Forside

Eksamensinformation
NDAA93062E - Computer Science Thesis 30 ECTS, Department of Computer Science - Kontrakt:128456 (Philip Jon Børgesen)

Besvarelsen afleveres af
Philip Jon Børgesen
rkh383@alumni.ku.dk

Eksamensadministratorer
DIKU Eksamen
uddannelse@diku.dk

Bedømmere
Troels Henriksen
Eksaminator
athas@di.ku.dk
+4535335718

Patrick Bahr
Censor
paba@itu.dk

Besvarelsesinformationer

Titel: Reduktion af Synkrone GPU Hukommelsesoverførsler
Titel, engelsk: Reducing Synchronous GPU Memory Transfers
Vejleder / eksaminator: Troels Henriksen
Tro og love-erklæring: Ja
Indeholder besvarelsen fortroligt materiale: Nej
Må besvarelsen gøres til genstand for udlån: Ja
Må besvarelsen bruges til undervisning: Ja

UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

MSc thesis

Reducing Synchronous GPU Memory Transfers

Design and implementation of a Futhark compiler optimisation

Philip Jon Børgesen

Advisor: Troels Henriksen

Submitted: June 20, 2022

Abstract

We present a series of dataflow dependent program transformations that reduce memory transfers between a GPU and its host, and show how the problem of minimising memory transfers to the host amounts to finding minimum vertex cuts in a series of data dependency graphs. We provide a specialised algorithm to solve these minimisation problems, based on the Ford-Fulkerson max-flow algorithm, and detail techniques to model conditional execution and loops in a pure functional programming language.

We present our work in context of the array programming language Futhark, in whose compiler we have implemented our techniques. Empirical evaluation of 27 benchmark programs on four GPUs show mean speedups of 117–158%, heavily skewed by significant improvements to a few programs.

Contents

1 Introduction

Futhark [1] is a purely functional, statically typed, data-parallel array programming language designed to be compiled to efficient parallel code for execution on general-purpose GPUs via OpenCL [2] and CUDA [3]. It is meant to be used for the compute-intensive, data-parallelisable parts of applications written in other languages and features a heavily optimising ahead-of-time compiler that generally allows programmers to not deal with the low-level details of how to write efficient GPU code [4].

A Futhark program compiled to run on a GPU does not exclusively run on the GPU. Instead the execution is done in a back-and-forth collaboration between the CPU and the GPU with the CPU orchestrating tasks to run on the GPU as a co-processor. A key decision in the design of Futhark's compiler is thus where individual parts of compiled programs should be run. By default all parallel operations of Futhark compiled programs are performed on the GPU while all sequential work happens to be done on the CPU. This split happened due to ease of implementation, coincidentally benefiting from CPUs generally having superior single-threaded performance to GPUs, but suffers from the communication costs of exchanging data among the two.

In this thesis we explore how moving some of the sequential work to the GPU can reduce memory transfers and speed up the execution of compiled Futhark programs,

subject to the constraint that not all work can be migrated. Our focus will be the runtime behaviour of GPU accelerated code generated by the OpenCL [5] and CUDA [6] Futhark compiler backends, which share a common compiler optimisation pipeline. Benchmarks and discussions that involve specific runtime implementation details will be in context of v0.21.10 of the language and its compiler.

In section 2 we present the principles of CPU/GPU communication and how they relate to the mostly asynchronous runtime of a Futhark program. We explain why synchronous communication, particularly blocking inter-device memory transfers, is problematic from a performance viewpoint, and demonstrate the performance potential of reducing such by moving some sequential work to the GPU.

In section 3 we describe a small language that resembles a subset of the internal compiler representation of Futhark programs. We use this later to facilitate discussions of the concrete problem and to describe program optimisations.

In section 4 we present an automatic program transformation that reduces inter-device memory transfers. We show how the problem of minimising inter-device memory transfers can be modelled as a graph problem, and present a variant of the Ford-Fulkerson method [7] that effectively solves it. We then describe a series of optimising program transformations that moves sequential work to the GPU based on said model. We end the section with a walk-through of how each language construct is modelled and transformed.

In section 5 we present and discuss experimental test results that quantify the impact of our presented optimisations. We provide aggregated benchmark results for the *futhark-benchmarks* [8] benchmark suite that is endorsed by the Futhark project and provide microbenchmarks as rationale for a specific transformation.

In section 6 we provide two ways that our work can be expanded upon to further reduce the occurrence of inter-device memory transfers. We conclude the thesis with a summary of our most important findings and contributions.

2 Background

2.1 The runtime of a Futhark program

It is difficult to generalise the exact nature of communication that occurs between a CPU and a GPU as considerable hardware variance exists. For instance, the latency and overhead of communicating with a GPU that shares memory with the CPU and is integrated on the same circuit die is considerably different from communicating with a discrete graphics card via a PCIe bus, both in regards to the link itself and whether data transfers demand copying. Due to the variance there is no general concept of efficient zero-copy shared memory that program code running on both pieces of hardware simultaneously can access, and so the Futhark runtime issues explicit commands to copy data from one piece to the other. How the Futhark runtime does this differs between its OpenCL and CUDA backends but the underlying approach is very similar.

Both OpenCL and CUDA exposes a pipelined programming model in which commands to the GPU are enqueued with a driver and executed asynchronously [2][3]. For this thesis only in-order GPU pipelines are relevant, which means that enqueued commands appear to execute in the order they are submitted. For many driver implementations, particularly those under CUDA, commands do execute in that order [3]. There are no ordering guarantees for commands enqueued to different pipelines. The

```
entry vector_norm [n] (A: [n]f32): [n]f32 =
  let pow2 = map (\x -> x*x) A
  let sum = reduce (+) 0 pow2
  let len = f32.sqrt sum
   in map (\a -> a / len) A
```

Listing 1: Vector normalisation in Futhark.

command types we will consider are the execution of GPU functions, so-called *kernels*, and host-device, intra-device, and device-host memory transfers. We use *host* to refer to the CPU and its associated RAM, and *device* to refer to the GPU and its associated memory. Whether the memory subsystems in practice overlap is unimportant.

The pipelined programming model is useful as its asynchronous nature allows the host thread(s) and GPU to work in parallel. While the device is doing work, the host can perform other tasks such as prepare more work for the GPU or consume transferred results. It also allows more efficient utilisation of the device and interconnecting hardware. For instance, the driver can send commands to the GPU in batches to amortise bus communication overhead, increasing throughput at the cost of latency, and different phases of different commands in the pipeline can be performed concurrently, in theory allowing one command to be sent to the GPU while another is being executed. We note that the latency between a command being enqueued and the start of its actual execution may be quite long, subject to the hardware and driver implementation. The OpenCL specification only promises the eventual execution of enqueued commands [2, p. 17] and GPUs are generally optimised for throughput, not latency.

The part of a Futhark program that runs on the host executes sequentially in a single thread, which both is responsible for pipelining GPU work, managing memory, handling errors, and evaluating the non-parallelised aspects of the source program. Only a single in-order GPU pipeline is used. In the immediate representation all program data is represented as scalars (booleans, floating-point numbers, and integers) and arrays of these. As an implementation detail, arrays are stored on device, but their sizes are known to the host, as it allocates them. Kernels can read and write scalars to arrays and can accept scalar arguments provided by the host, but cannot return a result. Results can thus only be communicated by writing them to arrays, which later can be read by the host. Futhark uses a uniqueness type-system to support in-place updates of arrays while maintaining purity.

Listing 1 shows an example of a simple Futhark program that normalises a vector of n 32-bit floating point numbers. Its **map** and **reduce** functions and their lambda arguments are compiled to code that execute as kernels. `A` and `pow2` are arrays that reside on device. **reduce** computes a single-element array of type [1]**f32** whose only element it reads to the host and returns, to be bound as the scalar variable `sum`. The square root of `sum` is then computed and passed as an argument to the kernel that executes the final **map** expression. The arrays that each kernel return results via are allocated before kernel execution and are also passed as kernel arguments.

Implementation-wise, all array reads performed on the host are done using synchronous operations. This means that the read required to bind `sum` in Listing 1 must wait for the **map** and **reduce** kernel[1] to finish after which the host incurs the inter-

[1]The Futhark compiler fuses these two operations into a single kernel.

device communication cost of reading the produced scalar. Since the host performs no useful work while waiting for the transfer to complete, program execution suffers as its degree of parallelism is reduced. What is worse, while the memory transfer occurs the GPU pipeline will be empty (we assume that no out-of-order execution occurs) and only when the read completes can `len` be computed and the next **map** kernel launch be initiated. The deeper the execution pipeline and the longer the start execution latency, the more this will hurt. By performing a blocking operation the host thus not only delays itself but also stalls device execution which is responsible for the vast majority of hardware threads that contribute to parallel program execution. If no other process or thread uses the GPU, system-wide GPU utilisation will drop.

The host also writes most scalars to arrays using blocking operations, with similar synchronisation costs. The primary exception is direct copying between arrays via slicing, which is implemented as an asynchronous intra-device memory transfer. Here slicing refers to an operation that returns an array-typed view of an existing array. The slice `A[2:5]` of `A = [1, 2, 3, 4, 5, 6]` equals `[3, 4, 5]`. In many cases the compiler can optimise the write of an individually read scalar into an asynchronous slice copy, turning `A[0]` $\leftarrow$ `B[7]` into `A[0:1]` $\leftarrow$ `B[7:8]`. The OpenCL backend also implements an optimisation that makes writing a scalar constant an asynchronous operation[2].

The runtime may also perform synchronous operations when allocating memory from the device driver but this is relatively uncommon due to a free list mechanism that enables memory reuse. From a performance viewpoint these are thus of less interest compared to the reads and writes that always block. We will not spend any effort on optimising memory allocation further.

2.2 An optimisation idea

One way to avoid the theoretical performance slowdown of synchronisation would be to make all data transfers asynchronous. The CUDA documentation however hints that the data transfers themselves may limit performance, stating that "applications should strive to minimize data transfer between the host and the device" [3, sec. 5.3.1]. In the same section it also mentions that it is better to batch many small transfers into a one large transfer due to overhead that each transfer incurs. Instead of making the data transfers between host and device asynchronous we will therefore instead examine how they can be reduced in number.

Two techniques already utilised by the Futhark compiler is *common subexpression elimination* (CSE) and *sinking*. CSE removes duplicate computations, replacing them with already computed results. At its most basic it can transform an expression such as `A[i] + A[i]` into **let** `x = A[i]` **in** `x + x`, where `A[i]` reads the `i`th element of array `A`. In some cases the compiler can also eliminate reads from arrays with known contents.

Sinking moves an array read that only is used within one branch of an **if** expression into that branch. For example, the code fragment

```
let x = A[0]
 in if A[1] == 0 then x+2 else 42
```

can be transformed into

[2]Based on feedback and starting with v0.21.11, this optimisation is now also implemented in the CUDA backend. v0.21.11 is the version that introduces the optimisations presented by this thesis.

```
if A[1] == 0 then A[0]+2 else 42
```

thereby eliminating the A[0] read in the best case (when A[1] is not zero) while maintaining the same number of reads in the worst case. The worst case would suffer if x was used in the branches of multiple if statements and moved into each of them.

While both techniques are capable of eliminating some blocking memory transfers, neither are capable of optimising the blocking read that occurs in Listing 1. The issue is not under which conditions the vector length is computed but *where* it is computed. The amount of work done by the host, i.e. computing a square root, is not proportional to the cost of reading its argument.

A suggestion found in the CUDA Performance Guidelines [3, sec. 5] is to move computations from host to device such that the intermittent transfers are avoided. This idea has merit and is orthogonal to already implemented techniques. Consider the following pseudo code transformation of the Listing 1 vector normalisation program

```
entry vector_norm' [n] (A: [n]f32): [n]f32 =
  let pow2 = map (\x -> x*x) A
  let S = reduce' (+) 0 pow2
  let L = gpu { f32.sqrt S[0] }
   in map (\x -> x / L[0]) A
```

Listing 2: Proposed optimisation of Listing 1.

where **reduce'** is a modified **reduce** that returns [1]**f32** instead of reading its element; **gpu** $\{e\}$ executes the expression e on device as a single-threaded kernel, returning the result of e in an array; and the kernel usage of `len` is replaced with `L`[0], reading the vector length from device memory rather than receiving it as a scalar argument. Now all program evaluation is done on the device, leaving the host just to administrative tasks, and the synchronous transfer is eliminated. When all allocations can be served by the free list, no mid-program synchronisation should occur at all.

Empirical evidence supports the worth of this idea. When benchmarking the Listing 1 program and a manually optimised version that corresponds to Listing 2 on a variety of data centre and consumer grade hardware we obtain a speedup of 108–154% (see Figure 1). This is significant and indicates that it is worth aggressively optimising data transfers away via code migration.

Blocking writes can also be transformed by means of migration. An update expression such as `A` **with** [`i`] = `x`, which synchronously writes `x` to the `i`th index of `A`, can be turned into `A` **with** [`i:i+1`] = **gpu** { `x` } which is an asynchronous slice copy from an asynchronously populated single-element array. Unfortunately, micro benchmarks indicate that in the simple case, the overhead of the added kernel launch degrades performance. The transformation only appears profitable when the same scalar is to be written two or more times, allowing reuse of its intermediate single-element array, or two kernels can be merged into one, such as to transfer two scalars to device at once.

Rather than pursuing a write optimisation based on these conditions it can be observed that writes to arrays in Futhark programs often depends on array reads. This is based on the *futhark-benchmarks* [8] benchmark suite, endorsed by the Futhark project and taken to be a realistic sample of Futhark programs, wherein all array writes in non-test code follow the pattern, even in external library dependencies. This means that if array reads are minimised by migrating dependent expressions to device, then

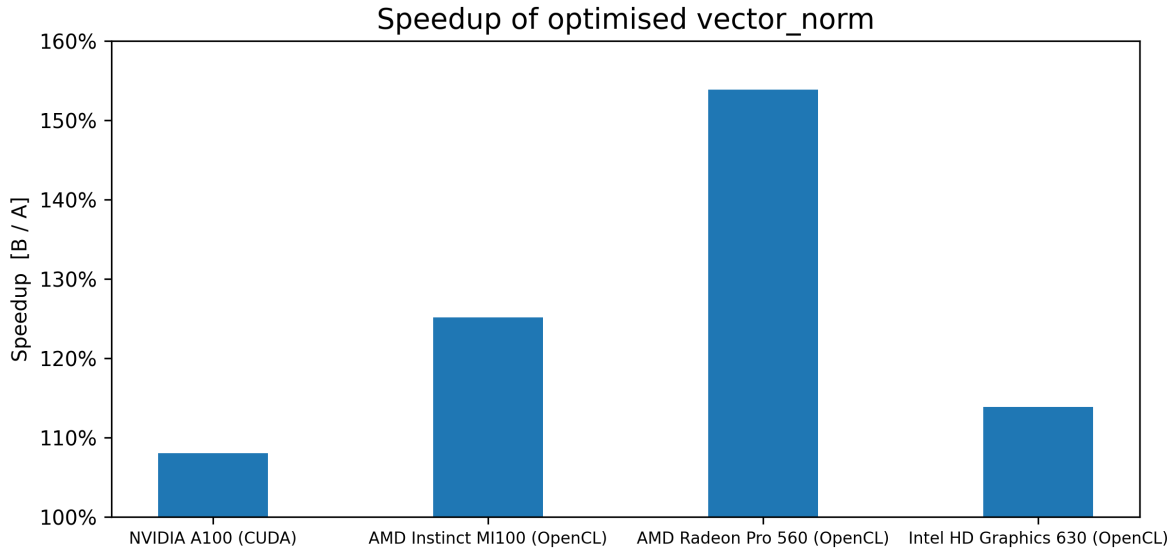


Figure 1: How much faster `vector_norm`′ is compared to `vector_norm` when executed on an array of ten million 32-bit floating-point numbers. The bars indicate the speedup for four different hardware configurations. For details about the hardware and benchmarking method, see section 5.

most writes can simultaneously be optimised since their write arguments already will be present on device.

3 Language

Before describing an algorithm that can drive a program transformation that reduces memory transfers via migration it is useful to define a language to operate upon. The language we describe in this section mimics a subset of the intermediate language representation used in the Futhark compiler, albeit simplified for conciseness. The language is in Administrative Normal Form [9], which means that all expression operands are constants or variables. Barring some builtin operators it is a monomorphic first-order language without recursive data types, and as such it is far more restrictive than the language used in examples given in section 2. The purpose of the language is to support presentation of the techniques used in the optimisation, not showing every instance of applying the techniques to the actual intermediate language.

3.1 Syntax

The following grammar describes the abstract syntax of our language. We let x be a meta-variable that ranges over all language variables, let f be a meta-variable that ranges over all possible function bindings, and let $int \in \mathbb{Z}$, $m \in \mathbb{N}$, and $n \in \mathbb{N}^+$.

$program ::= decl_1 \ldots decl_n$

$decl ::= stm \mid \textbf{def}\ f\ \textbf{=}\ fn$

$stm ::= \textbf{let}\ x_1, \ldots, x_n\ \textbf{=}\ e$

fn ::= $\lambda\, x_1 \ldots x_n \to e$

se ::= int | **true** | **false** | x

e ::= $stm_1 \ldots stm_n$ **in** e | $se_1, \ldots, se_n$ | $f\ se_1 \ldots se_n$ | $\neg\, se$ | $se_1 + se_2$ | $se_1 \leq se_2$
| **if** x **then** e_1 **else** e_2 | **loop** $x_1 = se_1, \ldots, x_n = se_n$ $form$ **do** e
| $[se_1, \ldots, se_m]$ | $x[dims]$ | x **with** $[dims] \leftarrow se$ | **copy** x | **iota** $se_1\ se_2\ se_3$
| **replicate** $[se_1, \ldots, se_m]$ se | **map** $fn\ x$ | **reduce** $fn\ se\ x$ | **gpu** e

$form$ ::= **for** $x <$ se | **for** x_1 **in** x_2 | **while** x

$dims$ ::= $dim_1, \ldots, dim_n$

dim ::= se | $se_1 : se_1 + se_2$

Furthermore let $n, r \in \mathbb{N}^+$. The types of our language is then given by the grammar:

τ ::= **int** | **bool** | $[se]\tau$ | $\tau_1 \to \ldots \to \tau_{n+1},\ \ldots,\ \tau_{n+r}$

3.2 Semantics and type constraints

The semantics and type rules of our language are mostly conventional. The primary exception is that some expressions may produce more than one value, which can be understood as them producing a tuple that immediately is unpacked. Additionally kernel expressions such as **reduce** and **gpu** produce array-typed values even when their inner expressions produce scalars, and array variables may *alias* each other, which means that their values are views of the same data. The x **with** $[dims] \leftarrow se$ expression invalidates x and all its aliases, essentially unbinding them to forbid future use. It semantically produces a new array with no aliases.

In the following we give a formal description of the language. A reader should feel free to skip ahead and merely reference the individual descriptions if the meaning of a language construct later proves unclear. The definitions we make in the next paragraph will be used however, and should thus not be skipped.

Let $\tau(s)$ be the type of s. Value v of type $\tau(v) \in \{\textbf{int}, \textbf{bool}\}$ is a scalar, while it is an array if its type is $\tau(v) = [z]\tau_R$ for some **int** typed constant or variable z, and some scalar or array type τ_R. The full type of an array might be $[x][y][10]$**int**, which denotes an integer array of rank three, each dimension respectively of length x, y, and 10. We denote x to be its outer dimension. The type $\tau_1 \to \ldots \to \tau_{n+1},\ \ldots,\ \tau_{n+r}$ is for functions that take n arguments and produce r values.

Let A_i denote the ith row of some array A, and let $v_i(e)$ be the ith value produced by the expression e. When $v_i(e) = w$ then $\tau(v_i(e_1)) = \tau(w)$. The runtime semantics and type constraints for each language rule is then given by:

- *program* is a sequence of top-level declarations, at least one function and any number of variable bindings. Variable bindings may depend on previous declarations in the sequence. Function bindings may depend upon any top-level declaration. A program is run by passing arguments to one of its functions.
- **let** $x_1, \ldots, x_n$ **=** e is a statement that binds x_i to $v_i(e)$.
- **def** f **=** fn binds f to the function fn. We have $\tau(f) = \tau(fn)$.

- $\lambda\ x_1 ...\ x_n \rightarrow e$ is some function *fn* that accepts n arguments. We have $\tau(fn) = \tau_1 \rightarrow ... \rightarrow \tau_{n+1},\ ...,\ \tau_{n+r}$ such that $\tau(x_i) = \tau_i$ and $\tau(v_j(e)) = \tau_{n+j}$.
- *se* is a subexpression which either evaluates to a constant value or a previously computed value:
  - *int* is an integer constant; $v_1(int) = int$. Its type is $\tau(int) =$ **int**.
  - **true** is a boolean constant; $v_1(\textbf{true}) =$ **true**. Its type is $\tau(\textbf{true}) =$ **bool**.
  - **false** is a boolean constant; $v_1(\textbf{false}) =$ **false**. Its type is $\tau(\textbf{false}) =$ **bool**.
  - When x is bound to w, $v_1(x) = w$.
- $stm_1 ...\ stm_n$ **in** *e* evaluates to *e* in an environment with the variables bound by the $stm_1 ...\ stm_n$ statements. We have $\tau(v_i(stm_1 ...\ stm_n\ \textbf{in}\ e)) = \tau(v_i(e))$.
- $se_1, ..., se_n$ evaluates to the respective subexpressions, i.e. $v_i(se_1, ..., se_n) = v_1(se_i)$.
- When *f* is bound to $\lambda\ x_1 ...\ x_n \rightarrow e$ with type $\tau_1 \rightarrow ... \rightarrow \tau_{n+1},\ ...,\ \tau_{n+r}$ then *f* $se_1 ...\ se_n$ evaluates to *e* in an environment where x_i is bound to se_i. Results have types $\tau(v_j(f\ se_1 ...\ se_n)) = \tau_{n+j}$ and the function arguments are constrained such that $\tau(se_i) = \tau_i$.
- $\neg\ se$ evaluates to **true** if $v_1(se)$ is **false**, otherwise **false**. Types are constrained such that $\tau(\neg\ se) = \tau(se) =$ **bool**.
- $se_1 + se_2$ evaluates to $v_1(se_1) + v_1(se_2)$. We have $\tau(se_1 + se_2) = \tau(se_1) = \tau(se_2) =$ **int**.
- $se_1 \leq se_2$ evaluates to **true** if $v_1(se_1) \leq v_1(se_2)$, otherwise **false**. The corresponding type rules are $\tau(se_1 \leq se_2) =$ **bool** and $\tau(se_1) = \tau(se_2) =$ **int**.
- **if** *x* **then** e_1 **else** e_2 evaluates to e_1 when $v_1(x)$ is **true**, otherwise to e_2.

  We have $\tau(x) =$ **bool** and $\tau(v_i(\textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2)) = \tau(v_i(e_1)) = \tau(v_i(e_2))$.
- **loop** $x_1 = se_1, ..., x_n = se_n$ *form* **do** *e* iteratively evaluates *e* until some condition ceases to hold, each time updating the environment of bound variables based on the values *e* produced. For iteration $k = 0$, the loop parameter x_i is bound to se_i. For iteration $k > 0$, x_i is bound to $v_i(e_{k-1})$ i.e. the values produced by *e* in iteration $k - 1$. When the condition no longer holds the loop expression produces the values $x_1, ..., x_n$. Thus $\tau(x_i) = \tau(se_i) = \tau(v_i(e)) = \tau(v_i(\textbf{loop}\ x_1 = se_1, ..., x_n = se_n\ form\ \textbf{do}\ e))$. The condition and other variables depend upon the loop form:
  - When *form* is **for** $x_I < se_N$ the loop repeats $v_1(se_N)$ times. In each iteration the variable x_I is bound to k. We have $\tau(x_I) = \tau(se_N) =$ **int**.
  - When *form* is **for** x_R **in** x_A then x_A binds an array of type $[se_N]\tau_R$ and the loop repeats $v_1(se_N)$ times. In each iteration the variable x_R is bound to the kth row of x_A with semantics corresponding to **let** x_R = $x_A[k]$. We have $\tau(x_A) = [se_N]\tau_R$ and $\tau(x_R) = \tau_R$.
  - When *form* is **while** x_c the loop repeats for as long as $v_1(x_c)$ is **true**. x_c is one of the variables $x_1, ..., x_n$ that are bound by the loop.

- $[se_1, ..., se_m]$ evaluates to an array with an outer dimension of m. If $A = [se_1, ..., se_m]$ then $A_i = se_i$ and $\tau(A) = [m]\tau_R$, implying $\tau(se_i) = \tau_R$. The array has no aliases.

  If every se_i is a constant, then the operation is allocation plus a single asynchronous intra-device memory transfer. Otherwise, the operation is allocation of an array A followed by m writes that correspond to A **with** $[i-1] = se_i$ for each $1 \leq i \leq m$. Note that indices are zero-based.

- *x*[*dims*] evaluates to the subset $s = \mathcal{D}(v_1(x); dims)$ of the array bound to *x* with rank $r \geq n$. We have $\tau(x) = [z]\tau_R$. Recall that $m \in \mathbb{N}, n \in \mathbb{N}^+$. The function $\mathcal{D}$ then identifies a subset of the array bound to *x*, given by

$$\mathcal{D}(A; dim_1, ..., dim_n) = \begin{cases} A_i & \text{for } n = 1, dim_1 \text{ is } se, i = v_1(se) \\ \mathcal{D}(A_i; dim_2, ..., dim_n) & \text{for } n > 1, dim_1 \text{ is } se, i = v_1(se) \\ \mathsf{s}_{f,l}(A; dim_2, ..., dim_n) & dim_1 \text{ is } se_1 : se_1 + se_2, \\ & f = v_1(se_1), l = v_1(se_2) \end{cases}$$

  where all *se* terms that occur in *dims* are of type **int**, and $\mathsf{s}_{f,l}(A; dim_1, ..., dim_m)$ identifies the array-typed slice A' of A given by

$$A'_i = \begin{cases} A_{f+i} & \text{for } m = 0 \\ \mathcal{D}(A_{f+i}; dim_1, ..., dim_m) & \text{for } m > 0 \end{cases} \qquad (\text{for } 0 \leq i < l)$$

  If $\tau(s) \in \{\textbf{int}, \textbf{bool}\}$ and *x*[*dims*] is evaluated on the host, then the operation is synchronous, involving an expensive device-host memory transfer that stalls execution. When evaluated on device, no inter-device communication occurs, the read can be considered cheap, and any latency involved in data fetching affects just the local device thread. When evaluated by a parallel kernel the cost can be considered negligible.

  If $\tau(s) = [z]\tau_R$ then s is an array of rank equal to the count of $se_1 : se_1 + se_2$ terms in *dims*, with respective dimensions of length se_2, s aliases *x*, and the operation has constant work complexity irrespective of where it is evaluated.

- *x* **with** [*dims*] $\leftarrow$ *se* evaluates to the array bound to *x*, with the contents of $v_1(se)$ written to the corresponding indices of $s = \mathcal{D}(v_1(x); dims)$. We have $\tau(s) = \tau(se)$ and $\tau(x \textbf{ with } [dims] \leftarrow se) = \tau(x)$. Afterwards neither *x* nor any variable that *x* directly or indirectly aliases (incl. aliases of aliased values) may be used before being rebound. Binding a variable to an array aliases it.

  Operation-wise the expression updates the underlying memory in-place. If $\tau(se) \in \{\textbf{int}, \textbf{bool}\}$ then the operation is an expensive host-device memory transfer when evaluated on the host. When *se* is a variable the memory transfer is synchronous; when it is a constant the memory transfer may or may not be asynchronous. When *se* is an array then the operation is performed asynchronously, involving no memory transfers with the host.

- **copy** *x* evaluates to an array that equals $v_1(x)$ but which has no aliases. It follows that $\tau(\textbf{copy } x) = \tau(x) = [z]\tau_R$. When evaluated on the host, copies are created using asynchronous intra-device memory transfers that can exploit the inherent parallelism of data copying.

- **iota** se_1 se_2 se_3 evaluates to an array A where $A_i = s \cdot i + b$ for $0 \leq i < t$ and $t = v_1(se_1), b = v_1(se_2), s = v_1(se_3)$. We have $\tau(\textbf{iota}\ se_1\ se_2\ se_3) = [se_1]\textbf{int}$ and $\tau(se_1) = \tau(se_2) = \tau(se_3) = \textbf{int}$. When evaluated on the host the array computation is parallelised by an asynchronous kernel. The array has no aliases.

- **replicate** [se_1, ..., se_m] se_e evaluates to an array of at least rank m with dimension $i \leq m$ being of length se_i. All rows of dimension m equals $v_1(se_e)$. The array has no aliases and has type $[se_1]$ ... $[se_m]\,\tau(se_e)$ where $\tau(se_i) = \textbf{int}$. When evaluated on the host the array rows are populated in parallel by an asynchronous kernel.

- **map** *fn x* applies the function *fn* of type $\tau_a \to \tau_b$ to every row of the array bound to *x*, evaluating to a new array of type $[z]\tau_b$ containing the results. The type of *x* is $[z]\tau_a$. When evaluated on the host the array rows are computed in parallel by an asynchronous kernel. The produced array has no aliases.

- **reduce** *fn se x* reduces a monoid to a single element. The variable *x* binds an array A of type $[z]\tau_e$ that holds z elements, *fn* is an associative binary function of type $\tau_e \to \tau_e \to \tau_e$, and $\epsilon = v_1(se)$ is the neutral element of the monoid, also of type τ_e. The expression evaluates to an array R of type $[1]\tau_e$ where $R_0 = \mathsf{R}_A(z)$ and

$$\mathsf{R}_A(z) = \begin{cases} \epsilon & \text{if } z = 0 \\ fn\,(\mathsf{R}_A(z-1),\ A_{z-1}) & \text{otherwise} \end{cases}$$

  When evaluated on the host the computation in done in parallel by an asynchronous kernel. The produced array has no aliases.

- **gpu** *e* evaluates *e* within an asynchronous, single-threaded kernel, producing an array for every value produced by *e* such that $v_i(\textbf{gpu}\ e)_0 = v_i(e)$. The type of $v_i(\textbf{gpu}\ e)$ is $[1]\,\tau(v_i(e))$. The produced arrays have no aliases.

4 Optimisation

While the program transformation that we are about to present also can be used to optimise the initial computation of top-level program variables, in this thesis we will only discuss the optimisation in the context of top-level functions. Since a program is run by invoking one of these, their semantics must not be altered. We limit the optimisation to how each top-level function individually can be transformed, subject to the constraints of their type signatures.

Assumption 1. To simplify our descriptions we assume that the expression *e* that occurs in the *fn* term and in **if**, **loop**, and **gpu** expressions always will be of the form

$$stm_1\ \ldots\ stm_m\ \textbf{in}\ se_1, \ldots, se_n \qquad \text{or just} \qquad se_1, \ldots, se_n$$

and that the expression form $stm_1\ \ldots\ stm_m$ **in** *e* otherwise does not occur. It is trivial to transform a program to abide to these restrictions and so no generality is lost. Figure 2 provides an example. For conciseness we will still use the full language in examples.

In a program abiding to Assumption 1, all computations of a function are done by statements that will be evaluated on either host or device. Statements that will be evaluated on device are grouped into constructs that represent kernels, such as **reduce** and

```
def double_sum = λA, sq →
  let d = let sum = if sq
                    then loop s = 0 for i in A do s+i
                    else let R = reduce (λa, b → a+b) 0 A
                         in R[0]
          in sum + sum
   in gpu d
```

```
def double_sum = λA, sq →
  let sum = if sq
            then let x = loop s = 0 for i in A do
                         let t = s+i in t
                 in x
            else let R = reduce (λa, b → let p = a+b in p) 0 A
                 let r = R[0]
                 in r
  let d = sum + sum
  let B = gpu d
   in B
```

Figure 2: Transforming a function to abide to Assumption 1. Top: Original representation of `double_sum`. Bottom: Transformed representation where every computation is done by a statement. `double_sum` has type [n]**int** → **bool** → [1]**int**.

gpu; we will simply call them "kernels" and overload the term. In Figure 2 the transformed statement **let** `p = a+b` **in** `p` is the only statement that is evaluated on device; the inner expression of **gpu** `d` merely evaluates to `d`, which is computed on host.

The variables bound by a statement logically reside in the same space (and kernel, if any) as that statement, either host or device. A statement may freely use variables that reside on the host[3] or in the same kernel as itself, but to use a value outside the kernel that computes it, that kernel must return the value in an array, and the usage site must read it from that array. This is a logical consequence of how kernels and variable scopes work. We use the phrase "return in an array" to emphasise that all kernel return values are communicated via arrays. While a device statement thus freely may use values computed on the host, it takes significant effort for a host statement to use values computed on device.

If all uses of a host read `h = A[i]` are within kernels, the device-host memory transfer can be prevented by inlining `h` into each of those kernels. That is the code fragment

```
let h = A[i]
let B = map (λx → x ≤ h) A
let C = map (λx → x + h) A
```

can be transformed into

```
let B = map (λx → let h = A[i] in x ≤ h) A
let C = map (λx → let h = A[i] in x + h) A
```

[3]Free in terms of the intermediate representation. The free variables of a kernel construct will be passed as arguments to the actual kernel function that is generated. This is generally cheap but not free.

This increases the number of device array reads but these do not stall execution and are generally much cheaper than inter-device memory transfers. If `h` is used by a set of host statements the host read can be prevented by also moving all of those to device, but then all host usages of their results must in turn be considered. The minimisation of host reads is thus a global optimisation problem: Select a set of host statements to move to device such that as few device values are used by host statements.

It is not possible to move all statements to device as some statements only can be initiated from the host due to compiler limitations. We say that such statements are *host-only* but will also use the term for statements that merely are infeasible to move. Most compiler limitations stem from the constraint that device memory is allocated by the host, which means that the backing memory of all arrays created by a kernel must be allocated before that kernel can be launched. This in turn requires the sizes of the respective arrays to be known in advance.

We likewise designate all kernel evaluated statements as being *device-only*, assuming their intended device execution to be optimal. We note that **gpu** expressions do not naturally occur in the intermediate Futhark representation that we study, and that it thus generally is not possible to reduce inter-device memory transfers by migrating statements from device to host. Since variables reside in the same space as the statement that binds them, by extension a variable bound by a host-only or device-only statement is itself respectively host-only or device-only.

4.1 Graph representation

To determine which statements of a given function that should be migrated from host to device we construct a directed graph $G = (V, E)$ where vertices are host variables with incoming edges from the scalar values they are computed from. We use the term "variable" loosely to also mean its vertex representation and only represent variables that meaningfully contribute to the problem, subject to later definition.

An edge $a \to b$ in such graph signifies that if the scalar value a resides on device, then either a must be transferred to the host (at great cost) or b must be computed on device, requiring its host statement to be migrated. There exists no outgoing edges from array-typed variables as migrated arrays can be made available to the host without any memory transfer. This is exemplified and explained in detail by Figure 3.

Any host variable that directly depends upon a device value via an array read, such as **let** `x = A[0]`, receives an edge from a distinct synthetic vertex s that represents the dependency. We call s a *source* as it is the source of an inter-device memory transfer that we wish to prevent, and we use the term *synthetic vertex* to denote a vertex with no variable representation in our language.

We only add a variable to the graph if it may be reached from a source as its migration otherwise can be foreseen to be unnecessary. Constants are trivially excluded. Device variables bound within kernels are irrelevant to the problem as they are considered device-only, already reside on device, and are inaccessible to host statements. Intra-kernel device variables are thus excluded, as demonstrated by Figure 4.

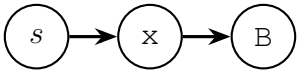
Some scalar variables bind values that must be made available to the host. To model this constraint we replace all outgoing edges from such a variable with a single edge to a synthetic vertex t, which we call a *sink*. We do not add new edges from a vertex that already has an edge to a sink as any edges beyond the sink-connecting one are pointless: If the variable resides on device, then it must be read to the host, and so

```
let x  = A[0]
let B  = [1,x]
let Bs = replicate [3] B
```

(a)

```
let B' = gpu let x = A[0] in [1,x]
let B  = B'[0]
let Bs = replicate [3] B
```

(b)



(c)

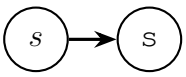
Figure 3:
(a) In this code fragment `x` depends on the device value read by `A[0]`, which entails a device-host memory transfer. The memory transfer can be prevented by migrating **let** `x = A[0]` to device but then either `x` must be transferred to the host, costing a memory transfer to no avail, or `B` must be computed on device. If **let** `B = [1,x]` is moved then `B` can be made available to the host without any memory transfer (see below), and therefore it receives no outgoing edge in (c) despite `Bs` depending upon it.

(b) By migrating the statements that compute `x` and `B` a device-host memory transfer can be prevented completely. The expression `B'[0]` evaluates in constant time without any inter-device communication and creates a slice of `B'` that corresponds to `B`.

(c) A graph representation of (a) where s represents the device value read by `A[0]`. Neither `A` nor `Bs` is represented by virtue of not depending on a scalar variable, which means that no memory transfer possibly can be prevented by migrating them.

```
def sum_cmp = λA →
  let S = reduce (λa, b → let p = a+b in p) 0 A
  let s = S[0]
   in map (λx → let c = x ≤ s in c) A
```

(a)

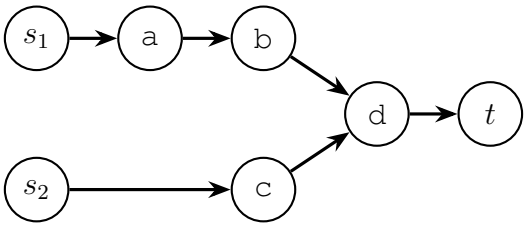


(b)

Figure 4: (a) A small function that compares integers to their sum and (b) its graph representation. `sum_cmp` has type **[**`n`**]int** → **[**`n`**]bool**. The source vertex s represents the device value read by `S[0]`, which causes an inter-device memory transfer. The device-only variables `a`, `b`, `p`, `x`, and `c` are excluded from the graph as they already reside on device and no host statement can depend on them but via an array read.

```
def add = λA, x →
  let a = A[0]
  let b = a + x
  let c = A[1]
  let d = b + c
   in d
```

(a)



(b)

Figure 5: (a) A small function that adds three integers and (b) its graph representation. `add` has type `[2]`**int** $\rightarrow$ **int** $\rightarrow$ **int**. The function arguments `A` and `x` are not represented in the graph as none of them are reachable from a source vertex. The vertices s_1 and s_2 are source vertices that represents the device values read by `A[0]` and `A[1]`. The vertex `t` is a sink that requires `d` to be made available to the host, to be used as a result. If the statements that bind variables `a`, `b`, `c` and `d` were migrated to device, the number of device-host memory transfers would be reduced from two to one.

none of its dependents need be migrated to use it. We connect scalar variables to sinks if they are used as the operands of host-only statements or are returned by the top-level function. The latter is demonstrated by Figure 5.

In subsection 4.5 we go into detail with how statements of each kind of expression is graphed, but to formalise the problem and present the algorithm that drives the migration analysis it suffices to provide Definition 1 and Statement 1.

Definition 1. We define the set $\mathcal{E}$ to include any statement that does *not* compute an expression of the form:

- $se_1, ..., se_n$
- **if** x **then** e_1 **else** e_2
- **loop** $x_1 = se_1, ..., x_n = se_n$ *form* **do** e
- **gpu** e

We leave the $\mathcal{E}$-membership of statements that evaluate stm_1 ... stm_n **in** e expressions be undefined due to Assumption 1.

Statement 1. *Let $F_e \subseteq V$ be vertices that correspond to variables that the expression e is computed from, and let F_e only contain source vertices and scalar variables that have no edge to a sink. When the $\mathcal{E}$ statement* **let** $x_1, ..., x_n$ **=** e *is graphed it then holds that:*

1. *The variables $x_1, ..., x_n$ are all added to V if and only if $F_e \neq \emptyset$.*
2. *The edges added to E for each variable x_i is those given by $\{u \rightarrow x_i \mid u \in F_e\}$.*

The variables bound by non-$\mathcal{E}$ statements directly correspond to constants or variables bound by other statements and thus must be considered for migration on their own. For our needs and purposes all host-only variables are bound by $\mathcal{E}$ statements with no direct dependence on device values. Since all scalar operands of host-only statements are connected to sinks, no edge will exist to a host-only variable. This makes them unreachable from a source, and thus they are not included in the graph.

4.2 Problem statement and basic solution properties

We can now formally define our optimisation problem:

Statement 2. *Given a graph representation $G = (V, E)$ of a top-level function, partition the vertices V into two disjoint sets, D and H, such that D contains all source vertices, H contains all sink vertices, and the vertices in D with an edge to a vertex in H is a minimum vertex cut denoted $C = (D, H)$, which secondarily minimises D.*

That is we want to minimise the number of vertices in D with an edge to a vertex in H, subject the given partitioning constraints, and such that D is as small as possible. This corresponds to minimising the number of inter-device memory transfers while migrating as few statements to device as possible, thereby keeping the overhead of running kernels small and exploiting that the host generally can perform sequential work faster than the device. We refer to D as the *device set* of a solution, and refer to H as its *host set*. We identify a solution by its vertex cut $C = (D, H)$ which contains the members of D that have an edge to a vertex in H. We emphasise that C is a set.

Solution properties

Given a solution $C = (D, H)$ to an instance of the Statement 2 graph problem, the set of $\mathcal{E}$ statements to move to device are those that bind a variable represented by a vertex in D. Statements of ungraphed variables remain where they reside, and so host-only statements will stay on the host. The graphing of non-$\mathcal{E}$ statements that we describe in subsection 4.5 is done such that C also provide a solution for reads that they depend upon. The criteria for their migration differ though.

Note that all variables bound by an $\mathcal{E}$ statement will be a subset of either D or H, so all variables bound by such statement will agree as to whether the statement should remain on host or be moved. Theorem 1 states this formally.

Theorem 1. *Let $C = (D, H)$ be a solution to the minimisation problem G, and let the set B be the vertices of all variables bound by any single $\mathcal{E}$ statement. Then B cannot intersect both D and H and thus must be a subset of one of them.*

Proof. Let F be the subset of D that the vertices in B each has an ingoing edge from, and assume that B intersects both D and H. Because B intersects D and D is minimum, F cannot be empty. Because B intersects H then F must also be a subset of C, and since C and D are both minimum then B must be disjoint from D.

This a contradiction as we assumed B to intersect D, so the assumption must be false. We conclude that the set of variables bound by any single $\mathcal{E}$ statement cannot intersect both D and H and thus must be a subset of exactly one of them. □

In subsection 4.1 we gave rationale for why a number of vertices may be excluded from the graph representation but did so without any formal proofs. In the following we argue that the partitioning of all excluded variables is predetermined, amounting to them remaining where they currently reside. We start by showing that any vertex not reachable from a source vertex will belong to H, which allows them to be excluded.

Lemma 1. *Let $C = (D, H)$ be a solution to the minimisation problem G. Then every vertex in D can be reached from a source vertex.*

Proof. Assume that some vertex $v \in D$ could not be reached from any source vertex, and let X be the set of all vertices in D that can reach v, incl. v itself. Then all vertices in X could be moved to H without increasing the size of the minimum vertex cut C, the partitioning would be valid since X contains no source vertex, and we would have $|D \setminus X| < |D|$. This implies that $|D|$ is not minimum, and C can thus not be a solution. We have found a contradiction, so v cannot exist, and we conclude that every vertex in D thus must be reachable from some source vertex. □

Corollary 1. *Any variable in G that no source vertex can reach will be a member of H.*

It can also be shown that a solution $C = (D, H)$ to the minimisation problem G is unique. This allows us to refer to *the* solution to any instance of Statement 2, not just one. More importantly, upon verifying a candidate solution to some problem G we can argue that any algorithm that solves G will produce that solution. The uniqueness of a solution is given by Lemma 2.

Lemma 2. *Let $C = (D, H)$ be a solution to the minimisation problem $G = (V, E)$. Then C is the only minimum vertex cut that satisfies the partitioning constraints and also minimises $|D|$.*

Proof. Assume for the sake of contradiction that an alternative solution $C' = (D', H')$ exists such that (D', H') is a valid partitioning of V, C' is a minimum vertex cut in G, $|D'|$ is minimum, and $C \neq C'$.

From $C \neq C'$ it follows that $D \neq D'$, and since D and D' are both minimum and thus of the same size, neither D nor D' can be a subset of the other. Their intersection $I = D \cap D'$ is not empty either as both device sets must contain all source vertices, and source vertices must exist since $C = (\emptyset, V)$ otherwise would be the only solution.

Let $J = D' \setminus I$. Since every vertex in D' by Lemma 1 can be reached by some source vertex in I, and C is the vertex cut (D, H) with $J \subset H$, then all vertices in J can be reached from vertices in $C \cap I$. We note that J cannot be empty as that would imply $D' = I$ and thus $D' \subseteq D$, which would be a contradiction.

The set $C \cap I$ can be partitioned into two sets, X and Y, such that $X \subseteq C'$ and $Y \not\subseteq C'$. It follows that no member of Y has an edge to a vertex in H', and all vertices in J can be reached via edges from vertices in Y. If the latter was not the case, then some vertices in $J \subset D'$ could only be reached from members of X, and thus C', which would contradict that C' and D' are minimum.

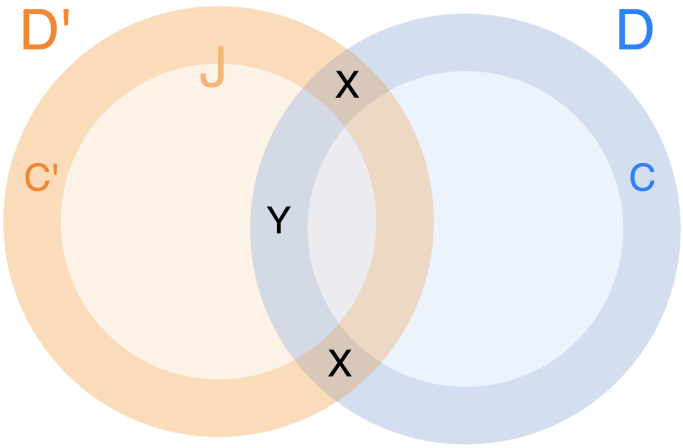


Figure 6: A visual representation of the sets D' and D.

Now consider the set of all vertices with an edge from J to H', that is $J \cap C'$, and compare its size to $|Y|$:

- If $|J \cap C'| < |Y|$ then $(D \cup J, H \setminus J)$ is a smaller vertex cut than C, which thus cannot be minimum.
- If $|J \cap C'| > |Y|$ then $(D' \setminus J, H' \cup J)$ is a smaller vertex cut than C', refuting that C' is minimum.
- If $|J \cap C'| = |Y|$ then it follows from the minimality of $|D'|$ that J must be empty.

Every possible outcome leads to a contradiction so we conclude that the initial assumption must be wrong. C is therefore the only solution to G. □

Corollary 1 and Lemma 2 imply that if $C = (D, H)$ is the solution to $G = (V, E)$ where V excludes any vertices $\hat{H}$ that cannot be reached from a source vertex, then $C' = (D, H \cup \hat{H})$ is the solution to $G = (V \cup \hat{H}, E)$, and vice versa. We can thus find a solution to the smaller problem of the two and trivially find the solution to the other.

Next, we prove that the solution to a graph without additional edges from sink-connected vertices also yields the solution to a graph where those edges and connected vertices are included. This allows exclusion of the non-sink edges.

Theorem 2. *Let $C = (D, H)$ be a solution to the graph problem $G = (V, E)$, let t be a meta-variable for any sink in V, and let $T = \{v \mid (v \to t) \in E\}$ be all vertices with an edge to a sink. Then $C' = (D, H \cup V^*)$ is the solution to the problem $G' = (V', E')$, where $V \subseteq V'$ and $E \subseteq E'$, under the assumption that no vertex in $V^* = V' \setminus V$ is a source vertex, and each edge in $E^* = E' \setminus E$ is of the form $v \to u$ for $v \in (T \cup V^*)$ and $u \in V'$.*

Proof. To prove that C' is the solution to G' we must prove three things:

1. That C' is a valid partitioning of V'.
2. That C' is a minimum vertex cut in G'.
3. That no minimum vertex cut exists in G' with a device vertex set smaller than D.

(1) Since $C = (D, H)$ is a valid partitioning of V and V^* contains no source vertices then $C' = (D, H \cup V^*)$ is trivially a valid partitioning of V'.

(2) Since $C = (D, H)$ is a minimum vertex cut in G, to prove that $C' = (D, H \cup V^*)$ is a minimum vertex cut in G' it is sufficient to show that none of the edges added by E^* is from a vertex in $X = D \setminus C$.

Let $v \to u$ be any edge in E^*. If $v \in H \cup V^*$ then it trivially holds that $v \notin X$. Otherwise $v \in D$ which implies $v \in T$ and the existence of some edge $v \to t$. Since $t \in H$ for C to be a valid partitioning of V we must have $v \in C$ and thus $v \notin X$.

We picked $v \to u$ to be any edge in E^*, so none of the edges in E^* is from a vertex in $X = D \setminus C$. Therefore C' must be a minimum vertex cut in G'.

(3) Compared to G we have only added vertices and edges to G', not removed any. Since D is minimum in regards to C it must thus also be minimum in regards to C'.

Having proved (1), (2), and (3) we use Lemma 2 to conclude that $C' = (D, H \cup V^*)$ is the unique solution to G'. □

Since (a) every scalar operand of all host-only statements receives an edge to a sink; (b) no edge exists from an array variable; and (c) none of the host-only statements that we consider have an edge from a source, then all host-only variables will be members of H if included. This insight allows all host-only variables to be excluded from the graph, and because all host-only statements for our needs and purposes are $\mathcal{E}$ statements, then they *will* be excluded, by Statement 1. Statement 2 thus need not include any constraints on the partitioning of host-only variables.

If we expand Statement 2 to constrain device-only variables to be partitioned into D we can make a similar argument in regards to the redundancy of adding intra-kernel device variables to the graph. Being device-only, all intra-kernel device variables K must be placed in D, and because no host statement can use an intra-kernel device variable as operand, it can be shown that the host set will be invariant to the inclusion of K. A full proof is not possible without first specifying graphing rules for intra-kernel device variables, which we consider to be out of scope. We therefore leave the reader with the intuition that it is the transfer of kernel results that matter, not the computations that produce them.

It is beneficial to reduce the optimisation problem by excluding vertices and edges with no impact on a solution, if not to speed up finding a solution then to reduce the space requirements for representing the graph. Reducing the graph size usually translates to more efficient lookup in underlying implementation data structures.

4.3 Routing-based migration analysis

The Statement 2 optimisation problem $G = (V, E)$ can be expressed as a max-flow problem and solved using the Ford-Fulkerson method [7], which is based on reversing edges along paths going from a single source to a single sink. By the max-flow min-cut theorem [7] solving such problem also gives a minimum (edge) cut, which can be translated into a solution to G if applied on a particularly engineered graph. The graphing of **loop** expressions that we show in subsection 4.5 however requires solving isolated subgraphs of the graph problem while building it, and so it is generally not possible to apply the method once and solve the problem in one go. The Ford-Fulkerson method *can* be used but if done naively the edges of subproblems will be revisited when parent problems are solved.

To avoid this inefficiency we propose an algorithm that can be shown to be a variant of Ford-Fulkerson for flow networks where multiple sources and sinks exist, such that attempting to find any sink from each source in isolation is as efficient as pathfinding from a single source to a single sink, irrespective of order. We will not present it as a max-flow algorithm however but describe it in terms of the concrete problem at hand.

Algorithm

The Statement 2 optimisation problem $G = (V, E)$ can be solved using an algorithm based on reversing edges along paths going from source vertices to sink vertices. We propose an algorithm where these paths are found using depth-first search (DFS), and since DFS will be repeated for each source vertex we propose caching its failures. Specifically, when DFS fails to find a sink in a subgraph reached via some edge we

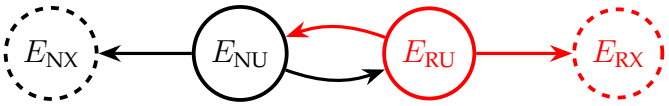


Figure 7: The possible subset memberships of an edge and transitions between them. Red states denote reversal and a dashed outline denotes exhaustion.

mark that edge as *exhausted*. No sink can be reached via an exhausted edge, and any subsequent search attempt can skip pathfinding along such edge.

In addition to marking edges as exhausted, we also mark edges that have been reversed, amounting to partitioning E into four subsets:

- E_{RU} contains the reversed edges of E that are unexhausted.
- E_{RX} contains the reversed edges of E that are exhausted.
- E_{NX} contains the non-reversed edges of E that are exhausted.
- E_{NU} contains the non-reversed edges of E that are unexhausted.

The edges described by subsection 4.1 are all non-reversed, unexhausted edges belonging to E_{NU}. The possible state transitions of an edge is given by Figure 7, where the arrow from E_{RU} to E_{NU} denotes that reversing a reversed edge restores it to its original, non-reversed state. We consider $(a \to b) \in E_{NU}$ to be distinct from $(a \to b) \in E_{RU}$, based on state, so parallel edges are allowed in G.

We define the following unions of E subsets to easily denote the presence or absence of individual marks:

- $E_R = E_{RU} \cup E_{RX}$ contains all reversed edges of E.
- $E_X = E_{RX} \cup E_{NX}$ contains all exhausted edges of E.

We define V_S to be the set of all source vertices in V, and define V_T to be the set of all sink vertices in V. We describe our algorithm in terms of a derived graph $G' = (V', E')$ created by the mapping $\mathcal{G}(G)$ but will later show how G can be operated upon directly.

Definition 2. The graph $G' = (V', E')$ derived from $G = (V, E)$ is given by $\mathcal{G}(G)$.

$$
\begin{aligned}
\mathcal{G}(G) = \ & \mathcal{G}((V, E)) = (V', E') \\
V' = \ & \{v_{in} \mid v \in V\} \cup \{v_{out} \mid v \in V\} \\
E' = \ & E'_{RX} \cup E'_{NX} \cup E'_{NU} \cup E'_{RU} \\
E'_{RU} = \ & \{a_{in} \to b_{out} \mid (a \to b) \in E_{RU}\} \cup \\
& \{v_{out} \to v_{in} \mid \exists x : (v \to x) \in E_{RU}, v \notin V_T\} \\
E'_{RX} = \ & \{a_{in} \to b_{out} \mid (a \to b) \in E_{RX}\} \cup \\
& \{v_{out} \to v_{in} \mid \exists x : (v \to x) \in E_{RX}, v \notin V_T\} \cup \\
& \{s_{out} \to s_{in} \mid \exists x : (x \to s) \in E, s \in V_S\} \\
E'_{NX} = \ & \{a_{out} \to b_{in} \mid (a \to b) \in E_{NX}\} \cup \\
& \{v_{in} \to v_{out} \mid \nexists x : (x \to v) \in E_R, \nexists y : (v \to y) \in E \setminus E_X\} \\
E'_{NU} = \ & \{a_{out} \to b_{in} \mid (a \to b) \in E_{NU}\} \cup \\
& \{v_{in} \to v_{out} \mid \nexists x : (x \to v) \in E_R, \exists y : (v \to y) \in E \setminus E_X\}
\end{aligned}
$$

The mapping $\mathcal{G}$, given by Definition 2, splits each vertex $v \in V$ into two, v_{in} and v_{out}, with a single edge connecting the two. When no edges have been reversed, all edges to v go to v_{in}, all edges from v go from v_{out}, and the edge $v_{\text{in}} \to v_{\text{out}}$ exists.

We will use v_{in} and v_{out} to denote specific vertices in V' that uniquely can be identified from the $v \in V$ that they were created from. Given v_{in} or v_{out} in V' we may likewise refer to their origin vertex $v \in V$. Based on Definition 2, to mirror the definitions made for G, we define sets of derived edges that are reversed or exhausted:

$$E'_{\text{R}} = E'_{\text{RU}} \cup E'_{\text{RX}} \qquad E'_{\text{X}} = E'_{\text{RX}} \cup E'_{\text{NX}}$$

We also define derived sets of sources and sinks, given by:

$$V'_{\text{S}} = \{s_{\text{in}} \mid s \in V_{\text{S}}\} \qquad V'_{\text{T}} = \{t_{\text{in}} \mid t \in V_{\text{T}}\}$$

The original graph $G = (V, E)$ can obtained from G' by its left inverse mapping $\mathcal{G}^{-1}(G')$.

Definition 3. The left inverse of $\mathcal{G}$ is $\mathcal{G}^{-1}$ which merges $v_{\text{in}}, v_{\text{out}}$ for each $v \in V$:

$$\begin{aligned}
\mathcal{G}^{-1}(G') &= \mathcal{G}^{-1}((V', E')) = (V, E) \\
V &= \{v \mid v_{\text{in}} \in V'\} \\
E &= E_{\text{RX}} \cup E_{\text{NX}} \cup E_{\text{NU}} \cup E_{\text{RU}} \\
E_{\text{RU}} &= \{a \to b \mid (a_{\text{in}} \to b_{\text{out}}) \in E'_{\text{RU}}\} \\
E_{\text{RX}} &= \{a \to b \mid (a_{\text{in}} \to b_{\text{out}}) \in E'_{\text{RX}}\} \\
E_{\text{NX}} &= \{a \to b \mid (a_{\text{out}} \to b_{\text{in}}) \in E'_{\text{NX}}\} \\
E_{\text{NU}} &= \{a \to b \mid (a_{\text{out}} \to b_{\text{in}}) \in E'_{\text{NU}}\}
\end{aligned}$$

The basic building block of our algorithm is ROUTE (Algorithm 1) which given some source vertex $s \in V_{\text{S}}$ attempts to find some path $p = s_{\text{in}} \rightsquigarrow t_{\text{in}}$ in $\mathcal{G}(G)$ that ends at any $t_{\text{in}} \in V'_{\text{T}}$. If found it reverses all edges along p. We call the reversed path $t_{\text{in}} \rightsquigarrow s_{\text{in}}$ a *route*, denoted $s_{\text{in}} \mapsto t_{\text{in}}$. An equivalent route $s \mapsto t$ is produced in the origin graph.

Figure 8 shows a graph after three calls to ROUTE and lists the edges that can arise between "in" and "out" vertices as a result of diverse edge configurations in E. Upon reconstructing $G'' = \mathcal{G}(G)$ from $G = \mathcal{G}^{-1}(G')$ it can be observed that some internal edges may be exhausted even though they never were traversed in G'. Since $G' \neq G''$ does not generally hold, then $\mathcal{G}$ is not an inverse mapping of $\mathcal{G}^{-1}$.

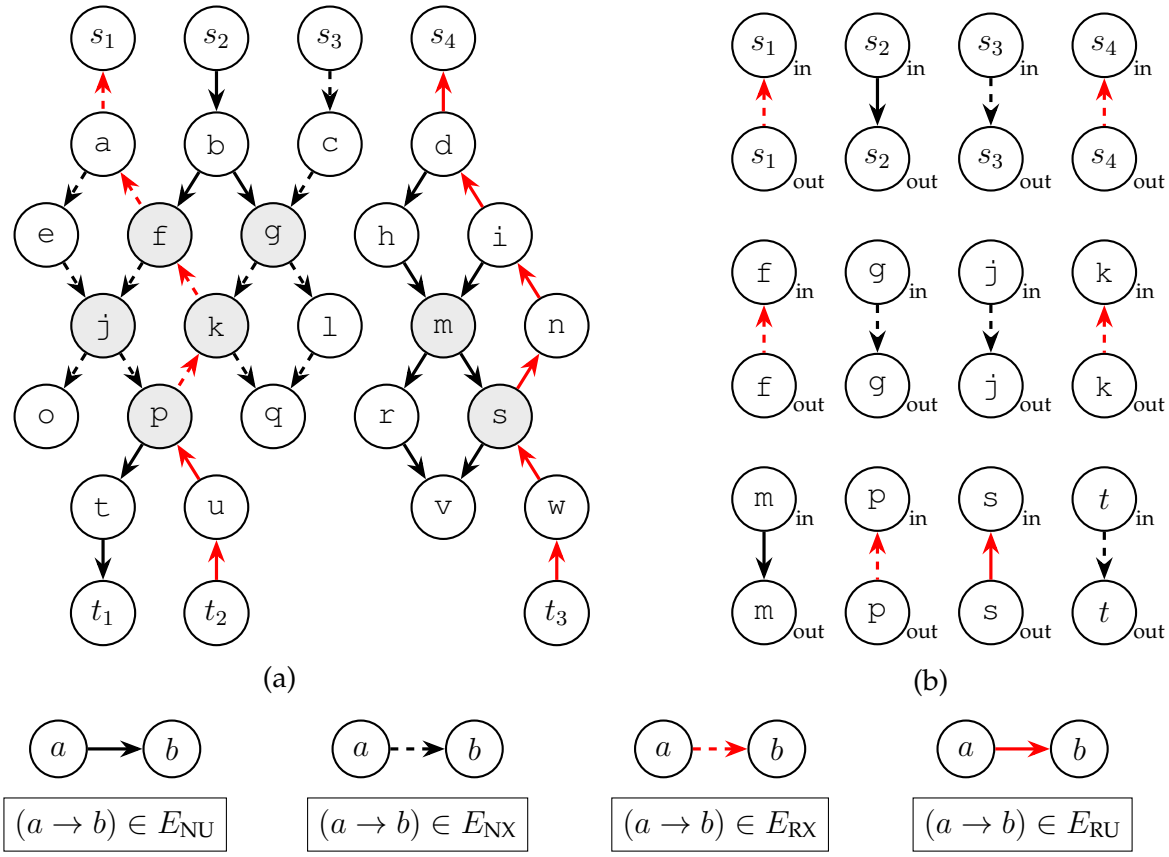


Figure 8: (a) Some graph G, (b) the internal edges between selected vertices in $\mathcal{G}(G)$, and (bottom) a legend of edge types. The routes $s_1 \mapsto t_2$ and $s_4 \mapsto t_3$ have been created in G, but the subsequent routing attempt from s_3 failed. No routing attempt has yet been made from s_2 but the efforts made to route s_3 have significantly reduced the problem to a search of just a few edges.
The highlighted vertices showcase seven distinct edge configurations that are representative of the combinations that may arise during routing. In (b) their internal edges are shown along those of the sources and sinks; the $t_{\text{in}} \to t_{\text{out}}$ edge is for all three sinks. The most interesting vertices are `m`, `f`, `g`, and `p`; the vertices `j`, `k` are minor variations.

Algorithm 1 The ROUTE function attempts to find a path in $\mathcal{G}(G)$ from the source vertex s_{in} to some sink t_{in}, reversing the edges along the path $s_{\text{in}} \rightsquigarrow t_{\text{in}}$ if found.
The $\mathcal{P}$ function used by ROUTE′ maps vertices with edges currently being searched to the depth that they were encountered. It is used to register cycles and allows detection of exiting a cyclic subgraph such that participating edges can be exhausted.

1: **function** ROUTE(G, s)
2: $G' \leftarrow \mathcal{G}(G)$
3: $s' \leftarrow$ the vertex $s_{\text{in}} \in G'.V'$ that was created from s
4: $(G', \_) \leftarrow$ ROUTE′$(G', s', \mathcal{P}, 0)$ where $\mathcal{P}(v) = \epsilon$ and $\epsilon \notin \mathbb{Z}$
5: **return** $\mathcal{G}^{-1}(G')$
6: **end function**
7:
8: **function** ROUTE′(G', v, $\mathcal{P}$, d)
9: **if** $v \in G'.V'_{\text{T}}$ **then**
10: **return** $(G',\ \text{found})$ ▷ Path found.
11: **else if** $\mathcal{P}(v) \neq \epsilon$ **then**
12: **return** $(G', (\mathcal{P}(v), \emptyset, \mathcal{P}))$ ▷ Cycle detected.
13: **end if**
14: let $\mathcal{P}'_0(u) = \begin{cases} d & \text{if } u = v \\ \mathcal{P}(u) & \text{otherwise} \end{cases}$
15: $U \leftarrow \{u \mid u \in G'.V', (v \to u) \notin G'.E'_{\text{X}}\}$ ▷ Determine adjacent vertices.
16: $(n, f) \leftarrow (|U|,\ \text{failed})$
17: **for** $k = 1, ..., n$ **do**
18: $u \leftarrow k$th vertex in U ▷ The set U is ordered.
19: $(G', r) \leftarrow$ ROUTE′$(G', u, \mathcal{P}'_{k-1}, d + 1)$
20: **if** $r =$ found **then**
21: $G' \leftarrow G'$ with the edge $v \to u$ reversed
22: **return** $(G',\ \text{found})$
23: **else if** $r =$ failed **then**
24: $G' \leftarrow G'$ with the edge $v \to u$ exhausted
25: let $\mathcal{P}'_k = \mathcal{P}'_{k-1}$
26: **else if** $r = (d_{\text{c}}, E_{\text{c}}, \mathcal{P}_{\text{c}})$ **then** ▷ Handle detected cycle.
27: $E'_{\text{c}} \leftarrow E_{\text{c}} \cup \{v \to u\}$
28: **if** $f =$ failed **then**
29: $f \leftarrow (d_{\text{c}}, E'_{\text{c}}, \mathcal{P}_{\text{c}})$
30: **else if** $f = (d_{\text{f}}, E_{\text{f}}, \_)$ **then**
31: $f \leftarrow (\min(d_{\text{c}}, d_{\text{f}}), E'_{\text{c}} \cup E_{\text{f}}, \mathcal{P}_{\text{c}})$
32: **end if**
33: let $\mathcal{P}'_k = \mathcal{P}_c$
34: **end if**
35: **end for**
36: **if** $f = (d_{\text{f}}, E_{\text{f}}, \_)$ **and** $d = d_{\text{f}}$ **then** ▷ Backtracking out of cycle.
37: $G' \leftarrow G'$ with every edge in E_{f} exhausted
38: $f \leftarrow$ failed
39: **end if**
40: **return** (G', f)
41: **end function**

When ROUTE has been invoked on every source vertex $s \in V_S$ of some complete graph G, a solution can be obtained by RESOLVE(G, V_S), given by Algorithm 2. It assigns the device set of a solution to be every $v \in V$ where v_{in} can be reached from a source vertex in $\mathcal{G}(G)$.

Algorithm 2 The function RESOLVE produces a solution $C = (D, H)$ to the optimisation problem G provided ROUTE has been invoked for every source vertex in $G.V_S$. The solution is given as (D, C, H) where C contains all vertices of the minimum vertex cut.

Require: G is the result of repeated calls to ROUTE$(_, s)$, once for each $s \in G.V_S$.

function RESOLVE(G)
  $G' \leftarrow \mathcal{G}(G)$
  $D' \leftarrow \{v \mid \exists (s_{\text{in}} \rightsquigarrow v) \in G', s \in G.V_S\}$ ▷ All vertices reachable from a source.
  $D \leftarrow \{v \mid v_{\text{in}} \in D'\}$
  $C \leftarrow D \setminus \{v \mid v_{\text{out}} \in D'\}$
  $H \leftarrow G.V \setminus D$
  return (D, C, H)
end function

SOLVE in Algorithm 3 provides a full solution by combining ROUTE and RESOLVE, and Figure 9 shows the intermediate states of a small graph as SOLVE operates upon it. While routing is done in the derived graph, for conciseness the path searches described by Figure 9 are given in terms of the origin graph. The full traversal done in $\mathcal{G}(G)$ to produce Figure 9b is

$$s_{1_{\text{in}}} \to s_{1_{\text{out}}} \to \mathtt{a}_{\text{in}} \to \mathtt{a}_{\text{out}} \to \mathtt{e}_{\text{in}} \to \mathtt{e}_{\text{out}} \to \mathtt{h}_{\text{in}} \to \mathtt{h}_{\text{out}} \to \mathtt{j}_{\text{in}} \to \mathtt{j}_{\text{out}} \to t_{2_{\text{in}}}$$

and the cycle encountered to produce Figure 9c is:

$$\mathtt{h}_{\text{in}} \to \mathtt{e}_{\text{out}} \to \mathtt{j}_{\text{in}} \to \mathtt{h}_{\text{out}} \to \mathtt{h}_{\text{in}}$$

Algorithm 3 The function SOLVE produces a solution to the graph problem G.

function SOLVE(G)
  $G \leftarrow$ ROUTE-MANY$(G, G.V_S)$
  return RESOLVE(G)
end function

function ROUTE-MANY(G, S)
  for each $s \in S$ **do**
    $G \leftarrow$ ROUTE(G, s)
  end for
  return G
end function

In the following subsections we prove the correctness of SOLVE, analyse its asymptotic time and space complexity, and provide guidance as to how it efficiently can be implemented.

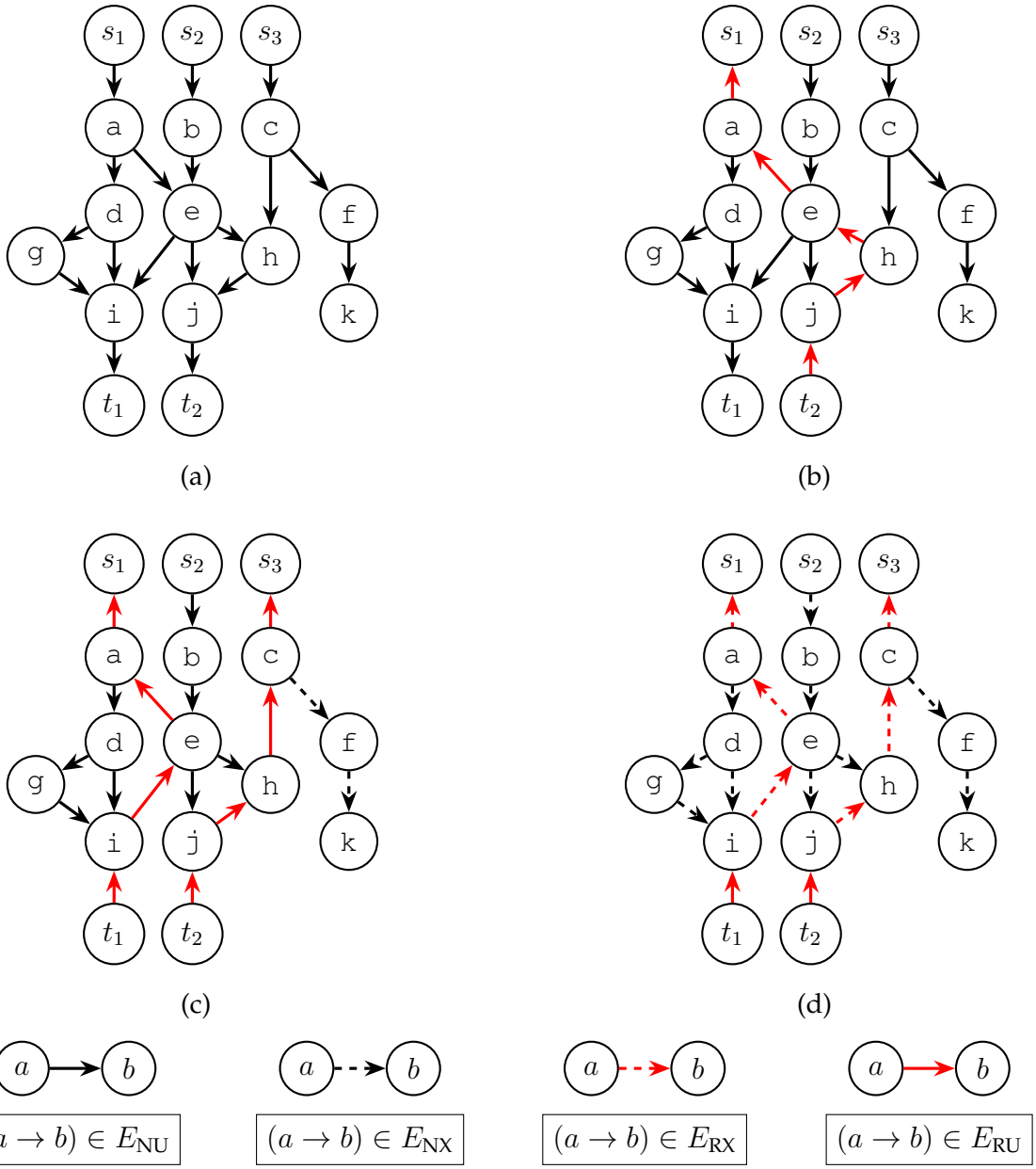


Figure 9: The intermittent states of a small graph being solved. The edges of vertices are visited in clockwise order. The sources are attempted routed in order s_1, s_3, s_2.

(a) The original graph as passed to SOLVE.

(b) A route is immediately found by traversing $s_1 \to \texttt{a} \to \texttt{e} \to \texttt{h} \to \texttt{j} \to t_2$.

(c) A sink is found by traversing reversed edges, effectively rerouting s_1 to t_1 and forming a new route $s_3 \mapsto t_2$. The search backtracks from the edges $\texttt{c} \to \texttt{f} \to \texttt{k}$, which are exhausted; encounters and escapes the cycle $\texttt{h} \to \texttt{e} \to \texttt{j} \to \texttt{h}$; and then finds a sink by traversing $\texttt{e} \to \texttt{i} \to t_1$. The cycle is not exhausted as it has edges left to explore.

(d) The routing attempt from s_2 fails. The search traverses all edges of the graph, except $\texttt{c} \to \texttt{f} \to \texttt{k}$, which are exhausted, and $\{t_1 \to \texttt{i}, t_2 \to \texttt{j}\}$ which are unreachable. The only edge that is available from $\texttt{e}_{\text{in}}$ is to $\texttt{a}_{\text{out}}$; the vertices of `j` and `h` are only available once `e` is reached via the edge $\texttt{i}_{\text{in}} \to \texttt{e}_{\text{out}}$. The edges between the vertices `e`, `a`, `d`, `g`, and `i` form two cycles which only are exhausted once search backtracks from $\texttt{e}_{\text{in}}$.

SOLVE finds the solution $(D, C, H) = (\{s_1, s_2, s_3, \texttt{a}, ..., \texttt{k}\}, \{\texttt{i}, \texttt{j}\}, \{t_1, t_2\})$ based on the vertices that can be reached from $s_{2_{\text{in}}}$.

Proof of correctness

SOLVE performs N routing attempts over N iterations to solve the graph problem $G = (V, E)$, where $N = |V_S|$. Let G'_k denote the state of the derived graph $G' = (V', E')$ after the kth iteration with $k = 1, 2, ..., N$, and let $G'_0 = \mathcal{G}(G)$ be its original state.

Each routing attempt involves a depth first search that starts from a source vertex $s \in V'_S$ and terminates at the first sink vertex $t \in V'_T$ that it encounters. This produces a *search path* upon success, denoted $p = s \rightsquigarrow t$. It follows that t is the only sink vertex in p, and since sources and sinks are distinct, $s \neq t$. This implies that p never is empty.

With search paths defined we provide Theorem 3 to prove that the exhaustion scheme of Algorithm 1 ensures that no sink can be reached via an exhausted edge.

Theorem 3. *Let $e \in E'$ be some edge $a \to b$. When* ROUTE *backtracks to a via e and all edges reachable via b have been exhausted, then e is marked exhausted. This implies that no sink is nor can become reachable via e.*

Proof. Assume that at any point in time some sink could be reached via an exhausted edge, and let t be the first such sink. Let e be some exhausted edge $a \to b$ on any path that leads to t, and let X be the set of vertices that can be reached from b by only traversing exhausted edges. We will show that t cannot exist.

No vertex can become a sink after it has been created, and an edge to some vertex u can only be marked exhausted if u is not a sink and all edges from u are exhausted. It follows that t cannot be a member of X. This implies that some vertex $v \in X$ has a non-exhausted edge g to some vertex $w \notin X$ through which t can be reached, and g must have been added after the last edge to v was exhausted. It follows that when g was added all existing edges from v was exhausted.

The algorithm never creates edges however, only reverses them, so g must have been an edge from w to v that was reversed. The only way g could be reversed however is if a search path was found via g and that is impossible since v is not a sink, all edges from v was exhausted, and ROUTE never traverses exhausted edges.

The edge g thus cannot exist, which means t cannot be reached via e and hence cannot exist either. We assumed t to be the first sink that could be reached via any exhausted edge and if the first such sink does not exist then inductively no subsequent sink can either. In conclusion it is impossible to reach a sink via any exhausted edge. □

A corollary to Theorem 3 is that failed routing attempts are irrecoverable.

Corollary 2. *Once* ROUTE(G, s) *has failed to find a search path from s, no future attempt from s can succeed, irrespective of the outcome of any* ROUTE(G, s') *call later made. All edges from s_{in} will have been exhausted, so the failure is irrecoverable.*

Next we study the properties of routes and the search paths that create them. We will use this to establish formal observations that can be used to show that Algorithm 3 is invariant to the order that source vertices are attempted routed.

A route $r = s \mapsto t$ in G'_k is a path from a source vertex s to some sink vertex t which is reached by only traversing in the opposite direction of reversed edges. Routes are formed when the edges along search paths are reversed, and they cannot share edges since reversing a reversed edge returns it to its original, non-reversed state. Every reversed edge thus belongs to exactly one route in G'_k. Figure 9 provides examples, particularly step (b) → (c) that shows one route being undone to form two new.

Because every vertex $u \in V'$ either is the v_{in} or v_{out} vertex for some $v \in V$, every route through u also goes through the internal edge that connects it with its twin. Since routes share no edges and each vertex pair only has one internal edge, then routes cannot share vertices either. Seeing that routes are created along search paths and do not overlap in any way, it follows that the end vertex t of a route is its only sink.

A search path $p = s \rightsquigarrow t$ traverses vertices via alternating segments of non-reversed and reversed edges. We define a non-empty (sub)path $a \rightsquigarrow b$ that only traverses non-reversed edges to be an *addition segment*, symbolically written $a \blacktriangleright b$, and define a non-empty (sub)path $c \rightsquigarrow d$ that only traverses reversed edges to be a *removal segment*, written $c \triangleright d$. Reversing the edges of an addition segment creates new route edges while reversing the edges of a removal segment eliminates edges from an existing route.

The first edges along any search path $s_{\text{in}} \rightsquigarrow t_{\text{in}}$ is $s_{\text{in}} \rightarrow s_{\text{out}} \rightarrow u_{\text{in}}$ for some vertex u_{in}, which means that every search path starts with an addition segment. No edge goes to s_{in} in G'_0 so no edge can exist from s_{in} if $s_{\text{in}} \rightarrow s_{\text{out}}$ is reversed.

A search path p furthermore cannot end with a removal segment. To see why, consider that a removal segment is a path in the opposite direction of some route $r = s \mapsto t$, that the only sink vertex in r is t, and that routes do not overlap. Therefore if p traverses $a \triangleright b$ then b cannot possibly be a sink vertex, and p therefore cannot possibly end there.

It follows that any search path $p = s \rightsquigarrow t$ can be generalised to be an addition segment $s \blacktriangleright a_0$ followed by $m \geq 0$ overlapping segment subsequences a_{i-1} $(\triangleright\, b_i \blacktriangleright a_i)$, where $a_m = t$. That is

$$p = s \blacktriangleright a_0\ (\triangleright\, b_1 \blacktriangleright a_1)\ \ldots\ (\triangleright\, b_m \blacktriangleright a_m) \qquad \text{for } m \geq 0 \tag{1}$$

Let $s = s_0 = b_0$. When p is reversed its ith subsequence can be understood as splitting some existing route $r_i = s_i \mapsto t_i$ into a source fragment $s_i \mapsto b_i$ and a sink fragment $a_{i-1} \mapsto t_i$, which the route fragment $s_{i-1} \mapsto b_{i-1}$ is extended to. This forms the route $s_{i-1} \mapsto b_{i-1} \mapsto a_{i-1} \mapsto t_i$. The addition segment $b_i \blacktriangleright a_i$ extends the disconnected source fragment $s_i \mapsto b_i$ to a sink, either via the sink fragment $a_i \mapsto t_{i+1}$, if $i < m$, or directly to t. Figure 9 demonstrates the reversal of $s_{1_{\text{in}}} \blacktriangleright t_{2_{\text{in}}}$ and $s_{3_{\text{in}}} \blacktriangleright \mathtt{h}_{\text{in}} \triangleright \mathtt{e}_{\text{out}} \blacktriangleright t_{1_{\text{in}}}$.

Observation 1. *Succeeding to find a search path from some source vertex $s \in V'_S$ and reversing its edges creates a route from s and possibly alters a number of existing routes.*

1. *Every source vertex with a route to a sink in G'_k is also routed to a sink in G'_{k+1}.*
2. *Every sink vertex that terminates a route in G'_k also terminates a route in G'_{k+1}.*
3. *For every iteration k that a search path is found the number of routes in G'_k will be one greater than in G'_{k-1}.*

Corollary 3. *The number of routes in G'_k increases monotonically with k.*

Whenever a search begins traversing reversed edges to find a sink it can be understood as an attempt to backtrack a previous search to connect its source differently. The route r that the previous search created blocks the path to a sink that otherwise could be reached from the current search origin s, and since it is unknown whether any free path from s to a sink remains it is worth to first check whether r can be routed differently such that the blockade disappears. If the search backtracks from the reversed edges then no alternative sink for r exists in that part of the graph and the blockade is unavoidable. Rerouting an existing route may involve rerouting other routes.

Observation 2. *Attempting to create a route from some source vertex s will fail iff*

1. *no sink vertex is reachable from s in G'_0, or*
2. *all paths to sinks reachable from s in G'_0 are blocked by other routes, and the existing routes cannot be routed differently to avoid this.*

Corollary 4. *A routing attempt from some source vertex s will fail iff there exists no way to simultaneously form routes from s and all the origins of existent routes.*

We are now ready to show that Algorithm 3 is invariant to the order that source vertices are attempted routed. Lemma 3 is lengthy, so here is a breakdown:

1. We will first partition V'_S into four subsets X_R, X_U, Y_B, Y_C and show that routing attempts from X_U always will succeed, while routing attempts from Y_C always will fail. These two subsets are thus invariant to the routing order.
2. We will then show that the successful routing from a member of X_R comes at the cost of failed routings from members of Y_B and consider alternative routing orders. We will pick a source vertex from Y_B and a particular source vertex from X_R, and show that only one of them successfully can be routed, that the one to succeed is the one to be attempted routed first, and that the outcomes of routing attempts from all other source vertices are invariant to the relative routing order of these two vertices. We then generalise this insight to any routing order, proving that the respective sizes of X_R and Y_B is invariant to the routing order.
3. We will conclude that the number of routes $M = |X_R| + |X_U|$ that can be created in G' is invariant to the routing order.

Lemma 3. *The number of routes M that can be created in G' is invariant to the order that routing attempts are made.*

Proof. Assume that a routing attempt has been made from every source vertex in G', and consider the state G'_N. The set V'_S can then be partitioned into two disjoint sets

$$X = \{s \in V'_S \mid \exists t : s \mapsto t, t \in V'_T\}$$
$$Y = V'_S \setminus X$$

such that X contains all source vertices that a route starts at, and Y is the remaining source vertices were no route starts. We further partition X into

$$X_R = \{s_2 \in X \mid \exists s_1 : s_1 \rightsquigarrow s_2, s_1 \in Y\}$$
$$X_U = X \setminus X_R$$

based on whether its source vertices can be reached from a vertex in Y, and we likewise partition Y based on whether its vertices can reach a vertex in X:

$$Y_B = \{s_1 \in Y \mid \exists s_2 : s_1 \rightsquigarrow s_2, s_2 \in X\}$$
$$Y_C = Y \setminus Y_B$$

The set Y_B is all source vertices from which a routing attempt failed due to condition (2) of Observation 2, and X_R is intuitively the set of source vertices with a route that

directly or indirectly caused these blockades. We will show that irrespective of which order the source vertices are attempted routed to create G'_N, the sizes of $X_\mathrm{R}, X_\mathrm{U}, Y_\mathrm{B}$, and Y_C remain the same.

The routing attempts from members of Y_C failed but were not blocked by any routes. This corresponds to failure by condition (1) of Observation 2, which means that no sink ever can be reached from members of Y_C, not even in G'_0. All routing attempts from members of Y_C are thus predetermined to fail and never be blocked, which implies that Y_C is invariant to the order of routing attempts.

By definition X_U contains exactly those source vertices that successfully were routed without blocking any other routing attempts from succeeding, which implies that every member of X_U is routed to a sink that no member of $X_\mathrm{R} \cup Y$ can reach in G'_0. Since the algorithm only fails a routing attempt if no alternative routing of existing routes can avoid it (Observation 2), then the members of X_U can always be (re)routed to a sink that no member of the other sets can reach, and membership of X_U is thus invariant to the routing order. It also follows that no member $X_\mathrm{R} \cup Y$ can be routed to such a sink, nor ever can be, and thus will remain excluded from X_U. The subset X_U is therefore invariant to the routing order.

Remaining are the sets X_R and Y_B, which are not invariant to the routing order. Let O denote the order that routing attempts were made to reach the state G'_N, let s' be any source vertex in Y_B, and let s be the source vertex in X_R which was routed last of all those reachable from s'. This implies that

1. the route from s, denoted r, blocked the routing attempt from s' at some vertex a.
2. the path $s' \rightsquigarrow a$ exists.
3. r is the route $r = s \mapsto a \mapsto t$ where s, a, t are distinct.
4. the only vertex shared by r and $s' \rightsquigarrow a$ is a.
5. s was attempted routed before s' was.

It also follows that the search path from s, denoted p, was of one of the following types:

1. $p = s \blacktriangleright a \blacktriangleright t$.
2. $p = s \blacktriangleright a \blacktriangleright b \triangleright c \rightsquigarrow t'$.
3. $p = s \blacktriangleright a \triangleright c \rightsquigarrow t'$.
4. $p = s \blacktriangleright b \triangleright c \rightsquigarrow t'$, and some route $\hat{r} = \hat{s} \mapsto c \mapsto b \mapsto a \mapsto t$ existed.

Now assume that some state G'_N is reached using the modified routing order R where the routing attempt from s' is postponed or advanced any number of iterations but the relative order of all other attempts remains as in O. Let $k(v)$ denote the iteration number in which the source vertex v is attempted routed under R, and let the sets X_R, Y_B, X_U, and Y_C be defined as before, albeit with vertex memberships according to R.

Since the outcome of a routing attempt is independent of any future routing attempts, the routing outcomes for all v with $k(v) < k(s')$ remain unchanged. If $k(s) < k(s')$ the attempt from s will thus still create r, and because r is the last route created that blocks s', the attempt from s' will still fail. Seeing that a failed attempt reverses no edge in G' the routing outcomes for all v with $k(v) > k(s')$ also remain the same. All

routing outcomes therefore stay the same, and no difference between O and R can be observed in G'_N.

Let us next consider the case of $k(s') = k(s) - 1$ that corresponds to when s' is attempted routed right before s:

- If the path $s' \rightsquigarrow a$ exists then one of the following search paths can be found from s', corresponding to traversing the tail of the search path that creates r when $k(s) < k(s')$:

  1. $s' \rightsquigarrow a \blacktriangleright t$
  2. $s' \rightsquigarrow a \blacktriangleright b \triangleright c \rightsquigarrow t'$
  3. $s' \rightsquigarrow a \triangleright c \rightsquigarrow t'$

- If the path $s' \rightsquigarrow a$ does not exist then some of its edges are missing. The path $s' \rightsquigarrow a$ exists when $k(s') = k(s) + 1$, so $s' \rightsquigarrow a$ must be incomplete because the edges along the search path from s, denoted p, have not been reversed.

  Since a is the only vertex that was shared by r and $s' \rightsquigarrow a$, then none of the missing edges can exist along r. It follows that p must be of type (2), (3) or (4), generalised as $p = s \blacktriangleright x \rightsquigarrow t'$, and it must be the reversal of edges along $x \rightsquigarrow t'$ that creates all the missing edges of $s' \rightsquigarrow a$.

  This implies that $x \rightsquigarrow t'$ contains the reverse of at least one edge in $s' \rightsquigarrow a$, which means that $x \rightsquigarrow t'$ and $s' \rightsquigarrow a$ share at least two vertices. Let y be the first such shared vertex that would be encountered when traversing along the path $s' \rightsquigarrow a$. Since it is the first shared vertex, the path $s' \rightsquigarrow y$ exists, and since it is shared, the path $y \rightsquigarrow t'$ also exists. The search path $s' \rightsquigarrow y \rightsquigarrow t'$ can thereby be found.

We see that the routing attempt from s' succeeds whenever $k(s') = k(s) - 1$ but that it does so by reversing edges along the search path that could be used to route s.

We will show that successfully routing s' must cause the routing attempt from s to fail but that the routing attempts from all other source vertices are unaffected. Let

- $K_\mathrm{A} \stackrel{\text{def}}{=} \{v \in X_\mathrm{R} \mid k(v) < k(s)\}$ be as determined under any R with $k(s') > k(s)$.
- $K_\mathrm{B} \stackrel{\text{def}}{=} \{v \in X_\mathrm{R} \mid k(v) < k(s)\}$ be as determined under R with $k(s') = k(s) - 1$.

When $k(s') = k(s) - 1$ the routing attempt from s' not only succeeds but also occurs after the attempt from every other v with $k(v) < k(s)$. Since the routing attempt from s' has no effect on prior attempts it follows that

$$K_\mathrm{B} = K_\mathrm{A} \cup \{s'\}$$

By Corollary 4, because K_B is proof that there exists a way to create routes from each $v \in K_\mathrm{B}$, the set $\{v \in X_\mathrm{R} \mid k(v) < k(s)\}$ remains the same under any R with $k(s') < k(s)$. When $k(s') > k(s)$ the routing attempts from all $v \in Y_\mathrm{B}$ with $k(v) < k(s)$ fail because members of K_A block their path. Seeing that $K_\mathrm{A} \subset K_\mathrm{B}$ it follows that these failures will happen no matter when s' is attempted routed.

In summary, the routing outcomes of all v with $k(v) < k(s)$, $v \neq s'$, remain the same irrespective of $k(s')$, and the routing attempt from s' succeeds whenever $k(s') < k(s)$.

Routing s' fails when $k(s') = k(s) + 1$, and by Corollary 4 this implies that there is no way to create routes from all $v \in K_\mathrm{A} \cup \{s, s'\}$, irrespective of routing order. Since

all routing attempts from $v \in K_A \cup \{s'\}$ succeed when $k(s') < k(s)$, then the routing attempt from s must fail whenever $k(s') < k(s)$.

Barring s and s', when $k(s') = k(s) - 1$ the route $r' = s' \mapsto a \mapsto t$ blocks and fails the exact same routing attempts that creating $r = s \mapsto a \mapsto t$ would have. This can be shown by considering each of the route fragments $s \mapsto a$, $a \mapsto t$, and $s' \mapsto a$:

- $s \mapsto a$: Routes from s and s' are mutually exclusive, which means that all alternative sinks that r or r' possibly can be rerouted to exists beyond a. If $r' = s' \mapsto a \mapsto t$ is created instead of $r = s \mapsto a \mapsto t$, all routes that $s \mapsto a$ would have blocked thus still cannot reach a sink.

  The change along $s \mapsto a$ would neither cause any new routes to become blocked, as blockades are caused by reversed edges and no new edge would become reversed along $s \rightsquigarrow a$.

- $a \mapsto t$: The sink fragment $a \mapsto t$ is the same irrespective of whether $s \mapsto a \mapsto t$ or $s' \mapsto a \mapsto t$ is created. How this fragment interacts with later routing attempts is thus unaffected.

- $s' \mapsto a$: Creating r not only blocks the routing attempt from s' but also all routing attempts that r' possibly could block. Therefore, when the source fragment $s' \mapsto a$ is created, no additional routes can possibly become blocked, and since creating r' blocks the only path to a sink via a like r would, then no fewer routes can become blocked either.

Since a routing attempt that would fail cannot be enabled to succeed by the prior creation of a route, the routing outcome for every v with $k(v) > k(s)$ thus remains unaffected when $k(s') = k(s) - 1$, and by Corollary 4 thus also for all $k(s') < k(s)$.

We have proved that the routing outcomes for all source vertices but s and s' remain the same irrespective of when s' gets routed, and that the first of s and s' to be attempted routed will succeed and block the other, causing it to fail. The sizes of $|X_R|$ and $|Y_B|$ thus remain unchanged no matter how much s' is postponed or advanced.

Since any routing order of $X_R \cup Y_B$ members can be created by repetitively performing this modification, and the members of $X_U \cup Y_C$ can be interspersed into this relative order anywhere, in any order without any effect on the routing outcomes, then it follows that the sizes of $|X_R|$ and $|Y_B|$ are invariant to the total routing order.

We have $M = |X| = |X_R| + |X_U|$ and thereby conclude that the number of routes M that can be created in G' is invariant to the routing order. □

Corollary 5. *M is the maximum number of simultaneous routes that possibly can exist in G', and G'_N contains that many routes.*

When Algorithm 3 is interpreted as a max-flow algorithm for flow networks with edges of equal capacity, Corollary 5 corresponds to showing that Algorithm 3 produces a maximum flow through such a network. We can therefore use the max-flow min-cut theorem [7] to prove that SOLVE produces a minimum cut in G'_N and use this as a stepping stone to prove that it also solves the Statement 2 problem G. Before providing a minimum cut proof based on the max-flow min-cut theorem we define what a cut is, and from that what a minimum cut is.

Definition 4. Let (S, T) be any partition of G' such that $V'_S \subseteq S$ and $V'_T \subseteq T$. The cut $C = (S, T)$ is then the set of edges from vertices in S to vertices in T, and a minimum cut of G' is a cut with size less than or equal to any other cut in G'.

Lemma 4. *Let $C' = (D', H')$ be a partition of V' where D' contains all vertices that can be reached from source vertices in G'_N, and $H' = V' \setminus D'$. Then C' is a minimum cut of G' where $|D'| \leq |S|$ for every minimum cut $C = (S, T)$ of G'.*

Proof. Corollary 5 states that G'_N contains the maximum number of possible routes M. This implies that no more search paths can be found, and thereby that no sink vertex can be reached from a source vertex. All source vertices of V' will hence be in D', all sink vertices will be in H', and by definition this makes C' a cut of G'_N.

Because the sink vertices are unreachable it must also be that C' is empty, which makes it minimum, and because D' contains just the vertices that can be reached from source vertices, no set of vertices can be moved from D' to H' without increasing the size of $|C'|$. This means that C' is a minimum cut of G'_N where $|D'|$ too is minimum, that is $|D'| \leq |S|$ for every minimum cut $C = (S, T)$.

A search path starts from a source vertex and ends at sink vertex. A search path found in iteration k of SOLVE thus passes through all possible cuts of G'_{k-1} an odd number of times, and when its segments are reversed, the net effect in G'_k is that the size of every cut is reduced by one.

To be specific, a search path may cross through any particular cut multiple times but the number of crosses from S to T vertices will be odd and the number of crosses from T to S will be one less. When the edges along the search path are reversed, an odd number of edges are thus eliminated from that cut but one fewer edges are also added. In total a cut of one fewer edges is produced.

Since M edges have been removed from all cuts to produce G'_N, every cut in G'_N corresponds to a cut that is M edges larger in G'_0. It follows that C' also identifies a minimum cut in $G'_0 = G'$ for which $|D'|$ remains minimum. □

Furthermore and most important, the set C' stated by Lemma 4 corresponds to a solution to the original Statement 2 problem $G = (V, E)$.

Theorem 4. *Let $C' = (D', H')$ be a partition of V' where D' contains all vertices that can be reached from source vertices in G'_N, and $H' = V' \setminus D'$. Then $C = (D, H)$ is a solution to the graph problem G, where $D = \{v \mid v_{in} \in D'\}$, the vertex members of C is $D \setminus \{v \mid v_{out} \in D'\}$, and $H = V \setminus D$.*

Proof. To prove that C is a solution to G we must prove three things:

1. That C is a valid partitioning of V.
2. That C is a minimum vertex cut in G.
3. That no minimum vertex cut exists in G with a device set smaller than D.

For each count we will use the properties of C' that were established by Lemma 4.

(1) If a vertex in V' is a source then it is the s_{in} vertex created from some source vertex $s \in V_{\text{S}}$. Similarly, if a vertex in V' is a sink then it is the t_{in} vertex created from some sink vertex $t \in V_{\text{T}}$. Since all source vertices are in D', all sink vertices are in H', (D', H') is a partitioning of V', and every $v \in V$ corresponds to exactly one $v_{\text{in}} \in V'$, then (D, H) must be a valid partitioning of V.

(2) Consider G'_0. For the cut $C' = (D', H')$ to be minimum and also minimise $|D'|$ then every edge in C' must be an internal edge between two vertices v_{in}, v_{out} created from some $v \in V$. If any of them was of the kind $u_{\text{out}} \to v_{\text{in}}$ for some $u, v \in V$ then $|D'|$ could be reduced by replacing $u_{\text{out}} \to v_{\text{in}}$ with $u_{\text{in}} \to u_{\text{out}}$.

Every edge in C' thus uniquely corresponds to a vertex $v \in V$ such that $v_{\text{in}} \in D'$, $v_{\text{out}} \in H'$, and thus $v \in D$ with one or more edges to vertices in H. If v had no edge to a vertex in H then v_{out} would only have edges to vertices in D', contradicting that C' could be minimum. The (edge) cut C' can thus be mapped to the vertex cut $C = (D, H)$, and vice versa, with $|C'| = |C|$. Since C' is minimum, so must C be.

(3) Assume that $|D|$ is not minimum. Some other minimum vertex cut $\widehat{C} = (\widehat{D}, \widehat{H})$ is then a solution to G with $|\widehat{D}| < |D|$. The cut in G' that corresponds to $\widehat{C}$ is $\widehat{C}' = (\widehat{D}', \widehat{H}')$, given by $\widehat{D}' = \left\{v_{\text{in}} \mid v \in \widehat{D}\right\} \cup \left\{v_{\text{out}} \mid v \in \widehat{D} \setminus \widehat{C}\right\}$ and $\widehat{H}' = V' \setminus \widehat{D}'$. All source vertices are in $\widehat{D}'$, all sink vertices are in $\widehat{H}'$, and the size of $\widehat{D}'$ is

$$\begin{aligned} |\widehat{D}'| &= 2|\widehat{D}| - |\widehat{C}| \\ &= 2|\widehat{D}| - |C| \\ &< 2|D| - |C| \\ &= |D'| \end{aligned}$$

which implies that C' did not minimise $|D'|$. This is a contradiction, so the assumption must be false, and $|D|$ must be minimum.

Having proved (1), (2), and (3) we conclude that $C = (D, H)$ is a solution to G. It is evident from (2) that the vertex members of C is $D \setminus \{v \mid v_{\text{out}} \in D'\}$. □

We note that RESOLVE of Algorithm 2 computes the exact sets D', D, C, H stated by Theorem 4, assuming its argument to be $\mathcal{G}^{-1}(G'_N)$, which indeed is what SOLVE passes. SOLVE(G) thus correctly computes a solution to the problem instance G of Statement 2.

Time and space complexity

Because the exhaustion and cycle detection scheme ensures that no edge is visited twice by the same routing attempt, in the worst case a single ROUTE call can be done in $O(E)$ time. We describe the techniques and data structures that allow this in section 4.3.

Only $|C|$ paths can be found, so each edge can only be traversed and reversed $|C|$ times before being exhausted. After $|C|$ paths have been found, each edge will thus at most be traversed once more for a total of $|C|+1$ times. This implies that SOLVE always terminates, with a time complexity of $O(E\,C)$. The size of C is bounded by

$$0 \leq |C| \leq \min(|V_{\text{S}}|, |V_{\text{T}}|) \tag{2}$$

with $|V_{\text{S}}| > |V_{\text{T}}|$ being typical for Futhark programs[4]. A graph will often be relatively small as a result of insignificant variables being excluded, and the graph will usually consist of multiple strong components. In practice the cost of each routing attempt is thus limited by the size of the most heavily connected component in G.

[4]The observations we make here are based on *futhark-benchmarks* [8].

The space complexity of the algorithm is $O(V + E)$, accounting for the space required to maintain a call stack, track cycles, and delay the exhaustion of participating cycle edges. Since no vertex in G is without an edge, and every edge is shared by two vertices, it holds that $|V| \leq 2|E|$. This means that the space complexity of the algorithm can be expressed as $O(E)$.

Efficient implementation

The outgoing edges from a vertex can be stored as linked adjacency lists, one for each possible edge state (recall Figure 7). This allows storage, iteration, and state change of edges at a constant cost per edge. If the exhaustion of some edges in an adjacency list are delayed until a cycle is exited, then all remaining edges of that list will be delayed and exhausted at once, voiding the need for random access. Since routes do not overlap, at most one outgoing edge from a vertex can be in E_{R}, and so the linked lists for E_{RU} and E_{RX} can simply be a single edge and a flag stating whether that edge is exhausted.

Vertices can be stored in an array, hash table, or other data structure that allows lookup in constant time. The function $\mathcal{P}$ can be implemented in a similar fashion with updates done in-place within a single, mutable data structure. The number of vertices is fixed and known, allowing efficient allocation.

The algorithm can work directly on G without manifesting the derived graph G'. The underlying idea is to track which kind of edge a vertex v is visited by, and based on that determine whether the corresponding v_{in} or v_{out} vertex is visited.

- If v is accessed by a non-reversed edge, then v_{in} is visited. If v has an outgoing edge e in E_{R} then e is the only accessible edge as the edge $v_{\text{in}} \to v_{\text{out}}$ must be reversed, and no two routes share a vertex. If e does not exist, then the edges in the E_{NU} and E_{NX} adjacency lists are available.
- If v is accessed by a reversed edge, then v_{out} is visited. Because a reversed edge exists to v_{out}, then the edge $v_{\text{out}} \to v_{\text{in}}$ must exist in E_{R}. This means that all edges of v are accessible.

The state of the internal edge between v_{in} and v_{out} can thus be determined solely from the adjacency lists of v and the edge used to visit v. Furthermore, because

- the edge $v \to t$ is the only edge to some sink t and the only edge from v,
- $v \to t$ becomes reversed and t unreachable when a route is created to t, and
- $t \in H$ by definition,

then t need not be manifested in the graph. It is sufficient to mark that v is connected to some sink and then terminate pathfinding when finding a vertex with such mark.

RESOLVE can be implemented as a single traversal of G by using a visitation set to avoid traversing an edge twice and avoid cycles. Any vertex that can be reached will be in D, and any vertex with an outgoing edge in E_{R} that only is visited by a non-reversed edge belongs to C.

4.4 Migration

The graph representation G of a top-level function f and its partition $(D, C, H) = \text{SOLVE}(G)$ provides a model of which statements in f that should be migrated to minimise inter-device memory transfers. To optimise f based on this model we propose a simple approach where each statement flagged for migration first is transformed into a standalone **gpu** kernel, and when all flagged statements have been migrated, a second program transformation merges **gpu** kernels to produce larger kernels.

The purpose of the merge transformation is to reduce the overhead of running kernels and to keep intermediate results in GPU registers, reducing memory transactions. GPU memory transactions are much cheaper than synchronous inter-device memory transfers but more expensive than any operation that only involves GPU registers [3]. A benefit of the two-phase design, in addition to its simplicity, is that it also optimises any existing **gpu** kernels present in f.

We migrate statements into single-threaded rather than multi-threaded kernels to preserve compute and memory resources. Most statements which depend on device-host memory transfers are simple scalar operations that cannot benefit from a multi-threaded setup. The drawback then is that all work must be done by that single thread, and so it is inefficient to migrate statements that otherwise could be computed in parallel. These and other limitations that we later describe can be solved by a richer language, amounting to more advanced code generation in Futhark, but for the scope of this thesis we consider them language restrictions to work around.

Migration of non-kernel $\mathcal{E}$ statements

We show how each kind of statement is migrated in subsection 4.5 but the overall technique can be described in terms of some non-kernel $\mathcal{E}$ statement S of the form **let** $x_1, ..., x_n$ **=** e. Assume that $x_1 \in D$, which implies that S should be migrated. Let $o_1, ..., o_k$ be the $k \geq 0$ scalar variable operands of e that have been migrated to device and no longer are available to the host, and let o'_i be a single-element array that stores the value of o_i. Furthermore let $a_1, ..., a_r$ be the subset of $x_1, ..., x_n$ for which $a_i \in C$ or $\tau(a_i) = [z]\tau$, and let a'_i denote the array with an outer dimension of one that will be made to store the device computed value of a_i. Then S is migrated by rewriting it from

```
let x_1, ..., x_n = e
```

into

```
let x'_1, ..., x'_n = gpu let o_1 = o'_1[0]
                          ...
                      let o_k = o'_k[0]
                      let x_1, ..., x_n = e
                       in x_1, ..., x_n
let a_1 = a'_1[0]
...
let a_r = a'_r[0]
```

Listing 3: Generic transformation to migrate some non-kernel $\mathcal{E}$ statement.

such that all operands of e are bound before it is evaluated, and all variables of S that still are to be used on host, or that are arrays, are rebound on the host. It holds that $a'_i \in \{x'_1, ..., x'_n\}$, and thus that a_i rebinds the value assigned to one of $x_1, ..., x_n$.

```
let a = A[0]
let b = A[1]
let c = a + b
let B = [a, c]
```

```
let a' = gpu let a = A[0] in a
let b' = gpu let b = A[1] in b
let c' = gpu let a = a'[0]
             let b = b'[0]
             let c = a + b in c
let B' = gpu let a = a'[0]
             let c = c'[0]
             let B = [a, c] in B
let B  = B'[0]
```

Figure 10: Concrete migration of four non-kernel $\mathcal{E}$ statements. Top: Non-transformed code where `a`, `b`, `c`, `B` $\in D$, implying that their statements should be migrated. Bottom: Rewritten code where the four statements have been rewritten into **gpu** kernels, such that all variables but `B` are made unavailable to the host but no inter-device memory transfer occurs.

Figure 10 provides a concrete example that involves four non-kernel $\mathcal{E}$ statements.

Rewriting of kernel statements

Kernel statements are generally not migrated but may depend upon host variables that are. The program transformation must therefore rewrite kernels such that any migrated variable is rebound before attempted use.

Pick any of the three kernel constructs of our language

- **map** $(\lambda\ x_1 \ldots\ x_n \rightarrow e)\ x$
- **reduce** $(\lambda\ x_1 \ldots\ x_n \rightarrow e)\ se\ x$
- **gpu** e

and consider its term e. Due to Assumption 1 it will be of the form:

$$stm_1 \ldots\ stm_m\ \textbf{in}\ se_1, \ldots, se_r \qquad \text{or just} \qquad se_1, \ldots, se_r$$

Let m be the number of statements in e, with $m = 0$ if e is of the form $se_1, \ldots, se_r$. Let $o_1, \ldots, o_k$ be the $k \geq 0$ scalar variable operands of statements and expressions nested in e where o_i has been migrated to device and no longer is available to the host, and let o_i' be the single-element array that stores the device computed value of o_i.

```
def sum_cmp = λA →
  let S = reduce (λa, b → let p = a+b in p) 0 A
  let s = S[0]
   in map (λx → let c = x ≤ s in c) A
```

```
def sum_cmp = λA →
  let S  = reduce (λa, b → let p = a+b in p) 0 A
  let s' = gpu let s = S[0] in s
   in map (λx → let s = s'[0]
                let c = x ≤ s in c) A
```

Figure 11: Concrete kernel rewrite of a **map** statement. Top: The `sum_cmp` function from Figure 4 where `s` is the only variable in D. Bottom: Transformed function where **let** `s = S[0]` has been migrated and the **map** kernel has been rewritten accordingly.

If $k = 0$ then e is not rewritten. Otherwise e is rewritten into

```
let o_1 = o'_1[0]
...
let o_k = o'_k[0]
stm_1
...
stm_m
 in se_1, ..., se_r
```

Listing 4: Generic rewrite of the inner expression of a kernel.

where stm_1 ... stm_m only occurs if $m > 0$.

Figure 11 provides a concrete example of how the inner expression of a **map** kernel is rewritten to deal with a migrated operand. It can be observed that the transformation is suboptimal however, as it makes the **map** kernel use a copy of `S[0]` rather than `S[0]` itself. The copy is unnecessary, so the **gpu** kernel that is executed to produce it is wasteful. A functional equivalent that avoids this work is given by Listing 5.

```
def sum_cmp = λA →
  let S = reduce (λa, b → let p = a+b in p) 0 A
   in map (λx → let s = S[0]
                let c = x ≤ s in c) A
```

Listing 5: Optimal rewrite of `sum_cmp` from Figure 4, such that `S[0]` is not copied.

In subsection 4.5 we discuss why we migrate array reads such as s, but for now we note that the inefficiency mostly will manifest when kernel expressions are the only dependents of such read. The merge transformation that we present next will typically make its overhead insignificant, if not eliminate it entirely.

The merge transformation

Consider the following sequence of statements that corresponds to migrating two statements to device, with some variables of the former statement being rebound.

```
let x'_1, ..., x'_n = gpu stm_1 ... stm_p
                      in se_1, ..., se_n
let a_1 = a'_1[0]
...
let a_r = a'_r[0]
let x'_{n+1}, ..., x'_{n+m} = gpu stm_{p+1} ... stm_{p+q}
                              in se_{n+1}, ..., se_{n+m}
```

We have x'_i = $[se_i]$, $a'_j \in \{x'_1, ..., x'_n\}$ and thus that a_j equals one of $se_1, ..., se_n$. This allows the two **gpu** kernels to be merged, even if the latter kernel depends on a_j or some x'_i bound by the former statement, as demonstrated by this basic rewrite:

```
let x'_1, ..., x'_{n+m} = gpu stm_1 ... stm_p
                          let x'_1 = [se_1]
                          ...
                          let x'_n = [se_n]
                          let a_1 = a'_1[0]
                          ...
                          let a_r = a'_r[0]
                          stm_{p+1} ... stm_{p+q}
                           in se_1, ..., se_{n+m}
let a_1 = a'_1[0]
...
let a_r = a'_r[0]
```

Listing 6: Generic merging of two **gpu** kernels.

The merged **gpu** kernel performs redundant work, but this can easily be remedied with simplification rules driven by the following observations, subject to Assumption 2.

1. Any usage of a_j by $stm_{p+1} ... stm_{p+q}$ can be replaced with some se_i. This allows each introduced **let** $a_j = a'_j[0]$ kernel statement to be eliminated.

2. Any usage of a variable x bound by some statement **let** x = $x'_i[0]$ in $stm_{p+1} ... stm_{p+q}$ can be replaced with se_i. This allows the **let** x = $x'_i[0]$ statement and thus a memory transaction to be eliminated. If x'_i has no other uses, which is the case when both kernels were introduced by migration, then **let** x'_i = $[se_i]$ can also be removed.

3. Any host variable $x'_1, ..., x'_n$ or $a_1, ..., a_r$ that only was used by the second **gpu** kernel can be eliminated. If the host variable x'_i is eliminated, then the corresponding term se_i must also be eliminated from $se_1, ..., se_{n+m}$. This can lead to a reduction in array allocations and memory transactions that store results.

4. When the merged kernels have been introduced by migration it is quite possible that they read the same element from some array, corresponding to a shared dependency on some scalar variable whose computation has been moved to device. By applying CSE (common subexpression elimination) those duplicate computations can be eliminated, further reducing memory transactions.

Assumption 2. For the sake of simplicity we assume that the second kernel of a merge does not contain an x **with** [*dims*] ← *se* statement that invalidates any of the bindings $x'_1, ..., x'_n$ introduced by the former statement.

Assumption 2 can be made guaranteed to hold by making merges conditional upon it.

Since two **gpu** kernels can be merged, logically any sequence of **gpu** kernels with interspersed host bindings can be merged. The merge transformation would thus merge all four kernels of Figure 10 into one, producing the program shown in Listing 7.

```
let B' = gpu let a = A[0]
             let b = A[1]
             let c = a + b
             let B = [a, c] in B
let B  = B'[0]
```

Listing 7: The merge transformation applied to the migrated statements of Figure 10.

We assume that `a`, `b`, and `c` only were used to compute `B`, meaning that no other statement depends on `a'`, `b'`, and `c'`. This allows them to be eliminated after the merge, thus eliminating the wasteful copies that migrating **let** `a = A[0]` and **let** `b = A[1]` caused.

Reordering statements to maximise merging

Some **gpu** kernels within a statement sequence $stm_1 \ldots stm_n$ may be separated by other statements than mere host rebindings of computed results. To minimise the total number of kernels within a sequence the merge transformation thus reorders the statements, clustering them into alternating groups of non-**gpu** and **gpu** expressions, subject to dependency constraints.

Imagine an infinite sequence of buckets (Figure 12), incrementally numbered from zero, with every bucket acting as a FIFO queue of ordered statements. Let every even bucket be designated to contain non-**gpu** statements, and let every odd bucket be designated to contain **gpu** statements. The reordering goal of the merge transformation is then to place every statement stm_i of the sequence $stm_1 \ldots stm_n$ into a bucket, such that

1. every bucket used only contains statements that it is designated for.
2. no statement is placed before the binding of one of its operands, unless the binding is of a **gpu** computed result and the statement is placed in the same bucket as that **gpu** kernel. This is only possible if the given statement is a **gpu** statement.
3. no statement that invalidates variables is placed before a statement that uses such variable. Only statements that evaluate x **with** [*dims*] ← *se* expressions do this in the language we study.
4. no **gpu** statement invalidates the binding of another statement in its bucket.
5. as few buckets contain **gpu** statements.

With such grouping of statements, the kernels of each **gpu** bucket can be merged into a single kernel, and an updated sequence of statements with a minimum of **gpu** kernels can be obtained by emptying the buckets in their incremental order. Restriction (4)

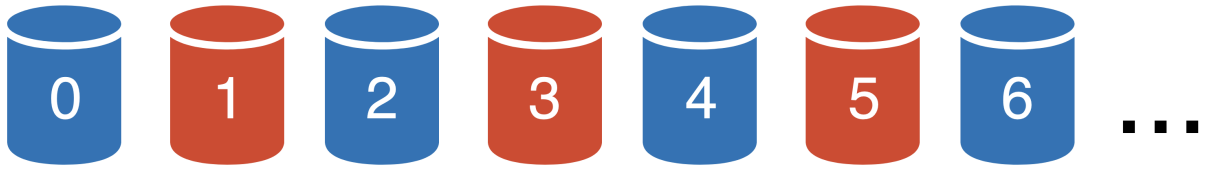


Figure 12: An infinite sequence of buckets.

ensures that Assumption 2 will hold when kernels are merged. The original sequence $stm_1 \ldots stm_n$ already abides to restrictions (2) and (3).

Let B_i be the variables bound by statements in the ith bucket, let F_i be all the free variables of statements in the ith bucket, and let $R_i \subseteq B_i$ be all variables that are rebindings of results computed by **gpu** kernels in the $(i-1)$th bucket. Let C_k be the set of variables invalidated by the kth statement in the sequence $stm_1 \ldots stm_n$, and let O_k be the set of all its free variables (its operands). Algorithm 4 then gives a solution to the reordering problem.

Algorithm 4 A solution to the reordering problem.

for $k = 1, \ldots, n$ **do**
  $i \leftarrow \infty$
  blocked $\leftarrow$ false
  while $i > 0$ **and not** blocked **do**
    $i \leftarrow i - 1$
    blocked $\leftarrow B_i \cap O_k \neq \emptyset$ **or** $F_i \cap C_k \neq \emptyset$ ▷ Ensure (2) and (3).
  end while
  if stm_k is a **gpu** statement **then**
    if i is even **then** ▷ Ensure (1).
      if $i \neq 0$ **and** $B_i \cap O_k \subseteq R_i$ **and** $F_i \cap C_k = \emptyset$ **then** ▷ Exception of (2).
        $i \leftarrow i - 1$
      else
        $i \leftarrow i + 1$
      end if
    else if $F_i \cap C_k \neq \emptyset$ **then**
      $i \leftarrow i + 2$ ▷ Ensure (4).
    end if
  else
    $i \leftarrow 2 \lfloor (i+1)/2) \rfloor$ ▷ Ensure (1).
  end if
  place stm_k in the ith bucket
end for

Algorithm 4 is a greedy algorithm that moves each **gpu** statement across as many groups of non-**gpu** statements as possible, and each non-**gpu** statement across as many groups of **gpu** statements as possible, subject to restrictions (2)–(4). Since no statement can be moved to a prior group in the produced sequence, its number of groups is minimum. It follows that when each **gpu** group is merged, the number of **gpu** kernels in the produced sequence becomes minimum.

The results produced by the merge transformation are good but not optimal. A

more complicated kernel merge procedure can remove the need for Assumption 2 and restriction (4), which for some programs can lead to a reduction in **gpu** kernels. Should such a reduction occur within a tight loop, the improvement may be noteworthy.

The algorithm does neither consider the number of results produced by each **gpu** kernel. This means that other minimum solutions may exist where merged kernels return fewer results, such that fewer allocations and memory transactions occur.

Reducing the cost of wasteful copying

Recall that the scalar read **let** `s = S[i]` will be rewritten to **let** `s' = gpu S[i]` if $s \in D$ even if `S[i:i+1]` remains available for use by all migrated statements that depend on `s`. If any host computation c depends on `s` and is migrated to device, then Algorithm 4 will ensure that they are merged unless c has some other indirect dependency on `s` via a non-**gpu** statement. If all dependents of s are merged into the same kernel, then `s'` will be eliminated. Otherwise, since c depends upon `s` and `S[i]` thus must be read, the only overhead that their merged kernel incurs is a memory transaction to write `s` to `s'`. The noticeable overhead to schedule and run a kernel is paid for by the work of c, which is assumed to be useful since it depends upon a read.

More generally the majority of the overhead associated with producing `s'` will be negated if **gpu** `S[i]` is merged with any kernel that does useful work. The host will incur some overhead in order to allocate and prepare `s'` but often such work will occur while the GPU performs work, and the work of the host will not be part of the critical execution path. Since `s'` is a single-element scalar array that most likely can be allocated from the free list, its allocation can mostly be assumed to be insignificant.

4.5 Graph creation and migration rules

In this section we provide explicit graph creation and migration rules for each section 3 statement/expression pair that Assumption 1 allows.

We give a generic template for each kind of statement S where the variables it binds are denoted $x_1, ..., x_n$ and its operands are $o_1, ..., o_m$. In some cases we have $m = n$. We provide a template for how S is transformed when migrated and describe the criteria for its migration, usually that $x_1 \in D$. The basis for the migration is some representation in the migration graph, which we also show.

When S is graphed, each operand o_i shown in the graph representation of S might not exist, or it might be array-typed or have an edge to a sink. In either case the edge(s) shown from o_i should not be added when graphing S. Unless otherwise noted, each variable x_i bound by S should only be added to the graph if it actually would receive one of the edges shown. We use s to denote a source vertex, and t to denote a sink.

In the migration template we use o'_i to denote the single-element array that stores o_i, assuming o_i has been migrated to device and no longer is available on the host. Since migrated arrays are rebound on host, as discussed in section 4.4, o'_i never stores an array. If o_i is a constant or is bound on the host, the corresponding **let** $o_i = o'_i[0]$ statement should be excluded when rewriting S. For conciseness we do not show the **let** $x_i = x'_i[0]$ statements that should be added after a **gpu** statement to rebind migrated variables that are in C or which are arrays. We refer the reader to section 4.4 for details.

Direct assignment

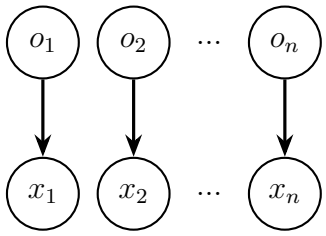
A statement that evaluates a $se_1, ..., se_n$ expression corresponds to n statements that each binds one variable. In that view such statement can be graphed and migrated as n individual $\mathcal{E}$ statements, such that each variable x_i receives an edge from operand o_i, and each assignment only is migrated if $x_i \in D$. It is beneficial to migrate the assignments separately to allow independent reordering by the merge transformation.

In the actual intermediate representation of Futhark the statement is restricted to $n = 1$, which allows it to be handled as any $\mathcal{E}$ statement. Even so the statement will never actually occur as prior program transformations have eliminated x_i, replacing any usage of x_i with o_i. A wasteful copy of o'_i will thus not occur.

For completeness we present graphing and migration rules below.

```
let x_1, ..., x_n = o_1, o_2, ..., o_n
```

```
let x'_1 = gpu let o_1 = o'_1[0] in o_1
let x'_2 = gpu let o_2 = o'_2[0] in o_2
...
let x'_n = gpu let o_n = o'_n[0] in o_n
```

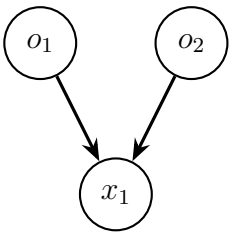


Scalar operations

Any variable x that binds the result of a basic scalar computation is graphed with an edge from each operand. The computation is migrated when $x \in D$ and rewritten as any other non-kernel $\mathcal{E}$ statement. Below we give explicit rules for $+$, $\leq$, and $\neg$ statements. They are indicative of how to handle any binary or unary scalar operation.

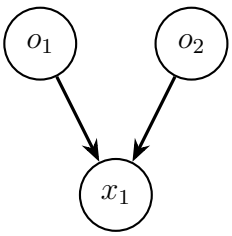
```
let x_1 = o_1 + o_2
```

```
let x'_1 = gpu let o_1 = o'_1[0]
               let o_2 = o'_2[0]
               let x_1 = o_1 + o_2 in x_1
```



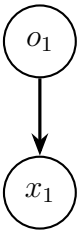

```
let x_1 = o_1 ≤ o_2
```

```
let x'_1 = gpu let o_1 = o'_1[0]
               let o_2 = o'_2[0]
               let x_1 = o_1 ≤ o_2 in x_1
```



```
let x1 = ¬o1
```

```
let x1' = gpu let o1 = o1'[0]
              let x1 = ¬o1 in x1
```

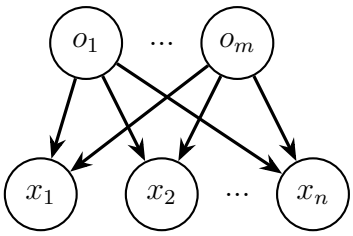


Function applications

The application of some function f produces n values from m operands, leading to the binding of n variables. Since applying f is a single unit of work, if any of the operands are migrated and remains on device then all n values must be computed and be bound on device. Migrating a function application to avoid one memory transfer thus gives n new subproblems to consider, as reflected by the graph representation below.

```
let x1, x2, ..., xn = f o1 ... om
```

```
let x1', x2', ..., xn' =
    gpu let o1 = o1'[0]
        ...
        let om = om'[0]
        let x1, x2, ..., xn = f o1 ... om
         in x1, x2, ..., xn
```



Theorem 1 guarantees that all variables of a statement will be partitioned the same, so we determine migration based on the first variable being bound, that is if $x_1 \in D$.

Not all function applications can be migrated however. If the definition of f uses any host-only statements then the application of f is also host-only[5], and we connect each scalar operand o_i to a sink. This makes the variables $x_1, x_2, ..., x_n$ unreachable from a source, excluding them from the graph.

For more fine-grained migration it is often possible to inline the statements of the function being applied, or to use function derivations that are specialised to which arguments that reside on device. We get fine-grained handling for free as the Futhark compiler aggressively inlines user-defined functions to the point where only use of simple, built-in math functions (`pow`, `sqrt`, ...) typically remain.

Array scalar reads

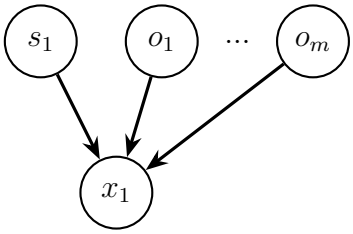
The statement **let** $x_1 = x_2[o_1, ..., o_m]$ evaluates an instance of the general indexing expression x_2[*dims*] where none of the terms in *dims* identifies a slice. We assume that m equals the rank of the array bound to x_2 such that the produced value is a scalar.

Operationally such instance causes a device-host memory transfer when evaluated on the host, for which reason we connect a source to x_1. We also add edges to x_1 from each of the indexing operands. The statement is migrated if $x_1 \in D$.

[5] In reality any function that declares an array-typed variable is host-only under Futhark, even function arguments that are unused. This is due to memory allocation limitations.

```
let x1 = x2[o1, ..., om]
```

```
let x1' = gpu let o1 = o1'[0]
              ...
              let om = om'[0]
              let x1 = x2[o1, ..., om] in x1
```



It can be observed that x_1' equals the slice $x_2[o_1 : o_1+1, ..., o_m : o_m+1]$ and thus may be wasteful to copy. There are three reasons why we migrate x_1 and create x_1' anyway:

1. An *x* **with** [*dims*] ← *se* statement may invalidate x_2. If x_1 is used beyond that point, a copy is required.

2. By making a copy we do not keep any references to x_2, possibly allowing its memory to be reused earlier. This is beneficial to avoid out-of-memory failures but also enables more allocations to be served by the free list. Allocation requests that cannot be served from the free list are expensive and may entail synchronisation, which we seek to avoid.

3. If any of the operands have been migrated, then the slice cannot be created on host. The slice equivalent also cannot be returned from a **gpu** kernel, as that would copy it. Avoiding the copy would thus entail duplicating the computation of x_1 to every site where x_1 is used. If x_1 is used as operand by some other scalar array read, then x_1 would also need to be recomputed at each of their usage sites.

It is possible to device a more advanced transformation that takes these constraints into account to introduce fewer wasteful copies. Seeing that the merge transformation presented in subsection 4.4 often makes the overhead of wasteful copies insignificant we have not deemed this venue to be worth exploring, expecting the complexity of a solution to outweigh its gains.

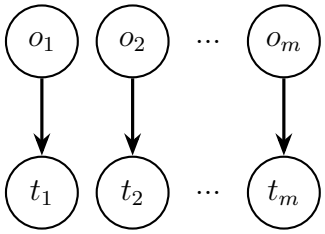
Array slicing

The statement **let** x_1 **=** x_2[*dims*] produces a slice of the array x_2 whenever *dims* contains fewer terms than the rank of x_2 or at least one term is of the type se_1 : se_1 + se_2. Let S be a concrete instance of such statement, let $o_1, ..., o_d$ be its operands that correspond to se_2 terms, and let $o_{d+1}, ..., o_m$ be all its other operands.

The statement S executes in constant time as it merely creates a view of the memory that backs x_2. Since a **gpu** kernel copies its return values into arrays we must only migrate S if we can guarantee that x_1 will not be returned. Otherwise, if such copy were to happen we would have changed the asymptotic cost of S from constant to linear, and we might run out of memory if enough simultaneous copies are created.

Our model is not strong enough to describe this constraint and so we generally must assume that if S is migrated then x_1 will be returned from some kernel produced by the merge transformation. For this reason we connect each indexing operand of S to a sink, thus ensuring that S is not selected for migration.

let $x_1 = x_2[dims]$
Not migrated on its own.



Note that S is *not* host-only; it is perfectly fine to evaluate on device, provided we can guarantee that its slice is not returned by a kernel. When considering the migration of loops and **if** statements this can sometimes be guaranteed, allowing those parent statements to be migrated along with S and other child statements. We delay a discussion of the implications of such parent migration but note that the operands $o_1, ..., o_d$ are *size variables*, as they appear in the type of x_2. This has implications for memory management and the conditions under which a parent statement can be migrated.

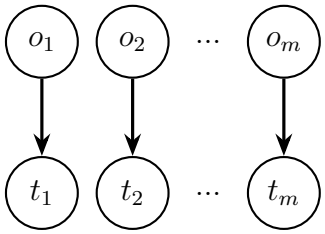
When x_1 would contain just a single array element, implying that all its dimensions are of length one, then S can be graphed with an edge from every operand to x_1 and be migrated like any other $\mathcal{E}$ statement. Such array costs as much to copy as a scalar.

Array scalar writes

The statement **let** x_1 **=** x_2 **with** $[o_1, ..., o_m] \leftarrow o_{m+1}$ evaluates an instance of the general expression x_2[*dims*] $\leftarrow$ *se* where none of the terms in *dims* identifies a slice. We assume that m equals the rank of the array bound to x_2, such that o_{m+1} is a scalar.

Operationally such an instance performs an in-place update of x_2 in constant time, which causes a host-device memory transfer when evaluated on the host. Because the cost of the operation is sublinear to the size of the updated array we must ensure that x_1 is not copied by being returned from a **gpu** kernel. This is very similar to situation just discussed for array slice statements, and the solution is the same; we connect each indexing operand to a sink.

let $x_1 = x_2$ **with** $[o_1, ..., o_m] \leftarrow o_{m+1}$
Not migrated on its own.



The observant reader will note that no sink is added to o_{m+1}, which means that migration may make it unavailable. Should this happen however, then the array o'_{m+1} that contains o_{m+1} will instead be available, so we can rewrite the statement into:

let $x_1 = x_2$ **with** $[o_1, ..., o_m : o_m + 1] \leftarrow o'_{m+1}$

This turns the scalar write into a single-element slice write, which allows us to keep o_{m+1} on device. Even better it turns the synchronous host-device memory transfer of o_{m+1} into an asynchronous intra-device memory transfer. As discussed in subsection 2.2 our hope is that this often will happen such that the write aspect of the state-

ment need not be considered. We disregard the case where o_{m+1} is a constant since some implementations of the runtime can transfer those asynchronously.

We note that the Futhark compiler already contains a form of this optimisation that performs the rewrite when o_{m+1} is the result of an array scalar read and o'_{m+1} can be represented as some slice of that array. The array o'_{m+1} in our case will thus not be a wasteful copy that was created by migrating a scalar array read.

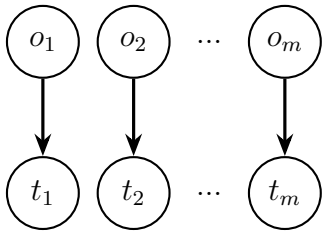
The Futhark compiler also contains optimisations that eliminate writes to single-element arrays. We therefore do not provide alternative graphing and migration rules for the case where x_1 could be copied at the cost of a scalar.

Array slice writes

The statement **let** x_1 = x_2 **with** [*dims*] $\leftarrow$ x_3 evaluates an instance of the general expression x_2[*dims*] $\leftarrow$ *se* where x_3 binds an array. Operationally x_2 is updated in-place using one or more asynchronous intra-device memory transfers that copy data from x_3.

In the general case we must block all migration of the statement to preserve its asymptotic cost. Unless x_3 contains just a single element we consider it to be host-only, and either way we connect every operand that occurs in *dims* to a sink.

let x_1 = x_2 **with** [*dims*] $\leftarrow$ x_3
Not migrated on its own.



The rationale for connecting the operands to sinks is to prevent x_1 from being returned from a kernel and copied, which would incur a cost linear to the size of x_2; the work done by the statement is linear to the size of x_3, and thus smaller. We assume that the write to x_2 would have been eliminated if x_2 and x_3 were of the same size.

The rationale for making the statement host-only is to ensure that the intra-device memory transfers are serviced by the device driver, and not by a **gpu** kernel. An intra-device memory transfer can be serviced via heavily optimised, built-in device kernels that exploit the inherent parallelism of data copying. A single-threaded **gpu** kernel can meanwhile only copy the elements from x_3 one at a time.

When x_3 is a single-element array however, then the data copy cannot benefit from any parallelism. Since the copy can be performed sequentially without loss we allow such instances of the statement to be migrated as part of a parent statement.

Array literals

The array literal statement **let** x_1 = [o_1, ..., o_m] binds an array that consists of the operands $o_1, ..., o_m$. This allows for a rather straightforward graph representation where each operand o_i simply receives an edge to x_1, such that x_1 is migrated if any of its operands are. Because x_1 is an array it can be rebound on the host essentially for free.

There is more to the problem though. When the operands are scalar-typed and at least one operand is a variable, then the runtime populates the array one element at

a time at the cost of m host-device memory transfers. In the CUDA implementation of Futhark, each of these memory transfers will be synchronous and thus particularly expensive. Under OpenCL the transfer of constants will be done asynchronously, and thus somewhat cheaper; they will still be expensive though.

It can be observed that each of the inter-device memory transfers will become an intra-device memory transfer—a plain memory transaction—if the array literal is selected for migration and is created by a kernel. Communication will be reduced as the m memory transfers become one kernel task, and since kernels execute asynchronously, each of the writes will likewise become asynchronous. Because constants are embedded into the program code of kernels, constant operands will neither wastefully be sent to the device every time the array literal statement is evaluated.

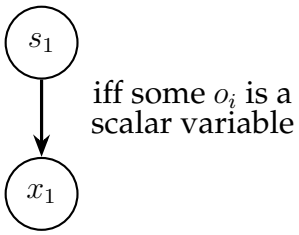
We can ensure that the statement gets migrated by connecting a source vertex to x_1. Because no outgoing edges are added from array variables, no route can be found from that source vertex, and so the array literal statement and no other will be migrated. Since the migration is guaranteed we need not add edges from the operands.

When the operands are array-typed or all operands are scalar constants, then the array is exclusively populated by means of asynchronous intra-device memory transfers. These are generally cheaper than **gpu** kernels and can populate the array more efficiently, as previously discussed. We therefore do not flag such instances for migration but treat them as host-only, provided they contain more than one scalar element.

The concluding rules for array literal migration is shown below. A statement is migrated if $x_1 \in D$, or, equivalently, one of the operands is a scalar variable.

$$\textbf{let}\ x_1 = [o_1, ..., o_m]$$

$$\begin{aligned}&\textbf{let}\ x_1' = \textbf{gpu}\ \textbf{let}\ o_1 = o_1'[0]\\ &\qquad\qquad\quad \ldots\\ &\qquad\qquad\quad \textbf{let}\ o_m = o_m'[0]\\ &\qquad\qquad\quad \textbf{let}\ x_1 = [o_1, ..., o_m]\ \textbf{in}\ x_1\end{aligned}$$



If statements

For programs that involve conditional execution it is generally impossible to design an "optimal" migration transformation as the behaviour that manifests depends on unknown and differing program inputs. When optimising **if** statements and loops we will thus aim to reduce the worst case number of inter-device memory transfers, expecting that the observed average runtime performance will improve.

When dealing with **if** statements this means that we will allow the program transformation to reduce reads into conditional branches but also out of them, and if we can prevent a branch condition from being read to host we will allow entire **if** statements to be migrated. To clarify our intent, we provide example transformations of four code fragments in Figure 13–16.

Migration restrictions While the **if** statement in example Figure 13 is rather simple we allow arbitrarily complex **if** statements to be migrated, provided (among others) that its branches contain no host-only statements and bind no size variables. Size variables (those that occur in array types as variable dimensions) must be bound on the

```
let c = A[0]
let x = if c then [1, 2] else [2, 3]
```

```
let c' = gpu A[0]
let x' = gpu let c = c'[0]
             in if c then [1, 2] else [2, 3]
let x = x'[0]
```

Figure 13: Migrating a whole **if** statement. The read of `c` is eliminated by migrating the entire **if** statement, which is feasible because its array result can be rebound on the host for free. The merge transformation can merge the kernels that produce `c'` and `x'`.

```
let x = A[0]
let y = A[1]
let z = if c then 42
            else let s = x + y in s
```

```
let x' = gpu A[0]
let y' = gpu A[1]
let z = if c then 42
            else let s' = gpu let x = x'[0]
                              let y = y'[0]
                               in x + y
                 let s = s'[0] in s
```

Figure 14: Delaying reads into a branch. Two reads are reduced to one read, or none (when `c` is true). In the transformed fragment the merge transformation would merge the kernels that produce `x'` and `y'`. We assume that `x` and `y` are used by some kernel not shown, such that they are migrated but cannot be sunk into the **false** branch of the **if** statement.

```
let x = A[0]
let y = if c then A[1] else 42
let z = x + y
```

```
let x' = gpu A[0]
let y' = if c then gpu A[1] else [42]
let z' = gpu let x = x'[0]
             let y = y'[0]
              in x + y
let z = z'[0]
```

Figure 15: Delaying a read out of a branch. Between one and two reads is reduced to a single read. The result of the **false** branch must also be migrated. In the transformed fragment the merge transformation would merge the kernels that produce `x'` and `z'`.

```
let x = if a then A[0] else 404
let y = if b then A[1] else 42
let z = x + y
```

```
let x' = if a then gpu A[0] else [404]
let y' = if b then gpu A[1] else [42]
let z' = gpu let x = x'[0]
             let y = y'[0]
              in x + y
let z = z'[0]
```

Figure 16: Delaying reads out of two branches. Zero, one, or two reads may occur in the original program but one read, no more, no less, will occur in the transformed program. Whether this opportunistic transformation pays off depends on the probability that both `a` and `b` is **true**.

host before new arrays with dimensions their size can be created. This is a necessity for the host to allocate appropriately sized memory buffers. Hence when an **if** statement calculates the size of an array that it creates, that **if** statement cannot be migrated. For simplicity we block migration if any size variables are bound, such that their actual use need not be considered. For the language we describe there is no real need to track size variables, as any statement that creates a variably sized array also is host-only. The actual intermediate representation of Futhark however contains some obscure expressions where this distinction is relevant. It should be rather self-explanatory why an **if** statement that evaluates a host-only statement cannot be migrated.

Another restriction on the migration of **if** statements is that they only return scalars or *copyable arrays*. An array of one scalar is as efficient to copy as that scalar and is thus considered copyable. The arrays returned by a statement are otherwise copyable if the copying of an equal or larger number of equally sized array elements would be prevented by migrating that statement. For ease of implementation, in practice we only consider a multi-element array to be copyable if its backing memory is guaranteed to have been initialised by a migratable array literal, created as part of the producing expression. Listing 8 provides an example with two nested **if** statements where the inner statement could be migrated but not the outer.

```
let a = [1, 2, x]
let b = if c then a
        else let d = if e then [1, x, 3]
                     else let f = [1, 2, 3, 4, 5]
                          let g = f[0:0+3]
                          let h = g with [0] ← x
                           in h
             in d
```

Listing 8: An example of copyable arrays. The array bound to `d` is copyable since all its possible values originate from within the returning if statement. The array bound to `b` is not copyable since `a` is bound outside the branch that returns it.

```
let x_1, ..., x_n =
    if c then stm_1 ... stm_p in y_1, ..., y_n
         else stm_1 ... stm_q in z_1, ..., z_n
```

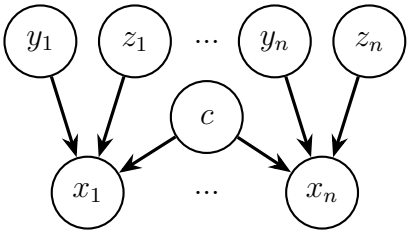


Figure 17: Graphing rules for **if** statements. The statements stm_1 ... stm_p and stm_1 ... stm_q are graphed as any other statements. When p or q is zero the respective **in** term does not appear. Migration is blocked by connecting c to a sink.

Finally an **if** statement may only be migrated if a read reduction can be obtained, that is a reduction of device-host memory transfers. Migrating the **if** statement moves all its bound variables to device, so avoiding the read of the branch condition only pays off if none of the bound scalar variables transitively will cost a read to use. In this regard the subproblem of migrating an **if** statement is similar to that of migrating a function application. It differs however since we consider **if** statements as whole subgraphs, not single units of work, and only statements for a subset of those subgraphs will actually evaluate.

Modelling Accurately modelling the mutually exclusive nature of branches transforms the minimisation problem into a combinatorial optimisation problem that our proposed solution in subsection 4.3 cannot handle. We believe that the combinatorial optimisation problem might be NP-hard and thus infeasible to solve for arbitrarily large programs. To stay within what our proposed algorithm can handle we therefore conservatively assume that *both* branches of each **if** statement are evaluated, modelling the problem as if the rewrite of Listing 9 had occurred.

```
let x_1, ..., x_n = if c then stm_1 ... stm_p in y_1, ..., y_n
                    else stm_1 ... stm_q in z_1, ..., z_n
```

```
stm_1 ... stm_p
stm_1 ... stm_q
let x_1 = f_if c y_1 z_1
...
let x_n = f_if c y_n z_n
```

Listing 9: Conceptual, functionally equivalent rewrite that our model builds on. The function application $f_{\text{if}}\ c\ y\ z$ produces y if c is **true**, otherwise z.

To graph an **if** statement we graph each statement of its branches as if they had been hoisted out of the **if** statement. Afterwards we graph the binding of variable x_i as assignment of branch return value y_i or z_i depending on branch condition c, such that using x_i entails a memory transfer if the computation of either operand c, y_i, z_i is migrated. We visualise these graphing rules in Figure 17.

Migration If c becomes unavailable to the host due to migration, that is $c \in D \setminus C$, then the entire **if** statement need to be migrated and is rewritten like a non-kernel $\mathcal{E}$

```
def two_branches = λA, a, b, i →
  let x = if a then A[0] else 42
  let y = if b then A[1] else i
  let z = x + y
   in z
```

```
let v' = 42
def two_branches = λA, a, b, i →
  let x' = if a then gpu A[0] else v'
  let y' = if b then gpu A[1] else gpu i
  let z' = gpu let x = x'[0]
               let y = y'[0]
               let z = x + y in z
  let z  = z'[0]
   in z
```

Figure 18: Moving alternate branch results to device. The function `two_branches` of type [2]**int** → **bool** → **bool** → **int** → **int** is transformed such that its worst case number of reads is reduced from two to one. The values of 42 and `i` are moved to device by different means.

statement (see section 4.4 and Figure 13). When $c \in C$ then c will be rebound on the host which permits fine-grained migration of branch statements based on the model. This keeps as much work possible on the host, within the constraints, which we hope is more efficient. When a restriction forbids the migration of an **if** statement as a whole we connect its branch condition c to a sink, thus ensuring that c will not appear in $D \setminus C$ and that the **if** statement hence will remain on host.

When an **if** statement is not migrated as a whole, the computation of individual variables returned by either of its branches may still be migrated. Let y_i and z_i be the ith value returned by the respective branches of some **if** statement, to be bound to the variable x_i. When each of y_i and z_i either is a constant or variable that remains bound on the host then no rewrite is needed of x_i, y_i and z_i.

If either y_i or z_i is migrated and no longer remains bound on the host, then its respective storage array y_i' or z_i' is returned in its place. If only one of y_i or z_i has been moved to device, then we also store and return the other in a single-element array such that the branches converge. The result will then be that the value of x_i has been migrated to device and now is stored in the array returned in its place. We replace x_i with x_i'. Note that $\{y_i, z_i\} \cap (D \setminus C) \neq \emptyset$ implies $x_i \in D$. Figure 18 provides an example.

When moving some value v to device to satisfy type constraints we must ensure that we do not introduce any new (synchronous) host-device memory transfers. If v is a constant then we can bind **let** v' **=** [v] as a top-level variable binding and return v' in place of v. When v is a variable we generally cannot store it once, ahead of time. We therefore add the statement **let** v' **= gpu** v at the end of the branch that returns v and returns v' in its stead. Figure 18 exemplifies both techniques.

Discussion A consequence of the inaccurate modelling of branches is that some statements needlessly may be migrated to device to minimise the worst case number of

```
def inaccurate = λA, c →
  let z = if c then let x = A[0] in x + 7
                else let y = A[1] in y + 505
   in z
```

```
def inaccurate = λA, c →
  let z' = if c then gpu let x = A[0] in x + 7
                else gpu let y = A[1] in y + 505
  let z = z'[0]
   in z
```

Figure 19: A shortcoming of the branching model. The function `inaccurate` has type `[2]int` → **bool** → **int**. Because both branches are taken according to the model, the reads that occur in the branches will be "reduced" out of the **if** statement. In reality the worst case number of inter-device memory transfers remain the same, namely one, and redundant overhead is incurred to compute the branch additions on device.

device-host memory transfers. Figure 19 shows one example of this. We note that this is not a flaw with Algorithm 3 as the device set of the graph indeed becomes minimum—it is just that the model that was built did not reflect the actual program.

The choice to reduce reads into **if** branches is rather non-controversial. The worst case number of reads is minimised, and the best case is improved further, similar to the sinking technique mentioned in subsection 2.2. Reducing reads through a set of branches may be suboptimal if the branches in practice rarely are taken, but the overhead of intermittent **gpu** kernels will be paid by the read reduction they secure.

This leaves a discussion of the design decision to reduce reads out of **if** statements. When reads are reduced out of both branches of an **if** statement, as in Figure 19, or when the read of one branch is reduced with a read outside the **if** statement, as in Figure 15, then the transformation cannot cause extra inter-device memory transfers. When we reduce reads out of single branches of multiple **if** statements, as in Figure 16, we gamble that at least one branch doing a read would be taken, such that no new reads occur. The more branches that we reduce reads from, the more likely it is that at least one of them would have manifested a read.

Our hope when reducing a read out of a branch is that that branch represents a non-exceptional case of the program logic. The simple bounds-checking case of **if** `i` $\leq$ `n` $+ -1$ **then** `A[i]` **else** `x`, where `A` has type `[n]`**int**, is one motivating example. When the price of an incorrect guess only is overhead to run **gpu** kernels, and not an increase in synchronous memory transfers, then we can guess wrong more often than not and still see an improvement on average. We may even be consistently wrong for some **if** statements, even at the cost of synchronous transfers, and still obtain a mean reduction from successfully predicting other **if** statements on the execution path.

While the experimental results we present in section 5 support that reducing reads out of branches can be a good idea, further study is required to determine whether our approach is too aggressive. We hope that program slowdowns will be uncommon and insignificant but would not be surprised to see it happen for some program inputs.

Loops

Like with **if** statements there generally is no optimal way to optimise loops as their evaluation tend to depend on program inputs. However, whereas an **if** statement evaluates to either of its two branches once, a loop evaluates its single "branch" $k \geq 0$ times which makes it more likely that the branch is taken than not. This gives an optimisation objective much different from **if** statements as we want to reduce reads out of loops as much possible, and definitely do not want to manifest any new reads inside of them. We also want to reduce a read of one iteration with one that occurs in the next iteration, potentially itself, and in some circumstances we want to migrate the whole loop to device, despite GPUs tending to be a poor fit for branch-heavy code.

For-in rewrite The read done by a **for**–**in** loop can produce a synchronous device-host memory transfer every iteration. The read is inseparable from the loop though, and so only allows a choice between migrating the entire loop or not. To obtain more fine-grained handling we rewrite **for**–**in** loops into **for** loops by the transformation

```
loop y_1 = o_1, ..., y_n = o_n for x_R in x_A do
    stm_1 ... stm_m
    in z_1, ..., z_n
```

```
loop y_1 = o_1, ..., y_n = o_n for k < N do
    let x_R = x_A[k]
    stm_1 ... stm_m
    in z_1, ..., z_n
```

where the outer dimension of x_A is N. We perform the rewrite before we build our migration model, to allow the sinking optimisation to make the read conditional. When x_R is a scalar and sinking occurs the transformation yields a best-case reduction in inter-device memory transfers just by itself. The mean overhead of any **gpu** kernel we introduce to migrate the binding of a sunk x_R likewise decreases.

Initial graphing By rewriting **for**–**in** loops we are left with just two loop forms to handle: **for** loops and **while** loops. These are very similar, as shown below.

```
let x_1, ..., x_n =
    loop y_1 = o_1, ..., y_n = o_n for k < N do
        stm_1 ... stm_m
        in z_1, ..., z_n
```

```
let x_1, ..., x_n =
    loop y_1 = o_1, ..., y_n = o_n while y_c do
        stm_1 ... stm_m
        in z_1, ..., z_n
```

To allow our model of either loop form to reduce reads between iterations we build a cyclic subgraph $\widehat{G}$ where each loop parameter y_i receives an edge from both o_i and z_i. The variable z_i is typically bound by stm_1 ... stm_m which depends on $y_1, ..., y_n$.

The cyclic graph accurately reflects the cyclic dependencies of inter-loop statements but does not allow irrelevant variables to easily be excluded from the model. We solve

```
let e = n + -1
let v = A[0]
let l = v ≤ m
let w, p =
  loop c = l, i = 0
    while c do
      let i' = i + 1
      let a  = i' ≤ e
      let c' =
          if a
          then let v' = A[i']
               let l' = v' ≤ m
                in l'
          else false
       in c', i'
```

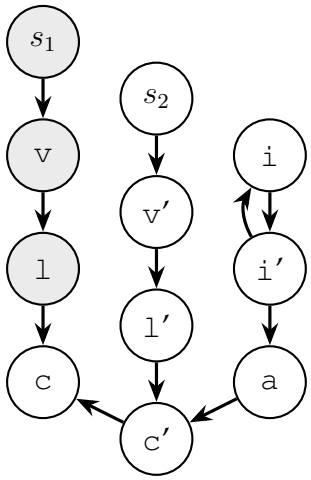


Figure 20: The initial state of the cyclic subgraph for a loop that finds the index of the first array element larger than some threshold. In pseudo code the code fragment corresponds to **loop** `i = 0` **while** `i` $<$ `n` $\wedge$ `A[i]` $\leq$ `m` **do** `i+1` for some non-empty array `A` of type `[n]`**int**.

the problem by assuming that each y_i will become reachable from a source vertex via z_i, thereby adding y_i to the graph even if o_i is ungraphed or connected to a sink. If we predict wrong, the redundant vertices and edges will never be traversed by Algorithm 1 and thus do not directly increase the difficulty of the optimisation problem. We do not add y_i if it is array-typed as no edge then can be received from either o_i or z_i. When each loop parameter of potential relevance has been added, the statements stm_1 ... stm_m are graphed as normal. We do not add the **for** loop variable k to the graph as it is constrained to the type **int** and thus must reside where the loop evaluates.

The variables N and y_c that are used as loop conditions are likewise type constrained, so if these variables are moved to device, a loop relying on either must be migrated as a whole. Seeing that these are the only loop operands that are type constrained we let their membership in $D \backslash C$ be the condition on which loops are migrated. If we find that a loop cannot or must not be migrated, to be discussed later, we connect its concrete N or y_c to a sink.

When the loop body has been graphed, and N or y_c if needed has been connected to a sink, then we add the respective edges $z_i \rightarrow y_i$ on the condition that $z_i \neq y_i$. For reasons we present later we connect z_c to a sink if y_c is connected to one, meaning the edge $z_c \rightarrow y_c$ sometimes need not be added.

In Figure 20–23 we provide examples of what the loop subgraph $\widehat{G}$ may look like at this stage. The examples include free variables that need to be graphed before their loops. All vertices that are external to the loops are shown in grey. Vertices for statement variables $x_1, ..., x_n$ have not yet been graphed and are thus absent. In Figure 24 we show two variants of the exemplified **while** loops where the loop condition variables have been connected to sinks.

```
let n = A[0]
let a = A[1]
let b = A[2]
let c = loop x = a for i < n
          do let y = x + b
             let z = f_host y i
             in z
```

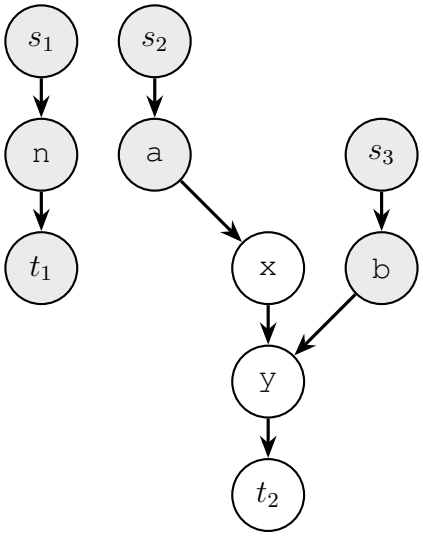


Figure 21: The initial state of the cyclic subgraph for a loop that computes a scalar over `n` iterations. The function f_{host} is host-only so `n` has been connected to a sink.

```
let S, s =
  loop B = A, x = 0 for i < n
    do let v  = B[i]
       let s' = x + v
       let C  = B with [i] ← s'
        in C, s'
```

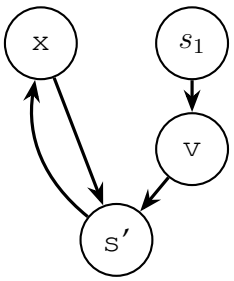


Figure 22: The initial state of the cyclic subgraph for a loop that computes (sub)sums. The arrays `A`, `B`, and `C` have type `[n]`**int**.

```
let d = B[7]
let c = 1 ≤ n
let t, i, v =
  loop c' = c, i' = n, v' = 0
    while c' do
      let i" = i' + -1
      let x  = A[i"]
      let a  = d ≤ v"
      let b  = 1 ≤ i"
      let c" = if a then false
                    else b
       in c", i", x
```

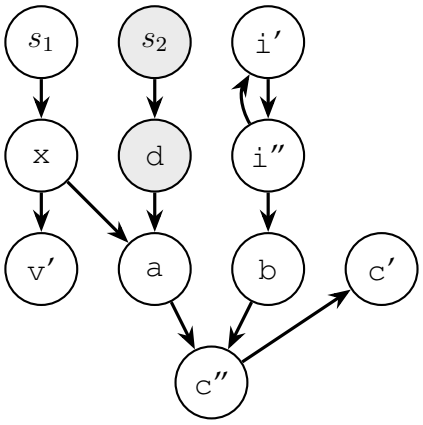
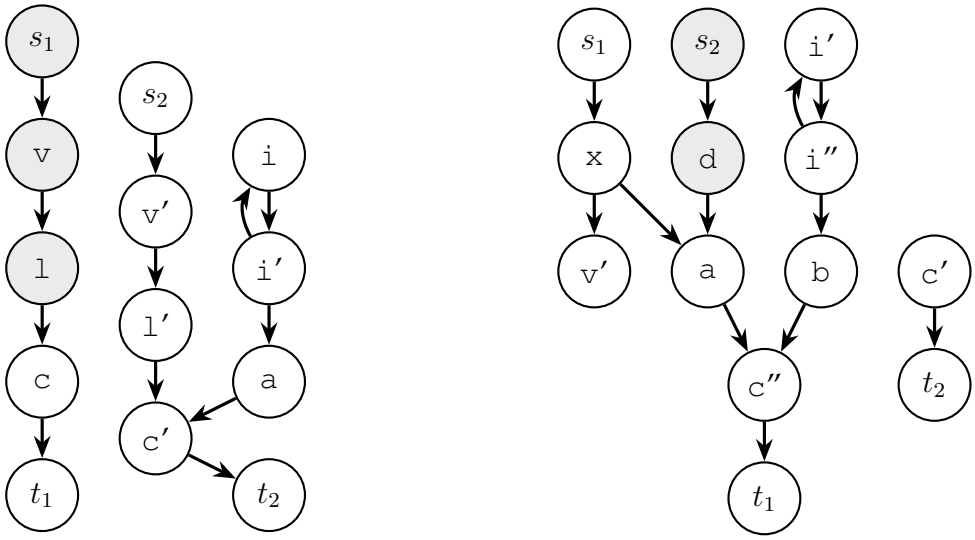


Figure 23: The initial state of the cyclic subgraph for a loop that finds the last value in `A` that is larger than some threshold `d`. The type of `A` is `[n]`**int**.



(a) Figure 20 variant. (b) Figure 23 variant.

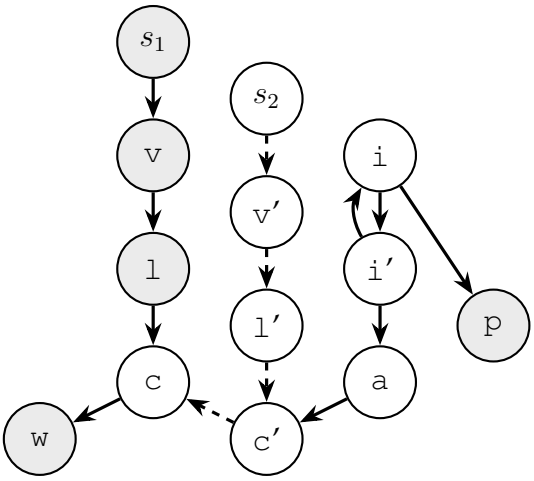
Figure 24: Variants of Figure 20 and Figure 23 where the loop condition has been connected to a sink.

Inter-loop routing A source vertex that occurs within the subgraph $\widehat{G}$ does not represent a single inter-device memory transfer but arbitrary many, equal to the k iterations the loop will make. They should thus only be routed to sinks that also occur within $\widehat{G}$, corresponding to an equal multiplicity of required host uses. If no such usage can be found we want to reduce the transfers out of the loop, expecting the prevented transfers to outnumber any new reads we cause.

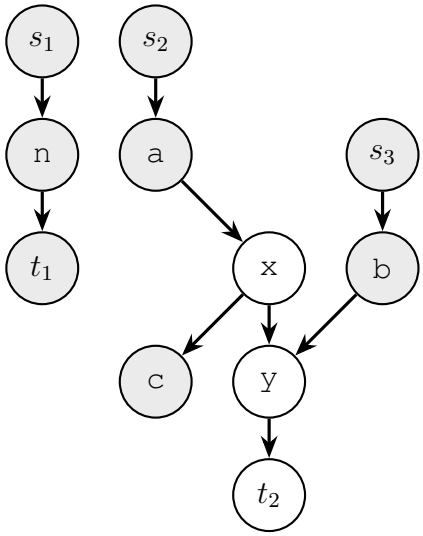
To make our model abide to these constraints we make a routing attempt from each source vertex in $\widehat{G}$ *before* any dependency edges exist to variables outside the loop. Concretely we invoke ROUTE-MANY(G, S) of Algorithm 3 to alter the state of our graph G while it still is under construction, passing the set S of newly added source vertices that occur within $\widehat{G}$. This produces a minimum worst case solution in respect to the loop whose determination we delay until RESOLVE is invoked for the entire graph. The source vertices in S should not be attempted routed again later but if they are, the exhaustion of edges will ensure that no new sink will be found outside the loop. The set S excludes source vertices that were processed when child loops were graphed.

After ROUTE-MANY has been called it is important that no edges are added, removed, or reversed within $\widehat{G}$. Whether a **while** loop is host-only thus must be determined before routing occurs, such that y_c can be connected to a sink. When the subgraph source vertices have been attempted routed, we graph the statement variables $x_1, ..., x_n$ such that each x_i receives an edge from y_i, subject to normal exclusion rules. Figure 25 shows the updated state of the Figure 20–24 graphs.

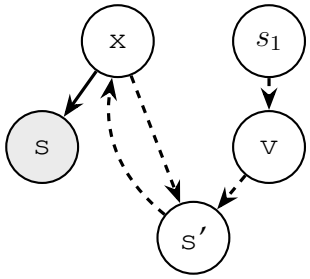
Post loop reduction If fewer routes are created in $\widehat{G}$ than there are source vertices then a read reduction may occur across iterations, as is the case in Figure 25c. This means that some loop return values may be migrated and thus must be read from device to be used by host statements outside the loop. To model this problem we connect a source vertex to each statement binding x_i whose associated y_i will exist in



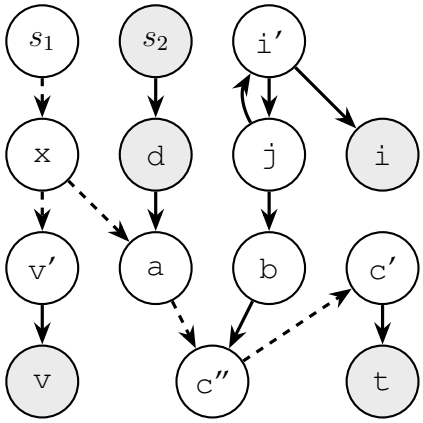
(a) Figure 20



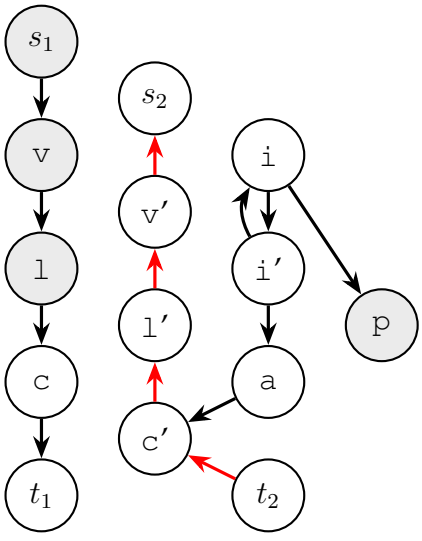
(b) Figure 21



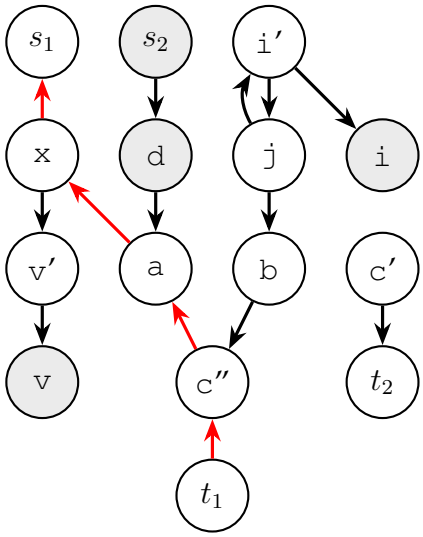
(c) Figure 22



(d) Figure 23



(e) Figure 24a



(f) Figure 24b

Figure 25: Loop subgraphs after the statement variables have been graphed.

D. This corresponds to connecting a source vertex to every x_i for which $x_{i_{\text{in}}}$ is reachable from a subgraph source vertex in the derived graph G'. This can be computed in $O(\widehat{E})$ time using memoisation techniques, where $\widehat{E}$ is the set of edges that are outbound from vertices in $\widehat{G}$.

Loop isolation We also must manipulate the graph such that Algorithm 3 does not cause a reduction that increases the number of reads within the loop. To ensure this we consider every free scalar variable v with one or more edges to vertices in $\widehat{G}$. The loop arguments $o_1, ..., o_n$ are counted among these free variables. If v can reach a sink within $\widehat{G}$, then we connect v to a sink, thereby ensuring that it cannot be reduced into the loop. Otherwise, we exhaust all edges from v to vertices in $\widehat{G}$, and add an edge from v to each statement variable $x_1, ..., x_n$ that it can reach. We note that the pathfinding required to make these changes can be done in the same pass that determines which statement variables the source vertices can reach. The time complexity remains $O(\widehat{E})$ as only $\widehat{V} \leq 2\widehat{E}$ edges can exist to members of $\widehat{G} = (\widehat{V}, \widehat{E})$.

Figure 26 shows the updated state of the Figure 20–24 graphs after additional source and sink vertices have been added. Figure 26b exemplifies how the graph transformation avoids manifesting a new read inside the loop. If `a` and `b` had not been connected to sinks the algorithm would have moved `a`, `b`, `x`, and `y` to device, manifesting a read within the loop to rebind `y`.

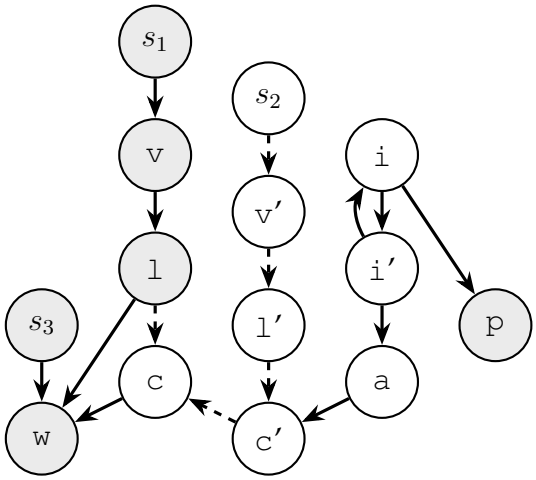
Figure 26d meanwhile demonstrates a situation where we do allow a read to be reduced into a loop. A read already occurs within the loop, so reducing that one with an external one does not cause any increase in memory transfers. In this particular case the overhead of doing so is small as `x` and `a` already will be migrated, which means that no extra kernels will have to be run to facilitate a reduction.

In the variant of Figure 26f a reduction is also allowed within the loop, but in this case an extra kernel execution will be required each iteration. The computation of variables `x`, `d`, `v'`, `a`, and `v` will be migrated if no sink can be reached via `v` when s_2 is attempted routed or a route from s_2 is mutually exclusive with another. Due to the inaccurate modelling of **if** branches, in principle a reduction might not actually occur.

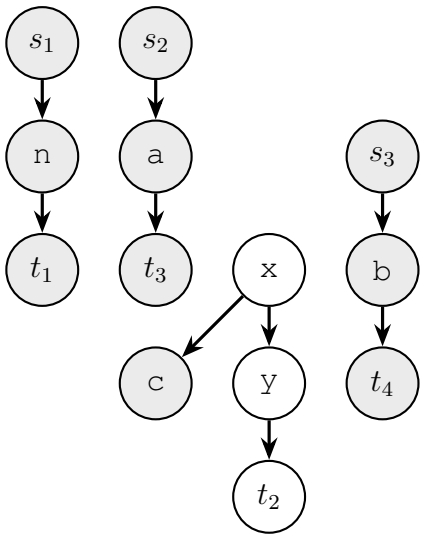
We connect each free variable v directly to the $x_1, ..., x_n$ vertices it can reach to support pathfinding through the loop subgraph without actually visiting its edges, which we know no sink can be reached by. Without the bypass edges we would have to unexhaust the exhausted edges in $\widehat{E}$ for a routing attempt to traverse the loop, and the $O(E\,C)$ asymptotic time complexity of Algorithm 3 would no longer hold. By exhausting the existing edges into $\widehat{G}$ we ensure that not even non-exhausted subgraph edges will be visited by future routing attempts, which is a small optimisation.

We can implement ROUTE such that bypass edges are ignored, and we can assign a subgraph tag to each vertex such that the processing of parent loops ignores the edges into child loops. For all purposes but RESOLVE the graph representation of a loop will then appear as a single statement with edges from operands to bound variables, with some operands being connected to sinks.

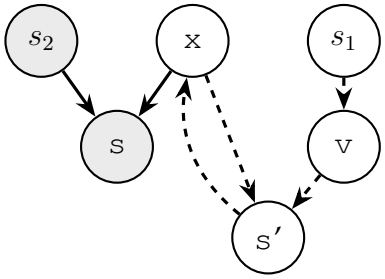
When we exhaust edges into $\widehat{G}$ and leave its inner edges be exhausted we might come to violate the invariant that no sink can be reached via an exhausted edge. This would happen if a sink later became reachable from one of $x_1, ..., x_n$. Since we add bypass edges however, the respective sinks will also become reachable from the free variables of the loop, and if ROUTE' backtracks and exhausts the last bypass edge from



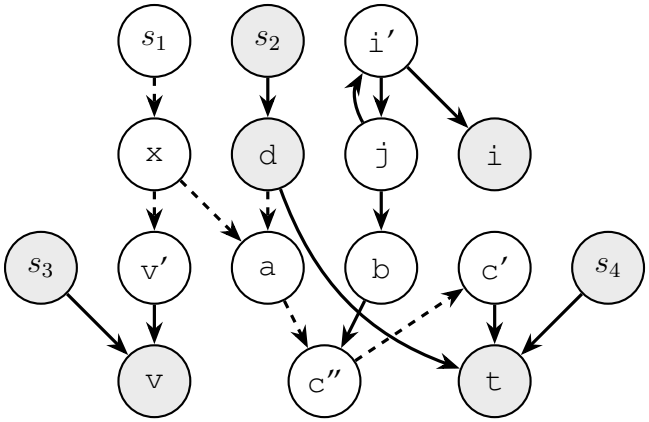
(a) Figure 20



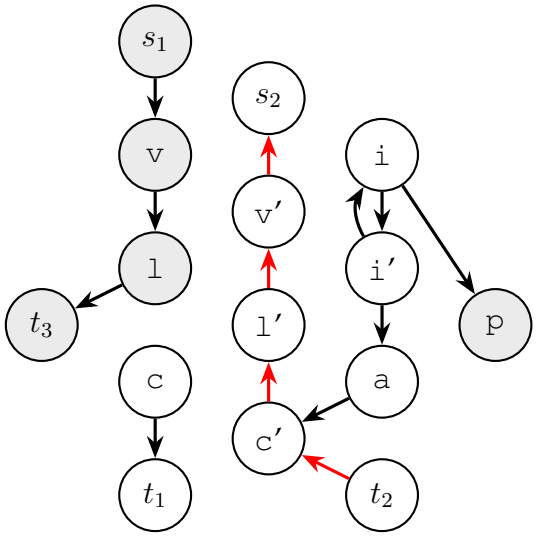
(b) Figure 21



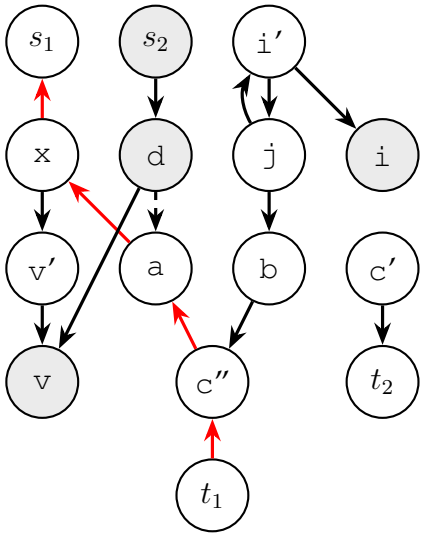
(c) Figure 22



(d) Figure 23



(e) Figure 24a



(f) Figure 24b

Figure 26: Loop subgraphs after extra source and sink vertices have been added.

such free variable, none of the sinks can be reached. While Theorem 3 thus in general will not hold when a loop has been graphed, Corollary 2 will still. As far as the overall proof of correctness goes, this is what matters. It can also be shown that the bypass edges have no effect on which vertices RESOLVE can reach.

Final graphing To finish the graphing of a loop we make one last change to its subgraph by connecting its loop condition (N or y_c) to each statement variable x_i. Concretely this means that a loop only will be migrated to avoid reading the loop condition if a reduction can be had, similar to as was done for **if** statements. Figure 27 shows the state of the Figure 20–24 loop subgraphs after this final change has been made.

Migration To migrate (part of) a loop we use the same techniques as were described for **if** statements. If the loop condition N or y_c is a member of $D \setminus C$ we migrate the whole loop and rewrite it like a non-kernel $\mathcal{E}$ statement (see section 4.4). Otherwise, we perform fine-grained migration of child statements according to the model, after which we consider each tuple (x_i, y_i, o_i, z_i) of variables.

- If $y_i \in H$ then it holds that $\{o_i, z_i\} \subset H \cup C$, which implies that bindings for both o_i and z_i exist on host. The loop was not migrated as a whole so the loop condition must also be in $H \cup C$. Since all other variables with an edge to x_i also can reach y_i, then $x_i \in H$. If x_i was in the device set, then so would y_i be.

  No special rewriting is thus needed.

- If $y_i \in C$ then it must be a vertex where one route blocks another from reaching a sink. Since the only two vertices that have edges to y_i are z_i and o_i, and loop source vertices are routed before external source vertices, then it must be a route via z_i that blocked the routing attempt through o_i. This means all three vertices are reachable from some source via o_i, and so $\{y_i, o_i, z_i\} \subset D$, which means that the storage arrays o_i' and z_i' exist.

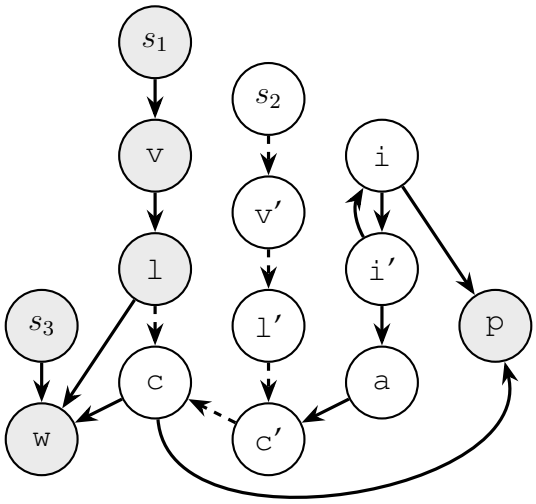
  Since x_i is unreachable from y_i, no bypass edges have been added to it. This means x_i only can have an edge from y_i, and maybe the loop condition (which is in $H \cup C$), and so it follows that $x_i \in H$.

  To rewrite the four variables, we replace the loop parameter binding $y_i = o_i$ with $y_i' = o_i'$, substitute the return value z_i with its storage array z_i', and rewrite the variable binding x_i to x_i'. At the beginning of the loop we then rebind y_i as **let** y_i **=** $y_i'[0]$, and immediately after the loop statement we rebind x_i as **let** x_i **=** $x_i'[0]$.

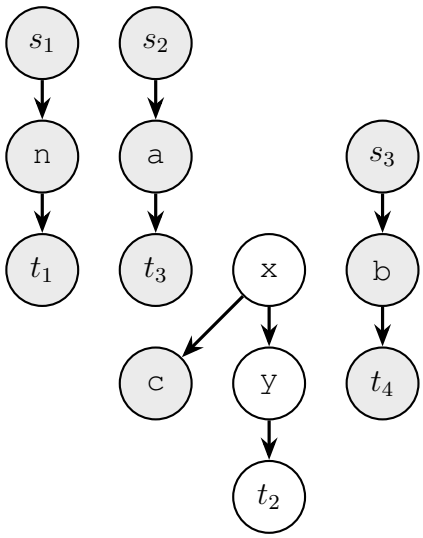
  Because we connect both of y_c and z_c to sinks when migration of a **while** loop is to be blocked, then y_c cannot become a vertex where one route blocks another, and so $y_c \notin C$. Since the only edge to y_c will be from o_c (if any) then for $|D|$ to be minimum we must have $y_c \in H$. This means that y_c is guaranteed to be available on the host and not be rewritten when migration has been blocked.

- If $y_i \in D \setminus C$ then at least one of o_i and z_i is also in the device set for D to be minimum; y_i has no direct edge from a source. Since $y_i \notin C$ then it must also hold that $x_i \in D$.

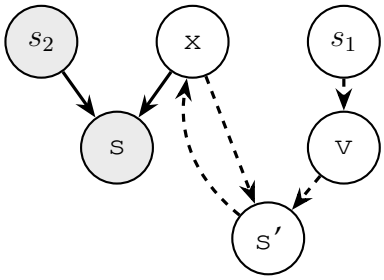
  To rewrite the four variables, we replace the loop parameter binding $y_i = o_i$ with $y_i' = o_i'$, substitute the return value z_i with z_i', and rewrite the variable binding



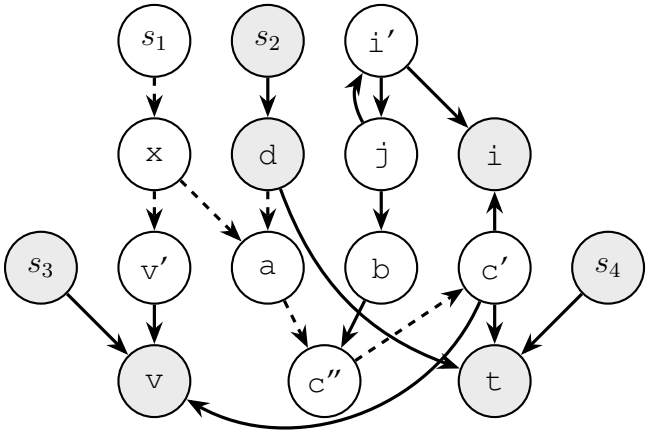
(a) Figure 20



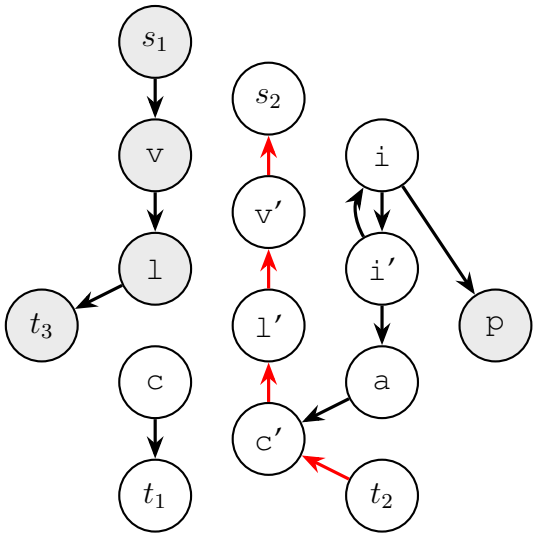
(b) Figure 21



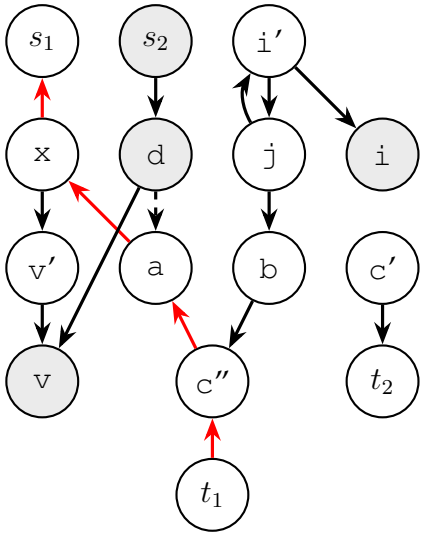
(c) Figure 22



(d) Figure 23



(e) Figure 24a



(f) Figure 24b

Figure 27: Final state of the loop subgraphs after all manipulation is complete.

x_i to x'_i. If the storage array o'_i does not exist because $o_i \in H$ then we create it by binding either **let** o'_i = $[o_i]$ or **let** o'_i = **gpu** o_i before the loop. We do the same for z'_i before the end of the loop. The location of binding and choice of migration method is the same as was discussed for **if** statements.

In Listing 10 and Listing 11 we show the rewrite that would be done for Figure 20 and Figure 22. The other examples would either be rewritten like Listing 10, would not be rewritten, or have no deterministic rewrite based on the available information.

Migration restrictions We now return to the discussion of when the loop condition should be connected to a sink such that the loop by guarantee stays on host. The three restrictions of **if** statements still make sense, and we thus block migration if the loop

- contains any host-only statements.
- binds any size variables.
- returns a non-copyable array.

In practice every multi-element array will be non-copyable since none of the initial loop parameter values originate from within the loop. Furthermore, because most of the statements that bind arrays are considered host-only, then a loop will generally be blocked from migrating unless it only performs scalar work.

Discussion Considering the design goals and intended use cases of Futhark we expect it to be rare for a sequential loop to only perform scalar computations *and* be long-running. Should such loop occur it is likely better to either rewrite it into some sequence of parallel operations that exploits available data-parallelism, or to write this part of the application in a language that excels at sequential execution. Based on this we suggest not blocking any loops from migrating solely on the ground that they are loops, even though their device execution may be slower. If migrated loops are short and simple then they should cost less than a synchronous device-host memory transfer.

A **for** loop always run for a fixed number of iterations, which usually corresponds to a size variable or an argument provided as program input. This means **for** loops probably rarely will be selected for migration, and so choosing whether to indiscriminately block their migration or allow it is likely a choice of no significant impact.

Allowing a **while** loop to migrate can meanwhile make a significant impact if its loop condition involves a scalar array read that depends on a loop parameter. By not connecting y_c to a sink, such loops will generally flag themselves for migration to prevent a read from occurring every iteration. Figure 27a and Figure 27d are examples of this, with the rewrite that results from Figure 27a being showcased by Listing 10.

In our work we have aimed to aggressively minimise the worst case number of inter-device memory transfers and thus synchronisations, assuming that the cost of running **gpu** kernels was insignificant compared to the cost of a synchronous memory transfer. While this generally may be the case we are not convinced that this assumption holds when statements are migrated inside loops to reduce external reads.

Additional **gpu** kernels are not always run when a read is delayed into a loop however. The whole loop may be migrated, making it beneficial to allow a reduction through or into it, and extra kernels that otherwise are added might be merged with existing ones, reducing their overhead significantly. It will take further study to

```
let e  = n + -1
let v' = gpu let v = A[0] in v
let l' = gpu let v = v'[0] in v ≤ m
let w', p' =
  gpu let v = v'[0]
      let l = l'[0]
      let w, p = loop c = l, i = 0 while c do
                   let i' = i + 1
                   let a  = i' ≤ e
                   let c' = if a then let v' = A[i']
                                      let l' = v' ≤ m
                                       in l'
                                  else false
                    in c', i'
       in w, p
```

Listing 10: The Figure 20 code fragment, rewritten. We assume that `v`, `l`, and `w` have no further use. Whether to rebind `p` depends on further use. The merge transformation would merge the three **gpu** kernels and eliminate `v'`, `l'`, and `w'`.

```
let x'' = [0]
let S, s' =
  loop B = A, x' = x'' for i < n
    do let v' = gpu let v = B[i] in v
       let s'' = gpu let x  = x'[0]
                     let v  = v'[0]
                     let s' = x + v in s'
       let C  = B with [i:i+1] ← s''
        in C, s''
```

Listing 11: The Figure 22 code fragment, rewritten. Whether to rebind `s` depends on further use. The merge transformation would merge the **gpu** kernels and eliminate `v'`.

determine under which circumstances it is better to manifest a read than migrating loop statements, and from that develop a more appropriate model. We hope that our model is "good enough" as is.

Copy

The statement **let** x_1 **= copy** x_2 makes a copy of the array x_2. Since no operands are scalar-typed, no graph representation is needed. Array copying from the host is done using asynchronous intra-device memory transfers that can exploit the inherent parallelism. To preserve the cost model of the statement we therefore consider it to be host-only, provided the copied array contains more than one scalar.

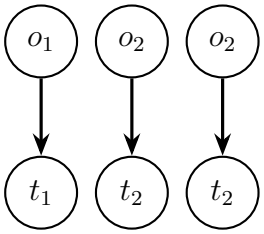
let x_1 = **copy** o_1
Not migrated on its own.

No representation.

Iota

The statement **let** x_1 **= iota** o_1 o_2 o_3 produces an array of size o_1. When evaluated on host the array is asynchronously created by a parallel kernel. To preserve its cost model, we consider the statement to be host-only. The operand o_1 is a size variable.

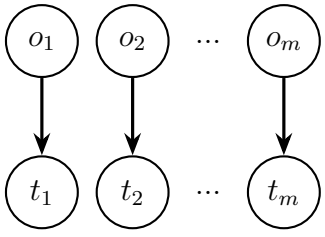
let x_1 = **iota** o_1 o_2 o_3
Not migrated on its own.



Replicate

The statement **let** x_1 **= replicate** $[o_1, ..., o_m]$ o_{m+1} produces an array whose ith dimension has length o_i. All rows of the mth dimension will equal o_{m+1}. The array is created by a parallel kernel when the statement evaluates on host, so we consider the statement to be host-only, again to preserve its cost model. The operands $o_1, ..., o_m$ are size variables.

let x_1 = **replicate** $[o_1, ..., o_m]$ o_{m+1}
Not migrated on its own.



We do not graph the dependency to o_{m+1} as a **replicate** statement can be transformed to handle the case where o_{m+1} is migrated and the storage array o'_{m+1} is bound in its stead. To be specific, when o_{m+1} is a scalar variable that has been migrated to device, then **let** x_1 **= replicate** $[o_1, ..., o_m]$ o_{m+1} can be rewritten into:

```
let x'_1 = replicate [o_1, ..., o_m] o'_{m+1}
let x_1 = x'_1[0:0 + o_1, ..., 0:0 + o_m, 0]
```

When $o_m = 1$ and $m > 1$ a simpler rewrite is also possible:

```
let x_1 = replicate [o_1, ..., o_{m-1}] o'_{m+1}
```

In the case of $o_m = 1$ and $m = 1$ we can rewrite the **replicate** to be a copy of o_{m+1}:

```
let x_1 = gpu let o_{m+1} = o'_{m+1}[0] in o_{m+1}
```

Using a **gpu** kernel instead of **copy** is a small micro optimisation that also allows kernel reduction via the merge transformation, albeit at the cost of lost semantics. Other rewrites are also possible if every dimension is of length one, in which case we need not treat the statement as host-only.

Parallel kernels

Parallel kernel statements such as **let** x_1 **= map** f_1 x_2 and **let** x_1 **= reduce** f_1 o_1 x_2 are by their nature host-only; an intrinsically parallel operation cannot be run by a single-threaded kernel like **gpu** e. We do not consider free variables of the f_1 function argument to be operands of the kernel statements, and thus do not connect those to sinks. If a free function argument variable gets migrated we perform the kernel rewrite detailed in section 4.4.

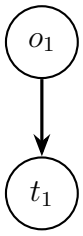
```
let x_1 = map f_1 x_2
```

Not migrated on its own.

No representation.

```
let x_1 = reduce f_1 o_1 x_2
```

Not migrated on its own.



Single-threaded kernels

The statement **let** $x_1, ..., x_n$ **= gpu let** stm_1 ... stm_m **in** $y_1, ..., y_n$ evaluates the given statements on device. The **gpu** statement only indirectly depends on host variables via these kernel statements and thus has no graph representation.

A **gpu** statement is host-only in the sense that the runtime cannot execute a **gpu** kernel within another kernel. We do not consider it to be host-only however as it can

be rewritten to not block the migration of a parent statement. Concretely if the parent of a **gpu** statement is migrated into a kernel, then we rewrite the child **gpu** kernel into:

stm_1
...
stm_m
let $x_1 = [y_1]$
...
let $x_n = [y_n]$

Afterwards the simplification rules discussed for the merge transformation (section 4.4) can be used to eliminate x_i bindings that only are used as $x[dims]$ expression operands.

let $x_1, ..., x_n$ = **gpu** e

Not migrated on its own.

No representation.

5 Benchmarks

To quantify the impact of our automatic program transformations as implemented in the Futhark compiler we provide experimental test results in two forms. First we present microbenchmark results that demonstrate the validity of always migrating array literals that contain at least one scalar variable. Second we present aggregated benchmark results for *futhark-benchmarks* [8], which is a benchmark suite endorsed by the Futhark project. We take these to be a representative sample of Futhark programs and use them to discuss the effectiveness of some of our optimistic transformations.

For both sets of benchmarks we provide results for four different hardware configurations that span a variety of hardware designs and driver implementations. Two configurations amounts to older consumer hardware while the other two represents data centre grade hardware of recent design. The four configurations are:

- **NVIDIA A100** is a dedicated data centre grade GPU from 2020 with CUDA and OpenCL support. Benchmarks were measured on a AMD EPYC 7352 equipped Linux installation, with ECC mode disabled.
- **AMD Instinct MI100** is a dedicated data centre GPU from 2020 with OpenCL support. Benchmarks were measured on the same Linux driven host hardware as the NVIDIA A100.
- **AMD Radeon Pro 560** is a dedicated mobile graphics chip from 2017 with OpenCL support. Benchmarks were measured on a macOS driven MacBook Pro from 2017, equipped with Intel i7-7920HQ.
- **Intel HD Graphics 630** is a graphics chip from 2016 with OpenCL support, integrated into Intel i7-7920HQ. Benchmarks were measured on the same macOS driven host hardware as the AMD Radeon configuration.

All benchmarks we present were measured with *futhark-bench* [10] and are the mean values of at least 100 runs, or 1000 runs for data centre grade configurations. A warm-up run was made for each benchmark before collecting measurements. We have remeasured each benchmark at least once to verify that reported results are representative.

5.1 Micro-benchmark of array literals

We use the following two pieces of Futhark code to demonstrate the performance benefits of migrating array literals where at least one element is a scalar variable. Listing 12 measures the effect when array literals of two scalars are migrated, and Listing 13 measures the effect when array literals of ten scalars are migrated.

```
loop (A, v) for i < 1000 do
  ([1, v], v + 1)
```

Listing 12: Migration benchmark for array literals of two scalars, incl. one variable.

```
loop (A, v) for i < 1000 do
  ([1, v, 0, 1, 1, 1, 0, 0, 1, 1], v + 1)
```

Listing 13: Migration benchmark for array literals of ten scalars, incl. one variable.

It is not possible to benchmark the effect with just one element as such literals are rewritten into **replicate** expressions. We create 1000 arrays per run for three reasons:

1. By measuring the creation of 100,000 or more arrays we achieve statistical robustness.

2. We reduce measurement inaccuracies by forming units of meaningful work. If we only measured the creation of one array literal per run a significant portion of the measured time would be external overhead, such as end-of-run GPU synchronisation to ensure all scheduled work have completed.

3. If we only created one array per run, each array creation would immediately be followed by end-of-run synchronisation and we would not observe any benefits from creating them asynchronously.

We alter the value of the scalar variable each iteration to prevent the compiler from eliminating the loop.

Benchmark results are reported by Figure 28. We observe significant speedups across all measured hardware configurations, the smallest improvement being of factor two and the largest being of factor eleven.

Discussion

The results in Figure 28 supports the hypothesis that migrating array literals with at least one scalar variable operand is advantageous.

Since we have prevented a number of writes equal to the number of array elements it makes sense to assume that the speedup exhibits linear growth. For the benchmarks run under CUDA each of the n prevented writes would be synchronous; correspondingly we see a speedup close to n.

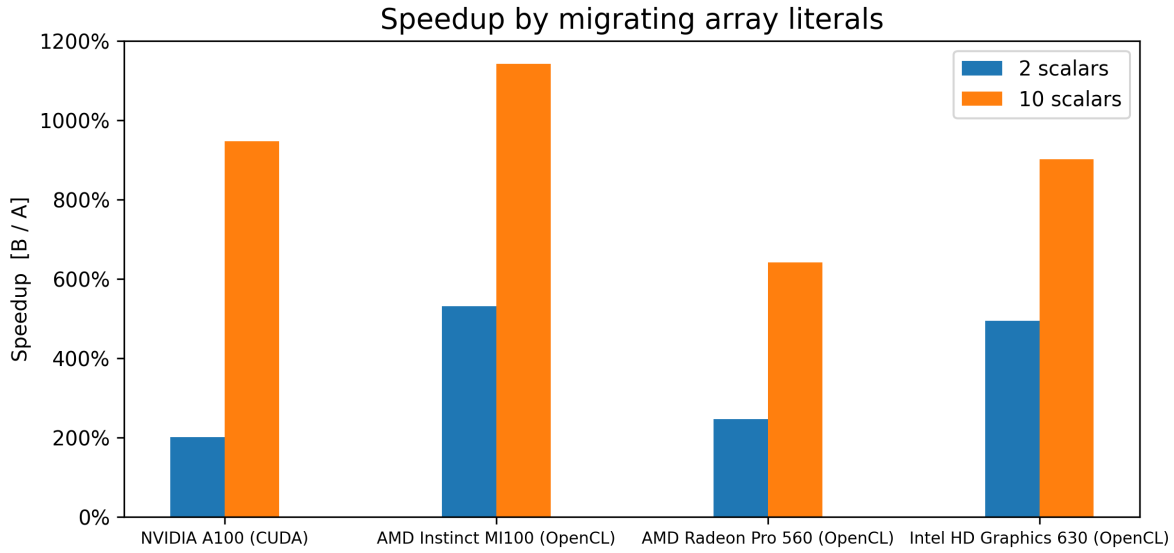


Figure 28: Speedup obtained by migrating array literals with a scalar variable operand.

For the benchmarks run under OpenCL we find that the speedup for one synchronous and one asynchronous write by far surpasses 20% of the speedup for one synchronous and nine asynchronous writes. This supports our hypothesis that synchronous inter-device memory transfers are more expensive than asynchronous ones.

That they remain less than 60% however also supports the hypothesis that any memory transfer in itself is costly and thus is to be avoided. We take this to be an indication that it is worth to reduce memory transfers by program transformation and not simply making them asynchronous by improved code generation.

5.2 *futhark-benchmarks*

futhark-benchmarks [8] contains Futhark implementations of various existing, published GPU benchmarks. Since its programs are written by independent authors and represent problem types that Futhark should be good at, and thus likely will be applied to, we take it to be a representative sample of typical Futhark programs. By providing benchmark results for complete programs we demonstrate the practical impact our transformations can have. If our hypotheses regarding aggressive worst case minimisation of reads do not hold or our cost model has flaws, then we would expect some benchmarks to slow down.

To facilitate a meaningful discussion we do not present results for benchmark programs that are unaffected by our work. Our results can thus not be used to estimate the average speedup of typical programs, only those that our transformations affect. For the 27 benchmark programs we do present results for, results are further aggregated; each program is benchmarked with up to multiple data sets, and considering each input separately is not fruitful to our discourse.

The results we present in Figure 29–32 are average speedups across respective benchmark data sets. We report the speedups as $B_{\text{efore}}/A_{\text{fter}} - 1$ such that improvements, slowdowns, and insignificant changes become visually distinct. Some of the benchmarks cannot consistently be run on all hardware configurations, for which reason some of the figures report results for fewer than 27 programs. The reader should be

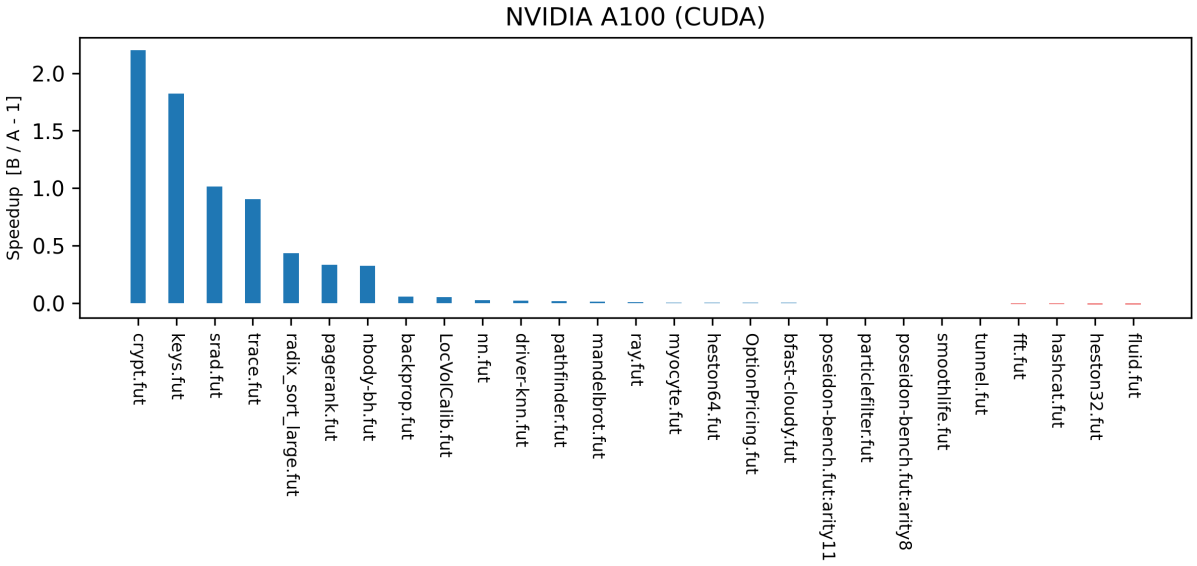


Figure 29: Aggregated benchmark speedups for NVIDIA A100 under CUDA. The mean speedup across the 27 programs is 117%.

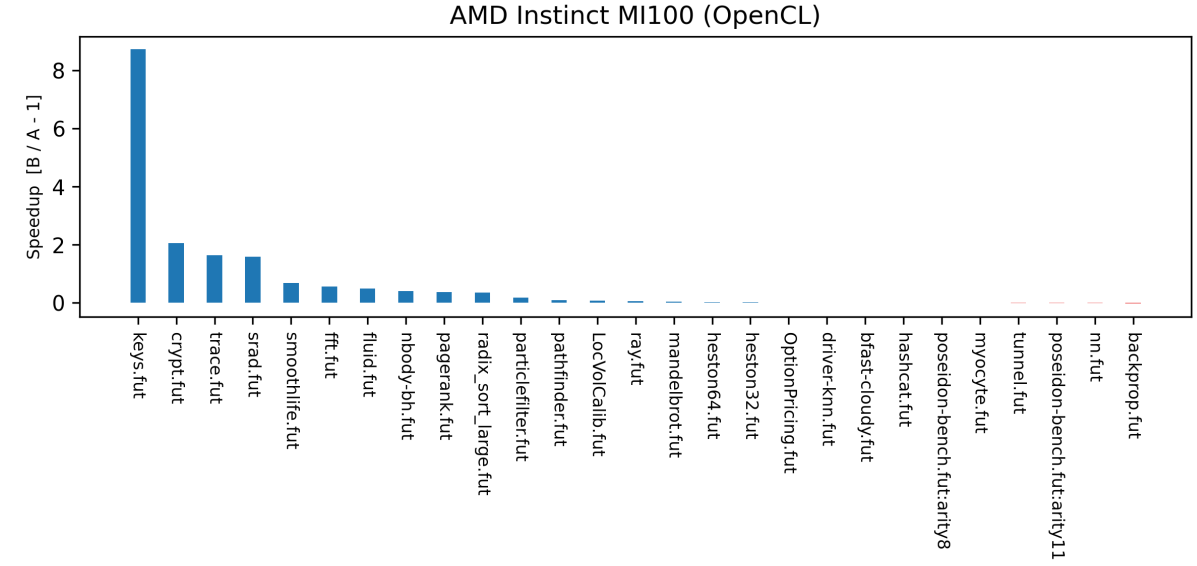


Figure 30: Aggregated benchmark speedups for AMD Instinct MI100 under OpenCL. The mean speedup across the 27 programs is 140%.

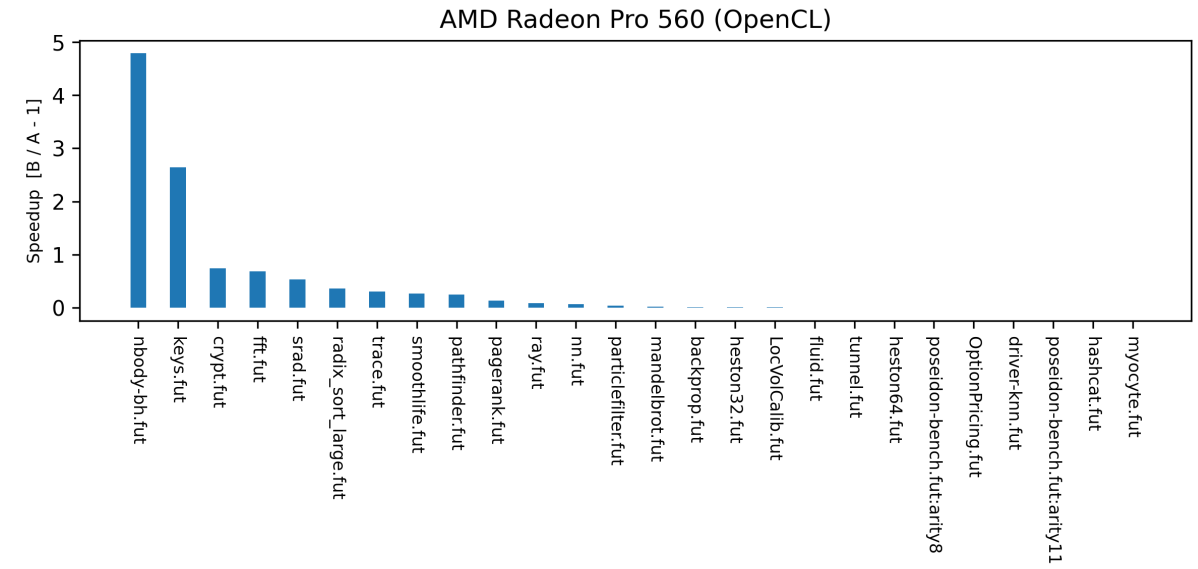


Figure 31: Aggregated benchmark speedups for AMD Radeon Pro 560 under OpenCL. The mean speedup across the 26 programs is 148%.

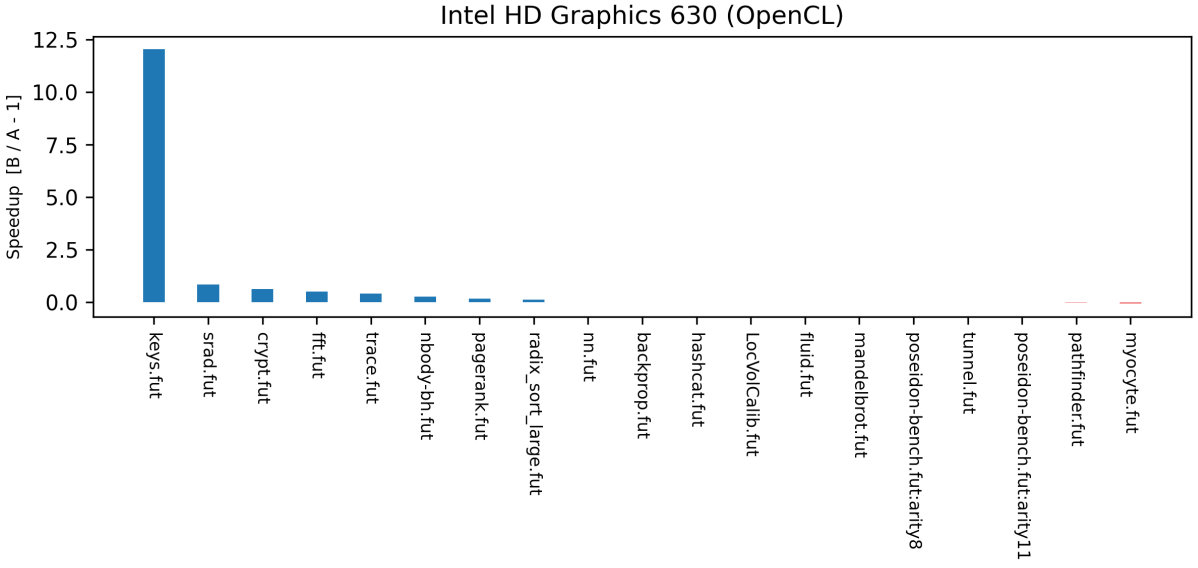


Figure 32: Aggregated benchmark speedups for Intel HD Graphics 630 under OpenCL. The mean speedup across the 19 programs is 158%.

aware that the y-axes vary. The mean speedup reported by each figure is an aggregate of the already aggregated program speedups and not of the individual data sets.

In the plots we see similar trends but also significant differences. For all hardware configurations we can find the `crypt`, `keys`, and `srad` benchmarks in the top five of most sped up programs, with `trace` frequently occurring as well. While `fft` can be found in the top six of the OpenCL configurations it is the fourth worst under CUDA, seeing a small slowdown. Similarly a small slowdown occurred for the `pathfinder` benchmark under Intel HD Graphics 630 while small, yet noticeable speed ups occurred under the other architectures.

The general trend is that a few programs saw significant speedups, skewing the reported means, and that many other programs saw small or insignificant improvements. Relatively speaking no program saw any significant slowdown, and the programs that did slow down differs among the configurations.

Relative measurements can be misleading however, hiding significant costs or gains. In Table 1 we show the top four slowdowns and gains for each hardware configuration, in absolute terms. For the OpenCL based benchmarks, all slowdowns are shown. For CUDA the remaining slowdowns are -61.62μs, -15.86μs, -15.49μs, -4.59μs, and -3.15μs. The -61.62μs slowdown is for the `particlefilter` benchmark and the -4.59μs is the insignificant slowdown of `fft` that likely can be attributed to noise.

In absolute terms only the slowdowns occurring for Intel HD Graphics 630 are significant, and it is interesting that `poseidon-bench` appears among the slowdowns for all four configurations. The program `myocyte` also reappears once, albeit with a much smaller slowdown.

In absolute terms NVIDIA A100 under CUDA was least affected by our transformations, while AMD Radeon Pro 560 was the most. For every configuration the gains outweigh the slowdowns, although only by a small margin for Intel HD Graphics 630. The benchmark programs `keys` and `srad` reappear as top ranking beneficiaries of our transformations. The `LocVolCalib`, `driver-knn`, `ray`, and `smoothlife` programs also appear repeatedly.

Discussion

The transformation of `srad`, `LocVolCalib`, `driver-knn`, `smoothlife`, `nbody-bh`, `ray`, and `fft` produced the largest absolute gains, and yet their rewrites were among the most simple, following a similar pattern: A kernel statement computed one or more results, typically within a loop, and another kernel used those results, potentially after a few scalar computations had been performed.

In each case the reads and scalar computations were migrated into **gpu** kernels, which the merge transformation reduced to a single kernel. In about half of the programs, no intermediate computations were done, and wasteful copies were created. In these cases those wasteful copies were clearly not a problem, but the usage pattern may be common enough to warrant getting rid of the inefficiency anyway. It is noteworthy that the largest savings were obtained with a simple subset of our work.

The `particlefilter` program followed a similar pattern except that reads were reduced from **reduce** kernels in mutually exclusive **if** branches before being used by a third kernel. Allowing to reduce reads out of **if** statements thus proved useful. A read was also reduced into a **for** loop and caused a rather long sequence of inter-loop scalar statements and math function applications to be migrated. It might have been better

NVIDIA A100 (CUDA)			
-613.97μs	heston32.fut	1573.07μs	keys.fut
-228.89μs	poseidon-bench.fut:arity8	2478.40μs	srad.fut
-131.69μs	poseidon-bench.fut:arity11	3837.48μs	LocVolCalib.fut
-102.55μs	smoothlife.fut	36171.22μs	driver-knn.fut
AMD Instinct MI100 (OpenCL)			
-2107.57μs	poseidon-bench.fut:arity11	9619.57μs	particlefilter.fut
-162.37μs	nn.fut	30381.57μs	ray.fut
-81.06μs	tunnel.fut	50896.89μs	driver-knn.fut
-33.44μs	backprop.fut	178237.85μs	smoothlife.fut
AMD Radeon Pro 560 (OpenCL)			
-1390.20μs	poseidon-bench.fut:arity11	38320.28μs	LocVolCalib.fut
-790.35μs	myocyte.fut	306614.48μs	ray.fut
-11.88μs	hashcat.fut	333133.43μs	nbody-bh.fut
33.84μs	fluid.fut	430191.53μs	smoothlife.fut
Intel HD Graphics 630 (OpenCL)			
-220724.13μs	myocyte.fut	21931.07μs	keys.fut
-23349.88μs	poseidon-bench.fut:arity11	23081.89μs	fft.fut
-377.10μs	pathfinder.fut	72450.26μs	srad.fut
2.00μs	tunnel.fut	103813.69μs	LocVolCalib.fut

Table 1: Top four slowdowns and gains for each configuration, in absolute terms.

if this read had been blocked from migrating, even though it lead to no increase of inter-loop **gpu** kernels. It is probably the cause of the small -61.62μs slowdown under CUDA. The program exhibited the fourth largest absolute gain on the AMD Instinct MI100 however, and was sped up by 120%. It cannot be determined whether the loop rewrite aided to obtain this result or worked against it.

The most significant speedup was obtained for `keys`, which improved by a factor of 13 on the Intel HD Graphics 630. It was also among the top-ranked programs by absolute gain, even though its concrete gains were on the small side. Several factors makes the program atypical but also interesting: It is an entirely sequential program that consist of array scalar reads, scalar writes, loops, **if** statements, and scalar computations. It uses no kernel statements. When transformed the reads are reduced out of **if** statements, several while loops are migrated to device, and all reads and writes of scalars are eliminated from the host by rewriting scalar writes into slice writes. We are excited to observe a full array of techniques working together but do not expect the finding to generalise well; the program does not benefit from the GPU in any way.

`trace` was sped up by 130–264%, solely by migrated three four-element array literals to device. This means that their condition-less migration has practical significance.

The `crypt` program consistently appeared in the top three by speedup. The transformations performed on the program however boils down to migrating eight scalar reads to device, half of which are wastefully copied. This is done once, outside a loop. That a significant speedup was achieved is likely because its two **map** statements were applied to small arrays, which means that the work done by the GPU was insignificant and that a large part of the program run time thus came from the overhead of those eight reads. We have reservations regarding how representative this program is.

Our cost model is unable to explain the slowdowns that occurred for the `myocyte`,

`poseidon-bench`, and `pathfinder` programs. The only transformations done to `poseidon-bench` were of top-level variables. We expect these to be initialised with the runtime and therefore they should not effect the benchmarks; yet clearly they do, indicating a flaw in our cost model for Futhark.

For `myocyte` the transformations migrate three scalar reads inside three distinct **if** statements that are guarded by the same condition. No kernels are introduced outside these branches. The compiler then eliminates those **if** statements, and the wastefully migrated reads are combined into one kernel by the merge transformation. We believe that the cause of the slowdowns might be a compiler bug as the **if** statements are special fallback branches that protect unsafe code. This could mean that the transformed program operates on arbitrary data.

For `pathfinder` our transformations rewrote a **for**–**in** loop such that the corresponding scalar read could be sunk. No other change occurred. A 102–125% speedup resulted on the dedicated GPUs, attesting to the usefulness of sinking, but somehow this change caused a 2% slowdown on the Intel HD Graphics 630. Further study is required to understand why this is.

The `heston32` program saw slight speedups under the dedicated OpenCL configurations but a slight slowdown under CUDA. The slowdown under CUDA was caused by a single data set that was slowed down by 20%. This is the only significant slowdown of any single benchmark data set. The transformations made two rewrites: In one branch a loop was rewritten to prevent reading the initial and updated values of a loop parameter, thus preventing a read from manifesting every iteration. This appears to have been a beneficial change. The second change made was preventing four reads of `reduce` results by migrating those and a significant number of intermediate computations that `map` kernels depended upon. We believe this might be the cause.

To prevent the four reads, 223 scalar statements were moved to device, producing 74 values to be read by kernels. Twenty of the migrated statements were `sqrt` computations. The merge transformation really shone as the migrated statements were merged into a single **gpu** kernel, eliminating 226 **gpu** kernels and reducing their array reads to just those of the four **reduce** results. It might however be that preventing those four reads came at too high a cost. From Figure 28 it appears like memory transfers in general might be cheaper under CUDA than OpenCL.

In summary we did not see any slowdown manifest from migrating whole loops, and we only saw benefit from allowing reductions out of **if** statements. Most gains followed from simple migration of scalar operations interspersed between GPU kernels but each of the other transformations we presented also saw use, except the **replicate** rewrites. We did not find conclusive evidence as to whether it is better to allow or block the reduction of reads into loops. The slowdowns we observed were mostly insignificant or indications of implementation flaws, thus supporting the theoretical underpinnings of our work.

6 Future work and conclusion

6.1 Future work

Our work can be expanded upon in two ways to further reduce memory transfers.

Probabilistic vertex cuts

We have aimed to migrate no more statements to device than are necessary, corresponding to finding the minimum vertex cut that minimises $|D|$. In reality other minimum vertex cuts might exists where the reduced reads have smaller probability of manifesting, such that it is worth to migrate more statements. A motivating example would be the code fragment

```
let a = A[0]
let b = A[1]
let c = a + b
let d = c + 42
let e = if x then d else 43
let f = f_host e
```

where $C = (\{\texttt{a}, \texttt{b}, \texttt{c}\}, \{\texttt{d}, \texttt{e}, \texttt{f}\})$ is the minimum vertex cut that causes the least statements to be migrated but $C' = (\{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}, \{\texttt{e}, \texttt{f}\})$ is a minimum vertex cut that allows the resulting read to be sunk and made conditional. In this case the overhead of also migrating `d` would be insignificant as no extra **gpu** kernels would have to be run.

Non-copying gpu kernels

In the language we described, a **gpu** kernel always copies its results to new arrays, even array-typed values. This makes their semantics and implementation simple; allowing array slices to be returned without copying have far-reaching implications for memory management and kernel scheduling.

If it is known in advance which array a **gpu** kernel will return, and the array have no aliases, then the array need not be copied however. The host will know its location in device memory, can manage its memory, and can pass it to subsequent kernels before its contents have been written. It is thus possible to return migrated array literals without copying them. If **replicate** or **iota** statements were migrated, the arrays they produce could likewise be returned without additional copying.

The real significance is that array scalar writes of the form **let** x_1 **=** x_2 **with** $[o_1, ..., o_m] \leftarrow o_{m+1}$ then can be migrated without altering its cost model, and any reads that its indexing operands depend upon need not be read to host. We note that such indexing pattern can be found in *futhark-benchmarks*, which means that a no-copy optimisation represents real potential for memory transfer reductions.

Another significant implication is that any scalar array write induced memory transfer then also can be prevented, simply by migration.

6.2 Conclusion

We have demonstrated a series of dataflow dependent program transformations that move sequential CPU computations to the GPU, such that device synchronisation is reduced and the worst case number of device-host memory transfers is minimised. To drive the transformations we developed a graph-based cost model, showed how such model can support conditional and loop-based computations, and presented a variant of the Ford-Fulkserson method that efficiently solves the modelled problem.

Experimental results generally show modest speedups in programs affected by our transformations but also that significant, manyfold speedups are possible. Because our

model was tailored to the Futhark programming language and its purely functional semantics it is uncertain to what degree our techniques and results generalise. With that said our work appears to be a useful contribution to the domain it was intended and shows that automatic optimisation of inter-device communication can be fruitful.

References

[1] Troels Henriksen. "Design and Implementation of the Futhark Programming Language". PhD thesis. Universitetsparken 5, 2100 København: University of Copenhagen, Nov. 2017.

[2] Khronos OpenCL Working Group. *The OpenCL™ Specification*. Nov. 19, 2021. URL: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

[3] NVIDIA Corporation. *CUDA Toolkit Documentation*. Mar. 24, 2022. URL: https://docs.nvidia.com/cuda/.

[4] *Why Futhark?* Apr. 24, 2022. URL: https://futhark-lang.org.

[5] *futhark-opencl*. Apr. 18, 2022. URL: https://futhark.readthedocs.io/en/v0.21.10/man/futhark-opencl.html.

[6] *futhark-cuda*. Apr. 18, 2022. URL: https://futhark.readthedocs.io/en/v0.21.10/man/futhark-cuda.html.

[7] L. R. Ford and D. R. Fulkerson. "Maximal Flow Through a Network". In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404. DOI: 10.4153/CJM-1956-045-5.

[8] Troels Henriksen et. al. *futhark-benchmarks*. Apr. 18, 2022. URL: https://github.com/diku-dk/futhark-benchmarks.

[9] Cormac Flanagan et al. "The Essence of Compiling with Continuations". In: *SIGPLAN Not.* 28.6 (June 1993), pp. 237–247. ISSN: 0362-1340. DOI: 10.1145/173262.155113. URL: https://doi.org/10.1145/173262.155113.

[10] *futhark-bench*. Apr. 18, 2022. URL: https://futhark.readthedocs.io/en/v0.21.10/man/futhark-bench.html.