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 any 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. 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.

ACM Reference Format:
Gijs van Cuyck and Sven-Bodo Scholz. 2021. In-Place-Folding of Non-Scalar Hyper-Planes of Multi-Dimensional Arrays. In *Proceedings of IFL 21: ACM Symposium on implementation and application of functional languages (IFL 21).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

High-Level array languages such as Futhark[13], Accelerate[5], Lift[20], Halide[17], or SaC[18] 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 patterns that are not yet covered well enough to allow the programmer to solely concentrate on the "what".

This paper focuses on one particular constellation which appears frequently in applications that naturally lend themselves to algorithmic descriptions on higher-dimensional arrays (dimensionality $\gg$ 3) such as for example CNNs (Convolutional Neural Networks)[25]. More specifically, it deals with fold-operations that fold away one or more dimensions of a higher-dimensional array that comprise neither the innermost nor the outermost one. 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. Given the folding arguments are entire sub-arrays this poses a challenge. One way to avoid this problem is to re-write the computation into a semantically equivalent one that performs the folding-operation on the innermost dimension(s). This re-write can even be automated by using optimisations such as *With-Loop-Scalarization* [11]. Unfortunately, such a re-write (i) is not always possible and (ii) it is not always desirable as it may inhibit the use of vector operations.

In this paper, we propose a code generation technique that allows such reductions on entire 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 [12]. 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 relevant features, section 3 provides a detailed account of the problem the new code generation technique 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

TBD...

IFL 21, September 01–03, 2021, Online

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

3 PROBLEM STATEMENT

As stated previously, SaC is a declarative language. This means that it should not matter how a programmer writes down a program, in terms of efficiency. As stated earlier in section 1 however, the performance of nested fold withloops often lags behind equivalent programs without fold loops. Using the SaC_{mini} formalism introduced in **??**, this performance discrepancy can be exposed. An example of how the same program can be written down in different ways can be seen in figure 1. The figure shows three ways in which an existing 2-dimensional array can be updated by adding several rows together. Each of the three programs calculates the same result *res* from input *a*, with the following property: $res[i,j] = a[i,j] + a[i+1,j] + a[i+2,j]$. For the last three elements of *res* where this calculation would fall outside the bounds of *a*, the result is 0 instead. Figure 1a takes the straightforward approach and just adds the individual elements together. Figure 1b Uses a more array based approach and adds entire rows at a time. Both of these options work fine for smaller cases, but if for instance a hundred rows need to be added the code becomes hard to read and maintain. The third option shown in figure 1c does not have this problem. It uses a fold withloop to calculate the same result as the first two options. The +3 on line 5 shows that currently three rows are being added, just like in the other options. However, to increase the number of rows here to a hundred, the +3 can simply be replaced by +100, as long as the bound of the outer withloop ($800000 - 3$) is also adjusted accordingly. For the first two options, this change would require adding more rows to the explicit addition. For adding a small number of rows though, one of the first two options might be preferable because it is less verbose. The programmer should be free to pick the most readable option for the current situation, without worrying about performance. Actually measuring the runtime of executing the three options tells a different story though. 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, both the first two versions are changed slightly to add this extra +0 to their calculations as well. The results of this testing can be seen in figure 2. This figure was obtained by executing a program that calls one of the three rowadd functions exactly once. To compensate for random background noise, each function was executed 100 times. To remove some outliers, the figure shows the 95 results closest to the mean for each function. As expected the V1 and V2 versions perform roughly equivalently with an average runtime of 5.8 seconds. The V3 version is however noticeably slower with an average runtime of 6.3 seconds. This means that the V3 version is around 10% slower than the other two versions.

```
1  int[800000,1000] rowadd_V1( int[800000,1000] a)
2  {
3    res = with {
4        ( [0,0] <= [i,j] < shape(a) - [3,0]) :
5            a[i,j]+ a[i+1,j]+a[i+2,j];
6      } : genarray([800000,1000],0);
7    return res;
8  }
```

(a) Adding rows by adding individual elements

```
1  int[800000,1000] rowadd_V2( int[800000,1000] a)
2  {
3    res = with {
4        ( [0] <= [i] < [800000-3]) :
5            a[i]+ a[i+1]+a[i+2];
6      } : genarray([800000],genarray([1000],0));
7    return res;
8  }
```

(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 < [800000-3]) :
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 withloop

Figure 1: Three ways of adding a low number of rows

Figure 1c is a specific example of the larger pattern of using a fold operation to produce a non-scalar result, and then combining multiple of these results together in an array. This specific example could be improved by unrolling the loop, because the total number of iterations is low. This is not always possible however. The goal of this paper is to solve the general problem of the slowdown observed in nested fold withloops. To do this, the example of figure 1c will be analysed in more detail. In order to better understand why this example is slower than the equivalent programs in figures 1a and 1b, more details on the SaC compilation process are required. These will be discussed next.

4 SAC COMPILER

The SaC compiler works in a number of phases, like parsing, optimisations, code generation, etc. The most important phase for the purpose of this paper is the memory phase. This phase determines where results are located in memory. This can have a big impact on the performance of nested withloops, as these have to deal with intermediate results. Storing these intermediate results in a naive way can cause overhead, or even require copies to be made. This section explores the memory phase in more detail, to better describe the reasons for this overhead, and what is being done to prevent it. After that, section 5 will apply this information to the previously discussed example of figure 1. A more complete overview of the memory phase can be found in Grelck and Trojahner [12]. A more detailed description can be found in Trojahner [22].

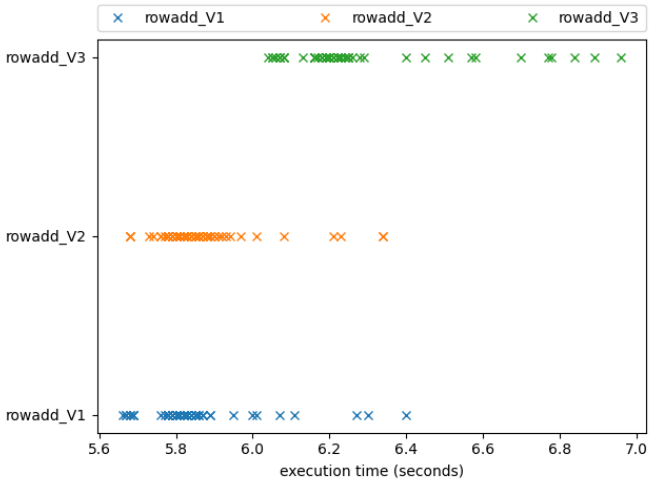


Figure 2: Testing results for the rowadd functions

4.1 Array representation

As stated in section 2, arrays in SaC consist of a shape description and data. Scalar values are arrays with an empty shape description. The data part is a sequence of all the data of the array, stored sequentially in row-major order. This is stored in a continuous piece of memory. The shape description determines the structure of the array. It consists of a sequence of numbers describing the number of elements in each dimension. An important detail is that while SaC allows the definition of multidimensional arrays with nested constructs such as array literals or withloops, internally all arrays are non-nested. Even a multidimensional array is a single array data structure, with one shape description and one continuous data section. This means that the compiler needs to translate nested definitions into code that generates a single array stored into continuous memory. Sections 4.2 and 4.3 will describe more details about the memory management in SaC. Then, section 6.3 will show how the memory of nested withloops is handled.

4.2 Reference counting

Automatically allocating memory is usually not that difficult. The compiler can statically add some extra memory allocation instructions in front of every statement that needs memory, and the problem is resolved. The trouble begins when memory supply is limited, and memory needs to be freed again. Any real system will have some limit on the amount of available memory, so freeing memory that is no longer used is important to keep the memory footprint of a program low. The memory footprint of a program is the minimum amount of memory required to run the program without crashing. By freeing memory as soon as possible the memory footprint can be minimised. The importance of this heavily depends on the host system, but especially on systems with low memory limits, such as embedded systems, this is important to keep the cost of hardware low. Additionally, for multi-threaded programs small differences in the memory footprint are multiplied further by the need to allocate memory in every thread. There are two main strategies to handle automatic memory freeing: delayed garbage collection and reference counting. SaC uses reference counting for this purpose. The choice between these two is heavily influenced by the main thing that needs memory in SaC: Immutable Arrays. The key challenge when dealing with immutable arrays is the need to copy the data whenever you update it. This copying can add a lot of overhead. This can be resolved by updating in-place without copying first. The downside is that blindly updating in-place means the immutability property is lost. This problem is also known as the aggregate update problem[15]. However, if an array is only used once, then it can be updated in-place at this usage without violating immutability. There are two main ways to deal with this. Either by automatically detecting when an array is only used once at runtime, or by ensuring that certain arrays can only be used once using the type system. An example of this second approach is uniqueness types[2], but other systems also exist. They share the common trait of putting more burden on the programmer. This does not fit well within the design philosophy of SaC, so SaC does not use it. The dynamic approach puts less burden on the programmer, but comes with extra runtime overhead. In order to tell if an array is still used after a certain point, the system needs to keep track of how many pointers to the data exist in the remainder of the program. Because the control flow of a program is usually not statically determinable, a dynamic solution is required. This can be done by keeping track of the level of sharing at runtime. This exact same system can also be used to free unused memory. Because if a pointer to some data still exists, then it cannot be freed. If no such pointer exists, then the memory can be safely freed, because no one can access it anymore anyway. This system of memory management is called reference counting or non delayed garbage collecting. In the context of array programming reference counting has established itself as a suitable trade off between programmer productivity and runtime performance[9, 12, 23].

The main idea of reference counting is to tag each piece of allocated memory with a number, which describes the amount of times that piece of memory will still be needed in the remainder of the program. This number starts of high, and is then reduced by one each time the memory is referenced. It can also increase again if at runtime a control flow is chosen that leads to a higher number of usages. This is mainly the case when at the end of a loop control switches back to the beginning instead of leaving the loop. If the number reaches zero, the memory can be freed. If the number reaches one, then the next statement that uses the memory is the only one that still needs it. This allows for more efficient updating without violating the functional semantics of SaC. A more detailed overview of exactly how this is done can be found in Trojahner [22], but for the purpose of this paper these details are not required. The mere fact that a reference count exists for each piece of memory, and what its value signifies is enough to explain the optimisation described in section 6.

4.3 Memory instructions

The main purpose of the memory phase of the compiler is to make all memory allocation and deallocation explicit. To do this, a couple of new expressions are added to the SaC language, and introduced

into the code. Memory instructions are only required for data allocated on the heap. In SaC, these are exactly all the non-scalar values. Scalars are allocated on the stack, and therefore don't need additional memory management to free them.

The memory phase adds the concept of memory to the program. Before this point, all variables can be seen as values. During the memory phase, two additional types of variables are introduced: empty memory, and filled memory. These three types of variables are referred to as having the types Val, Mem, and $MemVal$ respectively. At runtime, only filled memory can actually be used by the processor. This means that the compiler has to transform all non-scalar variables of type Val into variables of type $MemVal$. The following is an overview of the memory instructions that accomplish this:

- $alloc :: Val \rightarrow Mem$
  This is the most basic form of memory allocation, it takes a description of how much memory to allocate as an argument, and returns a pointer to the allocated memory.
- $fill :: Val \times Mem \rightarrow MemVal$
  The connection between the added memory instructions and the code that was already present. It takes a value as its first argument, and a pointer to some piece of allocated memory as its second argument. It then fills the memory with the given value.
- $copy :: MemVal \rightarrow Val$
  This primitive takes some expression that was already placed into memory, and returns its value. This is often used in combination with the $fill$ primitive to fill a second piece of memory with exactly the same contents as some existing memory, effectively making a copy.
- $suballoc :: Mem \times MemVal \rightarrow Mem$
  A more advanced form of memory allocation, often used to optimise memory allocation for withloops. It takes a pointer to some existing piece of allocated memory as its first argument, and an index into this memory as the second argument. The return value is then a pointer to a specific part of its first argument. This way memory can be filled one section at a time, in any order, instead of all at once front to back.
- $free :: MemVal \rightarrow Void$
  The $free$ instruction does exactly what it says on the tin. It takes some filled memory as an argument, and then frees it. This makes it available again for further use by $alloc$ expressions. There is no return value.

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

- $reuse :: MemVal \rightarrow Mem$
  The $reuse$ expression skips a memory deallocation and allocation step and transfers ownership of a piece of memory directly. It takes a $MemVal$ as an argument, and returns something of type Mem. This can for instance be used to calculate updates in-place, if it can be statically inferred that the old value is no longer necessary. This essentially performs a noop, as both filled and empty memory are essentially the same thing. The purpose of $reuse$ is to help keep the distinction between the two types of memory variables clearer. It also helps with the bookkeeping that is required for introducing reference counting, later in the memory phase.
- $alloc\_or\_reuse :: Val \times MemVal^{+} \rightarrow Mem$
  The $reuse$ primitive is great if you can statically infer that a piece of memory will not be used anymore after a certain point. This can however be difficult to check, mainly because loops and if statements can affect the control flow of the program in a dynamic way. Instead of just not trying to reuse in these situations, there is the $alloc\_or\_reuse$ expression. This expression, as the name suggests, contains the functionality of both an $alloc$ and a $reuse$ primitive. It takes all the arguments of both expression types, so information about how much memory to allocate, and a list of filled memory to reuse. At runtime, it checks the reference count of all the reuse candidates, and if any of them are one, then a pointer to the memory of that variable is returned. This is the same behaviour as the $reuse$ primitive, but now on a list of options instead of just one. If none of the variables can be reused, an entirely new piece of memory is allocated instead, as if a regular $alloc$ instruction had been used.
- $is\_reused :: Mem \times MemVal \rightarrow Val$
  This expression takes a Mem and a $MemVal$ variable as arguments, and then checks if the given $MemVal$ is using the given Mem as its memory. It returns $true$ if this is the case, and $false$ otherwise. This can be used to handle the initialisation of the memory allocated by $alloc\_or\_reuse$ instructions. If a $reuse$ happens, then often nothing has to be done. If an $alloc$ happens instead, then the contents of the newly allocated memory often have to be set to a copy of one of the reuse candidates. The is_reused expression can be used in combination with the if construct to make this choice at runtime, when this cannot be statically inferred at compile time.

Finally, one new memory primitive is added to the compiler to allow the optimisation discussed in section 6:

- $mem\_reuse :: Mem \rightarrow Mem$
  This primitive means almost the same as $reuse$. The main difference lies in the type. The existing $reuse$ primitive returns the memory used by some already filled memory, and thus takes a $MemVal$ as an argument. The new mem_reuse primitive takes memory as an argument, and returns that memory. This means that, again, a noop is performed, and this time not even the type changes. E.g., $x = \_mem\_reuse\_(y) \equiv x = y$. The main purpose of mem_reuse is to signify to the reference counting system that y is going to be used later in the form of x, and that it therefore cannot be statically freed, even if the reference count of y reaches 0. After the compiler phases that deal with reference counting are done, all mem_reuse instructions are removed from the code, and variable propagation is used to eliminate the extraneous variables. I.e., all references to x after $x = \_mem\_reuse\_(y)$ are replaced with y, and the assignment itself is removed.

4.4 Withloop in-place computation

As stated earlier, arrays in SaC occupy a continuous section of memory. A genarray withloop works by first allocating this memory,

and then filling it one element at a time. This becomes more complicated if the elements themselves are also defined with a withloop. If the same memory strategy is followed recursively, then the inner withloop will also allocate memory and fill it. This piece of memory allocated and filled by the inner withloop will then contain one element of the outer withloop. However, since this memory was freshly allocated, it is not part of the continuous section of memory reserved by the outer withloop. To fix this, the result of the inner withloop would have to be copied over to the correct memory of the outer withloop. This is of course not efficient. To prevent this, a sub-allocation can be done instead of a regular allocation of memory. This works in the same way, but instead of returning fresh memory, a piece of previously allocated memory is returned. By replacing the allocation done by inner withloops with sub-allocations into the memory of the outer withloop, the need for extra copies to move data can be resolved. This optimisation, called the in-place computation optimisation, can be applied later on in the compiler. This allows the initial memory allocation for withloops to follow the simple recursive scheme.

Using the memory primitives described in section 4.3, the memory allocation process for withloops can be made explicit. An example of how the memory management of a nested withloop looks before any optimisations change anything can be seen in figure 3a. This example abstracts away over computation specific details by using *italic* variables, which represent arbitrary expressions. Line 1 allocates memory for the outer withloop. At line 7 and 8, this memory is filled one element at a time using the *suballoc* and *fill* functions. The overhead this introduces can be seen in the form of the *copy* call on line 8. The in-place computation optimisation attempts to make the computation of withloop elements happen in-place. To do this, it replaces the *alloc* call responsible for allocating the memory of the elements with a *suballoc* call. This makes sure that the memory that is used to calculate the elements is the same memory where the result is expected to be. This way, the extra copy instructions at line 8 can be avoided. Figure 3b shows the results of applying the in-place computation optimisation to figure 3a. Because the *alloc* statement from line 3 is now gone, the other memory management dealing with moving and freeing this memory is now no longer needed, increasing performance.

For fold withloops, this works differently. Fold withloops do not allocate their own memory beforehand, but follow a different scheme instead. Functions in SaC allocate their own memory for their result. Because the result of a fold withloop is always determined by a function call to its folding function, fold withloops don't do the memory allocation for their result directly. If fold withloops would also allocate their own memory, then the result of the folding function would always need to be copied over to the memory the fold withloop allocated. Instead, the memory used for the final result of a fold withloop is almost always allocated within the function body of its folding function. The only exception being if the upper and lower iteration bounds of the fold withloop are chosen such that no iteration is executed. This means that the folding function is never applied, in which case the result will be the neutral element which is already located in memory.

```
1  \\ italic variables are placeholders
2  outer_mem = alloc(outer_shape);
3  outer = with { (iv_lb <= iv < iv_ub) {
4         inner_mem = alloc(inner_shape);
5         inner = with {
6             (jv_lb <= jv < jv_ub) :
7                 computed_element
8         }: genarray(inner_shape,inner_default_el)
9         res_mem  = _suballoc_(outer_mem,iv);
10        res_copy = _fill_( _copy_(inner),
11                           res_mem);
12        free(inner);
13        } :res_copy
14    }: genarray(outer_shape,outer_default_el)
```

(a) before in-place computation

```
1  outer_mem = alloc(outer_shape);
2  outer = with { (iv_lb <= iv < iv_ub) {
3         inner_mem = _suballoc_(outer_mem,iv);
4         inner = with {
5             (jv_lb <= jv < jv_ub) :
6                 computed_element
7         }: genarray(inner_shape,inner_default_el)
8         } :inner_mem
9     }: genarray(outer_shape,outer_default_el)
```

(b) after in-place computation

Figure 3: Nested genarray withloop with explicit memory instructions shown

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 withloops produce significantly slower code than equivalent programs defined with genarray withloops. To better understand this, we now look at how the SaC compiler tries to optimise the three versions of rowadd introduced in figure 1. The first two versions also contain the extra addition of 0 as discussed in section 3 to make them more comparable with the fold version. The hide function is used to make sure the compiler does not optimise this extra addition away, by hiding the fact that it is an all zero array. The semantics and implementation of the hide function are the same as the identity function. The first version can be seen in figure 4.

The main change is the addition of explicit memory instructions, as there is little to optimise here. The second version in figure 5 now has a nested withloop within the withloop that was already there. This second withloop represents the vector addition that rowadd_V2 is doing. While it is possible to write these non-scalar additions down directly using the + function, it is not possible to directly execute these additions in one step at runtime. The + function on vectors therefore needs to be translated into regular addition on scalar values. The + function is overloaded, and the SaC compiler picks the most fitting implementation from the standard library for the current context. In the case of rowadd_V2, the

```
1  int[800000,1000] rowadd_V1( int[800000,1000] a)
2  {
3    vect = hide(genarray([1000], 0));
4    res_mem = _alloc_([800000,1000]);
5    res = with {
6        ( [0,0] <= [i,j] < shape(a) - [3,0]) {
7          elem_mem = _suballoc_(res_mem,[i,j]);
8          elem_val = vect[j] + a[i,j] +
9                     a[i+1,j]+ a[i+2,j];
10         elem = _fill_(elem_val, elem_mem);
11       }: elem
12     } : genarray([800000,1000],0);
13 return res;
14 }
15
```

Figure 4: Optimised version of rowadd_V1

```
1  int[800000,1000] rowadd_V2( int[800000,1000] a)
2  {
3    vect = hide(genarray([1000], 0));
4    res_mem = _alloc_([800000,1000]);
5    res = with {
6      ( [0] <= [i] < [800000-3]) : with {
7        ( [0] <= [j] < [1000]) {
8          elem_mem = _suballoc_(res_mem,[i,j]);
9          elem_val = vect[j] + a[i,j] +
10                    a[i+1,j]+ a[i+2,j];
11         elem = _fill_(elem_val, elem_mem);
12       }: elem
13
14      } : genarray( [1000], 0);
15    } : genarray([800000],vect);
16   return res;
17 }
18
```

Figure 5: Optimised version of rowadd_V2

chosen implementation consists of a single withloop. Later on a different optimisation called withloop scalarization will optimise this further by merging the two one dimensional withloops into a single two dimensional withloop. This optimisation is out of scope for this paper, but more details can be found in Grelck et al. [11]. After withloop scalarization rowadd_V1 and rowadd_V2 look nearly identical, which explains the similar performance seen in figure 2. Rowadd_V3 looks different however. While the compiler will not insert an actual for loop as seen in figure 6, the generated executable code will operate roughly like this. The main structure of a translated fold withloop consists of an accumulator (*inner_res*) and an update step, which is represented by the for loop. The starting value of the accumulator is the neutral value given to the fold withloop, which in this case is a vector of all zeros. Then for each iteration of the fold loop, the accumulator gets overwritten with a new value. The final value of the accumulator is used as the result for the whole withloop. The actual operation performed within the update step is again a withloop, because almost all operations resulting in non-scalar arrays are internally implemented as withloops. The main difference between rowadd_V2 and rowadd_V3 is that the nesting of withloops in rowadd_V3 is not direct. The for loop between the outer and inner genarray withloop prevents withloop scalarization from being applied. This results in code that still contains nested withloops after all optimisations are applied.

This nesting introduces overhead, but this overhead can be prevented with the in-place computation optimisation as discussed in section 4.4. However, applying this optimisation to a nested fold-withloop is not as straightforward as with nested genarray withloops. The reason for this is that it relies on knowing exactly which *alloc* statement is allocating the memory for the withloop elements. This information is available as long as the the memory is allocated in the current scope. In figure 5, this is the allocation on line 4, as genarray withloops allocate their memory before the withloop itself. For figure 6, this allocation is contained within the for loop on line 9. It is not executed just one time, but once per iteration. This makes it impossible to just replace the *alloc* statement responsible for the final result with a *suballoc*, as this would be exactly the call to *alloc* in the final iteration. Doing the substitution would affect all iterations, not just the final one. This would transform the loop from allocating fresh memory in each iteration, to one that allocates the same memory every iteration. This could work fine for some functions, but won't work for others. It is non-trivial to determine if doing the *suballoc* substitution here will result in problems or not. As such, the SaC compiler currently does not apply the in-place computation optimisation to nested fold withloops at all.

As discussed, *rowadd_V3* results in a nested withloop, because the withloop scalarization optimisation is not applicable. The in-place computation optimisation is also not applicable to a nested fold withloop. This means that *rowadd_V3* ends up allocating its own memory within the folding function. At the end of the fold withloop, on line 20 of figure 6, the contents of this memory need to copied over to the memory allocated by the outer withloop. After this, the memory allocated by the inner withloop needs to be freed. This is exactly the overhead shown on lines 20 through 22 of figure 6. This overhead of one memory allocation, one copy operation over the contents of this memory, and one memory deallocation, could be the reason why *rowadd_V3* is slower than the other two versions. If this is the case, then the performance can be increased by finding a way to make the in-place computation optimisation applicable. The remainder of this paper will therefore focus on finding a way to make the in-place computation optimisation also applicable to fold withloops, and testing if this reduces the speed deficiency measured in figure 2.

The in-place computation optimisation can theoretically still be applied to nested fold withloops, but not as straightforward as with nested genarray withloops. The reason for this is that the programmer is free to choose a folding function for a fold withloop, with no restrictions on what can happen in the body of that function. Different functions behave in different ways, and not all of them allow for calculating the result in-place. If the result can't

```
int[800000,1000] rowadd_V3( int[800000,1000] a)
{
  vect = hide(genarray([1000], 0));
  res_mem = _alloc_([800000,1000]);
  res = with {
    ( [0] <= iv < [800000-3]) {
      inner_res = vect;
      for ( j=0; j<3; j++) {
        inner_mem = _alloc_or_reuse_(inner_res,
                                     [1000]);
        inner_res = with {
          ( [0] <= k < [1000]) {
            fold_iter_mem = _suballoc_(
                                inner_mem,[k]);
            fold_iter_val = inner_res[k] +
                           a[j,k];
            fold_iter = _fill_(fold_res_val,
                               fold_res_mem);
          }: fold_iter
         } : genarray( [1000], 0);
      }
      inner_res_mem = _suballoc(res_mem,iv);
      inner_res_copy = _fill_(_copy_(inner_res),
                              inner_res_mem);
      _free_(inner_res);
      } : inner_res_copy;
    } : genarray([800000],vect);
  return res;
}
```

Figure 6: Optimised version of rowadd_V3

be calculated in-place, then the in-place computation cannot be applied. This is for instance not possible with any folding function where the result occupies a different amount of memory than the intermediate accumulators. A specific example is for instance string concatenation. A list of strings can be folded together using string concatenation, but every intermediate accumulator would require a different amount of memory. A different example of where this is not possible is folding with the max function. The result will be one of the arguments, which will already be located in memory somewhere. To use the in-place computation, it would need to be known which regular allocation to replace. In order to know this, the result of the max function needs to be calculated first. By this time, it is already too late to use a suballoc, because the regular allocation was already executed. While both of these examples illustrate why sub-allocation with fold withloops is not always possible, it is certainly possible sometimes. An example of this is the code given in figure 6, and by extension the code in figure 1c. This example uses an accumulator of constant size, and is not overwriting it with content that is already allocated elsewhere. By detecting this in the compiler, the performance of rowadd_V3 can possibly be brought more in line with that of the other versions. The next section will discuss how this can be done.

6 IN-PLACE ACCUMULATOR OPTIMISATION

As discussed in section 4.4, the main reason why the in-place computation optimisation is not applicable to fold withloops is because the source of their memory is not clearly defined. For example, folding with a constant function would allocate fresh memory inside the function itself. 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. At that point, it is too late to apply the in-place computation optimisation and a copy is required. There is however one common source of the result memory of a fold withloop that does have potential to benefit from the suballoc system. This occurs when the fold withloop 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 common folding operations do some kind of reduction where they combine a value with an accumulator of a fixed size to create a new value for the accumulator. This is especially common with arithmetic operations such as addition or matrix multiplication on square matrices. These kinds of operations would greatly benefit from being calculated in-place, especially when they are called on large arrays.

The actual optimisation consists of three main steps. Each of these steps will be expanded upon in its own section.

(1) Check if the optimisation is applicable to a particular fold withloop. Only continue with the next step if it is. (section 6.1)
(2) Allocate memory for the accumulator outside the fold loop. Replace the attempt at reusing the old accumulator with explicitly reusing this new memory. (section 6.2)
(3) Replace the alloc statement for the memory of the accumulator with a suballoc statement, then delete the now unnecessary copy instructions at the end of the fold withloop. (section 6.3)

6.1 Applicability

The first step in applying the optimisation is checking if the optimisation can actually be applied. This can be done by traversing the program and looking for the pattern shown in figure 7a. This figure shows a simplified version of the code pattern as it looks like during the memory phase of the SaC compiler. It only shows relevant code, but the actual pattern allows for any amount of arbitrary statements to be inserted between any of the statements shown in the figure. Because of the functional semantics of SaC, each variable has exactly one definition. Even function calls in between the definition and usage of a variable cannot affect its value or where that value is located in memory. This allows for relatively straightforward tracing of where the memory of a variable expression is coming from. The main purpose of this step is tracing where the memory of the accumulator of the fold withloop is coming from, and checking if this is the same memory used by the previous accumulator.

Fold withloops 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

```
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     accu = fill_like_operation(expr, accu_mem);
7     iteration_res = _unshare_(accu, iv_i);
8     } :iteration_res
9   }: fold(fold_op, neutral_el)
```

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

⇓

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

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

Figure 7: In-place accumulator optimisation. Code before optimisation (top) and equivalent code after optimisation (bottom).

first iteration. If the withloop tries to reuse the memory of this accumulator variable for the result, then the optimisation is applicable. The first thing to do is therefore looking for an *_alloc_or_reuse_* statement on the accumulator variable, and then checking that the result of this statement is used as the memory for the result of the whole withloop, which is the expression at line 8. In this example, the memory of the accumulator is stored in *accu_mem*. Line 5 is an abstraction of the actual calculation the withloop is doing. This calculation usually consists of several steps, but always ends with a final step that stores the result in memory. Line 6 represents this last assignment to memory. It stores the result of the calculation, represented by *expr*, in the *accu* variable, using *accu_mem* as the needed memory. The $fill\_like\_operation$ here is a placeholder for whatever construct is used to fill *accu_mem* with the value of *expr*. This can simply be the fill primitive as discussed in section 4.3, but it could also be a withloop. The definition of *expr* can be any arbitrary expression, and the name *accu* is also just an example name for the result variable. The important part is that after this line, *accu* is using *accu_mem* as its memory, which is the same memory used by the accumulator. Line 7 represents code inserted by another optimisation, and returns the *accu* variable, as long as it does not use the same memory as *iv_i*. Then in line 8 the $iteration\_res$ variable, still using the memory from the old accumulator, is passed on as the result. 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 logical next question then becomes, where is this memory allocated? This is made explicit in the next step of the optimisation.

6.2 Explicit accumulator memory

After having determined that a fold withloop fulfils all the required conditions, the actual optimisation can be applied. As previously discussed, the main goal is to clearly define where the memory for this fold withloop is allocated. Finding the original *alloc* statement that does this is difficult and it is often outside of the local scope. E.g., within a different function than the one containing the fold withloop. This means that even if it is located, it might not be possible to do anything with it because other sections of code are also using that same statement in some way. The solution is to allocate an entirely new section of memory, which is therefore guaranteed to be local and not used by anything else. It is then also clearly known which statement to later adjust for in-place computation, namely the one that just got introduced. This will bring the fold withloop in line with the genarray withloop, which also allocates a fresh section of memory before the start of the loop.

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

(1) The actual allocation has to go via the run-time memory management system through the operating system. This takes a small amount of time, depending on the host system and the memory manager in use. (SaC supports several options, depending on the host operating system).
(2) The resulting memory often has to be initialised. This means doing a copy, which takes some time as well.

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 7a, 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 values, empty memory or filled memory, as discussed in section 4.3. They are respectively referred to as having type Val, Mem and $MemVal$. Mem variables are introduced by the compiler, and therefore they are only used in specific situations. Most notably, they are essentially write only. There are only a few constructs within SaC that can do something with a Mem variable, as discussed in section 4.3. These

are $fill$, $free$, *is_reused*, and *suballoc*. None of these can access the value of the memory. Using $fill$ completely overwrites the memory and $free$ simply frees the memory without looking into its value. The *is_reused* primitive only compares pointers to check if a value is stored in a piece of memory, which also does not require looking at the value of the memory. Finally, *suballoc* does look into the memory, but only does some pointer arithmetic, again without looking at the stored data. It returns a new *Mem* variable to a smaller piece of memory, but by induction that one will also not be read into. The only way to read from memory is to fill a *Mem* variable first using $fill$ or some other primitive with similar semantics. This will then return a $MemVal$ variable, which can be read from. This means that all *Mem* variables are uninitialised, and initialisation only happens when the memory is filled with some primitive. All of these primitives will then return a new $MemVal$ variable, which can be used for reading the values, but not for writing. The important detail from all 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 short explanation skipped over a lot of details about the memory management system which are irrelevant for the in-place accumulator optimisation. For instance, section 4.3 does not contain all memory primitives. The missing ones are all variations of $fill$ however, so the same reasoning still holds. A more thorough explanation about the details of memory variables and how they can and cannot be used can be found in Trojahner [22].

The fact that *Mem* variables are write only means that in the example of figure 7, *accu_mem* cannot be used to access the accumulator. This can only be done through the *accumulator* variable, which is a $MemVal$. On the other hand, writing to the accumulator is done exclusively through *accu_mem*. This means that after the replacement of *accum_mem* with $fold\_mem$ using the new *mem_reuse* primitive on line 4 of figure 7b, $fold\_mem$ does not need to be initialised, as it will not be read from anyway. It will exclusively be used to store the next version of the accumulator. On the first iteration this is the same as writing the result to a fresh piece of memory, which is always fine. On subsequent iterations, this will overwrite the previous accumulator, as that was using the same memory. This is also fine, because the un-optimised code was doing the exact same overwrite using *alloc_or_reuse*. Since after the optimisation the same memory gets overwritten in the same place in the code, this cannot cause any problems that were not already there.

6.3 In-place computations

The final step of the optimisation is to put the initial idea into action, with is to reuse the existing implementation of the in-place computation optimisation in the fold context. As planned, it turns out it suffices to mark the affected fold withloops with where their memory is allocated. Since the fold withloops 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. All that is required is to flag the fold withloops that can now be further optimised with their result memory, and the existing optimisation code takes care of the rest.

7 RESULTS

Theoretically, the optimisation proposed in section 6 should solve some of the extra overhead seen for $rowadd\_V3$ in figure 2. 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, the same test that gave the results for figure 2 is repeated for $rowadd\_V3$ using the new in-place accumulator optimisation. This means executing a program that calls the function once, a hundred times, calculating the mean execution time, and then plotting the 95 results closest to the mean. The results of this test, added to the previously gathered results, can be seen in figure 8. As expected $rowadd\_V3$ performs significantly better when compiled with the in-place accumulator optimisation enabled. The mean execution time shifted from 6.3 seconds to 6 seconds, which is a roughly 5% speedup. 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 withloops, which will always have some loop overhead compared to running just a single withloop. The in-place accumulator optimisation seems like a step in the right direction however.

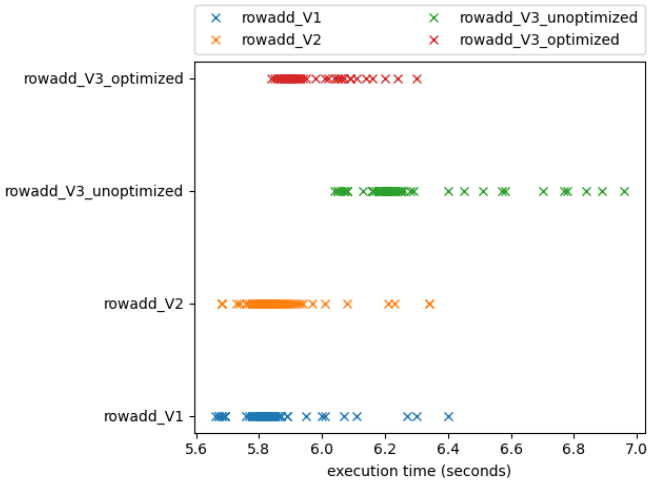


Figure 8: Testing results for the rowadd functions, including optimised V3

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 to this, we look at variants of the problem, where the two array dimensions and the amount of work being done is varied. We 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.

```
1  use Array: all;
2  use StdIO: all;
3  use Benchmarking: all;
4
5  \\ Id functions to prevent the compiler from
6  \\ calculating the entire program as a
7  \\ constant at compile time.
8  noinline int[INNER] id( int[INNER] a)
9  { return a; }
10
11 noinline int[OUTER,INNER] id( int[OUTER,INNER] a)
12 { return a;}
13
14 int main()
15 {   \\ Calculate initial elements. Hide
16     \\ details behind non-inlined functions
17     \\ so they are not seen as constants.
18     zeroes = id(genarray([INNER], 0));
19     a = id( genarray([OUTER,INNER], 1));
20
21     \\ Start benchmarking time.
22     i1 = getInterval( "vect", 0);
23     start( i1);
24
25     \\ Calculate a single update step
26     updated_a = with {
27         ( . <= iv < [OUTER-N]) {
28            \\ Calculate c = a[iv] + a[iv+1]
29            \\               + ... + a[iv+N].
30            c = with {
31               (iv <= jv < iv+N) : a[jv];
32                } : fold( +, zeroes);
33            } : c;
34       } : genarray( [OUTER], zeroes);
35     \\ Stop benchmarking time.
36     end( i1);
37     \\ Print part of result to make sure the
38     \\ calculation does not get optimised away.
39     print(updated_a[1,2]);
40     \\ Print benchmarking results
41     printResult( i1);
42     t,u = returnResultUnit( i1);
43     printf( "GFLOPS per %s: %f \n",
44      u, tod((OUTER-N)*INNER*N)/(1000000000.0*t));
45     return 0;
46 }
```

Figure 9: Benchmark which calculates an update step

The program shown in figure 9 is used as the basis of these tests. It is a generalised version of the example shown in figure 1c. 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 23 to line 31. 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. This is how many billion floating point operations the program can execute in a second, which is calculated by dividing the total amount of floating point operations by the execution time. The exact flops calculation used can be seen on line 41.

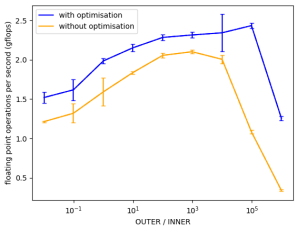
The main calculations done by the program can be summarised as follows:

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

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 10 shows the results of running the program with or without the optimisation using the inputs seen in figure 10b. N was kept constant at 2. 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 10a. 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 10b. 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 no matter the values for $OUTER$ and $INNER$. This makes sense, because the optimisation offers a scaling improvement in both cases. If $OUTER$ is big, then the inner fold withloop needs to be calculated many times. For each iteration of the outer withloop, 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 however. 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. This means that 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 withloop adds some overhead in setting up breaking down. The more time is spent entering and leaving withloops, 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

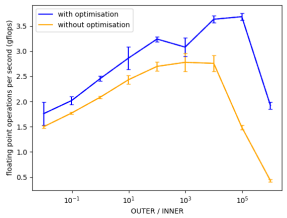
(b) Values for *INNER* and *OUTER*.

Figure 10: Comparison of code performance with and without the optimisation.

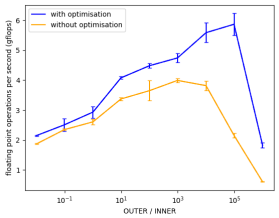
The effect of the loop overhead can be reduced by increasing the amount of work a single loop iteration does. The workload of the inner fold withloop can be increased by increasing the value for N. This means that it will add more rows together, by doing more iterations. This will then increase the workload for a single iteration of the outer withloop. The results of this can be seen in figure 11. 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 withloop ($N * INNER$) is still small. A smaller workload for the inner withloop 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 28 of figure 9 as described in figure 12.

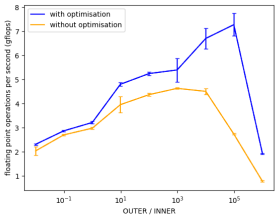
This change does not change the semantics, but it does allow the compiler to unroll the withloop because the index range of *jv* is now just depending on constants, and no longer on *iv*. After this



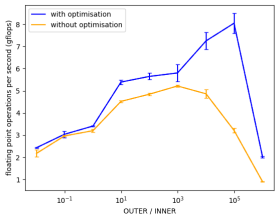
(a) Execution results for $N = 3$



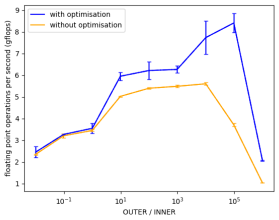
(b) Execution results for $N = 5$



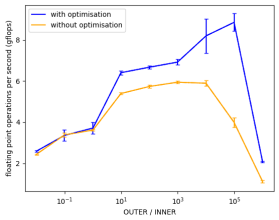
(c) Execution results for $N = 7$



(d) Execution results for $N = 9$



(e) Execution results for $N = 12$



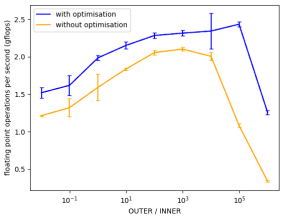
(f) Execution results for $N = 15$

Figure 11: More comparison results for bigger values of N

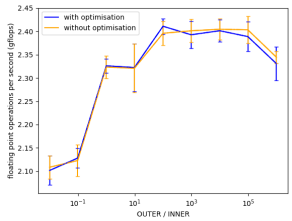
```
(iv <= jv < iv+N) : a[jv];

                ⇓

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

Figure 12: Modification to fold withloop that will allow withloop unrolling

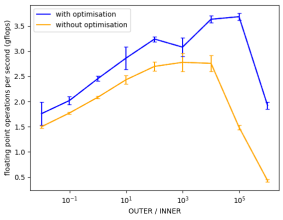
change, the compiler sees that the fold withloop is only calculating a total of N iterations. For small values of N (by default 9) the compiler will decide that it is more efficient to unroll the withloop. This means that instead of compiling into code with a loop like described in figure 6, the loop 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 loop is now gone, the reason why withloop scalarization was not applicable is now also gone. This means that the unrolled inner withloop gets merged with the outer withloop, removing all the overhead caused by the inner withloop in the process. Since the main goal of the in-place accumulator optimisation is to reduce this overhead, removing it entirely is the best achievable result. The



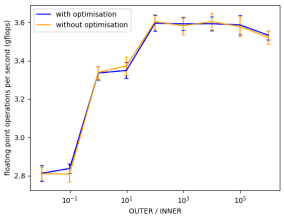
(a) Unmodified for $N = 2$



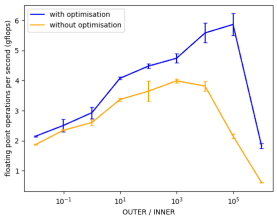
(b) Unrolled withloops for $N = 2$



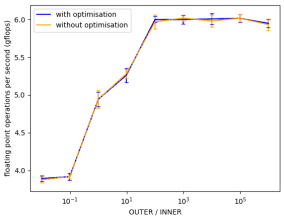
(c) Unmodified for $N = 3$



(d) Unrolled withloops for $N = 3$

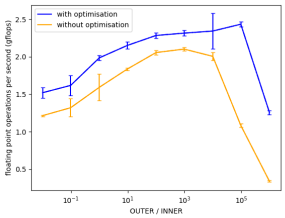


(e) Unmodified for $N = 5$

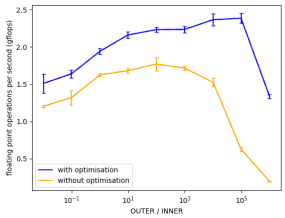


(f) Unrolled withloops for $N = 5$

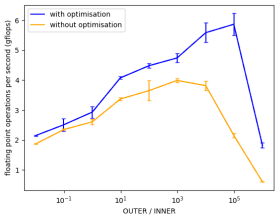
Figure 13: Comparison of results with and without the modification described in figure 12



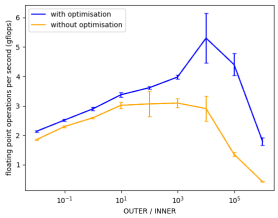
(a) Execution results with the phm on for $N = 2$



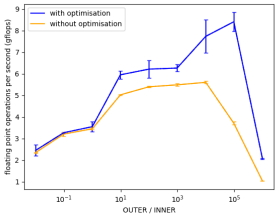
(b) Execution results with the phm off for $N = 2$



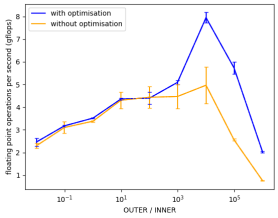
(c) Execution results with the phm on for $N = 5$



(d) Execution results with the phm off for $N = 5$



(e) Execution results with the phm on for $N = 12$



(f) Execution results with the phm off for $N = 12$

Figure 14: Comparison of results with and without the phm

results of this experiment can be seen in figure 13. The right side of the figure contains the results obtained by the modification in figure 12, which optimises the inner fold withloop away. The left side of the figure contains the equivalent results without this modification. As expected, with the change to the inner fold withloop the optimised and non optimised code are performing equivalently in this case. If there is no inner withloop, 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 withloop is fully optimised away. On the edges of the graphs, there is a larger gap in performance. This same gap is however also there with the non-optimised results (the orange lines) in the left graphs. This suggests that this gap is caused by the nesting of the withloops, and not by the loop overhead.

Since the optimisation aims to reduce overhead caused primarily by memory management, an interesting experiment is to look at the 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 private heap manager 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 private heap manager, unless explicitly stated otherwise. The previously shown results are shown next to results of the same experiments with the private heap manager off in figure 14. The general performance of the program without the private heap manager is lower that when it is enabled. This makes sense, because the private heap manager is specifically tailored for SaC. It therefore performs better than the more general system default memory manager does, which is used when the phm is disabled. 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 private heap manager can reduce this loop overhead by streamlining memory allocation and de-allocation. Without the private heap manager, 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 focused heavily on the specific problems caused by nesting fold withloops 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[13], Lift[20], Accelerate[5], Sisal[10], SkePU[8], Marrow[19], Halide[17], etc follow a similar design philosophy, and therefore might run into similar memory problems. These systems also run into the challenge of dealing with memory for nested computations in an efficient manner. This section describes a few of these related projects. Details on the exact memory strategies used by these languages are often not available. In addittion, 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[13, 14] 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 [4]. This approach has been refined over time, but is not yet as fast as hand optimised code [3, 7]. It might be possible to combine the flattening approach and in-place computation approach in one system, which would reduce the amount of cases where optimisation is not possible. However, Futhark is heavily focused on parallel computing, and this research has focused on generating sequential code.

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 [20]. 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.

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[5], SkePU[8], Marrow[19] and Halide[17]. 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 often focuses on flatting[6, 16]. Marrow however 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 form 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 is probably not combinable with defining specific compiler optimisations like the one proposed in this paper, but is likely to solve the same problem.

Other languages like the Sisal[10] take an aproach 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 might also be of help.

9 CONCLUSION

This paper has looked into the problem that SaC code using nested fold withloops performs worse than equivalent code without fold withloops. This violates the design philosophy of SaC, and forces programmers to think about implementation details again if they want efficient code. To make the problem more specific, section 3 introduced the three versions of *rowadd*. These did not have the same performance, while they did compute the same result. A similar performance is desirable to free the programmer from having to worry about efficiency. The hypothesis was that one of the main causes for the performance discrepancy is a lack of memory reuse in nested fold withloops. After analysis of the way the three versions of rowadd are compiled this turned out to be correct. Several other optimisations, such as withloop scalarization and in-place computations, are not applicable to nested withloops. This leads to less efficient memory management and extra copies in the generated code. Rewriting all these optimisations to also work for fold withloops is difficult at best and impossible at worst. Instead, this paper takes the approach to rewrite the fold-withloops themselves to make them more similar to other types of withloops. Fold withloops cannot predict where the final result will be allocated. This information is required for the existing memory reuse optimisations. The key insight of this paper is that if a fold withloop can be calculated in-place, then this information can be made available. The in-place accumulator optimisation introduced in section 6 is designed to detect when this is possible. It brings the memory management of fold withloops in-line with the memory management of genarray withloops, if the fold withloop accumulator can be calculated in-place. This allows the existing in-place computation optimisation to also work for these nested fold withloops. The source code for the in-place accumulator optimisation is available online [24].

After doing a performance evaluation in section 7, the conclusion is 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 that the optimisation gives an improvement when the inner fold withloop 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 withloop 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 withloop 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 withloop 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. While the in-place accumulator optimisation in its current form is already helpful in many situations, there is still room for further improvement.

Currently, it is only applicable to fold withloops of a specific form, namely those that attempt to reuse their accumulator for their result. By either making the optimisation applicable to more types of fold withloops, or by rewriting fold withloops such that the optimisation becomes applicable, more performance gains could be realised. Another road to further improvement lies in other optimisations that currently ignore fold-withloops because they are too different from other withloops. The in-place accumulator optimisation brings the memory management of affected fold-withloops more in line with that of other types of withloops. This decreases the difference between the withloop types, and might make it easier for other optimisations to also affect fold withloops in the future.

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] Erik Barendsen and Sjaak Smetsers. 1993. Conventional and uniqueness typing in graph rewrite systems. In *Foundations of Software Technology and Theoretical Computer Science*, Rudrapatna K. Shyamasundar (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–51. https://doi.org/10.1007/3-540-57529-4_42

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. 1990. A report on the sisal language project. *J. Parallel and Distrib. Comput.* 10, 4 (1990), 349–366. https://doi.org/10.1016/0743-7315(90)90035-N Data-flow Processing.

[10] 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

[11] 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

[12] 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.

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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[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] Sebastian Ullrich and Leonardo de Moura. 2020. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming. arXiv:1908.05647 [cs.PL]

[24] 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.

[25] 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. https://doi.org/10.1145/3315454.3464312