FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

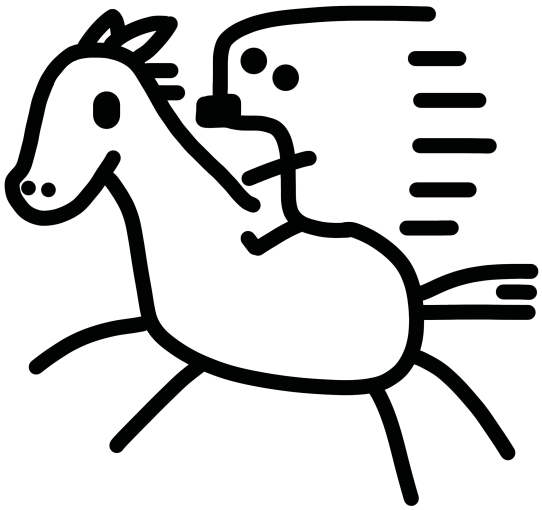
DEPARTMENT OF COMPUTER SCIENCE

Master's Thesis

Bjarke Pedersen lbs845@alumni.ku.dk

Oscar Nelin qgt268@alumni.ku.dk

General Array Locality Optimization by Permutation (GALOP)



Supervisor: Troels Henriksen

Submitted: 2023/12/19

Abstract

Programming in a high-level, data-parallel language such as Futhark alleviates programmers from hardware-specific details. This, in turn, requires such languages to have a general notion of optimal array layouts for their data-parallel operations. Both CPU and GPU architectures may benefit from good locality of reference, and when the user has no control over the program intrinsics, it is the compiler's responsibility not to produce programs with poor locality of reference. Research on this topic is very limited; therefore, we present a novel, hardware-agnostic, best-effort algorithm designed to perform statical analysis and optimize the locality of reference in programs by General Array Locality Optimization by Permutation (GALOP). That is, by strategic permutation on the order of array dimensions. We show that GALOP outperforms the existing solution on the GPU in most cases but achieves this mainly by being more conservative and making fewer bad decisions.

Contents

1 Introduction **4**
1.1 Overview 5

2 Background **6**
2.1 Coalescing Array Accesses 7
2.2 The Futhark Compiler 9

3 Prerequisites **11**
3.1 Language Syntax 11
3.2 Optimal Locality and Iteration Variables 12

4 Algorithm Description **16**
4.1 Overview 16
4.2 Analysis Stage 17
4.3 Layout Stage 19
4.4 Transformation Stage 23

5 Implementation **25**
5.1 Analysis Stage 25
5.2 PrimExp Analysis Stage 31
5.3 Layout Stage 32
5.4 Transform Stage 33

6 Evaluation **34**
6.1 Validation 34
6.2 Performance Comparison 36

7 Discussion **42**
7.1 Results 42
7.2 Future Work 42

8 Conclusion **45**

1 Introduction

Futhark is a pure, statically typed, data-parallel, array programming language that is designed to be compiled to efficient parallel GPU code using CUDA and OpenCL, or multi-threaded CPU code [1].

The Futhark compiler utilizes an immediate representation (IR), which enables it to perform a wide variety of optimizations common to different target architectures, such as single instruction multiple data (SIMD) and single instruction single data (SISD). Futhark can then optimize the general IR further for a target backend architecture, making the specialized IR somewhat hardware-dependent. This abstraction in the front-end language alleviates the programmer from making hardware-specific considerations when writing a program in Futhark.

The problem this thesis is concerned with is that accessing arrays in a manner that has inefficient ***locality*** of reference can severely negatively impact performance. For this reason, the compiler needs to have a way to reason about the efficiency of accesses of arrays with respect to their layouts, which also needs to be general enough to be applied to different architectures.

The Futhark compiler attempts to make locality optimizations using an existing compiler-pass called `babysitKernels`[1] that, at compile-time, attempts to determine whether there exists, for each array access in a program, a layout that improves the locality of the access. It does so by analyzing the way the array is accessed and, if a better layout is found, transforming the array layout by *permuting* the array's order of dimensions. The `babysitKernels` pass works for the most critical cases but cannot handle some rudimentary cases. Furthermore, the existing pass is heavily based on heuristics and would benefit from a more principled approach.

In this thesis, we present a best-effort algorithm for General Array Locality Optimization by Permutation (GALOP). GALOP decides how to lay out arrays in memory in order to optimize the locality of reference when executing on a given architecture. It does so by analyzing how arrays are accessed and making permutations on the order of dimensions of the arrays. It is general in the sense that the methodology applies to multiple vastly different architectures with distinctive computation models, in contrast to the current solution in Futhark, which is specialized only for the GPU backends.

An important consideration is the fact that there are often several layouts that perform better than the rest, sometimes there exists no "*good*" layout, and there are even situations in which such a layout is undesirable, as work spent on transforming the array would negatively impact the performance more than it would benefit. Importantly, our algorithm, GALOP, does not guarantee that it will find the optimal layout, which might not even exist, but makes sure to eliminate the worst-case array accesses whenever it can detect them.

[1] Url: `https://github.com/diku-dk/futhark/blob/dbdfbd72/src/Futhark/Pass/babysitKernels.hs#L19`

Premises. Throughout this thesis, we define several *premises*. It is on these premises that our algorithm is based, and they need to hold for the algorithm to work correctly. They should not be regarded as necessarily true assumptions but as a means to reduce the scope of the problem this thesis is concerned with.

1.1 Overview

In Section 2, Background, we elaborate on and visualize the problem that our algorithm aims to solve as well as present examples that illustrate how our solution should be able to optimize them. These examples are presented in a language that is a simplified subset of Futhark's IR, shown in Figure 3. This language is reduced to what we consider is the minimal set of terms, statements, and expressions that we need to show the functionality of our algorithm.

We use Section 3, Prerequisites, to cover key concepts that are used in Section 4, such as the syntax, *iteration variables*, *variance*, and *level*.

Section 4, Algorithm Description, describes the high-level algorithm of our proposed solution in detail, which we follow up with Section 5, Implementation that describes the implementation of the algorithm in the Futhark compiler.

In Section 6, we describe our process of how we validated the correctness of our implementation, and we present benchmarks of our solution compared with the existing solution.

In Section 7, we discuss the results from Section 6, the limitations of our solution, as well as any possible future work.

2 Background

There is a mechanism on CPUs to fetch contiguous elements from memory and copy them to cache memory. This operation significantly accelerates sequential reads, as cache memory is substantially faster than Random Access Memory (RAM). This hardware optimization relies on the proximity of elements in memory, i.e., *spatial locality*. Failure to leverage this access pattern can severely impact CPU performance [2]. For instance, accessing a location in memory far away from the previous access, referred to as *strided access*, can result in *cache thrashing*, negatively impacting performance.

On a CPU, the cache is divided into three levels: L1, L2, and L3, where the L1 cache is the fastest but also the smallest, and the L3 cache is the slowest but also the largest. Generally, L1 and L2 caches are private to each CPU core, while the L3 cache is shared among all cores [3]. If a program needs to fetch something from the RAM, it makes the program execution significantly slower. This fact is a crucial consideration, as it implies that good locality of reference of sequential loops is highly desirable while less important for parallel accesses.

While the optimal locality of reference occurs when consecutive accesses to memory access contiguous locations in memory, one does not need a stride of 1 between accesses to reap the performance benefits, but in general, the smaller the stride between accesses, the better the locality of reference.

Locality of Reference On GPU. On GPUs, the memory hierarchy is different from that of CPUs. GPUs are designed to be highly parallel and achieve this by having many cores. These cores are grouped into *blocks*, each with a small amount of cache memory shared among the cores in the block, called *shared memory*. Shared among all threads on the GPU is the *global memory*, which is roughly 100x slower than the shared memory [4]. Note that when discussing GPU code, we use the term *coalesced access* to refer to optimal *locality of reference* of global memory. That is, a read or write access to GPU global memory is said to be coalescing if the consecutive threads that execute in lockstep access consecutive global-memory locations in the corresponding SIMD load or store instruction [5].

Due to the efficiency of shared memory, coalesced access is unimportant for shared memory accesses. The reason coalesced access is crucial for performance on GPUs is when threads executing in lockstep access contiguous locations in *global memory*, then, depending on the GPU architecture, the GPU can fetch the data for all threads in a half warp (16 threads) or a full warp (32 threads) in a single memory transaction. Without coalesced access, this might require as much as 16 or 32 memory transactions in the worst case, respectively [6]. Coalesced access is critical for memory-bound programs where global memory accesses constitute the primary bottleneck. Using coalescing enables greater utilization of the memory bus.

In summary, the fundamental difference between why spatial locality is important for CPUs and GPUs is that on CPUs, good spatial locality is effective when contiguous elements in memory are accessed consecutively *in time*, and on GPUs, when contiguous elements in memory are accessed consecutively at the *same time*.

Premise 1
While locality of reference is important for both *reads* and *writes* to memory, the work in

this thesis only concerns itself with *reads*, specifically reads from *arrays*.

2.1 Coalescing Array Accesses

Consider the following Futhark program. This program takes as input a two-dimensional array of integers `A` and sums the rows of the array.

Example 2.1

```
def main [n][m] (A: [n][m]i32) : [n]i32 =
  map (\as -> foldl (+) 0 as) A
```

The program consists of a parallel map over the outermost physical dimension of the array `A` and a sequential reduction over the innermost physical dimension of the array. This results in an array access of the form `A[i,j]` where `i` is a thread ID and `j` is the sequential loop counter. Using our syntax for our simplified subset of Futhark's IR, which is defined in Section 3, we can represent the most essential parts of the program in Example 2.1 as follows:

```
let x_2 =
  kernel i < n do
    let x_1 =
      loop j < m do
        let x_0 = A[i,j]
        in x_0
    in x_1
in x_2
```

On a CPU, the locality of reference of the array access `A[i,j]` is optimal as the loop counter j accesses contiguous elements in memory. However, on a GPU the access is not coalesced as the thread ID i accesses the array with stride of m. Since the locality of reference is not optimal, we can improve the performance of the program by transforming the layout of the array `A`. Note that this reasoning is based on the following assumption.

Premise 2
The algorithm proposed in this thesis works with the assumption that all arrays are allocated in *row*-major order. Note that this is not always true for Futhark programs but is true in most cases.

To improve the locality of the access to A on GPU, we can transform the layout of A to be column-major order, i.e., transpose the array. This results in the thread ID i accessing contiguous elements of A and thus having coalesced access. This transformation is illustrated in Figure 1.

This brings us to the main assumption that the work in this thesis is based on:

Premise 3
We can achieve good locality optimizations by permuting the order of dimensions of the arrays, i.e., changing the layout of the arrays in memory, in such a way that the resulting layout corresponds to a transpose of row-major order. This relies on Premise 2. This also relies on the fact that we can make fast transpositions regardless of architecture, which has previously been demonstrated many times [7] [8].

One way to argue for this is that Futhark's model of programming, i.e., data parallelism, can fundamentally be regarded as sequential or parallel loops performing operations along

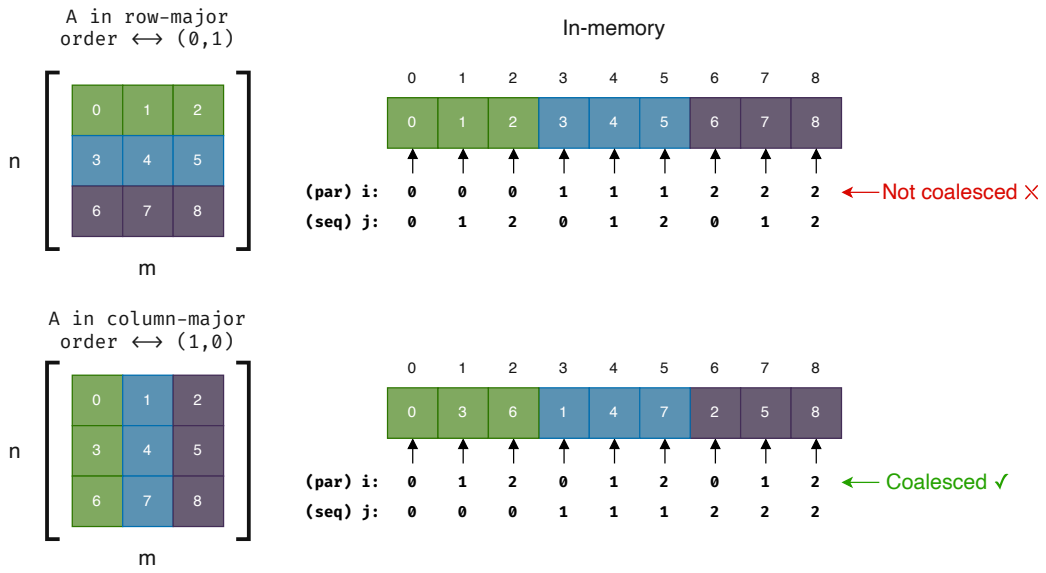


Figure 1: The coalescing effect of the layout of A from row-major order to column-major order.

the full length of individual dimensions. This allows us to only consider array accesses with the granularity of individual dimensions.
To make clear what we mean by Premise 3, we need to first define what it means for layout to correspond to a transpose. First, we define an *index function* for row-major order

$$I : \mathbb{N}^r \to \mathbb{N}$$

which maps from a pointer in r-dimensional space to a flat offset. For example, given an array access $A[i, j, k]$, to an array A with $m \times n \times o$ dimensions, we get $I(i, j, k) = i \times n \times o + j \times o + k$. We can generalize this to arbitrarily dimensional matrices, though impossible to illustrate on a 2-dimensional a4 paper, we can write the formula as the following. Let r be the rank of A, let n_i be the size of the ith dimension of A, and let d_i be the ith parameter of I, then

$$I(d_0, \ldots, d_{r-1}) = \sum_{0 \leq i < r} d_i \times \prod_{i < j < r} n_j \tag{1}$$

Note that when considering an array access to an r-dimensional array and we refer to the access to the *innermost* dimension of A, we specifically refer to the last parameter i.e., d_{r-1}, and when we refer to the access to the *outermost* dimension of A, we refer to the first parameter d_0.

Consider an ordered set S containing n elements, where each element represents a dimension of an array that has a rank of n (i.e., an n-dimensional array). We can represent a *layout* that describes the order of dimensions as a n-tuple where, for $0 \leq i < n$, the ith element of the tuple is the index of S to which the ith element of S is mapped. For example, consider the ordered set $S = \{1, 2, 3, 4\}$. A function $\rho : S \to S$ which exchanges the elements 2 and 4 and leaves the elements 1 and 3 fixed is a *permutation* of S. This permutation can be summarized in a table as follows:

s	1	2	3	4
$\rho(s)$	1	4	3	2

This permutation applied to the ordered set S leads to the new layout $\pi = (0, 3, 2, 1)$.

Transpositions. Recall Premise 3. For a two-dimensional array, the layout (1,0) is the transpose of an array with the layout (0,1) which is row-major order. For arrays with $n > 2$ dimensions, we use the term to refer to a layout that can be obtained by rearranging the order of dimension by splitting the dimensions k through n into two subsets of contiguous dimensions of the array and exchanging them, where $0 \leq k < n$. That is, we can split the dimensions of the array into three subsets where the first subset is fixed and the last two subsets are exchanged. For example, consider the identity layout for a three-dimensional array, (0,1,2). We can divide this layout into three subsets of contiguous dimensions, (0)(1)(2). Exchanging these last two subsets results in the layout (0,2,1). This layout is a transpose of the row-major order. For a five-dimensional array, the identity permutation (0,1,2,3,4) can, for example, be divided into the subsets (0)(1,2)(3,4). Exchanging the last two subsets results in the layout (0)(3,4)(1,2) which is a transpose of the original layout. An example that is not a transpose is the layout (1,2)(0)(3,4) which has exchanged the first two dimensions. This layout is not a transposition because the first dimension should, as mentioned, be held fixed.

Manifest. Consider Example 2.1 again. Based on Premise 2, we know that the array `A` is allocated in row-major order which is denoted by the layout (0,1). To achieve coalesced access, we need to transform the layout of `A` to column-major order, which is denoted by the layout (1,0). To do this, we can use the `manifest` function, which takes as input a layout and an array, and indicates to the compiler that it should make a copy of the data of the array and allocate it in memory in a way that respects the given layout. This is illustrated in Figure 2. Importantly, the `manifest` function does not change the semantics of the program, i.e., the program is still equivalent to the original program. Semantically, the `manifest` function is just the identity function. For example, the access `A[i,j]` from Figure 2a returns the same value as `A'[i,j]` from Figure 2b. Note that manifesting an array also requires the use of an index function different from the one we have defined for row-major order to access to same values. Also, note that the motivation behind performing locality optimizations in this manner using the `manifest` function, instead of permuting the order with which iteration variables are bound, is that it would necessitate reordering of loops and kernels which would make it much more complicated to maintain the semantics of the program.

In the case of an access $A[i, j, k]$ to a three-dimensional array that is allocated in row-major order, i.e., with the layout (0,1,2), where j is a thread ID and k is a sequential loop counter, then, on the GPU, we would like to permute the two innermost dimensions to achieve coalesced access. This results in the layout (0,2,1).

2.2 The Futhark Compiler

The Futhark compiler is structured as a series of *passes* that are executed in order and grouped into *pipelines*. The passes perform work like simplifying the IR of a program and various optimizations such as locality optimizations which are done by the `babysitKernels`

```
let x_2 =
  kernel i < n do
    let x_1 =
      loop j < m do
        let x_0 = A[i,j]
        in x_0
    in x_1
in x_2
```

(a) Original program.

```
let A' = manifest((1,0),A)
let x_2 =
  kernel i < k_0 do
    let x_1 =
      loop j < k_1 do
        let x_0 = A'[i,j]
        in x_0
    in x_1
in x_2
```

(b) Transformed program.

Figure 2: Original GPU code of the example program and desired GPU code.

pass. The first passes are shared between all backends while the last passes are backend-specific.

The locality optimizations are performed at a point in the pipeline where the subsequent passes annotate the IR with memory information and the preceding passes have transformed the program into a form where

- All polymorphic functions have been monomorphized.
- All higher-order functions have been defunctionalized.
- There is no recursion and most functions have been inlined.
- A lot of general optimizations have been performed (e.g., fusion, copy propagation, constant folding, common subexpression elimination (CSE), etc.).
- Embedded parallelism has been flattened (in the GPU backend) or handled in some other way (in the multi-core (MC) backend).
- Second-Order Array Combinators (SOACs) (e.g., map, reduce, scan, etc.) that need to be executed sequentially have been transformed into explicit sequential loops with explicit dimension accesses.

Hence, the locality optimizations are performed after all structural transformations but before any very low-level optimizations are performed.

The work in this thesis aims to replace the existing pass `babysitKernels` in the GPU pipeline as well as introduce the pass to the multi-core (MC) pipeline where no such pass exists. This implies that the implementation of our algorithm (i.e., the new pass) should be generalized to several backends and ideally share as much code as possible between the backends.

v ::=		**Values**
	k	Integer
|	x	Variable name
s ::=	let x = e	**Statement**
b ::=		**Body**
	s $\mathtt{b}_0$	Statements
|	in v	Result
se ::=		**Scalar expression**
	$\ominus$ v	Unary operator
|	$\mathtt{v}_0 \oplus \mathtt{v}_1$	Binary operator
|	v	Value
e ::=		**Expression**
	se	Scalar expression
|	$A[\mathtt{se}_0, \ldots, \mathtt{se}_n]$	Array access
|	If se then $\mathtt{b}_0$ else $\mathtt{b}_1$	Condition
|	kernel $i <$ v do b	Parallel loop
|	loop $j <$ v do b	Sequential loop
|	manifest$([\mathtt{k}_0, \ldots, \mathtt{k}_n], A)$	Manifestation

Figure 3: The syntax of our program which resembles a simplified subset of the IR of Futhark.

3 Prerequisites

In this section, we define some important concepts and terminology that will be used throughout the thesis that will be used as a basis for the description of our algorithm, GALOP, in Section 4.

3.1 Language Syntax

To aid in the illustration of our algorithm in Section 4, we define the syntax of a language that is a simplified subset of Futhark's IR. We show the syntax in Figure 3. The following discussion will use this syntax as a basis to illustrate our algorithm. A program is some body b, consisting of a sequence of statements parameterized over a list of variables, ending with a final expression which is the result of the body. Each statement consists of a binding of an expression e to some variable x. Note that throughout this thesis, we use the term *statement* interchangeably with *let-binding*. Lastly, note that in our syntax, an *array access* refers to an expression of the form $A[\mathtt{se}_0, \ldots, \mathtt{se}_n]$ which contains $n = \mathtt{rank}_f(A) - n$ scalar expressions. This is because we assume that all array accesses are complete. Importantly, our algorithm only considers array accesses that *read* from arrays and does not consider *writes* to arrays.

Example 3.1
The following is a contrived example of a program b in our syntax. We will use this example throughout the following section as well as Section 4 to illustrate our algorithm.

```
let x5 =
  kernel i < k0 do
    let x3 =
      loop j < k1 do
        let x0 = A[i*j+k0, j*k1, 0]
        let x1 = A[i,j,0]
        let x2 = x0 + x1
        in x2
    let x4 = B[x3]
    in x4
in x5
```

Throughout Section 4, we make use of what we will refer to as *oracle functions* that map their input to some output that can't be inferred from the input alone, but only through the use of an implicit (or *hidden*) domain of additional information. This information is the program body on which the algorithm is applied, and since the body does not change throughout the execution of the algorithm, the information inferred from it by the oracle functions could easily be precomputed and passed as an argument to the algorithm or computed by the algorithm itself. The following definition is one such oracle function since the rank of an array can't be inferred from the array name alone.

Definition 3.2
Let A be the name of an array, then `rank` is defined as

$$\texttt{rank}_f : A \to \mathbb{N}$$

that takes an array name A and returns the number of dimensions of the array.

Premise 4
Our algorithm assumes that the program is well-typed, and we assume that all bound names in a program are unique.

3.2 Optimal Locality and Iteration Variables

One of the main contributions of this thesis is the idea that we can *often* determine an optimal layout of an array based on the *iteration variables* of the expression that accesses it along with a *level* describing how deeply the definition of each iteration variable is nested inside a nest of program statements. In this section, we will define what we mean by iteration variables and level, how we can determine them, and why this information is often powerful enough to be all we need to determine the optimal layout of an array, if such a layout exists.

The problem space that our algorithm operates in is the problem of poor locality of reference. We reduce the problem space, that our algorithm should handle, to that where there exists a parallel or sequential memory access with a stride greater than one, which occurs when the following two conditions are met

1. There are one or more kernels or loops nested inside a kernel.
2. Several of the *kernel variables* (i.e., the names which thread IDs are bound to in a kernel definition) or *loop variables* (i.e., the names which loop counters are bound to in a loop definition) in the nest are used to access the same array.

This implies that the only variables that can affect the stride of parallel memory accesses are kernel variables and loop variables. We refer to these variables as **iteration variables**. Note that these two conditions being satisfied does not necessarily imply an uncoalesced memory access pattern since the iteration variables may be used to access the array in the same order as that of their definitions.

Premise 5
Depending on the hardware, optimal locality of reference occurs, on a GPU, only when the access to the innermost physical dimension of the array is **parallel** and, on a CPU, only when the access to the innermost physical dimension is **sequential**. Note that this premise also assumes that the array is allocated in row-major order, which is given by Premise 2. More formally, given a sequence of expressions $(\texttt{e}_0, ..., \texttt{e}_n)$ where each expression $\texttt{e}_i$ for $0 \leq i \leq n$ is either a kernel or a loop nested inside the body of $\texttt{e}_{i-1}$, given $i > 0$, and the expressions each define a kernel variable or loop variable x_i respectively, then an array access of the form $A[x_0, ..., x_n]$ that is nested inside body of the last expression $\texttt{e}_n$ will, on a GPU, be optimal only if x_n is a kernel variable and, on a CPU, be only if x_n is a loop counter, that is, when the access to the innermost physical dimension of the array is an *iteration variable*. Importantly, this condition being satisfied is a requirement for optimal locality but does **not** necessarily imply optimal locality.

For example, consider Example 4.6 in Section 4.3. Here, we have

$$\begin{aligned}
\texttt{e}_0 &= \text{“}\texttt{kernel } i < \texttt{k}_0 \texttt{ do } ...\text{”} \\
\texttt{e}_n &= \text{“}\texttt{loop } j < \texttt{k}_1 \texttt{ do } ...\text{”} \\
x_0 &= i \\
x_n &= j \ .
\end{aligned}$$

The innermost physical dimension of A is accessed by the loop variable x_n, which on a CPU is a requirement for optimal locality. However, on a GPU, the requirement is not met since the innermost physical dimension needs to be accessed by a kernel variable like x_0.

This notion that the locality of an array access is determined by the use of iteration variables, i.e., kernel variables and loop variables is the main motivation behind the data structure generated by the analysis stage.

Definition 3.3 *Iteration variables*
Let

$$IV : \texttt{se} \rightarrow \{x_0, x_1, ...\}$$

be an oracle function that takes a scalar expression and returns the names of kernel variables and loop variables that the variable depends on.

Definition 3.4 *Variance*
If a scalar expression `se` does not depend on any kernel variable or loop variable, we say that it is *invariant* and we have that $IV(\texttt{se}) = \emptyset$. By extension, we use the term *variance* to describe whether or not an iteration variable is *variant* or *invariant*. This can also be regarded as information that can be obtained by an oracle function.

We represent the iteration variables as an unbounded set of variables since there is no upper limit on the number of iteration variables that a scalar expression may depend on. Note that the iteration variables $IV(\texttt{se})$ are a subset of the free variables $FV(\texttt{se})$. Hence, we have that

$$IV \subseteq FV \ .$$

We quantify an iteration variable as being either *parallel* or *sequential*. In the case of a kernel variable, we say that it is parallel and in the case of a loop variable, we say that it is sequential.

Constants. The reader might wonder why we do not need to consider constants in the set of iteration variables. We intentionally impose this limitation for the following reasons.

Premise 6
Any occurrence of an acces into a dimension of some array by an expression that is invariant to all iteration variables is most likely to occur when the dimension accessed by the expression is small. In such a case, manifesting the array would lead to work spent on copying data that is asymptotically greater than the work saved by having optimal locality.

For this reason, we omit constants from the set of iteration variables. For example, consider in Example 3.1 the array access bound to x_1. Here, the innermost physical dimension of A is accessed by the constant `0` which is invariant to the iteration variables i and j. If the assumption holds that the last physical dimension of A is small, then j already has an equally small stride and manifesting A would likely lead to more work spent on copying data from the two larger dimensions than the work that would be saved by having optimal locality.

To clarify the term *invariant* used in Premise 6, consider Example 3.1. Here, the dimensions of the array A are accessed by the scalar expressions "i`*`j`+`$\texttt{k}_0$", "j`*`$\texttt{k}_1$", "`0`", "i", and "j" and the array B is accessed by the scalar expression "x_3". The iteration variables of these scalar expressions are

$$\begin{aligned}
IV(i\texttt{*}j\texttt{+}\texttt{k}_0) &= \{i\} \\
IV(j\texttt{*}\texttt{k}_1) &= \{j\} \\
IV(\texttt{0}) &= \emptyset \\
IV(i) &= \{i\} \\
IV(j) &= \{j\} \\
IV(x_3) &= \{i, j\}
\end{aligned}$$

In Example 3.1, both values $\texttt{k}_0$ and $\texttt{k}_1$ are invariant to the kernel variable i and the loop variable j, meaning each takes the same value for every value of both iteration variables. Likewise, the constant `0` used to access the last dimension of the array is invariant to i and j. Assuming Premise 6 holds, we can safely ignore the constants and only consider the iteration variables.

Level. As mentioned earlier, along with the iteration variables, we also need to know the *level* of each iteration variable. More concretely, given a nest of statements, we need to know for each iteration variable the level in the nest at which it is defined. For example, the levels

of the statements in Example 3.1 are as follows:

```
let x5 =                                          level(x5)=0
  kernel i < k0 do                                level(i) =0
    let x3 =                                      level(x3)=1
      loop j < k1 do                              level(j) =1
        let x0 = A[i*j+k0, j*k1, 0]               level(x0)=2
        let x1 = A[i,j,0]                         level(x1)=2
        let x2 = x0 + x1                          level(x2)=2
        in x2
    let x4 = B[x3]                                level(x4)=1
    in x4
in x5
```

Definition 3.5 *level*
We define the level of a variable as an oracle function

$$\texttt{level}_f : x \to \mathbb{N}$$

that takes a variable x and returns the level at which it is defined in the nest of bodies in a program. That is, each expression that defines a new body increases the level of the variables defined in that body by one.

4 Algorithm Description

This section will present a best-effort algorithm to improve the locality of reference on programs by determining an optimal dimension ordering of arrays. We begin by showing a high-level step-by-step (or rather, stage-by-stage) overview of the algorithm and the intermediate data structures, followed by a more detailed description and motivation of each stage.

Recall that the purpose of the algorithm is to avoid array accesses with an inefficient locality of reference by telling the Futhark compiler to allocate the arrays with their respective optimal layouts in memory with respect to those array accesses. When an array with a suboptimal layout in memory can be determined, the algorithm tries to determine a better layout of the array, which, if found, is then used to transform the array by permuting the order of its dimensions. Importantly, it is possible that there can be more than one layout that achieves better, or optimal, locality of reference. The layout is determined by how each array is used, which in turn means that an optimal layout is not unambiguous; each access might need different layouts. Lastly, it is not always possible to determine an optimal layout. For example, some scalar expressions used to access the arrays might be *too complex* for this algorithm to determine such a layout. In such cases, the algorithm does not transform the array. This has the benefit that the algorithm does not risk transforming an array that is already optimally laid out in memory, making the locality of reference worse.

4.1 Overview

Our algorithm is structured in three stages, each producing an output that is passed to the next stage. The algorithm takes as input the IR of a well-typed Futhark program and generates a new program with equivalent value semantics but with better locality of reference where possible.

1. The **Analysis** stage produces an *index table* $\mathcal{I}$ (shown in Definition 4.1) that for each array access in the input program, yields a tuple containing the name of the source array A, the variable to which the array access was bound to, and a tuple of all the scalar expressions used to access A that preserves the order in which they were used to access the dimensions of A.

2. The **Layout** stage takes as input the index table $\mathcal{I}$ produced by the previous stage and produces a *layout table* $\mathcal{L}$ (shown in Definition 4.3). In other words, we dissect the *rows* in $\mathcal{I}$ and collect information about how each array is accessed and produce a 3-tuple containing the name of the source array A, the variable to which the array access was bound to, and a *layout* that describes the desired order of dimensions of A meant to improve the locality of the array access. The layout stage does not include rows where it can not confidently determine an optimal layout or deems the work required to transform A is not worth the potential performance gain.

3. The **Transform** stage takes as input the layout table $\mathcal{L}$ produced by the previous stage and the program and produces a transformed program. We do so by walking through the program IR, creating new let-bindings of the `manifest` function applied to the

arrays and corresponding layouts dictated by the layout table $\mathcal{L}$, and replacing the names of the arrays in the relevant array access with those of the newly bound variables.

4.2 Analysis Stage

The analysis stage is perhaps the most important part of our algorithm as its result is what the rest of the transformation stages will need to process. In broad terms, what the analysis stage needs to do is boil down the program to the most essential information needed to later determine the optimal layout of arrays and to transform the program accordingly.

Definition 4.1 *Index table*
Given a program body $\mathtt{b}$, let A be the name of an array and let "$\mathtt{let}\ x = A[\mathtt{se}_0, \ldots, \mathtt{se}_n]$" be a statement in $\mathtt{b}$. Then, the result returned by the analysis stage for that statement is a 3-tuple of the form

$$(A, x, (\mathtt{se}_0, ..., \mathtt{se}_n)) \ .$$

Hence, for the entire body $\mathtt{b}$, we refer to the data structure returned by the analysis stage as an *index table* which can be described as a set of tuples:

$$\mathcal{I} = \{(A, x, (\mathtt{se}_0, ..., \mathtt{se}_n)) \mid \text{“}\mathtt{let}\ x = A[\mathtt{se}_0, ..., \mathtt{se}_n]\text{”} \in \mathtt{stms}_f(\mathtt{b})\}$$

where the function $\mathtt{stms}_f(\mathtt{b})$ returns the set of all bindings of the form "$\mathtt{let}\ x = A[\mathtt{se}_0, ..., \mathtt{se}_n]$" that occur in $\mathtt{b}$.

Note that $\mathcal{I}$ in Definition 4.1 can also be regarded as a mapping from array names, to the names which array accesses are bound to in let-bindings, to sets of scalar expressions. Note that not all arrays are necessarily mapped to an array access. These intead map to $\emptyset$. This is because, for the purpose of determining the optimal layout of arrays, we can safely ignore arrays with only a single dimension since it is not possible to change their layouts by permuting the dimensions when they only have one.
In summary, we can define the analysis stage as a function

$$\mathtt{analyze}_f : \mathtt{b} \to \mathcal{I}$$

that maps a program body $\mathtt{b}$ to an index table $\mathcal{I}$.

Example 4.2
The result of the analysis stage on the body $\mathtt{b}$ from Example 3.1 would be the following tuple:

$$\mathtt{analyze}_f(\mathtt{b}) = \left\{ \begin{array}{l} (A, x_0, (\text{“}i\mathtt{*}j\mathtt{+}\mathtt{k}_0\text{”},\ \text{“}j\mathtt{*}\mathtt{k}_1\text{”},\ \text{“}\mathtt{0}\text{”})), \\ (A, x_1, (i, j)), \\ (B, x_4, (i, j)) \end{array} \right\}$$

Pseudocode. Having established a clear definition of the data structure generated by the analysis pass, our focus now shifts to how it produces the result. Specifically, we delve into the process by which the analysis stage parses the program to generate an index table. We

Algorithm 1: `analyze(b)`

```
Input: b                                                    // Program body
Output: 𝓘                                                   // Index table
begin
    𝓘 ⟵ ∅
    foreach s in b do
        𝓘 ⟵ 𝓘 ∪ analyze_stm_f(s)
    end
    return 𝓘
end

Function analyze_stm_f(s) is
    s ⟶ "let x = e"
    case e do
        "A[se_0, ..., se_n]"          ⇒ return {(A, x, (se_0, ..., se_n))}
        "kernel A < k do b"           ⇒ return analyze_f(b)
        "loop A < k do b"             ⇒ return analyze_f(b)
        "if A then b_0 else b_1"      ⇒ return analyze_f(b_0) ∪ analyze_f(b_1)
        •                             ⇒ return ∅
    end
end
```

Figure 4: Pseudocode for the analysis stage.

can now properly define the $\texttt{analyze}_f$ function described earlier as

$$\texttt{analyze}_f(\texttt{s}_0\ \texttt{s}_1\ ...\ \texttt{s}_n) = \bigcup_{i \leq n} \texttt{analyze\_stm}_f(\texttt{s}_i)$$

where

$$\texttt{analyze\_stm}_f : \texttt{s} \to \mathit{IndexTable}$$

and

$$\begin{aligned}
&\texttt{analyze_stm}_f(\texttt{let}\ x = A[\texttt{se}_0, ..., \texttt{se}_n]) && = \{(A, x, (\texttt{se}_0, ..., \texttt{se}_n))\} \\
&\texttt{analyze_stm}_f(\texttt{let}\ x = \texttt{kernel}\ A < \texttt{k}\ \texttt{do}\ \texttt{b}) && = \texttt{analyze}_f(\texttt{b}) \\
&\texttt{analyze_stm}_f(\texttt{let}\ x = \texttt{loop}\ A < \texttt{k}\ \texttt{do}\ \texttt{b}) && = \texttt{analyze}_f(\texttt{b}) \\
&\texttt{analyze_stm}_f(\texttt{let}\ x = \texttt{if}\ A\ \texttt{then}\ \texttt{b}_0\ \texttt{else}\ \texttt{b}_1) && = \texttt{analyze}_f(\texttt{b}_0) \cup \texttt{analyze}_f(\texttt{b}_1) \\
&\texttt{analyze_stm}_f(\bullet) && = \emptyset\ .
\end{aligned}$$

This tells us that the $\texttt{analyze}_f$ function independently analyzes each statement in the program body and returns the union of the results of each analysis. The analysis of a statement by `amalyze_stm` is done by recursive pattern matching on the statement and returning a singleton index table with the appropriate tuple in the case of an array access. Statements that are neither an array access nor contain a body to recurse on are simply ignored and return the empty set. Additionally, we show the pseudocode for the analysis stage in Figure 4.

$$\mathcal{I} = \left\{ \begin{array}{l} (A, x_0, (\text{“}i\texttt{*}j\texttt{+k}_0\text{”},\ \text{“}j\texttt{*k}_1\text{”},\ \text{“0”})), \\ (A, x_1, (i, j)), \\ (B, x_4, (i, j)) \end{array} \right\}$$

$$\updownarrow$$

Array	**Binding**	**Scalar expressions**
A	x_0	(“i*j+k$_0$”, “j*k$_1$”, “0”)
A	x_1	(i, j)
B	x_4	(i, j)

Figure 5: It can be helpful to think of the data structure of the index table and layout table (though they do not exactly share the same type in the last element of their respective 3-tuples) as a table instead of a set of tuples.This figure shows the index table from Example 4.2, as such a table.

4.3 Layout Stage

Having, in the previous stage, analyzed the program and condensed it to the most essential information needed to reason about the locality of array accesses, the algorithm proceeds to the layout stage. Here, the task now becomes that of producing layouts that can be applied to arrays such that the array accesses achieve better locality.

Definition 4.3 *Layout Table*
Given a row (A, x, ν) in the index table $\mathcal{I}$, where $\nu = (se_0, \ldots, se_n)$, the layout stage produces a *layout table*, denoted $\mathcal{L}$, which is a set of 3-tuples of the form

$$(A, x, \pi)$$

where π is a layout describing the desired order of dimensions of A. Hence, the layout stage produces a layout table

$$\mathcal{L} \ : \ \{(A, x, \pi) \mid (A, x, \nu) \in \mathcal{I},\ \pi = \rho(\nu)\} \ .$$

where

$$\rho : (se_0, \ldots, se_n) \to \pi$$

is a function that maps a tuple of scalar expressions to some layout $\pi = (i_0, ..., i_n)$. Note that we use the term *row* when referring to these 3-tuples, as it may be a more intuitive way of reasoning about them as shown in Figure 5.

Before delving into the details of how the layout stage computes the layout, we must first make clear what is the desired application of the layouts. Recall that the goal of this stage of the algorithm is to produce layouts that describe the desired ordering of dimensions of arrays to improve the locality of array accesses. The fact remains that the solution to such a layout is heavily hardware-dependent. For this reason, in the following section, we will reduce the problem to a general one where the optimal solution is hardware-agnostic and later discuss the considerations that must be made to produce a hardware-specific solution.

To this end, we temporarily extend our syntax with a new iterative construct, `map`, which is a generalization of `kernel` and `loop` that is neither parallel nor sequential.

Example 4.4

```
let x2 =
  map i < k0 do
    let x1 =
      map j < k1 do
        let x0 = A[j,i]
        in x0
    in x1
in x2
```

Here, the locality of reference is not optimal, as the innermost dimension of the array is accessed by the iteration variable i, which is bound to the outermost `map`. Calculating the number of elements accessed by the iteration variables, we get that i accesses $\texttt{k}_0$ elements, and j accesses $\texttt{k}_0 \cdot \texttt{k}_1$ elements in total. This, combined with the observation that j accesses A with a stride of $\texttt{k}_0$, tells us that to achieve optimal locality, the iteration variable accessing the greatest number of elements should be the one with the small stride. Presuming this is true, we get the following conjecture.

Conjecture 4.5
Optimal locality is possible, in a nest of multiple `map` statements, only when the iteration variable with the greatest *level* is also the one that accesses the greatest number of elements in the array.

The layout stage works as follows. For each array A, array access x, and d-tuple of scalar expressions ν, in each row of the *index table* $\mathcal{I}$, the algorithm computes a layout $\pi = \rho(\nu)$. Then, if (A, x, ν, π) does not meet a set of conditions C, it omits the 3-tuple-row (A, x, π) from the *layout table* $\mathcal{L}$. We illustrate this in pseudocode in Figure 6.

Layouts. Based on Conjecture 4.5, we deduce that it is possible to produce a layout describing the order of dimensions of an array by only considering the levels of the iteration variables that access it. From this, we arrive at the idea that we can define the layout in terms of a non-strict total ordering on the *iteration variables* of the scalar expressions, i.e., the *variance* of scalar expressions. Let $\alpha_0 = IV(se_0)$ and $\alpha_1 = IV(se_1)$. If we define the empty set of iteration variables as the zero-element in the ordering, then $\alpha_0 < \alpha_1$ iff. $\alpha_0 = \emptyset$ and $\alpha_1 \neq \emptyset$. If neither set is empty we compare the iteration variables with the *greatest level* in each set:

$$\alpha_0 < \alpha_1 \qquad \text{iff.} \qquad \max(\{\texttt{level}_f\,(x) \mid x \in \alpha_0\}) < \max(\{\texttt{level}_f\,(x) \mid x \in \alpha_1\})$$

We then sort each scalar expression in each ν of the layout table based on this ordering. Note that this is a very naive way of ordering them based on the reduced scope of the problem as explained earlier.

Conditions. As mentioned before, after producing layouts, the algorithm omits rows from the resulting layout table that do not meet a set of conditions C. These conditions are meant to eliminate layouts that would lead to inefficient array accesses; manifests that can not be

Algorithm 2: *layout table*

```
Input: 𝓘                                                    // Index table
Output: 𝓛                                                   // Layout table
begin
    𝓛 ⟵ ∅
    foreach (A, x, ν) in 𝓘 do
        π ⟵ ρ(ν)
        discard ⟵ False
        // is the layout a good layout?
        foreach p in C do
            if p(A, x, ν, π) then discard ⟵ True
        end
        if not discard then
            𝓛 ⟵ 𝓛 ∪ {(A, x, π)}
        end
    end
    return 𝓛
end
```

Figure 6: Pseudocode for the layout stage.

done efficiently due to having to spend more work on copying values than would be saved by the improved locality; or where the arrays are accessed by scalar some expression that is too complex to confidently reason about. The conditions in C are the following.

$$P(A,x,\nu,\pi) = \begin{cases} \texttt{True} & \text{if} & IV(\texttt{se}_d) = \emptyset & (a) \\ \texttt{True} & \text{if} & \texttt{level}_f(A) > 0 & (b) \\ \texttt{True} & \text{if} & \neg\texttt{is\_map\_transpose}_f(\pi) & (c) \\ \texttt{True} & \text{if} & \pi = (0,\ldots,d) & (d) \\ \texttt{True} & \text{if} & \exists \texttt{se} \in \nu \text{ s.t. } \texttt{inscrutable}_f(\texttt{se}) & (e) \\ \texttt{True} & \text{if} & \exists \texttt{se} \in \nu \text{ s.t. } \texttt{stride}_f(\texttt{se}) > k & (f) \\ \texttt{False} & \text{otherwise} & & \end{cases}$$

Here, $\texttt{is\_map\_transpose}_f$[2] is a function that checks if the layout corresponds to a transposition, $\texttt{inscrutable}_f$ is a function that checks if any of the scalar expressions in ν are too complex to reason about, and $\texttt{stride}_f$ is a function that, given some scalar expression, which depends on some iteration variable i, computes the stride of i on the array it accesses. For example, if `se` is a scalar expression that reduces to the form "$i*\texttt{s} + \texttt{o}$", then the stride of i is `s`.

(a) Is the last scalar expression in ν invariant? This check is needed due to Premise 6.

(b) Is the array A allocated top-level? If not, we don't want to manifest it, since it would likely lead to a large number of unnecessary memory transactions.

[2]The Futhark-source equivalent is called `isMapTranspose`, found in `src/Futhark/IR/Prop/Rearrange.hs`

(c) Is the layout π a mapping of a transposition? I.e., can the layout of A be transformed to that of π efficiently?

(d) Is π the identity layout? In this case, (assuming Premise 2) we don't want to manifest the array, since it is already in row-major order.

(e) Are any of the scalar expressions in ν too complex to reason about?

(f) Do any of the scalar expressions in ν have a stride larger than some arbitrarily chosen threshold k.

The argument for having this many conditions is that we need to be very confident about the usefulness of the transformation; it is often faster to not do anything, rather than doing something.

Hardware Specifics. As mentioned earlier, we reduced the problem of finding the optimal layout to a hardware-agnostic one. The technique described above only works for the general case where you have a nest of similar iterators, which on GPU translates a nest of exclusively `kernel`s and, on CPU, a nest of exclusively `loop`s. However, on both GPU and CPU, you can also have a mixed nest of `kernel`s and `loop`s. In such a situation, the optimal layouts are not necessarily similar to each other. For example, consider the following program

Example 4.6

```
let x2 =
  kernel i < k0 do
    let x1 =
      loop j < k1 do
        let x0 = A[i,j]
        in x0
    in x1
in x2
```

In this case, on the GPU, the kernel variable i would have strided access to A with a stride of $\texttt{k}_1$, which, if A happens to be allocated in global memory, means that we should manifest A with the layout $(1, 0)$ to get coalesced access. However, on the CPU, the story is different. Here, in order to achieve optimal cache locality, we would *not* want to manifest A. The reason for this is that if we manifested A with the layout $(1, 0)$, then the loop variable j would have strided access to memory, which would lead to poor L1 and L2 cache locality which are not shared among CPU cores [3]. For these reasons, the ordering described earlier would be

$$\emptyset < \texttt{loop} < \texttt{kernel}$$

on GPU, and

$$\emptyset < \texttt{kernel} < \texttt{loop}$$

on CPU.

4.4 Transformation Stage

The final stage of our pass is the transformation stage. This is the stage that transforms the program, inserting `manifest` statements where appropriate based on the output of the layout stage. It takes as input the layout table $\mathcal{L}$ from the layout stage as well as the program body b, which can be represented as a sequence of n statements $\overline{s} = (s_0, s_1, ..., s_n)$, and produces a new sequence of $m \geq n$ statements $\overline{s}' = (s'_0, s'_1, ..., s'_m)$ of which $m - n$ are `manifest` statements.

More concretely, the transformation stage iterates over the statements in $\overline{s}$, and when it encounters a let-binding with a `loop` or `kernel` expression, it recursively traverses the body of the expression. Whenever it encounters an access to an array x, it looks up the name of the array and the name of the index expression in the layout table $\mathcal{L}$. If the access represented in $\mathcal{L}$ with a layout π, it inserts a new let-binding `let` $x' =$ `manifest`(π, x) in the outermost possible scope of the program, and replaces array name in the access with x'. To avoid manifesting the same array multiple times, it also keeps track of the arrays that have already been manifested in a set $\mathcal{M}$ and only manifests an array if it is not already in $\mathcal{M}$.

We show the pseudocode for the transformation stage in Figure 7. Note that when inserting the new `let`-bindings, the pseudocode shows them being inserted in the same scope as the array access. Because of this, to ensure that the new bindings are inserted in the outermost possible scope, the algorithm relies on a subsequent compiler pass to hoist the bindings to the outermost possible scope, and indeed this is also what we do in our implementation. The reason we want to insert the bindings in the outermost possible scope is that the transformations performed by the use of the `manifesst` function are done at runtime, which, if inside a loop body, could result in many redundant copies from memory.

Example 4.7
Applying the previous two stages of the algorithm to the program in Example 4.6 yields the layout table $\mathcal{L} = \{(A, x_0, (1, 0))\}$. Given this, applying the transformation stage to the program, using layout scheme we described for GPU, yields the following program:

```
let A' = manifest((1,0),A)
let x2 =
kernel i < k0 do
    let x1 =
    loop j < k1 do
        let x0 = A'[i,j]
        in x0
    in x1
in x2
```

Algorithm 3: `transform`$(\mathcal{L}, b)$

```
Input: 𝓛                                                    // Layout table
Input: b                                                    // Program body
Output: b′                                                  // Transformed program body
begin
    b′ ⟵ ∅
    𝓜 ⟵ ∅
    foreach s in b do
        s → "let x₁ = e"
        case e do
            "se" ⇒
                b′ ⟵ b′ ∪ {s}
            "A[se₀, ..., seₙ]" ⇒
                if (A, x₁, π) in 𝓛 then      // Lookup array name in layout table.
                    if (A′, A, π) in 𝓜 then  // Check if A is already manifested.
                        s′ ⟵ "let x₁ = A′[se₀, ..., seₙ]"
                        b′ ⟵ b′ ∪ {s′}
                    else                                    // Not yet manifested.
                        e′ ⟵ manifest(π, A)
                        A′ ⟵ new_name()
                        𝓜 ⟵ 𝓜 ∪ {(A′, A, π)}
                        s′ ⟵ "let x′ = e′"
                        b′ ⟵ b′ ∪ {s′}
                    end
                end
            "if x₀ then b₀ else b₁" ⇒
                b′₀ ⟵ transform(b₀)
                b′₁ ⟵ transform(b₁)
                e′ ⟵ if x₀ then b′₀ else b′₁
                s′ ⟵ "let x₁ = e′"
                b′ ⟵ b′ ∪ {s′}
            "kernel x₀ < k do b₀" ⇒
                b′₀ ⟵ transform(b₀)
                e′ ⟵ kernel x₀ < v do b′₀
                s′ ⟵ "let x₁ = e′"
                b′ ⟵ b′ ∪ {s′}
            "loop x₀ < k do b₀" ⇒
                b′₀ ⟵ transform(b₀)
                e′ ⟵ loop x₀ < v do b′₀
                s′ ⟵ "let x₁ = e′"
                b′ ⟵ b′ ∪ {s′}
            "manifest ([k₀, ..., kₙ], A)" ⇒
                b′ ⟵ b′ ∪ {s}
        end
    end
    return b′
end
```

Figure 7: Pseudocode for the transformation stage.

5 Implementation

This section walks through the implementation of the stages of our algorithm in detail. A new pass to the Futhark compiler, named `GALOP` contains the implementation of the algorithm. Since the code for the analysis stage could potentially be useful in other compiler-passes, it is located in a separate file named `AccessPattern.hs`.

Whereas the previous section described the algorithm based on a reduced problem space, skipping over some technicalities that would make the algorithm more complicated to describe, the primary focus of this section is to describe the so-called "dirty details" of the implementation necessitated by the technicalities that were skipped over. Hence, this section is not a line-by-line description of the implementation, but rather a description of the implementation in broad terms with a focus on the data structure and the differences between the actual implementation and the algorithm described in the previous section.

5.1 Analysis Stage

Given some access to an array, e.g., `A[i,j,k,foo]`, we want to be able to reason about the access pattern of each dimension of the array in order to determine the optimal memory layout of the array. To this end, we need to be able to answer the following questions about accesses to each dimension of the array:

1. What are the dependencies, i.e., free variables of the scalar expressions used to access the array? Recall Premise 5. In reality, we not only want to consider *iteration variables* but also constants and variables.

2. For each free variable, what is the level at which it is defined in the nest of bodies of the program?

3. For each free variable, what is its type? That is, is it a kernel variable, a loop variable, constant, or something else? The reason we need to store this information is to be able to distinguish between loop variables, kernel variables, and other variables. In Section 4, this was assumed to be implicitly known.

This information by itself is quite powerful. Recall that our working assumption is that we can create good locality optimizations simply by determining optimal layouts that describe the order of dimensions of an array such that, on the GPU backend, kernel variables access the innermost dimensions, i.e., have the smallest possible strides in memory and, on the multicore backend, sequential loop variables should access the innermost dimensions for optimal cache locality. The information described above is all we need to determine such layouts, and it translates to the following structure in Haskell for each dimension in the array access:

```haskell
data DimAccess rep = DimAccess
{ dependencies :: M.Map VName Dependency,
  originalVar :: Maybe VName
}

data Dependency = Dependency
{ lvl :: Int,
  varType :: VarType
}

data VarType
= ConstType
| Variable
| ThreadID
| LoopVar
```

Here, `dependencies` in `DimAccess` is a map from the names of the free variables, to their respective instance of the `Dependency` class, each of which contains the relevant information mentioned above. The `lvl` contains the answer to question 2 and the `VarType` stores the answer to question 3. For the answer to question 1, we need to store the aforementioned information for each dimension of the array access since there may be multiple. To do this, we need to store a `DimAccess` for each dimension of the array access. Hence, the answer to question 1 is represented by a list `[DimAccess]`. In summary, a `DimAccess` represents a single dimension in an array access, and a `[DimAccess]` represents an entire array access. The `originalVar` field is used to store the name of the original expression from which `dependencies` was computed.

In broad terms, the analysis stage is a traversal of the IR of the program where we collect information about memory accesses in the program. More concretely, the program performs a map over each kernel body in the program and analyses each array access in that kernel body. For each array access, it aggregates a `[DimAccess]` with information about the access to each dimension of the array. For each `[DimAccess]` we want to know the following:

- The name of its associated `Index` expression.
- The name of the array it accesses.
- The name of the `segOp` (i.e., kernel) in which the array is accessed.

Hence, the output of the analysis stage is the following mapping:

```haskell
type IndexTable rep = M.Map SegOpName (M.Map ArrayName (M.Map IndexExprName [DimAccess rep]))

type SegOpName = VName
type IndexExprName = VName
type ArrayName = (VName, [BodyType], [Int])
```

The reader might notice that the `IndexTable` corresponds to the index table from Definition 4.1 with the addition of the `SegOpName`. In our description of the algorithm in Section 4, this information about the `segOpName` was omitted from the definitions of index table since it is not necessary for the algorithm to work correctly. The reason we store this information is an optimization; when traversing the IR in the transformation stage, we can look up in

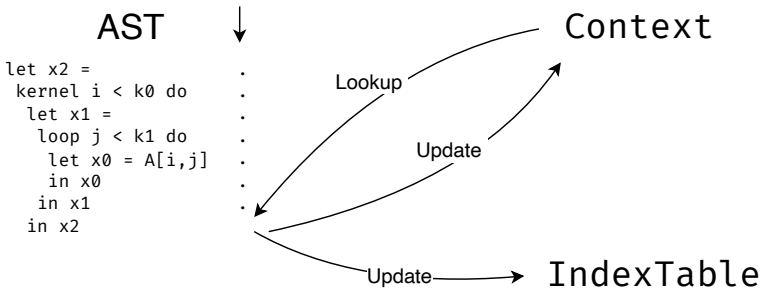


Figure 8: Illustration of the use of the context during the traversal of the program AST in the analysis stage of our pass.

the `IndexTable` the name of any `SegOp` we encounter. If it is not present, we don't need to traverse its body, since we know that it does not contain any array accesses that we want to manifest.

Furthermore, the reader might notice that the `ArrayName` is a tuple of three elements. The first element is the name of the array as one would expect, the second element is a list of `BodyType`, and the third element is a list of integers. The use of the `[BodyType]` list is explained in Section 5.3. To understand the need for the list of integers, recall from Premise 2 that the design of our algorithm is based on the assumption that all arrays are allocated in *row-major* order. In reality, this might not always be the case. For this reason, we need to store the existing layout of the array, which is what the list of integers is for. Without this, the algorithm might naively decide to manifest an array that is allocated in *column-major* order and already has optimal locality of reference, leading to a performance regression.

Context. To reduce scalar expressions used to access an array to their iteration variables during the traversal and compute the nest level of each `SegOp` and the `[BodyType]` for each array, we need to store information about the patterns we have reached so far at any given point in the traversal. In Section 4, this information was obtained using *oracle-functions* that were able to look up the information in the IR of the program. In reality, we need to store this information ourselves. We refer to this aggregate of information as the *context* which is illustrated in Figure 8. As an example, if some scalar expression $\texttt{se}_0$ depends on some expression $\texttt{se}_1$, then to reduce the dependencies of $\texttt{se}_0$, we need to be able to look up the dependencies of $\texttt{se}_1$ in the context. Hence, we must keep track of the following information in the context:

1. For each let-binding, we must keep track of the dependencies (i.e., free variables) of the pattern, its `VarType` type, and its level. This corresponds to the information provided by the oracle functions IV and $\texttt{level}_f$ in our description of our algorithm in Section 4. We need the `VarType` when reducing the dependencies of a scalar expression to determine whether it is a kernel variable, loop variable, or something else. With this information, we know that we can stop trying to reduce the dependencies further whenever we meet an iteration variable or a constant. Lastly, we need to store a sequence containing the `BodyType` of each `SegOp`, `loop`, or `If`-statement encountered up until the point in the traversal where we reached the let-binding. This is needed to determine the `[BodyType]` for each `ArrayName` in the `IndexTable`.

2. A sequence containing the `BodyType` of each `SegOp`, `loop`, or `If`-statement encountered up until the current point in the traversal. This is needed to determine the same information in point 1.

3. The level of recursion at any given point in the traversal. This is needed to label the level of each `SegOp` encountered which is needed to calculate the `level` required by point 1.

4. Information about each *slice* encountered during the traversal. Slices will be explained shortly.

Point 1 motivates the following type:

```
data VariableInfo rep = VariableInfo
  { deps :: Names,
    level :: Int,
    variableType :: VarType,
    parents_nest :: [BodyType],
  }
```

The remaining points motivate the following type:

```
data Context rep = Context
{ assignments :: M.Map VName (VariableInfo rep),
  parents :: [BodyType],
  currentLevel :: Int,
  slices :: M.Map IndexExprName (ArrayName, [VName], [DimAccess rep])
}
```

We compute the context during traversal of the IR of the program but since the IR does not change, it could just as well have been precomputed, which might enable simpler code. We leave this as future work.

Slices. An important aspect of Futhark's IR, which we have not covered in Section 4 is that of *slices*. In some situations, Futhark's compiler may decide to compile an array access to use a *slice*. A slice can be regarded as a view of a dimension of an array. In Futhark, a slice is essentially just syntax sugar for some linear access pattern into a dimension of an array. A slice is defined by an offset, a size, and a stride. The offset is the index of the first element of the dimension, the size is the number of elements to access, and the stride is the distance between each element. For example, in the array access `A[i, 0 :+ n * 1]`, the last dimension is sliced with an offset of `0`, a size of `n`, and a stride of `1`, which corresponds to accessing the first `n` consecutive elements of the dimension.

Example 5.1
Consider the futhark program from Example 2.1. Before the simplifying compiler pass is applied, the program will result in IR of the following form.

```
let x3 =
  kernel i < k0 do
    let s = A[i, 0 :+ m * 1]
    let x2 =
      loop a = 0 for j < k1 do
        let x0 = s[j]
        let x1 = a + x0
        in x1
    in x2
  in x3
```

Here, the array A is sliced in the last dimension, and the result, which is bound to s, is then accessed in the loop body by the loop variable `j`. Note that, for this example only, we have extended our syntax with a for-loop and an accumulator variable.

To reduce the problem space of the implementation concerning slices, we make the following assumption about slices.

Premise 7
The result of a slice is always accessed sequentially by a loop variable. In other words, any array access with a slice is bound to a name, which is then accessed inside a loop body by a loop variable, just like an array would be.

Given some access to a dimension of an array by a slice, Premise 7 allows us to represent it in the `IndexTable` as a single `DimAccess` with a single dependency of type `LoopVar`, which is the same as if the dimension was accessed by a loop variable. To do this, we also construct a placeholder name for the dependency. This allows the layout stage to treat slices in the same way as accesses by loop variables and compute the layout of the array accordingly, which, in turn, allows the transformation stage to manifest the array that is sliced.

Premise 7 is not necessarily always true, but we believe it holds in most cases. This needs more investigation in the future. Regardless, our implementation is highly modular and can easily be changed to handle slices differently if the need arises.

Note that the compiler passes that precede that of our algorithm will remove many slices, but not all. Hence, the handling of slices is still necessary, but will likely only have an effect in rare cases. For example, the IR of Example 5.1 will be transformed to the following by applying the simplifying compiler pass.

```
let x3 =
  kernel i < k0 do
    let x2 =
      loop a = 0 for j < k1 do
        let x0 = A[i,j]
        let x1 = a + x0
        in x1
    in x2
  in x3
```

Here, the slice has simply been removed and the array is accessed by the loop variable j in the same expresssion as the access by i.

The motivation behind the `slices` field in the `Context` class is that if we naively analyze an access to a slice in the same manner as an array access, then we would only consider manifesting the name which the *slice* is bound to instead of the *array* that is sliced. Whenever we encounter an access, being able to look up in the `Context` whether it is an access

to the result of a slice allows us to avoid this problem. For example, when the analysis encounters the access x_0 from Example 5.1, it produces an entry in the `IndexTable` of the form

```
x_3 : { A : { s : [ DimAccess {
                      originalVar = Nothing,
                      dependencies = {
                        i : Dependency {
                          lvl = 0,
                          varType = ThreadID
                        }
                      }
                    },
                    DimAccess {
                      originalVar = j,
                      dependencies = {
                        slice_0 : Dependency {
                          lvl = 1,
                          varType = LoopVar
                        }
                      }
                    }
                  ] } }
```

Here, the two accesses s and x_0 in Example 5.1 have been combined into a single entry in the `IndexTable`, where the array name is A, instead of s. Also note that the access to the last dimension of the array is represented by j, which, instead of treating as a loop variable, we treat as a scalar expression that depends on our fake loop variable *slice*_0. In the future, the this should be changed to use the actual iteration variable that iterates over the slice, e.g., j in the Example 5.1, instead of a *fake* dependency. This would eliminate the need for Premise 7 and allow us to handle slices more generally.

Generalization to Other Backends. At this point, the reader may have wondered about the purpose of the `rep` type constraint used throughout types and classes shown above. The purpose of this `rep` (short for representation) is to generalize code to multiple different backends. `rep` may, for example, take on the values `MC` for the multicore backend or `GPU` for the GPU backend. This makes it possible to reuse code that is the same regardless of the backend. This especially comes in handy in the layout stage of the algorithm the calculation of layouts is highly dependent on the backend.

One crucial difference between the algorithm described in Section 4 and the actual implementation is that in the implementation, the analysis stage does not produce a table containing the scalar expressions used to access the dimensions of arrays, instead, the analysis stage reduces the scalar expressions to their free variables. This means that the information produced by that stage is not sufficient to determine all the conditions described in Section 4.3. To resolve this, we have implemented a second analytical stage, which we refer to as the *PrimExp analysis stage.*

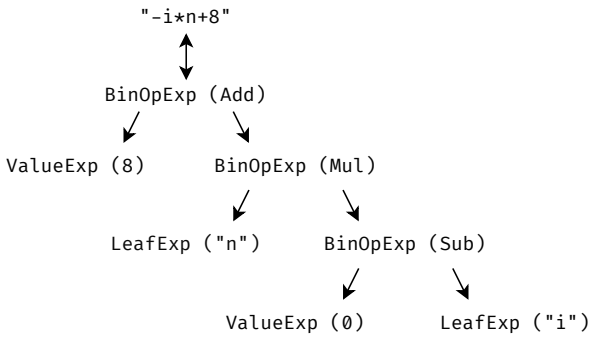


Figure 9: Illustration of the scalar expression "`-i*n+8`" converted to a `PrimExp`, represented as a binary tree of `PrimExp`s. Note that to aid the illustration, we have simplified the `PrimType` and `PrimValue` types.

5.2 PrimExp Analysis Stage

The purpose of this stage is quite simple: attempt to convert each scalar expression in the program to a `PrimExp` and store the result in a table along with its name such that we can look it up when needed (Notice that this is very much akin to the implementation of an *oracle-function*). We refer to this table as the *PrimExp table* and the implementation is the following:

```
type PrimExpTable = M.Map VName (Maybe (PrimExp VName))
```

Here, `PrimExp` is a type from the Futhark compiler that represents primitive expressions.

```
data PrimExp v
  = LeafExp v PrimType
  | ValueExp PrimValue
  | BinOpExp BinOp (PrimExp v) (PrimExp v)
  | CmpOpExp CmpOp (PrimExp v) (PrimExp v)
  | UnOpExp UnOp (PrimExp v)
  | ConvOpExp ConvOp (PrimExp v)
  | FunExp String [PrimExp v] PrimType
```

Note that not every scalar expression can be converted to a `PrimExp`. Hence, the `PrimExpTable` is a map from the name of the scalar expression to a `Maybe (PrimExp VName)`. If the scalar expression can be converted to a `PrimExp`, then the value is `Just (PrimExp VName)`, otherwise it is `Nothing`. We illustrate an example of a `PrimExp` as a tree in Figure 9.

The PrimExp analysis stage constructs the `PrimExpTable` by recursively traversing the IR of the program and attempting to convert each scalar expression it comes across to a `PrimExp`. This traversal makes use of the existing `Walker` class from the Futhark compiler, which enables us to succinctly define how to map over the different types of syntax nodes in the IR. The layout stage described in the next section traverses the IR in a mostly similar fashion. This is unlike the implementation of the analysis stage, which uses a custom implementation of the traversal that makes no use of monads. Refactoring the implementation of the analysis stage to traverse the IR in the same fashion as the layout stage and transformation stage could simplify the code and make it more maintainable. We leave this as future work.

5.3 Layout Stage

Recall from Section 4.3 that the purpose of the layout stage is to compute a layout table. The same holds for the implementation. However, like the implementation of the index table described in Section 5.1, the data structure proposed does not match that of the implementation exactly. Specifically, the implementation of the layout table also adds a `SegOpName` column to the table:

```
type LayoutTable = M.Map SegOpName (M.Map ArrayName (M.Map IndexExprName Layout))
```

Aside from this change, the implementation of the layout table is mostly analogous to the description in Section 4.3. The layout evaluates each row in the `IndexTable` and attempts to determine an optimal layout for the array. If no such layout can be found, or if the layout is deemed to be too expensive to apply, or the array should not be transformed for some other reason, which we will get to in a moment, then the row is discarded from the table.

One technicality that was skipped over in Section 4.3 is that all arrays that are defined in a `SegMap` at group-level are either allocated in shared memory or register memory, while arrays defined in a `SegMap` at thread-level are allocated in global memory. Manifesting arrays allocated in shared memory or register memory would bring no benefit and would only incur unnecessary overhead. It is for this reason, that we store the list `[BodyType]` in the `ArrayName`, as mentioned earlier, since it allows us to check this group or thread *level* of the `SegMap` in which the array is defined.

Note that our implementation only considers array accesses that are nested inside the body of a kernel. In principle, manifesting arrays that are accessed inside the body of a lop-level sequential loop could also be beneficial on the multicore backend, but we leave this consideration as future work.

Complexity Analysis. Given some access to an array, we want to be able to reason about the access so we can determine the optimal memory layout of the array. However, there are cases where the scalar expression used to access a dimension of an array is so complex that we can't confidently determine its effect on the locality of reference. We refer to such expressions as being *inscrutable*. As an example, if we have an array access, like `A[B[i]]`, where the access is a function of some access to another array whose values are unknown at compile-time, then we can't possibly reason about the access pattern. Another example is if the access is a function of a single iteration variable that consists of simple arithmetic operations on the iteration variable. For example, take the access `xss[i*2]`, where `i` is an iteration variable. In this example, the scalar expression can reduced to a strided access with a stride of two, and it is not inscrutable. However, the task of deciding the threshold for an acceptable stride is not trivial since it is highly dependent on the hardware. For this reason, we leave this as future work.

Premise 8
In general, if the access to a dimension of an array is a function of something that can't be determined at compile-time, or is too complex to reason about logically, then we will consider it inscrutable.

This is what motivates the need for the PrimExp analysis stage. Given a `PrimExpTable`, the layout stage can determine if a scalar expression is inscrutable or not or if the stride of

an iteration variable is too large. Note that this is why we need the `originalVar` field in the `DimAccess` for; we need to know which scalar expression to look up in the `PrimExpTable`. Our implementation considers a scalar expression to be inscrutable if it is **neither** a `LeafExp`, `ValueExp`, nor a `BinOpExp` where neither of the two operands recursively evaluate to *inscrutable*. It also considers a scalar expression to be inscrutable if it can't be reduced to a single iteration variable with a stride of less than 8. Our implementation handles strided accesses in order to be on par with the original pass `babysitKernels` which is also able to manifest some cases of arrays with strided accesses. If our implementation was not able to reduce such accesses, then it would only be able to handle scalar expressions that reduce to some iteration variable with a stride of one, or those that can be reduced to some constant. Our implementation is only able to reduce strides of scalar expressions that are functions of the addition, subtraction, and multiplication operators. However, we have implemented it in a way that makes it easy to extend to handle more operators in the future.

When the compiler encounters an inscrutable access, we don't need to consider manifesting the array, so we can just ignore it. There are two possible approaches to handle inscrutable accesses in the implementation. The first is akin to a blacklist, where we only ignore expressions that we can determine to be inscrutable. This is not optimal, as the implementation might be missing functionality to handle all possible inscrutable expressions. This could lead to the compiler manifesting array accesses that already have good locality of reference, making it worse. The second approach, which we have used, is akin to a whitelist, where we by default assume that all expressions are inscrutable, and only consider manifesting expressions that the implementation can specifically handle. This is more robust and makes sure that we only manifest arrays when we can confidently determine that we can improve the locality of reference of their accesses. It also makes it possible to expand on the implementation in the future, adding functionality to handle more complex expressions and by extension the ability to improve the locality of reference of more array accesses.

5.4 Transform Stage

The transformation stage works in much the same way as the PrimExp analysis stage. It traverses the IR of the program using the `Walker` class from the Futhark and manifests arrays. Concretely, whenever it encounters an array access that is present in the layout table, it checks whether or not the array has already been manifested. It does this by keeping track of which arrays have already been manifested and for those that have, the new name to which the manifested array was bound. If the array has already been manifested it simply changes the name of array that is being accessed to the name of the manifested array. If the array has not been manifested, it also inserts a new let-binding with the manifested array.

Note that manifests are inserted in the same scope as the accessing expressions. Hence, we rely on the `simplify` pass of the Futhark compiler to lift them to the outermost possible scope.

6 Evaluation

This section details how we validated the correctness the implementation of our algorithm. Secondly, we evaluate the performance of our implementation by providing benchmarks that compare our solution and `babysitKernels`, with not performing any locality optimizations.

6.1 Validation

To validate the correctness of our implementation, we have employed the following test strategy.

1. We provide a suite of unit tests to validate the correctness of various functions from each stage of our implementation. These tests expand the existing suite of unit tests in the Futhark compiler and can be run using the `make unittest` command.

2. As integration tests, we employ Futharks existing test suite to check if our implementation breaks any existing functionality. These can be run with the command `futhark test --backend=cuda tests`.

3. We provide a suite of black-box tests, using custom bash scripts that use the IR generated from Futhark programs to check whether each stage computes the output that we expect for a range of different inputs that, we believe, cover the most common cases as well as most edge cases. Note that tests could be incorporated into the existing suite of integration tests, but we leave this as future work.

4. Lastly, we employ an existing suite of benchmark tests to evaluate the performance of our implementation.

Note that two (out of 2265) of the integration tests fail. One is because our algorithm decides to manifest an accumulator in a loop, which is not desired behavior. We leave this as a future point of improvement. The second test fails because our algorithm does not produce the expected manifest. This is because the PrimExp analysis is not able to handle `min` or `max` expressions, which we leave as future work.

Analysis Stage

To validate the correctness of the analysis stage, we have provided the file `AnalyseTests.hs` containing unit tests for the analysis stage as well as the source directory **tests_analysis**[3] which includes a test script `scripts/test.sh` along with a series of test files that each contain a program represented in Futharks IR as well as the original program in a header comment, such that each test can be reproduced if necessary, and finally, a comment providing the pretty-printed expected output from the analysis. The script runs the analysis on each test file and compares the output with the expected output from the test file.

Each test file contains a minimal Futhark program intended to cover a specific case that the analysis should be able to handle. The tests cover the following cases of expected behavior:

[3]Url: `https://github.com/0undefined/futhark/tree/master/tests_analysis`

- Accesses nested inside several loops/kernels are properly caught and represented in the index table. This is for example checked by the file `multi_nested.fut_gpu`.
- Acceses to multiple dimensions of an array. This is checked by a number of files named `multi_*.fut_gpu`.
- It can properly handle array accesses by scalar expressions that depend on several iteration variables (`multi.fut_gpu`) as well as constants (`var_index.fut_gpu`).
- Indirect accesses, i.e. using the result array accesses to access other arrays, are handled correctly. Several edge cases related to this are checked by a number of files named `indirect_*.fut_gpu`.
- Accesses that only depend on constants, even though they reduce to nothing, should still be represented in the index table. This is for example checked by `multi_diff.fut_gpu`.
- Slices are analyzed as we expect. This is checked by the file `slice.fut_gpu`.
- When the order of dimensions accessed does not correspond to the iteration variables were bound. This, as well as various combinations of nests of loops and kernels, are checked by files like `par_seq_par.fut_gpu` and `par_par_seq.fut_gpu`.

We have chosen to design the tests around the IR as we can simply apply the analysis stage by itself and evaluate it independently from the other compiler passes. Additionally, if we apply the analysis stage to a Futhark program in the source language, the resulting IR might be different in the future as the preceding compiler passes undergo changes and improvements. For example, even small changes to the preceding compiler-passes can cause the variable names in the IR to change, which would cause the tests to fail. Hence, applying the pass to IR, improves the maintainability, as they don't need to be updated each time a change is made to preceding compiler passes. However, future changes to the compiler may also cause the tests to become irrelevant, as the compiler might no longer be able to produce the IR.

Note that the IR-based tests only cover the GPU backend. However, much of the functionality required for these tests to pass is shared between the GPU and MC backends, hence they are still useful for testing the MC backend. The unit tests also only cover the GPU backend. We leave the task of writing unit tests for the MC backend as future work.

Also note that the unit tests only consider the output of the analysis stage, i.e., the `IndexTable`, and not whether the `Context` contains the values we expect. This is because the context is itself an implementation detail that may change in the future.

To invoke the analysis stage of our implementation using Futharks `dev` argument we provide the `--memory-access-pattern` argument, or `-z` for short. This runs the analysis on the given program and pretty-prints the resulting index table. Specifically, the pretty-printer will show for each array access, the `SegOpName` and the type of the `SegOp`; the `ArrayName` with the layout `[Int]` and without the `[BodyType]` if the list is empty; and the array access represented with a line for each dimension of the array, where each line contains the `originalVar` and the set of reduced iteration variables along with the corresponding level and abbreviated `VarType` of each. When the `originalVar` is the same as the only variable in the set of dependencies, we omit the `originalVar` and just show the set of iteration variables.

Otherwise, we use the special syntax `x -> {...}` to specify that `x` is the `originalVar` and `{...}` is the set of iteration variables.

At the time of writing, all tests for the analysis stage successfully pass. Note that the absence of test failures does not guarantee the absolute correctness of the analysis stage, as the tests conducted may not encompass every conceivable scenario. Nevertheless, we are confident that these tests cover the most important functionality of the analysis stage.

PrimExp Analysis

For the PrimExp analysis stage, we provide unit tests in the file `AnalysePrimExpTests.hs` that check whether the analysis constructs the expected `PrimExp` given some AST. The file includes equivalent pairs of tests for both the GPU and MC backends.

The tests cover the following cases of expected behavior:

- Loop and kernel variables are correctly identified and converted to `PrimExp`.
- The analysis can properly traverse into the bodies of loops, kernels, and conditionals.
- Constants, variables, and binary operations are correctly converted to `PrimExp`.

At the time of writing, all tests for the PrimExp analysis stage successfully pass.

Layout Stage

For the layout stage, we provide unit tests as well as a test script that, like the analysis stage, runs the layout stage on a series of test files and compares the output with the expected output from the file. These tests also contain minimal Futhark programs represented in Futharks IR along with the expected outputs. The tests cover the following properties:

1. Does it compute the layouts we expect, given different array accesses?
2. Layout rejection: Does the implementation correctly identify layouts that should be rejected? Does it keep the layouts that should not?

6.1.1 Transformation Stage

We do not provide any tests for the transformation stage and leave this as future work. The reason for this is mainly that it is difficult to make unit tests for this stage, as it requires the AST of full programs and layout tables as units of input, the size of which makes it difficult to maintain. Alternatively, one could make test files in the same style as those for the analysis and layout stages.

At the time of writing, all tests for the layout stage successfully pass.

6.2 Performance Comparison

To evaluate the performance of our algorithm on GPU, we have run benchmarks on two different machines with different GPUs. We refer to these machines by the names of their respective GPUs. We will refer to these machines as “A100” and “RTX 2070 Super”, as shown in Table 1, without the brand name and connector type.

GPU Model	VRAM	Cuda Cores	Architecture	Nickname
NVidia A100 (PCIe)[4]	40GB	8192 [9]	Volta	A100
NVidia RTX 2070 Super	8GB	2176	Turing	RTX 2070 Super

Table 1: Though the A100 supports a higher host-device memory bandwidth, it was installed in the same configuration as the RTX 2070 Super (PCIe generations). These machines are used for the benchmarks with the Cuda (GPGPU) backend.

CPU Model	Cores/Threads	RAM	Nickname
AMD Ryzen™ 7 5700X	8/16	32Gi	"16c"

Table 2: We have named the machine after how many threads it has available, suffixed with *c*. The machine is used for the benchmarks with the multi-core backend.

All GPU benchmarks are run 250 times each on both machines using Futharks built-in `bench` command. This reduces the variance of the wall-clock time results. Regardless of the fact that the A100 machine runs in a shared cluster, where no one has exclusive access to the machine, the results on both machines are very consistent, as Futhark only measures the time of computations, and not the time it takes to transfer data to and from the GPU.

There are subtle differences between the programs generated using GALOP, `babysitKernels`, and performing no locality optimizations, which we will refer to as "no-op" (no optimization), throughout this section. Whether the differences introduced by our algorithm and `babysitKernels` leads to a positive performance impact depends on a variety of factors: the input data sizes, hardware, and the program itself can have a *big* impact on performance.

All of the examples are taken from the benchmarks. We have limited the benchmarks to exclude benchmarks where all implementations, ours, `babysitKernels`, and *no-op*, would produce the same program which eliminates 35 benchmarks. As we are working with GPUS we focus on benchmarks with large datasets, we therefore also remove small benchmarks with less than 16MB. We also exclude benchmarks where the normalized geometric averages are all within ± 0.008 of each other which removes another 17 benchmarks. Note that 0.008 is an arbitrarily chosen value. We normalized the benchmarks relative to *no-op*. Lastly, we exclude benchmarks where the geometric average for each backend is faster than 1 millisecond.

Because of the differences in hardware specifications we have excluded any benchmarks requiring more than 16GB of memory for the RTX 2070 Super GPU.

The full set of benchmark programs can be found at **github.com/diku-dk**[5]

- `poseidon-bench.fut`

  Here, `babysitKernels` makes *a lot* of manifests. GALOP inserts no manifests in this benchmark, as all of the `babysitKernels`-manifested arrays are allocated inside of either a loop, kernel, or both. As is shown in Table 3, on this benchmark GALOP performs better than `babysitKernels` where we observe a significant slowdown on the A100 GPU. However, `babysitKernels` sees a significant performance gain on the RTX 2070 Super GPU. That is, doing nothing seems to be the best strategy on the A100, while the manifests inserted by `babysitKernels` seem to be beneficial on the RTX

[4]The A100 was released with two different connection types, and though they have almost identical specifications they have some key differences, most notably thermal thermal design power (TDP) [10].

[5]Url: `https://github.com/diku-dk/futhark-benchmarks/tree/dec2ebe8`

speedup	benchmark & data set – 2070super
1.20	finpar/OptionPricing.fut OptionPricing-data/medium.in
1.17	pbbs/convexHull/convexhull.fut data/2DinSphere_100M.in
1.16	micro/scan.fut:sum_iota_i8 #0 (”10000i32”)
1.14	finpar/OptionPricing.fut OptionPricing-data/small.in
1.12	micro/reduce.fut:sum_f32 [1000000]i32
1.11	micro/reduce.fut:sum_i32 [1000000]i32
1.11	micro/reduce.fut:prod_iota_mat4_i32 #1 (”100000i32”)
⋮	⋮
0.88	micro/scan.fut:sum_iota_f32 #0 (”10000i32”)
0.86	micro/scan.fut:sum_iota_i32 #0 (”10000i32”)
0.85	micro/scan.fut:sum_iota_i32 #1 (”100000i32”)
0.84	micro/scan.fut:sum_iota_f32 #1 (”100000i32”)
0.83	micro/reduce-segmented.fut:sum_iota_i32 100000i32 1000i32
0.72	misc/ocean-sim/tke.fut [200][200][100]f32
0.67	misc/poseidon-bench.fut:arity11 [17600000]u64

speedup	benchmark & data set – A100
1.33	misc/poseidon-bench.fut:arity11 [17600000]u64
1.26	micro/reduce-segmented.fut:sum_iota_i32 1i32 100000000i32
1.20	finpar/OptionPricing.fut OptionPricing-data/medium.in
1.18	micro/reduce.fut:sum_i8 [100000]i32
1.17	micro/scan.fut:prod_mat4_i8 [1000000]i32
1.16	misc/ocean-sim/tridiag-test.fut:tridagNestedConst data/tridiag32-small.in
1.14	micro/scan.fut:lss_iota_f32 #0 (”10000i32”)
⋮	⋮
0.87	micro/reduce.fut:lss_iota_f64 #0 (”10000i32”)
0.86	parboil/sgemm.fut data/small.in
0.86	micro/reduce.fut:sum_iota_f64 #1 (”100000i32”)
0.86	micro/reduce.fut:lss_iota_f32 #0 (”10000i32”)
0.86	micro/reduce-segmented.fut:sum_iota_i32 1000000i32 100i32
0.82	micro/reduce-segmented.fut:sum_iota_i32 100000i32 1000i32
0.77	misc/ocean-sim/tke.fut [200][200][100]f32

Table 3: Best and worst speedups of our implementation, GALOP, compared to `babysitKernels` on two different graphics processing units

2070 Super GPU. This can likely be explained by some hardware differences between the two GPUs, but we can't confidently say which specific differences are responsible for this.

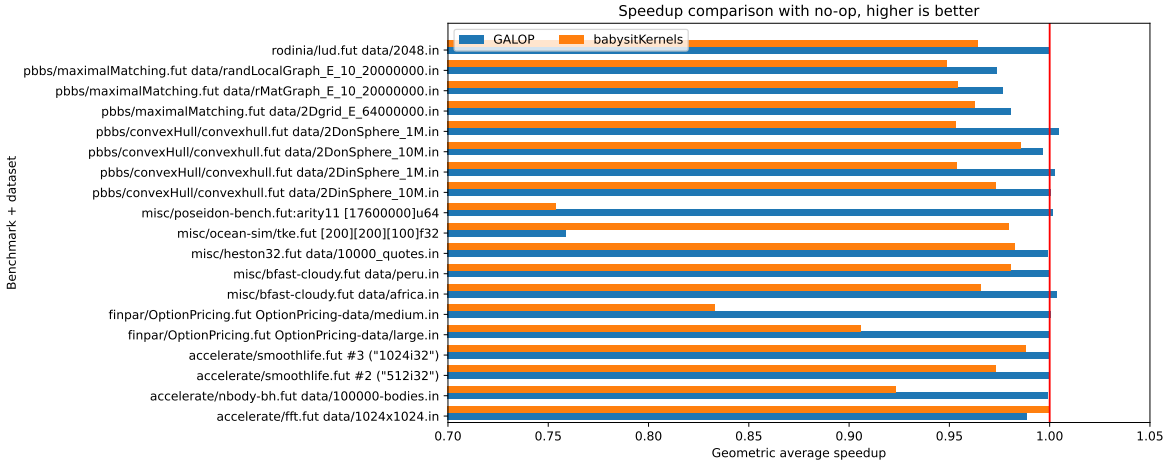
- `rodinia/lud.fut`
  Here, we observe slightly worse performance for `babysitKernels` while GALOP performs on par with *no-op*. It is not entirely clear why this is the case, since both produce many different manifests. However, `babysitKernels` makes a manifest that is inside a nested loop, which may lead to a large number of unnecessary memory transactions. Importantly, GALOP, does not decide to manifest this array access because the layout stage has a check that rejects manifests on arrays that are allocated inside a loop or kernel. Additionally, `babysitKernels` makes a manifest with the layout (0,1,2) which corresponds to row-major order. Assuming the array is already in row-major order, this manifest would only make an unnecessary copy of the array.

- For both `finpar/OptionPricing.fut` and `misc/bfast-cloudy` `babysitKernels` has slightly worse performance than *no-op*, while GALOP performs on par with *no-op*. This can likely be explained by the fact that `babysitKernels` makes manifest with the layout (2,1,0) which does not correspond to a transpose. Recall that GALOP only produces manifests that correspond to a transpose. For example, in `misc/bfast-cloudy`, GALOP manifests the same array as `babysitKernels` but with the layout (1,2,0) which *does* correspond to a transpose. Recall from Premise 3 that this is a highly important consideration, that our algorithm, GALOP, takes into account.

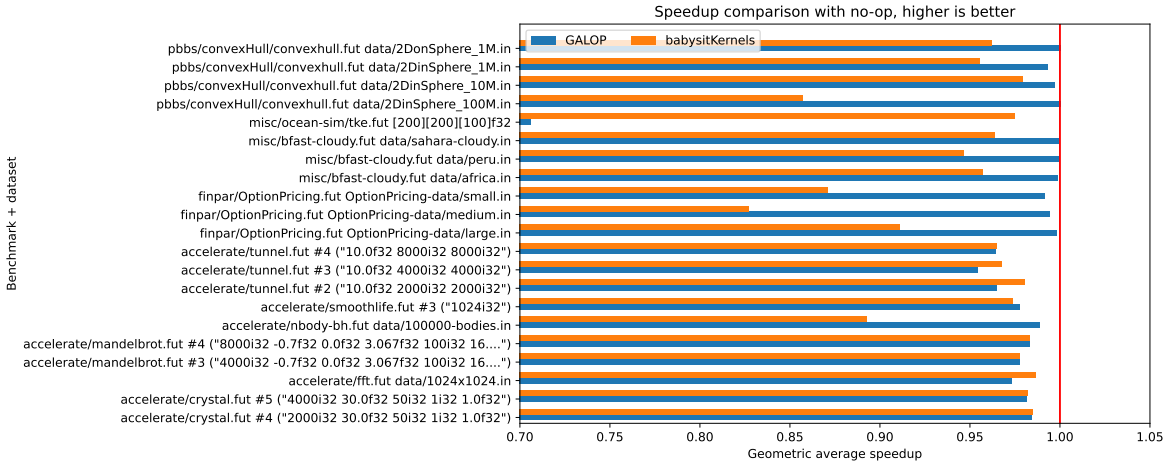
- `accelerate/nbody-bh.fut`
  Here, we observe that GALOP produces no manifests, while `babysitKernels` produces a manifest that leads to worse performance. The array access in question is of the form $A[i, j]$, where i is a kernel variable and j is a function of a single loop counter. Interestingly, this manifest would also be produced by GALOP, but is not. This is because j is computed with a sign-extension function, which our implementation of the layout stage does not yet handle.

- For `convexhull.fut` and `accelerate/smoothlife.fut` `babysitKernels` produces manifests that GALOP in principle would also make. However, these manifests are inside of loops, which leads to a potentially large quantity of memory transactions. GALOP does not produce any manifests when the array in question is allocated inside a loop or kernel to which the slightly better performance can be attributed.

- `maximalMatching.fut`
  Here, we observe that both GALOP and `babysitKernels` perform worse than *no-op*, yet GALOP performs slightly better than `babysitKernels`. This is related to the same issue uncovered by a test case, which we described earlier, where GALOP decides to manifest a loop accumulator. However, `babysitKernels` performs the same bad manifest and even performs more manifests than GALOP, which explains why its performance is even worse.



(a) A100



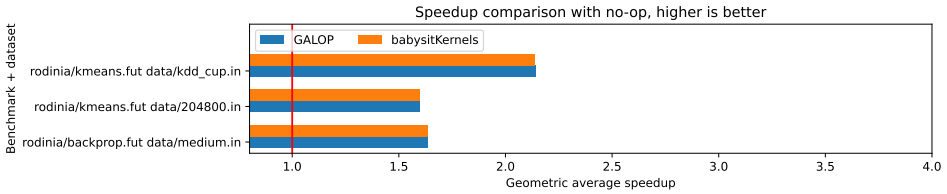
(b) RTX 2070 Super

Figure 10: Performance of our implementation, and `babysitKernels`, compared with using no locality optimizations (*no-op*), on different GPUs. These speedups were too big to be included in Figure 11a and Figure 11.

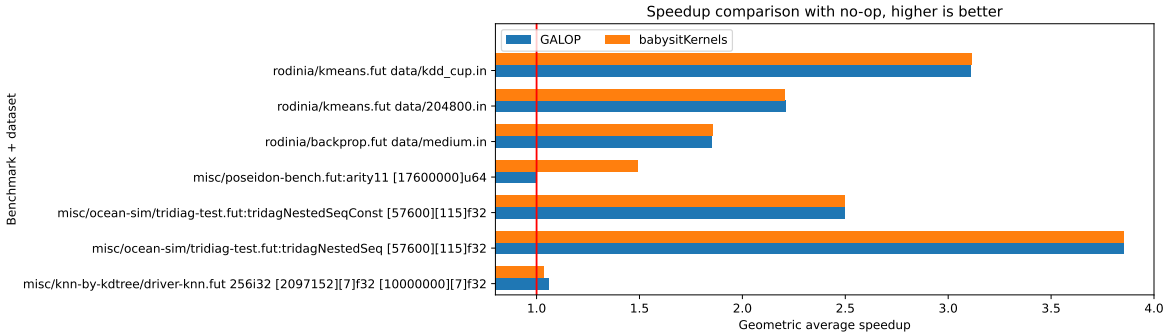
- `ocean-sim/tke.fut`

  Here, we observe that GALOP performs significantly worse than `babysitKernels` which only performs slightly worse than *no-op*. Here, GALOP for some reason incorrectly decides to manifest an array that is accessed inside a nest of two `SegMap` kernels in a manner that already has coalesced access. Specifically, its dimensions are accessed by three kernel variables in the order that we would expect to result in coalesced access. At closer inspection, we uncovered a bug in our analysis stage, which wrongly assigns the level of the innermost kernel variable to 0, which causes the layout stage to reorder the dimensions, when it should not. We leave this fix as future work.

Multicore CPU. To evaluate the performance of our algorithm on MC, we provide Figure 12, which shows the performance of our implementation, GALOP, compared to *no-op*. This figure does not show any results for `babysitKernels`, as it is not implemented for MC. Unfortunately our implementation is *not* very mature for the multi-core backend, but the results indicate that there are cases where speedups indeed can be achieved. In most cases



(a) Performance of our implementation and `babysitKernels`, compared with *no-op* on an A100 GPU.



(b) RTX 2070 Super

Figure 11: Performance of our implementation and `babysitKernels`, compared to *no-op* on an RTX 2070 Super GPU.

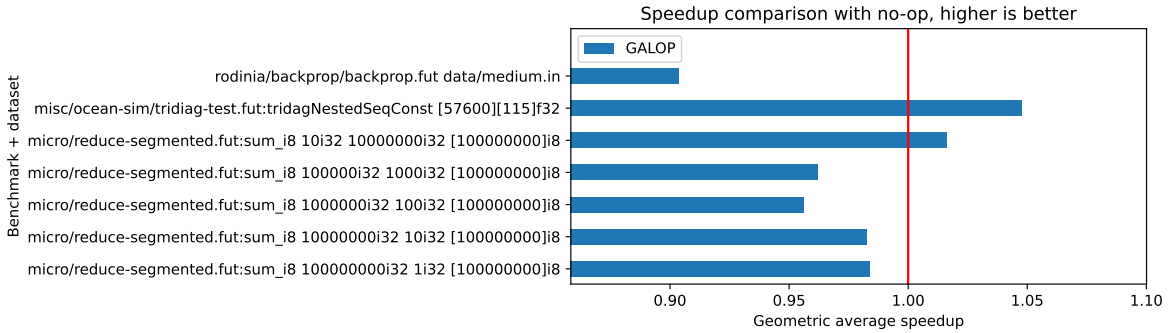


Figure 12: Performance of our algorithm on the multi-core backend on "16c" compared to *no-op*.

however, it performs slightly worse than *no-op*.

Note that we don't evaluate the performance of our implementation itself (i.e., the compiler-pass) since it and the existing pass, `babysitKernels`, already take very little time to execute compared to the rest of the compiler passes.

7 Discussion

7.1 Results

In most of the cases where our algorithm, GALOP, produces a result that is different from that of `babysitKernels`, we observe better performance than `babysitKernels` on GPU. However, in these cases, the performance of `babysitKernels` is *worse* than doing nothing at all (*no-op*), while the performance of our algorithm is on par with *no-op*. In a few cases, both GALOP and `babysitKernels` produce a result that leads to significant performance improvements, and in these cases, the performance of GALOP is identical to that of `babysitKernels`. Lastly, on the `poseidon` benchmarks, `babysitKernels` produces a result that leads to a significant performance improvement over GALOP and `no-op` on the RTX 2070 Super, while leading to a significant performance *degradation* compared to GALOP and *no-op* on the A100.

However, we observe that it often seems to be better to do nothing at all than to make use of either GALOP or `babysitKernels`. This effect is especially pronounced on the A100 compared to the older architecture of the RTX 2070 Super. This indicates that perhaps, locality optimizations are not as critical on professional server-grade GPU architectures as on consumer-grade ones. This can *possibly* be attributed to the different architectures of the A100 and RTX 2070 Super. The A100 is based on the VOLTA architecture which is a professional server-grade GPGPU that is optimized for high-performance compute. Even though the RTX 2070 Super is newer, it is based on the Turing consumer-grade architecture, which is designed primarily for fast rendering of graphics in video games and video production. The VOLTA architecture has received many improvements to memory bandwidth, streaming multiprocessors, and thread scheduling [11], compared to the Turing architecture. We speculate that these factors could explain the differences we observe, but the most likely reason is the fact that the A100 has 5 times as much memory as the RTX 2070 Super has.

However, the differences we observe could also be because locality optimizations are perhaps not as important on modern GPU architectures as they were on the older GPU architectures that were available when the `babysitKernels` pass was first implemented. A third, more likely possibility is that the capabilities of modern GPU architectures have simply outgrown the size of the datasets of Futhark's benchmarks. That is, the datasets used in the benchmarks need to be even bigger to see the benefits of locality optimizations.

7.2 Future Work

The algorithm we have presented in this thesis is a simple approach to implement, but it has a number of limitations that could be improved upon in the future. We list these areas of improvement here.

Avoid Manifesting Small Arrays

One limitation of our algorithm is that it has no way to reason about the size of dimensions and may, therefore, decide to manifest arrays where the dimensions are small. This is problematic since locality optimizations only make sense when the dimensions are large. The cost of manifesting small arrays is likely to outweigh the benefits of the improved locality of reference. Currently, our implementation relies on the heuristic described in Premise 6. One

could improve the algorithm by informing the layout stage with the size of the dimensions of arrays. We leave this as future work. However, our algorithm will at most make one unecessary manifest for each array access, meaning it will at most increase the memory usage by a factor of `2x` in the worst case.

Better Support For Slices

Our implementation relies on Premise 7 to handle slices. However, this premise is not always true, hence there can be cases where our implementation incorrectly assumes that the result of a slice is accessed sequentially by a loop counter, when in reality it is accessed in parallel by a kernel counter. This would incorrectly inform the layout stage and possibly cause the algorithm to insert a manifest that degrades performance. We leave further investigation of this as future work.

Refactor the Analysis Pass Code

As mentioned in Section 5.2, the implementation of the analysis stage uses a somewhat different style of traversal of IR than the PrimExp analysis and transform stages of the algorithm. Our implementation of the analysis pass is more complex and harder to maintain than the other stages that traverse the IR using Futhark's `Walker` class. Additionally, the implementation of this stage could be simplified even further by separating the computation of the context into its own stage (or function) that precedes the rest of the analysis, like how the PrimExp analysis stage computes information that is needed for the layout stage.

Extend the Handling of PrimExps

As mentioned in Section 5.2, the conversion of scalar expressions to `PrimExp` is not complete and could be extended to handle more types. This would make our implementation more robust and allow us to handle more cases, instead of giving up on cases where the scalar expressions can't be converted to `PrimExp`. For example, as mentioned earlier, we discovered a test case that failed because the PrimpExp analysis stage could not handle `min` and `max` expressions. Extending the number of patterns that we can reason about would help the algorithm insert more manifests and enable it to find more cases where manifests could help improve the performance. However, as noted for the `accelerate/nbody-bh.fut` benchmark, improving the handling of PrimExps would, in this case, lead to worse performance by enabling our algorithm to perform a suboptimal manifest. This indicates that some of the premises on which our algorithm is based may not be good.

Consider Existing Layouts

Currently, our algorithm is based on Premise 2, which assumes that all arrays are stored in row-major order. Since this may not always be the case, we could improve our algorithm by considering the existing layout of arrays when deciding whether to insert a manifest or not. Our implementation already has the necessary structure in place to support this with the `[Int]` in the `ArrayName` type, which at the moment always contains a row-major layout. In the future, our implementation could be extended to obtain the actual array layout from the program IR and use this to inform the layout stage.

Add Support For writes

As mentioned in Premise 1, our algorithm only handles array accesses that *read* from memory. However, good locality of reference can also be important for *writes* to memory. Hence, our algorithm could be extended to also analyze writes to memory. This would only require expanding the *analysis stage*. The result of the analysis stage would not need to change. Hence, the remaining stages of the algorithm would not need to be changed either.

Proper Tracking of Dependencies of Let-Bindings With Multiple Names

Currently, our implementation does not properly track dependencies of let-bindings with multiple names. This becomes especially noticeable for programs with a lot *tuples*. More concretely, consider the following contrived example, in Futhark. Note that this simple example would never occur, since the compiler would unravel the tuple into two separate let-bindings.

```
let (x,y) = (w * z, t)
```

Here, the analysis stage of our implementation would produce an index table where both `x` and `y` depend on `w`, `z`, and `t`, even though their dependencies should not be shared. This situation may arise, for example, when the accumulator in a loop is a tuple, or when a kernel or condition returns a tuple.

This issue is one of the key differences between the current `babysitKernels` implementation and GALOP. We leave proper handling of let-bindings with multiple names as future work.

Bugfixes

As mentioned earlier, we discovered a test where our algorithm decides to manifest an accumulator variable of a loop. This is undesired behavior. We speculate that this is because the analysis stage has a mistake that causes the accumulator to not have the loop, in which it is bound, be present in `parents` in the `Context`, since otherwise, the layout stage would prevent the manifest. This is because the layout stage ignores arrays that are bound in a loop or kernel. We leave this fix as future work.

Secondly, as mentioned earlier, we discovered a bug in the analysis stage that causes it to assign an incorrect level to a kernel variable, which significantly degrades the performance on `ocean-sim/tke.fut`. We also leave this fix as future work.

Expand Suite of Unit Tests

While the suite of unit tests we provide cover a wide range of cases, there might be some cases that we have not considered. Especially, the coverage of tests for the multicore backend is lacking.

Implement for More Backends

Our implementation currently only supports the multicore backend and GPU backends. However, as Futhark supports more backends, like ISPC, our implementation could be extended to support these backends as well. We have already put functions in place that should make it easy to extend our implementation to support more backends. Currently,

these unimplemented functions simply return error messages indicating that the backend is yet supported.

Lastly, while the MC backend is supported by our implementation, our primary focus in terms of optimal locality of reference has been with respect to GPU hardware. Hence, the MC backend likely requires more thought put into it to improve the results of our algorithm on this backend.

Incorporate Black-Box Tests Into Existing Test Suite

As previously mentioned, our suite of black box tests makes use of custom test scripts. These tests were never meant to be included in Futhark, but they turned out to be useful for testing our implementation. We leave as future work the task of incorporating these tests into the existing suite of tests which can be run with `futhark test --backend=cuda tests`.

8 Conclusion

This thesis introduces GALOP, a novel, hardware-agnostic algorithm designed to enhance the locality of reference of array accesses through strategic permutation on the order of array dimensions. Our algorithm consists of several highly modular stages that make it easy to extend to handle more backends and compute architectures. We have implemented our algorithm in the Futhark compiler for the GPU backends and the multi-core backend, the latter of which was not supported by the existing solution. By applying the algorithm to a suite of programs and comparing their performance, we have shown that GALOP generally performs slightly better than the existing solution on GPU while also matching the performance in the few cases where the existing implementation gains a performance boost. In a few instances, GALOP, as well as the existing implementation, performs worse compared to performing no locality optimizations. Still, these cases can be attributed to missing edge cases in our algorithm. We can therefore conclude that there is still a use for locality optimizations, yet modern GPU architectures require larger datasets to see the benefits of locality optimizations.

□

References

[1] *Futhark Programming Language website*. URL: https://futhark-lang.org/ (visited on 11/27/2023).

[2] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd. Pearson, 2016, p. 658.

[3] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd. Pearson, 2016, p. 667.

[4] *Using Shared Memory in CUDA C/C++*. URL: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/ (visited on 12/13/2023).

[5] Cosmin E. Oancea. "Lecture Notes for the Software Track of the PMPH Course". Programming Massively Parallel Hardware. 2018.

[6] *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. URL: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/ (visited on 11/27/2023).

[7] *An Efficient Matrix Transpose in CUDA C/C++*. URL: https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/ (visited on 12/14/2023).

[8] S.D. Kaushik, C.-H. Huang, J.R. Johnson, et al. "Efficient transposition algorithms for large matrices". In: *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. 1993, pp. 656–665. DOI: 10.1145/169627.169814.

[9] *NVIDIA Ampere Architecture In-Depth*. URL: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/ (visited on 08/31/2020).

[10] *NVIDIA A100 Tensor Core GPU*. URL: https://web.archive.org/web/20200831065916/https://www.nvidia.com/en-us/data-center/a100/ (visited on 08/31/2020).

[11] *NVIDIA Tesla V100 GPU architecture*. URL: http://images.nvidia.cn/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (visited on 12/17/2023).