On Mapping N-Dimensional Data-Parallelism Efficiently into GPU-Thread-Spaces

Niek Janssen
niek.janssen@student.ru.nl
Radboud University
Nijmegen, Netherlands

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

ABSTRACT

Data-Parallelism on multi-dimensional arrays can be conveniently specified by mapping element-computations to all elements of n-dimensional index-spaces. This paper proposes a small set of combinators for mapping multi-dimensional index spaces into multi-dimensional thread-index spaces suitable for execution on GPUs. For each combinator, we provide an inverse operation, that allows the original indices to be recovered within the individual threads. This setup allows arbitrary n-dimensional array computations to be executed on GPUs with arbitrary thread-space constraints as long as the overall resources required for the computation do not exceed those provided by the GPU.

KEYWORDS

SaC, CUDA, GPU, array transformations, functional languages

ACM Reference Format:
Niek Janssen and Sven-Bodo Scholz. 2021. On Mapping N-Dimensional Data-Parallelism Efficiently into GPU-Thread-Spaces. In *Proceedings of IFL 21: ACM Symposium on implementation and application of functional languages (IFL 21).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

For many performance hungry application areas such as scientific computations, financial applications, image processing or AI, algorithms can be conveniently expressed as data-parallel operations on n-dimensional arrays. Languages such as Futhark[10], Accelerate[6], Lift[16], or SaC [15] have demonstrated that such algorithms can be mapped into very efficient GPU codes. However, obtaining high-performance for a wide variety of such array algorithms poses many challenges. Amongst these are decisions about data representations in memory, memory allocation strategies, memory transfer optimisations, decisions about which parts to actually compute in parallel, as well as how to map these parallel computations onto the hardware.

This paper focusses on the last of the challenges mentioned above. We assume that we have a given computation that needs to be executed in parallel for a set of indices into an n-dimensional index space and that each such instance of the computation requires the n-element index it relates to. The challenge now is to come up with a thread-space that can be readily executed on a given GPU.

Given that GPUs support up to 6-dimensional thread-spaces, one might be inclined to consider this a non-issue: often a one-to-one correspondence between index-space and thread-space might do. Unfortunately, there are several issues with this approach. First of all, GPUs have constraints on the possible thread ranges and, secondly, the maximally possible thread-space dimensionality is fixed.

A better alternative is to flatten the index space completely and then cut this index as required by a given GPU's hardware requirements. The drawback of this approach is that recovering the original indices can be costly, in particular, when dealing with higher-dimensional arrays.

In this paper, we propose a more flexible approach. We identify a set of combinators for mapping n-dimensional index spaces into m-dimensional thread-spaces. Each of these combinators comes with a dedicated inverse mapping which allows to recover the original indices. That way, either the programmer can annotate a desired mapping or the compiler can infer a mapping so that the thread-space is as close as possible to the original index-space reducing costly index-recovery-operations to a minimum.

The individual contributions of the paper are

- we propose a set of combinators for transforming index-spaces
- we outline an implementation in the context of the compiler infra-structure for SaC, and
- we discuss a few strategies for attributing given computations with suitable strategies for given GPU targets

2 SAC

SaC (Single assignment C) [3] is a functional programming language build around multidimensional arrays. Each array in SaC consists of the array *data* and the *shape* of the array. The data part contains the array elements, while the shape part contains information about the dimensionality and the lengths of those dimensions. The shape can be accessed through a built-in function `shape`, and is a one-dimensional array itself.

In listing 1, we will discuss some exapmles and facts about SaC arrays.

```
// Declare array x
x = [[1, 2, 3], [4, 5, 6]];

// The shape of x can be requested
// with the shape function
```


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

```
print(shape(x));
// [2, 3]

// Note that the dimensionality of x
// is equal to the length of shape(x)
print(length(shape(x)));
// 2

// A new array can also be generated
// using the Stdlib function genarray.
// genarray takes a shape and a default
// element
print(genarray([2, 3], 42);
// [[42, 42, 42], [42, 42, 42]]

// Array elements can be accessed using
// index vectors:
iv = [1,1];
print(x[iv]);
// 5
```

Listing 1: Some SaC examples and facts

2.1 The with-loop

At the heart of the SaC programming language is the with-loop. A with-loop can be compared to the *map* function in the map-reduce computation model, although the with-loop is more flexible. A with-loop creates a new array using a given shape, and fills it with data in the manner defined in its body. A with loop consists of roughly three parts:

- A default array `x`. The shape of this array will be taken as the shape of the result and its elements are used as default elements.
- An index space S, defining which indices `iv` of array `x` have to be replaced with new values.
- A body, containing an expression which computes the new value at any given index vector `iv`.

A pair of an index space and a body is called a *partition*. A with loop can have multiple partitions, as shown in listing 2.

```
a = with {
    (index-space) : { body-expression };
    (index-space) : { body-expression };
} : default-array;
```

Listing 2: The schematic of a SaC with-loop

An index space is defined using a quadruple of four shape vectors (`L, U, T, W`), and a variable identifier. In the body expression, this variable contains the current index vector. The four shape vectors define which indexes are mapped in a partition. Using `iv` as variable identifier, we can define the constraints on the index space:

- Lower bound L (optional): `L <= iv`, restrict to indexes greater or equal to L
- Upper bound U: `iv < U`, restrict to indexes strictly smaller then U
- Step T and Width W (optional): `iv % T < W`, Every `T` elements, take `W` elements. If `W` is omitted, a vector of suitable length with all values being `1` is assumed.

In case of overlapping partitions, the semantics of SaC guarantees a top to bottom execution. Taking all this together, we can define a with-loop like this:

```
a = with {
    ([0, 1] <= iv < [9, 8]
        step [2, 3] width [1, 2])
        : 3;
    ([1, 0] <= iv < [8, 9]
        step [3, 2] width [2, 1])
        : 7;
} : genarray ([9, 9], 0);
```

Listing 3: An example usage of a SaC with-loop

Which creates a 2-dimensional array with the following output:

```
0 3 3 0 3 3 0 3 0
7 0 7 0 7 0 7 0 7
7 3 7 0 7 3 7 3 7
0 0 0 0 0 0 0 0 0
7 3 7 0 7 3 7 3 7
7 0 7 0 7 0 7 0 7
0 3 3 0 3 3 0 3 0
7 0 7 0 7 0 7 0 7
0 3 3 0 3 3 0 3 0
```

The formal semantics of the with-loops including further variants of with loops for modifying arrays, reducing arrays, and performing non-deterministic operations on unique objects can be found elsewhere [14][11].

2.2 With-loop execution

We can conceptualize the execution of a partition in a with-loop in a few subsequent steps. Firstly, we have an index space descriptor as a tuple of the four shape vectors L, U, T, W as defined above. Secondly, we have an index generator that computes all index vectors inside this index space descriptor. We call this set of index vectors an index space. Thirdly, we have the partition body, which transforms each index into a value. These values are then put in the resulting array on the correct positions.

Of course, this is only an abstract way of describing a with loop. When generating target-specific optimised code, the current SaC compiler `sac2c` takes special care when arranging the order in which indices are being generated and for some targets it also takes the liberty to interleave different partitions in order to improve locality (see [8][7] for example).

```
Index space descriptor
(lb, ub, step, width)

     |
     |  SaC Index generator
     V

Index space
({[10, 10], [10, 11], ...})
```

```
    |
    |  Partition body expression:
    |  37
    V

  Values
```

It is important to note that the compiler `sac2c` re-arranges with-loops massively during optimisation, ensuring that eventually there are no overlapping index sets within any with-loop and trying to ensure that further partitions are added to cover the entire legal index space, whenever possible. New partitions are typically being added when all partitions have a lower bound, upper bound, step or width and the compiler cannot be sure that all space has been covered by those partitions. We will discuss an example of such a with loop.

Example of a with-loop where multiple partitions spawned. All numbers below are indexes on the x and y axis.

```
With loop size: [7,7]
L = [1,1]
U = [6,6]
T = [3,2]
W = [2,1]

00 01 02 03 04 05 06
10 11 12 13 14 15 16
20 21 22 23 24 25 26
30 31 32 33 34 35 36
40 41 42 43 44 45 46
50 51 52 53 54 55 56
60 61 62 63 64 65 66
```

Firstly, the original (explicit) partition will cover it's own index space. Secondly, there will be *four* partitions covering the space above the upper bound and below the lower bound:

```
.. .. .. .. .. .. ..    00 01 02 03 04 05 06
.. 11 12 .. 14 15 ..    10 .. .. .. .. .. 16
.. .. .. .. .. .. ..    20 .. .. .. .. .. 26
.. 31 32 .. 34 35 ..    30 .. .. .. .. .. 36
.. .. .. .. .. .. ..    40 .. .. .. .. .. 46
.. 51 52 .. 54 55 ..    50 .. .. .. .. .. 56
.. .. .. .. .. .. ..    60 61 62 63 64 65 66
```

And lastly there will be two partitions covering the space between the step and width:

```
.. .. .. .. .. .. ..    .. .. .. .. .. .. ..
.. .. .. .. .. .. ..    .. .. .. 13 .. .. ..
.. 21 22 23 24 25 ..    .. .. .. .. .. .. ..
.. .. .. .. .. .. ..    .. .. .. 33 .. .. ..
.. 41 42 43 44 45 ..    .. .. .. .. .. .. ..
.. .. .. .. .. .. ..    .. .. .. 53 .. .. ..
.. .. .. .. .. .. ..    .. .. .. .. .. .. ..
```

So in the end, for this with-loop with seemingly one partition, 7 partitions are actually created.

2.3 CUDA and SaC

The CUDA programming model is very similar to the programming model of a with-loop. Similarly to a with-loop, CUDA performs operations on elements of an index space which we will refer to as *thread-space*. Where a with-loop uses a body with an expression, CUDA uses a function with arguments and a return value, but in effect a CUDA and a SaC operation uses the same concept: they execute a piece of code for each element of a certain index space.

In principle, because the programming models are so similar, it should be easy to utilise CUDA to execute with-loops on the GPU. We could just give the information about the index space to a CUDA thread space generator, wrap the expression body in a function, and let CUDA execute this function for each index vector.

```
Index space
(lb, ub, step, width)

     |
     |  CUDA Index generator
     V

Index/Thread space
({[10, 10], [10, 11], ...})

     |
     |  Partition body function:
     |  \iv -> 42
     V

  Values
```

The current implementation of the SaC compiler contains a mechanism to generate CUDA code, created by Jing Guo [9]. This implementation follows the approach described above. Currently, roughly the following steps are implemented to make this work:

- Identify with-loops eligible for Cuda kernel execution: Not all with-loops can be implemented using CUDA. CUDA cannot handle function calls to external functions for example, so all with-loops containing such function calls cannot be implemented using CUDA.
- Insert memory transfer primitives: GPU memory is separate from CPU memory, so necessary data has to be transferred to and from the GPU before and after the computations respectively.
- Create kernel functions: The creation of the functions wrapping the partition body expressions.
- The CUDA Index generator is given the index space information and it creates the kernel functions are run for each index in the index space.

3 LIMITATIONS ON CUDA AND HOW TO SOLVE THEM

Though the implementation created by Jing's works very well in many cases, there are still cases where it either performs rather poorly or does not work at all. The reason for this is fairly simple: SaC index spaces are more flexible and can handle cases which

CUDA's thread spaces simply cannot. To make matters more challenging, the concrete restrictions on a CUDA thread space can vary with each hardware device. Generally, older CUDA hardware has more restrictive constraints then newer devices, but this is not always the case.

The differences between SaC and CUDA index spaces can be summarized into four catagories:

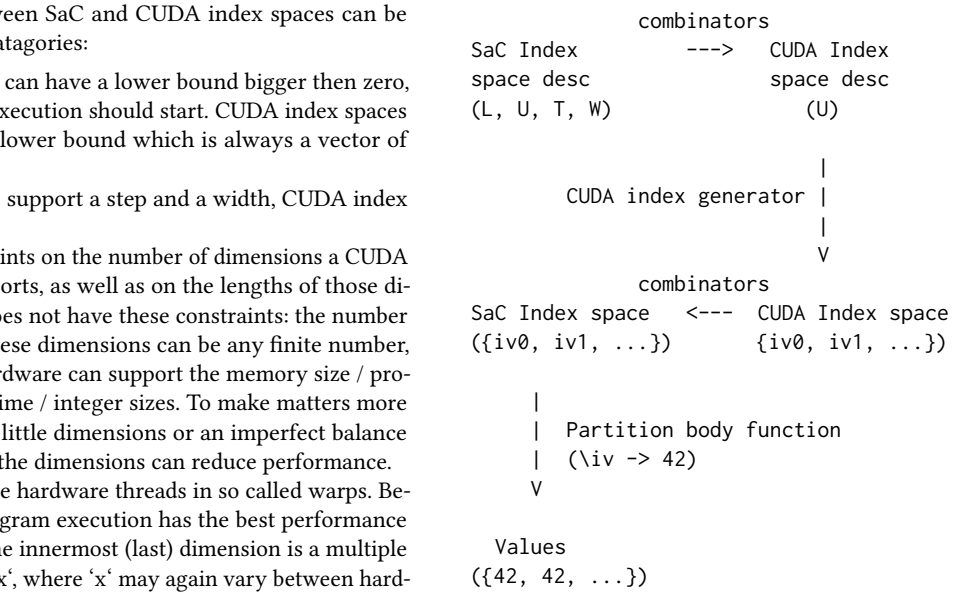
- SaC index spaces can have a lower bound bigger then zero, from where the execution should start. CUDA index spaces have an implicit lower bound which is always a vector of zeros.
- SaC index spaces support a step and a width, CUDA index spaces do not.
- There are constraints on the number of dimensions a CUDA index space supports, as well as on the lengths of those dimensions. SaC does not have these constraints: the number and lengths of these dimensions can be any finite number, as long as the hardware can support the memory size / program execution time / integer sizes. To make matters more complicated, too little dimensions or an imperfect balance of the lengths of the dimensions can reduce performance.
- CUDA spawns the hardware threads in so called warps. Because of this, program execution has the best performance if the length of the innermost (last) dimension is a multiple of the warp size 'x', where 'x' may again vary between hardware devices. This is not really a hard constraint, but the performance penalty is big enough to treat it as such.

The current implementation deals with some of these issues in a very crude way. The lower bound is handled perfectly, but the step and width are handled by masking the executing threads, effectively sacrificing parallelism. The constraints on the dimensionality and lengths of dimensions is by far the most challenging problem to solve, and the current implementation is based on some target-hardware-specific heuristics and, in some cases, cannot produce a working solution at all. Finally, the warp size constraint is fulfilled only when the dimensionality is at most two. This is, however, more of a side effect of how threads are spawned in these cases, then a real intended effect.

Though constraints on dimensionality, lengths of dimensions and warp sizes may vary across hardware devices, there are also a lot of similarities. First of all, the dimensionality is split between 3 block dimensions and 2 or 3 grid dimensions. The block dimensions can handle only smaller lengths, and the product of the block dimensions has got a constraint too. On a GTX 1060, for example, the product of the block dimensions has a limit of 1024. The grid dimensions can handle bigger lengths, but this may also vary per dimension.

3.1 Proposed solution

The general idea to solve this problem is that we are going to transform the original SaC index space into an index space that fits into the CUDA architecture. Before we give the index space descriptor (the tuple of L, U, T, W) to the index generator, we transform it to a dense thread space which can be described by only an upper bound U that fits the CUDA index space requirements. Then, in the partition body function, CUDA provides the grid and block coordinates. From these coordinates, we recompute an index in the original SaC index space. After both transformations have been completed, exactly the same indexes should be generated as in the original CPU implementation of the index generator allowing us to use the very some body code.



Because we have to accommodate for the broad spectrum of with-loops SaC supports, it is nearly impossible to implement one single transformation to handle all cases. Things are even trickier when trying to come up with a mapping that yields good performance for any given architecture.

In order to allow for experimentation and possible auto-tuning, we propose a set of combinators for transforming index spaces. These can be attributed to individual with-loops either by means of pragma annotations of the programmer, or by some form of inference mechanism through the compiler or a runtime system.

Each combinator by itself comes with two transformations which are inverse to each other, guaranteeing a bijection between the index space and the thread space.

4 INDEX SPACE TRANSFORMATIONS

In this section, we introduce the combinators for transforming the index spaces into thread spaces suitable for GPUs. We start with an informal description, motivate their need, and discuss the expected performance impact due to index transformations at runtime.

Note that each combinator is only responsible for generating code in two places: one in the host function just before the threads are spawned, and one in the kernel function executed on the GPU. As the latter is performed by all threads we focus on the code in the kernel function when we discuss performance cost.

4.1 *ShiftLB*

This combinator shifts the index space by the lower bound and thus makes the thread space start from index `0` within all dimensions. In doing so, we reflect the restriction of CUDA's thread spaces to always start with `0`. The overhead for regaining the original indices of this mapping amount to one scalar addition in each dimension.

While a more generic shift operation would be possible without incurring any immediate additional overhead, it turns out that restricting the shift to the lower bound is not only sufficient in all cases but also desirable: it guarantees the normalisation of the lower bound to `0`s which simplifies most of the other combinators significantly.

Example index space transformation with ShiftLB. Here with the index space `L = [1, 1], U = [6,6], T = [1,2], W = [1,1]` on the left and the thread space with `L = [0,0] U = [5, 5], T = [2,2], W = [1,1]` on the right:

```
11 .. 13 .. 15      00 .. 02 .. 04
21 .. 23 .. 25      10 .. 12 .. 14
31 .. 33 .. 35      20 .. 22 .. 24
41 .. 43 .. 45 ---> 30 .. 32 .. 34
51 .. 53 .. 55      40 .. 42 .. 44
```

The original indices can be recovered by subtracting `1` in each dimension.

4.2 *CompressGrid*

This combinator condenses grids into thread spaces that can be described without step (`T`) and width (`W`). It reflects CUDA's lack of a notion of non-dense grids.

To make the inverse mappings for recovering the original indices simpler, we require the original index space to have a lower bound of all `0`s. Note here, that *ShiftLB* can be used to achieve this.

CompressGrid obtains an extra parameter which needs to be of the same length as the dimensionality of the index space. All entries of this parameter either need to be `1` or `0` depending on whether the corresponding dimension is to be compressed or not.

Example index space transformation with CompressGrid (`[1,0]`). Here with the index space `L = [0, 0], U = [5,5], T = [2,2], W = [1,1]` on the left and the thread space with `L = [0,0] U = [3, 5], T = [1,2], W = [1,1]` on the right:

```
00 .. 02 .. 04       00 .. 02 .. 04
.. .. .. .. ..       10 .. 12 .. 14
20 .. 22 .. 24  ---> 20 .. 22 .. 24
.. .. .. .. ..
40 .. 42 .. 44
```

A compression with parameter `[1,1]` would affect both dimensions and yield a thread space `L = [0,0] U = [3, 3], T = [1,1], W = [1,1]`:

```
00 .. 02 .. 04       00 01 02
.. .. .. .. ..       10 11 12
20 .. 22 .. 24  ---> 20 21 22
.. .. .. .. ..
40 .. 42 .. 44
```

Non-surprisingly, index recovery is dimension-specific and only applies to those dimensions whose corresponding compression parameter is `1`. The method of recovering the original indexes depends on the value of `W` in the dimension of concern. If `W`, as in the example, is `1`, a multiplication with step value (here `2`) suffices.

However, if a width other than `1` is present, we need to uncompress the indexes in blocks the size of the width. Consider the following example: given an index space `L = [0, 0], U = [5,5], T = [3,1], W = [2,1]` on the left and we apply *CompressGrid* (`[1,0]`). This yields a thread space `L = [0, 0], U = [4,5], T = [1,1], W = [1,1]`:

```
00 01 02 03 04       00 01 02 03 04
10 11 12 13 14       10 11 12 13 14
.. .. .. .. ..  ---> 20 21 22 23 24
30 31 32 33 34       30 31 32 33 34
40 41 42 43 44
```

The recovery of the original index in the outermost dimension now requires several operations as we need to identify the sub-block and the position within that sub-block independently. In general, we have:

```
idx_i = (thread_i / width_i) * step_i
        + id_i % width_i;
```

In the example that translates into

```
idx_0 = (thread_0 / 2) * 3 + thread_0 % 2;
```

yielding `idx_0 == 3` for `thread_0 == 2`.

Note here, that we do not *require* any dimensions to be compressed. While this avoids the costly recovery cost of the original indices, it entails that some of the spawned CUDA threads will simply not contribute to the result. Consequently, this trades parallelism for compute complexity. Most likely, compression is only beneficial whenever the grid density is low enough that the gain in parallelism outweighs the index recovery overhead.

4.3 *FoldLast2*

FoldLast2 allows the two innermost (rightmost) dimensions to be folded into a single dimension. This is a functionality needed to reduce the overall dimensionality of the index set. It caters to CUDA's limitation to a maximum number of dimensions supported by the hardware. Current GPUs typically support a maximum of 6 dimensions. Similar to *CompressGrid*, *FoldLast2* requires the lower bounds to be all `0`s. Additionally, it requires the steps and widths to be all 1s, and the dimensionality of the index space to be at least 2.

Example index space transformation with FoldLast2. Given an index space with `L = [0, 0], U = [2,5], T = [1,1], W = [1,1]`, an application of *FoldLast2* yields a thread space `L = [0], U = [10], T = [1], W = [1]`:

```
00 01 02 03 04  ---> 0 1 2 3 4 5 6 7 8 9
10 11 12 13 14
```

Recovering the original indices only affects the innermost dimension of the thread-space. We can regain the first of the two folded indices by dividing the thread index by the innermost index shape (here `5`). The second index requires a modulo operation with the same value.

4.4 *SplitLast*

Dual to *FoldLast2*, *SplitLast* allows the innermost dimension to be split up into two dimensions. This is needed to cater for situations where there is just one dimension or where the last dimension is too big for a given GPU to handle. This combinator requires a parameter indicating the shape of the new innermost dimension.

Again, we demand the lower bounds to be all `0`s and the steps and width to be all 1s.

In contrast to *FoldLast*2, there is no guarantee that the dimension to be slit divides by the given parameter. In case it does not, the index recovery needs to identify the threads that relate to no indices of the original index space.

Example index space transformation with SplitLast (4). Given an index space with L = [0], U = [10], T = [1], W = [1] and an application of *SplitLast* (4), we obtain a thread space `L = [0, 0], U = [3,4], T = [1,1], W = [1,1]` with the last two threads being disabled:

```
0 1 2 3 4 5 6 7 8 9
  |
  V
00 01 02 03
10 11 12 13
20 21 .. ..
```

In principle, the original index can be recovered by multiplying the first of the new dimension's thread number with the parameter (here 4) and adding the second thread number. However, in general, we finally have to check against the original shape (here 10). This allows us to detect the two disabled threads.

4.5 *PadLast*

Even in situations where a change in dimensionality is not needed it sometimes is advisable to adjust the overall shape. A typical example for such a use case is the innermost dimension of the thread space. GPUs typically operate in so-called *warps*[1]. These are fixed numbers of hardware units that are scheduled simultaneously. It usually is advisable to choose the innermost dimension as a multiple of these. *PadLast* allows us to do exactly this. This combinator takes a number as parameter and adjusts the innermost shape to a multiple of that size. *PadLast* does not require any values for lower bound, step or width.

Example index space transformation with PadLast (4). Given an index space with L = [0,0], U = [5,7], T = [1,1], W = [1,1] and an application of *PadLast* (4), we obtain a thread space `L = [0, 0], U = [5,8], T = [1,1], W = [1,1]` with the last thread in the innermost dimension being disabled:

```
00 01 02 03 04 05 06      00 01 02 03 04 05 06 07 ..
10 11 12 13 14 15 16      10 11 12 13 14 15 16 17 ..
20 21 22 23 24 25 26      20 21 22 23 24 25 26 27 ..
30 31 32 33 34 35 36 ---> 30 31 32 33 34 35 36 37 ..
40 41 42 43 44 45 46      40 41 42 43 44 45 46 47 ..
```

For *PadLast*, the original index space is retained by a checking of the innermost thread index against the original shape.

4.6 *Permute*

Our final combinator is *Permute*. It requires a vector parameter which describes the desired permutation. This vector needs to contain the numbers of all dimensions of the index space to be mapped starting from `0`. The position of each number indicates which dimension in the thread space contains that number's original dimension's indices.

Being able to permute index spaces arbitrarily is a very powerful transformation. Not only does it allow is to apply other combinators such as *SplitLast* or *FoldLast*2 to arbitrary dimensions and, thus, adjust any any dimension if needed, it also allows optimisations due to memory access pattern such as coalescing [1].

Example index space transformation with Permute (`[1,0]`). Given an index space with L = [0,0], U = [5,7], T = [1,1], W = [1,1] and an application of *Permute* (`[1,0]`), we obtain a thread space `L = [0, 0], U = [8,5], T = [1,1], W = [1,1]`

```
00 01 02 03 04 05 06      00 01 02 03 04
10 11 12 13 14 15 16      10 11 12 13 14
20 21 22 23 24 25 26      20 21 22 23 24
30 31 32 33 34 35 36 ---> 30 31 32 33 34
40 41 42 43 44 45 46      40 41 42 43 44
                          50 51 52 53 54
                          60 61 62 63 64
```

The most interesting aspect of this transformation is that it comes virtually at no overhead. All that is required is a permutation of thread indices which can be implemented statically by the code generator.

5 THE SAC COMPILER

In this section, we outline those component's of the existing SaC compiler `sac2c` [3] that pertain to the execution on GPUs.

The SAC compiler takes any SAC program, transpiles it into C code, and calls a C compiler to compile it into machine code. In case of code generation for GPUs the compiler produces CUDA code and calls NVIDEA's C compiler `nvcc`.

5.1 CUDA kernel creation

When the compiler compiles a SaC program for the CUDA backend, it will do so using a few different steps. There are, among others, a step that determines what loops are to be converted to CUDA code, steps that insert the correct memory transfers to get arrays to and from the GPU, and steps that generate the CUDA kernel. For our problem, we are interested in the step where the CUDA kernels are generated. More information about the other steps involved can be found in Jing's paper of the original implementation [9].

We will now discuss how a with loop is converted to CUDA code in the old version of the compiler. We will use the running example as specified in listing 4.

```
// Let there be an array a
b = with {
  ([0] <= iv < [1.000] step [2]) : a[iv] + 1;
  ([1.000] <= iv < [1.500]) : a[iv] + 4;
} : genarray ([1.500], 0);
```

Listing 4: An example with-loop with two partitions

As we discussed in section 2.2, there will be an additional partition generated to fill the gaps created by the step. This is illustrated in listing 5.

```
// Let there be an array a
b = with {
  ([0] <= iv < [1.000] step [2]) : a[iv] + 1;
```

```
  ([1] <= iv < [1.000] step [2]) : 0;
  ([1.000] <= iv < [1.500]) : a[iv] + 4;
} : genarray ([1.500], 0);
```

Listing 5: Implicitly generated partitions are added to the with-loop

The SaC compiler will then generate kernels and calls to those kernels for those three partitions. Each kernel function will contain the body of the partition it was created for. Variables containing the lower bound, upper bound, step and width will be passed in as an argument. All GPU pointers to SaC arrays needed to execute the body are passed in as arguments as well. The current iv can be derived from the variables threadIdx and blockIdx, which both contain the properties x, y and z. Listing 6 shows what code will be created for the first partition.

```
// Host CPU code:
// Launch kernel for partition 0
dim3 grid = ...
dim3 block = ...

kernel_0<grid, block>(
  1, 1.000, 2, 1, a, b);

...

void kernel_0 (
    lb_0, ub_0, st_0, wi_0, a, b) {
  // Recompute iv from
  // threadIdx and blockIdx
  iv = ...

  b_d[iv] = a_d[iv] + 1;
}
```

Listing 6: The SaC compiler will generate CUDA kernels for each partition

5.2 Grid and block

In listing 6, there are a few lines of code left undefined. One of those open spaces is the definition of the grid and block. In this section, we will discuss the implementation of the grid and block computation. The original implementation is pretty straightforward. Let us define the lengths of the dimensions as $d_0 ... d_n$. d_0 is the outermost dimension, and d_n is the innermost one. Now we make a case distinction for the first five dimensionalities, and map them as described in table 1. For 6 or more dimensions, the current implementation gives a compiler error. Note that each GPU has a maximum block size. Because we do not know the shapes of the arrays in SAC at compile time, the implementation for dimensionalities 3-5 may throw a runtime error when this maximum block size is exceeded.

This implementation, however, only uses the lengths of the dimensions. A SaC index space is defined by a lower bound, upper bound, step and width. Jing solved this by substracting the lower bound from the upper bound, exactly as we do in our *ShiftLB* combinator. The step and width do not change the size of the dimensions at all, in Jing's solution. For our three partitions, the grid and block computation is shown in listing 7

d	grid.z	grid.y	grid.x	blk.y	blk.x
1	1	1	$d_0/32+1$	1	32
2	1	$d_0/32+1$	$d_1/32+1$	32	32
3	1	1	d_0	d_1	d_2
4	1	d_0	d_1	d_2	d_3
5	d_0	d_1	d_2	d_3	d_4

Table 1: Case distinction for mapping n-dimensional index spaces onto the GPU, using Jing's heuristics

```
// Host CPU code:
// Launch kernel for partition 0
// We take the upper bound, devide it by 32,
// and add a padding of 1 to make sure no
// indices are lost.
dim3 grid = dim3 (1, 1, 1.000 / 32 + 1);
dim3 block = dim3 (1, 1, 32);

kernel_0<grid, block>(
  0, 1.000, 2, 1, a, b);

// Launch kernel for partition 1
// The lowerbound is not 0, so we have to
// substract it.
dim3 grid = dim3 (1, 1, (1.000 - 1) / 32 + 1);
dim3 block = dim3 (1, 1, 32);

kernel_1<grid, block>(
  1, 1.000, 2, 1, a, b);

// Launch kernel for partition 2
dim3 grid = dim3 (1, 1, (1.500 - 1.000)
      / 32 + 1);
dim3 block = dim3 (1, 1, 32);

kernel_2<grid, block>(
  1.000, 1.500, 1, 1, a, b);
```

Listing 7: The grid and block are computed for all three partitions

5.3 Index recovery

Inside the kernel, the original iv is reconstructed from the `threadIdx` and `blockIdx` variables. For dimensionalities 3 to 5, we can directly take the values of `blockIdx.z`, `blockIdx.y`, `blockIdx.x`, `threadIdx.y` and `threadIdx.x`. However, for dimensionalities 1 and 2, `iv_0` and `iv_1` have to be recomputed, and because the dimension length has been padded to be divisible by 32, we also have to check it against the original dimension length.

The old implementation handles the lower bound the same way as our *ShiftLB* mapping. The step and width are removed with an if statement, similar to the if-statement used to prune excess elements after splitting them over `block.x` and `grid.x`. After combining all of this together, we can define our index recoveries for the three partitions in our example. The generated code for this can be seen in listing 8.

```
// Partition 0, inside the kernel
// Recompute iv from
// threadIdx and blockIdx
iv_0 = threadIdx.x + gridIdx.x * 32;
// Remove excess indices outside of grid
if (iv_0 % st_0 > 0) return;
// Remove excess indices above upper bound
if (iv_0 >= 1.000) return;

// Partition 1, inside the kernel
lb_0, ub_0, st_0, wi_0, a, b) {
// Recompute iv from
// threadIdx and blockIdx
iv_0 = threadIdx.x + gridIdx.x * 32;
// Remove excess indices outside of grid
if (iv_0 % st_0 > 0) return;
// Increase iv by the lowerbound again
iv_0 += 1;
// Remove excess indices above upper bound
if (iv_0 >= 1.000) return;

// Partition 2, inside the kernel
// Recompute iv from
// threadIdx and blockIdx
iv_0 = threadIdx.x + gridIdx.x * 32;
// There is no grid here, so the grid check
// can be omitted
// Increase iv by the lowerbound again
iv_0 += 1.000;
// Remove excess indices above upper bound
if (iv_0 >= 1.500) return;
```

Listing 8: Index recovery of the three partitions from example 4

6 COMBINATOR IMPLEMENTATIONS

The goal of our combinators is to replace the implementation given in sections 5.2 and **??**. In section 7, we will discuss how we will determine what combinators will get executed, and in what order. In this section, we will assume this information is all readily available as a of combinator applications with their parameters. As all preprocessing has already been done, we only have to change the code generation at two parts of the compiler: the point where the index space is computed as 3d grid/block shapes, and the point where the SaC index vector is recomputed from the thread coordinates.

Because we are dealing with code on two different levels, we will try to distinguish them as much as possible. When talking about code, data structures, or algorithms in the compiler, we will talk about compiler level. Any outputted code will be referred to as application level.

6.1 *Gen* and *GridBlock*

All the mappings we defined before take an index space, and result in an index space. However, we need a way to retrieve the index space for the first mapping, and split the dimensions of the resulting index space into a grid and block dimensions. To do this, we introduce two new combinators: *Gen* takes no arguments, and returns the original index space for the current partition. It is used as the innermost function for the nesting of combinators. *GridBlock* takes the index space as a result of the outermost combinator, and the number of block dimensions as an integer parameter. It splits the dimensions between block and grid dimensions, and creates the block and grid variables in the application code.

6.2 Generated code

Until now, we have considered the lower bound, upper bound, step, width and index vectors to be vectors. At the application level however, we will represent them as individual integer variables. At the compiler level we will still have vectors, but they will contain the variable names (as strings) of those individual variables at application level. This means that for any dimensionality n, we will have $4n$ or $5n$ variables at application level. It is not always $5n$, because the index vector does not exist when we are computing the index space before the kernel is started.

To illustrate this, we will use the example in listing 9. The statement #pragma gpukernel precedes the nesting of function calls.

```
// Our example is 2-dimensional, with unknown
// dimension lengths.
int[.,.] plusone(int[.,.] a) {
  b = with {
    // #pragma gpukernel precedes the nexting
    // of combinator applications. The
    // innermost combinator is always Gen,
    // the outermost is always GridBlock.
    #pragma gpukernel
        GridBlock (1,
        SplitLast(32,
        ShiftLB (
        Gen)))
    (0 <= iv < shape(a)) : a[iv] + 1;
  } : genarray (shape(a), 0);
  return b;
}
```

Listing 9: Example with loop, with a pragma to specify the combinators to be executed

The SaC code in listing 9 will generate the C code in listing 10. In the comments we will specify the compiler state if applicable. The nesting of combinator applications will be called twice: once for

the computation of the grid and block variables, and one (inversely) for the recomputation of the index vector.

```
int* plusone (int* a, /* shape info */) {
  ...
  // pragma Gen
  // Does not generate any code, but loads
  // the compiler state to:
  lb1 = shapeinfo_a_lb1;
  lb0 = shapeinfo_a_lb0;
  ub1 = shapeinfo_a_ub1;
  ub0 = shapeinfo_a_ub0;
  // lb: ["lb1", "lb0"]
  // ub: ["ub1", "ub0"]
  // ...

  // pragma ShiftLB
  ub1 = ub1 - lb1;
  ub0 = ub0 - lb0;
  // lb: ["0", "0"]
  // ub: ["ub1", "ub0"]

  // pragma SplitLast
  ub2 = ub0 / 32;
  ub0 = 32;
  // lb: ["0", "0", "0"]
  // ub: ["ub1", "ub2", "ub0"]

  // pragma GridBlock
  dim3 grid  = dim3 (0, ub1, ub2);
  dim3 block = dim3 (0, 0, ub0);

  // Start CUDA kernel
  ...
}

int* plusone_kernel(
        ub1, ub0, lb1, lb0, ...) {
  // pragma GridBlock
  iv0 = blockIdx.x;
  iv1 = gridIdx.x;
  iv2 = gridIdx.y;
  // lb: ["0", "0", "0"]
  // ub: ["ub1", "ub2", "ub0"]
  // iv: ["iv2", "iv1", "iv0"]

  // pragma SplitLast
  iv0 = iv2 * 32 + iv0;
  // lb: ["0", "0"]
  // ub: ["ub1", "ub0"]
  // iv: ["iv1", "iv0"]

  // pragma ShiftLB
  iv1 = iv1 + lb1;
  iv0 = iv0 + lb0;
  // lb: ["lb1", "lb0"]
  // ub: ["ub1", "ub0"]
  // iv: ["iv1", "iv0"]

  // pragma Gen
  iv = [iv1, iv0];
  ...
}
```

Listing 10: Code generated for the with-loop in example 9

6.3 Non-destructive upper bounds

In the example in listing 10, all operations can be classified as one of three operations: either replace variables with constants at compiler level, create new variables at compiler *and* application level, or modify variables at application level. For the upper bound, however, this may not always be appropriate behavior. Let us take, for example, following nested application of combinators:

```
#pragma gpukernel GridBlock(1,
  PadLast(64, ShiftLB(Gen)))
```

In this case, in the application code, *PadLast* will need to increase the upperbound to make it a multiple of 64 before the kernel launch, and check each thread for whether it is inside the original index space, or inside the extra padded space:

```
...
// Before kernel launch
ub0 = ub0 + (64 - ub0 % 64) % 64;
...
// Inside kernel
if (iv0 >= ub0) return;
...
```

This, however, does not work because of two reasons. First of all, ub0 has been overwritten by the new and padded upper bound. When the if-statement is executed, we need the upper bound *after* the *ShiftLB*, but *before* the *PadLast* has been executed. Secondly, inside the kernel, we are in a completely new function. We won't have access to the variables in the host function, before the kernel launch, anymore.

We can resolve the first problem by creating new variables every time we update the upperbound. At compiler level, we can cache the older variable names, so we can refer back to the upper bound at any stage in the index space computation.

The second problem is a bit harder to fix. The optimal solution would be to create an array of integers, and use this as the cache described above. When we launch the kernel, we move the this array to GPU memory and pass it to the kernel function. This way, the kernel has the same information as the host function. However, in the current state of the SaC compiler, it is difficult to pass this extra argument to the kernel function.

Until we can add such a variable to pass it to the kernel function, we recompute all steps of the upperbound inside each kernel function. This will create some extra overhead. However, this solution

works well and is sufficiently efficient until we can change the compiler.

7 DECIDING WHICH COMBINATORS TO EXECUTE

Providing the index space transformations as a set of combinators that can be applied allows the code generation choices to be externalised from the code generation process. As outlined in the previous section this is implemented through pragmas allowing programmers to enforce index space transformations in any way they see fit. At the same time the use of pragmas already suggests that the overall intention is to make these choices optional. In case no pragmas are present, we need to find ways to generate the pragmas we ultimately want to use. Many ways of inferring the pragmas can be envisioned: these could be heuristics based inferences or some form of auto-tuning, being it offline or online.

For the time being, we implement three different strategies for inferring these pragmas in case they are absent from a given program: *Jing*, *JingExt*, and *FoldAll*.

7.1 Jing

This strategy mimics the behaviour of the original back-end as implemented by Jing Guo and described in Section **??**. As can be seen from Table **??**, the combinators used here depend on the dimensionality of the index space. In detail we have:

- **1D** *GridBlock* `(` `1`, *SplitLast* `(` `32`, *ShiftLB* `(` *Gen*`)))`, i.e., split into blocks of `32` and generate a 1D grid of these blocks.
- **2D** *GridBlock* `(` `2`, *Permute* `(` `[0,2,1,3]`, *SplitLast* `(` `32`, *Permute* `(` `[1,2,0]`, *SplitLast* `(` `32`, *ShiftLB* `(` *Gen*`))))))`, i.e., split the innermost dimension by `32`, then bring the outermost dimension to the back; split that dimension by `32` as well, then swap the two dimensions in the middle so that we end up with two inner dimensions of `32` on the right hand side. Finally *GridBlock* turns the inner dimensions into 2D blocks of size `32`x`32`, while the outer two dimensions serve as a 2D grid.
- **3D, 4D, 5D** *GridBlock* `(` `2`, *ShiftLB* `(` *Gen*`))`, i.e., all these are taking the innermost two dimensions as block sizes and the outermost dimensions for the grid.

Once the index space is higher-dimensional than 5, the code is not executed on the GPU anymore. Notice here as well that no grid compression is ever done.

7.2 JingExt

This strategy is almost identical to the previous one. The key difference is that this method is able to handle more then 5 dimensions. There are still no safeguards against the lenghts of dimensions not fitting inside the GPU limitations. JingExt is a recursive strategy, defined by the following two rules:

- If the dimensionality is at most 5, we perform Jing's method
- If the dimensionality is more then 5, we fold all pairs of neighbouring dimensions together using a combination of *Permute* and *FoldLast*2 mappings. If the number of dimensions is uneven, we leave the innermost dimension intact. After this, we perform JingExt again.

The goal of this method is to allow the usage of Jing's method in more cases. While this method is not entirely safe to use in all cases, it uses a minimal amount of combinators to map onto the GPU.

7.3 FoldAll

This method is a brute force method intended to deal with all cases even reasonably well, assuming that no shape information is statically available. This method starts, as the name suggests, by folding all dimensions together into one. After that, this dimension is split again, using a predetermined pattern. This pattern is dependent on the specific GPU device and the estimated size of the array. The current size estimation is very crude, and uses the dimensionality as an estimate. Once better size estimations are available, they can be used to determine an optimal pattern to split the dimension.

8 RELATED WORK

Here we will look at other projects focusing on generating GPU code. We will quickly discuss their focus, and their strategies for mapping arrays or vectors onto the GPU.

Futhark [10] is a functional language, specialized on multicore execution. They mainly use OpenCL as a backend, but they recently start CUDA as well [5]. The focus of their project lies on *moderate flattening*, which means they flatten nested maps/loops, while restraining themselves to parallelism that is cheap to access. Code that cannot efficiently be parallelized, is turned into fast sequential code.

Our understanding of the Futhark project is that they do not try to retain the shape of the original computations. They use the thread blocks only to spawn threads in warps, and use only the grid x dimension to spawn their threads.

One feat they do exploit is *size inference*. With the help of size inference, they can get an estimate of the order of size of a certain array/computation. They use this in their compiler to determine where moderate flattening should be applied. In the case of SaC, it may be interesting see if we could use these methods to improve our pragma generation algorithms.

Lift/SkelCL [16] is an extension of the Scala language [4], commonly used in big data computations. They use OpenCL as their backend. They provide a set of skeleton functions like map, zip and reduce as higher level functions, which the programmer can use in code they want to be executed on the GPU. Furthermore, they provide a set of rewrite rules to obtain a faster execution time. These rewrite rules may, for example, combine two kernels into one.

Lift/SkelCL operates on two data types: vector and matrix. This means they only support one and two dimensional arrays. This means that they do not face the same challenges as SAC does when trying to map multi-dimensional index spaces onto a GPU. Our understanding is that they use a similar approach to futhark, only spawning blocks of exactly the warp size, and then using the grid to specify the problem space.

Accellerate [6] is an extension of the Haskell language [2]. They use CUDA for their backend. Like SaC, they support multi-dimensional arrays. Accellerate distincts itself by generating the CUDA kernels at runtime, instead of at compile time. This means that it can optimize the implementation for the current specific hardware. They generate these kernels using skeleton functions, which get instantiated with the correct parameters whenever they are needed. These instantiations are cached, so when a part of CUDA code has to be run again with different data, they can re-use the previously generated kernel. In SAC, optimizing kernels at runtime would be beneficial as well, as we can then optimize the kernel for a certain shape. However, this would prevent us from kernel-reusing, unless the shapes of the consequent input arrays are very similar. To our knowledge, Accellerate does also not make use of the thread and block y and z variables.

9 CONCLUSIONS

This paper offers a small set of combinators for transforming n-dimensional index spaces. Their design is tailored to satisfy the needs when mapping n-dimensional rectangular grids into dense index spaces suitable for thread creation on GPUs. We define the transformations, explain how to regain the original indices, and we show how we implemented them in the context of the current SaC compiler. We also demonstrate how the heuristics-based implementation that pre-existed can be modelled using our combinators.

The implementation of the combinators as pragmas rather than a fixed series of transformation that is hardwired to the compiler allows for a high-level manipulations of the code generation process similar to the strategies in Halide [13] or what Elevator offers for Rise [12].

We also introduce a simple strategy for inferring suitable mappings for any given SaC source code. Further work could look into inferences based on some target hardware specific cost models or even dynamic auto-tuning approaches.

REFERENCES

[1] URL https://developer.nvidia.com/accelerated-computing-training.

[2] URL https://www.haskell.org/.

[3] URL https://www.sac-home.org/.

[4] URL https://www.scala-lang.org.

[5] J. S. Bertelsen. Implementing a cuda backend for futhark. 2019.

[6] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, page 3–14, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304863. doi: 10.1145/1926354.1926358. URL https://doi.org/10.1145/1926354.1926358.

[7] C. Grelck. Shared memory multiprocessor support for functional array processing in sac. *Journal of Functional Programming*, 15(3):353–401, 2005. doi: 10.1017/S0956796805005538. URL smmsffapis.pdf.

[8] C. Grelck, D. Kreye, and S.-B. Scholz. On code generation for multi-generator with-loops in sac. In P. Koopman and C. Clack, editors, *Implementation of Functional Languages, 11th International Workshop (IFL'99), Lochem, The Netherlands, Selected Papers*, volume 1868 of *Lecture Notes in Computer Science*, pages 77–94. Springer, 2000. doi: 10.1007/10722298_5. URL sac2c-codegen-WL-lochem-99.pdf.

[9] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the gpu programming barrier with the auto-parallelising sac compiler. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 15–24, 2011.

[10] T. Henriksen. *Design and Implementation of the Futhark Programming Language*. PhD thesis, Department of Computer Science, Faculty of Science, University of Copenhagen, 2017.

[11] S. Herhut, S.-B. Scholz, and C. Grelck. Controllling chaos — on safe side-effects in data-parallel operations. In M. Chakravarty and L. Peterson, editors, *4th Workshop on Declarative Aspects of Multicore Programming (DAMP'09), Savannah, USA*, pages 59–67. ACM Press, 2009. ISBN 978-1-60558-417-1. doi: 10.1145/1481839.1481847. URL 2009_5.pdf.

[12] T. Koehler and M. Steuwer. Towards a domain-extensible compiler: Optimizing an image processing pipeline on mobile cpus. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 27–38, 2021. doi: 10.1109/CGO51591.2021.9370337.

[13] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, Dec. 2017. ISSN 0001-0782. doi: 10.1145/3150211. URL https://doi.org/10.1145/3150211.

[14] S.-B. Scholz. Single assignment c — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003. doi: 10.1.1.138.6995. URL SACESFHLAOIAFS.pdf.

[15] S.-B. Scholz. Single assignment c: Efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, Nov. 2003. ISSN 0956-7968. doi: 10.1017/S0956796802004458. URL https://doi.org/10.1017/S0956796802004458.

[16] M. Steuwer. *Improving programmability and performance portability on many-core processors*. PhD thesis, University of Münster, 2015. URL https://www.lift-project.org/publications/2015/steuwer15phdthesis.pdf.