	**loop** $\overline{p}^N = \overline{v}^N$ **for** $d < d$ **do** b	(Loop)

Figure 3.1: The grammar for the EXPLICITMEMORY intermediate representation.

A **let** binding consists of a left-hand-side pattern p and a right-hand-side expression. A pattern p consists of

- a fresh variable name v;

- an optional existential context $\overline{p^{\exists}}$, used e.g. by some **`if`** and **`loop`** expressions; and
- a type ρ, possibly a memory-annotated one of kind $\tau@m\{i\}$, consisting of a primitive scalar type, a memory block name, and an index function.

For simplicity, the grammar in figure 3.1 assumes that a pattern can only hold a single variable, but note that real Futhark programs can have expressions that return multiple values.

The grammar describes two kinds of bodies: b_{out} and b_{in}. This is because Futhark programs in this representation (among other representations) supports kernel map nests, in which only some expressions are supported. As such, we use b_{out} to refer to a body whose bindings can contain any expression, and b_{in} to refer a limited body inside a kernel. For example, a memory allocation (**`alloc`**...) can only appear in b_{out}, because dynamic allocations are not supported in GPU kernels.

There are two kinds of arrays in Futhark:

- Ordinary arrays: arrays that are created in full with **`scratch`**, **`copy`**, **`concat`**, `[`i`..<`j`]`, array literals, and a few other constructs.
- Delayed arrays: arrays that use the memory of another array, but with a different index function. These arrays can be created by aliasing operations like **`reshape`** and **`rearrange`**.

A note about **`scratch`**: This is *not* a construct available in the source language. Its purpose in the intermediate representation is to create an array of a certain size, but with no contents, so that non-scalar loops have somewhere to write their results.

Index Functions

In Futhark, an index function specifies how array elements are to be mapped to memory locations, providing the means for supporting delayed arrays without manifesting them in separate memory blocks.

Figure 3.2 presents the representation. The `Direct` constructor specifies the default, row-major mapping, which is used e.g. whenever a new array is produced by **`scratch`**.

The other constructors correspond to operators such as applying a statically-known permutation to an array's dimensions, reshaping an array's dimensions, and slicing an array.

$$
\begin{array}{lcllll}
IxFun & ::= & \texttt{Direct} & \overline{e^{alg}} & & \text{(Row major)} \\
& | & \texttt{Permute} & \overline{c} & IxFun & \text{(Permuting with constants)} \\
& | & \texttt{Reshape} & \overline{e^{alg}} & IxFun & \text{(Reshaping with scalar expressions)} \\
& | & \texttt{Slice} & \overline{\left(e^{alg}_{start}, e^{alg}_{length}\right)} & IxFun & \text{(Slicing with scalar expressions)}
\end{array}
$$

$$
\begin{array}{lcll}
DimIndex & ::= & e^{alg}_{index} & \text{(Fix index in this dimension)} \\
& | & (e^{alg}_{start\ offset}, e^{alg}_{element\ length}, e^{alg}_{stride}) & \text{(Take } \texttt{length} \text{ elements)} \\
\\
Slice & ::= & \overline{DimIndex} &
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{shape} & : & IxFun \to \overline{\texttt{i32}} \\
\texttt{slice} & : & IxFun \to Slice \to IxFun \\
\texttt{rebase} & : & IxFun \to IxFun \to IxFun
\end{array}
$$

Figure 3.2: Index function constructors, slice constructors, and helper functions.

For constructor applications, we will typically write i`->`$constructor_1\ \ arg_1$ instead of $constructor_1\ \ arg_1\ \ i$.

For example, let a be an array of type $[p][q]t_0@m\{i\}$. Then:

- Let b be a delayed array produced by the expression **`rearrange`** $(1,0)\ \ a$. Then b has the type $[p][q]t_0@m\{i\}$`->Permute`$(1,0)$.
- Let c be a delayed array produced by the expression $a[k]$. Then c has the type $[p][q]t_0@m\{i\}$`->`$Slice(k,1)$`->`$Slice(0,q)$, i.e. the slice in the outer dimension starts at offset k and has length 1, and in the inner dimension it starts at offset 0 and has the full length q.

We now discuss their semantics:

- `shape` i gives the shape of an array with the index function i.
- `slice` $i\ \ Slice$ creates a new index function where the $DimIndex$ parts are converted to the matching parts in the tuples of the index function $Slice$ construct.
- `rebase` $i_1\ \ i_2$ creates a new index function by substituting the $Direct$ part of i_1 with i_2.

Transformation Into ExplicitMemory Form

At some point in the compilation phase of a program, the Futhark compiler needs to transform the program from an IR without memory annotations *into* the ExplicitMemory representation. The former representation, which supports existential shapes and dependently-typed array

shapes [10, 12] is similar to EXPLICITMEMORY, except without **`alloc`** and **`scratch`** statements (and without memory blocks).

Intuitively, the transformation works by traversing the program statements and checking every statement:

- If the statement creates an ordinary array, assign a new memory block to it.
- If the statement creates a delayed array x of some other array y, assign the memory block of y to x as well, and construct a matching index function based on the aliasing operation in the statement.
- If the statement expression is scalar, nothing need to be changed.

For a very simple example, see figure 3.3. This program contains a single statement that computes the array `[`$0, 1, \ldots,$ `i` $- 1$`]`. When it gets transformed, this array gets an associated memory block and index function. This is an ordinary array, so it has a $Direct$ access. We informally write $m_{\texttt{a}}$ to mean "the allocated memory block containing the contents of the array `a`". For readability purposes, many of the code snippets presented in this thesis will diverge from the grammar by omitting the type of the variable in the **`let`** binding, and documenting its memory block, index function, and other attributes in a comment preceding the binding.

```
let main (i: i32): []i32 =
  let result = [0..<i]
  in result
```

```
let main (i: i32): []i32 =
  let m_result = alloc i · bytesize(i32)
  -- Memory: m_result; index function: Direct(i)
  let result = [0..<i]
  in result
```

Figure 3.3: A very small program that is transformed into its matching EXPLICITMEMORY form.

For a slightly larger example, see figure 3.4. This program has a delayed array through a **`reshape`** call, so two arrays end up having the same memory block, and the second array has a different index function.

```
let main [n] (ns: [n]i32): [1][n]i32 =
  let t0 = map (+ 1) ns
  let t0a = reshape (1, n) t0
  in t0a
```

```
-- Memory of ns: m_ns; index function: Direct(n)
let main [n] (ns: [n]i32): [1][n]i32 =
  let m_t0 = alloc n · bytesize(i32)
  -- Memory: m_t0; index function: Direct(n)
  let t0 = map (+ 1) ns
  -- Memory: m_t0; index function: Direct(1, n)
  let t0a = reshape (1, n) t0
  in t0a
```

Figure 3.4: A small program with a delayed array that is transformed into its matching EXPLICITMEMORY form.

The existing transformation also handles more complex cases where a delayed array creation re-indexes another delayed array.

Finally, EXPLICITMEMORY also supports *existential* memory blocks: memory blocks that are not directly associated with an allocation. These pop up in two places:

- *If-then-else.* If the `then` branch and the `else` branch return arrays in different memory blocks, then the entire expression gets assigned an existential memory block. The only purpose of this is to have *some* notion of a memory block that can encompass both results. This is represented as $p^{\exists}$ in the grammar.
- *Loops.* If a loop uses an array as part of its accumulator, then that array may be in different memory blocks across different iterations. For example, in figure 3.5, the accumulator array is `ys`, and can both refer to the initial value of `xs`, or the value of a previous iteration `ys1`, each of which are separate arrays and thus have separate memory blocks. To solve this, the memory block of the loop result `xs1` is existential.

```
let main (xs: [n]i32): [n]i32 =
  let xs1 = loop ys = xs for i < n do
    let ys1 = map (+ i) ys
    in ys1
  in xs1
```

Figure 3.5: A loop that will get an existential memory block when transformed to EXPLICITMEMORY form.

When the compiler transforms a loop to its ExplicitMemory form, it inserts a *double-buffer memory block* if the loop ends up using existential memory. A loop in the source language of the form

```
1 let main [n] (xs0: [n]i32): [n]i32 =
2   loop xs = xs0 for _i < n do
3     map (+ 1) xs
```

will be transformed into

```
1 let main [n] (xs0: [n]i32): [n]i32 =
2   loop {mem(xs0_mem_size) xs_mem;
3         [size]i32 xs} = {xs0_mem, xs0}
4     for _i:i32 < n do {
5       let res = kernel map ...
6       -- Memory of double_buffer_array: m_double_buffer_mem
7       let double_buffer_array = copy res
8       in {double_buffer_mem, double_buffer_array}
9     }
```

where `xs_mem` is an existential merge parameter with size `xs_mem_size` that can vary across iterations.

- On line 3 the merge paramaters are initialised to the `xs0` variants.
- On line 8 the body returns. These return values are used as the new merge paramater values in the next iteration of the loop.

The `let double_buffer_array = copy res` statement is inserted as a safety measure, and cannot *always* be optimised away. Throughout this thesis we will ignore the details of this transformation and informally refer to memory blocks like `xs_mem` as "accumulator memory blocks".

To clarify, this explicit allocation transformation in loops does *not* affect scalar loops or on-array-working loops that only perform in-place updates.

We have now presented the intuition behind transforming a program into its ExplicitMemory form. The next chapter describes how to perform analyses and transformations within this representation.

Chapter 4

Analyses and Transformations

In this chapter we give an overview the various program analyses and program transformations that are part of our optimisations. We think of them as three distinct parts:

Enabling analyses Program analyses that simplify the description and implementation of the transformations.

Enabling transformations Program transformations that enable the core transformations to extract more memory block mergings.

Core transformations Program transformations that actually end up merging memory blocks.

We describe them in detail in separate chapters, starting with the core transformations. We write $\mathcal{V}_k$ to denote a set of elements of kind k, and $\mathcal{V}_{[k]}$ to denote a set of sets.

All described analyses and transformations are fully implemented and working. See appendix A for the details, and chapter 10 for an evaluation.

Enabling Analyses

These include:

Memory block aliases
For all existential memory blocks, find what actual memory blocks they can refer to. This is not so much an analysis as it is a walkthrough of `if` and `loop` expressions, gathering all existential memory-related changes made by the explicit memory transformation.

The result of the analysis is a finite mapping $\mathcal{V}_m \xrightarrow{\text{fin}} \mathcal{V}_{[m]}$, i.e. from memory block name to a set of memory blocks names.

Last use analysis
Find all last uses of arrays and their memory blocks. This is analogous to last use analysis on registers, and is implemented as a top-down single passthrough of a program.

The result of the analysis is a mapping $\mathcal{V}_v \xrightarrow{\text{fin}} \mathcal{V}_{[m]}$, i.e. from variable name to a set of the names of the memory blocks lastly used at the binding of v (actually it is a bit more complicated than that; see the chapter for the details).

Interference analysis

Find all interferences between memory blocks. This is analogous to interference analysis on registers. We use the last use analysis for the interference analysis.

The result of the analysis is a mapping $\mathcal{V}_m \xrightarrow{\text{fin}} \mathcal{V}_{[m]}$, with the meaning that a memory block maps to its interfering memory blocks (by names).

We describe these in detail in chapter 6.

Enabling Transformations

We use two kinds of *hoisting* transformations:

- allocation hoisting; and
- allocation size hoisting.

These sound alike, but are actually used for different optimisations, although they partly use the same underlying approach. We use allocation hoisting to enable more memory block mergings in the *coalescing* optimisation, and use allocation size hoisting to enable more memory block mergings in the *reuse* optimisation.

We describe these in detail in chapter 7.

Core Transformations

We have two core transformations, inconspicuously named *memory block coalescing* and *memory block reuse*. We describe these in detail in the next chapter, chapter 5.

Chapter 5

Core Transformations

This chapter describes the transformations applied to a program as part of the memory block merging optimisations. The goal is two-fold:

- Reduce data copying in the generated programs, hopefully making the programs slightly faster in the process.
- Reduce memory usage in the generated programs, allowing for the loading of larger datasets.

As described in chapter 3, a memory block is a place in memory (unless it is existential) with a certain size. We take *memory block merging* to mean the case where an array variable x that uses memory block m_x in the original programs gets to, after an analysis that deems it okay, use a different memory block m_y in the transformed program; the memory blocks m_x and m_y are merged.

See chapter 8 for an attempt at a formal description of some of the transformations described in this chapter.

Memory Coalescing: Motivation and Intuition

We start by presenting the intuition behind the optimisations that we intend to formalise and implement, and the existing building blocks that they depend on.

Consider the Futhark program in figure 5.1.

```
1 let main (src: []i32): []i32 =
2   let dst = copy src
3   in dst
```

Figure 5.1: A small Futhark program with `copy`, ready for memory block merging.

This (extremely trivial) program takes an array `src` ("source") with type "single-dimensional array of 32-bit integers" as input, copies it into `dst` ("destination"), and returns `dst`. The `copy` construct is used in Futhark to create a unique array (meaning it is certain not to alias existing memory), which it accomplishes through a deep copy.

By quick inspection we see that the return value `dst` has the same contents as `src`, and that `src` is not used for anything else than its copying to `dst`; by this we reason that `dst` and `src` can share the same memory without changing the result of running the program. This is part of what the optimisations do.

In the program above, `src` and `dst` initially have separate memory blocks; but after the optimisation, `dst` is set to use the memory block of `src`.

Now consider the program in figure 5.2.

```
1 let main (ns: []i32): []i32 =
2   let t0 = map (+ 1) ns
3   let t1 = map (* 2) ns
4   let t2 = concat t0 t1
5   in t2
```

Figure 5.2: A small Futhark program with `concat`, ready for memory block merging.

This program first creates two new arrays `t0` and `t1`, both with the same length k, mapping $\lambda n \to n + 1$ and $\lambda n \to n \times 2$ on `ns`, respectively. It then concatenates them using the `concat` construct into a third new array `t2`, which is returned; see figure 5.3 for an illustration of its contents; `t2` has length $2k$, where the first k elements are the k elements from `t0`, and the last k elements are the k elements from `t1`.

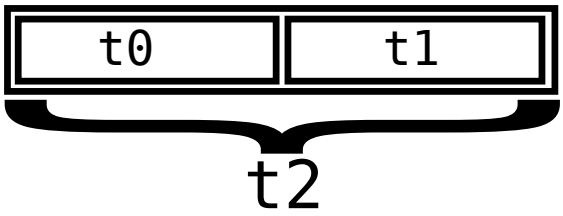


Figure 5.3: The contents of `t2`.

We look at this relationship and reason that, since `t2` consists of non-overlapping arrays `t0` and `t1`, these constituents should be able to store their results directly in the memory block of `t2` instead of first using their own memory blocks and then copying them into `t2`.

We first show the memory-annotated version of the original program in figure 5.4. As covered in chapter 3, this transformation into a program with explicit memory blocks already happens in the Futhark compiler; our goal is to take the program *in this representation* and transform it into a program *in the same representation* with less copying and with more efficient memory usage.

```
let main (ns: []i32): []i32 =
  let m_t0 = alloc k · bytesize(i32)
  let m_t1 = alloc k · bytesize(i32)
  let m_t2 = alloc 2k · bytesize(i32)
  -- Memory: m_t0
  let t0 = map (+ 1) ns
  -- Memory: m_t1
  let t1 = map (* 2) ns
  -- Memory: m_t2
  let t2 = concat t0 t1
  in t2
```

Figure 5.4: The original program with memory block annotations. Every array has its own memory block.

We then show our suggested transformed program in figure 5.5.

```
let main (ns: []i32): []i32 =
  let m_t2 = alloc 2k · bytesize(i32)
  -- Memory: m_t2; offset: 0
  let t0 = map (+ 1) ns
  -- Memory: m_t2; offset: k
  let t1 = map (* 2) ns
  -- Memory: m_t2; offset: 0
  let t2 = concat t0 t1
  in t2
```

Figure 5.5: The transformed program with memory annotations. Every array uses the same memory block, but with varying element offsets.

We set both `t0` and `t1` to use the memory block of `t2`, and remove the old allocations for $m_{\texttt{t0}}$ and $m_{\texttt{t1}}$, since they are not used anymore (in the compiler this is done through an existing simplifier pass).

Directly using the memory block $m_{\texttt{t2}}$ for both `t0` and `t1` is wrong; this would mean that `t1` writes to the same memory *area* as `t0`, thus overwriting it. As shown in the illustration, it is the second part of `t2` that contains `t1`, so we need a way to describe that in the transformed program. We use offsets to describe where in a memory block to store an array. Recall that $m_{\texttt{t2}}$ has space for $2k$ elements; by using the offset 0 for `t0`, and the offset k for `t1`, we ensure that `t0` will be stored in the first part of $m_{\texttt{t2}}$, and `t1` in the second part, thus keeping the semantics of `concat`.

This concept of offsets is part of the more general idea of *index functions* introduced in chapter 3, which supports more memory block merging cases. An offset can be represented in an index function by using `slice` on the outer dimension (where the offsetting occurs). We use offsets here to keep it simple.

We have now looked at how to merge memory blocks in simple cases of `copy` and `concat`. We look at one more small coalescing case before we move on to something larger. Consider the program in figure 5.6. This program contains an *in-place update* in the `t2` statement. We have annotated it with memory blocks.

```
1  let main [n] (i: i32, ns: [n]i32): [n][n]i32 =
2    let m_t0 = alloc n · bytesize(i32)
3    let m_t1 = alloc n · n · bytesize(i32)
4    -- Memory: m_t0; offset: 0
5    let t0 = map (+ 1) ns
6    -- Memory: m_t1; offset: 0
7    let t1 = replicate n (replicate n 0)
8    -- Memory: m_t1; offset: 0
9    let t2 = t1 with [i] <- t0
10   in t2
```

Figure 5.6: A small Futhark program with an in-place update, ready for memory block merging.

In line 1, the first occurence of `[n]` declares the shape variable `n`, which is then used in the one-dimensional type of `ns`, and the two-dimensional return type (after the colon).

In line 9, the value `t2` becomes the value of `t1`, except with `t0` written into index `i`. This is possible because `t1` (and thus `t2`) has the shape `[n][n]i32`, and `t0` has the shape `[n]i32`. Note that `t1` and `t2` use the same memory block m_{t1}; this is because the operation occurs in-place in Futhark programs.

We would like to get rid of m_{t0} and let `t0` instead use m_{t1}. We reason that `t0` can use the memory block of its destination if its offset is where it ends up anyway; this approach is illustrated in figure 5.7.

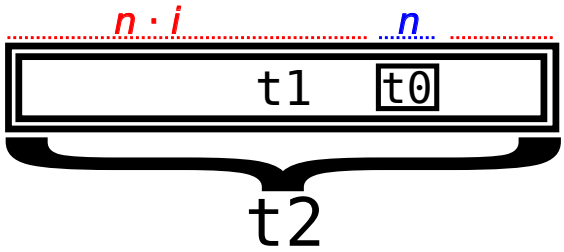


Figure 5.7: The contents of `t2`: `t1` except for the insertion of `t0` after $n \cdot i$ elements (and then `t1` again afterwards).

We show the transformed program in figure 5.8.

```
let main [n] (i: i32, ns: [n]i32): [n][n]i32 =
  let m_t1 = alloc n · n · bytesize(i32)
  -- Memory: m_t1; offset: n · i
  let t0 = map (+ 1) ns
  -- Memory: m_t1; offset: 0
  let t1 = replicate n (replicate n 0)
  -- Memory: m_t1; offset: 0
  let t2 = t1 with [i] <- t0
  in t2
```

Figure 5.8: A small Futhark program with an in-place update, with `t0` coalesced into $m_{\texttt{t1}}$.

We have removed the allocation of $m_{\texttt{t0}}$, and set `t0` to use $m_{\texttt{t1}}$.

The program transformations so far all share the same pattern: A source array is set to use the memory block of the destination it is put into. We refer to this kind of memory block merging as *memory block coalescing*, and our goal is to handle many more of these coalescings cases. We have shown examples for the three basic cases:

- **let** $x =$ **copy** y
- **let** $x =$ **concat** $y_0 \dots y_n$
- **let** $x[\overline{d}] =$ **copy** y

All of the previous examples have had a single memory coalescing. It gets more interesting for programs that have chains of all these patterns. We present one such program in figure 5.9.

```
let main [n] (ns: [n]i32): [][]i32 =
  -- Will initially be set to use the memory of t1.  Will end up using the
  -- memory of t3 through t2 through t1.
  let t0 = map (+ 1) ns

  -- Will use the second part of index 1 of the memory of t3 through the
  -- memory of t2.
  let t1 = copy t0

  -- Will use index 1 of the memory of t3.
  let t2 = concat ns t1

  -- Will be the only remaining memory block.
  let t3 = replicate 2 (replicate (n * 2) 0)
  let t3[1] = t2

  in t3
```

Figure 5.9: A Futhark program with all three coalescing-enabled constructs.

This program will have three coalescings:

- On line 15, `t2` is coalesced into $m_{\texttt{t3}}$.
- On line 11, `t1` is coalesced, via `t2`, into $m_{\texttt{t3}}$.
- On line 8, `t0` is coalesced, via `t1` and `t2`, into $m_{\texttt{t3}}$.

The simplifier will subsequently be able to remove $m_{\texttt{t0}}$, $m_{\texttt{t1}}$, and $m_{\texttt{t2}}$ from the program, greatly reducing memory usage and copying of data.

This gives an overview of what the optimisation is capable of, but note that real-world programs are *much* less clear-cut.

Memory Reuse: Motivation and Intuition

Next up we take a quick look at the intuition behind reusing non-interfering, lastly-used memory blocks, which we call *memory block reuse*. Consider the memory-annotated program in figure 5.10.

```
1  let main [n] (xs0: [n]i32, i: i32): [n]i32 =
2    let m_xs = alloc n · bytesize(i32)
3    let m_ys = alloc n · bytesize(i32)
4    -- Memory: m_xs
5    let xs = map (+ 1) xs0
6    let k = xs[i]
7    -- Memory: m_ys
8    let ys = replicate n k
9    in ys
```

Figure 5.10: A small Futhark program with a potential for memory reuse.

The program has two arrays, each with its own memory block:

`xs` is created at line 5 and lastly used at line 6.

`ys` is created at line 8 and lastly used at line 9.

We see that the two arrays have different *liveness intervals*; `xs` is lastly used before `ys` is created. Also, both arrays have the same size. By this we reason that it is okay to set `ys` to use the memory of `xs`; see the transformed program in figure 5.11.

```
let main [n] (xs0: [n]i32, i: i32): [n]i32 =
  let m_xs = alloc n · bytesize(i32)
  -- Memory: m_xs
  let xs = map (+ 1) xs0
  let k = xs[i]
  -- Memory: m_xs
  let ys = replicate n k
  in ys
```

Figure 5.11: The small Futhark program after memory reuse.

Similarly to memory coalescing, the memory reuse optimisation supports chains of memory mergings. We will look at more complex programs later in this chapter.

These are the program patterns we wish to handle in our optimisations, with the goal that the transformed programs use less memory, and maybe get small speedups.

Informal Algorithm: Memory Block Coalescing

We now define memory block coalescing.

A memory block $m_{\texttt{src}}$ can be set to use the memory block $m_{\texttt{dst}}$ if one of the statements below exist in the program:

1. **let** `dst` = **copy** `src`
   We know that the size of `dst` is always equal to the size of `src`.
2. **let** `dst` = **concat** ... `src` ...
   We know that the size of `dst` is equal to the sum of the sizes of `src` and of the other concatenated arrays.
3. **let** `dst`$[\bar{d}]$ = **copy** `src`
   We know that the size of `dst`$[\bar{c}]$ is always equal to the size of `src`, since the existing compiler infrastructure guarantees this.

We would like an optimisation that uses the memory block of the left-hand side for the right-hand side variable, i.e. we would like to store `src` directly in $m_{\texttt{dst + offset}}$, where `offset` depends on the case – and not in $m_{\texttt{src}}$, which would require an extra copying operation and is thus suboptimal.

We define five safety conditions that we informally argue are sufficient for guaranteeing safety for a coalescing relation rooted in one of the three cases presented above:

1. The memory of the right-hand side variable is in its last use, i.e. neither `src` nor any of the variables aliased with it is used on any execution path following the coalescing statement. If the program has not been optimized for memory reuse, this is equivalent to saying

that $m_{\texttt{src}}$ is not used beyond the statement in question. This condition is not absolutely necessary, but it captures the common case and significantly simplifies our analysis.

2. The allocation of $m_{\texttt{dst}}$ occurs before the creation of `src`, i.e. the first use of $m_{\texttt{src}}$. This ensures that the contents of `src` can be stored in a previously allocated memory block, i.e. one that exists at the creation of `src`.

3. There is no use of the left-hand side memory block $m_{\texttt{dst}}$ (and of any other memory blocks in which $m_{\texttt{dst}}$ is transitively coalesced into) after the creation of `src` and before the current statement. This ensures that `src` does not implicitly depend on `dst`, which might change the semantics of the transformed program, e.g. because the data of `dst` might be prematurily overwritten with the data of `src`.

4. `src` is or can be traced back (through a chain of aliasing operations) to a newly created (i.e. by expressions such as **`scratch`**, **`copy`**, and **`ker`**) array `src0`. This condition limits the search, since it ensures that, at the creation of `src0`, there are no other variables that aliases `src0`.

5. The new index function of `src` only uses variables declared prior to the first use of $m_{\texttt{src}}$. This ensures that the new index function is valid.

We show four example programs that describe the importance of the safety conditions – excluding safety condition 1 since its purpose is to limit our search rather than secure our transformation – in figure 5.12, figure 5.13, figure 5.14, and figure 5.15, respectively.

```
1 let main [n] (i: i32, ys: [n]i32): [n][n]i32 =
2   let xs = reshape (n, n) [0..<(n * n)]
3   let xs[i] = ys
4   in xs
```

Figure 5.12: A program where safety condition 2 makes sure that an invalid memory coalescing does not occur.

We cannot fulfill safety condition 2 in figure 5.12 in the potential coalescing on line 3, since `ys` is allocated before the function body is run, so `xs`, which is created in the body, can never be allocated before `ys`.

```
1 let main [n] (xs: *[n][n]i32, ys0: [n]i32, i: i32): ([n][n]i32, i32) =
2   let ys = map (+ 1) ys0
3   let zs = map (+ ys[0]) xs[i]
4   let xs[i] = ys
5   in (xs, zs[i])
```

Figure 5.13: A program where safety condition 3 makes sure that an invalid memory coalescing does not occur.

We cannot fulfill safety condition 3 in figure 5.13 in the potential coalescing on line 4, since `xs` is used while `ys` is live. If we do the coalescing anyway, we will get an incorrect program where `zs` maps over the contents of `ys` instead of the original contents of `xs[i]`.

```
1 let main [n] (xs: *[n][n]i32, zs0: [n][n]i32, i: i32, j: i32): [n][n]i32 =
2   let zs = map (\z -> map (* 3) z) zs0
3   let ys = zs[i]
4   let xs[j] = ys
5   in xs
```

Figure 5.14: A program where safety condition 4 makes sure that an invalid memory coalescing does not occur.

We cannot fulfill safety condition 4 in figure 5.14 in the potential coalescing on line 4, since `ys` is a slice of `zs` and is thus part of the memory of `zs`. If we try to do the coalescing anyway, we will end up with the nonsensical result that `ys` is both contained in the memory of `zs` *and* the memory of `xs`.

```
1 let main [n] (ns: [n]i32, t1: *[n][n]i32, i0: i32): [][]i32 =
2   let t0 = map (+ 1) ns
3   let i1 = t0[i0]
4   let t1[i1] = t0
5   in t1
```

Figure 5.15: A program where safety condition 5 makes sure that an invalid memory coalescing does not occur.

We cannot fulfill safety condition 5 in figure 5.15 in the potential coalescing on line 4, since the index function of `t1` includes `i1`, which depends on the result of `t0`; in a coalescing of `t0` into $m_{\texttt{t1}}$, `t0` would need to update its index function to include `i1`, which is not possible.

For each potential coalescing, we need to check these safety conditions, *and* we need to support chains of coalescings like that in figure 5.9. We present our basic algorithm for finding coalescings in figure 5.16.

1. Let `coalesced_intos` be a mapping $\mathcal{V}_a \overset{\text{fin}}{\to} \mathcal{V}_{[a]}$, i.e. from variable name to variable names. Initialize it to be empty. This is used if there are chains of coalescings, and a previous coalescing needs to be updated to use a new memory block.

2. Let `result` be a mapping $\mathcal{V}_a \overset{\text{fin}}{\to} \mathcal{V}_m \times \mathcal{V}_i$, i.e. from statement variable name to memory block and index function. Initialize it to be empty.

3. Traverse every statement in the program in a top-down fashion. For each **`copy`**, **`concat`**, or in-place update statement **`let`** `dst` $= e$ with memory block $m_{\texttt{dst}}$:

   (a) For each source array `src` with memory block $m_{\texttt{src}}$ in e:

      i. Check that all safety conditions are satisfied w.r.t. `dst` and `src`.

      ii. Find all variable aliases of `src` and get back a list `src_aliases`.

      iii. Lookup `src` in `coalesced_intos` and get back a (possibly empty) set of arrays `src0s` previously coalesced into $m_{\texttt{src}}$.

      iv. Check that all safety conditions are satisfied w.r.t. `dst` and all `src`-like variables in `src` and `src0s`.

      v. If all checks succeed:

   (b) Add `src` to `coalesced_intos`(`dst`).

   (c) We want to record a new coalescing. Let i_{new} be the new index function for `src`, which in the transformed program will now use $m_{\texttt{dst}}$. We define i_{new} like this:

   - If e is a **`copy`** expression, $i_{new} = i_{\texttt{dst}}$.
   - If e is a **`concat`** expression, $i_{new} = i_{\texttt{dst}} + \texttt{offset}$, where `offset` is the total size of the arrays in the **`concat`** arguments before `src`.
   - If e is an in-place update with $DimFix$ indices $\overline{d}$, then $i_{new} = \texttt{slice}\ \ i_{\texttt{dst}}\ \ \overline{d}$.

   Set `result`(`src`) to $(m_{\texttt{dst}}, i_{new})$. For each array variable v in `src0s` and `src_aliases`, get its current index function i_v either through `result`(v) or by looking at the v binding directly, and set `result`(v) to $(m_{\texttt{dst}}, \texttt{rebase}\ \ i_{\texttt{dst}}\ \ i_v)$.

4. Return `result`.

Figure 5.16: The algorithm for finding memory block coalescings.

We follow up this algorithm with a transformation that traverses the program and updates every array `a` with its new memory block and index function taken from `result`(`a`), whenever the latter is defined.

This algorithm does **not** cover all edge cases handled in the implementation. We present several of these throughout the remainder of this chapter and informally extend the algorithm.

Informal Algorithm: Memory Block Reuse

We now define memory block reuse.

An array x with memory block m_{x} can be set to reuse a different memory block m_{y} if

1. m_{x} and m_{y} do not interfere;
2. m_{x} does not interfere with any memory blocks already merged into m_{y}; and
3. $\texttt{bytesize}(m_{\mathsf{x}}) = \texttt{bytesize}(m_{\mathsf{y}})$.

The first two safety conditions should be almost self-explanatory: If two arrays have interfering memory blocks, no memory block merging can occur. The third safety condition ensures that there is enough space for x in m_{y}. We define this condition very conservatively here, but will later loosen it up.

We first show a small example in figure 5.17 of a program where interference limits memory block reuse. At first a is created, then b is created, and only after that do we have the last use of a. Since they interfere, we cannot set b to use m_{a} – otherwise the a_end statement would read from the b array in the a[0] expression!

```
1 let main [n] (xs: [n]i32): i32 =
2   let a = map (+ 1) xs
3   let b = map (+ 10) xs
4   let a_end = a[0]
5   let b_end = b[0]
6   in a_end + b_end
```

Figure 5.17: A program where the two memory blocks interfere.

We then present our basic algorithm for finding reuses in figure 5.18. This algorithm is inspired by the linear scan register allocation algorithm.

1. Let `uses` be a mapping $\mathcal{V}_m \overset{\text{fin}}{\to} \mathcal{V}_{[m]}$, i.e. from memory block name to memory block names. Initialize it to be empty. The map denotes which memory blocks have been merged into which memory blocks.

2. Let `result` be a mapping $\mathcal{V}_v \overset{\text{fin}}{\to} \mathcal{V}_m \times \mathcal{V}_i$, i.e. from statement variable name to memory block and index function. Initialize it to be empty. This is the same as in memory block coalescing.

3. Traverse every statement in the program. For each creation of array a:

   (a) Look through every mapping `mem_used` $\mapsto$ `already_merged` in `uses`. For each mapping, check if m_a can be set to use the memory of `mem_used` w.r.t. the safety conditions.

   - If one or more of such mappings are found, pick one of them. Update `uses`(m_a) to `uses`$(m_a) \cup$ `mem_used`, and set `result`(a) to `mem_uses`.
   - Else, set `uses`(a) to m_a

4. Return `result`.

Figure 5.18: The algorithm for finding memory block reuses.

Code Generator Optimisations

After optimisations in the EXPLICITMEMORY representation, a Futhark program is converted into another intermediate representation, IMPCODE. This representation is more low-level and will itself be converted into runnable C or OpenCL code.

The memory coalescing transformation can find out and record that in expressions such as **`let`** $y =$ **`copy`** x, then x and y can use the same memory block. This is useful for two reasons:

1. We get rid of one allocation. This saves memory usage, but should not cause a great speedup, since memory allocation is mostly about record keeping.

2. We get rid of memory copying. Since x and y inhabit the same memory, there is no need to copy it.

In the EXPLICITMEMORY representation, **`copy`** and **`concat`** are still represented by special constructs. It is only in the IMPCODE representation that they are converted into low-level memory copyings. To satisfy the latter reason, we have added a check to make sure that these low-level memory copyings are not inserted if the memory blocks and index functions of the source and destination are identical.

This was already present for **`copy`**, but not for **`concat`**.

Coalescing Extension: If-then-else

The basic coalescing algorithm described in figure 5.16 does not do anything extraordinary to handle if-then-else expressions. However, this can result in invalid transformations. See figure 5.19 for an example of where the current algorithm goes wrong.

```
1 let main [n] (cond: bool, x: *[n][n]i32, a: [n]i32): (*[n][n]i32, i32) =
2   let b = map (+ 1) a
3   let (y, r) =
4     if cond
5     then let y0 = map (+ 1) a
6          in (y0, b[0])
7     else (b, 0)
8   let x[0] = y
9   in (x, r)
```

Figure 5.19: A program where an if-then-else restrics a coalescing.

In this program the compiler will try to coalesce `y` into $m_{\texttt{x}}$ on line 8. Since `y` is the first part of the return value of an **`if`** expression, the compiler will then try to coalesce the first parts of the return values of the **`if`** *branches* into $m_{\texttt{x}}$ as well – `y0` and `b` – since they are aliased by y. The problem with this is that `b` is used *after* the creation of `y0` in the **`then`** branch, so if they are set to use the same memory block, the `b[0]` expression on line 6 will actually read `y0[0]`, which is wrong. In conclusion, this program needs to fail at any coalescing.

We look at a few more programs before we decide on an extra safety condition for **`if`** expressions. First consider the program in figure 5.20, where both branches return arrays that are defined *outside* the **`if`** expression. Here, $m_{\texttt{ys}}$ will be existential, and `ys` will alias the branch result arrays `ys0` and `ys1`, so the algorithm will try to coalesce both arrays into $m_{\texttt{x}}$. Since both arrays are created outside the **`if`**, they will at some point *exist at the same time*: after a coalescing, `ys0` and `ys1` will both use $m_{\texttt{xs}}$, but this means that the `ys1` definition on line 3 will override the contents of `ys0`, so we can never allow this case.

```
1 let main [n] (xs: *[n][n]i32, cond: bool, i: i32): [n][n]i32 =
2   let ys0 = [0..<n]
3   let ys1 = map (+ 1) [0..<n]
4   let ys = if cond
5            then ys0
6            else ys1
7   let xs[i] = ys
8   in xs
```

Figure 5.20: A program where both branches in the `if` expression return arrays that are defined outside itself.

We then look the program in figure 5.21. Here, both branches return arrays that are defined *inside* the **if** expression. Contrary to the program in figure 5.20, these arrays do not exist at the same time: `ys0` only ever exists in the **then** branch, and `ys1` only ever exists in the **else** branch. By this we reason that it is okay for them to use the same memory block, so we go ahead with the coalescing.

Also, since `xs` is a unique paramater (signified with the $*$ character), i.e. allocated by the caller, we actually end up with zero allocations after the coalescing transformation.

```
let main [n] (xs: *[n][n]i32, cond: bool, i: i32): [n][n]i32 =
  let ys = if cond
           then let ys0 = [0..<n]
                in ys0
           else let ys1 = map (+ 1) [0..<n]
                in ys1
  let xs[i] = ys
  in xs
```

Figure 5.21: A program where both branches in the `if` expression return arrays that are defined inside itself.

Next up we take a look at a hybrid **if** expression where one array is defined outside the expression, and one array is defined inside; see figure 5.22. Here, `ys0` exists when `ys1` is created, while `ys1` does not exist when `ys0` is created. To have a safe coalescing in this case, we reason that we must ensure that `ys0` is not actually *used* in the `ys1` branch; we want to make sure that they do not depend on each other at all. We can see that this is the case, so in this case we would go ahead with the coalescing.

Note that the initial program in figure 5.19 also exhibits the one-array-created-outside, one-array-created-inside structure, but in that case the return value of the **else** branch is used in the **then** branch, in which case coalescing will not work.

```
let main [n] (xs: *[n][n]i32, cond: bool, i: i32): [n][n]i32 =
  let ys0 = [0..<n]
  let ys = if cond
           then ys0
           else let ys1 = map (+ 1) [0..<n]
                in ys1
  let xs[i] = ys
  in xs
```

Figure 5.22: A program where the `if` expression has one branch return an array defined outside the expression, and one return an array defined inside.

We have looked at three overall cases now:

- **`if`** expressions where both branches return arrays that are defined inside themselves.
- **`if`** expressions where both branches return arrays that are defined outside the expression.
- **`if`** expressions where one branch returns an array defined outside the expression, and one branch returns an array defined inside the expression.

To handle these cases correctly, we extend the coalescing algorithm in figure 5.16 with an extra safety condition to check when trying to coalesce `src` into $m_{\texttt{dst}}$:

Extra safety condition for `if`:

If `src` is an **`if`** expression, or aliases an **`if`** expression, we check that not just is $m_{\texttt{dst}}$ not used after the creation of `src` and before the creation of `dst` (safety condition 3), but neither is any memory block that is aliased by $m_{\texttt{dst}}$. Also check that at least one branch returns an array that was created inside that branch.

Reuse Extension: Max Trick

As we pointed out in section 4, the third safety condition is very conservative; requiring that two arrays have the exact same size can be too limiting, so we would like loosen up the size safety condition. We show a small program in figure 5.23 that would benefit from this.

```
1  let main [n] (xs0: [n]i32, i: i32): []i32 =
2    let m_xs = alloc n · bytesize(i32)
3    -- Memory: m_xs
4    let xs = map (+ 1) xs0
5    let k = xs[i]
6    let n1 = n - 1
7    let m_ys = alloc n1 · bytesize(i32)
8    -- Memory: m_ys
9    let ys = replicate n1 k
10   in ys
```

Figure 5.23: A program where `ys` could feasibly be set to reuse the memory of `xs`, but where an overly conservative approach would refuse to do so.

We want to set `ys` to use $m_{\texttt{xs}}$, and thus merge $m_{\texttt{ys}}$ into $m_{\texttt{xs}}$. We can see that $m_{\texttt{ys}}$ is 4 bytes smaller than $m_{\texttt{xs}}$ ($(n-1)$ 32-bit elements compared to n 32-bit elements), so we should be able to perform this merge, *assuming* we can make the compiler run a size analysis that detects this.

Or maybe there is another way? Consider the program in 5.24. The only difference is that `ys` is now size $n+1$ instead of $n-1$. Now there is no space for it in $m_{\texttt{xs}}$! What then? Should we try to merge $m_{\texttt{xs}}$ into $m_{\texttt{ys}}$ instead? But then we lose the nice-to-reason-about top-down linear-pass approach of the algorithm.

```
let main [n] (xs0: [n]i32, i: i32): []i32 =
  let m_xs = alloc n · bytesize(i32)
  -- Memory: m_xs
  let xs = map (+ 1) xs0
  let k = xs[i]
  let n1 = n + 1
  let m_ys = alloc n1 · bytesize(i32)
  -- Memory: m_ys
  let ys = replicate n1 k
  in ys
```

Figure 5.24: The same program parts, but with the larger array at the end instead.

Instead we introduce the *max trick* whereby we maximise the allocation size of the to-be-merged-into memory block to accomodate both the original array and the new array. We show the desired result of applying this to the program of figure 5.24 in figure 5.25. After the transformation, $m_{\texttt{xs}}$ has space for `maxed` (from a new statement) 32-bit integers instead of `n`, allowing `ys` to reuse it. This approach also works for the first program in figure 5.23, in which case it just adds a superfluous `max n (n - 1)` expression.

```
let main [n] (xs0: [n]i32, i: i32): []i32 =
  let n1 = n + 1
  let maxed = max n n1
  let m_xs = alloc maxed · bytesize(i32)
  -- Memory: m_xs
  let xs = map (+ 1) xs0
  let k = xs[i]
  -- Memory: m_xs
  let ys = replicate n1 k
  in ys
```

Figure 5.25: The program with maxed allocation size in $m_{\texttt{xs}}$.

It might seem a bit roundabout to insert these `max` statements even in cases where a size analysis could deem memory block merging safe without them. The benefit is that it is a simple transformation and easy to reason about. This is particularly important in the actual code handled by the compiler, since this is often much more confusing than the examples we present in this report – at the point of the memory block optimisations, a program has already been through many transformations. Also, an added `max` calculation should not impact speed noticably (and might even be simplified away in a separate compiler pass).

Furthermore, the max trick allows us to merge memory blocks of sizes that cannot be statically determined. The approach can be generally applied whenever the size computation can be hoisted up to before the allocation of both memory blocks.

We extend the reuse algorithm in figure 5.18 for reusing a memory block $m_{\mathtt{y}}$ in an array `x` with memory block $m_{\mathtt{x}}$:

- Apart from returning a mapping `result`, also maintain and return a mapping `maxes` of $\mathcal{V}_m \xrightarrow{\text{fin}} \mathcal{V}_{[v]}$, i.e. from memory block name to the variable names that needs to be maxed and used at the new allocation size for the memory block. Initialize it to be empty.
- We change the third condition from
  - $\mathtt{bytesize}(m_{\mathtt{x}}) = \mathtt{bytesize}(m_{\mathtt{y}})$.

  to
  - $\mathtt{bytesize}(m_{\mathtt{x}}) = \mathtt{bytesize}(m_{\mathtt{y}})$; *or*
  - $\mathtt{bytesize}(m_{\mathtt{x}}) \neq \mathtt{bytesize}(m_{\mathtt{y}})$, and the allocation size of $m_{\mathtt{x}}$ is in scope at the allocation of $m_{\mathtt{y}}$, and the computation of the size of $m_{\mathtt{y}}$ can be hoisted up to before the allocation of $m_{\mathtt{x}}$. In this case we add the allocation size variables of $m_{\mathtt{x}}$ and $m_{\mathtt{y}}$ to $\mathtt{maxes}(m_{\mathtt{y}})$.

Reuse Extension: Unique Parameter Reuse

Recall that a parameter in Futhark function definitions can be *unique*, signified with the $*$ character, meaning that the caller is guaranteed to never access the array again. For our purposes this just means that we can reuse its memory.

A program as simple as

```
1  -- Memory of xs: m_xs
2  let main (xs: *[]f32): []f32 =
3    let m_ys = alloc ...
4    -- Memory: m_ys
5    let ys = map (+ 1) xs
6    in ys
```

benefits from this and results in `ys` being set to use $m_{\mathtt{xs}}$, and a program without any allocations.

We informally extend the reuse algorithm to, at the beginning of its top-down pass, also record the memory blocks of unique array parameters as memory blocks with reuse potential.

Assertion Tracking

For this extension we may have to track generated `assert` statements in a Futhark program. If a function definition in the source language has multiple arguments of the same size, internally they will have different size variables (for reasons that are out of scope of this thesis), followed by asserts in the code that check that they are equal.

For example, we can have the program in the source language

```
let main [n] (x: *[n]i32, y: [n]i32): [n]i32 =
  let k = x[0]
  let z = map (+ k) y
  in z
```

which will be transformed into the program (ignoring many low-level details)

```
let main ([size_1]i32 x) ([size_2]i32 y) =
  let assert_arg = size_1 == size_2
  let cert = assert(assert_arg, ...)
  ...
```

where `assert` is an internal construct for runtime guarantees. We need to record these equality assertions to handle e.g. this case: We reason that it should be possible for `z` to reuse the unique input memory of `x`, but we can only do this when the compiler knows that the size of `x` (`n` in the source language) is the same as the size of `y` (also `n` in the source language).

We extend the reuse algorithm to also keep track of a set of equality assertions as it discovers them through its top-down pass. For every size comparison, consider two size variables to be equal if they are in the same assertion.

Reuse Extension: Data Race Interferences

This extension only applies to the very specific program pattern where there is a kernel in which

- each thread returns an array (instead of a scalar, which is what typically happens), and
- the kernel body creates multiple arrays, so that one of them can maybe have its memory reused.

One program that requires this extension is the the Option Pricing benchmark program, which we look at in chapter 10.

When an array is created inside a kernel body, it is always an *indexed* creation: As shown in the grammar in figure 3.1 in chapter 3, only bodies of kind b_{out} can have allocation statements – kernels, which have the b_{in} body kind, cannot contain allocation statements. (This design choice is driven by the limitations of the GPU model into which code can be generated.)

To handle this, a previous compiler pass hoists all allocations out of kernels. Instead of having each thread allocate memory for its own arrays, the compiler places statements outside the kernel that allocates memory for *all* threads for all local arrays. This results in kernel body arrays with new index functions. Instead of each array having an index function $Direct(\overline{size})$, it gets an index function $Direct(\overline{size'})\ldots$ `->`$Slice(0, \texttt{group\_id})$`->`$Slice(0, \texttt{local\_tid})$ instead, where

- `group_id` and `local_tid` are thread-local specifiers;
- $size'$ includes the kernel size variables `num_groups` and `group_size` (the number of threads is `num_groups` $\cdot$ `group_size`; and

- The '...' part may refer to other index function constructs, e.g. a $Permute$.

We want to detect the cases where two indexed creations in a kernel body can potentially overlap on a thread-level if merged.

As an example, think of a program that ends up with these two array creations inside the same kernel:

```
result_first: [chunk][m]f32@mem_first->
    Direct(num_groups, chunk, m, group_size)->Permute([0, 3, 1, 2])
    ->Slice(0, group_id)->Slice(0, local_tid)->Slice(0, chunk)->Slice(0, m)

result_second: [o][m]f32@mem_second->
    Direct(num_groups, o, m, group_size)->Permute([0, 3, 1, 2])
    ->Slice(0, group_id)->Slice(0, local_tid)->Slice(0, o)->Slice(0, m)
```

Both create an array, but neither create the *entire* array: They each have an index in their index functions, and the index functions are not structurally equal: `result_first` indexes into `chunk`, and `result_second` indexes into `o`.

Assume that the last use of `result_first` is before `result_second`, so naively they do not interfere. However, since these creations occur inside kernels, and since they write to different parts of their thread-local arrays, they do actually interfere. If `result_second` were writing to the same index range as `result_first`, it would be okay to merge their two memory blocks, but in this case we might end up with a program execution like this if we merge them:

1. Value a is written to `result_first` in thread t.
2. Value b is written to `result_second` in thread u.
3. Value b (and not a, which is the expected behaviour) is read from `result_first` in thread t.

We say that this is a data race interference. This happens because writing to `result_second` in thread u can, as an unwanted side-effect, also write to `result_first` in thread t. On the GPU this happens because not all threads are guaranteed to run in lockstep.

We solve this by by finding every combination of two different index arrays x and y in a kernel body, and record that their memory blocks m_x and m_y interfere if

- their index functions are not structurally equal, *or*
- the byte sizes of the base types of x and y are different – for example, even if x and y have the same index functions, they will not refer to the same memory indices if x is a 32-bit array and y is a 64-bit array.

Reuse Extension: Max Trick Inside Kernels

This extension applies to the same pattern as in the previous section, and has the potential to ignore some of the interferences, thereby permitting memory block reuse.

Let a kernel body have two indexed array creations `result_0` with size `s0` and `result_1` with size `s1`, and with the index functions

`result_0` : $ixfun\_start\_0$->$indices\_start\_0$->$Slice(0, \texttt{s0})$

`result_1` : $ixfun\_start\_1$->$indices\_start\_1$->$Slice(0, \texttt{s1})$

where

- $ixfun\_start\_0$ and $ixfun\_start\_1$ are index functions in their own right; and
- $indices\_start\_0$ and $indices\_start\_1$ are nonempty slices.

We put forward the additional requirements that

- $ixfun\_start\_0$ is structurally equal to $ixfun\_start\_1$ except for mentions of the sizes `s0` and `s1`.
- $indices\_start\_0$ is structurally equal to $indices\_start\_1$.

Here is an example taken from the Option Pricing benchmark:

`result_0` : $Direct(\texttt{num\_groups}, \texttt{s0}, \texttt{group\_size})$->$Permute([0, 2, 1])$
->$Slice(0, \texttt{group\_id})$->$Slice(0, \texttt{local\_tid})$->$Slice(0, \texttt{s0})$

`result_1` : $Direct(\texttt{num\_groups}, \texttt{s1}, \texttt{group\_size})$->$Permute([0, 2, 1])$
->$Slice(0, \texttt{group\_id})$->$Slice(0, \texttt{local\_tid})$->$Slice(0, \texttt{s1})$

By default `result_0` and `result_1` will be set to interfere because of the data race analysis. We can fix this my making both index functions describe the same access pattern except for the final dimension, which will be retained, so that the arrays inside the kernel body do not change shape.

For the example above, these are the index functions that we want to get:

`result_0` : $Direct(\texttt{num\_groups}, \texttt{s\_max}, \texttt{group\_size})$->$Permute([0, 2, 1])$
->$Slice(0, \texttt{group\_id})$->$Slice(0, \texttt{local\_tid})$->$Slice(0, \texttt{s0})$

`result_1` : $Direct(\texttt{num\_groups}, \texttt{s\_max}, \texttt{group\_size})$->$Permute([0, 2, 1])$
->$Slice(0, \texttt{group\_id})$->$Slice(0, \texttt{local\_tid})$->$Slice(0, \texttt{s1})$

where we add the statement `let s_max = max s0 s1`. This makes them cover the same area in space, so that there are no data race interferences. The final index slices `s0` and `s1` are kept as they were to keep the array shapes as they were. This change means that the smallest array will not be writing to all of its available space: There will be some bytes in each thread that go unused, but this is outweighed by the fact that it reuses already used memory.

We need to check:

- Is `s1` in scope at the allocation?

- Does `result_0` and `result_1` have the same base type size?

If true, modify the program as such:

- Insert a `s_max` statement before the allocation.
- Change the allocation size to use `s_max` instead of `s0`.
- Modify both index functions to use `s_max` instead of `s0` and `s1`, respectively, except for at the final index slice.

Extension: If an array reuses a previously reused array, remember to update *all* index functions. Currently we avoid these cases for simplicity of implementation.

Handling Choice

In this section we describe how the two core transformations handle *choices*, and what the alternatives may be. In this context we define "choice" to mean that the compiler can choose to perform one memory block merging or another, but not both.

Coalescing

Consider the program in figure 5.26.

- On line 3 we create an array `t0`.
- On line 6 we create an array `annoying` whose memory block allocation depends on the *value* of `t0` (through the `t0[0]` expression), not the *shape* of `t0`. This makes it impossible to hoist the allocation up to before the `t0` creation.
- On line 8 we have a potential coalescing of `t0` into $m_{\texttt{t1}}$.
- On line 10 we have a potential coalescing of `t1` into $m_{\texttt{t2}}$. (We also have a potential coalescing of `coalescing` into the same memory block, but we ignore that, as it is always possible independently of which choices the algorithm makes.)

There are two potential coalescings, but we cannot do both!

- Say we first coalesce `t0` into $m_{\texttt{t1}}$. Then, if we want to coalesce `t1` into $m_{\texttt{t2}}$, that also means coalescing `t0` into the same memory block. However, the allocation of $m_{\texttt{t2}}$ needs to be hoisted up to before the creation of `t0` for that to work, which is not possible due to its allocation size depending on k (the size of `annoying`), which itself cannot be hoisted up to before `t0`.
- Say we instead first coalesce `t1` into $m_{\texttt{t2}}$. Then, if we want to coalesce `t0` into $m_{\texttt{t1}}$, we notice that $m_{\texttt{t1}}$ has been merged with $m_{\texttt{t2}}$, and so we run into the same problem of not being able to hoist the allocation of $m_{\texttt{t2}}$ up to before the creation of `t0`.

```
1  let main [n] (ns: [n]i32): []i32 =
2    -- Memory of t0: m_t0; element size: n
3    let t0 = map (+ 1) ns
4    let k = t0[0]
5    -- Memory of annoying: m_annoying; element size: k
6    let annoying = [0..<k]
7    -- Memory of t1: m_t1; element size: n
8    let t1 = copy t0
9    -- Memory of t2: m_t2; element size: n + k
10   let t2 = concat t1 annoying
11   in t2
```

Figure 5.26: A choice between two memory block coalescings.

For program patterns like this, where an array size depends on a value from another array, our implementation unintelligently chooses the first coalescing as a side-effect of being a top-down pass over the program; a bottom-up pass might do the opposite. A clever approach would have a heuristic for determining which coalescing reduces the most memory usage and copying, maybe based on a kind of size analysis, and pick that one.

Reuse

Consider the program in figure 5.27.

- On line 3 and line 5 we create two arrays `xs` and `ys`
- On line 6 we lastly use both `xs` and `ys`.
- On line 8 we create an array `zs`. This array can reuse either m_{xs} or m_{ys}.

```
1  let main [n] (xs0: [n]i32, ys0: [n]i32, i: i32): [n]i32 =
2    -- Memory of xs: m_xs
3    let xs = map (* 2) xs0
4    -- Memory of ys: m_ys
5    let ys = map (* 3) ys0
6    let k = xs[i] + ys[i]
7    -- Memory of zs: m_zs
8    let zs = replicate n k
9    in zs
```

Figure 5.27: A choice between two memory block reuses.

For program patterns like this, where an array can choose between reusing the memory of different arrays, our implementation just picks the first one; we have no grand strategy.

Chapter 6

Enabling Analyses

In this chapter we describe in detail what we call *enabling program analyses*: program analyses whose results are necessary for performing our program transformations.

Memory Block Aliases

The Futhark compiler already supports finding variable aliases: A variable aliases another variable if they use the same memory block.

Since our transformations are very memory-centric, we define a new kind of aliasing on memory blocks: We write that a memory block aliases another memory block if they refer to the same memory.

This is only necessary because an expression can use an existential memory block in the case of loops and ifs.

To get the full picture of the memory usage in a function body, we also take the transitive closure of the memory aliases: If memory block `x` aliases block `y`, and `y` aliases block `z`, then `x` should also alias `z`.

Note that memory block aliasing is not commutative. For example, a loop memory block can in different iterations alias two different other memory blocks. In effect it aliases both, since it cannot statically know which iteration it is in, but the aliased blocks do not alias each other.

Liveness Analysis

Before we can merge two memory blocks, we need to know several liveness-related attributes:

- The first uses of memory;
- The last uses of memory; and

- The interferences of memory.

We simplify our first and last use analyses slightly by describing them in terms of array variables instead of memory block names. This avoids an extra layer, and is okay to do, since at the end, the analyses can be rethought of in terms of memory blocks:

- either no memory block merging optimisation has been run prior to these analyses, in which case we have a one-to-one mapping between memory block names and array variables (ignoring delayed arrays); or
- there has been a previous memory block merging optimisation, in which case a memory block can be associated with different arrays – but here we can still reason in terms of *multiple* liveness intervals based on the arrays that the memory blocks maps to.

Once converted to memory block space, these analyses then use the results from the memory block aliasing analysis to find the full extents. For example, a last use of an existential memory block m is extended to also include the last uses of the memory block aliases of m.

With this out of the way, the first use analysis is then simply stated: An array a is firstly used in a statement **`let`** $a = e$ if e creates an ordinary array (e.g. with **`copy`**).

Last use analysis and interference analysis is a bit more complicated.

Informal Algorithm: Last Use Analysis

An array a is lastly used in a statement **`let`** $\overline{v} = e$ if a is part of e, and if neither a nor any of its aliases are used on any execution path following the statement.

In chapter 4 we wrote that the result of the analysis is a mapping $\mathcal{V}_v \xrightarrow{\text{fin}} \mathcal{V}_{[m]}$, i.e. from variable name denoting a let binding (statement) to a set of the names of the memory blocks that are lastly used at the binding of v.

However, this is too limiting, as exemplified in figure 6.1. Two arrays `dst0` and `dst1` are created. In this limited analysis, two last uses are found:

- `dst0` is lastly used in the creation of `k`; and
- `src` is lastly used in the creation of `dst1`.

The memory block reuse optimisation will notice this and let `dst1` reuse the memory of `dst0`, completely oblivious to the fact that the actual last use of `dst0` (and `dst1`) is in the *result* of the body, in the process creating an invalid program.

```
1 let main (src: []i32): ([]i32, []i32) =
2   let dst0 = map (+ 1) src
3   let k = dst0[0]
4   let dst1 = map (* k) src
5   in (dst0, dst1)
```

Figure 6.1: A Futhark program with an important last use in the body result.

To handle these cases, we extend the analysis to instead return a mapping $\mathcal{V}_{StmOrRes} \xrightarrow{\text{fin}} \mathcal{V}_{[m]}$, where

$$\begin{array}{rcll} StmOrRes & ::= & FromStm & v \\ & | & FromRes & v. \end{array}$$

We show the basic algorithm for finding last uses in figure 6.2. Note that step 3.(a) overwrites any previous mappings of a in `optimistics`.

1. Let `optimistics` be a mapping $\mathcal{V}_a \xrightarrow{\text{fin}} \mathcal{V}_v$, i.e. from array variable to statement variable. Initialize it to be empty.
2. Let `result` be a mapping $\mathcal{V}_v \xrightarrow{\text{fin}} \mathcal{V}_{[a]}$, i.e. from statement variable to a set of array variables. Initialize it to be empty.
3. Traverse top-down every statement in the program. For each statement **`let`** $v = e$:
   (a) For each array variable a in e:
      i. Add the mapping $a \mapsto v$ to `optimistics`.
4. For each mapping $a \mapsto v$ in `optimistics`:
   (a) Add a to `result`(v).
5. Return `result`.

Figure 6.2: The algorithm for discovering last uses.

Loop Complications

The algorithm above is complicated by the nested bodies in loops and kernels. See figure 6.3 for an example. Here, the array `t` is seemingly lastly used on line 4 at the binding of `u`. Having its last use there allows the memory block reuse optimisation to let the `v` array created on the next line reuse the memory of `u`.

However, since this is a loop, and loops can have multiple iterations, doing that would make the next iteration read wrong data from `t`, since `t` would refer to the same memory as `v`. We solve this by stating that the last use of `t` occurs in `res`, i.e. in the entire loop body.

```
let main (k: i32): []i32 =
  let t = [0..<k]
  let res = loop acc = t for _i < k do
    let u = map (+) t acc
    let v = map (* 2) u
    in v
  in res
```

Figure 6.3: A Futhark program with a last use seemingly inside a loop body. (This is a simplified program. In actuality, a previous pass will have inlined some of the code, thus avoiding the case under discussion.)

We need a general way of handling last uses when there is a risk of cycles. We extend the algorithm in figure 6.2 to keep track of whether a variable has its first use inside or outside the body currently being traversed. Only if the variable was created inside the body do we add it to `optimistics`; otherwise its last use becomes part of the entire body statement.

Loop Extension: Multiple Liveness Intervals

Consider the loop-heavy program in figure 6.4.

```
1  let main [n] (ns: [n]i32): [n]i32 =
2    let xs0 = copy ns
3    let loop_result = loop xs = xs0 for i < n do
4      let ys = map (+ 1) xs
5      let k = ys[i]
6      let ts0 = map (+ k) ns
7      let k0 = ys[n - i - 1] + ts0[i]
8      let ts1 = map (+ k0) ns
9      in ts1
10   in loop_result
```

Figure 6.4: A Futhark program

- On line 2, create an array `xs0` whose memory we can reuse.
- On line 4, we have the last use of `xs`.
- On line 6, create another array `ts0`. The compiler will try to reuse memory for this array. It has access to the non-existential memory blocks $m_{\texttt{xs0}}$ and $m_{\texttt{ys}}$. $m_{\texttt{ys}}$ is lastly used later, so that cannot be reused; $m_{\texttt{xs0}}$ is aliased by the existential $m_{\texttt{xs}}$, so conservatively it cannot be used either, since $m_{\texttt{xs}}$ and its aliases can be said to interfere with the entire loop.

  However, since `ts1` on line 8 creates a new array that does not read from $m_{\texttt{xs}}$, and since `ts1` is returned in some iteration, we know that the existential $m_{\texttt{xs}}$ will refer to $m_{\texttt{ts1}}$ in the

next iteration, and we can instead argue that $m_{\texttt{xs}}$ has a "temporary" last use in the creation of `ys` on line 4, and a new first use in the creation of `ts1`. This allows us to reuse the aliased non-existential $m_{\texttt{xs0}}$ inbetween `ys` and `ts1` – assuming $m_{\texttt{xs0}}$ does not have its own direct last use later on, overriding any inherited last use from the existential $m_{\texttt{xs}}$.

Since `ts0` is created between the creations of `ys` and `ts1`, we can set it to reuse $m_{\texttt{xs0}}$, and simplify $m_{\texttt{ts0}}$ away.

To handle this in the general case, we informally extend the last use analysis to record an array variable a as lastly used at a statement if

- some existential array variable b that aliases a has its last use at that statement; and
- a itself does not have a last use later on in the loop body

Informal Algorithm: Interference Analysis

We build upon the memory block variants of our first and last analyses and show the basic algorithm for finding interferences in figure 6.5.

1. Let `live` be $\mathcal{V}_m$, i.e. a set of currently live memory blocks. Initialize it to be empty.
2. Let `result` be a mapping $\mathcal{V}_m \xrightarrow{\text{fin}} \mathcal{V}_{[m]}$, i.e. from memory block to a set of memory blocks. Initialize it to be empty.
3. Traverse every statement in the program in a top-down fashion. For each statement **`let`** $v = e$:
   (a) For each first use of a memory block m in $\overline{v}$ (i.e. array creation point):
      i. Extend `result` with all combinations $m \to m_l$ and $m_l \to m$, $\forall m_l \in$ `live`.
      ii. Add m to `live`.
   (b) For each last use of a memory block m in $\overline{v}$:
      i. Remove m from `live`.
4. Return `result`.

Figure 6.5: The algorithm for finding memory block interferences.

Interference Exceptions

The basic algorithm is simple and works, but also captures interferences that are not *really* interferences: interferences on memory blocks where we can use index access analysis to determine that there can be memory reuse without changing the meaning of the program; see figure 6.6 for a small example. Here, an array `ys` is created by a `map` over `xs`. By the interference analysis, since `xs` does not have its last use until the creation of `ys`, their two memory blocks interfere.

```
let main (ns: []i32): []i32 =
  let xs = map (+ 1) ns
  let k0 = xs[0]
  let ys = map (+ k0) xs
  in ys
```

Figure 6.6: A Futhark program where `xs` and `ys` would appear to interfere, but where index access analysis shows that they do not. We use `k0` to avoid fusion of the two arrays.

If we view this program in its EXPLICITMEMORY form, we have the two kernels in figure 6.7 (though we only spell out the second one). The nested body of `ys` is run for every thread, for as many elements as there are in `xs`. The `gtid` variable is a scalar that refers to the current thread index, and a thread writes its result to the output array at this index – in this case `ys[gtid]`.

We notice that each thread at index `gtid` reads from just `xs[gtid]` and writes to just `ys[gtid]`, in that order. By this we reason that there is no problem with `xs` and `ys` using the same memory.

```
let main (ns: []i32): []i32 =
  let xs = kernel map ...
  let k0 = xs[0]
  let ys = kernel map (gtid < size_xs)
    let binop = xs[gtid]
    let res = binop + k0
    return res
  in ys
```

Figure 6.7: The program in a pseudo-EXPLICITMEMORY form.

We handle this for both loops and kernels by extending the algorithm in figure 6.5 to, for every body, find all interference exceptions and remove those from the ordinarily calculated interferences. We find the interference exceptions for a body this way:

1. Find all pairs of memory blocks (m_{new}, m_{killed}), where m_{new} is the memory block of a firstly used variable, and m_{killed} is the memory block of a lastly used variable in the same body. This is a body-local analysis. In the example above, `xs` has its local last use in the `binop` statement, and `res` (which is written to `ys`) has its first use, so there would be one pair with $(m_{\mathtt{ys}}, m_{\mathtt{xs}})$.
2. Filter the pairs:
   - Do their related arrays have the same index functions?
   - Are they accessed with the same slices, i.e. is m_{new} written to in the same way that m_{killed} is read from?
   - Do the arrays have the same primitive byte sizes? For example, an array of 32-bit integers still interferes with an array of 64-bit integers even if the other constraints are

fulfilled, since otherwise they will have different alignments.

This extra check works for arbitrarily nested bodies.

Chapter 7

Enabling Transformations

In this chapter we describe the program transformations that are not absolutely critical to the core transformations, but which will make the core transformations extract more memory block mergings. We cover *allocation hoisting* and *allocation size hoisting*.

Allocation Hoisting

We use allocation hoisting to enable more memory block mergings in the *memory block coalescing* core transformation.

Consider the program in figure 7.1. We would like to coalesce $m_{\texttt{arr}}$ into $m_{\texttt{res}}$ through the `copy` expression on line 7. However, we cannot currently satisfy safety condition 2: That the destination memory exists at the creation of the source array. If we go ahead with the coalescing, `arr` will be set to use memory that has not been allocated yet!

```
1 let main (n: i32): []i32 =
2   let m_arr = alloc n · bytesize(i32)
3   -- Memory: m_arr
4   let arr = [0..<n]
5   let m_res = alloc n · bytesize(i32)
6   -- Memory: m_res
7   let res = copy arr
8   in res
```

Figure 7.1: A program in need of allocation hoisting.

We solve this by hoisting the allocation of $m_{\texttt{res}}$ up to the beginning of the body; see figure 7.2. Safety condition 2 can now be satisfied, and we can perform the coalescing (not shown).

```
1  let main (n: i32): []i32 =
2    let m_arr = alloc n · bytesize(i32)
3    let m_res = alloc n · bytesize(i32)
4    -- Memory: m_arr
5    let arr = [0..<n]
6    -- Memory: m_res
7    let res = copy arr
8    in res
```

Figure 7.2: The program with memory allocations hoisted to the top.

In general we cannot just hoist an allocation to the top of the body. An allocation depends on a size, and that size can be declared either in a function paramater or in a statement of its own; in both cases, the allocation must occur after the size declaration.

In some cases it is not enough to hoist *just* the allocation: if an array is written to in an in-place update, we will also have to hoist its creation to avoid overwriting the update; see figure 7.3 and figure 7.4 for a non-transformed program and the naively allocation-only transformed program with coalescing, respectively.

Now `t0` writes to part of $m_{\texttt{t1}}$ on line 4 – but right after that, on line 6, `t1` writes to the entire array, overvriting the contents of `t0`. We solve this by also hoisting the creation array `t1`, as show in figure 7.5.

The other two coalescing cases, `copy` and `concat`, do not suffer from this problem, as they write to the entirety of an array, and not just part of it.

```
1  let main [n] (i: i32, ns: [n]i32): [n][n]i32 =
2    let m_t0 = alloc n · bytesize(i32)
3    -- Memory: m_t0
4    let t0 = map (+ 1) ns
5    let m_t1 = alloc n · bytesize(i32)
6    -- Memory: m_t1
7    let t1 = replicate n [0..<n]
8    let t1[i] = t0
9    in t1
```

Figure 7.3: A program where hoisting of $m_{\texttt{t1}}$ also requires hoisting of `t1` itself.

```
1  let main [n] (i: i32, ns: [n]i32): [n][n]i32 =
2    let m_t1 = alloc n · bytesize(i32)
3    -- Memory: m_t1; element offset: i · n
4    let t0 = map (+ 1) ns
5    -- Memory: m_t1
6    let t1 = replicate n [0..<n]
7    let t1[i] = t0
8    in t1
```

Figure 7.4: The program where only $m_{\texttt{t1}}$ has been hoisted, but where we have performed a coalescing on line 8 nevertheless.

```
1  let main [n] (i: i32, ns: [n]i32): [n][n]i32 =
2    let m_t1 = alloc n · bytesize(i32)
3    let t1 = replicate n [0..<n]
4    -- Memory: m_t1; element offset: i · n
5    let t0 = map (+ 1) ns
6    -- Memory: m_t1
7    let t1[i] = t0
8    in t1
```

Figure 7.5: The program where both $m_{\texttt{t1}}$ and `t1` has been hoisted, and where we have performed the coalescing.

We show our algorithm overview in figure 7.6.

1. Traverse every statement in the program. For each statement **`let`** $m =$ **`alloc`** d:
   (a) Filter out any m that is used by neither `copy` nor `concat` – those are the only kinds of memory we care about, since those are the cases handled by coalescing.
   (b) If the array creation that uses m is used in an in-place update (like in the example above), also handle that statement.
   (c) For each statement **`let`** $v = e$:
      i. Find all free variables in e and recursively hoist their statements (unless they are parameters) upwards in the program body as much as possible.
      ii. Move the v statement to the line just after the (potentially new) location of the least hoisted variable of e.

Figure 7.6: The basic hoisting algorithm.

However, we will have to extend this a bit in section 3.

Allocation Size Hoisting

We use allocation size hoisting to enable more memory block mergings in the *memory reuse* core transformation.

First consider the program in figure 7.7. We would like to let `ys` reuse $m_{\texttt{xs}}$. We follow the algorithm laid out in chapter 5 and maximise the size of $m_{\texttt{xs}}$ to include the size of $m_{\texttt{ys}}$, as described in section 5.7, ending up with the program in figure 7.8.

```
1  let main [n] (xs0: [n]i32, i: i32): []i32 =
2    let m_xs = alloc n · bytesize(i32)
3    -- Memory: m_xs
4    let xs = map (+ 1) xs0
5    let k = xs[i]
6    let n1 = n + 1
7    let m_ys = alloc n1 · bytesize(i32)
8    -- Memory: m_ys
9    let ys = replicate n1 k
10   in ys
```

Figure 7.7: A program in need of allocation size hoisting.

```
1  let main [n] (xs0: [n]i32, i: i32): []i32 =
2    let maxed = max n n1
3    let m_xs = alloc maxed · bytesize(i32)
4    -- Memory: m_xs
5    let xs = map (+ 1) xs0
6    let k = xs[i]
7    let n1 = n + 1
8    -- Memory: m_xs
9    let ys = replicate n1 k
10   in ys
```

Figure 7.8: The program after performing the `max` trick, but not doing any allocation size hoisting.

The size of $m_{\texttt{xs}}$ has been extended by using the new scalar `maxed` from line 2, so it should be okay for `ys` to use the memory. However, `maxed` depends on the variable `n1`, which is not defined until later, so this is an invalid transformation, and will not be allowed by the core transformation. To make it doable we need to first perform allocation size hoisting on all allocation sizes – just `n` and `n1`, and only `n1` is declared in a statement. We show the proper program in figure 7.9.

```
let main [n] (xs0: [n]i32, i: i32): []i32 =
  let n1 = n + 1
  let maxed = max n n1
  let m_xs = alloc maxed · bytesize(i32)
  -- Memory: m_xs
  let xs = map (+ 1) xs0
  let k = xs[i]
  -- Memory: m_xs
  let ys = replicate n1 k
  in ys
```

Figure 7.9: The program after performing both allocation size hoisting and the `max` trick.

The body of the allocation size hoisting algorithm is the same as the allocation hoisting algorithm; the difference lies in the first step: Instead of finding all allocation statements, we find all statements that declare scalars *used in* allocation statements.

Allocation hoisting will also end up performing allocation size hoisting, since recursively hoisting dependencies of an allocation statement includes hoisting its size, but having this as a separate enabling transformation allows us to apply memory block reuse without first having to perform memory block coalescing, which is useful for testing.

Hoisting Heuristics

So far we have taken the approach of unapologetically hoisting the interesting parts – allocations or just allocation sizes – and their dependencies up as much as possible, and hoping for the best. This works as expected for most of our written-for-the-occasion tests, but we have had to limit the extent of the hoisting a bit to avoid it restricting coalescings: If we hoist too much, we can end up changing the base structure of a program in a way that is hurtful.

We present a program in figure 7.10 that cannot handle brakes-off allocation hoisting (to keep it short, we show it without memory annotations).

```
1  let main [n] (wsss0: [n][n][n]i32, ns: [n]i32, i: i32, j: i32):
2    ([n][n][n]i32, [n][n][n]i32) =
3    let wsss = map (\wss -> map (\ws -> map (+ 1) ws) wss) wsss0
4    let xs = map (+ 1) ns
5    let k = xs[0]
6    let use_wsss = map (\wss -> map (\ws -> map (+ k) ws) wss) wsss
7    let vss = replicate n (replicate n 2)
8    let vss[j] = xs
9    let wsss[i] = vss
10   in (wsss, use_wsss)
```

Figure 7.10: A program where full-on hoisting is limiting.

We find these key observations about the program:

- Line 3 creates an array `wsss` into which other arrays can be coalesced.
- Line 4 creates an array `xs` that, at some point, will be inserted into `vss`.
- Line 6 uses `wsss`. It is important that this is use is *before* the creation of `vss`.
- Line 7 creates an array `vss`.
- On line 8, we note that `xs` cannot be coalesced into `vss[j]`, since `vss` is used (in the previous statement) after the creation of `xs`, thus violating safety condition 3.
- On line 9, we note that `vss` can be coalesced into `wsss[i]` – even without allocation hoisting, since $m_{\texttt{wsss}}$ will be declared at the very top of the body (just before `wsss`).

So far we know that we can extract a single memory block coalescing from this program – `vss` into $m_{\texttt{wsss}}$. However, we always run allocation hoisting prior to checking this; see the result of this transformation in figure 7.11 (again we leave out allocations and memory block annotations; these are not interesting in this case).

Ignoring allocations, which will all have moved to the top, the only change is that `vss` has moved up to line 4, just after `wsss`. This transformation makes it impossible to perform *any* coalescing:

- We still cannot coalesce `xs` into $m_{\texttt{vss}}$, for the same reasons as before.
- Additionally, we now cannot coalesce `vss` into $m_{\texttt{wsss}}$, since `wsss` is used between the creation of `vss` and its insertion into `wsss` (safety condition 3).

```
let main [n] (wsss0: [n][n][n]i32, ns: [n]i32, i: i32, j: i32):
  ([n][n][n]i32, [n][n][n]i32) =
  let wsss = map (\wss -> map (\ws -> map (+ 1) ws) wss) wsss0
  let vss = replicate n (replicate n 2)
  let xs = map (+ 1) ns
  let k = xs[0]
  let use_wsss = map (\wss -> map (\ws -> map (+ k) ws) wss) wsss
  let vss[j] = xs
  let wsss[i] = vss
  in (wsss, use_wsss)
```

Figure 7.11: The program with allocations and related statements aggressively hoisted.

In this case our enabling transformation has actually *restricted* a core transformation! The core of this problem is that this enabling transformation is overly simple, and the best solution might be to design a better one. Instead we have extended the existing transformation to never hoist across kernels or loops. When a statement is moved upward in a program body, switching places with other statements one by one, it will stop if it tries to switch places with a loop or a kernel.

This is not a good solution, as we have no real arguments for why this is a good heuristic, but it works well on our many written-for-the-purpose tests, among them the program above.

Chapter 8

A First Attempt at Formalising Memory Block Coalescing

This chapter presents an attempt at formalising a subset of the memory block coalescing transformation described in chapter 5.

Refer to chapter 3 for the grammar of the Futhark language in its EXPLICITMEMORY representation.

Environment

We first describe the environment necessary to describe our main transformation rules; see figure 8.1.

Recall from the grammar in chapter 3 that: m denotes a memory block name, d a scalar name, x an array name, and v denotes any variable name. e^{alg} denotes a scalar expression, c denotes a constant, and i denotes an index function.

$$
\begin{array}{ll}
\mathcal{V}_k & \text{Set of } k \\
\mathcal{V}_{[k]} & \text{Set of sets of } k
\end{array}
$$

Static environment parts (computed separately):

$$
\begin{array}{rcl}
\mathcal{B} & \in & \mathcal{V}_v \xrightarrow{\text{fin}} \mathcal{V}_t \times \mathcal{V}_e \\
\mathcal{LU} & \in & \mathcal{V}_v \xrightarrow{\text{fin}} \mathcal{V}_{[m]} \\
\mathcal{BBC} & \in & \mathcal{V}_x \xrightarrow{\text{fin}} \mathcal{V}_{[x]} \\
\mathcal{VA} & \in & \mathcal{V}_v \xrightarrow{\text{fin}} \mathcal{V}_{[v]} \\
\mathcal{S} & \in & \mathcal{V}_d \xrightarrow{\text{fin}} \mathcal{V}_{e^{alg}}
\end{array}
$$

Dynamic environment parts (computed in the top-down pass):

$$
\mathcal{CI} \in \mathcal{V}_a \xrightarrow{\text{fin}} \mathcal{V}_{[(a,e^{alg})]}
$$

Full environment:

$$
\Sigma = (\mathcal{B}, \mathcal{LU}, \mathcal{BBC}, \mathcal{VA}, \mathcal{S}, \mathcal{CI})
$$

Auxiliary functions:

$$
\begin{array}{rcl}
\mathcal{UB} & \in & \mathcal{V}_v \times \mathcal{V}_v \to \mathcal{V}_{[v]} \\
\mathcal{FI} & \in & \mathcal{V}_i \to \mathcal{V}_{[x]} \\
\mathcal{EI} & \in & \mathcal{V}_i \times \mathcal{S} \to \mathcal{V}_i
\end{array}
$$

Figure 8.1: Environment and auxiliary functions for analysis.

Most of these are needed to facilitate the safety condition checks described informally in chapter 5:

- $\mathcal{B}$ is a finite mapping from a variable x to its type and its expression in the binding of x.
- $\mathcal{LU}$ is a finite mapping from a variable to the memory blocks that are lastly used at the binding of the variable.
- $\mathcal{BBC}$ is a finite mapping from a variable to all previously bound variables that are available at the binding of the variable.
- $\mathcal{VA}$ is a finite mapping from each delayed array variable to its source array variable. We define $\mathcal{VA}(v)$ to be $\emptyset$ when v is an ordinary array.
- $\mathcal{S}$ is a finite mapping from a scalar variable to its scalar value expression in the program.
- $\mathcal{CI}$ is a finite mapping from an array variable a to a set of array variables that have all been coalesced into m_a. We use this to keep track of chains of coalescings, and modify it as we traverse the program: If at some point we want to try letting a reuse memory m_b, we need to know which arrays have been coalesced into m_a, since now they should instead be coalesced into m_b along with a. The e^{alg} describes the offset at which a previous merging occured. In a remerge, we need to use both the old offset and any new offset.

- $\mathcal{UB}$ computes which arrays are used (alive) between two statements in which the first statement dominates the second statement.
- $\mathcal{FI}$ computes the set of all free variables in an index function.
- $\mathcal{EI}$ transforms each variable in an index function to more primitive constituents if possible, and repeats until a fixpoint is reached.

Rules

In this section we present inference rules for the memory block coalescing algorithm. The analysis gets quite complex, so we also explain them in detail in words, and provide examples where necessary.

The inference rules use the previously defined environment and auxiliary functions, and has the shape $\boxed{\Sigma \vdash b_{out} \Rightarrow \mathcal{R}}$.

Here, b_{out} is an unoptimized body, and $\mathcal{R}$ is a mapping $\mathcal{V}_a \overset{\text{fin}}{\to} \mathcal{V}_m \times \mathcal{V}_i$ describing the result of the memory block coalescing transformation. $\mathcal{R}$ describes a mapping from an array name to its new memory block and index function. This result can then be used to transform a program as described in chapter 5.

We first declare a macro `mergeable` for checking that `src` can be set to to use the memory of `dst` (possibly with a new index function as well; that is described further down in the text).

$$
\begin{aligned}
&\text{mergeable}(\Sigma, \texttt{src}, \texttt{dst}) = \\
&(\mathcal{B}, \mathcal{LU}, \mathcal{BBC}, \mathcal{VA}, \mathcal{S}, \mathcal{CI}) = \Sigma, \\
&\text{mergeable}_0(\Sigma, \texttt{src}, \texttt{dst}), \\
&\wedge_{\texttt{src}_k \in \text{map}(\#1, \mathcal{CI}(\texttt{src}))} (\text{mergeable}_0(\Sigma, \texttt{src}_k, \texttt{dst}))
\end{aligned}
$$

$$
\begin{aligned}
&\text{mergeable}_0(\Sigma, \texttt{src}, \texttt{dst}) = \\
&(\mathcal{B}, \mathcal{LU}, \mathcal{BBC}, \mathcal{VA}, \mathcal{S}, \mathcal{CI}) = \Sigma, \\
&(\_ \ @m_{\texttt{dst}}\{i_{\texttt{dst}}\}, \quad \_) = \mathcal{B}(\texttt{dst}), \\
&m_{\texttt{src}} \in \mathcal{LU}(\texttt{dst}), \\
&m_{\texttt{dst}} \in \mathcal{BBC}(\texttt{src}), \\
&m_{\texttt{dst}} \notin \mathcal{UB}(\texttt{src}, \texttt{dst}), \\
&\mathcal{VA}(\texttt{src}) = \emptyset, \\
&\mathcal{FI}(\mathcal{EI}(i_{\texttt{dst}})) \subseteq \mathcal{BBC}(m_{\texttt{src}})
\end{aligned}
$$

Here, $\text{map}(\#1, k)$ returns a set of all first elements in the set k of tuples. The final five constraints correspond to the five informally given safety conditions in chapter 5 for coalescing a variable `src` into a memory block $m_{\texttt{dst}}$:

$m_{\textbf{src}} \in \mathcal{LU}(\textbf{dst})$
: The memory of the right-hand side variable is in its last use.

$m_{\textbf{dst}} \in \mathcal{BBC}(\textbf{src})$
: The allocation of $m_{\texttt{dst}}$ occurs before the creation of `src`.

$m_{\texttt{dst}} \notin \mathcal{UB}(\textbf{src}, \textbf{dst})$
: There is no use of the left-hand side memory block $m_{\texttt{dst}}$ after the creation of `src` and before the creation of `dst`.

$\mathcal{VA}(\textbf{src}) = \emptyset$
: `src` is a newly created array.

$\mathcal{FI}(\mathcal{EI}(i_{\textbf{dst}}, \mathcal{S})) \subseteq \mathcal{BBC}(m_{\textbf{src}})$
: The new index function of `src` only uses variables declared prior to the first use of $m_{\texttt{src}}$. $\mathcal{EI}(i_{\texttt{dst}}, \mathcal{S})$ takes the index function $i_{\texttt{dst}}$ and replaces every scalar variable name d with an expression from $\mathcal{S}(d)$ if possible, and continues to do so until a fixpoint is reached, and the index function cannot be further expanded. Note that, in this simple formulation, this has the risk of expanding the index function more than needed by also incorporating variables defined prior to `src`, and thereby introducing redundant recalculations in the index functions.

Note that when we take the union $m = m_1 \cup m_2$ of two finite maps m_1 and m_2, and the two maps have a key k in common, we set $m(k) = m_2(k)$.

We then describe the three main cases: **`copy`**, **`concat`**, and in-place updates:

$$\frac{\begin{array}{l}(\mathcal{B}, \mathcal{LU}, \mathcal{BBC}, \mathcal{VA}, \mathcal{S}, \mathcal{CI}) = \Sigma, \\ (\tau_{\texttt{dst}} @ m_{\texttt{dst}}\{i_{\texttt{dst}}\}, \quad \_) = \mathcal{B}(\texttt{dst}), \\ \Sigma' \vdash b \Rightarrow \mathcal{R}' \\ \text{WHERE} \\ \mathcal{R}' = \mathcal{R} \cup \begin{cases} \mathcal{R}_0 \cup \mathcal{R}_1 & \text{if mergeable}(\Sigma, \texttt{src}, \texttt{dst}) \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{R}_0 = \texttt{src} \mapsto (m_{\texttt{dst}}, i_{\texttt{dst}}) \\ \mathcal{R}_1 = \bigcup_{(\texttt{src}_k, e_k^{alg}) \in \mathcal{CI}(\texttt{src})} (\texttt{src}_k \mapsto (m_{\texttt{dst}}, \texttt{offset}(i_{\texttt{dst}}, e_k^{alg}, \texttt{src}_k)) \\ \mathcal{CI}' = \mathcal{CI} \cup \begin{cases} (\texttt{dst} \mapsto ((\texttt{src}, i_{\texttt{dst}}) \cup \mathcal{CI}(\texttt{src}))) & \text{if mergeable}(\Sigma, \texttt{src}, \texttt{dst}) \\ \emptyset & \text{otherwise} \end{cases} \\ \Sigma' = (\mathcal{B}, \mathcal{LU}, \mathcal{BBC}, \mathcal{VA}, \mathcal{S}, \mathcal{CI}') \end{array}}{\Sigma \vdash \quad \textbf{let}\, \texttt{dst} : \rho = \textbf{copy}\, \texttt{src}\, \textbf{in}\, b \Rightarrow \mathcal{R}} \quad \text{(CoAL-Copy)}$$

Here, we define $\texttt{offset}(i, e^{alg}, \texttt{src})$ to be shorthand for $i\texttt{->}Slice(e^{alg}, \texttt{element\_length}(\texttt{src}))$.

As an example of how this works its way through a body, consider the program

```
1 let main (n: i32): []i32 =
2   -- Memory: m_a; index function: Direct(n)
3   let a = [0..<n]
4   -- Memory: m_b; index function: Direct(n)
5   let b = copy a
6   -- Memory: m_c; index function: Direct(n)
```

```
7     let c = copy b
8     in c
```

In the actual Futhark compiler, one of the `copy` expressions will be simplified away in a pass prior to the memory block merging optimisations, but we ignore that. We pass through the program:

- On line 5 we have a `copy` expression. $\text{mergeable}(\Sigma, \mathsf{a}, \mathsf{b})$ succeeds, so we get
  - $\mathcal{R}' = [\mathsf{a} \mapsto (m_{\mathsf{b}}, Direct(n))]$, and
  - $\mathcal{CI}' = [(\mathsf{b}, [\mathsf{a}])]$.
- On line 7 we have another `copy` expression. We check both $\text{mergeable}(\Sigma, \mathsf{b}, \mathsf{c})$ and $\text{mergeable}(\Sigma, \mathsf{a}, \mathsf{c})$ (through $\mathcal{CI}(\mathsf{b})$). Both succeed, so we get a new
  - $\mathcal{R}' = [\mathsf{a} \mapsto (m_{\mathsf{c}}, Direct(n)), \mathsf{b} \mapsto (m_{\mathsf{c}}, Direct(n))]$, and
  - $\mathcal{CI}' = [(\mathsf{c}, [\mathsf{b}, \mathsf{a}])]$.

This final $\mathcal{R}'$ tells us that we need to transform the program so that `a` is set to use m_{c}, and `b` is also set to use m_{c}, all with the same index functions.

`concat` expressions can take more than one input array, and can thus result in more than one coalescing:

$$
\frac{\begin{array}{l}
(\mathcal{B}, \mathcal{LU}, \mathcal{BBC}, \mathcal{VA}, \mathcal{S}, \mathcal{CI}) = \Sigma, \\
(\_ \; @ m_{\mathtt{dst}}\{i_{\mathtt{dst}}\}, \quad \_) = \mathcal{B}(\mathtt{dst}), \\
\Sigma' \vdash b \Rightarrow \mathcal{R}' \\
\text{WHERE} \\
\mathcal{R}' = \mathcal{R} \cup \left(\bigcup_{s}^{N} \begin{cases} \mathcal{R}_0(\mathtt{src}_s, i_{\mathtt{cur}}(s)) \cup \mathcal{R}_1(\mathtt{src}_s, i_{\mathtt{cur}}(s)) & \text{if mergeable}(\Sigma, \mathtt{src}_s, \mathtt{dst}) \\ \emptyset & \text{otherwise} \end{cases} \right) \\
i_{\mathtt{cur}}(s) = \mathtt{offset}(i_{\mathtt{dst}}, \sum_{j}^{s-1} \text{size}(\mathtt{src}_j), \mathtt{src}_j) \\
\mathcal{R}_0(\mathtt{src}_s, i_{\mathtt{s}}) = \mathtt{src}_s \mapsto (m_{\mathtt{dst}}, i_{\mathtt{s}}) \\
\mathcal{R}_1(\mathtt{src}_s, i_{\mathtt{s}}) = \bigcup_{(\mathtt{src}_k, e_k^{alg})}^{\mathcal{CI}(\mathtt{src}_s)} (\mathtt{src}_k \mapsto (m_{\mathtt{dst}}, \mathtt{offset}(i_{\mathtt{s}}, e_k^{alg}, \mathtt{src}_k))) \\
\mathcal{CI}' = \mathcal{CI} \cup \left(\bigcup_{s}^{N} \begin{cases} (\mathtt{src}_s, i_{\mathtt{cur}}(s)) & \text{if mergeable}(\Sigma, \mathtt{src}_s, \mathtt{dst}) \\ \emptyset & \text{otherwise} \end{cases} \right) \\
\Sigma' = (\mathcal{B}, \mathcal{LU}, \mathcal{BBC}, \mathcal{VA}, \mathcal{S}, \mathcal{CI}')
\end{array}}{\Sigma \vdash \quad \mathbf{let}\, \mathtt{dst} : \rho = \mathbf{concat}\, \overline{\mathtt{src}}^N \, \mathbf{in}\, b \Rightarrow \mathcal{R}}
$$

(COAL-CONCAT)

Here, $\text{size}(a)$ gives the outer shape size of a, which we can use as an offset. A **`concat`** expression can have arbitrarily many arguments, and we need a way to support coalescing each of them. We accomplish this by extending the result $\mathcal{R}$ with a union of all the succeeding merges, and also updating $\mathcal{CI}$ in the same manner.

In-place updates are similar to **copy**, except the index functions of `src` and `dst` will not be the same even if merged.

$$
\frac{\begin{array}{l}
(\mathcal{B}, \mathcal{LU}, \mathcal{BBC}, \mathcal{VA}, \mathcal{S}, \mathcal{CI}) = \Sigma, \\
(\_\ @m_{\texttt{dst}}\{i_{\texttt{dst}}\}, \quad \_) = \mathcal{B}(\texttt{dst}), \\
\Sigma' \vdash b \Rightarrow \mathcal{R}' \\
\text{WHERE} \\
\mathcal{R}' = \mathcal{R} \cup \begin{cases} \mathcal{R}_0 \cup \mathcal{R}_1 & \text{if mergeable}(\Sigma, \texttt{src}, \texttt{dst}) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{R}_0 = \texttt{src} \mapsto (m_{\texttt{dst}}, \texttt{slice}(i_{\texttt{dst}}, \overline{k})) \\
\mathcal{R}_1 = \bigcup_{(\texttt{src}_k, e_k^{alg})}^{\mathcal{CI}(\texttt{src})} (\texttt{src}_k \mapsto (m_{\texttt{dst}}, \texttt{offset}(\texttt{slice}(i_{\texttt{dst}}, \overline{k}), e_k^{alg}, \texttt{src}_k))) \\
\mathcal{CI}' = \mathcal{CI} \cup \begin{cases} (\texttt{dst} \mapsto ((\texttt{src}, \texttt{slice}(i_{\texttt{dst}}, \overline{k})) \cup \mathcal{CI}(\texttt{src}))) & \text{if mergeable}(\Sigma, \texttt{src}, \texttt{dst}) \\ \emptyset & \text{otherwise} \end{cases} \\
\Sigma' = (\mathcal{B}, \mathcal{LU}, \mathcal{BBC}, \mathcal{VA}, \mathcal{S}, \mathcal{CI}')
\end{array}}{\Sigma \vdash \quad \textbf{let}\ \texttt{dst}[\overline{k}] : \_ \ @\ \_\{\_\} = \textbf{copy}\ \texttt{src}\ \textbf{in}\ b \Rightarrow \mathcal{R}}
$$

(COAL-COPY-INPLACE)

Here, `slice` refers to the function defined in chapter 3, whereby an index function gets to consist of the $\overline{Slice\left(e_{start}^{alg}, e_{length}^{alg}\right)}$ constructor.

Discussion

The attempted coalescing formalisation only covers a very small part of the actual transformation. Specifically, we have not described

- Coalescings through aliasing operations – as described in chapter 5 we do have limited support for coalescing through **reshape** expressions.
- Aliasing operations in general – if an array gets a new memory block and index functions, all of its aliased arrays should also be updated, which our rules do not describe.
- The special handling necessary for **if** and **loop** expressions.
- Having to handle all previously-merged entries from $\mathcal{CI}$ at every potential coalescing (since they may have to reassigned to a new memory block) has inefficient worst-case quadratic time. If we instead were to coalesce in a bottom-up fashion, we would follow the direction of the coalescings more naturally, and might not have to revisit previous mergings as much.

Refer to chapter 5 for the full algorithms, albeit more informally stated.

Chapter 9

Limitations

We describe in this chapter the limitations of our transformations and the implementation of them. We divide the limitations into two categories: Those caused by our choice of algorithm, and those caused by a lacking implementation. It is possible that this list is incomplete.

Memory Coalescing

Array Reshapes

We currently do not handle the coalescing of reshapes from a source array if it has aliases. See figure 9.1 for an example. This program has two arrays: `t0` and `t1`. It is instructive to look at the memory block-annotated version of the program; see figure 9.2.

```
1 let main [n] (ns: [n]i32): [1][n]i32 =
2   let t0 = map (+ 1) ns
3   let t0a = reshape (1, n) t0
4   let t1 = copy t0a
5   in t1
```

Figure 9.1: A potential coalescing "hidden behind" a `reshape` operation.

```
1  -- Memory of ns: m_ns
2  let main (ns: [n]i32) =
3    let n_bytes = n * 4
4
5    let m_t0 = alloc n_bytes
6    -- Memory: m_t0; index function: Direct(n)
7    let t0 = kernel map ...
8
9    -- Memory: m_t0; index function: Direct(1, n)
10   let t0a = reshape (1, n) t0
11
12   let m_t1 = alloc n_bytes
13   -- Memory: m_t1; index function: Direct(1, n)
14   let t1 = copy t0a
15
16   in t1
```

Figure 9.2: The program in figure 9.1 run through the GPU pipeline in the Futhark compiler, and simplified a bit.

Since $m_{\texttt{t1}}$ has the same size as $m_{\texttt{t0}}$, and since $n = 1 \cdot n$, it should be possible to coalesce `t0a` into $m_{\texttt{t1}}$ if we ignore safety condition 4. If we do this, and we change `t0a` to use $m_{\texttt{t1}}$, we however also need to update all its aliased variables – `t0` – and set them to use the index function of `t1`.

Assuming we are not doing this wrong, this does *not* seem way too cumbersome to properly describe and make work. However, now consider the (un-annotated) program in figure 9.3.

```
1  let main [n] (t1: *[2][1][n]i32, ns: [n]i32): [2][1][n]i32 =
2    let t0 = map (+ 1) ns
3    let t0a = reshape (1, n) t0
4    let t1[1] = copy t0a
5    in t1
```

Figure 9.3: A program with `reshape` and an in-place update.

Since `t1` is an in-place update copying `t0a`, its index function will be different from `t0a`, whose index function is already different from `t0` because of the `reshape`. Coalescing `t0a` into `t1` would then mean changing both the index function of `t0a` and `t0`, but in different ways.

Again, assuming we could formalize this, it should be doable, but it *does* start getting hairy.

Note that the implementation actually ad-hoc handles `reshape (dim)` expressions, i.e. `reshape` operations with only one dimension. These are occasionally used in the Futhark compiler to require that two sizes are equal – e.g. if you have a sequence

```
let xs = some array of size t
let ys = reshape (u) xs
```

then you guarantee that the program can only complete if $t = u$. This case is simple to handle, as it does not require changing index functions.

Lack of Memory Block Merge Parameters

Consider the program in figure 9.4.

The coalescing of `a1` into $m_{\texttt{x}}$ on line 5 causes problems for the loop: In the first iteration `b0` uses $m_{\texttt{a0}}$ – which has no special index function – but in the second iteration, if there is a coalescing, `b0` uses $m_{\texttt{x}}$ with an index function representing the nonzero offset of the first `concat` argument (since `a1` is the second argument).

Currently this case is disabled, but a better solution is maybe to extend the internal `loop` accumulators with index functions, so that each iteration can use different index functions. Currently we have just disabled the case where a loop is coalesced into an array with an index function more complex than a $Direct(\ldots)$.

This is not just a problem for `concat`; the same happens for in-place updates, since they add a slice to the index function of the coalesced array.

```
let main [n] (a0: [n]i32): []i32 =
  let a1 = loop b0 = a0 for _i < n do
    let b1 = map (+ 1) b0
    in b1
  let x = concat a0 a1
  in x
```

Figure 9.4: A Loop with merge paramaters in its EXPLICITMEMORY form.

Index Analysis Across Loop Iterations

Consider the program in figure 9.5.

The `b1` memory currently cannot be coalesced into the accumulator memory, because `b1` and `b0` can refer to the same memory block due to aliasing. As described in chapter 6, we do have an index access analysis that finds the cases where writing to one array while reading from another array happens in the same indices (interference exceptions), resulting in potentially more reuses.

However, this analysis does not work across loop iterations right now, which is why this program will end up with two allocations – `b1` and a generated accumulator memory block – instead of one – just the accumulator memory block, reused by `b1`.

```
let main [n] (a0: [n]i32): []i32 =
  let a1 = loop b0 = a0 for _i < n do
    let b1 = map (+ 1) b0
    in b1
  in a1
```

Figure 9.5: A program where the memory block can vary across iterations.

It should be possible to extend the algorithm to handle the very specific cases for this.

`concat` Memory Copying Removal

The `concat` feature described in section 5.5 – whereby code generation for `concat` expressions does not copy the memory of its arguments if they have been coalesced – is currently implemented as a kludge. It would be nice to do this properly.

If the memory block coalescing optimisation has been run prior to this pass, it may have coalesced the source and destination memory of some **`concat`** expressions, in which case there is no need to insert a data copy in the imperative code.

We have extended the compiler with the first four lines in the snippet

```
let needs_copy = not usesMemoryBlockMergingCoalescing
                 || (let srcmem = memLocationName srcloc
                     in srcmem /= destmem)
when needs_copy $
  copy (elemType xtype) destloc srcloc $ arrayOuterSize yentry
```

where the fifth line is the original handling. It should be self-explanatory.

We have manually verified that this indeed does generate C code without memory copying, but this does not scale, implementation-wise. For this reason it exists only at the benchmarking *branch* (used for benchmarking Futhark programs) described in appendix A. This minor optimisation should only have an effect on speed, not memory usage.

In the master branch of Futhark, we would like to do this properly by comparing the index functions in the general Futhark compiler `copy` Haskell function, instead of only at the **`concat`** handling. This is more elegant and more future-proof.

Compiler Inefficiencies Because of Modularization

The current compiler implementation for the transformations described in chapter 5 is likely not as efficient as it could be. Note that we are discussing the speed of the code *generator*, not the *generated code*.

For the memory coalescing transformation, several of the five safety conditions described in chapter 5 are implemented in separate modules like this:

Safety condition 2 Traverses each function body once and records all allocations.

Safety condition 3 Has a function that traverses each function body once and records all variables that are used between two bindings. This is $\mathcal{O}(n)$ time, but the function can be called many times.

Safety condition 5 Traverses each function body once and records, for every memory block that it finds, all the variable names in scope at that point. The traversal is $\mathcal{O}(n)$ time, but the resulting mapping can become quite large for large programs.

The benefit is that the implementation is arguably fairly readable, but the downside is that we have unappealing time complexities, and several intermediate structures from the results of having multiple separate passes through programs. We have not made any measurements, but we believe there are many low-hanging fruits to pick in this area – though we warn against starting an efficiency tirade without considering the complexity of this project. There are so many edge cases, and it would be very, very nice if the implementation remains readable.

Memory Reuse

Limitations of the Linear Scan Algorithm

The program in figure 9.6 is a demonstration of how the linear scan register allocation-inspired algorithm of memory reuse does not produce an optimal result. The optimisation can reduce the number of allocations from 4 to 3, while an optimal analysis can reduce it to 2 allocations.

```
1  let main [n] (ns: [n]i32): [n]i32 =
2    let k0 = 1
3    let xs0 = map (+ k0) ns
4
5    let k1 = xs0[0]
6    let xs1 = map (+ k1) ns
7
8    let k2 = xs1[0]
9    let xs2 = interfering_map k2 xs1
10
11   let k3 = xs2[0]
12   let xs3 = interfering_map2 k3 xs0 xs2
13
14   in xs3
```

Figure 9.6: A program that our linear scan cannot handle optimally.

where `interfering_map` and `interfering_map2` are `map` functions in nature, but with uneven index accesses for reading and writing, so that our input reuse (index access) analysis will not remove their interferences. We define them as so:

```
let interfering_map [n] (k: i32) (t: [n]i32): [n]i32 =
  map (\i -> t[n - i - 1] + k) [0..<n]

let interfering_map2 [n] (k: i32) (t: [n]i32) (u: [n]i32): [n]i32 =
  map (\i -> t[n - i - 1] + u[n - i - 1] + k) [0..<n]
```

The actual results do not matter; we only care about them producing interferences in the analysis.

We get this interference table:

1. `xs0` interferes with nothing
2. `xs1` interferes with nothing
3. `xs2` interferes with `xs1`, `xs3`
4. `xs3` interferes with `xs0`, `xs2`

By following the reuse algorithm we get this top-down traversal:

1. `xs0` cannot use anything
2. `xs1` uses `xs0`; `xs0` and `xs1` now interfere
3. `xs2` cannot use anything
4. `xs3` cannot use anything

We get a single memory block merging. We have tried to ad-hoc find a better way, and managed to end up with these mergings:

1. `xs2` uses `xs0`
2. `xs3` uses `xs1`
3. `xs0` cannot use anything
4. `xs1` cannot use anything

These mergings are valid because there are no interferences between `xs2` and `xs0`, or between `xs3` and `xs1`. We have shown that the reuse algorithm can produce non-optimal results.

Conditional Last Uses

See the program in figure 9.7.

```
1  let main [n] (xs0: [n]i32, cond: bool, i: i32): [n]i32 =
2    let xs = map (+ 1) xs0
3    let k = xs[i]
4    let ys =
5      if cond
6      then let zs_then = replicate n k
7           in zs_then
8      else let zs_else = xs
9           in zs_else
10   in ys
```

Figure 9.7: A program with different last uses depending on which branch is taken.

Depending on which branch you take in the if-then-else expression on line 5, `xs` can be thought of to have different live intervals:

- If the program execution takes the `then` branch, the last use of `xs` is in the `k` statement on line 3.
- If the execution takes the `else` branch, the last use of `xs` is in the `zs_else` statement on line 8.

We would like the `zs_then` array in the `then` branch to reuse the memory of `xs`, which should be possible because we know it has been lastly used if we the program execution enters that branch.

However, our current analysis declares conservatively that the last use of `xs` is in the entire loop statement of `ys`, so we do not get this memory merging.

A way to handle this could be "conditional last uses": For example, the `k` statement would include the last use of $m_{\texttt{xs}}$ with an attribute denoting "this is only true if the execution takes the `then` branch in the `if` expression whose result is stored in `ys`", but this idea needs way more work.

Limitations on First-Order-Transformed Programs

When compiling Futhark code into C via a CPU-oriented pipeline in the compiler, it first follows the initial steps of the general pipeline and at some point ends up with a program in an intermediate representation with kernels – similar to EXPLICITMEMORY, but without memory annotations. A CPU-specific pass than performs a first-order transform on all kernels and transforms them to loops. These loops calculate exactly the same as the kernels, just sequentially instead of in parallel.

Our analyses and transformations work on both kernels and loops. However, implementation-wise, loops originating from kernels are often harder to analyse than their original kernels, so we have kludges here and there to make sure we extract the right relationships for both kinds of constructs. This works well for the most part.

See figure 9.8 for a program that gets two fewer memory block mergings in the CPU pipeline compared to the GPU pipeline.

```
let main (xss: *[][]i32): [][]i32 =
  map (\xs ->
       let ys = map (+ 1) xs
       let k = ys[0]
       let zs = map (+ k) ys
       in zs) xss
```

Figure 9.8: The program in the source language.

In the GPU pipeline, the program gets flattened into the two two-dimensional kernels in figure 9.9, corresponding to the two inner maps in the source program. This is easy to analyse and results in two memory reuses into the unique parameter memory.

```
let main (xss: *[][]i32): [][]i32 =
  ...
  kernel map
  ...
  kernel map
```

Figure 9.9: The program structure in its kernel representation.

In the first-order-transformed program, we end up with the structure in figure 9.10. In this representation, each `map` expression from the source language corresponds to one of the inner loops *and* the outer loop. To handle this, we need to maintain a context of which loop we are inside, and try to reconstruct the original construct, and make sure that its guarantees still hold.

```
let main (xss: *[][]i32): [][]i32 =
    loop
        loop
        loop
```

Figure 9.10: The program, first-order-transformed from kernels into loops.

Alternatively, we could maybe perform the first-order transform after memory block merging, or we could keep the kernel structure around for a while, but this is outside the scope of this thesis.

Meta-limitation: Constructing Interesting Programs

As described in chapter 3, the memory block transformations happen in the EXPLICITMEMORY representation. Before a Futhark program reaches this point, it has been through a long pipeline of various compiler transformations, so it can be hard to write a Futhark program in the source language with the intention of it looking a certain way when it reaches the EXPLICITMEMORY representation.

For example, we were not able to write a really good test for data race interferences (described in section 5.9), even though we know it occurs in a benchmark program. A programmer currently needs to be intimate with *both* the theoretical foundations of the published Futhark transformations, *and* the internals of the Futhark compiler, to be able to predict what is going to happen with a program.

It might be nice with a way to write Futhark programs in their intermediate representations. All our transformations take a program in EXPLICITMEMORY form and returns a program in EXPLICITMEMORY form, so we would only absolutely need to write programs in that representation. Several disadvantages pop up, though:

- We would have to freeze the intermediate representation(s). Currently we have the benefit of being able to change them without affecting the programmer experience, as long as the source language does not change.
- Someone would have to make a proper grammar for the intermediate representation(s). Currently they can be formatted, but this formatting is ad-hoc and likely not precise enough on its own to be able to write a parser.
- Programmers would be able to write a program in an intermediate representation that would never have gotten there "naturally", i.e. through the normal pipeline. This is not necessarily bad, but might lead to writing programs for meaningless scenarios.

Chapter 10

Evaluation

In this chapter we evaluate our algorithms and implementation; qualitatively by analysing the effects of the optimisations on purposely-written pathological test programs and four benchmark programs, and quantitatively by running a large automatic test suite. We also show the improvements and slowdowns we get by enabling our optimisations on more than 30 pre-existing Futhark benchmark programs.

Existing Transformation: In-Place Lowering

Futhark has an **existing** memory block optimisation called *in-place lowering*, which we want to be better than. We describe here its purpose, so we can argue how well our memory block optimisations can replace it.

In-place lowering is an optimisation that reduces memory copying by moving in-place updates into loops. As an example, the `map` expression of

```
let main [n][m] (rss: *[n][m]i32): [][]i32 =
  map (\(rs: *[]i32) ->
        loop rs for j < m do
          let rs[j] = rs[j] + 1
          in rs) rss
```

is – in the CPU pipeline – first turned into (morally)

```
  loop rss for i < n do
    let arg = rss[i]
    let rs = copy arg
    let rs = loop rs for j < m do
      let rs[j] = rs[j] + 1
      in rs
    let rss[i] = copy rs
```

```
8       in rss
```

in which both the outer and inner loops make in-place updates. The generated `copy` statement on line 7 is an easy way to ensure that the transformed program is still valid, and can often be optimised away. Applying in-place lowering to the program in its current state results in

```
1    loop rss for i < n do
2      let arg = rss[i]
3      let rs = copy arg
4      in loop rss for j < m do
5        let rss[i, j] = rss[i, j] + 1
6        in rss
```

where the last `copy` has been removed in favour of *lowering* the `rss[i] = ...` statement into the inner loop body by replacing occurences of `rs[j]` with `rss[i, j]`. This is *one* way to get rid of the copying. As mentioned earlier, our memory block coalescing algorithm instead does this by index function trickery and merging memory blocks.

For this program example, the end result is the same in terms of peak memory usage and memory access for both in-place lowering and memory block coalescing, but this is just one example. We will look at at least one benchmark program that our memory block optimisations seemingly do not handle as well as in-place lowering.

In-Depth Analyses

In this section we manually go through large Futhark programs and describe how they are handled by our optimisations, serving as an evalution of those. We completely ignore what these benchmarks are supposed to *do*, and only care about their *structure*.

Memory Coalescing

Benchmark Analysis: Srad

We start by analysig a small benchmark program where the GPU pipeline leads to no coalescings, while the CPU pipeline leads to several ones. Srad (`rodinia/srad/srad.fut` in the Futhark benchmarks repository) is a benchmark ported from the Rodinia[24] benchmark suite into Futhark.

However, the coalescings in the CPU pipeline are not due to a more easily-analysable program, but rather the inserted double-buffer memory blocks described above. These go away with the in-place lowering algorithm, and they also go away with our memory block coalescing algorithm.

Note: This benchmark has no change in memory usage compared to using the existing in-place lowering optimisation. We show it to argue that our optimisation performs the same duties as in-place lowering. See figure 10.1 for an abstract overview.

```
1 main =
2     loop
3     loop
4         loop
5             loop
6         let res = loop
7         copy res
8     loop
```

Figure 10.1: An overview of the structure of the `srad` program run through the CPU pipeline.

We ignore variable names except when they are used later on in a `copy` expression, of which we have one. We extend line 6 and 7 in figure 10.2.

```
1 let {[size_1][size_2]f32 res} =
2   -- map_outarr : *[size_1][size_2]f32@mem_outarr->Direct(size_1, size_2)
3   loop {*[size_1][size_2]f32 map_outarr} = {result_3994}
4   for i:i32 < size_1 do {
5     ...
6   }
7 -- double_buffer_array : [size_1][size_2]f32@double_buffer_mem->Direct(size_1, size_2)
8 let {[size_1][size_2]f32 double_buffer_array} = copy res
```

Figure 10.2: A closeup of the expression and its copy statement.

It is important to note that it is irrelevant that `res` is itself a loop; the double buffer memory and the copy statement is inserted because they are inside the loop started at *line 3*.

Our optimisation correctly identifies this as a coalescing opportunity, and ends up having `map_outarr` write to `double_buffer_mem` instead of its original `mem_outarr`.

Benchmark Analysis: Series

Contrary to the srad benchmark, the `series` benchmark (`jgf/series/series.fut` in the Futhark benchmarks repository) has coalescing – two of them – that are *not* handled by the in-place lowering algorithm (while still handling all the cases already handled by in-place lowering). The series benchmark is ported from the jgf benchmark suite.

We present the core part of the source program in figure 10.3. Both `first` and `rest` are arrays of the same type of tuples, though they have different sizes: `first` has element size 1, while `rest` has element size $\texttt{array\_rows} - 1$. At the end, these two arrays are concatenated, resulting in a return array of `array_rows` elements. `unzip` is a syntanctic construct that takes an array of tuples and gives back a tuple of arrays.

```
let main (array_rows: i32): ([array_rows]f64, [array_rows]f64) =
  let first = [(..., 0.0)]
  let rest = map (\i -> ...) [0..<(array_rows-1)]
  in unzip (concat first rest)
```

Figure 10.3: An overview of the structure of the `series` source program.

We focus only on the memory block mergings that the coalescing optimisation will bring, and not those that already occur when using in-place lowering; see the program overview after being run through the GPU pipeline in figure 10.4 – we leave out memory annotations, but know that every array name in the program has an associated memory block.

```
main =
  let first_0 = replicate(1, ...)
  let first_1 = replicate(1, 0.0)
  let (rest_0, rest_1) = kernel map
  let result_0 = concat@0(first_0, rest_0)
  let result_1 = concat@0(first_1, rest_1)
  in (result_0, result_1)
```

Figure 10.4: An overview of the structure of the `series` program run through the GPU pipeline.

The silly thing about this small benchmark is that, to accomodate the structure of the return arrays – one initial element, followed by $n - 1$ "ordinary" rest elements – it has large memory blocks for both the return arrays *and* the intermediate rest arrays, and copies every element once. If we were writing this in an imperative programming language, we would have rewritten it to not have this inefficiency; instead our compiler pass handles it with index function trickery, as shown in figure 10.5.

```
let main =
  -- Memory of first_0: m_result_0; index function: Direct(1)
  let first_0 = replicate([1], ...)
  -- Memory of first_1: m_result_1; index function: Direct(1)
  let first_1 = replicate([1], 0.0)
  -- Memory of rest_0: m_result_0; index function:
  --      Direct(array_rows)->Slice(1, array_rows - 1)
  -- Memory of rest_1: m_result_1; index function:
  --      Direct(array_rows)->Slice(1, array_rows - 1)
  let (rest_0, rest_1) = kernel map
  -- Memory of result_0: m_result_0; index function: Direct(array_rows)
  let result_0 = concat@0(first_0, rest_0)
  -- Memory of result_1: m_result_1; index function: Direct(array_rows)
  let result_1 = concat@0(first_1, rest_1)
  in (result_0, result_1)
```

Figure 10.5: An overview of the structure of the `series` program run through the GPU pipeline.

The *interesting* thing about the benchmark is how much it benefits from the `concat` part of our coalescing optimisation. As shown in section 6, this benchmark actually gets a 50% reduction in peak memory usage just by merging memory blocks.

Memory Reuse

Benchmark Analysis: Canny

We present the canny (`accelerate/canny/canny.fut` in the Futhark benchmarks repository; ported from Accelerate) benchmark program in figure 10.6.

```
let main input =
             -- Unique input memory:  m_0
  map        -- Create memory block:  m_1
  map        -- Create memory block:  m_2
  map        -- Create memory block:  m_3
  map        -- Create memory blocks: m_4, m_5
  map        -- Create memory block:  m_6
  scanomap   -- Create memory blocks: m_7, m_8
  replicate  -- Create memory block:  m_9
in m_9
```

Figure 10.6: A memory block-centric overview of the canny program as run through the CPU pipeline.

By running our interference analysis on the program, we get the interferences in figure 10.7.

$$
\begin{aligned}
&m_0 \to m_1, \quad && m_1 \to m_0, m_2, \quad && m_2 \to m_1, m_3, \quad && m_3 \to m_2, m_4, m_5, \\
&m_4 \to m_3, m_5, m_6, \quad && m_5 \to m_3, m_4, m_6, \quad && m_6 \to m_4, m_5, m_7, m_8, \\
&m_7 \to m_6, m_8, m_9, \quad && m_8 \to m_6, m_7, m_9, \quad && m_9 \to m_7, m_8
\end{aligned}
$$

Figure 10.7: Interferences in canny.

We perform this analysis on the CPU pipeline. This makes the analysis slightly simpler, as the GPU pipeline introduces additional memory blocks to aid its kernels.

Recall that an interference table is not enough to determine if a memory block can be reused. It must also be the case that the memory block sizes match or can be maximised. For simplicity, we omit a sizes table. See figure 10.8 for the program after our pass.

```
1  let main input@m0 =
2    map@m1            -- No reuses.
3    map@m2            -- Reuses:  m2 reuses m0
4    map@m3            -- Reuses:  m3 reuses m1
5    map@m4, m5        -- Reuses:  m4 reuses m0
6    map@m6            -- Reuses:  m6 reuses m1
7    scanomap@m7, m8   -- Reuses:  m7 reuses m5
8    replicate@m9      -- No reuses.
9  in m9
```

Figure 10.8: The results of running the memory reuse pass on the canny program.

We have gone from 10 to 5 memory blocks: m_0, m_1, m_5, m_8, m_9. Since the `input` memory is allocated by the caller, this means that we have gone from 9 to 4 memory block *allocations*. For each of the 4 still-allocated memory blocks, we investigate why they cannot reuse other memory. Recall that the algorithm works by doing a single top-down pass.

m_1 At this point in the program, only m_0 is available, but this block interferes with m_1.

m_5 Available memory blocks:

- m_0 Currently reused by m_2, m_4. m_5 interferes with m_4, so it cannot reuse m_0.
- m_1 Currently reused by m_3. m_5 interferes with m_3, so it cannot reuse m_1 either.

m_8 Available memory blocks:

- m_0 Currently reused by m_2, m_4. None of these memory blocks interfere with m_8, but m_0 has a different size than m_8. Since m_0 is created by the caller, we cannot change its size.

m_1 Currently reused by m_3, m_6. m_8 interferes with m_6, so it cannot reuse m_1.

m_5 Currently reused by m_7. m_8 interferes with m_7, so it cannot reuse m_5 either.

m_9 Available memory blocks: m_0, m_1, m_5, m_8. Some interfere, some do not. However, they all have a different size than m_9. The size of m_9 is only determined at the end of all previous calculations, so it cannot be hoisted up before any of the previous memory allocations, and so none of the available memory blocks can have their size changed to accomodate m_9.

The take-away so far is that m_1 and m_9 will never be able to reuse any memory, and that the linear algorithm results in 5 reuses. We want to find out if this is optimal: Can we get more reuses if we pick them by hand? See figure 10.9 for an ad-hoc attempt at this.

```
1  let main input@m0 =
2    map@m1            -- No reuses.
3    map@m2            -- Do not reuse anything.  Instead let m2 be available.
4    map@m3            -- Reuses:  m3 reuses m0
5    map@m4, m5        -- Reuses:  m4 reuses m1, m5 reuses m2
6    map@m6            -- Reuses:  m6 reuses m1
7    scanomap@m7, m8   -- Reuses:  m7 reuses m5, m8 reuses m2
8    replicate@m9      -- No reuses.
9  in m9
```

Figure 10.9: The result of an ad-hoc reuse strategy.

For no particular reason, we choose to not let m_2 reuse m_0, allowing us to reuse m_2 for both m_5 and m_8, thus getting one more memory reuse in total.

We can perform these two reuses because

- Neither m_5 or m_8 interferes with m_2 or any of the existing reuses – at the point of m_5, none; at the point of m_8, only m_5, and
- m_2 and m_5 have the same size, and
- m_2 and m_8 do not have the same size, but the size of m_8 is entirely determined by the function parameters, and so they are all in scope at the creation of m_2, meaning that the size of m_2 can be extended.

In section 9.2 we showed that the linear scan algorithm was non-optimal for a pathological test created for the occasion. Here we have shown that non-optimality can also occur in a real-world benchmark.

Benchmark Analysis: Option Pricing

We present the Option Pricing benchmark program (`finpar/OptionPricing.fut` in the repository) in figure 10.10. This benchmark is ported from the Finpar[1, 20] benchmark suite.

```
let main =
    replicate@m1
    replicate@m2
    kernel chunked_map@m3
        loop@m4
        copy@m5
        copy@m6
        loop
            if-then-else@m7
            loop@m8
            loop@m9
            loop@m10
                loop@m11
                loop@m10
                loop@m10
                copy@m10
            loop@m12  -- Reuses:  m12 reuses m8
                loop@m13
                    loop@m13
                    copy@m13
            loop@m14
            copy@m5
            copy@m6
        loop@m15  -- Reuses:  m15 reuses m4 (or m15 reuses m5)
    if ... then
        -- Reuses:  m16 reuses m1 (or m16 reuses m2)
        kernel segmented_redomap__large_comm_one@m16
            combine@m17
    else
        kernel segmented_redomap__large_comm_many@m18
            combine@m19
        if ... then
            -- Reuses:  m20 reuses m2
            kernel segmented_redomap__large_comm_one@m20
                combine@m21
        else
            kernel segmented_redomap__small_comma@m22
                combine@m23
                combine@m24
                scan@m23
    kernel map@m25  -- Reuses:  m25 reuses m3
```

Figure 10.10: The results of running the memory reuse pass on the OptionPricing program.

Note that for this benchmark we have already gone through the coalescing part, which explains

why some expressions already share memory blocks. Also, we present it in its GPU pipeline version to highlight the changes that may lead to.

This program is very large compared to the canny benchmark. There are 25 memory blocks (though a few of them are existential), and too many interferences to show. We have left out many details to keep it on a single page. For all these reasons we choose to take a bird's eye approach to analysing it.

We only get five memory block mergings. Before the program reaches this form, it has been through many passes:

- A part of the source program has been turned into a chunked `map` kernel, meaning it maps over an array in chunks, which is why the body of the kernel contains several loops: Each chunk is sequentialised.
- A part of the source program has been turned into a segmented `redomap`, meaning a kind of `reduce . map` fusion being mapped in segments.

These transformations (among others) have led to this program that we find too hard to properly understand all aspects of; we give up on analysing it any further, and leave that for future work.

In general, our memory block optimisations do analyses both outside and inside kernel bodies – which is also evident from the reuses in the program that occur inside a kernel – but perhaps they need more extensions and heuristics to do it better, and to somehow reduce the interference table, whose size is the main culprit in limiting mergings here. However, it is not clear to us whether this giant interference table is overly conservative, or if the transformed benchmark program is just impossible to optimise further without resorting to completely different algorithms.

Quantitative Testing

We have created 95 memory block-specific Futhark test programs of varying complexity, many of which have been used as examples throughout this thesis. All of these tests are annotated with at least one (and typically both) structure tests for the number of allocations in the transformed programs:

- `structure cpu { Alloc` n `}`
- `structure gpu { Alloc` n `}`

Here, n is the number of expected allocations in the final program, fully optimised program, which we have a tool to automatically check. `structure cpu` covers the CPU pipeline, while `structure gpu` covers the GPU pipeline. This is a very rough check: We essentially measure a side-effect of our optimisations, namely the subsequent removal of unused allocations.

In addition to structure test, we have also annotated almost all programs with input-output checks:

- `input {` $arguments\ to\ the$ `main` $function$ `}`

- `output { `*expected return values*` }`

These ensure (to some extent) that, not just are the programs transformed as expected, but they still give the right results. The automatic tool also handles these annotations. For input-output annotations we need to tell the *tool* which pipeline we want to run. We run both.

Additionally, the Futhark compiler distribution contains 765 general non-memory-block-specific tests (as of this writing), many of which also have input-output annotations. For our purposes, these tests do not check whether our transformations work as expected, but they do check (to some extent) that any transformations that *do* happen still produce programs that give the right results.

We have run all tests with in-place lowering disabled, memory block coalescing enabled, and memory block reuse enabled.

All of our 95 memory block tests validate on both kinds of automatic tests, and on both pipelines. All 765 pre-existing general tests also validate.

Benchmarking

The Futhark compiler distribution comes with 39 benchmarks (as of this writing), Several of them variations over a theme; if we were to discount benchmarks that are merely variations of other benchmarks and compute the same things, the number goes down to 30. The benchmarks cover areas such as physics simulations, financial contracts, graph algorithms, and more.

Each benchmark program has a set of input-output dataset pairs. We have run each benchmark (including variants), checked that it validates, and measured several metrics. Each benchmark has been run in two configurations:

Before

- In-place lowering enabled.
- Memory block coalescing disabled.
- Memory block reuse disabled.

After

- In-place lowering disabled.
- Memory block coalescing enabled.
- Memory block reuse enabled.

The **Before** configuration is the vanilla Futhark compiler behaviour.

We have collected the following metrics:

Peak memory usage
: *Per-dataset.* How many bytes does the program use at its peak? In this section we will focus on the average peak memory usage over all datasets of a benchmark, but all raw

measurements are in the appendix. These kinds of measurements are coloured blue in the plots.

Runtime
Per-dataset. How long does it take to run the program in wall-clock time (ignoring initial and final input-output data transfers)? Each dataset is averaged over 10 runs, and preceded by a warmup run whose result is ignored. As in the peak memory usage metric, we will look at the averaged runtimes over all datasets of a benchmark. These kinds of measurements are coloured red in the plots.

Additionally we have also measured

1. the total number of bytes allocated (per-dataset);
2. the total number of bytes freed (per-dataset);
3. the number of memory coalescings (in the compilation phase); and
4. the number of memory reuses (in the compilation phase)

but choose not to focus on them in this section; metrics 1 and 2 are partly covered by the peak memory usage metric, and metrics 3 and 4 are more interesting as secondary metrics.

We have run the benchmarks on a system with these attributes:

- CPU: 4 HT cores of Intel Core i7-4720HQ at 2.60GHz
  - L1 cache: 32 KiB
  - L2 cache: 256 KiB
  - L3 cache: 6 MiB
- GPU: GeForce GTX 960M

See appendix B for the raw measurements.

All benchmarks validate with all input-output datasets in the **After** configuration, except one – called `heston32` – which segfaults in the program generated with the CPU pipeline; we discuss this in section 7.

We have measured the peak memory usage metric for all benchmark programs. We have measured the runtime metric *only* for the benchmark programs that show a non-zero reduction in peak memory usage in the **After** configuration. We found this restriction to be necessary to avoid overly long total benchmarking times – runtimes are averaged over 10 runs, while peak memory usage is found in a single run. This way we could get all interesting runtimes overnight.

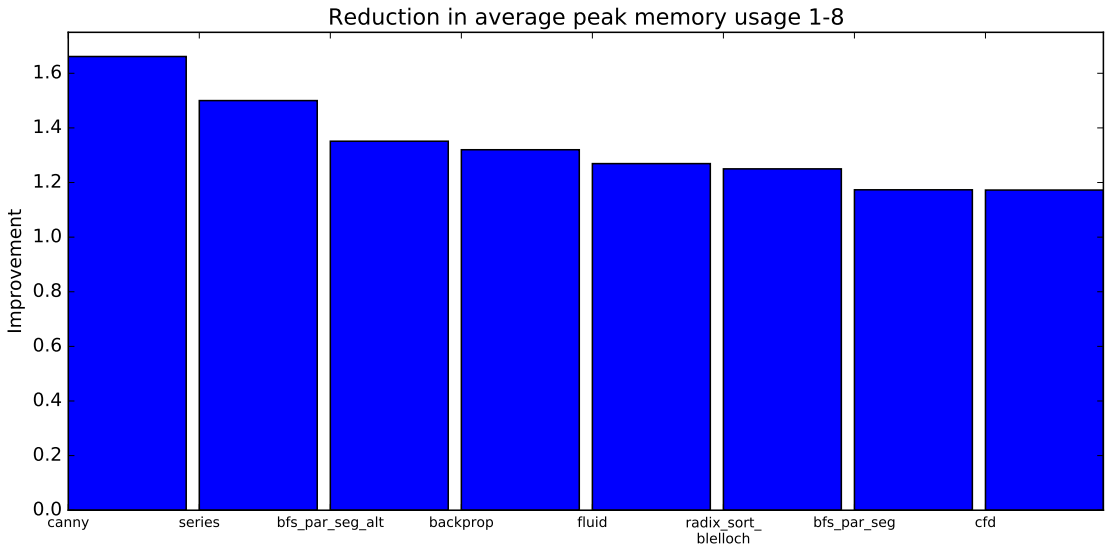
We now present our measurements. The show the relative improvement going from the **Before** configuration to the **After** configuration. For each metric we leave out all benchmarks with zero change between the **Before** and **After** configurations.

Figure 10.11 shows the average peak memory usage improvement in the CPU pipeline.

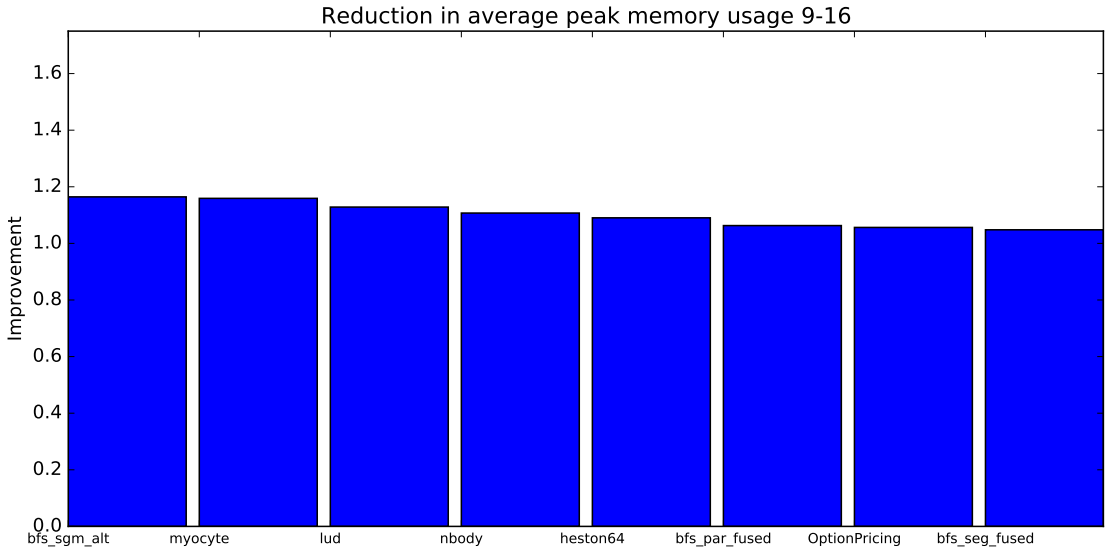
Figure 10.12 shows the average runtime improvement in the CPU pipeline.

Figure 10.13 and figure 10.14 show the average peak memory usage improvement in the GPU pipeline.

Figure 10.15 and figure 10.16 show the average runtime improvement in the GPU pipeline.

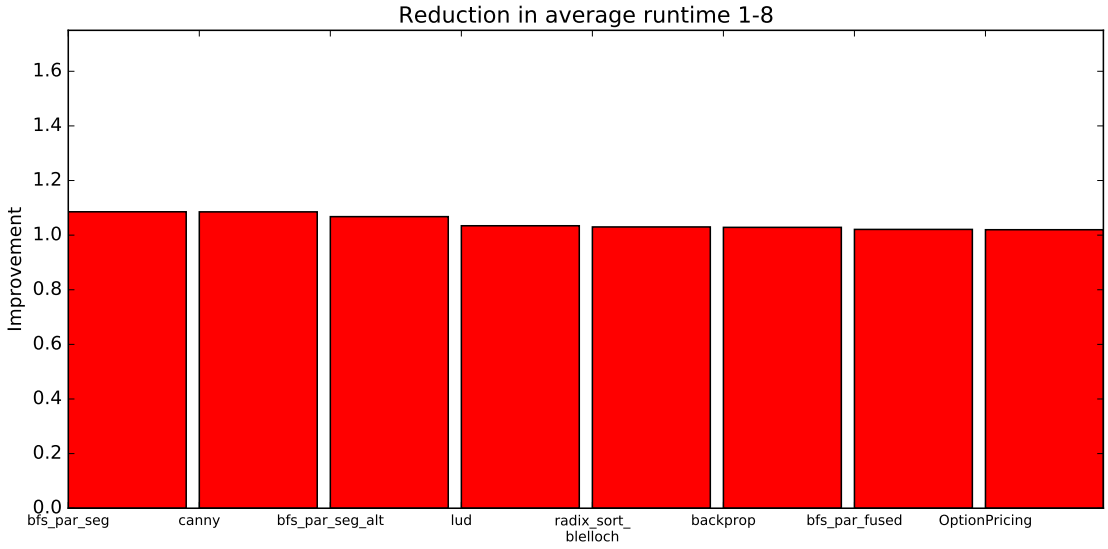


canny	series	bfs_par_seg_alt	backprop	fluid	radix_sort_ blelloch	bfs_par_seg	cfd
x1.66	x1.50	x1.35	x1.32	x1.27	x1.25	x1.17	x1.17

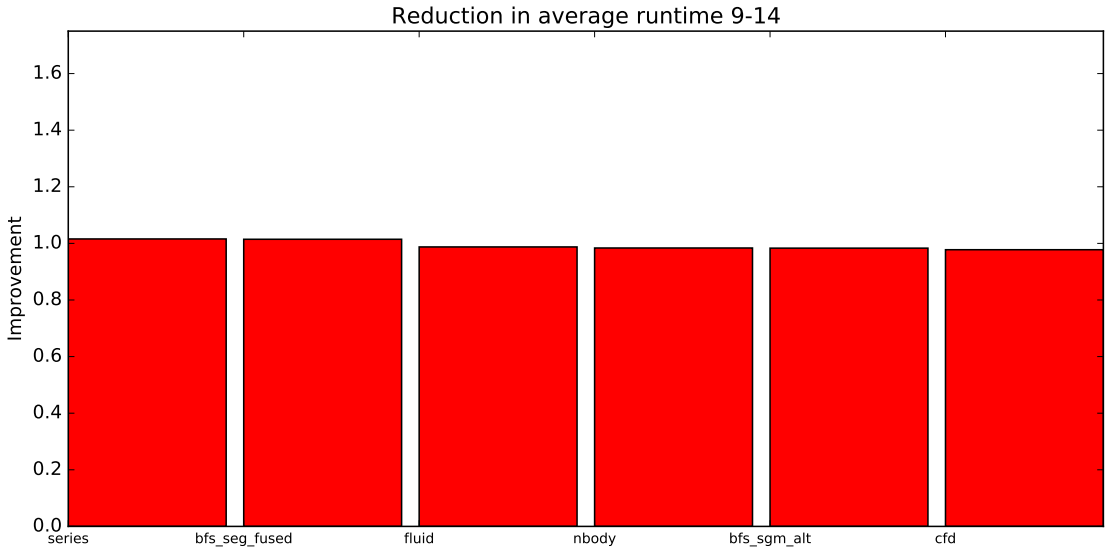


bfs_sgm_alt	myocyte	lud	nbody	heston64	bfs_par_fused	OptionPricing	bfs_seg_fused
x1.16	x1.16	x1.13	x1.11	x1.09	x1.06	x1.06	x1.05

Figure 10.11: Average peak memory usage improvement in the CPU pipeline.

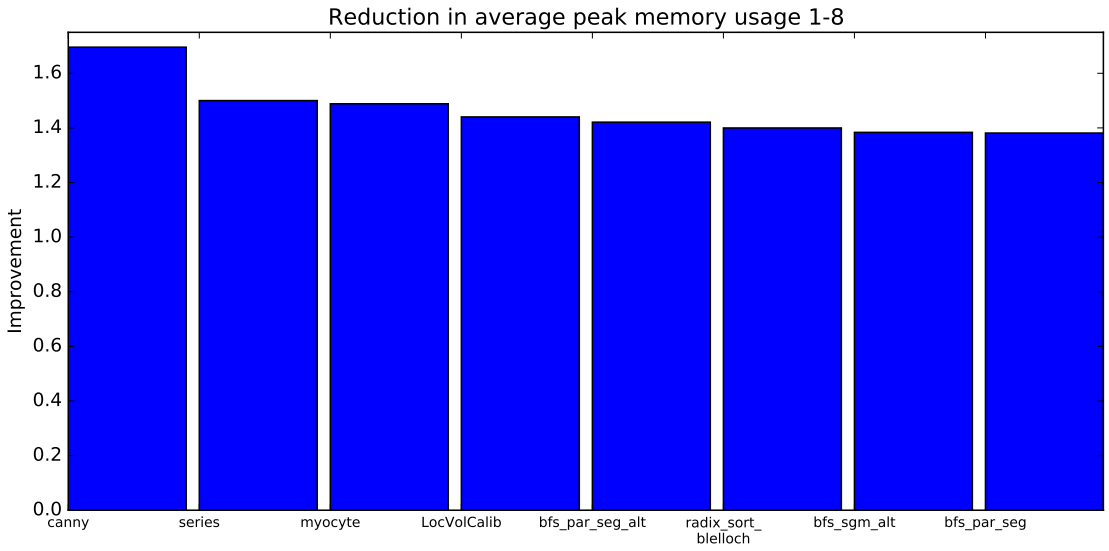


bfs_par_seg	canny	bfs_par_seg_alt	lud	radix_sort_ blelloch	backprop	bfs_par_fused	OptionPricing
x1.09	x1.09	x1.07	x1.03	x1.03	x1.03	x1.02	x1.02

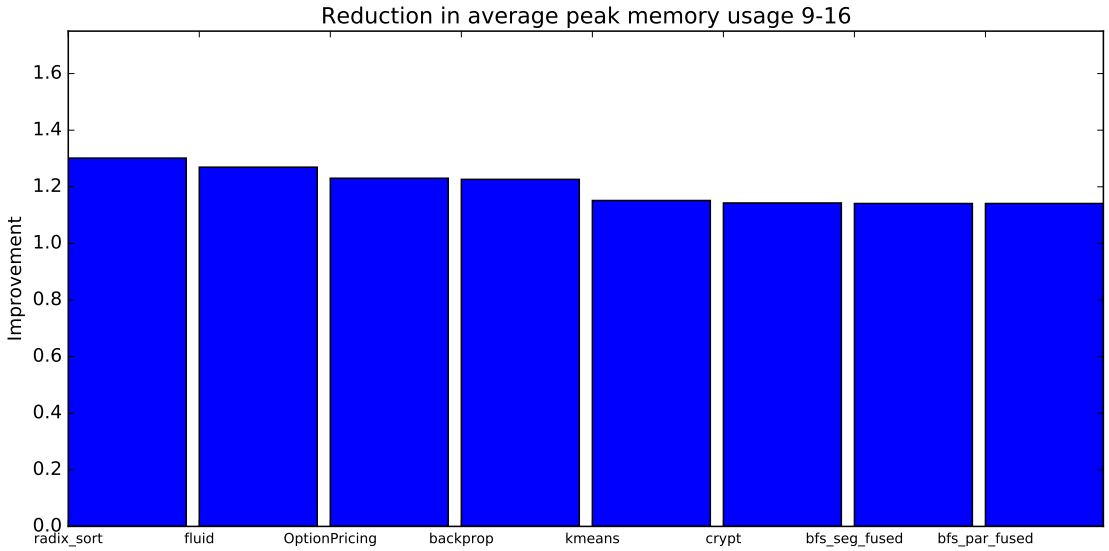


series	bfs_seg_fused	fluid	nbody	bfs_sgm_alt	cfd
x1.02	x1.01	x0.99	x0.98	x0.98	x0.98

Figure 10.12: Average runtime improvement in the CPU pipeline.

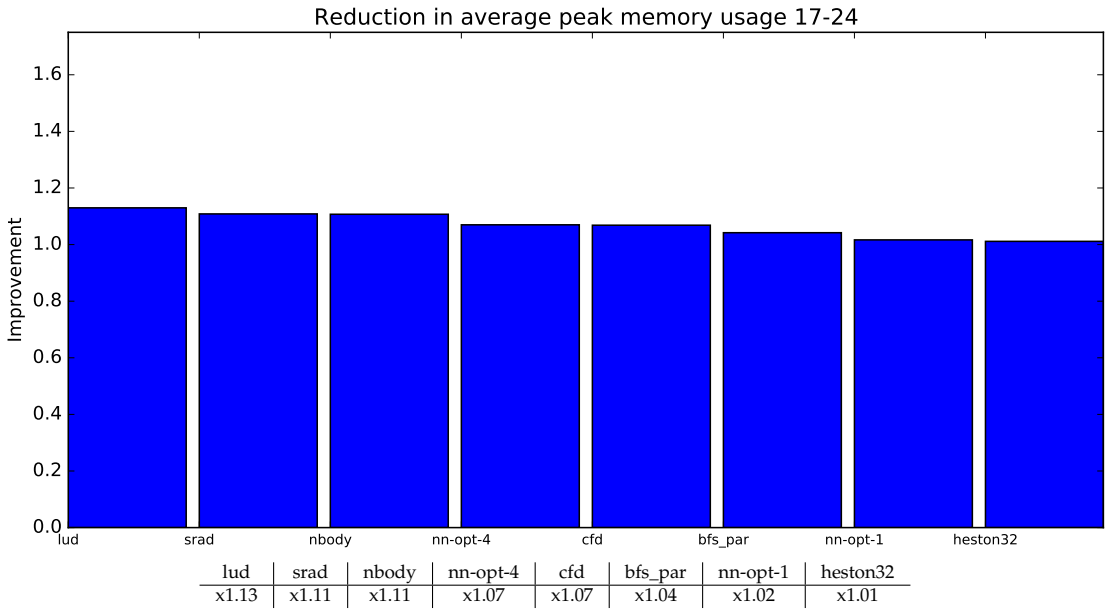


canny	series	myocyte	LocVolCalib	bfs_par_seg_alt	radix_sort_ blelloch	bfs_sgm_alt	bfs_par_seg
x1.70	x1.50	x1.49	x1.44	x1.42	x1.40	x1.38	x1.38



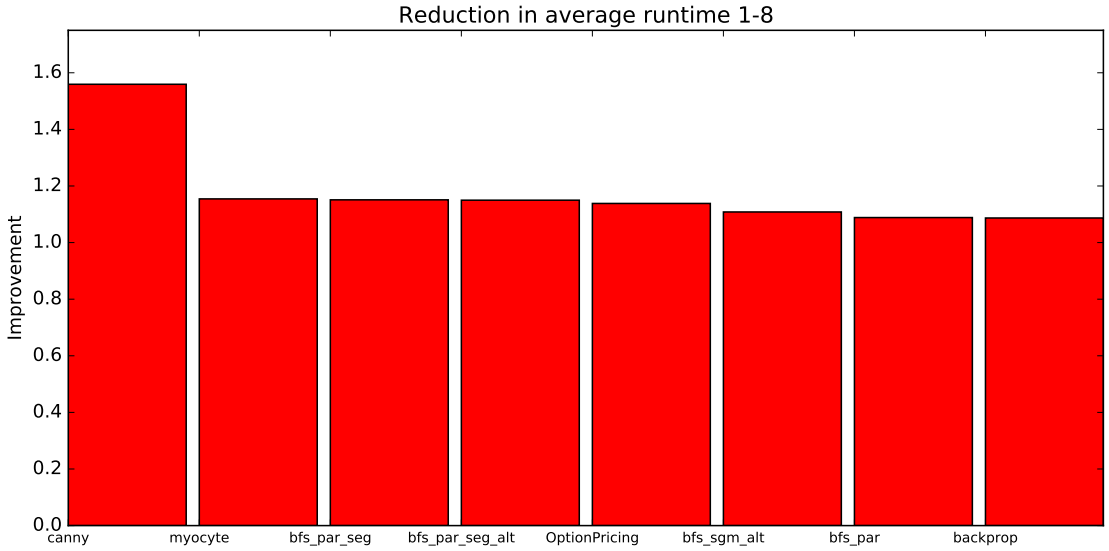
radix_sort	fluid	OptionPricing	backprop	kmeans	crypt	bfs_seg_fused	bfs_par_fused
x1.30	x1.27	x1.23	x1.23	x1.15	x1.14	x1.14	x1.14

Figure 10.13: Peak memory usage improvement in the GPU pipeline.

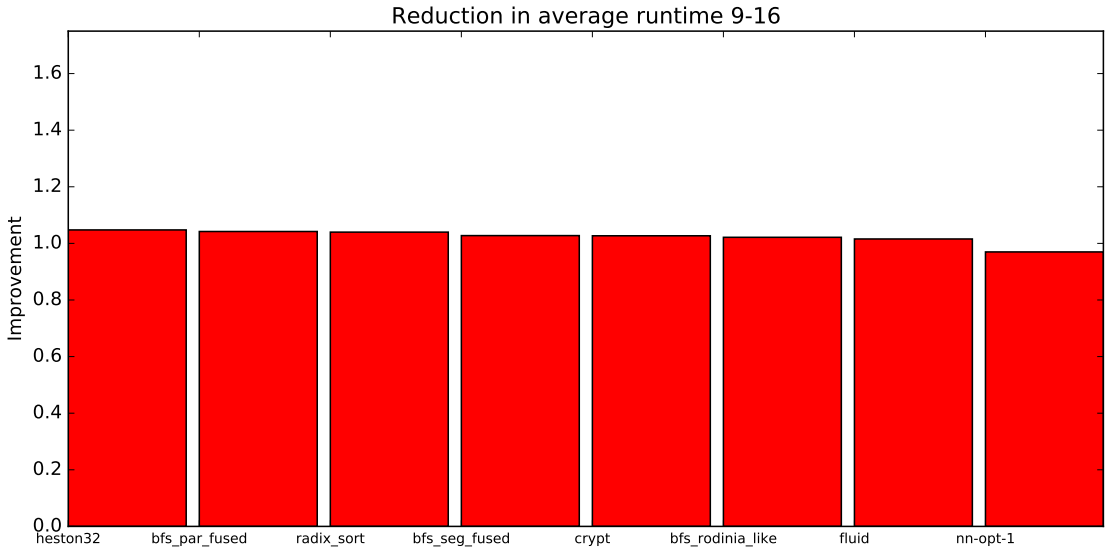


lud	srad	nbody	nn-opt-4	cfd	bfs_par	nn-opt-1	heston32
x1.13	x1.11	x1.11	x1.07	x1.07	x1.04	x1.02	x1.01

Figure 10.14: Peak memory usage improvement in the GPU pipeline, continued.

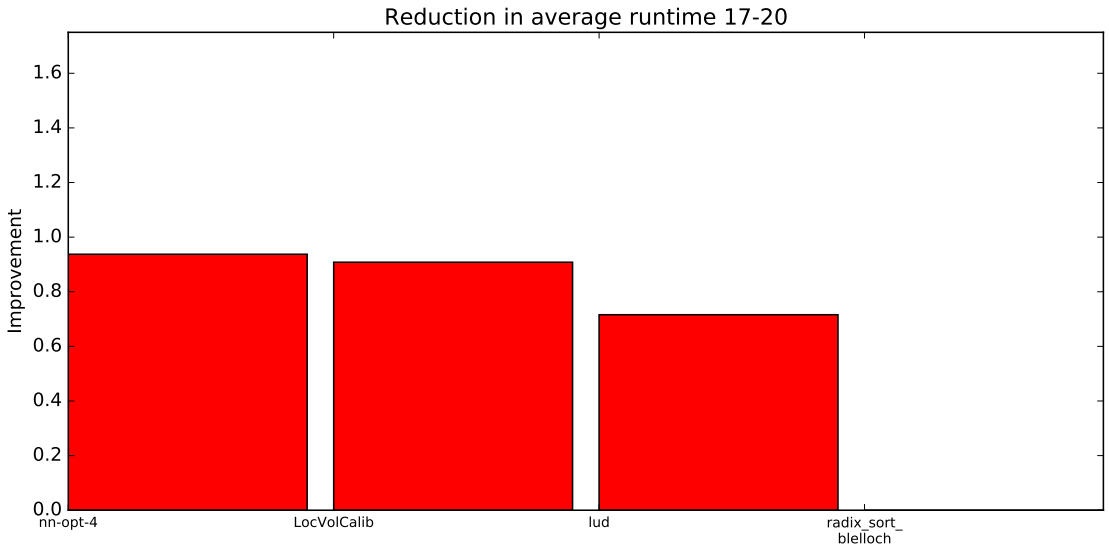


canny	myocyte	bfs_par_seg	bfs_par_seg_alt	OptionPricing	bfs_sgm_alt	bfs_par	backprop
x1.56	x1.15	x1.15	x1.15	x1.14	x1.11	x1.09	x1.09



heston32	bfs_par_fused	radix_sort	bfs_seg_fused	crypt	bfs_rodinia_like	fluid	nn-opt-1
x1.05	x1.04	x1.04	x1.03	x1.03	x1.02	x1.02	x0.97

Figure 10.15: Average runtime improvement in the GPU pipeline.



nn-opt-4	LocVolCalib	lud	radix_sort_ blelloch
x0.94	x0.91	x0.72	x-0.05

Figure 10.16: Average runtime improvement in the GPU pipeline, continued.

We can derive several conclusions from these measurements. Our main takeaways are that

- The peak memory usage **never** gets any worse in the **After** configuration compared to the **Before** configuration; there is always a non-negative reduction. This is true for both the average peak memory usage – as shown in the plots – and per-dataset peak memory usages – as shown in the appendix. The best improvement is in the `canny` benchmark, which gets a 70% reduction.
- The runtimes get better for some benchmarks, and worse for other benchmarks.
- The range in runtime improvements in the *CPU* pipeline are low, ranging from -5% to $+11\%$.
- The range in runtime improvements in the *GPU* pipeline are larger, ranging from -28% to $+15\%$.

We choose to ignore two measurements:

- We ignore the `canny` runtime results; the noted x1.56 average speedup in the GPU pipeline is based on a single, very small dataset. We would need much larger datasets to be able to state improvements with some certainty.
- We ignore the `radix_sort_blelloch` runtime results; for some reason our automatic benchmark runner tool measured its runtime in the **After** configuration as more than twice the length as that in **Before**. Upon rerunning it manually, we have found that there is very little variation.

We now take a closer look at *some of* the benchmarks with the largest improvements or largest regressions. We limit the number of picks in order to go into details with the ones that we do pick, and hope that our picks will exhibit optimisation patterns that also occur in other benchmarks with large improvements or regressions.

Failure: heston32

When compiled with the CPU pipeline and then run, the benchmark `heston32` (located in `misc/heston/` in the Futhark benchmarks repository) gives a segmentation fault. This happens only when memory block reuse is enabled; it validates with memory block coalescing enabled. Everything validates with the GPU pipeline.

There are two variants of the Heston benchmark: `heston64`, which uses 64-bit floats, and `heston32`, which uses 32-bit floats. The 64-bit version validates in all configurations, so it is interesting that the 32-bit version fails; other than different float precisions, they are identical (by utilising a module system that is part of the Futhark source language).

We discovered this very late in the thesis process, and have not been able to fix it.

Regression: LocVolCalib

The LocVolCalib benchmark (located in `finpar/` in the Futhark benchmarks repository) has a x1.44 reduction in average peak memory usage in the GPU pipeline, but a x1.10 increase (x0.91 reduction) in average runtime; the **After** configuration is almost 300 milliseconds slower in the largest dataset.

We have tried to find the cause of this regression by running the benchmark in different configurations. The runtime is very close to that of the **Before** configuration in all configurations where memory reuse is *disabled*, so it appears that *enabling* the memory reuse algorithm is the root cause.

To further narrow it down, we have tried disabling every memory reuse extension, keeping only the conservative "the memory blocks must not interfere, and the size variables must be exactly the same" requirements. This causes less of a slowdown, but only about half; the largest dataset is now about 150 milliseconds slower.

We are at a loss as to why this regression happens. The structure of the program is fairly simple: First a sequence of allocations (which becomes shorter after the optimisations) and map kernels, and then a loop with map kernels.

Regression: LUD

The LUD benchmark (located in `rodinia/lud/` in the Futhark benchmarks repository) has a x1.13 reduction in average peak memory usage in the GPU pipeline, but a x1.39 increase (x0.72 reduction) in average runtime. We note that this number is averaged over all datasets

measurements, and that in the largest dataset – likely the one with the least noise effect – it is "just" a x1.11 increase.

Running the benchmark in different configurations, we see that having in-place lowering *disabled* always results in a runtime increase very close to that of the **After** configuration; it seems very likely that in-place lowering manages to remove data copying statements – and thus become faster – in places where our memory block optimisations do not.

We have found that in-place lowering lowers exactly one memory copying: The generated EXPLICITMEMORY code has a statement

```
let dst[i] = copy res
```

which in-place lowering optimises away by lowering the in-place update into the expression of `res`. The copying only copies 256 bytes, but happens in a loop, so it does so for many iterations, which explains why it adds up to a noticable increase in runtime.

We would expect our memory block coalescing algorithm to also handle this case, but our debugging tells us that safety condition 3 is not fulfilled, i.e. the memory of `dst` is used between the definition of `res` and the `dst[i]` statement.

We have manually verified that this is indeed the case: the expression of `res` is a map kernel whose body contains several uses of the memory of `dst`, meaning we consider the memory to be used *between* `res` and `dst[i]`. However, the memory block is not used anywhere *outside* the `res` expression (ignoring the final `dst[i]` use), so it is a limited case of violating safety condition 3.

There is a reason why we consider a memory block m to be used *between* two statements x and y even if m is only used inside the expression of the x statement, and not in a *separate statement* between x and y: The expression of x might read from one index in m but write to another, either in a kernel or in a loop, resulting in the overwriting of data that was supposed to be read later on.

We think this can be solved by using the index access analysis described in section 6.4 for potential coalescings of this pattern as well. This way, we would ignore safety condition 3 if the expression of x reads from and writes to the same index in each thread or iteration. The `lud` benchmark exhibits this behaviour in the expression of `res`, so we think it is possible to remove its efficiency dependency on in-place lowering this way.

Improvements: Myocyte and Option Pricing

The `myocyte` benchmark (located in `rodinia/myocyte/` in the Futhark benchmarks repository) has a x1.49 reduction in average peak memory usage in the GPU pipeline, and a x1.15 reduction in average runtime; the **After** configuration is about 140 milliseconds faster in the largest dataset. For the Option Pricing benchmark we get a similar x1.14 average GPU runtime decrease.

Running the two benchmarks in different configurations, we see that having memory block coalescing *disabled* always results in a runtime increase very close to that of the respective **Before** configurations; it seems very likely that memory block coalescing manages to remove data copying statements – and thus become faster – in places where in-place lowering cannot.

We have counted 4 memory block coalescings in `myocyte`, and 3 in Option Pricing. We have also compared each generated program with and without in-place lowering, and we can see that they

are identical, meaning our memory block coalescing algorithm handles cases where in-place lowering gives up – as we want it to.

But why do these two benchmarks have better runtime reductions than the other benchmarks? They do not have *that* many coalescings compared to the large sizes of the programs. We think it is because of the locations of the coalescings: In both benchmarks, several of the coalescing-enabling `copy` statements are located inside loops or kernels, meaning they are potentially executed *many* times, which means that our optimisation reduces data copying *many* times.

Note that the runtime reductions in the *CPU* pipeline are less impressive – x1.01 and x1.02, respectively. We have counted 2 memory blocks coalescings in each benchmark, which is less than in the GPU pipeline for both benchmarks, and might explain at least *part* of this lack of speedup.

Chapter 11

Related Work

Mechanisms for supporting memory abstraction free the programmer from worrying about the low-level hardware details, such as the memory hierarchy and efficient access patterns. This section surveys several classes of approaches and compares them with the solutions proposed in this thesis.

Reference-tracing garbage collection (GC) is perhaps the most popular mechanism for automatic memory management; for example, it is used by many mainstream languages, such as Java, C#, F#, Haskell, SML, and by computer algebra systems, such as Maple and Mathematica. A rich body of literature has explored techniques aimed at reducing the (non-negligible) runtime overhead of garbace collection: this is e.g. achieved by allowing GC to proceed concurrently with the execution of the program [4], or by parallelizing "stop-of-the-world" (generational-copying) collectors [18, 21].

One problem, however, is that GC solutions seem (fundamentally) unsuitable for programs executing on heterogeneous hardware such as GPUs, because, for example, global-memory (de)allocation is not permitted from inside GPU kernels. A second problem is that interoperability solutions [3, 22] between GC'ed languages are often complicated by the need to provide protocols whereby the GCs of the two systems can cooperate when structures span the two system heaps [29].

In comparison, our approach allows us to *statically* optimize – i.e. with no runtime overheads – the memory reuse and copying overheads, even between arrays used inside the same (or across multiple) GPU kernel(s), and does not hinder interoperability solutions. For example, the current infrastructure that enables interoperability [5, 9] between APL, Python and Futhark would still work without any modification. We note that the uniqueness-type mechanism provides the means for transferring the control of memory from the host language to Futhark: for example, if an array is passed as unique, then it is guaranteed that it cannot be referenced from the host language after the foreign function call, meaning its memory can be safely reused within the Futhark program.

Region-based memory management [28] refers to a class of techniques, initially developed in the context of the SML language, that are aimed at promoting an optimized stack-based memory management policy. In essence, the analysis statically infers a grouping of "related" objects

(e.g. recursive datatypes) that are to be allocated into the same memory region. Regions are then inserted syntactically in the program as **`let`** regions, which allows bulk deallocation of the objects in a region when the region goes out of scope. The approach has the nice property that it can be formalized within the type theory domain [8]: instantiating the technique over (all) valid type-unification strategies will result in all possible memory-correct programs.

Note that region-based techniques are not as simple as we make them sound: for example, regions are still allocated on the heap because their size might grow or shrink dynamically in ways that cannot be statically (even conservatively) approximated.

More importantly, please note that not all programs are suitable for region management, because a stack (de)allocation policy assumes a nested lifetime of objects, which does not hold for all programs. It follows that practical solutions implement memory policies that integrate region-based memory management with automatic-heap (GC) memory management [7].

More recently, region-based allocation has been proposed for managing pointer-heavy data structures in the Harlan programming language [15], which is aimed at GPU execution.

Similarly to garbage collection, region-based memory management does not optimize the copying overhead and is fundamentally unsuitable for executing programs on heterogeneous hardware because it essentially models the object lifetime to be the interval between the allocation and deallocation points – and this granularity is too coarse-grained for GPU systems, where (de)allocations are not supported from inside GPU kernels.

Futhark's approach to memory draws inspiration from region-based memory management in the manner in which explicit memory blocks and allocations are introduced into memory-agnostic programs, but diverges by aggressively hoisting allocations (hopefully all the way to the CPU space), and by separating the notions of where the memory is allocated from where its first use is (the array creation point). This results in a very different and seemingly more complex analysis, which unfortunately we have not been able to formalize yet with inference rules.

Fusion. Existing approaches to minimising the copying overhead and the memory footprint of (functional) array programming languages are primarily based on *fusion* [6, 19], but this has limited applicability for optimising memory usage inside the sequential code of parallel regions, or operations that are not fusable in the typical sense, such as concatenation.

The delayed array representations used in Repa [16] and Obsidian [27] are able to handle concatenation without manifestation, but at the cost of an unpredictable degree of redundant computation and code explosion.

In this sense, Futhark supports in its *index functions* a restricted version of fusion and delayed arrays, which are guaranteed not to duplicate computation [13].

Register allocation maps the arbitrary number of scalar variables in a program to a finite set of hardware registers. This requires computing the liveness intervals of scalar variables, and, based on this, building an interference graph, i.e. a representation which variable pairs cannot be mapped to the same registers.

Finally, the mapping between variables and registers is obtained by an optimal or a heuristic graph-coloring algorithm.

For a long time it was believed that optimal register allocation requires solving a NP-complete problem. However, Bouchez et al. has shown that register allocation through optimal graph

coloring is actually possible in polynomial time under the assumption that bindings in the program are in SSA form[2]. (However, this holds only if we do not consider optimal register spilling and coalescing.)

Our approach to memory-block merging has drawn inspiration from the linear-scan register allocation [23] proposed by Poletto and Sarkar, which has quickly become the tool of choice in just-in-time compilers because its simplicity translates directly to fast compilation time. The method was later extended by Sarkar and Barik with new heuristics, and was shown to achieve code quality competitive with colouring-based algorithms [25], while still allowing fast compilation.

Our work has aimed for lifting the register allocation and coalescing techniques to operate on arrays rather than scalars. Since the intermediate representation of Futhark is essentially an SSA form, an interesting future research direction would be to extend our analysis to support optimal graph coloring, for example across memory blocks allocated in the same scope.

Futhark. Previous work on Futhark includes fusion[13]; an internal construct for expressing certain kinds of parallism called `redomap`[11]; the development of a new source language construct `scatter`[26][1]; the development of a segmented version of `redomap`[17]; and more. A core design principle of the Futhark language is to only support the most necessary constructs for expressing computations, and in particular avoid exposing constructs that are hard to parallelise, unless they are very commonly needed in real-world programs. Futhark is a sum of compromises, e.g.

- Futhark has sequential loops, since not *everything* can be expressed in terms of parallel operations.
- Futhark supports nested parallelism, but requires all arrays to be regular.
- Futhark is a pure language that supports in-place updates.

A stated goal of the language is for it to be used mainly for performance-critical computation sections of larger programs written in other languages. Futhark already works well in integration with Python, and more foreign function interfaces are being developed.

[1] Called `write` at the time.

Chapter 12

Conclusion and Future Work

We have presented two optimisations in the EXPLICITMEMORY intermediate representation of the Futhark programming language: Memory block coalescing and memory block reuse.

With the aid of enabling analyses and enabling transformations, these optimisations both perform *memory block merging*, which we have shown to be effective in reducing *peak memory usage* across several pre-existing Futhark benchmarks, and never increase it, ranging from 0% to 70% in average reductions on the GPU.

Our optimisations introduce one discovered-but-not-fixed bug in the benchmark program of `heston32`, and several clear regressions in benchmark runtimes; average runtime reductions range from -28% (a 39% *slowdown*) to +16%.

We have validated our optimisations on 95 memory block-specific small test programs, manually annotated (crudely) with the expected structure after the optimisations, and automatically testable with a tool. We have also validated our optimisations on pre-existing general Futhark test programs (and on the Futhark benchmarks).

Future Work

As described in detail in chapter 9, both the current algorithms and the current implementation are limited in several ways. To reiterate:

- Our memory block coalescing optimisation has only limited support for coalescing through delayed arrays, such as those created by **`reshape`** statements. We plainly need to describe and implement this.
- We have found at least one case where we could get an extra memory block merging if index functions were part of the merge parameters ("accumulators") of loops. We need to think about whether this is a good idea in general, and how to represent it internally.
- We need to extend our index access analysis to also work across loop iterations.

- We need to find an elegant, maintainable way of not generating copying statements in the code generator in the case of a **let** `dst` = **concat** ... `src` ... statement where `dst` and `src` share the same memory block. This is not a hard theoretical problem, but more of a minor software design problem, and is likely easy to solve.
- Our implementation is slower than it needs to be; we make several independent passes through a program instead of having only a few (and likely not in a way that the Haskell compiler can optimise), and we have a top-down worst-case quadratic time coalescing algorithm that we suspect could be handled in linear time by a bottom-up version. However, we note that it is *very important* that this efficiency tirade does not take precedence over the readability of the implementation – compilers are complex enough just by their nature.
- The linear scan algorithm that we are using for our memory reuse algorithm does not always produce optimal results. We would like to try out other register allocation-based approaches with different heuristics, and see if some variations are better both in terms of the number of memory block mergings and speed.
- The representation of our last use analysis does not have fine-grained support for **if** expressions, limiting memory block mergings in some cases. We think it is doable to implement this support, but are not sure about the details.
- Loops that have been first-order transformed from kernels are harder to analyse than the original kernels. We mostly handle this in the implementation, but not in all cases. It might be a good idea to think of alternative internal representations.
- It can be hard to write Futhark programs in the source language that end up in their EXPLICITMEMORY form as one expects. In the far future, when the language has stabilised, it might be beneficial to support writing programs directly in intermediate representations instead of exclusively writing in the source language.

Additionally, we note several limitations in our approaches throughout this thesis:

- Our formalization attempt of memory block coalescing is very incomplete. We need more time to extend it to describe all coalescing cases, and to try formalizing the memory block reuse optimisation as well.
- We have compared peak memory usage and runtime measurements only internally between Futhark benchmarks run with different configurations. For better reflection we also need to compare these measurements with those of the original benchmarks in other languages (CUDA, OpenCL, Haskell's Accelerate, etc.) from which the Futhark benchmarks are ported.

There are seemingly always more program patterns to handle when doing compiler optimisations; most of our choices boil down to deciding which patterns appear to be most prominent; then describing and implementing transformations for them; and finally repeating the process.

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] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the np-completeness proof of chaitin et al. really prove? or revisiting register allocation: Why and how. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, pages 283–298, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. Parametric Polymorphism for Computer Algebra Software Components. In *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Comput.*, SYNASC '04, pages 119–130. Mirton Publishing House, 2004.

[4] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 113–123, New York, NY, USA, 1993. ACM.

[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] Clemens Grelck and Sven-Bodo Scholz. SAC - A Functional Array Language for Efficient Multi-Threaded Execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[7] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *ACM Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press, June 2002. Berlin, Germany.

[8] Fritz Henglein, Henning Makholm, and Henning Niss. *Advanced Topics in Types and Programming Languages*, chapter Effect type systems and region-based memory management. MIT Press, Cambridge, Mass., 2005.

[9] 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.

[10] 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.

[11] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. Design and gpgpu performance of futhark's redomap construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 17–24, New York, NY, USA, 2016. ACM.

[12] 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.

[13] 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.

[14] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[15] Eric Holk, Ryan Newton, Jeremy Siek, and Andrew Lumsdaine. Region-based memory management for gpu programming languages: Enabling rich data structures on a spartan host. *SIGPLAN Not.*, 49(10):141–155, October 2014.

[16] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.

[17] Rasmus Wriedt Larsen and Troels Henriksen. Strategies for regular segmented reductions on gpu. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, pages 42–52, New York, NY, USA, 2017. ACM.

[18] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM.

[19] Trevor L. McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs. In *Procs. of Int. Conf. Funct. Prog. (ICFP)*, 2013.

[20] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. Financial software on gpus: Between haskell and fortran. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '12, pages 61–72, New York, NY, USA, 2012. ACM.

[21] Cosmin E. Oancea, Alan Mycroft, and Stephen M. Watt. A new approach to parallelising tracing algorithms. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 10–19, New York, NY, USA, 2009. ACM.

[22] Cosmin E. Oancea and Stephen M. Watt. Domains and Expressions: An Interface Between Two Approaches to Computer Algebra. In *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*, ISSAC '05, pages 261–268, New York, NY, USA, 2005. ACM.

[23] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, September 1999.

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), pp. 44-54, Oct. 2009.

[25] Vivek Sarkar and Rajkishore Barik. *Extended Linear Scan: An Alternate Foundation for Global Register Allocation*, pages 141–155. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[26] Niels G. W. Serup. Extending Futhark with a write construct. `http://hiperfit.dk/pdf/niels-thesis.pdf`, 2016.

[27] Joel Svensson, Mary Sheeran, and Koen Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*, IFL'08, pages 156–173, Berlin, Heidelberg, 2011. Springer-Verlag.

[28] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher Order Symbolic Computation*, 17(3):245–265, 2004.

[29] S. M. Watt. A study in the integration of computer algebra systems: Memory management in a maple-aldor environment. In *Proc. International Congress of Mathematical Software*, pages 405–411, 2002.

Appendix A

Implementation

This chapter briefly describes the basic implementation details of the analyses and transformations shown in chapter 4.

The home of Futhark implementation is GitHub:
`https://github.com/diku-dk/futhark/commits/master`

Our (not very well documented) benchmarking tools are located in the `niels-memory-block-merging-benchmarking` *git branch* on GitHub; see
`https://github.com/diku-dk/futhark/tree/niels-memory-block-merging-benchmarking`

Futhark is under constant development. This thesis is based on the `d8b078cd5e2267c59ffbd8407c377418984bcf37` commit from October 4; see
`https://github.com/diku-dk/futhark/tree/d8b078cd5e2267c59ffbd8407c377418984bcf37`

Our implementation is located in the `src/Futhark/Optimise/MemoryBlockMerging/` directory.

The directory `memory-block-merging` contains a `README.md` file that describes how to enable and disable memory block merging optimisations, and a `tests` directory with all memory block merging-specific test programs.

Appendix B

Measurements

The final memory measurements are from 8 October 2017. The runtime measurements are from 10 October 2017.

All measurements are based on the `https://github.com/diku-dk/futhark/tree/d8b078cd5e2267c59ffbd8407c377418984bcf37` commit from the master branch and the `https://github.com/diku-dk/futhark-benchmarks/tree/24cba07b5b70d3cbcf4195108a4caddbb649481f` commit from the `futhark-benchmarks` repository.

canny

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 66.141%

lena256.in	lena512.in
Before: 2380.496 kB	**Before**: 9502.276 kB
After: 807.632 kB	**After**: 3210.820 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 69.561%

lena512.in
Before: 10551.876 kB
After: 3211.844 kB

Runtime with the CPU pipeline:

Average runtime reduction: 8.522%

lena256.in	lena512.in
Before: 5.006 ms	**Before**: 20.738 ms
After: 4.679 ms	**After**: 18.559 ms

Runtime with the GPU pipeline:

Average runtime reduction: 55.936%

lena512.in
Before: 2.450 ms
After: 1.079 ms

crystal

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

Dataset #0	Dataset #1	Dataset #2	Dataset #3	Dataset #4	Dataset #5
Before: 160.000 kB **After**: 160.000 kB	**Before**: 80.000 kB **After**: 80.000 kB	**Before**: 320.000 kB **After**: 320.000 kB	**Before**: 320.000 kB **After**: 320.000 kB	**Before**: 16000.000 kB **After**: 16000.000 kB	**Before**: 64000.000 kB **After**: 64000.000 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.000%

Dataset #4	Dataset #5
Before: 16000.000 kB **After**: 16000.000 kB	**Before**: 64000.000 kB **After**: 64000.000 kB

fluid

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 26.923%

medium.in
Before: 1082.016 kB **After**: 790.704 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 26.923%

medium.in
Before: 1082.016 kB **After**: 790.704 kB

Runtime with the CPU pipeline:

Average runtime reduction: -1.255%

medium.in
Before: 19.252 ms **After**: 19.494 ms

Runtime with the GPU pipeline:

Average runtime reduction: 1.559%

medium.in
Before: 2.733 ms **After**: 2.691 ms

mandelbrot

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

Dataset #0	Dataset #1	Dataset #2	Dataset #3	Dataset #4
Before: 1920.000 kB **After**: 1920.000 kB	**Before**: 4000.000 kB **After**: 4000.000 kB	**Before**: 16000.000 kB **After**: 16000.000 kB	**Before**: 64000.000 kB **After**: 64000.000 kB	**Before**: 256000.000 kB **After**: 256000.000 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.000%

Dataset #1	Dataset #2	Dataset #3	Dataset #4
Before: 4000.000 kB **After**: 4000.000 kB	**Before**: 16000.000 kB **After**: 16000.000 kB	**Before**: 64000.000 kB **After**: 64000.000 kB	**Before**: 256000.000 kB **After**: 256000.000 kB

nbody

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 10.714%

1000-bodies.in	10000-bodies.in	100000-bodies.in
Before: 112.000 kB **After**: 100.000 kB	**Before**: 1120.000 kB **After**: 1000.000 kB	**Before**: 11200.000 kB **After**: 10000.000 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 10.714%

10000-bodies.in	100000-bodies.in
Before: 1120.000 kB **After**: 1000.000 kB	**Before**: 11200.000 kB **After**: 10000.000 kB

Runtime with the CPU pipeline:

Average runtime reduction: -1.630%

1000-bodies.in	10000-bodies.in	100000-bodies.in
Before: 5.466 ms **After**: 5.278 ms	**Before**: 514.613 ms **After**: 550.853 ms	**Before**: 51487.686 ms **After**: 52148.345 ms

Runtime with the GPU pipeline:

Average runtime reduction: 0.567%

10000-bodies.in	100000-bodies.in
Before: 3.314 ms **After**: 3.285 ms	**Before**: 290.478 ms **After**: 289.680 ms

tunnel

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

Dataset #0	Dataset #1	Dataset #2	Dataset #3	Dataset #4
Before: 1922.400 kB **After**: 1922.400 kB	**Before**: 4004.000 kB **After**: 4004.000 kB	**Before**: 16008.000 kB **After**: 16008.000 kB	**Before**: 64016.000 kB **After**: 64016.000 kB	**Before**: 256032.000 kB **After**: 256032.000 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.000%

Dataset #0	Dataset #1	Dataset #2	Dataset #3	Dataset #4
Before: 1920.000 kB **After**: 1920.000 kB	**Before**: 4000.000 kB **After**: 4000.000 kB	**Before**: 16000.000 kB **After**: 16000.000 kB	**Before**: 64000.000 kB **After**: 64000.000 kB	**Before**: 256000.000 kB **After**: 256000.000 kB

LocVolCalib

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.911%

large.in	medium.in	small.in
Before: 1066.960 kB **After**: 1063.888 kB	**Before**: 141.776 kB **After**: 139.600 kB	**Before**: 140.432 kB **After**: 139.152 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 44.017%

large.in	medium.in	small.in
Before: 1677997.008 kB **After**: 939536.336 kB	**Before**: 104998.736 kB **After**: 58729.808 kB	**Before**: 13118.864 kB **After**: 7349.584 kB

Runtime with the CPU pipeline:

Average runtime reduction: 0.132%

large.in	medium.in	small.in
Before: 99765.564 ms **After**: 99733.178 ms	**Before**: 6045.985 ms **After**: 6006.034 ms	**Before**: 3071.849 ms **After**: 3080.939 ms

Runtime with the GPU pipeline:

Average runtime reduction: -9.187%

large.in	medium.in	small.in
Before: 4243.993 ms **After**: 4540.246 ms	**Before**: 287.813 ms **After**: 293.309 ms	**Before**: 205.179 ms **After**: 243.488 ms

OptionPricing

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 5.637%

large.in	medium.in	small.in
Before: 187.992 kB **After**: 182.104 kB	**Before**: 2.660 kB **After**: 2.564 kB	**Before**: 0.236 kB **After**: 0.212 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 23.017%

large.in	medium.in	small.in
Before: 1299602.544 kB **After**: 1058823.268 kB	**Before**: 18352.788 kB **After**: 14813.832 kB	**Before**: 2097.860 kB **After**: 1442.488 kB

Runtime with the CPU pipeline:

Average runtime reduction: 1.982%

large.in	medium.in	small.in
Before: 6888.391 ms **After**: 6825.220 ms	**Before**: 884.256 ms **After**: 859.534 ms	**Before**: 741.809 ms **After**: 725.244 ms

Runtime with the GPU pipeline:

Average runtime reduction: 13.828%

large.in	medium.in	small.in
Before: 313.847 ms **After**: 267.221 ms	**Before**: 20.381 ms **After**: 16.455 ms	**Before**: 9.153 ms **After**: 8.479 ms

crypt

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

medium.in
Before: 3000.208 kB **After**: 3000.208 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 14.279%

medium.in
Before: 7003.152 kB
After: 6003.152 kB

Runtime with the GPU pipeline:

Average runtime reduction: 2.685%

medium.in
Before: 2.022 ms
After: 1.968 ms

keys

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.000%

userkey0.txt
Before: 0.224 kB
After: 0.224 kB

series

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 50.000%

10000.in	100000.in	1000000.in
Before: 320.000 kB	**Before**: 3200.000 kB	**Before**: 32000.000 kB
After: 160.000 kB	**After**: 1600.000 kB	**After**: 16000.000 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 50.000%

10000.in	100000.in	1000000.in
Before: 320.000 kB	**Before**: 3200.000 kB	**Before**: 32000.000 kB
After: 160.000 kB	**After**: 1600.000 kB	**After**: 16000.000 kB

Runtime with the CPU pipeline:

Average runtime reduction: 1.565%

10000.in	100000.in	1000000.in
Before: 1708.053 ms	**Before**: 17307.590 ms	**Before**: 167936.157 ms
After: 1678.099 ms	**After**: 16794.450 ms	**After**: 167973.308 ms

Runtime with the GPU pipeline:

Average runtime reduction: 0.087%

10000.in	100000.in	1000000.in
Before: 124.807 ms	**Before**: 1221.331 ms	**Before**: 12130.306 ms
After: 124.542 ms	**After**: 1221.828 ms	**After**: 12119.401 ms

heston32

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 1.141%

100000_quotes.in	10000_quotes.in	1062_quotes.in
Before: 754804.867 kB	**Before**: 69504.164 kB	**Before**: 7688.954 kB
After: 752185.655 kB	**After**: 69109.132 kB	**After**: 7496.046 kB

Runtime with the GPU pipeline:

Average runtime reduction: 4.755%

100000_quotes.in	10000_quotes.in	1062_quotes.in
Before: 2182.923 ms	**Before**: 235.183 ms	**Before**: 41.113 ms
After: 2197.321 ms	**After**: 208.166 ms	**After**: 39.700 ms

heston64

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 9.029%

100000_quotes.in	10000_quotes.in	1062_quotes.in
Before: 64263.791 kB	**Before**: 5906.420 kB	**Before**: 636.958 kB
After: 58579.815 kB	**After**: 5381.100 kB	**After**: 577.410 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.973%

100000_quotes.in	10000_quotes.in	1062_quotes.in
Before: 1503390.023 kB	**Before**: 138435.744 kB	**Before**: 15315.682 kB
After: 1500773.819 kB	**After**: 137887.080 kB	**After**: 14955.862 kB

Runtime with the CPU pipeline:

Average runtime reduction: 0.135%

100000_quotes.in	10000_quotes.in	1062_quotes.in
Before: 268449.793 ms	**Before**: 24602.883 ms	**Before**: 3490.106 ms
After: 268624.746 ms	**After**: 24593.673 ms	**After**: 3474.956 ms

Runtime with the GPU pipeline:

Average runtime reduction: 0.787%

100000_quotes.in	10000_quotes.in	1062_quotes.in
Before: 22915.282 ms	**Before**: 2149.019 ms	**Before**: 300.731 ms
After: 22913.375 ms	**After**: 2144.552 ms	**After**: 294.276 ms

radix_sort

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 30.126%

Dataset #0	radix_sort_100.in
Before: 0.344 kB	**Before**: 4.024 kB
After: 0.240 kB	**After**: 2.816 kB

Runtime with the GPU pipeline:

Average runtime reduction: 3.977%

Dataset #0	radix_sort_100.in
Before: 1.946 ms	**Before**: 1.880 ms
After: 1.837 ms	**After**: 1.836 ms

radix_sort_blelloch

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 25.000%

radix_sort_100K.in	radix_sort_10K.in	radix_sort_1M.in
Before: 3200.000 kB	**Before**: 320.000 kB	**Before**: 32000.000 kB
After: 2400.000 kB	**After**: 240.000 kB	**After**: 24000.000 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 39.971%

radix_sort_100K.in	radix_sort_10K.in	radix_sort_1M.in
Before: 4002.048 kB	**Before**: 400.640 kB	**Before**: 40002.048 kB
After: 2402.048 kB	**After**: 240.640 kB	**After**: 24002.048 kB

Runtime with the CPU pipeline:

Average runtime reduction: 3.000%

radix_sort_100K.in	radix_sort_10K.in	radix_sort_1M.in
Before: 23.891 ms	**Before**: 2.506 ms	**Before**: 288.671 ms
After: 23.851 ms	**After**: 2.343 ms	**After**: 281.985 ms

Runtime with the GPU pipeline:

Average runtime reduction: -105.232%

radix_sort_100K.in	radix_sort_10K.in	radix_sort_1M.in
Before: 5.977 ms	**Before**: 1.628 ms	**Before**: 48.807 ms
After: 7.722 ms	**After**: 6.205 ms	**After**: 51.370 ms

mri-q

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

large.in	small.in
Before: 5292.032 kB	**Before**: 729.088 kB
After: 5292.032 kB	**After**: 729.088 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.000%

large.in	small.in
Before: 5292.032 kB	**Before**: 729.088 kB
After: 5292.032 kB	**After**: 729.088 kB

sgemm

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

medium.in	small.in
Before: 16904.192 kB	**Before**: 274.432 kB
After: 16904.192 kB	**After**: 274.432 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.000%

medium.in
Before: 16904.192 kB **After**: 16904.192 kB

stencil

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

default.in	small.in
Before: 201457.664 kB **After**: 201457.664 kB	**Before**: 6307.840 kB **After**: 6307.840 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.000%

default.in	small.in
Before: 201457.664 kB **After**: 201457.664 kB	**Before**: 6307.840 kB **After**: 6307.840 kB

tpacf

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

large.in	medium.in	small.in
Before: 20052.208 kB **After**: 20052.208 kB	**Before**: 7915.448 kB **After**: 7915.448 kB	**Before**: 957.296 kB **After**: 957.296 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.240%

large.in	medium.in	small.in
Before: 501112.216 kB **After**: 500162.608 kB	**Before**: 197582.200 kB **After**: 197186.552 kB	**Before**: 23624.248 kB **After**: 23546.192 kB

Runtime with the GPU pipeline:

Average runtime reduction: -0.305%

large.in	medium.in	small.in
Before: 8605.008 ms **After**: 8628.811 ms	**Before**: 1292.830 ms **After**: 1299.625 ms	**Before**: 29.360 ms **After**: 29.394 ms

backprop

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 32.000%

medium.in
Before: 209715.976 kB **After**: 142606.984 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 22.619%

medium.in
Before: 352324.396 kB **After**: 272632.408 kB

Runtime with the CPU pipeline:

Average runtime reduction: 2.849%

medium.in
Before: 1357.597 ms
After: 1318.917 ms

Runtime with the GPU pipeline:

Average runtime reduction: 8.688%

medium.in	small.in
Before: 32.173 ms	**Before**: 2.264 ms
After: 31.593 ms	**After**: 1.911 ms

bfs_seg_fused

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 4.800%

512nodes_high_edge_variance.in	graph1MW_6.in
Before: 834.807 kB	**Before**: 53803.283 kB
After: 832.246 kB	**After**: 48803.282 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 14.088%

4096nodes.in	graph1MW_6.in
Before: 262.078 kB	**Before**: 63899.670 kB
After: 225.149 kB	**After**: 54899.157 kB

Runtime with the CPU pipeline:

Average runtime reduction: 1.453%

512nodes_high_edge_variance.in	graph1MW_6.in
Before: 1.991 ms	**Before**: 259.645 ms
After: 1.945 ms	**After**: 258.137 ms

Runtime with the GPU pipeline:

Average runtime reduction: 2.751%

4096nodes.in	graph1MW_6.in
Before: 1.397 ms	**Before**: 28.146 ms
After: 1.341 ms	**After**: 27.722 ms

bfs_par_mapwrite

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 794.624 kB	**Before**: 2622.464 kB	**Before**: 185999.880 kB
After: 794.624 kB	**After**: 2622.464 kB	**After**: 185999.880 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.031%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 811.592 kB	**Before**: 2625.040 kB	**Before**: 190000.912 kB
After: 811.012 kB	**After**: 2624.516 kB	**After**: 189999.884 kB

Runtime with the GPU pipeline:

Average runtime reduction: 0.545%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 4.163 ms	**Before**: 3.502 ms	**Before**: 108.175 ms
After: 4.036 ms	**After**: 3.393 ms	**After**: 113.080 ms

bfs_par_seg

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 17.331%

512nodes_high_edge_variance.in	graph1MW_6.in
Before: 2998.873 kB	**Before**: 110379.783 kB
After: 2486.725 kB	**After**: 90970.091 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 38.116%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 619.022 kB	**Before**: 4531.001 kB	**Before**: 154417.891 kB
After: 387.056 kB	**After**: 2742.055 kB	**After**: 96677.753 kB

Runtime with the CPU pipeline:

Average runtime reduction: 8.549%

512nodes_high_edge_variance.in	graph1MW_6.in
Before: 1.630 ms	**Before**: 192.014 ms
After: 1.470 ms	**After**: 178.052 ms

Runtime with the GPU pipeline:

Average runtime reduction: 15.103%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 4.156 ms	**Before**: 3.230 ms	**Before**: 50.106 ms
After: 3.358 ms	**After**: 2.560 ms	**After**: 47.415 ms

bfs_par_seg_alt

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 35.125%

512nodes_high_edge_variance.in	graph1MW_6.in
Before: 6771.712 kB	**Before**: 223999.010 kB
After: 4312.064 kB	**After**: 147999.370 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 42.099%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 1231.236 kB	**Before**: 9234.452 kB	**Before**: 300002.750 kB
After: 713.284 kB	**After**: 5334.788 kB	**After**: 173999.992 kB

Runtime with the CPU pipeline:

Average runtime reduction: 6.770%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 1.071 ms	**Before**: 4.837 ms	**Before**: 512.597 ms
After: 1.006 ms	**After**: 4.198 ms	**After**: 507.164 ms

Runtime with the GPU pipeline:

Average runtime reduction: 14.980%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 5.000 ms	**Before**: 10.445 ms	**Before**: 60.001 ms
After: 4.366 ms	**After**: 6.845 ms	**After**: 61.331 ms

bfs_par

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

512nodes_high_edge_variance.in	graph1MW_6.in
Before: 1942.500 kB	**Before**: 112725.680 kB
After: 1942.500 kB	**After**: 112725.680 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 4.195%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 538.488 kB	**Before**: 1948.896 kB	**Before**: 122137.936 kB
After: 505.668 kB	**After**: 1943.788 kB	**After**: 114530.108 kB

Runtime with the GPU pipeline:

Average runtime reduction: 8.841%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 8.435 ms	**Before**: 5.584 ms	**Before**: 30.961 ms
After: 7.113 ms	**After**: 5.011 ms	**After**: 30.781 ms

bfs_par_fused

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 6.290%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 220.923 kB	**Before**: 834.807 kB	**Before**: 53803.283 kB
After: 200.442 kB	**After**: 832.246 kB	**After**: 48803.282 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 14.088%

4096nodes.in	graph1MW_6.in
Before: 262.078 kB	**Before**: 63899.670 kB
After: 225.149 kB	**After**: 54899.157 kB

Runtime with the CPU pipeline:

Average runtime reduction: 2.096%

512nodes_high_edge_variance.in	graph1MW_6.in
Before: 2.216 ms	**Before**: 239.428 ms
After: 2.236 ms	**After**: 227.187 ms

Runtime with the GPU pipeline:

Average runtime reduction: 4.201%

4096nodes.in	graph1MW_6.in
Before: 1.413 ms	**Before**: 29.894 ms
After: 1.317 ms	**After**: 29.413 ms

bfs_rodinia_like

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 184.323 kB	**Before**: 829.955 kB	**Before**: 44999.883 kB
After: 184.322 kB	**After**: 829.954 kB	**After**: 44999.882 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.016%

4096nodes.in	graph1MW_6.in
Before: 208.966 kB	**Before**: 51095.246 kB
After: 208.901 kB	**After**: 51094.733 kB

Runtime with the GPU pipeline:

Average runtime reduction: 2.150%

4096nodes.in	graph1MW_6.in
Before: 1.180 ms	**Before**: 50.725 ms
After: 1.131 ms	**After**: 50.658 ms

bfs_sgm_alt

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 16.435%

512nodes_high_edge_variance.in	graph1MW_6.in
Before: 5134.339 kB	**Before**: 177999.253 kB
After: 4312.066 kB	**After**: 147999.372 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 38.364%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 1042.434 kB	**Before**: 7596.562 kB	**Before**: 254002.476 kB
After: 639.553 kB	**After**: 4720.385 kB	**After**: 156000.079 kB

Runtime with the CPU pipeline:

Average runtime reduction: -1.683%

512nodes_high_edge_variance.in	graph1MW_6.in
Before: 2.574 ms	**Before**: 356.861 ms
After: 2.690 ms	**After**: 352.802 ms

Runtime with the GPU pipeline:

Average runtime reduction: 10.818%

4096nodes.in	512nodes_high_edge_variance.in	graph1MW_6.in
Before: 1.434 ms	**Before**: 2.538 ms	**Before**: 45.020 ms
After: 1.284 ms	**After**: 2.002 ms	**After**: 44.654 ms

cfd

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 17.241%

fvcorr.domn.097K.toa	fvcorr.domn.193K.toa
Before: 22514.692 kB	**Before**: 44885.988 kB
After: 18632.852 kB	**After**: 37147.028 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 6.847%

fvcorr.domn.097K.toa	fvcorr.domn.193K.toa
Before: 28356.172 kB	**Before**: 56499.388 kB
After: 26415.252 kB	**After**: 52629.908 kB

Runtime with the CPU pipeline:

Average runtime reduction: -2.237%

fvcorr.domn.097K.toa	fvcorr.domn.193K.toa
Before: 106753.567 ms	**Before**: 218708.013 ms
After: 108115.990 ms	**After**: 225703.621 ms

Runtime with the GPU pipeline:

Average runtime reduction: 0.130%

fvcorr.domn.097K.toa	fvcorr.domn.193K.toa
Before: 2910.776 ms	**Before**: 5913.457 ms
After: 2903.814 ms	**After**: 5912.205 ms

hotspot

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

1024.in	512.in	64.in
Before: 16777.216 kB	**Before**: 4194.304 kB	**Before**: 65.536 kB
After: 16777.216 kB	**After**: 4194.304 kB	**After**: 65.536 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.000%

1024.in	512.in	64.in
Before: 16777.216 kB	**Before**: 4194.304 kB	**Before**: 65.536 kB
After: 16777.216 kB	**After**: 4194.304 kB	**After**: 65.536 kB

kmeans

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.004%

204800.in	kdd_cup.in
Before: 61445.868 kB	**Before**: 148209.420 kB
After: 61442.412 kB	**After**: 148207.260 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 15.149%

100.in	204800.in	kdd_cup.in
Before: 987.912 kB	**Before**: 249633.380 kB	**Before**: 435610.224 kB
After: 894.452 kB	**After**: 208442.372 kB	**After**: 350728.712 kB

Runtime with the CPU pipeline:

Average runtime reduction: -0.778%

204800.in	kdd_cup.in
Before: 3587.055 ms	**Before**: 4503.055 ms
After: 3575.044 ms	**After**: 4588.174 ms

Runtime with the GPU pipeline:

Average runtime reduction: -0.389%

100.in	204800.in	kdd_cup.in
Before: 5.066 ms	**Before**: 1038.420 ms	**Before**: 1304.263 ms
After: 5.152 ms	**After**: 1035.081 ms	**After**: 1301.634 ms

lavaMD

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

10_boxes.in
Before: 4052.000 kB
After: 4052.000 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.000%

10_boxes.in
Before: 6164.000 kB
After: 6164.000 kB

lud

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 12.832%

2048.in	256.in	512.in
Before: 133730.560 kB	**Before**: 2061.568 kB	**Before**: 8288.512 kB
After: 116936.960 kB	**After**: 1790.208 kB	**After**: 7229.696 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 12.982%

2048.in	256.in	512.in	64.in
Before: 169086.976 kB	**Before**: 2789.376 kB	**Before**: 10817.536 kB	**Before**: 335.872 kB
After: 151027.968 kB	**After**: 2392.320 kB	**After**: 9470.208 kB	**After**: 286.976 kB

Runtime with the CPU pipeline:

Average runtime reduction: 3.434%

2048.in	256.in	512.in
Before: 1941.034 ms	**Before**: 4.270 ms	**Before**: 32.000 ms
After: 1821.336 ms	**After**: 4.306 ms	**After**: 30.410 ms

Runtime with the GPU pipeline:

Average runtime reduction: -28.410%

2048.in	256.in	512.in	64.in
Before: 95.050 ms	**Before**: 4.000 ms	**Before**: 9.290 ms	**Before**: 1.056 ms
After: 105.738 ms	**After**: 5.177 ms	**After**: 11.238 ms	**After**: 1.605 ms

myocyte

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 15.909%

medium.in	small.in
Before: 23871.912 kB	**Before**: 17.172 kB
After: 23866.452 kB	**After**: 11.712 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 48.831%

medium.in	small.in
Before: 787218.989 kB	**Before**: 5685.382 kB
After: 405537.325 kB	**After**: 2889.498 kB

Runtime with the CPU pipeline:

Average runtime reduction: 0.955%

medium.in	small.in
Before: 23423.790 ms	**Before**: 11.229 ms
After: 23183.334 ms	**After**: 11.130 ms

Runtime with the GPU pipeline:

Average runtime reduction: 15.435%

medium.in	small.in
Before: 1079.390 ms	**Before**: 330.103 ms
After: 938.318 ms	**After**: 271.345 ms

nn-opt-1

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

medium.in
Before: 20527.544 kB
After: 20527.544 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 1.660%

medium.in
Before: 15793.712 kB
After: 15531.568 kB

Runtime with the GPU pipeline:

Average runtime reduction: -3.018%

medium.in
Before: 41.647 ms
After: 42.904 ms

nn-opt-4

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

medium.in
Before: 10264.304 kB
After: 10264.304 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 6.985%

medium.in
Before: 15011.488 kB
After: 13962.912 kB

Runtime with the GPU pipeline:

Average runtime reduction: -6.266%

medium.in
Before: 43.451 ms
After: 46.174 ms

nn

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

medium.in
Before: 10264.160 kB
After: 10264.160 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.000%

medium.in
Before: 10265.192 kB
After: 10265.192 kB

pathfinder

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

medium.in
Before: 41200.120 kB
After: 41200.120 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 0.000%

medium.in
Before: 41200.120 kB
After: 41200.120 kB

srad

Peak memory usage with the CPU pipeline:

Average peak memory usage reduction: 0.000%

image.in
Before: 8506.892 kB
After: 8506.892 kB

Peak memory usage with the GPU pipeline:

Average peak memory usage reduction: 10.837%

image.in
Before: 8513.948 kB
After: 7591.252 kB

Runtime with the GPU pipeline:

Average runtime reduction: 0.219%

image.in
Before: 32.849 ms
After: 32.778 ms