Memory Optimizations in an Array Language

Philip Munksgaard
University of Copenhagen
Copenhagen, Denmark
philip@munksgaard.me

Troels Henriksen
University of Copenhagen
Copenhagen, Denmark
athas@sigkill.dk

Ponnuswamy Sadayappan
University of Utah
Salt Lake City, USA
saday@cs.utah.edu

Cosmin Oancea
University of Copenhagen
Copenhagen, Denmark
cosmin.oancea@diku.dk

***Abstract*—We present a technique for introducing and optimizing the use of memory in a functional array language, aimed at GPU execution, that supports correct-by-construction parallelism. Using *linear memory access descriptors* as building blocks, we define a notion of memory in the compiler IR that enables cost-free change-of-layout transformations (e.g., slicing, transposition), whose results can even be carried across control flow such as ifs/loops without manifestation in memory. The memory notion allows a graceful transition to an unsafe IR that is automatically optimized (1) to mix reads and writes to the same array inside a parallel construct, and (2) to map semantically different arrays to the same memory block. The result is code similar to what imperative users would write. Our evaluation shows that our optimizations have significant impact ($1.1\times$–$2\times$) and result in performance competitive to hand-written code from challenging benchmarks, such as Rodinia's NW, LUD, Hotspot.**

***Index Terms*—GPU, parallelism, functional programming, optimizing compiler**

I. Introduction

Imperative languages allow and even encourage the user to equate memory blocks with arrays and to perform memory-related optimizations aimed at (1) optimizing locality and copying overheads by reading and writing to the same array inside parallel loops, and (2) decreasing memory footprint by placing semantically different arrays in the same memory blocks. Although this allows the user to write low-level code that utilizes the memory system efficiently, the resulting code may be less maintainable due to the use of complex indexing such as flattened indices, and might hinder compiler optimizations.

For example, many of the challenges pertaining to automatic parallelization are the result of the compiler having to reverse-engineer the users' memory optimizations, as seen in the rich amount of work in the context of SUIF [1], [2], Polaris [3]–[5], and polyhedral compilation [6]–[8]. Such work relies on sophisticated analyses which are conservative in nature, i.e., parallel loops might not be recognized as such.

In functional languages, all available parallelism is explicitly declared by means of constructs such as map, reduce, or scan, that take arrays and possibly functions as arguments, and which always produce new arrays. Parallelism is "correct by construction", because data races cannot appear when the read and write accesses are performed on different arrays.

But consider a program that adds to each diagonal element of an $n{\times}n$ matrix A the corresponding element of the first row. It can be safely implemented with one parallel loop, but its

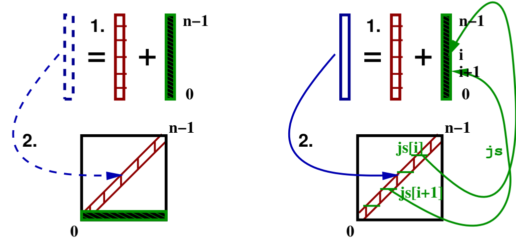


Fig. 1. **Left:** adding to each diagonal element its corresponding element on the first row. **Right:** adding to each diagonal element at position i, the diagonal element at position $js[i]$.

functional expression requires two distinct parallel operations to allow simple verification of race-free parallelism:

1. $X = \texttt{map2}\ (\lambda d\ r \rightarrow d + r)\ A[0:n:n+1]\ A[0:n:1]$
2. $A[0:n:n+1] = X$

We use A as a linearized (1D) array of length $n \cdot n$ and use triplet notation for slicing[1]. The left side of fig. 1 depicts the semantics of the two parallel operations:

1. `map2` computes a new array X of length n that temporarily stores the values of the new diagonal elements,[2]
2. the structured update operation performs an in-place update of the diagonal slice of A with the elements of X[3].

In contrast, a program that adds to each diagonal element at position (i, i) the diagonal element at position $(js[i], js[i])$—where js is an array of statically unknown numbers—cannot be implemented with one parallel loop because it might exhibit cross-iteration anti dependencies (WAR). The parallel implementation demands a separation between the reads from A and the writes to A, as demonstrated in the code below whose semantics is depicted in the right side of fig. 1:

1. $X = \texttt{map2}\ (\lambda d\ j \rightarrow d + A[j \cdot n + j])\ A[0:n:n+1]\ js$
2. $A[0:n:n+1] = X$

[1] $A[0:n:n+1]$ produces n diagonal elements by starting at index 0 and advancing with stride $n+1$.

[2] In the code, d and r, the formal parameters of the lambda function, iterate over the corresponding values of the two arrays input of `map2`: the diagonal slice $A[0:n:n+1]$ and the first row $A[0:n:1]$, respectively.

[3] Semantically, this is equivalent to a copy of A with some elements changed. Section II-C discusses how this can be safely done in-place in a purely functional language.

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

Relative to imperative programming models, a challenge with the functional approach is (1) sub-optimal memory footprint—because each parallel construct creates a new array—and (2) various copying overheads, e.g., introduced by syntax-directed translation, or the ones necessary to provide race-free guarantees (the update).

This paper discusses the optimization of a functional array language that encourages the use of array slicing, race-free parallelism, and is aimed at GPU execution. We propose compiler analyses that introduce a notion of memory in the compiler IR and aggressively optimize its use in order to produce low-level code with performance comparable to one implemented in an imperative language.

Central to our approach is the *linear memory access descriptor*, LMAD [9], [10], which offers a structured representation of a linearized set of memory references. We use LMADs in several related directions:

First, we use LMADs to extend the source language and IR with a generalized form of slicing, which can express blocked matrices or the blocked diagonal of a matrix. This not only allows a shorter and nicer notation, but also hints to the compiler that such read/write accesses may be worth analyzing since they have structure.

Second, we use LMADs in the IR for mapping array indices to concrete memory locations. This can represent chains of index space transformations that change the logical layout of an array,[4] but not its elements, in a manner that (1) is still amenable to further analysis, and (2) incurs $O(1)$ overhead,[5] even when the resulting arrays are carried across control flow such as branches. Our notion of memory has no *semantic* meaning, but only an *operational* one. We preserve the property that if the memory annotations are deleted, the program remains semantically unchanged, in the sense that the program could still be interpreted using purely functional semantics. The memory information can be seen as an "add-on" to the IR that provides a convenient framework in which a compiler can express memory optimizations.

Third, and most important, we use LMADs to aggregate and analyze sets of memory references in an optimization that aims to eliminate the copying overhead that is often introduced by:

- correct-by-construction parallel programming, and
- syntax-directed translation of high-level code.

Our *bottom-up analysis* detects candidate arrays for optimization at the point of update expressions such as $A[0:n:n+1] = X$, where X is lastly used. The analysis attempts to determine whether it is legal to allocate and compute X directly in the memory block of A—which we refer to as *short-circuiting*. This requires us to re-map the memory of all aliases of X, and to prove that the construction of X does not interfere with any uses of A. The legality of short-circuiting is determined after analyzing the first use of X (the `map2` expression). If successful, the update is turned into a no-op, eliminating the copying overhead. As an example, the left part of fig. 1 can be succesfully handled, but the right part cannot.

Finally, we report a full implementation of the proposed enhancements in a copy of the publicly available Futhark compiler [11], [12], and present an experimental evaluation on six public benchmarks that demonstrates that the short-circuiting optimization (1) has significant impact (between $1.1-2\times$), and (2) successfully optimizes challenging applications, such as NW and LUD from the publically available GPU benchmark suite Rodinia [13], to the extent that it outperforms their hand-written GPU code by a factor between $1.1 - 1.5\times$[6].

[4]Examples of such operations are slicing, transposition, or reshaping.

[5]The resulting arrays do not need to be manifested again in memory.

[6]The Rodinia benchmark suite covers computational kernels from various application domains such as data mining, bioinformatics, physics simulations, image processing and graph algorithms, which expose diverse computational patterns.

II. Preliminaries

Before diving into our contributions, we introduce the concepts of index functions and LMADs, including how LMADs can be used as index functions, as well as a brief description of the language used for our examples.

A. Index Functions for Arrays

Semantically, arrays can be seen as functions whose domains are isomorphic to contiguous subsets of integers. For example, an $n \times m$ matrix is equivalent to a function $\mathbb{N}^2 \rightarrow \mathbb{R}$. Some functional languages build on this idea and represent arrays simply as functions from an *index space* to a *value space*, which is sometimes called *pull arrays* [14] or *views* [15]. One attractive property of this approach is that certain optimizations, such as parallel loop fusion, become merely instances of function composition. However, this does not address the storage of arrays in memory.

Our work is inspired by this idea, but we use functions solely to describe the layout of in-memory arrays. We associate each array A with a *memory block* A_{mem} that can be thought of as a pointer to the start of an allocation, and an *index function* $ixfn_A$, which is a mapping from indexes to a *flat offset* into the corresponding memory block. For example, an $n \times m$ matrix has an index function of type $\mathbb{N}^2 \rightarrow \mathbb{N}$. To access an element $A[i,j]$, we would execute $A_{mem}[ixfn_A(i,j)]$. Index functions thus describe how arrays are laid out in memory. By manipulating memory blocks and index functions of arrays, we can express both footprint and locality optimizations inside a compiler. To represent index functions, we use one or more *linear memory access descriptors*.

B. Linear Memory Access Descriptor (LMAD)

An LMAD [10] defines a set of linearized uni-dimensional points that have a regular, quasi-affine structure:

$$t + \{\overline{(n:s)}^q\} \equiv \{\ t + i_1 \cdot s_1 + \ldots + \ i_q \cdot s_q \mid 0 \leq i_k < n_k,\ k = 1 \ldots q\ \} \quad (1)$$

A q-dimensional LMAD consists of an offset t and a sequence of q tuples $(n_i : s_i)$ that represent for each dimension i:

n_i: its number of points, referred to as the *cardinality*, and

s_i: the linearized distance between two consecutive points on that dimension, referred to as the *stride*.

Complex inter-procedural analyses [3], [5], [16] have used LMADs as building blocks for summarising memory references across large loop nests, for example in analyses aimed at proving parallelism based on set equations written in terms of read-only, write-first and read-write sets [9].

The power of LMADs resides in the fact that they represent a set of flat indices but "form" dimensions according to how the underlying memory is being used, rather than to the shape of the declared array.[7] The example below demonstrates how the flat (non-affine) write access to A is aggregated across the two nested loops of indices `i` and `j`.

```
-- assuming strictly positive M, N, k
do i = 0 ... m-1           -- W = ∪_{i=0}^{m-1} W_i = t + {(m : m), (n : k)}
  do j = 0 ... n-1         -- W_i = ∪_{j=0}^{n-1} W_{i,j} = t + i * m + {(n : k)}
    A[t + i*m + j*k] = ... -- W_{i,j} = t + i * m + j * k  +  {}
```

Initially, inside the two loops, the LMAD $W_{i,j}$ is the point $\{t + i * m + j * k\}$. Aggregating the write accesses across the inner loop of index $j = 0 \dots n - 1$ results in summary $W_i = \cup_{j=0}^{n-1} W_{i,j}$, which is obtained by promoting the term $j * k$ of the offset of $W_{i,j}$ to a new LMAD dimension of

- *cardinality* equal to the count of the inner loop $n_1 = n$,
- *stride* equal to the the difference between the offsets of $W_{i,j}$ for two consecutive points on the new dimension: $s_1 = t + i * m + (j + 1) * k - (t + i * m + j * k) = k$.

It follows that $W_i = t+i*m+\{(n : k)\}$, and the aggregation across the inner loop of index j is deemed successful because j is not used inside W_i.[8] Similarly, W_i is successfully aggregated across the outer loop of index $i = 0 \dots m-1$, resulting in LMAD $W = \cup_{i=0}^{m-1} W_i = t + \{(m : m), (n : k)\}$.

For example, if $t = 1$, $k = 2$, and $m \geq 2 \cdot n$ then a possible interpretation is that A is a $m \times m$ matrix, and the write accesses correspond to the slice of A that contains the first n odd indices from each row of A.

C. Language

To discuss our ideas, we use an informally specified functional language, equivalent to a subset of Futhark's core IR. This is a standard functional language where parallelism is primarily expressed with `map`, generalised to `map`n for simultaneously mapping over n arrays. Application is by juxtaposition, so we write $f\ x$ instead of $f(x)$. The language supports both creation of *fresh* arrays, i.e. arrays that do not alias any other array, using `map`, `copy`, `iota`, `scratch`, and `concat`[9], as well as "free" index-space transformations such as `reshape`, `transpose`, and slicing. The statement `let` $(x : \tau) = e$ binds variable x of type τ to the result of e. When x is an array, τ also contains information about the memory block and index function. We will often elide or shorten τ for brevity. A `let`-statement can bind multiple variables, and these are called the *pattern*. Loop expressions denote a local tail-recursive function, where `loop` $p = x$ `for` $y = 0$ .. $z - 1$ `do` b initially binds p to x, then evaluates b (the body) z times, each time binding p to the result, and ultimately returns the final value of p. For example, $n!$ can be expressed as `loop` $acc = 1$ `for` $x = 0$ .. $n - 1$ `do` $acc * (1 + x)$.

[7]For example, LMADs allow analyses to be extended across procedure boundaries where arrays are permitted to change shape (e.g., in Fortran77).

[8]If j appears inside the cardinal of one of the original-LMAD dimensions, then an overestimate could still be computed by substituting j with whichever bound maximizes the cardinal, i.e., either its lower (0) or upper bound ($N-1$).

[9]`iota` $n = [0, \dots, n-1]$ and `scratch` n `f32` creates a new array of length n of single-precision floats (`f32`), whose elements are uninitialized.

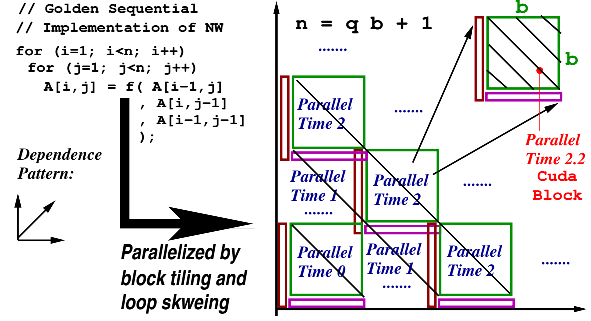


Fig. 2. NW parallel access patterns

In contrast to most functional languages, we support array *updates*, written as A `with` $[i] = e$. A uniqueness type system based on affine logic ensures that the "old" value of A is not used on any subsequent execution path, enabling the update to be implemented as an in-place write to the memory of A [17]. In cases where we reuse the name of the argument for the result, we use the form `let` $A[i] = e$ as syntatic sugar, but semantically this is merely variable shadowing, not a true imperative effect.

III. Bird's Eye View by Example

This section demonstrates the intuition behind the techniques presented in this paper, namely the use of LMADs for:

- providing a stronger abstraction than triplet notation for array slicing in the source language (section III-B),
- introducing a notion of memory in the compiler IR for a memory-agnostic source program (section III-C),
- driving the index analysis that enables memory optimizations, e.g., that recovers the efficient in-place update of parallel containers in a manner that is determined safe by compiler analysis but not verifiably correct by type checking (section III-D).

Section III-A explains the running example, based on Rodinia's parallel implementation of the Needleman-Wunsch (NW) method for DNA sequence alignment.

A. Running Example

The left side of fig. 2 shows the sequential implementation of NW and its dependency pattern. Rodinia's parallel code can be obtained by a combination of loop skewing and block tiling, whose access patterns are depicted on the right:

- the green $b \times b$ blocks forming an anti-diagonal are computed in parallel (i.e., the write set), but successive anti-diagonals are computed sequentially;

- each $b \times b$ green block is computed by one CUDA block of threads that takes as input the horizontal and vertical red bars adjacent to the block (the read set), and similarly computes the internal anti-diagonals in parallel.

The resulting pseudo-Futhark code is shown below, where n is the number of elements in a row, b is the block size, and q is the number of blocks in a row, while *input* is a flat array of size $n \times n$ and `process_block` is a function computing the value of one block given its two input perimeters.

```
loop A for i < q do
  let R_vert_slc = A[i*b + {(i+1 : n*b - b), (b+1 : n)}]
  let R_horiz_slc= A[i*b + 1 + {(i+1: n*b - b),(b : 1)}]

  let X = map2 process_block R_vert_slc R_horiz_slc
  let A[i*b + n + 1 +
        {(i+1: n*b - b), (b : n), (b : 1)}] = X
  in A
```

The loop shown computes the first half of the matrix, and is followed by another loop for the second half, which, apart from the values used for indexing, is identical. We use some unconventional slicing, which we'll explain next.

B. LMADs as Generalized Slicing at Language Level

Functional approaches typically require all parallelism to be verifiably deterministic by simple type-checking techniques. In practice, this means that reads and updates to a parallel container are split into two parallel operations: one that produces a new array, and one that performs the update:

$$\texttt{let } X = \texttt{map2 } f \ A[R^{vert}] \ A[R^{horiz}]$$
$$\texttt{let } A[W] = X$$

For NW we would like to express the block-level parallelism elegantly as above, where (1) W, R^{vert}, R^{horiz} are notations for generalized slicing, and (2) `map2` computes in parallel each of the green blocks on the current anti-diagonal by applying f—the function that computes a block—to its corresponding adjacent vertical and horizontal bars. In order to achieve this, we need to be able to express a slice $A[R^{vert}]$ that produces all the vertical bars on an anti-diagonal, and similarly for $A[R^{horiz}]$. The update would similarly need to express that the resulting array X updates the slice of A corresponding to the green-blocks on the anti-diagonal, $A[W]$.

Assuming A is a 1-dimensional $n{\cdot}n$ array, with $n = q{\cdot}b{+}1$ for some $q \geq 1$, and i is the index of the current anti-diagonal, we can express[10] these slices as LMADs, defined in eq. (1):

- $W \ = \ i{\cdot}b{+}n{+}1 + \{(i{+}1 : n{\cdot}b{-}b), \ (b : n), \ (b : 1)\}$[11]
- $R^{vert} \ = \ i \cdot b \ + \{(i+1 : n \cdot b - b), \ (b+1 : n)\}$
- $R^{horiz} = \ i \cdot b \ + \ 1 \ + \{(i+1 : n \cdot b - b), \ (b : 1)\}$

We have implemented the following semantics in both the source and IR languages: a read slice is an $O(1)$ operation that creates an array of the same rank and dimension lengths (cardinality) as the LMAD, whose elements are taken from the source array at the indices specified by the LMAD. An update based on an LMAD slice is fully parallel and requires work proportional to the product of the cardinals of LMAD dimensions, i.e., the number of indices defined by an LMAD. In the source language, dynamic checks are inserted for slices whenever necessary to verify that all strides are non-zero, and that the LMAD dimensions do not overlap [9], meaning that the update is guaranteed to not introduce output dependencies.

While it is possible to express NW without LMAD slicing, the code would use (1) complex, error-prone indexing, and (2) a `scatter` update operation that is typically applied to a very irregular set of indices, thus hinting that the code is likely not statically optimizable. In contrast, an LMAD update guarantees structured indexing, and hints at optimization opportunities.

C. LMADs as Index Function for Memory Abstraction

At the language level, operations such as slicing produce a new array, which can be bound to a variable and freely used in the remaining program, e.g., `let` $B = A[R^{vert}]$ semantically creates a $(i+1) \times (b+1)$ matrix B. Such change-of-layout operations[12] are supported in $O(1)$ time: their elements are the same as (a subset of) those of the input array, but re-arranged according to a regular structure.

Index functions, represented as LMADs, are the glue that allows the mapping of such arrays to memory. For example, assuming that (*fresh*) array A holds elements of type t and is stored in memory block A_{mem} in row-major form—then the element $B[j,k]$ resides at memory location:

$$R^{vert}(j,k) = A_{mem} + (i \cdot b + j \cdot (n \cdot b - b) + k \cdot n) \cdot \texttt{sizeof}(t)$$

This is a direct application of the definition of LMAD in eq. (1), but now seen as an index function rather than an abstract set.[13] By knowing the structure of the LMAD of an array at compile time, we can emit an expression such as the above when generating code for an array access. Section IV discusses how this abstraction is introduced inside the compiler.

D. LMADs as Building Blocks for Index Analysis

Having introduced a memory notion in the compiler IR, we turn our attention to optimizing it. With our example

$$\texttt{let } X = \texttt{map2 } f \ A[R^{vert}] \ A[R^{horiz}]$$
$$\texttt{let } A[W] = X$$

this essentially corresponds to determining whether it is legal to update A directly inside the `map2` operation, which would eliminate the overhead of the following update.

Section V presents an analysis that attempts to *short-circuit* arrays A and X by constructing X directly in the memory block A_{mem} of A with the new index function W.

[10]The triplet notation cannot express such slices because it cannot create "new" dimensions, i.e., slicing is applied to each of the array dimensions.

[11]For example, the cardinal of the outer dimension of W is $i+1$, denoting the number of green blocks on the ith anti-diagonal, and its stride is $n \cdot b - b$, denoting the distance between the start of two consecutive green blocks: one needs to jump up b lines of length n, and then go back horizontally b positions. The offset $i \cdot b + n + 1$ corresponds to the start offset of the bottom block of the ith anti-diagonal: this is horizontally preceded by i blocks $(i{\cdot}b)$, and since the green blocks start at $(1,1)$, we add one line and one element $(n+1)$.

[12]Examples include reshaping, rotating, reversing, permuting (e.g., transposition) array dimensions, and slicing based on LMAD or triplet notation.

[13]One important difference is that any abstract-set LMAD can be normalized to have only positive strides, but this is not possible for the index-function that corresponds to reversing a 1D array of length n: $L^{rev} = n{-}1{+}\{(n : -1)\}$.

For NW, the challenging part is the index analysis, presented in section V-B, which, intuitively, verifies that the write set of the moved X—the green boxes on an anti-diagonal—does not overlap the read set of A—the vertical and horizontal bars on the same anti-diagonal, i.e., $W \cap (R^{vert} \cup R^{horiz}) = \emptyset$.

While fig. 2 shows that this holds, proving it is nontrivial; section V-C presents an adaptation of the algorithm in [9] that provides a sufficient-condition test for proving empty intersection of two LMADs (which succeeds for NW).[14]

If the analysis succeeds, the update will be redundant and treated as a no-op. Essentially, introducing a memory notion in IR is what allows a graceful transition to a representation that, while not safe by construction, is able to express imperative-style memory reuse.[15] For NW, proving that X can be short-circuited is equivalent to proving that the loop that processes the anti-diagonal blocks is parallel; the difference is that the failure of the latter is catastrophic (i.e., leads to sequential execution), while the failure of the former results in slowdowns of only $1.1 - 1.5\times$, i.e., paying the overhead of the update.

IV. LMAD-based Representation of Memory

A. LMADs as Index Functions

A q-dimensional array is associated with a q-dimensional LMAD. To index an array, we apply the associated LMAD to the given indexes. We define application of an LMAD $L = x + \{(d_1 : s_1), \ldots, (d_q : s_q)\}$ as:

$$L(y_1, \ldots, y_q) = x + \sum_{1 \le i \le q} y_i \cdot s_i$$

Now consider how to represent an $n \times m$ matrix. Two possible index functions are as follows:

$$L_1 = (0 + \{(n : m)(m : 1)\}) \quad L_2 = (0 + \{(n : 1)(m : n)\})$$

They both describe an index space of shape $n \times m$. However, L_1 describes a row-major order layout, while L_2 describes column-major order. These cases are particularly common, so as an abbreviation we define $\mathcal{R}(d_1, \ldots, d_q)$ and $\mathcal{C}(d_1, \ldots, d_q)$ as index functions for arrays of shape $[d_1] \cdots [d_q]$ in respectively row-major and column-major order, with zero offset.

B. Transformations of index functions

Index transformations are implemented by changing the index functions of the array. We are concerned with the following transformations: transposition, slicing, and reshaping.

Transposition, or indeed any permutation of dimensions, is done simply by permuting the components of the LMAD. To slice an LMAD we compute an augmented offset by multiplying each of the slice offsets a_i with the strides of the original LMAD. For example, to extract column i from a row-major $n \times m$ matrix with the triplet slice $[0 : n : 1, i : 1 : 0]$[16]

[14]Our test is inspired by and extends the one from [9], which neither distributes the terms of the offset, nor splits overlapping dimensions.

[15]If the memory annotations are deleted, the program is again guaranteed to express correct-by-construction parallelism, i.e., no data-races are possible.

[16]This actually produces an index function for a $n \times 1$ "matrix", which can be seen as a column vector.

```
let as = (0 .. 63)            -- ixfn_as= 0 + {(64:1)}
let bs = unflatten 8 8 as -- ixfn_bs= 0 + {(8:8), (8: 1)}
let cs = transpose bs         -- ixfn_cs= 0 + {(8:1), (8: 8)}
let ds = cs[1:3:2, 4:8:1] -- ixfn_ds= 1+4*8+{(2:2),(4:8)}
let es = (flatten ds)[2:]
--ixfn_es=L_2oL_1 with L_1=2+{(6:1)} and L_2= 33+{(2:2),(4:8)}
in es[5]
-- To find the flat offset of es[5] in the memory of as:
-- 1. Interpret index 5 by L_1: L_1(5) = 2 + 5*1 = 7
-- 2. Unrank 7 to L_2's dims: i = 7 / 4 and j = 7 mod 4
-- 3. Interpret (i,j) by L_2: L_2(1,3)= 33 + 1*2 + 3*8= 59
```

Fig. 3. Index function computations for complicated slices. Please note that none of these operations manifest new arrays in memory.

we compute a new offset $0 \cdot m + i \cdot 1 = i$, producing the LMAD $i + \{(n, m)(1, 0)\}$. LMAD-slices, like the ones used to compute NW, can be handled in a similar manner.

The most difficult operation to handle is arbitrary reshaping. While some common special cases can be handled (e.g. flattening a row-major matrix), there are cases that cannot be expressed with a single LMAD. For example, when flattening a column-major matrix to a single-dimensional array, the resulting index function cannot be expressed as a single LMAD. Therefore we allow index functions to comprise multiple LMADs. Applying these is done by applying the final LMAD to the initial index producing an offset, which is then unranked with respect to the q-dimensional index space of the remaining LMADs, giving an q-dimensional point, to which the remaining LMADs are applied. Unranking involves costly division and remainder operations at run-time, but fortunately this case rarely occurs in real programs. Figure 3 demonstrates the treatment of a non-trivial example of slicing.

C. Introducing Memory Information

The source language exposed to programmers does not expose a notion of memory. All memory information is inserted by the compiler. For statements that create fresh arrays, we insert an `alloc` statement that allocates a memory block of appropriate size, and use a row-major order index function by default. For example, the statement,

$$\texttt{let } (y : [n][m]\texttt{int}) = \texttt{copy } x$$

will be turned into

$$\texttt{let } (y_{mem} : \texttt{mem}) = \texttt{alloc } (n * m * \texttt{sizeof(int)})$$
$$\texttt{let } (y : [n][m]\texttt{int@}y_{mem} \to \mathcal{R}(n, m)) = \texttt{copy } x$$

Note how the binding of y contains both the name of the memory block and the index function describing its layout. For statements that perform index transformations of an array, we insert no `alloc` statement, but instead use a transformed index function. For example,

$$\texttt{let } (z : [m][n]\texttt{int}) = \texttt{transpose } y$$

becomes

$$\texttt{let } (z : [m][n]\texttt{int@}y_{mem} \to \mathcal{C}(m, n)) = \texttt{transpose } y$$

Note that z resides in the same memory block as y.

Handling `if` is more tricky because the values returned by the two branches can be in different memory blocks and have

```
1 let as = scratch m f32
2 let bs = scratch n f32
3
4 ... fill in arrays as and bs ...
5
6 let xss = concat as bs^lu
```

(a) Trivial Concatenation Example.

```
1 let as = scratch n f32
2 ... use of arrays aliased with xss ...
3 ... fill in array as ...
4 let bs = chg-layout-op-1 as
5 let cs = chg-layout-op-2 bs
6 ... use of arrays as, bs, cs ...
7 ... use of arrays aliased with xss ...
8 let xss[W] = bs^lu
```

(b) Challenges to Analysis.

Fig. 4. Simple example and challenges for short-circuiting array bs in xss.

different index functions. Our solution is to compute the *least general generalization* (lgg) of the involved index functions, via anti-unification [18]. Suppose that the input statement is

$$\texttt{let}\ (z : [n][m]\texttt{int}) = \texttt{if}\ c\ \texttt{then}\ x\ \texttt{else}\ y$$

and that x resides in x_{mem} with index function $\mathcal{R}(n, m) = 0 + \{(n, m)(m, 1)\}$, and y in y_{mem} with $\mathcal{C}(n, m) = 0 + \{(n, 1)(m, n)\}$. The lgg of these index functions is $0 + \{(n, a)(m, b)\}$ for some a, b. Thus, the pattern for the statement is augmented with bindings of z_{mem}, a, b, for which the branches return specific values:

$$\texttt{let}\ (z_{mem}, a, b, z : [m][n]\texttt{int}@z_{mem} \rightarrow 0 + \{(n, a)(m, b)\}) = \\ \texttt{if}\ c\ \texttt{then}\ (x_{mem}, m, 1, x)\ \texttt{else}\ (y_{mem}, 1, n, y)$$

A similar technique is used for loops. Anti-unification is not supported in some rare cases, e.g., where the index functions differ in number of constituent LMADs, or when the loss of information would prevent locality optimizations, e.g., coalesced accesses to memory. In such cases we insert copy statements to normalise the arrays to a uniform representation.

V. Array Short-circuiting Optimization

We denote by *circuit point* a statement

$$\texttt{let}\ x = \texttt{concat}\ a\ b^{lu}$$

or

$$\texttt{let}\ y[W] = b^{lu}$$

where W denotes a triplet-notation or LMAD slice and b^{lu} denotes the last use of b.

The simplest example demonstrating the proposed optimization is shown in fig. 4a and concerns creating a new array xss by concatenating the elements of two existent arrays as and bs. If bs is a fresh array and it is lastly used here, then essentially one may allocate and compute bs directly in xss_{mem}, the memory of xss—i.e., bs will have index function $m + \{(n : 1)\}$. If as is lastly-used too, then it is similarly treated and concatenation becomes a no-op, since xss_{mem} already has the expected content.[17]

Our analysis is implemented as a *bottom-up* pass that identifies candidates for short-circuiting, such as bs, at circuit points (last use of bs) and validates the optimization at the definition of their corresponding fresh arrays (first use). We use fig. 4b to demonstrate the four properties that we verify:

[17]Please note that `let` xss = `concat` bs bs^{lu} cannot be perfectly optimized: one copy of bs is still needed, and in our analysis this is reflected by the fact that only one of the two uses of bs can possibly be a last use.

(1) bs is lastly used in the circuit point (denoted b^{lu}).[18]
(2) xss_{mem} is in scope (already allocated) at the definition point of the fresh array associated to bs. This is enabled by a preceding pass that aggressively hoists out allocations.[19] In fig. 4b, the fresh array is as, created at line 1, and bs is obtained from as at line 4.
(3) the analysis optimistically assigns a new memory block and index function to bs at the circuit point (line 8). As it moves up towards the definition of its fresh array as, the analysis needs to be able to compute and assign new (valid) index functions to all variables that are in an alias relation to bs, for example as and cs. In particular, these might use some of the program variables of W, which might be defined after the creation of as, bs, cs, hence they need to be "translatable" at those creation points.
(4) since the analysis attempts to lay out bs in the memory of xss, it needs to conservatively prove that, semantically, there is no write to bs that would override a memory location that is read or written via xss—in between the definition of as and the circuit point of bs. As such, a use of array xss at line 2 would be safe, because no element of as (bs) has been written yet, but a use of xss at line 7 would need to be proven to not overlap the preceding writes to bs (as and cs).

Essentially, (1) bounds the analysis scope to the live range of as/bs, (2) and (3) are legality checks satisfying the IR requirements that any variable is memory typed at its definition point, and (4) ensures the preservation of program semantics in the case when xss is accessed during the live range of bs: Since xss and bs are not aliased, a write to bs should not overwrite a location that is later read from xss, and conversely, a write to xss should not overwrite a location previously written by (aliases of) bs.[20] Property 1 and 2 are easy to check, so we will concentrate on verifying properties 3 and 4.

We use a syntax-directed approach [19] for implementing the analysis (a.k.a., structural), in which each syntactic category is implemented by a translation rule that uses case analysis on the IR constructors[21]. We informally discuss several such translation rules, demonstrated on concrete examples.

A. Verifying the Third Safety Property

We first discuss the case of layout transformations using the example in fig. 4b, and then explain how to extend analysis across `if` and `loops` that can return memory blocks.

For simplicity, we assume that xss is a fresh array—i.e., laid out in memory xss_{mem} in row-major order—and W is an LMAD slice. Our analysis then attempts to construct array bs in xss_{mem} with the new index function W (at line 8).

[18]We have implemented a last-use analysis (as a preceding pass) that conservatively guarantees that neither bs nor any array in an alias relation with bs can possibly be used on any path following a bs^{lu}-annotated statement.

[19]Pathological cases exist in which this property cannot be satisfied, e.g., if the size of xss_{mem} is data or control dependent on the elements of bs.

[20]This also covers the illegal case when a write to xss is later read from bs because then bs has been necessarily written before the write to xss.

[21]E.g., a program is a block of statements; a statement can be simple additions or ifs or loops, which contain their own block of statements.

```
let (bs_mem , n,
     bs @ bs_mem → R(n)) =
  if cond
  then
    let bs^th_mem = alloc ...
    let bs^th @ bs^th_mem → R(t)
      = mapnest (i<t)...
    in  (bs^th_mem , t, bs^th)
  else
    let bs^el_mem = alloc ...
    let bs^el @ bs^el_mem → R(q)
      = Mapnest (i<q)...
    in  (bs^el_mem , q, bs^el)
...
let xss[i, 0:n] = bs
```

(a) If-Then-Else Example.

```
let as0_mem = alloc n f32
let as0 @ as0_mem → R(n) =
      mapnest (i<n) ...
let (bs_mem , bs @ bs_mem → R(n))=
 loop (as_mem , as @ as_mem → R(n)) =
      (as0_mem , as0)
 for i = 0 .. n-1 do
  let as'_mem = alloc ...
  let as' @ as'_mem → R(n) = f(as^lu)
  let bs'_mem = alloc ...
  let bs' @ bs'_mem → R(n) =
    mapnest (j<n)
            (as'[j]*as'[j+1])
  in  (bs'_mem , bs')
...
let xss[n:2*n] = bs
```

(b) Loop Example.

Fig. 5. Extending Analysis to Compound Statements If and Loops.

```
let yss_mem = alloc ...
let as_mem  = alloc ...
let bs_mem  = alloc ...
let cs_mem  = alloc ...
let yss@yss_mem → R(n, 2 * n)=
  mapnest (i<n, j<2*n)...
let as @ as_mem → R(n) =
  mapnest (j<n) ...
let bs @ bs_mem → R(n) =
  mapnest (j<n) ...

let cs @ cs_mem → R(2 * n) =
        concat as^lu bs^lu
let yss[i] = cs^lu
```

(a) Chaining Example.

```
let xss_mem = alloc ...
let xss @ xss_mem → R(n, n) =
 mapnest (i<n)
   let rs_mem = alloc ...
   let rs_0 @ rs_mem → R(n) =
       scratch n f32
   let rs_0[0] = as[i,0]
   let rs' @ rs_mem → R(n) =
     loop (rs@rs_mem → R(n))=(rs_0)
     for k = 1 .. n-1 do
       let rs[k] = as[i,k] +
              sqrt(rs[k-1])
       in   rs
   in rs'^lu
```

(b) `mapnest` Example.

Fig. 6. Chaining and `mapnest` Examples

a) Change-of-Layout Transformations: Transformations of bs such as the one on line 5 producing cs—or any other arrays produced by a sequence of layout transformations from bs—is always supported, because the index function of cs can be directly computed by applying the layout transformation to W, denoted $\texttt{chg-layout-2} \circ W$, in a manner similar to how the memory abstraction was introduced in section IV.

The change-of-layout transformation at line 4, however, is applied to the fresh array as and produces the short-circuited array bs. This is more difficult to support because it corresponds to the equation $W = \texttt{chg-layout-1} \circ ixfn_{as}$, where $ixfn_{as}$ is the unknown denoting the "new" (*rebased*) index function that needs to be assigned to as. This equation does not always have a valid solution: e.g., if W is a dense slice, such as $i \cdot n + \{(n : 1)\}$ and $\texttt{chg-layout-1}$ is a slice that selects every other element of as, then the $2 \cdot n$ elements of as cannot possibly fit inside the n memory elements associated with $xss[W]$. While a more general solution might exist, we currently support only the transformations that are "invertible"—such as rotating/reverting the elements of a dimension and permuting an array dimensions[22]—by equation:

$$ixfn_{as} = \texttt{chg-layout-1}^{-1} \circ W$$

b) Index-Function Translation: When the analysis reaches the definition of an array variable, it is possible that its rebased index function contains variables that are not in scope (i.e. defined later). We address this by extending the symbol table of the bottom-up analysis to record the integral variables that are defined by simple arithmetic operations. Translating the index function consists of substituting to a fixpoint the keys of the symbol tables with their arithmetic expressions; analysis fails if the translation still uses variables not in scope.

c) If and Loops: We explain the treatment of `if` statements based on fig. 5a. As before, the analysis tries to rebase bs in the memory of xss (last line). But now bs is produced by an `if` statement and resides in memory bs_{mem}. We know that bs_{mem} has been allocated, but cannot uniquely pinpoint where: it is one of bs^{th}_{mem} or bs^{el}_{mem}, which are returned from the two branches, respectively. Similarly, bs has index function $\mathcal{R}(n)$, where n is either t or q. The analysis for bs is reduced to solving two sub-problems that attempt to short-circuit in xss the results of the `then` and `else` branches *within the corresponding bodies*. For example, the index function of bs^{th} is rebased and translated ($n \mapsto t$) to $i+\{(0 : t)\}$ and, as before, the success will be determined after analyzing the `mapnest` that creates (the fresh array of) bs^{th}, and similar for bs^{el}.

We use fig. 5b to explain loops, which are treated similarly: the result bs of a loop, originally residing in existential memory bs_{mem}, can be short-circuited in xss if (1) its size is invariant through the loop, (2) the result of a loop iteration bs' is successfully short-circuited within the iteration body, (3) at a definition point that comes after the last use of the corresponding iteration input as, and (4) the loop initializer as_0, defined prior to the loop, can also be short-circuited. If analysis succeeds, arrays as_0, bs, as and bs' (but not as') will be placed in xss_{mem} with index function $n + (n : 1)$.

Essentially, (1) ensures that the index functions of as_0, as still fit in the destination array, and (2-4) subsume a conservative condition for reusing memory for as_0, as, bs', e.g., because the liveness of as and bs' do not overlap (3).[23]

d) Transitive Chaining: Figure 6a shows a source program in which fresh arrays as and bs can be short-circuited (by concatenation) into array cs, which, at its turn, can be short-circuited into array yss. Our bottom-up analysis supports such transitive cases, by attempting to first rebase cs into the memory yss_{mem} of yss, then attempting to rebase as and bs in the "new" memory of cs which is now yss_{mem}[24]. However, the success of short-circuiting as and bs is flagged to be conditional on the success of cs; if analysis of cs fails, then it is remembered, and analysis is re-run (to a fix point) to allow as and bs to be circuited in the memory of cs.

[22]For example, the inverse of the transformation that permutes the dimensions of an array by permutation p is the permutation by p^{-1}, the inverse of p. The inverse of matrix transposition is matrix transposition $(M^T)^T = M$.

[23]The case of an iterative stencil would not conform with (3) and would be unsafe to reuse memory across the input and result stencil. In contrast, a loop that adds one to each element of a loop-variant array would not conform but would be safe. Instead of complicating analysis to "guess" implicit circuiting points, we expect the user to insert them explicitly inside the loop body.

[24]Assuming yss is a fresh array of dimensions $n \times 2 \cdot n$, and denoting by $t = i \cdot 2 \cdot n$, the rebased index functions are: $ixfn_{cs} = t + \{(2 \cdot n : 1)\}$, $ixfn_{as} = t + \{(n : 1)\}$, and $ixfn_{bs} = t + n + \{(n : 1)\}$.

e) Mapnests: We explain the semantics and treatment of `mapnests` on the example in fig. 6b. A `mapnest` is essentially a perfect nest of parallel loops (in our case of depth one), whose indices and counts are written as $\overline{(i < n)}$. Its body computes a per-thread i result rs', which is always explicitly returned at the end (`in` rs'). In our example, rs_0, rs and rs' essentially correspond to the same array which is computed in place, one element at a time, inside the sequential loop.[25] These arrays are normally laid out $\mathcal{R}(n)$ in memory rs_{mem}, which is allocated inside the `mapnests`.

The semantics of the `mapnest` is that there is an implicit copy of each per-thread result rs' into the array result of the `mapnest`, which is denoted xss, i.e., $xss[i] = rs'$. Our analysis treats this as an *implicit circuit point*, and attempts to short-circuit rs'—and its aliases rs and rs_0—into memory xss_{mem} with rebased index function $i \cdot n + (n : 1)$. This has high impact on the LBM and LocVolCalib benchmarks.

```
-- Success if reached!
1. let bs = scratch ...

-- L_bs^wt = slice S_bs^wt ixfn_bs^new
-- W_bs = L_bs^wt
-- Fails if: U_xss ∩ L_bs^wt ≠ ∅
2. let bs[S_bs^wt] = ...

-- L_xss^wt = slice S_xss^wt ixfn_xss
-- U_xss = L_xss^rd ∪ L_xss^wt
3. let xss[S_xss^wt] = ...

-- L_xss^rd = slice S_xss^rd ixfn_xss
-- U_xss = L_xss^rd
4. let ... = f (xss[S_xss^rd])

-- W_bs = ∅,  U_xss = ∅
-- ixfn_bs^new = slice S^circ ixfn_xss
5. let xss[S^circ] = bs^lu
```

(a) Straight-Line Code

```
let bs_0@bs_mem : R(q̄)=scratch...
...
let bs @ bs_mem : R(q̄) =
  loop (bs_i@bs_mem : R(q̄))=(bs_0)
  for i = 0 .. n-1
    -- W_bs^i, U_xss^i are the
    -- summaries of the body
    let bs_i[S_bs_i^wt] = ...
    let xss[S_xss^wt] = ...
    ... = f(xss[S_xss^rd])

    -- W_bs^i = ∅,  U_xss^i = ∅
    in bs_i

-- W_bs, U_xss denote the
-- summaries at this point
...
-- W_bs = ∅,  U_xss = ∅
let xss[S^circ] = bs^lu
```

(b) Loop Aggregation and Safety

Fig. 7. Index Analysis for Straight-Line Code and Loops

B. LMAD-*Based Index Analysis*

The previous section has presented the gist of our analysis for the simple case in which the (memory of) array xss, which bs is short-circuited into, is not used in-between the first and last use of bs. This section explains how to relax this restriction, without which analysis would fail on important and challenging benchmarks such as NW and LUD.

The core idea is that for each short-circuit candidate, denoted by `let` $xss[S^{circ}] = bs^{lu}$, the bottom-up analysis maintains two summaries of memory locations, viewed as abstract sets and represented as unions of LMADs:

- $\mathcal{U}_{xss}$: aggregates all uses of the memory xss_{mem} of xss from the circuit point (backward) until the current statement;
- $\mathcal{W}_{bs}$: aggregates the memory locations that are semantically written via (aliases of) bs, which is now rebased in xss_{mem}.

The safety property that successful analysis must verify is that any write to (aliases of) the rebased bs does not overlap in memory with any successor uses of xss aliases (held in U_{xss}).

a) Straight-line code free of control flow: This is demonstrated in fig. 7a. At the circuit point (line 5), bs is rebased in xss_{mem}, hence its new index function, $ixfn_{bs}^{new}$, is computed by applying the slice S^{circ} to the index function of xss, and the summaries $\mathcal{U}_{xss}$ and $\mathcal{W}_{bs}$ are initialized to the empty set.

Line 4 reads the slice S_{xss}^{rd} of xss. The analysis computes the read set of memory references by (1) applying the slice S_{xss}^{rd} to $ixfn_{xss}$ and (2) re-interpreting the resulting LMAD L_{xss}^{rd} as an abstract set,[26] which is added to $\mathcal{U}_{xss}$. Line 3 writes xss, and $\mathcal{U}_{xss}$ is similarly updated to $L_{xss}^{rd} \cup L_{xss}^{wt}$.

Line 2 writes a slice S_{bs}^{wt} of bs. The corresponding set of memory locations L_{bs}^{wt} is (1) computed by applying the slice S_{bs}^{wt} to the new (rebased) index function of bs and (2) is added to $\mathcal{W}_{bs}$. At this point, analysis must verify the safety property, which is reduced to verifying the non-overlap of a finite number of LMAD pairs (which is discussed in section V-C):

$$L_{bs}^{wt} \cap \mathcal{U}_{xss} = \emptyset \quad \Longleftrightarrow \quad L_{bs}^{wt} \cap L_{xss}^{rd} = \emptyset \,\wedge\, L_{bs}^{wt} \cap L_{xss}^{wt} = \emptyset$$

b) Loops: This is demonstrated in fig. 7b, where we assume the summaries $\mathcal{W}_{bs}$ and $\mathcal{U}_{xss}$ contain the corresponding memory references just before the analysis reaches the loop statement. The idea is to apply analysis independently (recursively) on the loop body, where the summaries $\mathcal{U}_{xss}^{i}$ and $\mathcal{W}_{bs}^{i}$ of (some) iteration i are initialized to the empty set before the last statement of the body, and are fully computed after the first statement of the loop body was analyzed.

Denoting by $\mathcal{U}_{xss}^{>i}$ the (partial) union of memory references in the iterations following i, and by $\mathcal{U}_{xss}^{loop}$ and $\mathcal{W}_{bs}^{loop}$ the total union across all iterations, i.e.,

$$\mathcal{U}_{xss}^{>i} = \bigcup_{j=i+1}^{n-1} \mathcal{U}_{xss}^{j}, \quad \mathcal{U}_{xss}^{loop} = \bigcup_{i=0}^{n-1} \mathcal{U}_{xss}^{i}, \quad \mathcal{W}_{bs}^{loop} = \bigcup_{i=0}^{n-1} \mathcal{W}_{bs}^{i}$$

the safety property is equivalent to verifying that (1) the writes to bs in (any) iteration i do not overlap with the uses of xss in any iteration following i, and (2) the writes to bs across the whole loop do not overlap with the uses of xss after the loop:

$$\mathcal{U}_{xss}^{>i} \;\cap\; \mathcal{W}_{bs}^{i} = \emptyset \qquad \wedge \qquad \mathcal{U}_{xss} \;\cap\; \mathcal{W}_{bs}^{map} = \emptyset$$

Finally, summaries are updated with the accesses within the loop, so that analysis can advance towards the first use of bs, which is represented by the definition of bs_0 (via `scratch`):

$$\mathcal{W}_{bs} \;=\; \mathcal{W}_{bs} \,\cup\, \mathcal{W}_{bs}^{loop}, \quad \mathcal{U}_{xss} \;=\; \mathcal{U}_{xss} \,\cup\, \mathcal{U}_{xss}^{loop}$$

The treatment of `mapnests` is similar, but with two differences:

- The `mapnest` requires that each iteration ends with an implicit update $bs[i] = r$, where r and bs denote the result of the `mapnest` body and of the `mapnest`. It follows that $\mathcal{W}_{bs}^{i} = slice\ ([i, \bar{:}])\ ixfn_{bs}^{new}$.
- Denoting by $\mathcal{U}_{xss}^{<i} = \cup_{j=0}^{i-1} \mathcal{U}_{xss}^{j}$ the uses of xss in all iterations prior to i, the safety property needs to

[25]The imperative reader may think of rs_0, rs and rs' as SSA names referring to the same source array.

[26]If the resulting index function is a composition of LMADs rather than one LMAD then its abstract set is conservatively overestimated, e.g., to $[-\infty, \infty]$.

procedure $NonOverlap(L_1, L_2)$
$t_1, t_2 \leftarrow$ the offsets of L_1, L_2
$I_1, I_2 \leftarrow$ convert L_1, L_2 to sum of intervals of matching strides by distributing the terms of $t_1 - t_2$ positively across LMADs dimensions.
if both I_1 and I_2 have all dimensions non-overlapping
then return whether exists a corresponding pair of non-overlapping intervals
else $L_1^1, L_1^2 \leftarrow splitAnOverlappingDimensionInto2LMADs(I_1)$
$L_2^1, L_2^2 \leftarrow splitAnOverlappingDimensionInto2LMADs(I_2)$
if $L_1^1 =$ **Fail or** $L_2^1 =$ **Fail**
then return False
else return $NonOverlap(L_1^1, L_2^1)$ **and** $NonOverlap(L_1^1, L_2^2)$
and $NonOverlap(L_1^2, L_2^1)$ **and** $NonOverlap(L_1^2, L_2^2)$

Fig. 8. Pseudocode for the procedure that tests that two LMADs do not overlap.

also check that $\mathcal{W}_{bs}^i$ does not overlap $\mathcal{U}_{xss}^{<i}$, because the (parallel) iterations of a `mapnest` execute out-of-order:

$$\mathcal{U}_{xss}^{<i} \cap \mathcal{W}_{bs}^i = \emptyset \;\; \wedge \;\; \mathcal{U}_{xss}^{>i} \cap \mathcal{W}_{bs}^i = \emptyset \;\; \wedge \;\; \mathcal{U}_{xss} \cap \mathcal{W}_{bs}^{map} = \emptyset$$

The partial and total unions are implemented by expanding LMAD dimensions [9], [10], as demonstrated in section II-B.

C. Statically Checking Non-Overlap of a Pair of LMADs

Our non-overlap test is rooted in the following theorem:

Theorem (Non-Overlap)**.** *Given two sum-of-strided intervals with matching strides* $I^1 = \Sigma_{j=1}^{d}[l_j^1 \dots u_j^1] \cdot s_j$ *and* $I^2 = \Sigma_{j=1}^{d}[l_j^2 \dots u_j^2] \cdot s_j$*, such that* $\forall j,\; s_j > 0 \;\wedge\; l_j^1 \geq 0 \;\wedge\; l_j^2 \geq 0$*, then a sufficient condition for non-overlap* $I^1 \cap I^2 = \emptyset$ *is:*

- *Both* I^1 *and* I^2 *have no overlapping dimensions, i.e.,* $s_i > \Sigma_{j=1}^{i-1} u_j^1 \cdot s_j \;\wedge\; s_i > \Sigma_{j=1}^{i-1} u_j^2 \cdot s_j,\; \forall i = 2 \dots d$*, and*
- $\exists j,\; 1 \leq j \leq d$*, such that* $[l_j^1 \dots u_j^1] \cap [l_j^2 \dots u_j^2] = \emptyset$*.*

Our procedure, summarized in fig. 8, converts the pair of LMADs to a pair of sum of intervals of matching dimensions, by exploiting LMAD properties [9], such as: (1) an LMAD can always be normalized to have only positive strides, and (2) dimensions of length 0 can be introduced or removed at will. This step requires a bit of computer algebra support for subtracting the offsets, and iteratively simplifying and distributing the most-complex term of the (sum-of-terms) result to the interval whose (leading term of the) stride is the best match for that term, and such that the term accounts with a positive sign.[27]

If one or both of the sum-of-intervals, say I^1, have overlapping dimensions, then we apply a heuristics that rewrites the interval that has produced the overflow as a union between the last point of the interval and the rest of the interval. This essentially decomposes I^1 into a union of sum-of-intervals $I^1 = I_1^1 \cup I_2^1$, and the test is applied recursively for every possible combination of pairs. Figure 9 shows how non-overlap is proven for the NW example discussed in section III-B.

[27]For example, if the strides of I_1 and I_2 are $nb - b$, n, 1, and the offset is simplified to $t_0 = t_{I_1} - t_{I_2} = nb - b - n - 1$, then the term nb, assumed positive, best matches the stride $nb - b$. t_0 is re-written as $t_0 = (nb - b) + b - b - n - 1$ and its first term $nb - b$ contributes with a "+1" to the lower and upper bounds of the interval of stride $nb - b$ of I_1. The remaining offset is simplified to $t_1 = -n - 1$, and its terms, having negative signs, are (made positive and) distributed to I_2's intervals of strides n and 1.

To Prove: $W \cap R^{vert} = \emptyset$ **assuming** $n = qb + 1,\; 2 \leq q,\; 1 \leq b,\; 0 \leq i$
$W \;=\; ib + n + 1 \;+\; \{(i+1 : nb - b),\; (b : n),\; (b : 1)\}$
$R^{vert} \;=\; ib \;+ \{(i+1 : nb - b),\; (b + 1 : n)\}$

Re-write as sum-of-intervals:
$W_{\cup I} = [0 \dots i] \cdot (nb - b) + [1 \dots b] \cdot n + [1 \dots b] \cdot 1$
$R_{\cup I}^{vert} = [0 \dots i] \cdot (nb - b) + [0 \dots b] \cdot n + [0 \dots 0] \cdot 1$
Both have overlapping dimensions, e.g., $nb - b \not> nb$
Solution: Split 2nd **dim, e.g.,** $[0 \dots b] \cdot n = [0 \dots b - 1] \cdot n \;\cup \{nb\}$

To Prove: $(W_{\cup I}^1 \cup W_{\cup I}^2) \cap (R_{\cup I}^{vert,1} \cup R_{\cup I}^{vert,2}) = \emptyset,$ **where**
$W_{\cup I}^1 = [0 \dots i] \cdot (nb - b) + [1 \dots b - 1] \cdot n + [1 \dots b] \cdot 1$
$W_{\cup I}^2 = [1 \dots i + 1] \cdot (nb - b) + [0 \dots 0] \cdot n + [1 + b \dots 2b] \cdot 1$
$R_{\cup I}^{vert,1} = [0 \dots i] \cdot (nb - b) + [0 \dots b - 1] \cdot n + [0 \dots 0] \cdot 1$
$R_{\cup I}^{vert,2} = [1 \dots i + 1] \cdot (nb - b) + [0 \dots 0] \cdot n + [b \dots b] \cdot 1$

All have non-overlapping dimensions if $q \geq 2 \wedge b \geq 1$
$W_{\cup I}^1 \cap R_{\cup I}^{vert,2} = \emptyset$ **holds because** $[1 \dots b - 1] \cdot n \cap [0 \dots 0] \cdot n = \emptyset$
$W_{\cup I}^1 \cap R_{\cup I}^{vert,1} = \emptyset \;\;\wedge\;\; W_{\cup I}^2 \cap R_{\cup I}^{vert,1} = \emptyset \;\;\wedge\;\; W_{\cup I}^2 \cap R_{\cup I}^{vert,2} = \emptyset$
holds because non-overlap in the first dimension, e.g.,
$[1 + b \dots 2b] \cdot 1 \cap [b \dots b] \cdot 1 = \emptyset$ **and** $[1 \dots b] \cdot 1 \cap [0 \dots 0] \cdot 1 = \emptyset$

Fig. 9. Proving the NW example in section III-B. In practice we need to prove less, i.e., $W_q \;\cap R_{j>q}^{vert} = \emptyset$, which succeeds statically when $q \geq 1$.

D. Implementation Details

The techniques and methods described in this paper are fully implemented as an automated pass in a dedicated version of the Futhark compiler. In all, it takes up around 5000 lines of Haskell code. The simple inequalities resulting from the analysis in section V-C are currently solved using an external SMT solver when checking LMAD non-overlap, but we are working on replacing this with a simpler symbolic algebra engine inside the compiler. Note that the SMT solver cannot by itself verify non-overlap of e.g. NW; we use it only for testing the inequalities generated by the non-overlap theorem.

Short-circuiting causes around 10% compile time overhead for most benchmarks, with only NW and LUD taking longer. Particular, due to the SMT solver, NW takes 17 seconds to compile with short-circuiting compared to 1 second without.

VI. Experimental Evaluation

To investigate the impact of array short-circuiting, we have implemented it in the Futhark compiler, and validated it by testing it on a collection of benchmarks.

A. Experimental Methodology

Each benchmark was run on an NVIDIA A100 and AMD MI100 GPU. To ensure accurate measurements, we run each benchmark a number of times, as indicated by the header of each table, always discarding the first run and measuring the average wall time of the rest. The full experimental methodology is described in the artifact description.

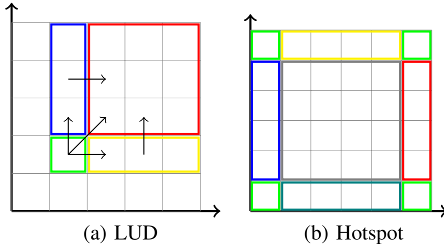
B. Case Study: NW

The NW benchmark has already been extensively discussed. It uses a complex parallelization pattern as shown in fig. 2, but our LMAD slices allow us to take a flattened representation of the matrix and directly express the LMADs we want.

Using LMAD slices, our short-circuiting pass correctly recognizes that the blocks in each iteration can be computed in-place, which reduces copying overhead, resulting in significant

TABLE I
NW PERFORMANCE (1000 RUNS)

	Dataset	Ref.	Unopt. Futhark	Opt. Futhark	Opt. Impact
A100	8192	9ms	0.99x	1.16x	1.17x
	16384	21ms	0.96x	1.19x	1.24x
	32768	58ms	1.04x	1.36x	1.31x
MI100	8192	15ms	0.71x	0.88x	1.24x
	16384	44ms	0.64x	0.78x	1.21x
	32768	325ms	1.01x	1.14x	1.13x



(a) LUD (b) Hotspot
Fig. 10. The access patterns for LUD and Hotspot.

performance improvements as shown in table I. Overall, we see impacts of between $1.1\times$ and $1.3\times$, which means that we are outperforming the hand-written Rodinia implementation on the largest datasets.

C. Case Study: LUD

The Futhark implementation of LUD closely mimics Rodinia's hand-written OpenCL implementation. At a high level, the implemented algorithm separates the input matrix into blocks and iteratively processes the resulting blocked matrix along the diagonal. Figure 10a shows an example at the second iteration of the outer loop: First, the green diagonal block is processed individually. The result is used to compute the blue and yellow blocks, all of which are used to updated the remaining inner red blocks. The loop then continues inside the red blocks. In principle, and in the Rodinia implementation, this can all be done in-place, but without arrray short-circuiting Futhark will put the intermediate arrays in temporary memory allocations, with significant copying overhead.

Short-circuiting determines that the yellow and red blocks can be constructed in-place. Due to various other compiler optimizations (e.g. layout changes to enable coalesced access for the blue blocks) the green and blue blocks are not computed in-place, but we still see significant performance improvements from short-circuiting ($1.19 - 1.39\times$), as shown in table II. The resulting code is more efficient than Rodinias implementation, because Futhark automatically performs both block and register tiling, while Rodinia's code only uses block tiling—this seems to have higher impact on A100.

D. Case Study: Hotspot

Hotspot from Rodinia is a repeated stencil computation. The stencil boundaries are treated separately as shown in fig. 10b: The corners (in green) are handled first, then the four edges (which have similar access patterns) and finally the internal

TABLE II
LUD PERFORMANCE (10 RUNS)

	Dataset	Ref.	Unopt. Futhark	Opt. Futhark	Opt. Impact
A100	8192	190ms	1.08x	1.34x	1.25x
	16384	1445ms	1.19x	1.53x	1.29x
	32768	11547ms	1.21x	1.60x	1.32x
MI100	8192	173ms	0.60x	0.72x	1.19x
	16384	1248ms	0.74x	0.98x	1.32x
	32768	10511ms	0.83x	1.14x	1.39x

TABLE III
HOTSPOT PERFORMANCE (10 RUNS)

	Dataset	Ref.	Unopt. Futhark	Opt. Futhark	Opt. Impact
A100	8192	9ms	0.47x	0.84x	1.78x
	16384	29ms	0.46x	0.94x	2.04x
	32768	117ms	0.46x	0.94x	2.05x
MI100	8192	8ms	0.33x	0.64x	1.96x
	16384	34ms	0.35x	0.68x	1.97x
	32768	142ms	0.37x	0.73x	1.98x

TABLE IV
LBM PERFORMANCE (100 RUNS)

	Dataset	Ref.	Unopt. Futhark	Opt. Futhark	Opt. Impact
A100	short	29ms	0.84x	0.92x	1.09x
	long	860ms	0.86x	0.95x	1.10x
MI100	short	49ms	0.65x	1.04x	1.59x
	long	1423ms	0.63x	1.01x	1.60x

cells. Because the new value of each cell depends on the old value of its neighbours, we cannot perform the computation in-place. Instead we compute the different parts separately and concatenate them at the end. Without short-circuiting, each of the intermediate arrays reside in separate memory blocks and must be copied to form the result. Short-circuiting causes the intermediate arrays to be constructed directly in the result memory, giving speedups of up to 2x, as shown in table III. The optimised code almost reaches the performance of the hand-written Rodinia implementation on the A100.

E. Case Study: LBM

The LBM benchmark from Parboil [20] is an implementation of the Lattice-Boltzmann Method. Table IV shows the impact of our optimization on the Futhark implementation. We see significant improvement on the MI100, which results in outperforming the reference implementation slightly. On the A100, Futhark was already quite close to matching Parboil's performance, but the $1.1\times$ optimisation impact brings it even closer.

TABLE V
OPTIONPRICING PERFORMANCE (1000 RUNS)

	Dataset	Ref.	Unopt. Futhark	Opt. Futhark	Opt. Impact
A100	medium	1ms	0.78x	0.80x	1.03x
	large	18ms	0.58x	0.70x	1.21x
MI100	medium	13ms	4.19x	4.70x	1.12x
	large	28ms	0.65x	0.74x	1.14x

TABLE VI
LOCVOLCALIB PERFORMANCE (10 RUNS)

	Dataset	Ref.	Unopt. Futhark	Opt. Futhark	Opt. Impact
A100	small	103ms	0.97x	1.05x	1.08x
	medium	50ms	1.18x	1.27x	1.07x
	large	169ms	0.63x	0.68x	1.08x
MI100	small	207ms	1.08x	1.20x	1.12x
	medium	84ms	0.92x	0.97x	1.06x
	large	431ms	0.76x	0.79x	1.04x

F. Case Study: OptionPricing

OptionPricing is an implementation of the extended option pricing engine from Finpar [21]. The impact here is more modest, but still up to $1.2\times$. As seen in table V our implementation doesn't quite reach the performance of the reference implementation (except for one case on the MI100 where the reference implementation is unusually slow), but this is at least no longer due to the overhead of copying.

G. Case Study: LocVolCalib

The LocVolCalib benchmark from FinPar is an implementation of contract price volatility calibration. Except for largest dataset, short-circuiting allows us to match or out-compete the reference implementation, as seen in table VI.

H. Case Study: NN

The NN benchmark from Rodinia is an implementation of K-nearest neighbors. The Futhark version contains a loop with a reduction whose result is used in an in-place update, resulting in a copy. Short-circuiting correctly identifies that the result of the reduce can be put directly in the memory of the result, eliminating a copy. Table VI shows the performance of our Futhark implementation with and without optimizations, compared to the reference Rodinia implementation. Rodinia is significantly slower, because it uses a sequential reduction.

VII. RELATED WORK

Our IR design is philosophically related to the use of *region inference* for memory management in compilers for functional languages [22], as for example used in MLKit [23]. But whereas regions can be seen as lexically scoped *heaps* that potentially contain multiple objects, our memory blocks represent single allocations and can have non-lexical lifetimes. In contrast to standard functional languages, array languages such as Futhark exhibit fewer but larger allocations. *Destination-passing style* [24] is an adaptation of the region approach to an array language, but lacks our notion of index functions to address the *layout* of arrays, and also does not support non-lexical lifetimes, or index-analysis based optimizations such as short-circuiting arrays.

TABLE VII
NN PERFORMANCE (100 RUNS)

	Dataset	Ref.	Unopt. Futhark	Opt. Futhark	Opt. Impact
A100	855280	70ms	9.82x	15.19x	1.55x
	8552800	631ms	76.48x	93.18x	1.22x
	85528000	6194ms	197.66x	208.02x	1.05x
MI100	855280	70ms	5.06x	6.78x	1.34x
	8552800	630ms	39.11x	46.08x	1.18x
	85528000	6280ms	115.72x	126.18x	1.09x

Sisal's "Build-in-Place" analysis seeks to avoid copying when incrementally constructing an array [25], and is similar to—but more limited than—array short-circuiting, and has no way to express general memory layout optimizations.

Driven by polyhedral analysis, SMO [26] reduces the memory footprint of imperative programs by aggressively reusing the same memory block for multiple semantically distinct arrays, as long as their *per-element* live ranges do not overlap.

A significant body of work presents dataflow-graph DSLs. DFGL [27] proposes a dependency-based notation that is lowered and optimized in a polyhedral framework, e.g., supporting legality checks for absence of deadlocks and safety of parallelization. Our work addresses the reverse problem: that of a non-restricted language using conventional type checking that requires the separation of reads and writes, and in which the efficiency of the in-place specification is recovered by short-circuiting arrays. Other approaches build on the idea of separating the program specification from the optimization recipe, pioneered by Chill [28], [29]. For example, several dataflow DSLs are aimed at fusing image-processing pipelines [30]–[32], e.g., by means of overlapped-tiling and sliding window transformations. Here, memory is introduced at the very end in a way tailored to the optimization recipe, to optimize the placement/footprint of intermediates in the memory hierarchy. Our work has a different focus, for example standalone transformations on the memory IR (circuiting) that are not expressible on the array IR.

Short-circuiting is related at a high-level with analyses developed in the context of automatic parallelization of loop-based (Fortran) code [1]–[3], [9], [16], [33], [34]. Such analyses classify memory references into, for example, read-only (RO), write-first (WF) and read-write (RW) kinds, aggregate each kind across complex (target) loops, and model loop parallelism as a set equation that uses the resulting RO, WF and RW summaries. LMADs [9], [10] have been used as the building blocks for representing such summaries [3], [5], [35]. Short-

circuiting addresses a simpler (analogous) problem in which parallelism is already expressed but the copying needs to be optimized. This leads to a simpler classification of memory accesses—the reads of bs and the uses (read+writes) of xss—and more importantly to simpler formulas for aggregating accesses, which in our case require only (repeated) unions of LMADs, but not subtraction and intersection, which are much more challenging. Finally, if the analysis conservatively fails, in our context we pay with an overhead between $1.1 - 2\times$, while failure of automatic parallelization is catastrophic.

Our test for empty intersection of LMADs is inspired from and is an extension of the one presented in [9]. The difference is that we use a sum-of-interval representation to determine overlap of (1) dimensions within an LMAD, and (2) pairwise between two LMADs. When LMAD dimensions overlap we use heuristics like splitting the offending dimension rather than failing immediately, leading to less conservative results.

LMADs are closely related to *dope vectors*, which have a long history of being used to represent metadata about multidimensional arrays in array languages. For example, the run-time representation of arrays in Sisal [36] uses a dope vector whose contents are equivalent to a single LMAD, as well as extra metadata such as reference counts. However, dope vectors have been treated as a run-time data structure, to be referenced whenever an array is indexed. In contrast, we use LMADs at *compile-time*, for letting the compiler reason about and optimise based on array layouts, and the actual structure of the LMAD for a given array is inlined for every array access during final code generation.

VIII. Conclusions

We have shown that LMADs can be used to generalize slicing in the source language and IR of a functional language. We demonstrated that LMADs can be used to extend the IR of a purely functional language to add a notion of memory. This representation allows the compiler to express memory based optimizations akin to the ones found in imperative languages. We detailed such an optimization, called array short-circuiting, which aims to mimic a common imperative programming technique. Finally, we have implemented said optimization in a purely functional array language, and we showed that it has significant impact on some benchmarks, sometimes even beating the reference hand-written implementation.

Acknowledgments

We are grateful to Niels G. W. Serup for his initial prototyping work related to memory optimizations. This work has been supported by the Independent Research Fund Denmark (DFF) under the grants *FUTHARK: Functional Technology for High-performance Architectures* and *Monitoring Changes in Big Satellite Data via Massively Parallel AI*, and by the UCPH Data-Plus grant: *High-Performance Land Change Assessment*.

References

[1] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, “Interprocedural Parallelization Analysis in SUIF,” *Trans. on Prog. Lang. and Sys. (TOPLAS)*, vol. 27(4), pp. 662–731, 2005.

[2] S. Moon and M. W. Hall, “Evaluation of Predicated Array Data-Flow Analysis for Automatic Parallelization,” in *Int. Symp. Princ. and Practice of Par. Prog. (PPoPP)*, 1999, pp. 84–95.

[3] S. Rus, J. Hoeflinger, and L. Rauchwerger, “Hybrid Analysis: Static & Dynamic Memory Reference Analysis,” *Int. Journal of Par. Prog*, vol. 31(3), pp. 251–283, 2003.

[4] H. Yu and L. Rauchwerger, “Techniques for Reducing the Overhead of Run-Time Parallelization,” in *Procs. Int. Conf. on Compiler Construction*, 2000, pp. 232–248.

[5] C. E. Oancea and L. Rauchwerger, “Logical inference techniques for loop parallelization,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’12. New York, NY, USA: ACM, 2012, pp. 509–520. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254124

[6] A. Venkat, M. Hall, and M. Strout, “Loop and data transformations for sparse matrix code,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 521–532. [Online]. Available: https://doi.org/10.1145/2737924.2738003

[7] P. Chatarasi, J. Shirako, and V. Sarkar, “Polyhedral optimizations of explicitly parallel programs,” in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, ser. PACT ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 213–226. [Online]. Available: https://doi.org/10.1109/PACT.2015.44

[8] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’08. New York, NY, USA: ACM, 2008, pp. 101–113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595

[9] J. Hoeflinger, Y. Paek, and K. Yi, “Unified Interprocedural Parallelism Detection,” *Int. Journal of Par. Prog*, vol. 29(2), pp. 185–215, 2001.

[10] Y. Paek, J. Hoeflinger, and D. Padua, “Efficient and Precise Array Access Analysis,” *Trans. on Prog. Lang. and Sys. (TOPLAS)*, vol. 24(1), pp. 65–109, 2002.

[11] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea, “Incremental flattening for nested data parallelism,” in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’19. New York, NY, USA: ACM, 2019, pp. 53–67. [Online]. Available: http://doi.acm.org/10.1145/3293883.3295707

[12] T. Henriksen, S. Hellfritzsch, P. Sadayappan, and C. Oancea, “Compiling generalized histograms for gpu,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’20. IEEE Press, 2020.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 10 2009, pp. 44–54.

[14] J. Svensson, “Obsidian: GPU kernel programming in Haskell,” Ph.D. dissertation, Chalmers University of Technology, 2011.

[15] A. Paszke, D. D. Johnson, D. Duvenaud, D. Vytiniotis, A. Radul, M. J. Johnson, J. Ragan-Kelley, and D. Maclaurin, “Getting to the point: Index sets and parallelism-preserving autodiff for pointful array programming,” *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, aug 2021. [Online]. Available: https://doi.org/10.1145/3473593

[16] S. Rus, M. Pennings, and L. Rauchwerger, “Sensitivity Analysis for Automatic Parallelization on Multi-Cores,” in *Procs. Int. Conf. on Supercomp*, 2007, pp. 263–273.

[17] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, “Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 556–571. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062354

[18] G. D. Plotkin, “A Note on Inductive Generalization,” in *Machine Intelligence*, 1970, pp. 153–163.

[19] T. A. Mogensen, *Introduction to Compiler Design*, 1st ed. Springer Publishing Company, Incorporated, 2011.

[20] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[21] C. Andreetta, V. Bégot, J. Berthold, M. Elsman, F. Henglein, T. Henriksen, M.-B. Nordfang, and C. E. Oancea, "Finpar: A parallel financial benchmark," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 2, pp. 18:1–18:27, Jun. 2016.

[22] M. Tofte and J.-P. Talpin, "Implementation of the typed call-by-value λ-calculus using a stack of regions," in *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 188–201. [Online]. Available: https://doi.org/10.1145/174675.177855

[23] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg, "A retrospective on region-based memory management," *Higher-Order and Symbolic Computation (HOSC)*, vol. 17, no. 3, pp. 245–265, 9 2004.

[24] A. Shaikhha, A. Fitzgibbon, S. Peyton Jones, and D. Vytiniotis, "Destination-passing style for efficient memory management," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, ser. FHPC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 12–23. [Online]. Available: https://doi.org/10.1145/3122948.3122949

[25] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the sisal language project," *Journal of Parallel and Distributed Computing; (United States)*, 12 1990. [Online]. Available: https://www.osti.gov/biblio/5001807

[26] S. G. Bhaskaracharya, U. Bondhugula, and A. Cohen, "Smo: An integrated approach to intra-array and inter-array storage optimization," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 526–538. [Online]. Available: https://doi.org/10.1145/2837614.2837636

[27] A. Sbîrlea, J. Shirako, L.-N. Pouchet, and V. Sarkar, "Polyhedral

[34] ——, "Scalable conditional induction variables (civ) analysis," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 213–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=2738600.2738627

optimizations for a data-flow graph language," in *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, ser. LCPC 2015. Berlin, Heidelberg: Springer-Verlag, 2015, p. 57–72. [Online]. Available: https://doi.org/10.1007/978-3-319-29778-1_4

[28] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," Tech. Rep., 2008.

[29] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance cuda code," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, jan 2013. [Online]. Available: https://doi.org/10.1145/2400682.2400690

[30] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 519–530.

[31] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 429–443. [Online]. Available: https://doi.org/10.1145/2694344.2694364

[32] E. C. Davis, M. M. Strout, and C. Olschanowsky, "Transforming loop chains via macro dataflow graphs," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 265–277. [Online]. Available: https://doi.org/10.1145/3168832

[33] C. E. Oancea and L. Rauchwerger, "A hybrid approach to proving memory reference monotonicity," in *Languages and Compilers for Parallel Computing*, S. Rajopadhye and M. Mills Strout, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 61–75.

[35] S. Rust, G. He, C. Alias, and L. Rauchwerger, "Region array SSA," in *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006, pp. 43–52.

[36] R. Oldehoeft, "Implementing arrays in sisal 2.0," in *Proceedings of the 2nd SISAL Users Conference, San Diego, California, USA*, 1992, pp. 209–222.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Our paper shows how LMADs can be used to extend the IR of a functional language in a way that lets the compiler express memory-based optimizations like the ones found in imperative languages. As an example of one such optimization, we introduce short-circuiting, which aims to reduce the use of intermediate arrays and eliminate redundant copies, resulting in improved performance. In many cases the resulting code should closely mimic what would be written by hand. To validate our work, we've implemented short-circuiting as a compiler pass in the Futhark compiler and compared the performance of seven benchmarks with and without said optimization, as well as compared to a reference hand-written optimization. For each benchmark, our paper shows a table, which can be reproduced using the artifacts included, given the right hardware. In particular, we have used computers equipped with the following GPUs: NVIDIA A100 and AMD MI100.

We support two different methods of running the experiments from the paper: Directly on your host-machine or inside one of the provided containers. We recommend using one of the provided containers.

The DOI contains instructions and trouble-shooting tips on how to get everything working, but for convenience we have summarized the instructions below.

Running benchmarks in a container

For maximum reproducibility, we supply Docker-containers which can be used to replicate the results from our article, as described below.

For NVIDIA devices, additional steps are needed to ensure that Docker containers have access to the hosts GPU devices. Follow the instructions to listed on at https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html#docker to set up and install the NVIDIA Container Toolkit.

We supply two containers:

(1) `futhark-mem-sc22:cuda` - targeted at CUDA devices (such as NVIDIAs A100)
(2) `futhark-mem-sc22:rocm` - targeted at ROCM devices (such as AMDs MI100)

The containers have been uploaded to the Github container registry. They can be executed using the following commands for CUDA and ROCM respectively.

`docker run –rm –i –t –gpus all ghcr.io/diku-dk/futhark-mem-sc22:cuda bash`

`docker run –rm –i –t –device=/dev/kfd –device=/dev/dri –security-opt seccomp=unconfined –group-add video ghcr.io/diku-dk/futhark-mem-sc22:rocm bash`

Running the commands will pull and execute the container in question, putting you in a command prompt in the benchmarks directory. There, you can run `make tables` to run all benchmarks or e.g. `make table1` to reproduce individual tables. Use `make help` for additional information.

Alternatively, you can automatically run and display all tables by executing one of the following commands:

`docker run –rm –t –gpus all ghcr.io/diku-dk/futhark-mem-sc22:cuda`

`docker run –rm –t –device=/dev/kfd –device=/dev/dri –security-opt seccomp=unconfined –group-add video ghcr.io/diku-dk/futhark-mem-sc22:rocm`

Running benchmarks on your host-machine

Alternatively, you can run the benchmarks directly on your host machine using the repository in the futhark-mem-sc22 artifact.

If you have installed and configured an OpenCL capable GPU (we use NVIDIAs A100 and AMDs MI100 in our article), you should be able to run the experiments using :

`make all`

This will compile and run both reference- and Futhark-implementations of all benchmarks using the Futhark binary in `bin` and show the resulting performance tables in ASCII. To use another version of Futhark, use `make FUTHARK=my-futhark all`.

Alternatively, you can reproduce the experiment for each table individually by running e.g. `make table1` in the benchmarks directory:

Benchmark results are cached, so running `make table1` a second time will be instantaneous. To cleanup cached results, use `make clean`.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1
Persistent ID: `https://doi.org/10.5281/zenodo.6452039`
Artifact name: futhark-mem-sc22

Artifact 2
Persistent ID: `ghcr.io/diku-dk/futhark-mem-sc22:cuda`
Artifact name: futhark-mem-sc22:cuda

Artifact 3
Persistent ID: `ghcr.io/diku-dk/futhark-mem-sc22:cuda`
Artifact name: futhark-mem-sc22:cuda

Artifact 4
Persistent ID: `https://github.com/diku-dk/futhark-mem-sc22/`
Artifact name: futhark-mem-sc22 on Github

Reproduction of the artifact with container: To run the CUDA container, which can be used to reproduce the A100 results from the paper, execute the following command:

docker run –rm -i -t –gpus all ghcr.io/diku-dk/futhark-mem-sc22:cuda bash

To run the ROCM container, which can be used to reproduce the MI100 results from the paper, execute the following command:

docker run –rm -i -t –device=/dev/kfd –device=/dev/dri –security-opt seccomp=unconfined –group-add video ghcr.io/diku-dk/futhark-mem-sc22:rocm bash

Executing either of the two commands above will give you access to a shell within the container. Now you can reproduce the tables individually by running e.g. "make table1", or all of the tables by running "make tables".

For additional information, please consult the README.md file in the "futhark-mem-sc22" artifact.