# On Making Modulo-Based Array-Element Rotations Affordable

Michiel Verloop
Michiel.Verloop@ru.nl
Radboud University
Nijmegen, Netherlands

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

## ABSTRACT

Modulo operations on array indices provide a convenient means for specifying rotations of the elements of arrays. Naive compilation of such operations introduces overhead due to the integer divisions involved. This paper proposes an optimisation for avoiding such overheads. The key idea is to systematically replace modulo operations within loops by partitioning such loops and replacing the modulo computations with division-free equivalents.

We present a code transformation scheme for the proposed optimisation, sketch an implementation in the context of the array programming language SaC, and provide an initial performance evaluation. We look at several benchmarks ranging from micro-benchmarks to some more realistic applications and run them on multi-core systems. Depending on problem size, machine characteristics, and application context, we see performance benefits of up to a factor of 14.

## KEYWORDS

Array Programming, Rank-Polymorphism, Modulo, Optimisation, Rotation

**ACM Reference Format:**
Michiel Verloop and Sven-Bodo Scholz. 2023. On Making Modulo-Based Array-Element Rotations Affordable. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Declarative programming languages advocate a programming style that focuses on the "what" rather than the "how". This style usually improves programmer productivity [3], facilitates formal reasoning about programs [1], and it offers a lot of potential for optimisation and code generation for parallel executions [11]. The price for all these advantages lies in the fact that the semantic gap between declarative programs and the executing hardware is considerably higher than the semantic gap between imperative languages and hardware.

In this paper, we look at a performance challenge that arises from a more intuitive algorithmic specification rather than a higher level of abstraction. Cyclic operations can be elegantly defined using a modulo operation on indices. As a simple example for such an operation, consider code for rotating all elements of a vector `v` of `100` integers by `3` elements towards decreasing indices. In an array language such as SaC, this can be conveniently expressed as a tensor comprehension:

```
{ iv -> v[mod (iv+3, [100])] | iv < [100]}
```

The tensor comprehension computes a result array of 100 elements by computing each result element at index `iv` by selecting an element from the vector `v` at position `mod (iv+3, [100])`. A naive compilation of this code leads to 100 evaluations of the modulo operation `mod` which constitutes a considerable overhead, given that all we do is copy 100 elements.

Looking at this simple example, we immediately see that we can avoid the modulo operation by breaking up the comprehension into two partitions:

```
{ iv -> v[iv+3]   | iv < [97];
  iv -> v[iv-97] | iv < [100]}
```

While it may be up for debate which of these two formulations is better from a declarative perspective, it is clear that the latter will perform better when compiling both naively. Defining rotate is more complex when creating a generic abstraction that allows for rotation by arbitrary values and is rank-polymorphic, i.e. applicable to arrays of arbitrary dimensionality. Using `mod`, however, this can be achieved concisely in a rank-polymorphic array programming language such as SaC:

```
int[*] rotate (int[.] offset, int[*] array)
{
   shp = shape(array);
   return { iv -> array[mod (iv - offset, shp)]
          | iv < shp};
}
```

Note that this definition relies on a few properties. It requires that the modulo operation always returns a value ranging between `0` and `shp` (exclusive), even for negative values. Furthermore, the length of the vector `offset` must match the dimensionality of `array`. Finally, it relies on the rank-polymorphic behaviour of tensor comprehensions. Transforming this definition into one that avoids modulo operations is a non-trivial task. Doing so requires up to $2^d$ partitions, where $d$ equates the dimensionality of `array`. Overall, this ends up in a rather complex code iterating over the dimensionality of `array` and some pre-processing to determine the partition bounds and offsets.

This paper aims to resolve the conflict between the declarative power of modulo and the performance loss typically associated with it. We propose a generic optimisation for modulo operations that is used on iteration variables within loops by splitting such loops into multiple separate loops that do not require any modulo operations within their respective bodies. When applying this optimisation to the example above, we obtain a multi-partition tensor comprehension without any modulo operations in its $2^d$ partitions.


*Conference'17, July 2017, Washington, DC, USA*

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

The contributions of this paper are:

- We present two transformation schemes that capture different application scenarios of modulo operations within loops. These allow for the systematic removal of most modulo operations from loop bodies.
- We include a formal derivation of the transformation schemes from the algebraic properties of the modulo operation.
- We sketch the implementation in the context of the SaC compiler `sac2c`.
- We conduct a performance analysis that quantifies the impact of this optimisation in a range of different scenarios. We look at three different shared-memory multi-core architectures, including a GPU-accelerated one, and we examine a range of different applications, from micro-benchmarks towards realistic applications.

# 2 BACKGROUND

## 2.1 Modulo

In array languages such as APL or SaC, the modulo operation is defined in a way that is suitable for defining index rotations. This leads to two properties that set it apart from remainder functions that are usually available in languages such as C. When the divisor is zero, the quotient is also zero. This avoids a division by zero and means that a modulo computation is never ill-defined. The quotient is defined using floored division, which guarantees that $a \bmod d$ always produces values between 0 and $d$, provided $d \neq 0$. Hence, for $a, d \in \mathbb{Z}$, modulo is defined as

$$a \bmod d = a - a \operatorname{div} d \cdot d$$

$$a \operatorname{div} d = \begin{cases} 0 & \text{if } d = 0 \\ \lfloor \frac{a}{d} \rfloor & \text{otherwise} \end{cases}$$

For reasoning with modulo, we establish the following properties:

*2.1.1 Property 1.* $\forall a, d \in \mathbb{Z}: d \neq 0 \wedge a \bmod d = 0 \leftrightarrow \frac{a}{d} = \lfloor \frac{a}{d} \rfloor$

Proof:

$$\begin{aligned} a \bmod d &= a - a \operatorname{div} d \cdot d \\ 0 &= a - a \operatorname{div} d \cdot d \\ a &= a \operatorname{div} d \cdot d \\ \frac{a}{d} &= a \operatorname{div} d \\ \frac{a}{d} &\overset{d \neq 0}{=} \lfloor \frac{a}{d} \rfloor \end{aligned}$$

*2.1.2 Property 2.* $\forall a, d \in \mathbb{Z}: d \neq 0 \wedge a \bmod d = 0 \leftrightarrow a \operatorname{div} d = (a - sign(d)) \operatorname{div} d + 1$

Proof:

$$\begin{aligned} a \operatorname{div} d &= (a - sign(d)) \operatorname{div} d + 1 \\ \frac{a - (a \bmod d)}{d} &= (a - sign(d)) \operatorname{div} d + 1 \\ \frac{a}{d} &= (a - sign(d)) \operatorname{div} d + 1 \\ \frac{a}{d} &\overset{d \neq 0}{=} \lfloor \frac{a - sign(d)}{d} \rfloor + 1 \\ \frac{a}{d} &= \lfloor \frac{a}{d} - \frac{sign(d)}{d} \rfloor + 1 \\ \frac{a}{d} &\overset{Prop1}{=} \frac{a}{d} + \lfloor -\frac{sign(d)}{d} \rfloor + 1 \\ -1 &= \lfloor -\frac{sign(d)}{d} \rfloor \end{aligned}$$

*2.1.3 Property 3.*

$\forall a, d, i \in \mathbb{Z}$:

$a \bmod d = 0 \wedge ((d > 0 \wedge i \in [a, a + d)) \vee (d < 0 \wedge i \in (a + d, a]))$

$\leftrightarrow$

$a \operatorname{div} d = i \operatorname{div} d$

Proof:

$$\begin{aligned} a \operatorname{div} d &= i \operatorname{div} d \\ \lfloor \frac{a}{d} \rfloor &\overset{d \neq 0}{=} \lfloor \frac{i}{d} \rfloor \\ \lfloor \frac{a}{d} \rfloor &\overset{i' = i - a}{=} \lfloor \frac{i' + a}{d} \rfloor \\ \lfloor \frac{a}{d} \rfloor &= \lfloor \frac{i'}{d} + \frac{a}{d} \rfloor \\ \frac{a}{d} &\overset{Prop1}{=} \lfloor \frac{i'}{d} \rfloor + \frac{a}{d} \\ 0 &= \lfloor \frac{i'}{d} \rfloor \end{aligned}$$

$d > 0$:

$$0 \overset{0 \leq i' < d}{=} 0$$

$d < 0$:

$$0 \overset{d < i' \leq 0}{=} 0$$

# 3 SHARED OFFSET PARTITIONING

One of the transformations we introduce to eliminate modulo computations from with-loops is Shared Offset Partitioning (SOP). For this transformation, a partition that contains a modulo computation is split up into one or multiple partitions. Each partition has the modulo computation replaced by an addition of the original index vector and an offset that matches that partition.

To quickly illustrate the transformation, consider this simplified example:

```
offset = 6;
res = with {
    ([0] <= iv < [10]): mod(iv + offset, 10);
}: genarray([10], 0);
```

This code can be transformed as follows:

```
offset1 = mod(0 + 6, 10) - 0; // =  6
offset2 = mod(4 + 6, 10) - 4; // = -4
res = with {
```

```
    ([0] <= iv < [ 4]): iv + offset1;
    ([4] <= iv < [10]): iv + offset2;
}: genarray([10], 0);
```

In this example, the modulo computation is removed, and the with-loop partition is split into two new partitions spanning the same range as the original. Each new partition replaces the modulo computation with an addition computation of the index variable and an offset specific to that partition. While determining this offset still requires a modulo computation, the computation happens outside the with-loop. Hence, the modulo is computed once per partition instead of once per iteration.

## 3.1 Partition definition

For Shared Offset Partitioning, a partition is defined as a range of values $i \in \mathbb{Z}$ that, given parameters $o, d \in \mathbb{Z}$, yield the same result for $(i + o)$ div $d$, where *div* indicates floored integer division.
Hence, partitions range between $[b, b + d)$ or $(b + d, b]$ where $b \in \mathbb{Z} \mid (b + o) \bmod d = 0$

## 3.2 Partition bounds

The value of $b$, as introduced in the previous section, can be found for index $i \in \mathbb{Z}$, offset $o \in \mathbb{Z}$, divisor $d \in \mathbb{Z}$.
For $d = 0$, the quotient is always 0, meaning there is only one partition, ranging between negative and positive infinity.
For $d \neq 0$, this is done by rewriting $(b+o)$ div $d = (i+o)$ div $d$ and making use of the property $(b + o) \bmod d = 0$.

$$\begin{aligned}
(b + o) \text{ div } d &= (i + o) \text{ div } d \\
\frac{b + o - (b + o) \bmod d}{d} &= (i + o) \text{ div } d \\
\frac{b + o}{d} &= (i + o) \text{ div } d \\
\frac{b + o}{d} &= \frac{i + o - (i + o) \bmod d}{d} \\
b + o &= i + o - (i + o) \bmod d \\
b &= i - (i + o) \bmod d
\end{aligned}$$

Hence, for an arbitrary index $i$, $b$ can be computed as $i - (i + o) \bmod d$. This yields a range for partitions of $[i - (i+o) \bmod d, i - (i + o) \bmod d + d)$. Note that when $d$ is negative, the right side is lower than the left, and the bounds are swapped, leading to $(i - (i + o) \bmod d + d, i - (i + o) \bmod d]$.

## 3.3 Shared offset

A modulo computation $(i + o) \bmod d$ in a partition can be replaced with the addition computation of an index variable and a constant offset $o_n$. This offset can be found by solving for $o_n$ in $i + o_n = (i + o) \bmod d$.

$$\begin{aligned}
i + o_n &= (i + o) \bmod d \\
o_n &= (i + o) \bmod d - i \\
o_n &= i - i + o - (i + o) \text{ div } d \cdot d \\
o_n &= o - (i + o) \text{ div } d \cdot d
\end{aligned}$$

Property 3 shows that $(i+o)$ div $d$, and by extension, $o_n$ is constant within a partition. Hence, the original modulo computation can be replaced with $i + o_n$, where $o_n$ only has to be computed once for an entire partition and is easily computed as $(i + o) \bmod d - i$.

## 3.4 Visible partitions

When applying the Shared Offset Partitioning transformation to a specific dimension in a with-loop partition ranging from $[l, u)$, care must be taken not to exceed or fall short of this range. The partitions that the range $[l, u)$ can be split into are called visible partitions.
Since the partitions at $l$ and $u - 1$ can span beyond the $[l, u)$ range, the first and last partitions are truncated to fit this range. They use $l$ and $u$ as the lower and upper bound, respectively.
Sometimes the exact number of visible partitions cannot be determined at compile-time, but an upper bound can be established. While such an upper bound allows the transformation to occur, it can lead to situations where the partition bound computation exceeds $u$. This can be addressed by replacing the partition bound computations with $min$(bound computation, $u$). This can lead to the overestimated partitions ranging from $[u, u)$, but such partitions are ignored at run-time and cause no further issues.

## 3.5 Number of visible partitions

Determining an upper bound or the exact number of visible partitions that span the range of a with-loop partition in a specific dimension at compile-time can be done using different formulae.
Compile-time knowledge is denoted as $C(expression)$. For example, $C(u = d, o)$ indicates that the value of $o$ and the expression $u = d$ are known at compile-time, even if the values of $u$ and $d$ are unknown.
For with-loops, the lower bound $l$ is always at least 0. Hence, when there is no knowledge of $l$ at compile-time, $l = 0$ is the worst case. This leads to a potential overestimation of the number of visible partitions.

*3.5.1* $C(d = 0)$. When $C(d = 0)$, there is exactly one partition with infinite bounds. The visible portion of the partition gets truncated to span $[l, u)$.

*3.5.2* $C(u, d, o)$. When $C(u, d, o)$, the exact number of visible partitions can be found by computing one plus the difference between the dividends of the lower bound and upper bound values of the original partition. Exactly $C(1 + |(u - 1 + o) \text{ div } d - (l + o) \text{ div } d|)$ partitions are required. It should be noted that when $l$ is set to 0 but is not guaranteed to be zero, this formula can overestimate the number of visible partitions.

*3.5.3* $C(u = d)$. When $C(u = d)$, there are at most 2 visible partitions. Given additional knowledge that $C(o = 0)$, there is exactly 1 visible partition.

To determine this, $d$ is substituted with $u$ in the $C(u,d,o)$ case.

$$\begin{aligned}
1+|\lfloor\frac{u-1+o}{d}\rfloor-\lfloor\frac{l+o}{d}\rfloor| &\overset{d=u}{=} 1+|\lfloor\frac{u-1+o}{u}\rfloor-\lfloor\frac{l+o}{u}\rfloor| \\
&= 1+|\lfloor\frac{u}{u}+\frac{o-1}{u}\rfloor-\lfloor\frac{l+o}{u}\rfloor| \\
&= 1+|\lfloor 1+\frac{o-1}{u}\rfloor-\lfloor\frac{l+o}{u}\rfloor| \\
&= 1+|1+\lfloor\frac{o-1}{u}\rfloor-\lfloor\frac{l+o}{u}\rfloor| \\
&\overset{*}{=} 1+|1+\lfloor\frac{o'-1}{u}\rfloor-\lfloor\frac{l+o'}{u}\rfloor|
\end{aligned}$$

At $\overset{*}{=}$, $o$ is replaced with $o' = o \bmod u$. This substitution can be made in this particular formula because whenever $\frac{o}{u} = \lfloor\frac{o}{u}\rfloor$, it is both added and subtracted to the formula.
Given $0 \le l < u$, pieces of the formula can be rewritten:

$$\lfloor\frac{o'-1}{u}\rfloor = \begin{cases} -1 & \text{if } o' = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$-\lfloor\frac{l+o'}{u}\rfloor = \begin{cases} 0 & \text{if } l+o' < u \\ -1 & \text{otherwise} \end{cases}$$

Combining the equations results in

$$\lfloor\frac{o'-1}{u}\rfloor-\lfloor\frac{l+o'}{u}\rfloor = \begin{cases} -2 & \text{if } o'=0 \wedge l+o' \ge u \equiv \bot \\ -1 & \text{if } o'=0 \wedge l+o' < u \equiv o'=0 \\ & \text{or } o'>0 \wedge l+o' \ge u \\ 0 & \text{if } o'>0 \wedge l+o' < u \end{cases}$$

Substituting this back into $1+|1+\lfloor\frac{o'-1}{u}\rfloor-\lfloor\frac{l+o'}{u}\rfloor|$ yields exactly 1 visible partition if $o'=0$ or $l+o' \ge u$, and at most 2 partitions otherwise.
Since knowledge of $u$ is unavailable at compile-time in this scenario, $l+o' \ge u$ cannot be determined for arbitrary $l$ and $o$. Determining the other scenario, $o'=0$, also requires knowledge of $u$, except when $o=0$, since it implies $o'=0$ regardless of the value of $u$. Hence, for $C(u=d, o=0)$, there is exactly one visible partition, and for $C(u=d)$, there are at most two visible partitions.

*3.5.4* $C(u=-d)$. For $C(u=-d)$, a similar algebra to the $C(u=d)$ case can be applied to the $C(u,d,o)$ case. When $C(u=-d)$, there are at most two visible partitions. Given additional knowledge $C(o=1)$ or $C(o=0, l\ge 1)$, there is exactly 1 visible partition.
In this scenario, $d<0$ since $-d=u>l\ge 0$.

$$\begin{aligned}
1+|\lfloor\frac{u-1+o}{d}\rfloor-\lfloor\frac{l+o}{d}\rfloor| &= 1+|\lfloor\frac{u-1+o}{-u}\rfloor-\lfloor\frac{l+o}{-u}\rfloor| \\
&= 1+|\lfloor\frac{1-o-u}{u}\rfloor-\lfloor\frac{-o-l}{u}\rfloor| \\
&= 1+|-1+\lfloor\frac{1-o}{u}\rfloor-\lfloor\frac{-o-l}{u}\rfloor| \\
&\overset{*}{=} 1+|-1+\lfloor\frac{1-o'}{u}\rfloor-\lfloor\frac{-o'-l}{u}\rfloor|
\end{aligned}$$

At $\overset{*}{=}$, $o$ is again replaced with $o' = o \bmod u$ which is valid for the same reason as in the $C(u=d)$ case.

Pieces of the formula can be rewritten:

$$\lfloor\frac{1-o'}{u}\rfloor = \begin{cases} 0 & \text{if } 0 \le o' \le 1 \\ -1 & \text{otherwise} \end{cases}$$

$$-\lfloor\frac{-o'-l}{u}\rfloor = \begin{cases} 0 & \text{if } o'=l=0 \\ 1 & \text{if } 1 \le o'+l \le u \\ 2 & \text{otherwise} \end{cases}$$

Combining the equations results in

$$\lfloor\frac{1-o'}{u}\rfloor-\lfloor\frac{-o'-l}{u}\rfloor = \begin{cases} -1 & \text{if } \bot \\ 0 & \text{if } o'=l=0 \wedge 0 \le o' \le 1 \\ & \text{or } o'>1 \wedge 1 \le o'+l \le u \\ 1 & \text{if } 0 \le o' \le 1 \wedge 1 \le o'+l \le u \\ & \text{or } o'>1 \wedge o'+l > u \\ 2 & \text{if } 0 \le o' \le 1 \wedge o'+l > u \end{cases}$$

Given that the combined equation can only result in 0, 1, or 2, the only scenario where $1+|-1+\lfloor\frac{1-o'}{u}\rfloor-\lfloor\frac{-o'-l}{u}\rfloor|$ returns 1 is when the combined equation returns 1.
Hence, when $0 \le o' \le 1 \wedge 1 \le o'+l \le u$ or $o'>1 \wedge o'+l' > u$, it returns 1.
The second clause cannot be determined without knowledge of $u$ and is therefore unusable. The first clause, however, can be simplified further.
When $o'=0$, it yields $1 \le l \le u \equiv 1 \le l$.
When $o'=1$, it yields $1 \le 1+l \le u \equiv \top$.
Similarly, to the $C(u=d, o=0)$ case, $o=0$ implies $o'=0$. For $o=1$, however, this is slightly different. $o=1 \rightarrow o'=1 \vee d=-1$.
When $d=-1$, there is only 1 visible partition since $(i+o) \bmod -1$ is 0 for all values of $i$ and $o$. Hence, for $C(u=-d, o=1)$ and $C(u=-d, o=0, l\ge 1)$, there is exactly 1 visible partition. In all other cases, there are at most two visible partitions.

*3.5.5* $C(u,d)$. When $C(u,d)$, there is no longer enough information to determine the quotients at $l$ and $u-1$. As such, the exact number of visible partitions cannot be computed. A worst-case scenario can, however, be computed. In the worst-case scenario, a partition ends at index $l$, resulting in a 1-index partition. Accounting for the offset $o$, the highest value of $l+o$ for which the dividend does not change is $d$ if $d>0$ and $-d$ if $d<0$. Hence, $l+o = d' \mid d' = sign(d)\cdot d$.

$$\begin{aligned}
1+|\lfloor\frac{u-1+o}{d}\rfloor-\lfloor\frac{l+o}{d}\rfloor| &= 1+|\lfloor\frac{u-1+o}{d}\rfloor-\lfloor\frac{d'}{d}\rfloor| \\
&= 1+|\lfloor\frac{u-1+o}{d}\rfloor-sign(d)| \\
&= 1+|\lfloor\frac{u-1+d'-l}{d}\rfloor-sign(d)| \\
&= 1+|\lfloor\frac{u-1-l}{d}\rfloor|
\end{aligned}$$

Hence, for $C(u,d)$, an upper bound of the number of visible partitions can be computed with $1+|\lfloor\frac{u-1-l}{d}\rfloor|$.

## 4 SHARED RESULT PARTITIONING

While SOP is excellent at minimizing partitions when the upper bound is roughly as large as the absolute value of the divisor, it creates an egregious number of partitions when the divisor gets

closer to zero.
To address this, we introduce another transformation to eliminate modulo computations from with-loops: Shared Result Partitioning (SRP). In this transformation, the number of partitions created equals the absolute value of the divisor.
To illustrate the transformation, consider this example:

```
res = with {
    ([0] <= iv < [10]): mod((iv - 1) * 3 + 2, 2);
}: genarray([10], 0);
```

This code can be transformed as follows:

```
res1 = mod((0 - 1) * 3 + 2, 2); // = 1
res2 = mod((1 - 1) * 3 + 2, 2); // = 0
res = with {
    ([0] <= iv < [10] step [2]): res1;
    ([1] <= iv < [10] step [2]): res2;
}: genarray([10], 0);
```

Each partition has its lower bound offset by one compared to the previous partition, with the first partition starting at the original lower bound. All partitions have their upper bound set to the original upper bound. The step size of all newly created partitions is set to the absolute value of the divisor.
This configuration ensures that all index variables in a partition are equivalent under modulo, also known as congruent. Operations preserving congruence, namely addition, subtraction, and multiplication, can be freely applied to the index variable, provided only with-loop invariant variables are used.
As long as congruence is preserved, the modulo computation can be lifted out of the with-loop, replacing the computation with a reference to the lifted computation.

SRP serves as the counterpart to SOP: it is excellent at minimizing the partitions when the divisor is close to zero, yet creates an egregious number of partitions as the divisor gets farther from zero. When the divisor is exactly zero, however, applying SRP is unsound.

# 5 IMPLEMENTATION

The SOP and SRP transformations are implemented in `sac2c`, which is a C-based SaC compiler that compiles SaC code to C code. In `sac2c`, optimisations can be written as traversals on the abstract syntax tree (AST) that can change the nodes during the traversal. During the optimisation phase, the AST is in Static Single Assignment (SSA) form. In that form, computations are indivisible in the sense that they cannot be simplified further, and variables are only assigned a value once. Most optimisations are run in a cycle, allowing the code to become progressively more optimised. This construct allows optimisations to be mostly independent of other optimisations. To implement the transformations described in this paper, a new traversal, With-loop Modulo Partitioning (WLMP) was added to the optimisation cycle.
For a sound application of the transformations, the variables used inside a modulo operation must either be the iterator or with-loop invariant. For a complete application, all of these cases must be identified.
In the implementation, outside of iterators, only variables defined outside with-loops are used for the transformation, guaranteeing soundness due to SSA form. Completeness is guaranteed by running the With-loop Invariant Removal optimisation immediately before WLMP, which lifts all with-loop invariant variables out of their with-loops.

## 5.1 Traversal design

The WLMP traversal is split up into two subtraversals. First, there is the discovery traversal. This traversal is used to identify patterns in the AST where either SOP or SRP can be applied. When a pattern matches and enough information about the matched variables is available at compile-time, the required number of partitions for both SOP and SRP are determined. The transformation that requires the least number of partitions is selected. When the two methods result in an equal number of partitions, SOP is preferred as more optimisations can apply to partitions with step size 1 compared to other step sizes.
If the minimum number of required partitions is less than a limit set in the command line (default 2), the aplmod operation is deemed a valid candidate. The operation is flagged as such, and the discovery traversal halts discovery, folding back up to the partition that directly contains the aplmod operation.
Next, this partition is copied as often as necessary, and the copied partitions are retraversed with the adjustment traversal. The job of the adjustment traversal is to change the partition bounds and step size as necessary, as well as to find the flagged aplmod operation in each copied partition. Once found, the aplmod is replaced with the replacement computed by SOP or SRP.
With-loop invariant values that are required for replacing the bounds or the aplmod are stored in front of the current with-loop as variables to be referred in the appropriate locations.
Once all these changes have been made for all newly created partitions, the secondary partition aborts, relinquishing control to the discovery traversal. The discovery traversal then retraverses the same partition, repeating the process until no more transformable aplmod operations are found. At that point, the next partition is selected, repeating the process.

## 5.2 Pattern matching

Identifying aplmod operations to replace consists of multiple separate steps. First, on the level of the traversal, when a partition is entered, information about it is stored. Notably, whether there are 'scalar iteration variables' that track the iteration for a specific dimension. When a primitive function is encountered inside the partition, it is pattern matched against the patterns `_aplmod_SxS_(a, x)` and `_aplmod_SxS_(a + b, x)`. To consolidate the patterns, the former is converted to `_aplmod_SxS_(a + b, x)` where `b = 0` after matching.
Pattern matches for subtractions are not required since the traversal is placed in a part of the optimisation cycle where all subtractions are replaced with additions of the negation of the subtracted value. The multiplications that are valid for SRP are not implemented.
In both patterns, exactly one of the variables must equal a scalar index variable of the immediately surrounding partition. The other variable must be with-loop invariant.

If all checks up to this point pass, both SOP and SRP determine an upper bound on the required number of partitions. For SOP, this is done using the formulae detailed in section 3.5, in the order that they are listed. This guarantees the smallest upper bound that can

be determined for SOP is used. For SRP, the upper required number of partitions is exactly the absolute value of the divisor.
Out of the two transformations, the one that produces the lowest valid upper bound on the required number of partitions is selected. If neither transformation produces a valid number, no transformation is applied. In a stalemate, SOP is selected. If the required number of partitions is less than or equal to the maximum number of allowed partitions, the selected transformation is applied to the aplmod.

### 5.3 Shortcutting

When a partition uniquely refers to a code block containing a pattern-matched aplmod, the discovery traversal can directly replace the matched aplmod without copying partitions and invoking the adjustment traversal.
This shortcut is an effective way to reduce compile-times for rotates where one or multiple dimensions are 'rotated' by 0.

### 5.4 SOP Replacement offset

When SOP is applied to a partition, the shared offset for the first partition that replaces it is computed as `aplmod(l + o, d) - l`, conforming exactly to section 3.3. Subsequent shared offsets are instead computed by subtracting `abs(divisor)` from the previous shared offset.

### 5.5 Bound computations

For SOP, the bounds cannot always be computed at compile-time. Instead, expressions are created that are evaluated at run-time. In all cases, the lower bound of the first partition is set to the original lower bound, and the upper bound of the last partition is set to the original upper bound. This ensures those bounds are never exceeded.

To determine the other bounds, a recursive scheme is used.
The upper bound of a new partition is computed by adding the absolute divisor to the lower bound of that partition, or taking the original upper bound, whichever is smaller.
The lower bound is the same as the upper bound of the previous partition.
The lower bound of the first partition is not used directly as it is overwritten with the original lower bound, but is still computed to provide a base case for the recursion scheme. It is computed as follows:

```
zero_case           = toi(d == 0) * l
base_case           = toi(d != 0) * -replacement_offset
neg_adjustment      = toi(d  < 0) * (d + 1))
replacement_offset  = aplmod(l + o, d) + -o
initial_lower_bound = zero_case + base_case
                    + neg_adjustment
```

Note that `toi` converts the booleans true and false to 1 and 0, respectively.
When the divisor is zero, there is ideally only one partition ranging from the original lower and upper bounds, but the number of partitions can be overestimated. When the divisor is zero, the initial lower bound is effectively computed as `l`, which is just the lower bound. This results in the last partition ranging between the original lower and upper bound, with all other partitions having a length of 0 as they range to and from the original lower bound.

When the divisor is positive, the initial lower bound is effectively computed as `-replacement_offset`, which is the negation of the offset used in the replacement for the aplmod computation.

When the divisor is negative, the initial lower bound is effectively computed as `-replacement_offset + d + 1`. This added $d + 1$ compared to the positive case is to compute a variant of the range $(i - (i + o) \bmod d + d, i - (i + o) \bmod d]$ where the lower bound is inclusive and the upper bound is exclusive.
When these situations are dealt with for the initial lower bound, the other bounds can be built on top of it without issue.

### 5.6 SRP implementation

The SRP transformation is only partially implemented: the ability to match multiplications inside modulo operations is not present. This makes the pattern matching the exact same as for the SOP transformation.
Other aspects of SRP are fully implemented. The bounds are computed in a recursive manner, similar to SOP. The lower bound of the first partition uses the original lower bound. Subsequent partitions simply add one to the result to arrive at their lower bound. For each partition, a copy of the matched modulo operation is made and placed in front of the with-loop. The iterator variable in the copied modulo operation is replaced with the lower bound for the associated partition. The original modulo operation is removed and replaced with a reference to the copied operation. As for the upper bound and step size, those are both set to the original upper bound and the divisor for all created partitions.

## 6 EVALUATION

In this section, we try to quantify the performance impact our proposed optimisation has. We start by looking at micro-benchmarks to get an idea of the possible impact of our optimisation and then investigate more realistic benchmarks to obtain an idea of the possible real-world effects. To cover a wide range of architectures, we look at three different code generation scenarios: sequential code, multi-threaded code and GPU code.

We use three different machines for our experiments which we refer to as Intel, AMD-WSL, and AMD-GPU. The hardware and software specifications of these machines are shown in Table 1.

| | Intel | AMD-WSL | AMD-GPU |
|---|---|---|---|
| CPU | Xeon E-2378 | 7 7800X3D | EPYC 7313 |
| # Cores | 8 | 8 | 16 |
| Hyperthreading | yes | yes | yes |
| Clock Freq.[GHz] | 2.60 | 4.20 | 3.00 |
| Boost Freq. [Ghz] | 4.80 | 5.00 | 3.70 |
| Boost disabled | yes | yes | yes |
| RAM bw. [GB/s] | 2x25.6 | 2x51.2 | 8x25.6 |
| L3 size [MB] | 16 | 96 | 128 |
| L3 bw. [GB/s] | | 2500 | |
| GPU | — | — | Nvidia A30 |
| OS version | Ubuntu 22.04 | Debian 12* | Ubuntu 22.04 |
| SaC compiler | sac2c v1.3-unmerged-branch | | |
| C compiler | GCC 11.3.0 | GCC 12.2.0 | GCC 11.4.0 |

*Run in a WSL2 environment.

**Table 1: Hardware/Software configurations used.**

Where possible, we inhibit frequency scaling (boost) and run our benchmarks exclusively, i.e., without any other user-initiated processes in the background. All experiments are run ten times. We present the results as the average of our measurements with a vertical bar indicating the full range of our measurements.

## 6.1 Micro Benchmarks

We start with a simple rotation benchmark. It iteratively rotates `array` by one element per axis, which can be encoded in SaC as

```
for (i = 0; i < nr_of_rotates; i++) {
    array = {iv -> array[mod (iv + 1, shape(array))]
             |iv < shape (array)};
}
```

This benchmark aims to identify an upper bound of the possible gains from the SOP transformation. In the unoptimised case, rotating an element requires one addition and modulo operation per dimension. In the optimised version, rotating an element only requires a single addition or subtraction, regardless of the dimension. We start with `array` as a vector and then look at arrays with increasing rank to systematically increase the potential for optimisation gains.

The number of iterations is chosen in a way that ensures that we have to copy at least 50 GB so that we obtain sufficiently large runtimes. Furthermore, we ensure that neither the SaC compiler nor the C compiler fuses the iteration into a single, multi-element rotation.

Fig. 1 shows the performance on the Intel system on the left and the AMD-WSL system on the right. We use multi-threaded executions, varying the number of threads used from 1 to 16. Since the optimised version has only one addition or subtraction per axis as an element, we depict the effective memory transfer bandwidth as a performance measure. To do so, we assume that each rotation requires exactly two transfers per array element, one read and one write.

Looking at the Intel architecture in Fig. 1a first, we make the following observations. For executions with a single thread, we see in the 1D case an improvement from roughly 9.6 GB/s to 13.1 GB/s, equating to a factor 1.4 improvement. When looking at higher ranks, this factor increases to about 2.6 in the 5D case. When looking at higher levels of parallelism, these improvements shrink as the optimised versions never improve beyond 23.0 GB/s. Even for the unoptimised cases, this limit is hit when the number of threads is increased. Considering the optimised code, this does not come as a surprise, given that the architecture has a peak bandwidth of 25.6 GB/s per memory controller and performing up to 5 integer operations per 2 integer transfers clearly is memory bound.

On the AMD-WSL architecture in Fig. 1b, we see a similar picture. Here we have a speedup ranging from 1.7 in the 1D case to 4.8 in the 5D case. The scaling limit is reached at 40.3 GB/s which, again, reasonably matches the RAM bandwidth of this architecture.

To avoid the RAM bandwidth being the limiting factor, we repeat the experiment of rotating arrays, but now we operate on arrays that are small enough to fit the level 3 cache. The level 3 cache of the AMD-WSL machine in particular has a peak bandwidth of 2.5 TB which should avoid the memory operation being the primary bottleneck.

The results of this experiment are shown in Fig. 2b. Here, we see a very different picture. Both architectures are no longer limited by their respective RAM bandwidths and scale beyond that. Even in the single-threaded case, we already have performance improvements between a factor of 2.5 (1D case) and a factor of 3.3 (4D case) on the Intel architecture, and factors between 8.1 (1D case) and 13.7 (5D case) on AMD-WSL. As the number of threads increases, the throughput increase is almost linear on both architectures. However, we see some degradation towards higher thread counts on AMD-WSL which we attribute to the fact that the overall bandwidth of the cache is much higher than that of the Intel machine allowing for other hardware effects to affect the overall runtime.

From these experiments, we can see that the impact of the proposed optimisation on isolated rotations crucially depends on the memory bandwidth. If the memory bandwidth is too low, the additional divisions are hidden behind the memory latency, diminishing the performance gains. This raises the question of what happens if we look at slightly more complex operations that require some more computation per element.

## 6.2 Towards Real-World Benchmarks

As a slightly more realistic benchmark, we look at an iteration over a 5-point stencil operation on rank two arrays with cyclic boundary conditions. In SaC, we can express these as:

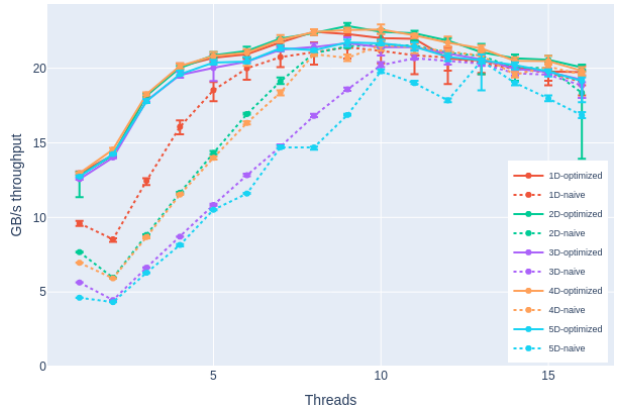
```
for (i = 0; i < nr_of_stencils; i++) {
    grid = {iv -> sum({ov -> stencil[ov]
                           * rotate (1-ov, grid)[iv]})
            |iv < shape (grid)};
}
```

where `grid` denotes a rank two array of integers and `stencil` is a matrix of shape `[3,3]` with 5 non-zero values. For our experiment, we chose a stencil with five distinct non-zero non-one values to ensure our additional stencil operations cannot be statically optimised away. For the optimised version, this leads to a total of five multiplications and four additions per element for the stencil computation, on top of the four index modifications due to the rotate. When compiled without the optimisation enabled, another 5 aplmod computations are required.

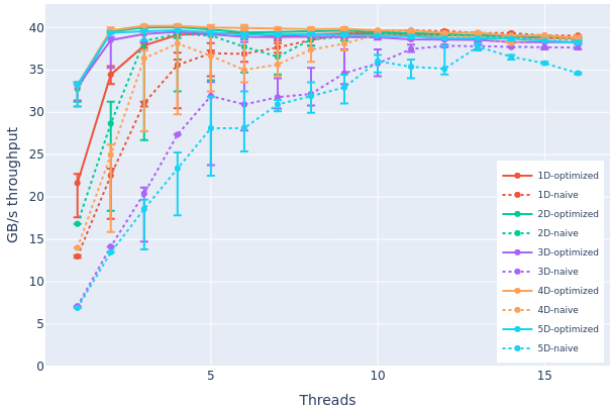
Fig. 3a displays the throughput for the 2D rotate and 5pt-stencil on the Intel machine with 1GB integer arrays. While the 5-point stencil effectively consists of 5 rotates, the optimised 5-point stencil is only 14% slower than the 2D rotate in a single-threaded scenario. Meanwhile, the naive version of the 5-point stencil is 2.6 times slower than the 2D rotate.

The fact that both optimised versions are roughly as fast as each other while the naive versions differ a lot makes it so the optimisation is a lot more effective for the 5-point stencil. Specifically, the optimisation improves the bandwidth for rotate by a factor of 1.7 and improves the 5-point stencil by a factor of 3.8 for the 5-point stencil in a single-threaded scenario. For the multi-threaded scenario, the numbers converge as memory bandwidth becomes the bottleneck.

On the AMD-WSL machine in Fig. 3b, the same single-threaded effects can be observed, albeit much less pronounced. The 2D rotate program sees an improvement of a factor of 2.0, and the 5-point stencil sees a similar improvement at a factor of 3.1.



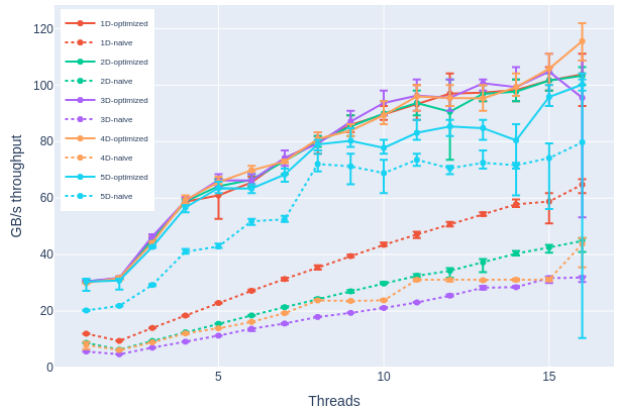

**(a) Intel**



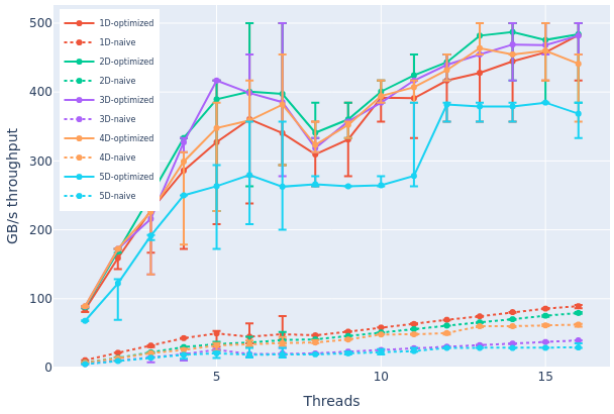

**(b) AMD-WSL**

**Figure 1: The rotate benchmarks applied to integer arrays of a size of roughly 1GB running on the Intel and AMD-WSL systems, using 1-16 threads. The chosen iterations and shapes are:**

| Name | Iterations | Shape |
|---|---|---|
| 1D | 25 | [268435456] |
| 2D | 25 | [16384, 16384] |
| 3D | 25 | [645, 645, 645] |
| 4D | 25 | [128, 128, 128, 128] |
| 5D | 24 | [49, 49, 49, 49, 49] |




**(a) Intel**



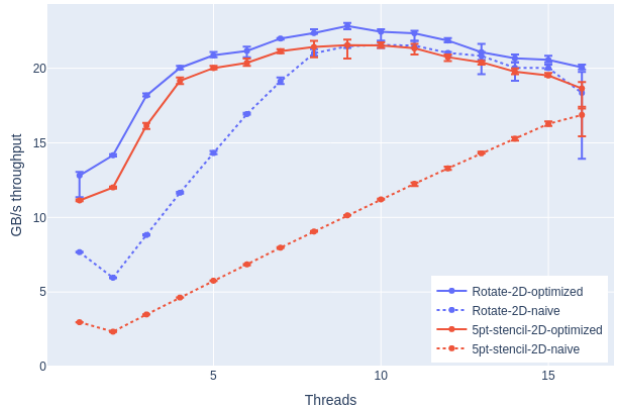

**(b) AMD-WSL**

**Figure 2: The rotate benchmarks applied to integer arrays of a size of roughly 4MB for Intel and 24MB for AMD-WSL, using 1-16 threads. The array sizes are chosen to populate half the L3 cache on each machine. The chosen iterations and shapes are:**

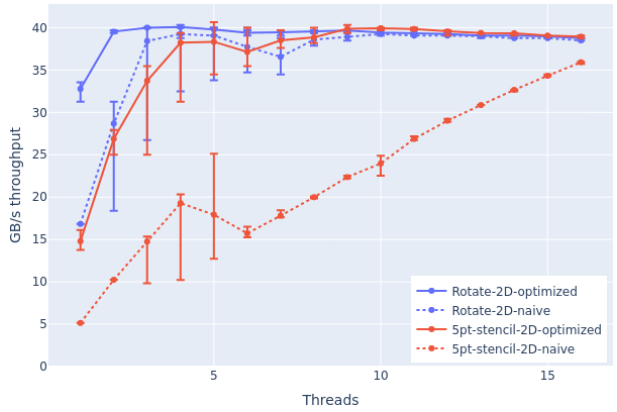
| | Intel | | AMD-WSL | |
|---|---|---|---|---|
| Name | Iterations | Shape | Iterations | Shape |
| 1D | 6400 | [1048576] | 1067 | [6291456] |
| 2D | 6400 | [1024,1024] | 1067 | [2508, 2508] |
| 3D | 6324 | [102,102,102] | 1060 | [185, 185, 185] |
| 4D | 6400 | [32,32,32,32] | 1074 | [50, 50, 50, 50] |
| 5D | 6400 | [16,16,16,16,16] | 1043 | [23,23,23,23,23] |

The multi-threaded scenario again converges as the memory bandwidth becomes the bottleneck.

Like the experiment comparing rotations over arrays of different dimensions, this one is also rerun with array sizes that fit in the L3 cache of Intel and AMD-WSL systems. Fig. 4a shows the results of this experiment for Intel. Again, we observe that the optimised rotate and optimised 5-point stencil reach nearly the same bandwidth, even as the number of threads increases. For the single-threaded

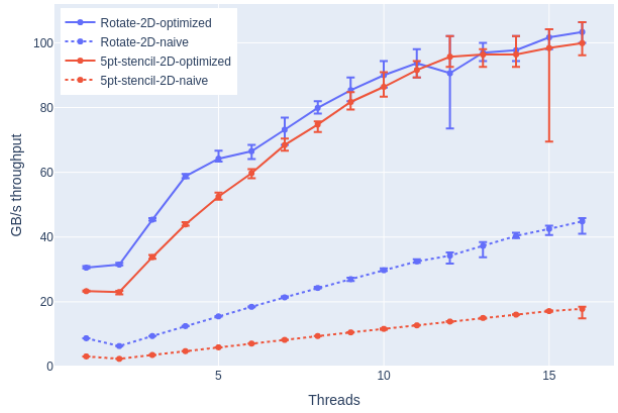



(a) Intel

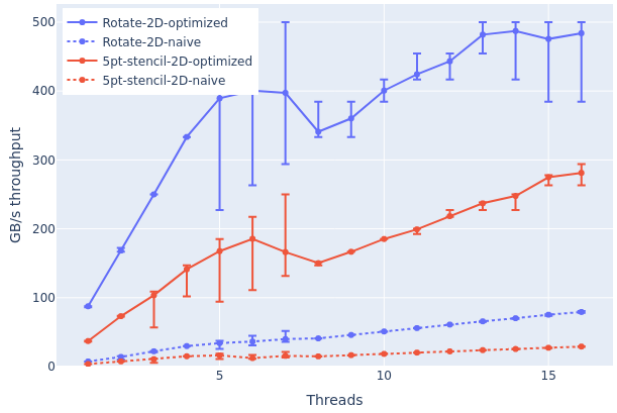



(b) AMD-WSL

**Figure 3: The rotate and stencil benchmarks applied to 2-dimensional integer arrays with a size of 1GB and shape [16384, 16384]. The respective operations were applied on 25 iterations.**




(a) Intel




(b) AMD-WSL

**Figure 4: The rotate and stencil benchmarks applied to 2-dimensional integer arrays of a size of roughly 4MB for Intel and 24MB for AMD-WSL, using 1-16 threads. The array sizes are chosen to populate half the L3 cache on each machine. For intel, the number of iterations is 6400, and the shape is [1024, 1024]. For AMD-WSL, the number of iterations is 1067, and the shape is [2508,2508]**

case, the optimised version of rotate has a 3.5 times larger throughput than the unoptimised version, and the optimised 5-point stencil sees an improvement of 7.5 times over the unoptimised version. The factor of improvement remains fairly constant as the number of threads is increased.

The figure for AMD-WSL, Fig.4b now clearly tells a different story than Intel. Even in the multithreaded case, the optimised 2D rotate is around 2.3 times faster than the optimised 5-point stencil. Single-threaded, the optimised 2D rotate and 5-point stencil benchmarks are 11.7 and 9.8 times faster than their unoptimised counterparts. The relative improvement stays mostly consistent regardless of the number of threads.

## 7 RELATED WORK

The work presented here is most closely related to a class of optimisations from classical compiler construction, referred-to as *strength reduction* [4]. The key idea of strength reduction is to replace computationally expensive operations with less expensive ones. This rather vague classification, over time, has led to the inclusion of many different techniques under the umbrella of the name strength reduction. A classic example of strength reduction is the replacement of a multiplication with an iterator within a loop by successive additions. A lot of work has been put into the identification of such opportunities while dealing with possible constraints that result from the surrounding control- or data-flow [4, 6, 7].

A different form of strength reduction that is more closely related to our work focuses on eliminating remainder and modulo

operations. In [5], the authors suggest a technique based on keeping track of the operands and inserting adequate conditionals into the loop bodies. This work is refined by [9]. While the aim of this work is similar to ours, we partition the iteration space rather than introducing conditionals. Furthermore, our optimisation operates on index spaces with no inherent execution order whereas the work of Cocke and Markstein requires a sequential loop execution.

Another approach is put forward in [10]. It is based on the observation that on some systems multiplications are less expensive than divisions and it strives to replace divisions by multiplications with the inverse of the divisor.

Yet another line of work in the context of strength reduction that focuses on modulo operations is presented in [15]. It builds on the idea of combining *loop partitioning* [2], a technique typically used in the context of parallelising compilers [16], with the handling and eventual elimination of modulo operations. As it turns out, the basic idea is very similar to the transformation idea presented in this paper. However, in contrast to our work, which is based on data-parallel loops with guaranteed absence of side-effects and data-dependencies, it is embedded into the setting of conventional, inherently sequential loops. Furthermore, it cannot benefit from the range guarantees that the APL-style modulo operations provide that are at the core of this work.

Finally, this work relates to the large body of work on optimisations in code generators for high-level languages such as those in [8, 12–14]. All these works have in common that they demonstrate the need for optimisations on a higher level of abstraction than the level of the C-code that they generate. We are not aware that any of these high-level optimisations tackle the particular case of modulo operations.

## 8 CONCLUSION

Cyclic indexing can be elegantly expressed using a modulo operation. While elegant, it typically comes with a hefty performance cost. A more involved way to express cyclic indexing eliminates this cost but is error-prone due to added complexity and does not naturally scale for higher dimensions.

In this paper, we propose and implement a compiler optimisation to convert modulo operations in loops into an optimised version, requiring minimal compile-time information. In particular, this works for modulo-based array indexing as soon as the divisor is known to equal the upper bound, and the dimensionality is known at compile-time. Additionally, all variables in the modulo operation other than the iterator must be loop invariant. The optimisation takes the complexity out of the hands of the programmer, allowing them to take full advantage of the expressive power of modulo without losing any performance compared to a traditionally more performant yet complex version.

If all compilers in the underlying toolchain of SaC 2c were to implement the optimisation, the implementation in sac2c would likely remain beneficial. The sac2c compiler tends to have more compile-time knowledge about SaC programs than the C and Cuda compilers that sac2c can compile to. This knowledge would allow sac2c to optimise more situations than the other compilers. Additionally, the optimisation benefits or outright enables other optimisations like with-loop folding, which improves the ability to parallelise code.

To determine the performance gain of the optimisation, two types of benchmarks are run. In the first benchmark, an array varying between 1 and 5 dimensions is rotated. For the other benchmark, a 5-point stencil is applied to a 2-dimensional array. For both benchmarks, when memory throughput is not a bottleneck, the performance increase caused by the optimisation is close to constant w.r.t. the number of cores used. Across benchmarks, the factor of speedup ranges around $5 \pm 2.5$ for Intel and $10 \pm 2.5$ for AMD-WSL, respectively.

For future work, our work can be combined with the work of Sheldon et al. [15] to provide a complete analysis of all strength reductions that can be applied to modulo and integer division with different rounding behaviours. Such work, with a focus on requiring minimal compile-time knowledge, could serve as a catalogue for strength reductions applicable to modulo and integer divisions in the context of loops and array indexing. This would allow compilers to adopt the optimisations, regardless of the rounding used for the division of their modulo implementation, and even with limited compile-time information.

## REFERENCES


[1] Patrick Bahr and Graham Hutton. 2015. Calculating correct compilers. *Journal of Functional Programming* 25 (2015), e14. https://doi.org/10.1017/S0956796815000180

[2] Utpal K. Banerjee. 1993. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, USA.

[3] Edwin Brady. 2017. *Type-driven development with Idris.* Simon and Schuster.

[4] John Cocke and Ken Kennedy. 1977. An Algorithm for Reduction of Operator Strength. *Commun. ACM* 20, 11 (nov 1977), 850–856. https://doi.org/10.1145/359863.359888

[5] John Cocke and Peter W Markstein. 1980. Strength reduction for division and modulo with application to accessing a multilevel store. *IBM Journal of Research and Development* 24, 6 (1980), 692–694.

[6] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. 2001. Operator Strength Reduction. *ACM Trans. Program. Lang. Syst.* 23, 5 (sep 2001), 603–625. https://doi.org/10.1145/504709.504710

[7] D.M. Dhamdhere. 1989. A new algorithm for composite hoisting and strength reduction optimisation. *International Journal of Computer Mathematics* 27, 1 (1989), 1–14. https://doi.org/10.1080/00207168808803702

[8] Franz Franchetti, Tze-Meng Low, Thom Popovici, Richard Veras, Daniele G. Spampinato, Jeremy Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proceedings of the IEEE, special issue on "From High Level Specification to High Performance Code"* 106, 11 (2018).

[9] Cédric Ghez, Miguel Miranda, Arnout Vandecappelle, Francky Catthoor, and Diederik Verkest. 2000. Systematic high-level address code transformations for piece-wise linear indexing: illustration on a medical imaging algorithm. In *2000 IEEE Workshop on SiGNAL PROCESSING SYSTEMS. SiPS 2000. Design and Implementation (Cat. No. 00TH8528).* IEEE, Lafayette, Louisiana, 603–612.

[10] Torbjörn Granlund and Peter L Montgomery. 1994. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation.* ACM, Orlando, Florida, USA, 61–72.

[11] Clemens Grelck and Sven-Bodo Scholz. 2003. Sac — from High-level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters* 13, 3 (2003), 401–412. https://doi.org/10.1142/S0129626403001379

[12] Clemens Grelck and Sven-Bodo Scholz. 2003. Sac — from High-level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters* 13, 3 (2003), 401–412. https://doi.org/10.1142/S0129626403001379

[13] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. 2018. Modular Acceleration: Tricky Cases of Functional High-performance Computing. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (St. Louis, MO, USA) *(FHPC 2018).* ACM, New York, NY, USA, 10–21. https://doi.org/10.1145/3264738.3264740

[14] Trevor L. McDonell, Manuel M T Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. In *ICFP '13: The 18th ACM SIGPLAN International Conference on Functional Programming.* ACM.

[15] Jeffrey Sheldon, Walter Lee, Ben Greenwald, and Saman Amarasinghe. 2001. Strength reduction of integer division and modulo operations. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, Cumberland Falls, KY, USA, 254–273.

[16] Michael Joseph Wolfe, Carter Shanklin, and Leda Ortega. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., USA.