In-Place-Folding of Non-Scalar Hyper-Planes of Multi-Dimensional Arrays

Gijs van Cuyck
gijs.vancuyck@ru.nl
Radboud University
Nijmegen, Netherlands

Sven-Bodo Scholz
SvenBodo.Scholz@ru.nl
Radboud University
Nijmegen, Netherlands

ABSTRACT

Memory management plays a key role when trying to compile functional programs into efficiently executable code. In particular when using flat representations for multi-dimensional arrays, i.e., when using a single memory block for the entire data of a multi-dimensional array, in-place updates become crucial for highly competitive performance.

This paper proposes a novel code generation technique for performing fold-operations on hyper-planes of multi-dimensional arrays, where the fold-operation itself operates on non-scalar sub-arrays, i.e., on vectors or higher-dimensional arrays. This technique allows for a single result array allocation over the entire folding operation without requiring the folding operation itself to be scalarised. It enables the utilisation of vector operations without any added memory allocation or copying overhead. We describe our technique in the context of SaC, sketch our implementation in the context of the compiler `sac2c` and provide some initial performance measurements that give an indication of the effectiveness of this new technique.

CCS CONCEPTS

• **Software and its engineering → Compilers**; *Functional languages.*

KEYWORDS

compiler optimisation, reference counting, SaC, memory management

ACM Reference Format:
Gijs van Cuyck and Sven-Bodo Scholz. 2021. In-Place-Folding of Non-Scalar Hyper-Planes of Multi-Dimensional Arrays. In *33rd Symposium on Implementation and Application of Functional Languages (IFL '21), September 1–3, 2021, Nijmegen, Netherlands.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3544885.3544893

1 INTRODUCTION

High-Level array languages such as Futhark[11], Accelerate[4], Lift[19], Halide[15], or SaC[16] have demonstrated that it is possible to generate very efficient parallel codes from abstract problem specifications. This resonates very well with the functional credo of the "what not how" and it opens up competitive parallel performance to domain experts without requiring them to become HPC experts. While it has been shown across many different projects that this goal can be reached in principle, we are still far from having techniques that can cope with all possible high-level expressions equally well, let alone having a single tool chain that comprises all known techniques. In particular when applying the array approach to new application areas, we typically identify new code pattern that are not yet covered well enough to allow the programmer to solely concentrate on the "what".

This paper focuses on one particular code-pattern which appears frequently in applications that naturally lend themselves to algorithmic descriptions on higher-dimensional arrays (dimensionality ≥ 3) such as for example CNNs (Convolutional Neural Networks)[24]. More specifically, it deals with fold-operations that are mapped across the outer dimensions of higher-dimensional arrays whose folding operation itself computes non-scalar arrays from non-scalar arrays. The challenge here is memory management. In order to avoid excessive memory allocations or copying, we need to perform the reduction operation in-place on the result array. As the folding operation expects arrays as arguments and produces arrays as results, any intermediate "accumulator values" are arrays as well. Furthermore, since the folding operation does not operate on the outermost axis but is mapped across the outermost axis, any in-place folding has to be done entirely on sub-arrays, requiring accumulator values to be mapped into parts of the result array.

One way to avoid this problem is to re-write the computation into a semantically equivalent one that performs folding-operations on the innermost dimension(s). Optimisations such as *With-Loop-Scalarization* [9] aim at such transformations. Unfortunately, such a re-write (i) is not always possible, (ii) for the given scenario leads to poor spatial locality of memory accesses, and (iii) it typically inhibits the use of vector operations.

In this paper, we propose a code generation technique that allows such reductions on sub-arrays to be performed in place. It constitutes a novel extension of the code generation techniques that have been developed in the context of SaC [10]. Our contributions are:

- a clear identification of the code pattern that poses the memory management challenge
- an analysis how this challenge can be met in the context of flat array representations
- a code generation scheme that enables in-place reductions on non-scalar hyper-planes

Section 2 gives a short overview of the SaC language and some of its features relevant to the presented work, section 3 provides a detailed account of the problem the new code generation technique

IFL '21, September 1–3, 2021, Nijmegen, Netherlands
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8644-9/21/09.
https://doi.org/10.1145/3544885.3544893

will resolve. Section 4 briefly explains the pre-existing code generation before section 5 relates this to the problem at hand. Section 6 then proposes a solution in the form of the "in-place accumulator optimisation", a modification of the code generation process. We quantify the effect of our optimisation in section 7 where we discusses the results of several benchmark comparisons between optimised and non-optimised code. Sections 8 and 9 then discuss related work and summarise the drawn conclusions, respectively.

2 SAC

SaC is an array language with N-dimensional arrays at its core. Scalar values are considered zero-dimensional arrays, vectors are one-dimensional arrays, matrices are two-dimensional arrays, and so forth. Arrays in SaC are described by their data and their shape. The shape of an array can be accessed with the `shape` function. For instance, an array with shape `[2,4,5]` has three dimensions. The first dimension has two elements, and each element can be seen as an array of shape `[4,5]`. Indexing in SaC looks and works the same as in C. An array can be suffixed with square brackets and an index to select a specific part of the array.

There are three main ways in SaC to create arrays. An array literal can be used, enumerating all the elements of an array. These can also be nested to any depth. For instance, `[[1,2,3],[4,5,6]]` is an array of shape `[2,3]`. The empty array, `[]`, has shape `[0]`, a one-dimensional array with no elements. The number 2 is considered an array of shape `[]`, a zero-dimensional array. The ability to look at the shape of any value, even the shape of a shape, or the shape of a number, allows the definition of algorithms that work for arbitrarily shaped input.

The `genarray` construct is a slightly more general approach to creating arrays. It takes a shape description and a single element, and then fills an array of the given shape with copies of that element. For instance, `genarray([2,3],1)` will produce the same array as `[[1,1,1],[1,1,1]]`. This will also work if the element is a non-scalar array. The same array as the previous example can for instance also be obtained by `genarray([2],[1,1,1])`, or by `genarray([2],genarray([3],1))`.

The third and most general approach to defining arrays in SaC is the with-loop. It gives a description of an array, by successively calling some code that generates an individual element, parameterised by the index of that element. This can be seen as a for loop, but instead of a variable representing the iteration number, there are variables representing the current indices into the resulting array. As an example, figure 1 uses a with-loop to describe the array shown in figure 1b with the code `9 - abs(x-y)`, where `x` and `y` are the indices into the array. This with-loop uses the keyword `genarray`, to indicate that it is creating a new array of shape `[5,10]` with default element `0`. While the example of figure 1 uses constants to keep it simple, the bounds of the with-loop (`[0,0]` and `[5,10]`) can be arbitrary expressions. Even the vector of indices (`[x,y]`) can be represented by a single variable to abstract away over the number of indices. The lower and upper bound can be defined by using the `shape` function over an existing array. This can then be used to define functions that are polymorphic over the shape of their arguments. Indices within the bounds are filled using the body of the withloop. Any indices outside the bounds will be filled using

```
diagonal = with {
    ( [0,0] <= [x,y] < [5,10] ): 9 - abs(x-y);
  } : genarray( [5,10], 0);
```

(a) Example code

```
Result:
                         9, 8, 7, 6, 5, 4, 3, 2, 1, 0
                         8, 9, 8, 7, 6, 5, 4, 3, 2, 1
                         7, 8, 9, 8, 7, 6, 5, 4, 3, 2
                         6, 7, 8, 9, 8, 7, 6, 5, 4, 3
                         5, 6, 7, 8, 9, 8, 7, 6, 5, 4
```

(b) Value of `diagonal` variable

Figure 1: A genarray-with-loop example

```
int[.,.] diagonal (int x, int y) {
    return with {
        ( [0,0] <= ivec < [x,y] ) :
            (max(x,y) - 1) - abs(ivec[0]-ivec[1]);
      } : genarray( [x,y], 0);
}
```

Figure 2: A shape-polymorphic function with with-loop body

the default element. Figure 2 contains a function calculating a more general version of the code in figure 1. Instead of always calculating a matrix of constant size, the dimensions of the matrix can now be provided as arguments, and the return type is adjusted to `int[.,.]` expressing the result's dimensionality only.

In addition to the `genarray`-with-loop, SaC has several other types of with-loops. In this paper, we focus on the `genarray` and `fold` variants, as these are the with-loops that are involved in reductions over hyper-planes. An example of a fold-with-loop can be seen in figure 3. This example shows how the summation of the rows of a matrix can be defined by a with-loop. The with-loop describes the selection of individual rows by the parameterised expression `input[iv]`. Instead of putting all of these expressions into an array, the fold-with-loop uses a reduction operation (+), and a neutral element (`genarray([shape(input)[1]],0)`), to combine all the elements into a single result. The neutral element here describes a vector of zeroes, whose length matches the second dimension of `input`. These example with-loops describe their result using a single expression. For more complex with-loops a block of local assignments can precede the expression.

Apart from these basic tools, there are many array operations defined through with-loops in the standard library of SaC such as take, drop, map, etc. They fall outside the scope of this paper, but are described in more detail in the official SaC tutorial [17].

3 PROBLEM STATEMENT

As running example, we consider computing a rolling sum over three rows of a matrix. For simplicity, we fill the last two rows

```
int[.] sum(int[.,.] input){
  return with {
        ( [0] <= iv < shape(input)[0] ) :
            input[iv];
    } : fold( +, genarray([shape(input)[1]],0));
}
```

Figure 3: Computing the sum of rows of a two dimensional array with a with-loop

of the result with the value `0`. Figure 4 shows three possible implementations in SaC. Figure 4a defines this operation through an element-wise addition using the expression `a[i,j] + a[i+1,j] + a[i+2,j]` to express the summation of corresponding `j`-th elements in the `i`-th, `i+1`-th, and `i+2`-th row. This operation is mapped to all elements of the given matrix `a`, but those of the last two rows. Figure 4b looks very similar. However, here the explicit summation is specified for entire rows, which are obtained by selections `a[i]`, `a[i+1]`, and `a[i+2]`. Consequently, this operation is mapped across row indices rather than element indices. Finally, we look at a slightly more generic row-wise formulation shown in figure 4c. It uses a fold-with-loop to compute the sum of rows. The +3 on line 6 ensures that we look at adding three rows here. However, the number of rows to be summed up can be adjusted by changing that number, provided the bound of the outer with-loop (`800000-2`) is adjusted accordingly.

While the results of these three implementations are equivalent, their runtimes are not. The first two are roughly equivalent, but the fold version is significantly slower. One possible cause for this is that the fold version does one addition more than the other versions. I.e., it calculates `0 + a[i] + ...`, while the others only calculate `a[i] + ...`. To compensate for this, the first two versions are changed slightly to enforce the extra addition of `0` as well.

All runtime results presented in this paper are obtained using a dual-core Intel E5-2650L at 2.5 GHz with 128 GB ram running Arch-Linux kernel 5.10. For compilation, we use sac2c 1.3.3-MijasCosta and gcc 10.2.0. The memory results are obtained by using the `-profile m` option of the SaC compiler. The measurements for all three versions are shown in figure 5. They are obtained by executing a program that calls one of the three rowadd functions exactly once. To compensate for random background noise, each function is executed 100 times.

Figure 5a shows the 95 runtimes closest to the mean runtime for each function. The versions V1 and V2 perform roughly equivalently with an average runtime of 5.8 seconds. The V3 version, however, is noticeably slower with an average runtime of 6.3 seconds. This means that the fold-based implementation for this example is almost 10% slower than the other two versions.

The memory use of Rowadd_V3 stands out as well, as seen in figure 5b. While all three versions have the same memory footprint of 6.25 GB, i.e., the maximum memory allocated at any given time, rowadd_V3 performs 800.000 memory allocations while the other two versions only perform 3 allocations. Given that integers take 4 bytes of memory, we can see that at any given time no more than two copies of the entire matrix with 800M elements are allocated. When looking at the total allocation figures, we see that versions V1

```
int[800000,1000] rowadd_V1( int[800000,1000] a)
{
  res = with {
      ( [0,0] <= [i,j] < shape(a) - [2,0]) :
          a[i,j]+ a[i+1,j]+a[i+2,j];
    } : genarray([800000,1000],0);
  return res;
}
```

(a) Adding rows by adding individual elements

```
int[800000,1000] rowadd_V2( int[800000,1000] a)
{
  res = with {
      ( [0] <= [i] < [800000-2]) :
          a[i]+ a[i+1]+a[i+2];
    } : genarray([800000],genarray([1000],0));
  return res;
}
```

(b) Adding rows by adding whole rows at a time

```
1  int[800000,1000] rowadd_V3( int[800000,1000] a)
2  {
3    res = with {
4        ( [0] <= iv < [800000-2]) :
5          with {
6              (iv <= jv < iv+3) : a[jv];
7            } : fold( +, genarray([1000], 0));
8      } : genarray([800000],genarray([1000],0));
9    return res;
10 }
```

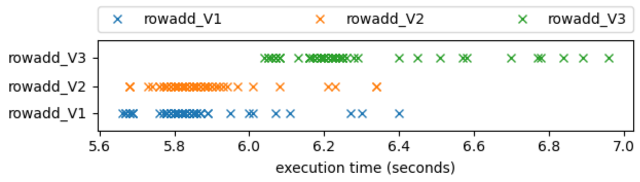
(c) Adding rows using a fold-with-loop

Figure 4: Three ways of a rolling addition of three rows of a matrix

and V2 allocate these two copies exactly once. In contrast, version V3 over the execution time uses 800.000 allocations to allocate the equivalent of one more copy. Given that this has no noticeable impact on the memory footprint, we can deduce that V3 most likely allocates and frees almost 800.000 rows, one at a time, to hold intermediate results. In order to better understand why this is the case, let us look at the SaC compilation process.

4 SAC COMPILER

After scanning and parsing the source code, the SaC compiler transforms the code in several phases. The most relevant phase for the purpose of this paper is the phase that introduces an explicit notion of memory. It adds explicit memory allocations, pointers, and assignments, as well as static and dynamic memory management operations. Memory management in SaC is based on reference counting as this enables destructive updates of arrays whenever possible, providing a solution to the aggregate update problem[13]. A detailed description of the memory phase can be found in Trojahner [22].



(a) Runtime measurements for the rowadd functions

version	# allocations	total allocated	max allocated
rowadd_V1	3	6.25 GB	6.25 GB
rowadd_V2	3	6.25 GB	6.25 GB
rowadd_V3	800000	9.42 GB	6.25 GB

(b) Memory measurements for the rowadd functions

Figure 5: Rowadd measurement results

4.1 Array representation

One of the key choices of the SaC compiler infrastructure is to allocate all data of an array within a single chunk of memory, irrespective of the array's dimensionality or shape. This representation, often referred to as "flat array representation", keeps the overhead for dynamic memory management to a minimum and it guarantees constant offsets between elements and their neighbouring elements with respect to any dimension. The drawback of such a flat representation is that combining new arrays from existing arrays, such as creating matrices from vectors, requires copying the elements from the smaller arrays into a single, larger chunk of memory. While there are several optimisations that reorganise code in a way that higher-dimensional arrays are defined directly from scalars, these are not always applicable, at least not without introducing a considerable computational overhead (see e.g. Grelck et al. [9]). In the sequel, we describe how the current code generation process nevertheless in most cases can avoid copying upon nested array creation.

4.2 Values, Memory and MemVals

The memory phase makes memory explicit by introducing a notion of three different kinds of variables: `Val`, `Mem`, and `MemVal`. Variables of kind `Val` represent values in the classical functional programming sense, i.e., all variables before the memory phase can be seen as being of this kind. `Mem`-variables represent stateful memory locations not holding any value yet. Finally, variables of kind `MemVal` are referring to memory locations that hold a value. Ultimately, all values that need to exist at some point during runtime need to be transferred from variables of type `Val` into variables of type `MemVal`.

For the basic handling of `Val`, `Mem`, and `MemVal` variables the compiler provides a few built-in operations:

- `_alloc_` :: `Val` ⟶ `Mem`
  Allocates memory for an array of shape `Val`.
- `_fill_` :: `Val` × `Mem` ⟶ `MemVal`
  Fills a memory location with a value.
- `_copy_` :: `MemVal` ⟶ `Val`
  Copies the value of a section of filled memory.
- `_suballoc_` :: `Mem` × `MemVal` ⟶ `Mem`
  Indices `Mem` with index `MemVal` to obtain a subsection of `Mem`. The returned `Mem` value is treated in the same way as freshly allocated memory despite pointing into some pre-existing `Mem`. This constitutes the key operation to avoid copying when constructing arrays from smaller arrays.
- `_free_` :: `MemVal` ⟶ `Void`
  Frees previously filled memory.

Apart from these basic memory instructions, there are also some more complicated expressions introduced to reduce memory allocation and deallocation as much as possible:

- `_reuse_` :: `MemVal` ⟶ `Mem`
  Skips a memory deallocation and allocation step and transfers ownership of a piece of memory directly. Used for in-place updates.
- `_alloc_or_reuse_` :: `Val` × `MemVal`$^+$ ⟶ `Mem`
  Performs a `reuse` on one of the `MemVal` arguments if this is possible according to their reference counts. Performs an `alloc` of size `Val` otherwise. This is used if the reference counts are not statically determinable.
- `_wl_assign_` :: `Val` × `MemVal` × `Mem` ⟶ `Void`
  Specialised fill operation for with-loops. Indices `Mem` with `MemVal`, then fills result with `Val`.

```
a = [1,2,3,4];                     \\ Val
b = _neg_V_(a);                    \\ Val
```

(a) Before memory phase

```
a_mem = _alloc_([4]);              \\ Mem
a = _fill_([1,2,3,4], a_mem);      \\ MemVal
_inc_rc_(a, n);
b_mem = _alloc_or_reuse_([4], a);  \\ Mem
b = _fill_(_neg_V_(a), b_mem);     \\ MemVal
_inc_rc_(b,m);
_dec_rc_(a,1);
```

(b) After memory phase

Figure 6: Memory phase transformation example

An example of how the memory instructions are inserted is shown in figure 6. Figure 6a shows a small two line code snippet that negates all values of a vector using the built-in _neg_V_ operation. The transformed code is presented in figure 6b. Memory is allocated on line 1, before being filled in line 2 with the value `[1,2,3,4]`. Note here, that _alloc_ always initialises the reference count with 1. In line 3, the reference count of the `MemVal` `a` is increased by `n`, assuming that there are n more references to a later in the code. Line 4 tries to reuse the memory from a for b. This will only work if `n=0`, so in the situation where line 2 in figure 6a is the last reference to a. Otherwise fresh memory is allocated. Lines 6 and 7 then update the reference counts, again assuming m further references to b in the remainder of the code.

4.3 Handling memory for with-loops

Figure 7 shows an example of how memory management for genarray-with-loops is done. Memory for the result of a genarray-with-loop

```
1 a = with {
2     ([0,0] <= [i,j] < [5,10]) : i*10+j;
3   } : genarray( [5,10], 0);
```

(a) Before memory phase

```
1 a_mem = _alloc_([5,10]);
2 a = with {
3   ([0,0] <= [i,j] < [5,10]) {
4     val = i*10+j;
5     res = _wl_assign_(val,[i,j], a_mem);
6   }: res
7  } : genarray( [5,10], 0);
```

(b) After memory phase

Figure 7: Genarray-with-loop memory example

is introduced directly in front of the with-loop, on line 1 of figure 7b. It is then filled one element at a time using `_wl_assign_` on line 5. This works fine in cases where `_wl_assign_` takes scalar values as its first argument, but when with-loops are nested, the naive translation scheme leads to less efficient code. Figure 8 shows

```
1 a = with {
2     ([0] <= [i] < [10]) : with {
3         ([0] <= [j] < [5]): i*5 + j;
4         } : genarray( [5], 0);
5     } : genarray( [10], [0,0,0,0,0]);
```

(a) Before memory phase

```
1  a_mem = _alloc_( [10,5]);
2  a = with {
3    ([0] <= [i] < [10]) {
4      val_mem = _alloc_([5]);
5      val = with {
6        ([0] <= [j] < [5]) {
7          inner = i*5+j;
8          res = _wl_assign_( inner, [j], val_mem);}: res
9        } : genarray( [5], 0);}
10     outer_res = _wl_assign_( _copy_(val), [i], a_mem);
11     _free_(val_mem);
12     } : outer_res
13   } : genarray( [10], [0,0,0,0,0]);
```

(b) With naive memory instructions

```
1  a_mem = _alloc_( [10,5]);
2  a = with {
3    ([0] <= [i] < [10]) {
4      val_mem = _suballoc_(a_mem,[i]);
5      val = with {
6        ([0] <= [j] < [10]) {
7          inner = i*5+j;
8          res = _wl_assign_( inner, [j], val_mem);}: res
9        } : genarray( [5], 0);}
10   } : genarray( [10], [0,0,0,0,0]);
```

(c) After memory phase

Figure 8: Nested genarray-with-loop memory example

such an example. Here, we see in line 4 of figure 8b that for each

```
1 val = with {
2            ([0] <= [i] < [10]) : a[i];
3          } : fold( +, [0,0,0,0,0]);
```

(a) Before memory phase

```
1 accu_init_mem = _alloc_([5]);
2 accu = _fill_([0,0,0,0,0],accu_init_mem);
3 for (i=0; i < 10; i++) {
4      accu_mem = _alloc_or_reuse_([5], accu);
5      accu = _fill_(_add_VxV_(accu , a[i]), accu_mem);
6 }
```

(b) After memory phase

Figure 9: Fold-with-loop memory example

execution of the outer with-loop body a new 5-element vector is allocated which in line 10 is copied into the result matrix that was allocated in line 1. This is precisely the copying that results from a flat array representation as explained in section 4.1. This copying can be avoided by means of an in-place computation optimisation explained in Trojahner [22]. The key idea is to add the `_suballoc_` primitive which can replace the local call to `_alloc_` making sure that the result of the inner with-loop is directly placed into the correct position of the overall result, rendering the copying superfluous. The final code is shown in figure 8c. Here, we can see that not only the call to `_copy_` could be elided but the second call to `_wl_assign_` has disappeared completely. This is correct as the inner with-loop already assigns all the values to their final location in the memory allocated in line 1.

For fold-with-loops, memory management works differently. An example fold-with-loop is shown in figure 9. The size of the result of a fold-with-loop, in general, cannot be statically determined as the shape of the result depends on the folding operation and potentially even depends on the individual values to be folded. Consequently, there is no explicit pre-allocation of the result memory. Instead, we allocate an accumulator `accu` which is initialised with the neutral element of the folding operation (see line 2 in figure 9b). The compilation of the folding function (`_add_VxV_` on line 5) in turn is responsible to allocate the memory for the result. Once this memory is filled in line 5, the resulting `MemVal` either serves as accumulator for the next iteration of the folding operation or as the final result.

5 PROBLEM REVISITED

Using the compiler details introduced in section 4, the problem from section 3 can now be analysed in greater detail. To recap, the main problem is that programs defined with nested fold-with-loops produce significantly slower code than equivalent programs defined with genarray-with-loops. To better understand this, we now look at how the SaC compiler tries to optimise the three versions of rowadd introduced in figure 4. The first two versions also contain the extra addition of 0 as discussed in section 3 in the form of the `n` parameter, to make them more comparable with the fold version. The first version can be seen in figure 10.

The main change is the addition of explicit memory instructions, as there is little to optimise here. The second version in figure 11

```
int[800000,1000] rowadd_V1( int[800000,1000] a,
                            int[1000] n)
{
  res_mem = _alloc_([800000,1000]);
  res = with {
      ( [0,0] <= [i,j] < shape(a) - [3,0]) {
        elem_val = n[j] + a[i,j] + a[i+1,j]+ a[i+2,j];
        elem = _wl_assign_(elem_val,[i,j] res_mem);
      }: elem
    } : genarray([800000,1000],0);
return res;
}
```

Figure 10: Optimised version of rowadd_V1

now has a nested with-loop within the with-loop that was already there. This would normally create extra overhead because of the introduction of intermediate results and more memory management. However, as discussed in the previous section, the use of sub-allocation can prevent this overhead by calculating the inner with-loop in-place. This explains the similar performance seen in figure 5. Rowadd_V3 looks different however. The main difference between rowadd_V2 and rowadd_V3 is that the inner withloop at line 9 does not compute one element per iteration. Instead, the entire fold-loop, represented by lines 7 to 16 of figure 12, needs to be computed before the final values of the elements are known. Because the memory for the fold-with-loop is not allocated once before the loop, but once per iteration of the loop, applying in-place computation here is difficult. Every iteration computes an intermediate result, which needs memory. This means that replacing an `_alloc_` with a `_suballoc_` will not just affect the final allocation of the result, but also the allocation for all the intermediate results. This can result in problems when done naively, such as when the intermediate results are shared by other pieces of code, or when they have a different size as the final result. It is non-trivial to determine if doing the `_suballoc_` substitution here will result in problems or not. As such, the SaC compiler does not apply the in-place computation optimisation to nested fold-with-loops at all at the time of this research.

Because the in-place computation optimisation is not applicable to a nested fold-with-loop, `rowadd_V3` ends up allocating its own memory within the folding function (line 8 of figure 12). At the end of the fold-with-loop, on line 18, the contents of this memory need to copied over to the memory allocated by the outer with-loop. After this, the memory allocated by the inner with-loop needs to be freed. This overhead shown on lines 17 through 20 is exactly the overhead that is normally resolved by the in-place computation optimisation. This suggests that the performance of `rowadd_V3` can be improved by finding a way to make an in-place computation optimisation applicable similar to the existing one on genarray-with-loops.

6 IN-PLACE ACCUMULATOR OPTIMISATION

The main reason why the in-place computation optimisation is not applicable to fold-with-loops is because the source and sometimes

```
int[800000,1000] rowadd_V2( int[800000,1000] a, int[1000] n)
{
  res_mem = _alloc_([800000,1000]);
  res = with {
    ( [0] <= [i] < [800000-3]) {
      inner_mem = _suballoc_(res_mem,[i])
      inner_res = with {
        ( [0] <= [j] < [1000]) {
          elem_val = n[j] + a[i,j] + a[i+1,j]+ a[i+2,j];
          elem = _wl_assign_(elem_val,[j] inner_mem);
        }: elem
      } : genarray( [1000], 0);
    } : inner_res
  } : genarray([800000],n);
  return res;
}
```

Figure 11: Optimised version of rowadd_V2

```
1  int[800000,1000] rowadd_V3( int[800000,1000] a, int[1000] n)
2  {
3    res_mem = _alloc_([800000,1000]);
4    res = with {
5      ( [0] <= iv < [800000-3]) {
6        inner_res = n;
7        for ( j=0; j<3; j++) {
8          inner_mem = _alloc_or_reuse_([1000], inner_res);
9          inner_res = with {
10           ( [0] <= k < [1000]) {
11             fold_iter_mem = _suballoc_(inner_mem,[k]);
12             fold_iter_val = inner_res[k] + a[iv+j,k];
13             fold_iter = _fill_(fold_iter_val, fold_iter_mem);
14           }: fold_iter
15          } : genarray( [1000], 0);
16       }
17       inner_res_mem = _suballoc_(res_mem,iv);
18       inner_res_copy = _fill_(_copy_(inner_res),
19                               inner_res_mem);
20       _free_(inner_res);
21       } : inner_res_copy;
22     } : genarray([800000],n);
23   return res;
24 }
```

Figure 12: Optimised version of rowadd_V3

even the size of their memory is not statically decidable. For example, folding a list with a filter operation has a result size that depends on the value of the elements being filtered. Folding with a max function would return some existing piece of memory, but it is not possible to predict which memory without first calculating all the elements and checking which one is the biggest. There is, however, one common source of the result memory of a fold-with-loop that does have potential to benefit from the suballoc system. This occurs when the fold-with-loop is trying to reuse the memory of its internal accumulator for the result. Whether or not this happens depends on what operation is used to do the actual folding. Many folding operations do some kind of reduction where they combine

```
1  res = with { ( lb_i <= iv_i <  ub_i) {
2      accumulator = _accu_(iv_i);
3      accu_mem = _alloc_or_reuse_(..., accumulator);
4      . . .
5      expr = . . .
6      iteration_res = fill_like_operation(expr, accu_mem);
7      . . .
8      } :iteration_res
9    }: fold( fold_op,  neutral_el)
```

(a) Code pattern before in-place accumulator optimisation.

```
1  fold_mem = _alloc_(shape(neutral_el));
2  res = with { ( lb_i <= iv_i <  ub_i) {
3      accumulator = _accu_(iv_i);
4      accu_mem = _mem_reuse_(fold_mem);
5      . . .
6      expr = . . .
7      iteration_res = fill_like_operation(expr, accu_mem);
8      . . .
9      } :iteration_res
10   }: fold( fold_op,  neutral_el)
```

(b) Code pattern after in-place accumulator optimisation.

Figure 13: In-place accumulator optimisation. Code before optimisation (top) and equivalent code after optimisation (bottom).

a value with an accumulator of a fixed size to create a new value for the accumulator which has the same size. These kinds of operations would benefit from being computed in-place, especially when they are performed on large arrays.

The actual optimisation consists of three main steps: First, we check if the optimisation is applicable to a given fold-with-loop (Section 6.1). If it is, we then allocate memory for the accumulator outside of the fold-loop and replace the attempt of reusing the old accumulator with an explicit reuse of this new memory (Section 6.2). Finally, we replace the alloc statement for the memory of the accumulator with a suballoc statement and delete the no longer needed copy instructions at the end of the fold-with-loop (Section 6.3).

6.1 Applicability

To check if the optimisation is applicable, a program traversal is done looking for the pattern in figure 13a. This figure shows a simplified version of the code pattern as it looks like during the memory phase of the SaC compiler. Because of the functional semantics of SaC, each variable has exactly one definition during the memory phase of the compilation process. The main purpose of this step is tracing where the memory of the accumulator of the fold-with-loop is coming from, and checking if this is the same memory used by the previous accumulator.

Fold-with-loops always start by creating a variable for the accumulator using the `_accu_` primitive. This will either contain the result of the previous iteration, or the neutral element if it is the first iteration. If the with-loop tries to reuse the memory of this accumulator variable for the result, then the optimisation is applicable. In this case there will be an `_alloc_or_reuse_` statement on the accumulator variable. Additionally, the result of this statement has to used as the memory for the result of the whole with-loop, which is the expression at line 8 of figure 13a. In the pattern, the memory of the accumulator is stored in `accu_mem`. Line 5 is an abstraction of the actual calculation the with-loop is doing. Line 6 represents the last assignment of this calculation to memory. It stores the result, represented by `expr`, in the `iteration_res` variable, using `accu_mem` as the needed memory. The `fill_like_operation` here is a placeholder for the construct that is used to fill `accu_mem` with the value of `expr`. Currently this can either be a fill primitive or another with-loop. After this line, `iteration_res` is using `accu_mem` as its memory, which is the same memory used by the accumulator. In line 8 the `iteration_res` variable, is passed on as the result of the loop. This confirms that the next accumulator will use the same memory as the old accumulator. Since this fold-loop is therefore using the same memory for its accumulator in each iteration, this memory is guaranteed to contain the final result. By using a `suballoc` instead of a regular `alloc` on the place where this memory is allocated, the entire calculation can be done in-place. The next step of the optimisation will make this place explicit.

6.2 Explicit accumulator memory

After having determined that a fold-with-loop fulfils all the required conditions, the actual optimisation can be applied. Finding the original `alloc` statement that allocates memory for the result is difficult. And even if it is found, it might not be possible to do anything with it because other sections of code are also able to access the value in that memory. The solution is to allocate an entirely new section of memory, which is then guaranteed to not be used by anything else. This will bring the fold-with-loop in line with the genarray-with-loop, which also allocates a fresh section of memory before the start of the loop. The result of this transformation can be seen in figure 13b.

A fold-with-loop needs to both read from and write to its accumulator. It does this by using two variables at once. The `accumulator` variable is used for reading, and the `accu_mem` variable is then used to write the next accumulator. During the first iteration, the source for `accu_mem` was unclear. Maybe the `_alloc_or_reuse_` succeeded and it was the memory of the neutral element. Or, if the reuse failed, it was fresh memory. For further iterations, the source is the previous accumulator. The desired behaviour is therefore to allocate fresh memory during the first iteration, but to perform a guaranteed reuse in all other iterations. This resolves all ambiguity about where the memory comes from, and allows other optimisations to more easily interact with it. To achieve this, a new memory primitive is introduced on line 4 of figure 13b:

$$\_mem\_reuse\_ :: Mem \rightarrow Mem$$

The semantics of `_mem_reuse_` are the same as that of `_reuse_`, only the type is different. On the first iteration of figure 13b, the freshly allocated memory `fold_mem` will be reused, which will do nothing because it is still uninitialised. On subsequent iterations, it will contain the result of the previous iteration, and a regular reuse operation will be performed.

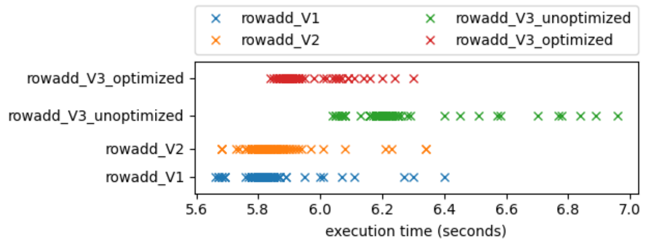
Allocating fresh memory instead of reusing what was already allocated introduces extra overhead in two ways:

(1) The actual allocation and de-allocation of memory takes a small amount of time.
(2) The resulting memory has to be initialised. In this situation this means making a copy of the neutral element.

However, in this specific scenario, both of these can be prevented from happening. The first point gets completely eliminated by the in-place computation phase later. The `alloc` that is introduced here is guaranteed to be replaced with a `suballoc`, and `suballoc` does not allocate fresh memory. Instead, it simply returns a pointer to previously allocated memory. In fact, the resulting code gets faster, because the allocation done by the `alloc_or_reuse` statement on line 3 of figure 13a, which was not compatible with suballoc, is replaced by an allocation that is. Additionally, since the old allocation is prevented, the statement that frees that memory also gets removed. This is because memory allocated with suballoc does not need to be individually freed. It will be freed when the memory that is sub-allocated into is freed all at once. This means that introducing this fresh allocation here will actually reduce the total amount of memory allocations and de-allocations by one each.

The second cause of overhead can also be avoided here, because of the distinction between variables representing abstract values, empty memory and filled memory, as discussed in section 4.2. They are respectively referred to as having type `Val`, `Mem` and `MemVal`. `Val` variables are not located in memory, and therefore cannot exist at runtime. The `Mem` and `MemVal` variables are introduced by the compiler to deal with this, but they are more restricted in how they can be used. Since `Mem` variables represent empty memory, they are never read from. Similarly, since `MemVal` variables represent filled memory and variables can only have one definition, they are never written to. A consequence of this is that when a `Memval` variable gets replaced with fresh memory, it needs to be initialised to maintain the same semantics. However, when replacing a `Mem` variable with fresh memory, this is not required because all memory variables are uninitialised to begin with. This explicit read/write distinction allows for the core of the optimisation: merging the copy operation required to maintain safe functional semantics with the computation itself to remove overhead. Instead of first making a copy in fresh memory, and then doing the whole computation in this new memory, the first iteration reads from the old memory and writes to the fresh memory. Every subsequent iteration then reads and writes from the fresh memory. This makes an effective copy without the associated overhead.

The overwriting of `accu_mem` in subsequent iterations is safe, as indicated by the presence of the `_alloc_or_reuse_` primitive. It signifies that it has been statically determined that the accumulator can safely be reused here, except for one condition: the reference count is unknown. By allocating fresh memory, the reference count of that memory becomes known, and safe reuse can be statically guaranteed. This short explanation skipped over a lot of details about the memory management system which are irrelevant for the in-place accumulator optimisation. A more thorough explanation about the details of memory variables and how they can and cannot be used can be found in Trojahner [22].



(a) Runtime measurements for the rowadd functions

version	# alloc's	total allocated	max allocated
V1	3	6.25 GB	6.25 GB
V2	3	6.25 GB	6.25 GB
V3 unoptimized	800000	9.42 GB	6.25 GB
V3 optimized	3	6.25 GB	6.25 GB

(b) Memory measurements for the rowadd functions

Figure 14: Testing results for the rowadd functions, including optimised V3

6.3 In-place computations

The final step of the optimisation is to put the initial idea into action, which is to reuse the existing implementation of the in-place computation optimisation in the fold context. It turns out that it suffices to annotate the affected fold-with-loops with where their memory is allocated. Since the fold-with-loops that were affected by the in-place accumulator optimisation now also have their memory allocation in front of the loop, the existing system can be reused.

7 RESULTS

Theoretically, after applying the in-place accumulator optimisation, the performance of `rowadd_V3` should be more similar to that of the other two versions. To test this, we repeat the experiment from figure 5 for `rowadd_V3` using the new in-place accumulator optimisation. The results of this test, added to the previously gathered results, can be seen in figure 14. The extra memory allocations are completely eliminated by the in-place accumulator optimisation. This caused the mean execution time to shift from 6.3 seconds to 6.0 seconds. It is still not as efficient as the other two versions, but it is now only 3% slower, instead of 10%. The performance gap is smaller now, moving from 0.5 seconds to 0.2 seconds, which means that roughly 60% of the fold overhead is resolved. The remaining performance difference can be explained by the fact that there is still a nesting of with-loops, which will always have some loop overhead compared to running just a single with-loop. The in-place accumulator optimisation seems like a step in the right direction however.

Throughout the paper the fixed example of rowadd was used to explain the problem of fold overhead. This example is an instance of a typical pattern where the fold performance hit comes into play. The in-place accumulator optimisation works well on this example, but we also want to know if it works well in general. To investigate this, we look at variants of the problem, where the two array dimensions and the amount of work being done is varied. We

```
1  use Array: all;
2  use StdIO: all;
3  use Benchmarking: all;
4  \\ Id functions to prevent the compiler from
5  \\ calculating the entire program as a
6  \\ constant at compile time.
7  noinline int[INNER] id( int[INNER] a)
8  { return a; }
9
10 noinline int[OUTER,INNER] id( int[OUTER,INNER] a)
11 { return a;}
12
13 int main()
14 {   \\ Calculate initial elements. Hide
15     \\ details behind non-inlined functions
16     \\ so they are not seen as constants.
17     zeroes = id(genarray([INNER], 0));
18     a = id( genarray([OUTER,INNER], 1));
19     \\ Start benchmarking time.
20     i1 = getInterval( "vect", 0);
21     start( i1);
22     \\ Calculate a single update step
23     updated_a = with {
24         ( . <= iv < [OUTER-N]) {
25            \\ Calculate c = a[iv] + a[iv+1] + ... + a[iv+N].
26            c = with {
27               (iv <= jv < iv+N) : a[jv];
28                } : fold( +, zeroes);
29            } : c;
30       } : genarray( [OUTER], zeroes);
31     \\ Stop benchmarking time.
32     end( i1);
33     \\ Print part of result to make sure the
34     \\ calculation does not get optimised away.
35     print(updated_a[1,2]);
36     \\ Print benchmarking results
37     printResult( i1);
38     t,u = returnResultUnit( i1);
39     printf( "GFLOPS per %s: %f \n",
40         u, tod((OUTER-N)*INNER*N)/(1000000000.0*t));
41     return 0;
42 }
```

Figure 15: Benchmark which calculates an update step

try to systematically evaluate what the impact of the optimisation is across these three axis. With this, we get a better idea of what the impact of the optimisation is on more general programs.

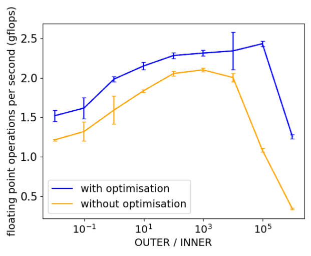
The program shown in figure 15 is used as the basis of these tests. It is a generalised version of the example shown in figure 4c. The program has three parameters labelled `INNER`, `OUTER` and `N`. These parameters are set using macros. This variation, in combination with several compiler options, gives a set of benchmark tests. The program itself calculates an abstract version of an update step, as found in several algorithms. It starts with the array `a`, and then calculates `updated_a` from that. This happens in line 24 to line 32. Most of the rest of the program is boilerplate to prevent the constant propagation optimisation to replace the entire calculation with the result at compile time. The compiler can normally do this, because the input is already fully specified at compile time. The benchmarking code itself is also given explicitly. It uses the benchmarking library to keep track of time during the actual calculation, which is defined as the code between the `start(i1)` and `end(i1)` lines. The benchmarking output is given in giga floating point operations per second, or gflops for short.

The main calculations done by the program can be summarised as follows:

$$updated\_a[i] = \sum_{n=0}^{N} a[i+n]$$

This is done for every valid value of `i`, which ranges from 0 to `OUTER-N`. The summation itself computes `N` additions. The values that are being added are arrays themselves, because `a` is two-dimensional and it is being indexed with a scalar value. This results in an array of shape `[INNER]`. Adding two such arrays together comes down to adding them index-wise, which is `INNER` floating point operations. Therefore the entire program computes `(OUTER-N)*N*INNER` floating point operations.

Figure 16 shows the results of running the program with or without the optimisation using the inputs seen in figure 16b. To account for the effects of varying system load on performance, each test was run 20 times and the average has been plotted in figure 16a. Bars are shown at each measuring point to signify the standard deviation measured. Unless stated otherwise, all results in this section use `N=2` and the values of `INNER` and `OUTER` as seen in figure 16a. The x-axis shows the value of `OUTER` divided by `INNER`, in order to show the changes of both variables on a single axis. The values for `OUTER` and `INNER` are chosen in such a way that the size of the resulting array is the same for each combination. What changes is the relation between how big the outer dimension is and how big the inner dimension is. The figure shows that the optimisation gives a reasonably consistent improvement for all values of `OUTER` and `INNER`. This matches our expectation, as the optimisation offers a scaling improvement for both variables. If `OUTER` is big, then the inner fold-with-loop needs to be calculated many times. For each computation of the inner with-loop, a memory allocation, de-allocation and copy operation are optimised away. This will therefore give more visible results if more iterations are executed. The other way around also holds. All the previously mentioned operations that are optimised away are normally executed over an array of shape `[INNER]`. Especially for the copy operation, this means that if `INNER` is big, the operation takes more time. Because of this the in-place accumulator optimisation will also give better results if `INNER` gets bigger. From the figure it seems like the extra performance gain for large `OUTER` is more significant than the one for large `INNER`, because the difference in performance gets bigger on the right side of the graph. The sudden dip in overall performance, both with and without the optimisation, on the right side of the graph also indicates that performing many small operations (large `OUTER`, small `INNER`) is less efficient in general than executing less big operations (small `OUTER`, large `INNER`). This makes sense, because every with-loop adds some overhead in setting up breaking down. The more time is spent entering and leaving with-loops, the less time is spent actually calculating the result, which decreases the floating point operations per second.



(a) Execution results for N=2

INNER	100000	32000	10000	3200	1000	320	100	32	10
OUTER	1000	3125	10000	31250	100000	312500	1000000	3125000	10000000

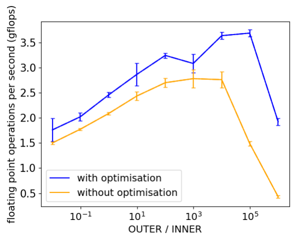
(b) Values for `INNER` and `OUTER`.

Figure 16: Comparison of code performance with and without the optimisation.

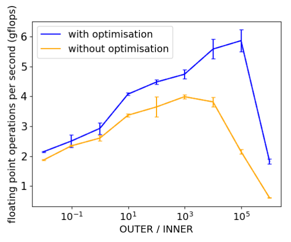
The effect of the loop overhead can be reduced by increasing the amount of work a single loop iteration does. The workload of the inner fold-with-loop can be increased by increasing the value for `N`. This means that it will add more rows together, by doing more iterations. This will then increase the workload for a single iteration of the outer with-loop. The results of this can be seen in figure 17. Because the total amount of floating point operations has gone up, while the amount of loop iterations stayed the same, the effect of the loop overhead relative to the time spend performing calculations has gone down. This results in a higher amount of floating point operations per second, which increases further as `N` goes up. Another notable change is that the performance gain on the left side of the figures goes down, eventually disappearing entirely. The performance gain on the right side of the figures seems to be unaffected however. This is because even with a bigger value for `N`, if the value for `INNER` is very small, the total workload of the inner fold-with-loop (`N*INNER`) is still small. A smaller workload for the inner with-loop means the effects of the loop overhead are more distinct. This in turn means the effects of reducing the loop overhead are more visible.

To get an idea of what these numbers mean and if they are good or not, a baseline is required. This can be obtained by modifying the code of line 27 of figure 15 as described in figure 18.

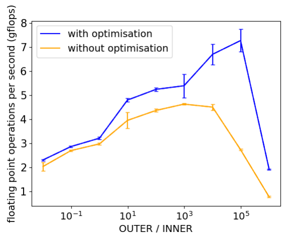
This change does not change the semantics, but it does allow the compiler to unroll the with-loop because the index range of `jv` is now just depending on constants, and no longer on `iv`. After this change, the compiler sees that the fold-with-loop is only calculating a total of `N` iterations. For small values of `N` (by default 9) the compiler will decide that it is more efficient to unroll the with-loop. This means that instead of compiling into code with a loop like described in figure 12, the loop gets optimised away. This can be done by copying the body of the loop one time for each iteration, resulting in a sequential program with code duplication. However, since the fold-loop is now gone, the reason why other optimisations such as with-loop scalarization or in-place computation were not applicable is now gone. In practice this means that the unrolled inner with-loop gets merged with the outer with-loop, removing all the overhead caused by the inner with-loop in the process. Since



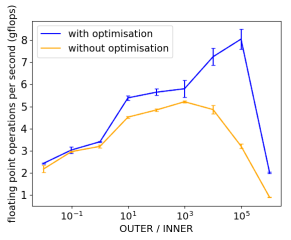
(a) Execution results for N=3



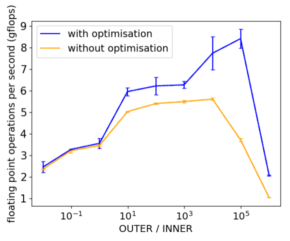
(b) Execution results for N=5



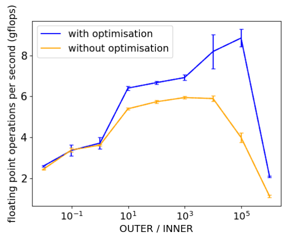
(c) Execution results for N=7



(d) Execution results for N=9



(e) Execution results for N=12



(f) Execution results for N=15

Figure 17: More comparison results for bigger values of `N`

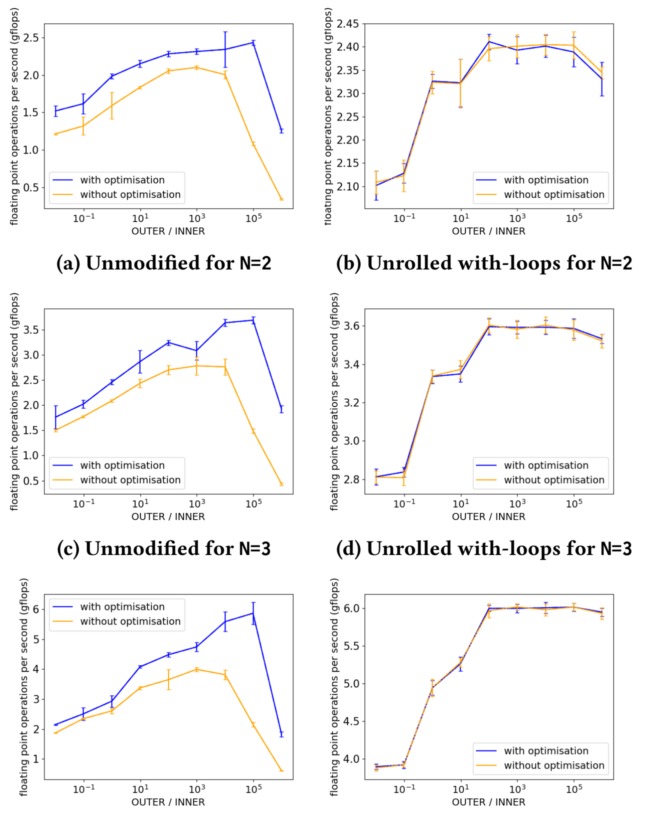
```
(iv <= jv < iv+N) : a[jv];

        ⇓

(0 <= jv < N) : a[iv+jv];
```

Figure 18: Modification to fold-with-loop that will allow with-loop unrolling

the main goal of the in-place accumulator optimisation is to reduce this overhead, removing it entirely is the best achievable result. The results of this experiment can be seen in figure 19. The right side of the figure contains the results obtained by the modification in figure 18, which optimises the inner fold-with-loop away. The left side of the figure contains the equivalent results without this modification. As expected, with the change to the inner fold-with-loop the optimised and non optimised code are performing equivalently in this case. If there is no inner with-loop, then the in-place accumulator optimisation has nothing to optimise. When comparing right and left graphs, it also becomes visible that in roughly the middle of the graphs, the performance achieved by the in-place accumulator optimisation is approaching the same performance as when the with-loop is fully optimised away. On the edges of the graphs, there is a larger gap in performance.

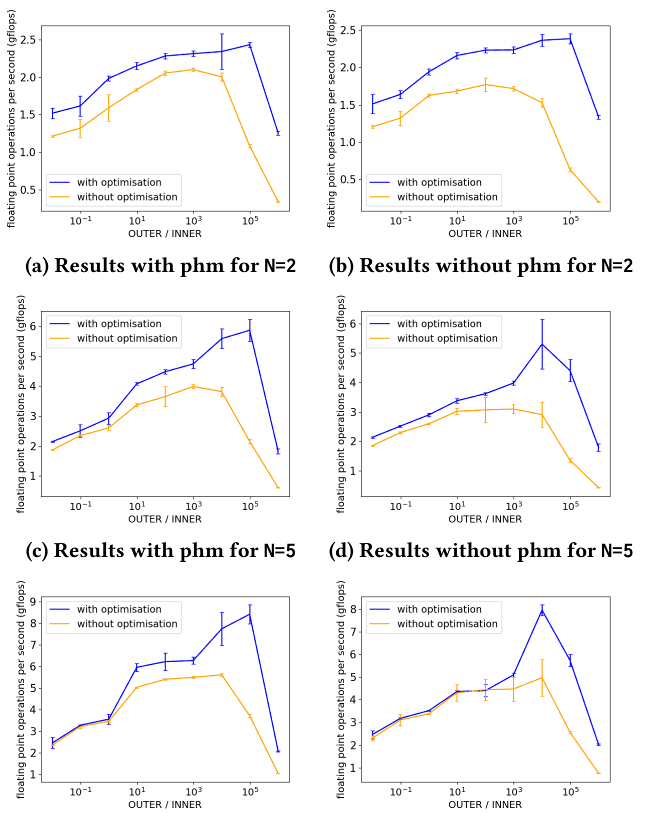


(a) Unmodified for N=2 **(b) Unrolled with-loops for N=2**

(c) Unmodified for N=3 **(d) Unrolled with-loops for N=3**

(e) Unmodified for N=5 **(f) Unrolled with-loops for N=5**

Figure 19: Comparison of results with and without the modification described in figure 18



(a) Results with phm for N=2 **(b) Results without phm for N=2**

(c) Results with phm for N=5 **(d) Results without phm for N=5**

(e) Results with phm for N=12 **(f) Results without phm forN=12**

Figure 20: Comparison of results with and without the phm

Since the optimisation aims to reduce overhead caused primarily by memory management, an interesting experiment is to look at the results when using different memory management systems. SaC has its own memory management system called the private heap manager (phm). This can be disabled to use the system default, which uses `malloc` and `free` as defined by the local c compiler. The phm is not available on macOS, so both the behaviour with and without the phm is relevant. All other results in this section are obtained by compiling with the phm, unless explicitly stated otherwise. The previously shown results are shown next to results of the same experiments with the phm off in figure 20. The general performance of the program without the phm is lower that when it is enabled. This makes sense, because the phm is specifically tailored for SaC. It therefore performs better than the more general system default memory manager does. For low values of `N`, the optimisation gives a bigger improvement without the memory manager than with it. If `N` is low, there is relatively more loop overhead. The phm can reduce this loop overhead by streamlining memory allocation and de-allocation. Without the phm, the effect of memory (de)allocations is higher, so when the in-place accumulator optimisation removes some of them, this has a greater effect. In general the choice for heap manager does not really matter based on the values for `INNER` and `OUTER`. In some specific cases there are differences. These seem to be an artefact of the way the heap managers allocate arrays of different sizes.

8 RELATED WORK

While this paper focuses on the specific problems caused by nesting fold-with-loops within the SaC programming language, similar problems also exist in other languages. The idea of compiling a high level functional language to high performance, system specific code is not limited to just SaC. Other projects such as Futhark[11], Lift[19], Rise[20], Accelerate[4], Sisal[8], SkePU[7], Marrow[18], Halide[15], etc follow a similar design philosophy, and therefore might run into similar memory problems. These systems also need to deal with memory for nested computations in an efficient manner. Details on the exact memory strategies used by these languages are often not publicly available. In addition, memory focus often lies on effective use of device memory (GPU memory). This paper focuses on system memory (main memory). The proposed optimisation might be transferable, but this needs further research. If the internal memory representation of arrays is not flat, but a nested structure using pointers, than in-place computation is not required, as intermediate results do not need to be moved around. However, flat memory representations have several advantages, and are therefore more likely to be in use.

The Futhark language[11, 12] has many similarities with SaC. They both are array languages with functional semantics and a focus on having the same code be compiled efficiently for different systems. Futhark also runs into the problem that code with nested constructs is often easy to write, but not as efficient to execute. While this paper aims to reduce this problem by reducing the overhead caused by nesting, the Futhark compiler focuses on removing nesting by various types of flattening, for instance Blellochs algorithm [3]. This approach has been refined over time, but is not yet as fast as hand optimised code [2, 6]. It should be possible to carry the in-place computation approach over to the Futhark setting, in particular as both Futhark and SaC try to statically infer uniqueness of references to facilitate in-place-updates. Both languages allow for explicit uniqueness annotations and have an aliasing analysis under the hood for uniqueness inference. While Futhark, to our knowledge, does not support in-place-updates in statically undecidable situations, this capability of SaC is irrelevant for the optimisation proposed here as the optimisation relies on a statically inferred reuse-guarantee. However, the research in the context of Futhark seems to primarily focus on parallel execution on GPUs, and our proposed optimisation sofar is only applicable in a single threaded context.

Another high level functional language focusing on high performance parallel computation is Lift. In addition to compiler optimisations, Lift also limits the expressiveness of some constructs. For instance, arbitrary array indexation is not possible. This means that arrays can only be accessed through certain constructs such as map or reduce. By limiting the number of constructs that can access arrays, it becomes easier to reason about data sharing. An early publication on Lift stated that no memory reuse was being done [19]. A later publication talks about memory reuse, but does not give an implementation[21]. This same publication does state that Lift also runs into the problem of overhead caused by generating intermediate results. The proposed solution is to fuse operations together. There is no description of what happens when this fails or is inefficient, which is where in-place computations might help, depending on the used memory layout.

The Shine compiler for the Rise language is based on Lift, but uses multiple intermediate representations during compilation to more effectively specialise for certain compilation phases. There is no support for memory reuse, but the generated code is otherwise very similar to that of Lift. The benchmark comparison done with Lift therefore shows the effects of memory reuse on performance[20].

Another approach to obtain a high level language for parallel constructs with high performance is to use domain specific languages. Examples of this include Accelerate[4], SkePU[7], Marrow[18] and Halide[15]. These work by embedding inside a general purpose language such as Haskell or C++. This means that they often do not do their own memory management, as this is handled by the host language. The same problems with intermediate results caused by nesting also appear here. However, and the chosen solution seem to focus on flatting[5, 14]. Marrow explicitly focuses on allowing the nesting of parallel constructs. In addition, flattening too much makes implementing compiler optimisations harder, which can result in less efficient code. When flattening fails or is otherwise undesired, and the language has direct control over its memory management, in-place computations could help improve performance. Halide stands out here because it explicitly decouples what an algorithm computes from how/where it is executed. This allows manual control over the location of intermediate results. Work has also been done on using machine learning to automate finding the most optimal execution strategy[1]. This approach can probably not be combined with defining specific compiler optimisations like the one proposed in this paper, but is likely to tackle a similar problem.

Other languages like the Sisal[8] take an approach similar to SaC. They use reference counting to focus on in-place updating wherever possible, in addition to fusing loops. In such a setting the in-place accumulator optimisation should be directly applicable.

9 CONCLUSION

This paper investigates why SaC code using fold-with-loops on hyper-planes with non-scalar fold results performs worse than other specifications for computing the same results. As a running example, we use three versions of `rowadd` as introduced in section 3. These do not have the same performance, while they do compute the same result. This violates the design philosophy of SaC, and forces programmers to think about implementation details again if they want efficient code. From an analysis of the way the three versions of rowadd are compiled we see that one of the main causes for the performance discrepancy is a lack of memory reuse in nested fold-with-loops. Several other optimisations, such as with-loop scalarization and in-place computations, are not applicable to nested fold-with-loops. This leads to less efficient memory management and extra copies in the generated code. Rewriting all these optimisations to also work for fold-with-loops is difficult at best and impossible at worst. Instead, this paper takes the approach to change the code generation for fold-with-loops making them more similar to genarray-with-loops.

In general, fold-with-loops cannot predict where the final result will be allocated. Yet this information is required for the existing memory reuse optimisations. The key insight of this paper is that if a fold-with-loop can be computed in-place, then an upfront allocation becomes possible. The in-place accumulator optimisation introduced in section 6 is designed to identify these situations and to leverage this information to bring the memory management of fold-with-loops in-line with the memory management of genarray-with-loops. This allows the existing in-place computation optimisation to also work for these nested fold-with-loops. The source code for the in-place accumulator optimisation is currently available online [23], and is planned to be part of the next major SaC release.

From the performance evaluation in section 7, we see that the in-place accumulator optimisation significantly improves the performance of `rowadd_V3`. The performance gap between the `rowadd` functions is reduced by 60%. More general testing shows an improvement when the inner fold-with-loop is executed a lot, because every iteration is slightly faster, so more iterations compound the effect. Similarly, there is also an improvement when the result of the inner fold-with-loop is a large array, because a copy operation on this array is prevented. The iteration based improvement is notably bigger than the size based improvement. Both of these improvements are more distinct when the fold-with-loop that is being optimised has a low workload. If this workload gets bigger, by increasing the amount of calculations done in the body of the loop, the effects of reducing the loop overhead become less noticeable. However, especially if the with-loop is executed a lot, there is still a noticeable improvement. None of the benchmark tests show a loss in performance while using the in-place accumulator optimisation. The worst observed performance is still as good as the performance without the optimisation.

Currently, the proposed optimisation is only applicable when the fold-with-loop is executed sequentially. It does not interfere with possible code vectorisation, but it does interfere with multi-threaded executions, be it on multi-core systems, GPUs or clusters. For that reason, the current implementation is only enabled when generating code for single-threaded execution. An extension in this regard is left for future work.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful suggestions.

REFERENCES

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (July 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

[2] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. 2013. Data-Only Flattening for Nested Data Parallelism. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/2442516.2442525

[3] Guy E. Blelloch and Gary W. Sabot. 1990. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel and Distrib. Comput.* 8, 2 (1990), 119–134. https://doi.org/10.1016/0743-7315(90)90087-6

[4] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (Austin, Texas, USA) *(DAMP '11)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1926354.1926358

[5] Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. 2017. Streaming Irregular Arrays. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (Oxford, UK) *(Haskell 2017)*. Association for Computing Machinery, New York, NY, USA, 174–185. https://doi.org/10.1145/3122955.3122971

[6] Martin Elsman, Troels Henriksen, and Niels Gustav Westphal Serup. 2019. Data-Parallel Flattening by Expansion. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Phoenix, AZ, USA) *(ARRAY 2019)*. Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/3315454.3329955

[7] A. Ernstsson, J. Ahlqvist, S. Zouzoula, and C. Kessler. 2021. SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters. *International Journal of Parallel Programming* (2021). https://doi.org/10.1007/s10766-021-00704-3

[8] Jean-Luc Gaudiot, Tom DeBoni, John Feo, Wim Böhm, Walid Najjar, and Patrick Miller. 2001. *The Sisal Project: Real World Functional Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 45–72. https://doi.org/10.1007/3-540-45403-9_2

[9] Clemens Grelck, Sven-Bodo Scholz, and Kai Trojahner. 2005. With-Loop Scalarization – Merging Nested Array Operations. In *Implementation of Functional Languages*, Phil Trinder, Greg J. Michaelson, and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 118–134. https://doi.org/10.1007/978-3-540-27861-0_8

[10] Clemens Grelck and Kai Trojahner. 2004. Implicit Memory Management for Sac. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, Clemens Grelck and Frank Huch (Eds.), Vol. 4. University of Kiel, Institute of Computer Science and Applied Mathematics, 335–348. https://www.sac-home.org/_media/publications:pdf:greltrojifl04.pdf Technical Report 0408.

[11] Troels Henriksen. 2017. *Design and Implementation of the Futhark Programming Language*. Ph.D. Dissertation. Department of Computer Science, Faculty of Science, University of Copenhagen.

[12] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. 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* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[13] Paul Hudak and Adrienne Bloss. 1985. The Aggregate Update Problem in Functional Programming Systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '85)*. Association for Computing Machinery, New York, NY, USA, 300–314. https://doi.org/10.1145/318593.318660

[14] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. *Optimising Purely Functional GPU Programs*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2500365.2500595

[15] Jonathan Ragan-Kelley. 2014. *Decoupling algorithms from the organization of computation for high performance image processing*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. http://hdl.handle.net/1721.1/89996

[16] Sven-Bodo Scholz. 2003. Single Assignment C: Efficient Support for High-Level Array Operations in a Functional Setting. *J. Funct. Program.* 13, 6 (Nov. 2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[17] Sven-Bodo Scholz, Stephan Herhut, Frank Penczek, Clemens Grelck, Artem Shinkarov, and Hans-Nikolai Viessmann. 2021. Single assignment C tutorial. https://sac-home.org/_media/docs:tutorial.pdf.

[18] Fábio Soldado, Fernando Alexandre, and Hervé Paulino. 2016. Execution of compound multi-kernel OpenCL computations in multi-CPU/multi-GPU environments. *Concurrency and Computation: Practice and Experience* 28, 3 (2016), 768–787. https://doi.org/10.1002/cpe.3612 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3612

[19] Michel Steuwer. 2015. *Improving programmability and performance portability on many-core processors*. Ph.D. Dissertation. University of Münster. https://www.lift-project.org/publications/2015/steuwer15phdthesis.pdf

[20] Michel Steuwer, Thomas Koehler, Bastian Köpcke, and Federico Pizzuti. 2022. RISE & Shine: Language-Oriented Compiler Design. *arXiv:2201.03611 [cs]* (Jan. 2022), 12 pages. http://arxiv.org/abs/2201.03611

[21] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 74–85. https://doi.org/10.1109/CGO.2017.7863730

[22] K. Trojahner. 2005. *Implicit Memory Management for a Functional Array Processing Language*. Master's thesis. Universität zu Lubeck. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.408.8837&rep=rep1&type=pdf

[23] Gijs van Cuyck. 2021. SaC in-place accumulator optimisation. https://gitlab.sac-home.org/gvcuyck/sac2c/-/blob/develop/src/libsac2c/memory/fold_in_place_accumulator.c.

[24] Artjoms Šinkarovs, Hans Viessmann, and Sven-Bodo Scholz. 2021. Array Languages Make Neural Networks Fast. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Virtual,Canada) *(ARRAY 2021)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3315454.3464312