Radboud University Nijmegen

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

Extended version

Niek Janssen
niek.janssen@student.ru.nl

Sven-Bodo Scholz
SvenBodo.Scholz@ru.nl

December 5, 2021

Contents

Chapter 1

Context

1.1 SAC

SAC (Single assignment C) [1] is a functional programming language build around multidimensional arrays. Each such an array 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 the built-in shape function, and is a one-dimensional array itself.

Some examples 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
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 built-in 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
// the index vector:
iv = [1,1];
print(x[iv]);
// 5
```

The with-loop

At the heart of the SAC programming language is the with-loop. A with-loop can be compared with the 'map' function in the map-reduce computation model, although the with-loop is a bit more advanced. A with-loop creates a new array using a given shape, and fills it with data in the manner defined in it's 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. If an index vector `iv` exists for which the with-loop body does not define a value, the value `x[iv]` will be used instead.
- An index space `S`, defining which indexes in array `x` have to be replaced with new values.
- A body, containing an expression which is used to compute the new value at index vector `iv`.

A pair of an index space and a body is called a "partition." A with loop can have multiple partitions:

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

An index space is defined using a tuple 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 constrain what indexes are mapped in a partition. Using `iv` as variable identifier, we can define the constraints on the index space:

- Lowerbound L (optional): `L <= iv`, restrict to indexes greater or equal to L
- Upperbound 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, `W = 1`.

Combining this all together, we may create 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);
```

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

```
0 3 3 0 3 3 0 3 0
2 0 2 0 2 0 2 0 2
2 3 2 0 2 3 2 3 2
0 0 0 0 0 0 0 0 0
2 3 2 0 2 3 2 3 2
2 0 2 0 2 0 2 0 2
0 3 3 0 3 3 0 3 0
2 0 2 0 2 0 2 0 2
0 3 3 0 3 3 0 3 0
```

With-loop execution

We can conceptualize the execution of a partition in a with-loop in a few consequent 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. SAC will optimize the execution of the with-loop to use the least amount of CPU-cycles and memory.

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

      |
      |  SAC Index generator
      V

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

      |
      |  Partition body expression:
      |  37
      V

  Values
```

1.2 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. 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 utilize CUDA to execute with-loops on the GPU. Because the GPU architecture is in theory faster for these kinds of tasks then the CPU architecture, we should see significant performance improvements. We could just give the information about the index space to a CUDA index 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 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 J. Guo [2]. This implementation follows the approach described above. Currently, roughly the following steps are impelemented to make this work:

- Identify with-loops eligible for cudarizing: Not all with-loops can be impelemented 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 transfered 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.

CUDA index spaces

A big problem with this approach is the limitations of the CUDA index spaces. The SAC with-loop index spaces have no constraints on dimensionality or dimension lengths other that they have to be finite. SAC index spaces also support a step and a width. CUDA index spaces for a GTX 1060 for example, can handle at most 6 dimensions, with the product of the lengths of the innermost 3 dimensions at most containing 1024 elements. Furthermore, these constraints can be different across the generations of hardware, and they do not support a step and width at all.

The current implementation deals with some of these issues, but not all of them. Furthermore, the current solution is very pragmatic and will have to be replaced with a more sophisticated mechanism with the flexibility to deal with index spaces of all shapes and sizes. In this thesis, we are going to design, implement and test this more sophisticated solution.

The general idea is that we are going to create transformations that 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 will have to transform it to only an upperbound U that fits the CUDA index space requirements. Then, in the partition body function, we have to map the individual indexes of the transformed index space back to indexes from the original 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.

Because we have to accommodate for the broad spectrum of with-loops we have to support, it is nearly impossible to implement one single transformation to handle all cases. Instead, we are going to break the problem down into smaller pieces, and provide transformations for all of them. These transformations can then be combined together to fit each partition's needs.

```
                transform
SAC Index          --->    CUDA Index
space desc                 space desc
(L, U, T, W)                  (U)


                                |
        CUDA index generator    |
                                |
                                V

               transform
SAC Index space    <---  CUDA Index space
({iv0, iv1, ...})        {iv0, iv1, ...})


     |
     |  Partition body function
     |  (\iv -> 42)
     V


  Values
({42, 42, ...})
```

Because with-loops come in many different shapes, sizes, steps and widths, we will need to have a very flexible way of defining these transformations. The easiest way to do that, is to create multiple mappings that can be combined together. In the next chapter, we will formalize the with-loop index generation and the mappings we can use to transform it.

1.3 CUDA limits and solutions

In this section, we will discuss the limitations and the mappings to handle these limitations.

CUDA index spaces do not support lowerbounds.

To handle this, we introduce the *ShiftLB* mapping. This mapping shifts the index space so that the lowerbound is a vector of 0's. In the partition body function, the lowerbound is added to each individual index again. The cost of this mapping is expected to be very low.

Example index space transformation with `L = [1, 1]`:

```
00 01 02 03 04 05       11 12 13 14 15
10 11 12 13 14 15       21 22 23 24 25
20 21 22 23 24 25       31 32 33 34 35
30 31 32 33 34 35 ---> 41 42 43 44 45
40 41 42 43 44 45       51 52 53 54 55
50 51 52 53 54 55
```

CUDA index spaces do not support step/width.

To handle this, we introduce two alternative mappings: *PruneGrid* and *CompressGrid. These mappings can only be executed on index spaces without a lowerbound.*

PruneGrid leaves the upperbound intact, but checks in each partition body function call whether the current index vector is inside the grid. If it is, the function can continue it's execution, but if it's not it is returned. This can potentially result in a lot of unnecessary GPU threads, but other then that the overhead is expected to be low.

Example index space transformation with
`T = [3, 2], W = [2, 1]`:
(.. means the function for this index vector will be returned)

```
00 01 02 03 04 05      00 01 .. 03 04 ..
10 11 12 13 14 15      .. .. .. .. .. ..
20 21 22 23 24 25      20 21 .. 23 24 ..
30 31 32 33 34 35 ---> .. .. .. .. .. ..
40 41 42 43 44 45      40 41 .. 43 44 ..
50 51 52 53 54 55      .. .. .. .. .. ..
```

The step/width mechanism creates regularly spaced n-dimensional blocks with indexes for which the body has to be executed, separated by indexes for which the body is *not* executed. *CompressGrid* compresses the index space by placing all n-dimensional blocks of threads to be executed directly beside each other. As a result, no unnecessary GPU threads are created, but the overhead of shifting those blocks may be a bit higher. Note that no actual data is moved, just the indexes are modified. Additionally, *CompressGrid* gets an extra argument C specifying which dimensions should be compressed. This way, it is possible to compress some dimensions while pruning the others.

Example index space transformation with
`T = [3, 2], W = [2, 1], C = [1, 1]`:

```
00 01 02 03 04 05      00 01 03 04
10 11 12 13 14 15      20 21 23 24
20 21 22 23 24 25      40 41 43 44
30 31 32 33 34 35 --->
40 41 42 43 44 45
50 51 52 53 54 55
```

In general, *PruneGrid* should be used on high density index space dimensions, meaning the majority of function calls should be executed. On low density index space dimensions, *CompressGrid* should be executed.

CUDA index spaces support only a limited number of dimensions, with a limited length.

To handle this, we introduce two inverse mappings: the *SplitLast* and *FoldLast*2. These two mappings can best be used in combination with the *Permute* mapping introduced a bit further below. *These mappings can only be executed on index spaces without a lowerbound, step or width.*

SplitLast splits the last (innermost) dimension into two, increasing the dimensionality by one. The mapping takes the length of the innermost dimension as an argument l. Note that the length of the original dimension may be increased to make sure the upperbound is a multiple of the given length. All function executions for those extra index vectors will be returned, so they will not be executed, so some overhead is expected. In the partition body function, the original index is recomputed from the indexes on the two new dimensions. The index vectors are placed in the two new dimensions using the second to last dimension as mayor and the last dimension as minor dimensions.

Example index space transformation with
`l = 6`:
(.. means the function for this index vector will be returned)

```
00 01 02 03 04 05 06 07 08 09
   |
   V
00 01 02 03 04 05
06 07 08 09 .. ..
```

*FoldLast*2 folds the last (innermost) two dimensions into one, decreasing the dimensionaligy by one. It is the inverse mapping to *SplitLast*. In the partition body function, the original two indexes are recomputed from the index on the new dimension. The index vectors are placed in the new dimension using the original second to last dimension as mayor and the original last dimension as minor dimensions.

Example index space transformation with
`l = 6`:

```
00 01 02 03 04 05
10 11 12 13 14 15
   |
   V
00 01 02 03 04 05 10 11 12 13 14 15
```

CUDA index spaces spawn threads in warps of 32.

Having the total number of index vectors something else then a multiple of 32 will slow down the hardware thread creation process significantly. Because of this, we want to pad the number of threads to a multiple of 32. The *PadLast* mapping handles this for us. Note that the number 32 may differ per GPU device.

PadLast takes one parameter p. It adds indexes to the end of the last (innermost) dimension until it its length is a multiple of p. In the partition body function, all threads added here are returned again. Unless this mapping is used to generate loads of extra indexes, the overhead should be reasonably small.

Example index space transformation with

p = 7:
(.. means the function for this index vector will be returned)

```
00 01 02 03 04 05      00 01 02 03 04 05 ..
10 11 12 13 14 15      10 11 12 13 14 15 ..
20 21 22 23 24 25      20 21 22 23 24 25 ..
30 31 32 33 34 35 ---> 30 31 32 33 34 35 ..
40 41 42 43 44 45      40 41 42 43 44 45 ..
50 51 52 53 54 55      50 51 52 53 54 55 ..
```

CUDA performs caching.

The GPU will be fastest when the indexes are ordered in a way that neighboring data is accessed by neighboring threads. Because SAC is a functional language, it may replace some operations on mappings with accessor functions. This means that the order of the dimensions in the data may be different to the order of the dimensions in the index space. *Permute* can be used to rectify that. Additionally, *Permute* can be used to setup the index space for a *SplitLast*, *FoldLast*2 or *PadLast*.

Permute takes the permutation as argument P. It changes the order of dimensions by permuting L, U, T and W. In the partition body function, it changes the order of the dimensions of the iv back to the original permutation. The overhead of this mapping is exactly 0, as this can be fully handled in the compiler. No code is generated for this mapping.

Example index space transformation with
p = 7:

```
00 01 02 03 04 05      00 10 20 30 40 50
10 11 12 13 14 15      01 11 21 31 41 51
20 21 22 23 24 25      02 12 22 32 42 52
30 31 32 33 34 35 ---> 03 13 23 33 43 53
40 41 42 43 44 45      04 14 24 34 44 54
50 51 52 53 54 55      05 15 25 35 45 55
```

1.4 Mental model

TODO: change or delete?

To help understanding what's going on, we will create a simple mental model of the steps we need to implement this. The mental model follows the structure of the figure above.

- On the left hand side, we have the SAC index-space and indexes
- On the right hand side, we have the CUDA thread-space and indexes
- When executing a mapped partition, we perform the following steps:
  - Firstly, we map the index space itself, moving right in our mental model
  - Secondly, we have CUDA generate the threads and indexes, moving down in our mental model
  - Thirdly, we have to map the generated indexes back to their SAC counterparts, moving left in our mental model
  - Lastly, we execute the with-loop body. We will not discuss this in further detail, so it is not included in our mental model.

Chapter 2

Formalization of mappings

2.1 Preliminary definitions

Before we jump into the definitions of index spaces and their mappings, we will first create some preliminary definitions and notations.

Definition 2.1 (Preliminary assumptions)**.** (preliminaries)

- $\mathbb{B} = \{\top, \bot\}$ is the set of boolean values.
- $\mathbb{N}$ is the set of natural numbers, including 0
- A^n denotes a vector of type A of length $n \in \mathbb{N}$.
- Usually, we distinguish capital variables from lowercase variables:
  - If we talk about capital U, L, …, they denote the *vectors*
  - If we talk about U_i, L_i, …, they denote the value of a vector in dimension i.
  - If we talk about lower u, f, …, they denote either a normal value not being a vector, or the variable is used as a shorthand for a vector variable when the dimension is apparent from context.
- If $f : A \to B$ is a function, then for any n we can apply that function on vectors $A^n \to B^n$ by applying it to each dimension of the vector:

  $f(I) = I'$, with $\forall d \in [0..n-1], I'_d = f(I_d)$
- If $f : A \to \mathbb{B}$ is a function, then for any n, $f^{\downarrow} : A^n \to \mathbb{B}$ we will apply that same operation to each dimension of the vector. The result is then $\top$ iff all results are $\top$:

$$f^{\downarrow}(N) = \left\{ \begin{array}{ccl} \top & | & f(A) = \top^n \\ \bot & | & otherwise \end{array} \right.$$

We will mostly work with vectors of natural or whole numbers. In line with the preliminary assumptions, these types are denoted as $\mathbb{N}^n$ and $\mathbb{Z}^n$, for some $n \in \mathbb{N}$.

Definition 2.2 (Posets using product order)**.** (poset)

We define two partial orderings (posets) on the set of all vectors of type $\mathbb{Z}^n$.

The first poset we define is the product order [3]. In our context, the product order for two vectors $I, I' \in \mathbb{N}^n$ will be defined as the poset $(\mathbb{Z}^n, \leq)$ with for all $I, I' \in \mathbb{Z}^n$:

$I \leq I' \iff \forall d \in [0..n-1], I_d \leq I'_d$

The required proofs, such as transitivity, reflexivity and antisymmetry are given in [3].

The second poset we define is the poset $(\mathbb{Z}^n, \leq^{\downarrow})$ using the definition of $<^{\downarrow}$ as described in definition 2.1. We still have to prove $(\mathbb{Z}^n, \leq^{\downarrow})$ is a valid poset, by showing the properties *reflexivity*, *antisymmetry* and *transitivity* on it.

Proof. reflexivity: $I \leq^{\downarrow} I$. By definition 2.1, this is equal to $\forall d \in [0..n-1], I_d \leq I_d$. This is always true, as for every $x \in \mathbb{Z}$ holds $x \leq x$.

antisymmetry: if $I \leq^{\downarrow} I'$ and $I' \leq^{\downarrow} I$, then $I = I'$. We prove this by contradiction. Let's state that $I \neq I'$ and $I \leq I'$. Note that for $I \neq I'$ to hold, we need $n \geq 1$. This means that $\forall d \in [0..n-1], I_d < I'_d$. This contradicts the assumption that $I' \leq^{\downarrow} I$. $\lightning$

transitivity: if $I \leq^{\downarrow} I''$ and $I'' \leq^{\downarrow} I'$ then $I \leq^{\downarrow} I'$. We know that $\forall d \in [0..n-1], I_d \leq I''_d \leq I'_d \to I_d \leq I'_d$, as it is the case for all $x, y, z \in \mathbb{Z}$. From these assumptions combined, we can conclude that $I \leq^{\downarrow} I'$. ∎

Note. There is a difference between $I < I'$ and $I <^{\downarrow} I'$. $[0, 1] < [1, 1]$, but $[0, 1] \not<^{\downarrow} [1, 1]$. The same difference exists between $I \leq I' \not\Longleftrightarrow I \leq^{\downarrow} I'$.

Note. $\mathbb{N}^n \subset \mathbb{Z}^n$, so this partial order works on natural numbers too.

2.2 Index spaces and mappings

Let us start with some definitions. Note that some definitions are further specified in definition 2.4.

Definition 2.3 (Index space transformations)**.** (index-space-transform)

- Let $S \in \mathbb{S}$ be an index space. The exact definition of S and $\mathbb{S}$ will be discussed in definition 2.4.
- Let $^*S \in {}^*\mathbb{S}$ be an index space descriptor. An index space descriptor is a tuple of values describing what values should be in the index space. The exact definition of $^*S$ and $^*\mathbb{S}$ will be discussed in definition 2.4.
- Let $I \in S$ be an index in index space S respectively.
- Let $gen : {}^*\mathbb{S} \to \mathbb{S}$ be the index space generator function. It maps the index space descriptor to an index space.
- Let $tf^s_{\rightarrow} : {}^*\mathbb{S} \to {}^*\mathbb{S}$ be an index space mapping. It maps an index space descriptor from the SAC side (left) more towards the CUDA side (right).
- Let $tf^i_{\leftarrow} : S' \to S$ be an index mapping. It maps an index from the CUDA side (right) more towards the SAC side (left). The domain and codomain of $tf^i_{\leftarrow}$ are dependent on the index space descriptor of the codomain and the gen and $tf^s_{\rightarrow}$ functions: $^*\mathbb{S}' = tf^s_{\rightarrow}(^*S)$, $S = gen(^*S)$ and $S' = gen(^*S')$.
- A partition mapping is a pair of $(tf^s_{\rightarrow}, tf^i_{\leftarrow})$. In normal with-loop execution, we will first call $tf^s_{\rightarrow}$ on the index space descriptor, then generate the index space using gen and finally transform all indexes back to the initial index space. The result of these function calls should be equal to calling gen directly:

  $gen(^*S) = \{tf^i_{\leftarrow}(I) | I \in gen(tf^s_{\rightarrow}(^*S))\}$

Before we can formalize the definition of an index space further, we have to discuss the requirements. We already know that an index space descriptor is a set of integer vectors with the same length, defining the lowerbound, upperbound, step and width values. $tf^s_{\rightarrow}$ can change these parameters when transforming an index space descriptor, as long as $tf^i_{\leftarrow}$ can transform the resulting index space back into the original index space.

Because we need to recompute the original left-hand indexes from the mapped right-hand indexes, the mapped index space can never be smaller then the original index space on the left. However, some mappings may introduce extra elements in an in index space. An example of such a mapping is $PadLast$. $PadLast^s_{\rightarrow}$ will purposefully increase the size of the index space by increasing the upperbound. However, the extra partition body function calls for these index vectors should not actually be executed. We introduce the $\perp$ element as a way to remove an index vector from an index space: a $tf^i_{\leftarrow}$ can discard a partition body function call by mapping it's index to $\perp$. Because no indexes can be created out of thin air, we can assume that $tf^i_{\leftarrow}(\perp)$ is always equal to $\perp$ as well.

We can now formally define our index spaces. We will differentiate between three types of index spaces. Grid index spaces, normalized index spaces, and dense index spaces. Grid index spaces are index spaces defined by the lowerbound, upperbound, step and width. Normalized index spaces are a subset of grid index spaces without a lowerbound, or where the lowerbound is zero. Dense index spaces are a subset of normalized index spaces without a step and width, or where the step equals the width.

Definition 2.4 (Index spaces)**.** (index-space)

- Index space $S^n \in \mathbb{S}^n := \mathcal{P}(\mathbb{N}^n \cup \{\perp\})$, with $n \in \mathbb{N}$ and $\mathcal{P}$ the power set.
- Grid index space $^*S^n_g := (L, U, T, W) \in {}^*\mathbb{S}^n_g \subset \mathbb{S}^n$
  - $L, U, T, W \in \mathbb{N}$, $L \leq U$ and $W \leq T$
  - $gen(^*S^n_g) = \{I | I \in S^n \wedge (I = \perp \vee (L \leq I <^{\downarrow} U \wedge (I-L) \overrightarrow{\%} T <^{\downarrow} W))\}$
- Normalized index sp. $S^n_n := (U, T, W) \in \mathbb{S}^n_n \subset \mathbb{S}^n_g$
  - $gen(^*S^n_n) = gen((0, U, T, W))$
- Dense index space $S^n_d := (U) \in \mathbb{S}^n_d \subset \mathbb{S}^n_n$
  - $gen(^*S^n_n) = gen((0, U, 1, 1))$
- For all $tf^i_{\leftarrow}$: $tf^i_{\leftarrow}(\perp) = \perp$

Lemma 2.5 (*gen* of normalized and dense index spaces)**.** *(gen-norm-dense)*

Because gen of normalized en dense index spaces are just specialized versions of the gen of grid index spaces, we can simplify their formula's a bit.

- $gen(^*S^n_n) = \{I | I \in S^n \wedge (I = \perp \vee (0 \leq I <^{\downarrow} U \wedge (I - 0) \overrightarrow{\%} T <^{\downarrow} W))\}$

  $\iff$

  $gen(^*S^n_n) = \{I | I \in S^n \wedge (I = \perp \vee (I <^{\downarrow} U \wedge I \overrightarrow{\%} T <^{\downarrow} W))\}$
- $gen(^*S^n_d) = \{I | I \in S^n \wedge (I = \perp \vee (0 \leq I <^{\downarrow} U \wedge (I - 0) \overrightarrow{\%} 1 <^{\downarrow} 1))\}$

  $\iff$

$gen(^*S^n_n) = \{I | I \in S^n \wedge$
$(I = \perp \vee I <^{\downarrow} U)\}$

Mapping correctness

As stated before, transforming the index space right and transforming the generated indexes back left should be equivalent to generating the indexes on the original index space:

$\forall S^n \in \mathbb{S}^n : gen(S^n) = \{tf^i_{\leftarrow}(I) | I \in gen(tf^s_{\rightarrow}(S^n))\}$

However, this predicate is not entirely accurate. Because sets cannot contain duplicate elements, there may be more then one element I' in $S^{n\prime}$ with $tf^i_{\leftarrow}(I') = I$ with $I \in S^n$. When executing a with-loop partition, we do not want two partition body function calls with the same index. The one exception to this statement is $\perp$. Index invokations with index $\perp$ won't be executed anyway, so there may be as many as we like.

To formalize this difference, we will distinguish between the *operative subset* and the *excess subset* of an index space.

Definition 2.6 (Operative and excess subsets)**.** (operative-subset)

- The operative subset of an index space $\mathcal{O}(S^n)$ is the index space without $\perp$.

  $\mathcal{O}(S^n) = S^n \backslash \{\perp\}$
- The excess subset of an index space $\mathcal{E}(S^n)$ is $\{\perp\}$.

  $\mathcal{E}(S^n) = \{\perp\}$
- The operative subset of an index space S^n after a partition mapping ($\mathcal{O}_{tf}(S^n)$) is the subset of $S^{n\prime}$ that gets mapped to the operative subset of S^n by $tf^i_{\leftarrow}$.

  $\mathcal{O}_{tf}(S^n) = \{i | tf^i_{\leftarrow}(i) \in \mathcal{O}(S^n) \wedge i \in S'\}$
- The excess subset of an index space S after a partition mapping ($\mathcal{E}_{tf}(S^n)$) is the subset of $S^{n\prime}$ that $tf^i_{\leftarrow}$ maps to the excess subset of S^n, e.g. that $tf^i_{\leftarrow}$ maps to $\perp$.

  $\mathcal{E}_{tf}(S^n) = \{i | tf^i_{\leftarrow}(i) = \perp \wedge i \in S^{n\prime}\}$

Note that, by definition, for any well-defined tf and S:

- $\mathcal{O}(S^n) \cup \mathcal{E}(S^n) = S^n$
- $\mathcal{O}_{tf}(S^n) \cup \mathcal{O}_{tf}(S^n) = S^{n\prime}$
- $\mathcal{O}(S^n) \cap \mathcal{E}(S^n) = \emptyset$
- $\mathcal{O}_{tf}(S^n) \cap \mathcal{O}_{tf}(S^n) = \emptyset$

We can now fully formalize the requirements of a partition mapping. We will call these requirements *mapping equivalence.*

Definition 2.7 (Mapping equivalence)**.** (mapping-equivalence)

A partition mapping tf is mapping equivalent iff:

- $tf^s_{\rightarrow}$ is well-defined
- $tf^i_{\leftarrow}$ is well-defined
- $tf^i_{\leftarrow}$ is operational bijective:

  $tf^i_{\leftarrow} : \mathcal{O}_{tf}(S^n) \rightarrow \mathcal{O}(S^n)$ is a bijection

2.3 Proving mapping equivalence

In proving mapping equivalence for the different mappings, we will oftentimes use similar very similar techniques or lemma's. In this section we will specify a few of those in lemma's and definitions. We also talks about general approaches for some situations.

Notation of domains and codomains

Whenever we are discussing a mapping, either as a single transformation or as a pair of transformations, we define the following properties over the domains and codomains:

- The right hand side of the mapping (the codomain of a $tf^s_{\rightarrow}$ and the domain of a $tf^i_{\leftarrow}$) will have the same variables as the left hand side of the mapping, with a prime (') added. These can be S and $\mathbb{S}$ and variations, I, L, U, T, W and lowercase variants, and any other helper variables we define in the process.
- The left hand side of the mapping (the domain of a $tf^s_{\rightarrow}$ and the codomain of a $tf^i_{\leftarrow}$) will have the non-prime version of variables.
- For every $^*S$, we assume there to be an S with $S = gen(^*S)$. Likewise, for every S we assume there to be a $^*S$.

Well-definedness of $tf^s_{\rightarrow}$

Lemma 2.8 (Well-definedness of $tf^s_{\rightarrow}$)**.** *(tf-wds)*

For any $tf^s_{\rightarrow} : {}^*\mathbb{S} \rightarrow {}^*\mathbb{S}'$ we will have to prove well-definedness by proving that for any $^*S \in {}^*\mathbb{S}$: $tf^s_{\rightarrow}(^*S) \in {}^*\mathbb{S}'$. To see exactly what this entails, we have to expand the definition and constraints of $S \in \mathbb{S}$. For $^*S^{n\prime}_g \in {}^*\mathbb{S}^n_g$, we have to prove that:

- $L', U', T', W' \in \mathbb{N}^n$
- $L' \leq U'$
- $T' \leq W'$

By definition of ${}^*\mathbb{S}^n_n$ and ${}^*\mathbb{S}^n_d$, we have to substitute $L = 0$ and $T = 1, W = 1$ respectively. This means we can simplify the proofs for any ${}^*S^n_n \in {}^*\mathbb{S}^n_n$ and ${}^*S^n_d \in {}^*\mathbb{S}^n_d$.

- *For ${}^*S^n_n \in {}^*\mathbb{S}^n_n$ it is sufficient to prove:*
  - $U', T', W' \in \mathbb{N}^n$
  - $T' \leq W'$

  We can omit $0 \leq U'$, as this holds for any $U' \in \mathbb{N}^n$.

- *For ${}^*S^n_d \in {}^*\mathbb{S}^n_d$ it is sufficient to prove:*
  - $U' \in \mathbb{N}^n$

Well-definedness of $tf^i_{\leftarrow}$

Lemma 2.9 (Well-definedness of $tf^i_{\leftarrow}$)**.** *(tf-wdi)*

For any $tf^i_{\leftarrow} : S' \to S$ we will have to prove well-definedness by proving that for any $I' \in S' : tf^i_{\leftarrow}(I') \in S$. To see what exactly this means, we have to expand the definition of gen for that particular index space. For any $tf^i_{\leftarrow}$, we have to prove:

Given I' with $I' = \bot \vee (L' \leq I' <^{\downarrow} U' \wedge (I' - L') \overrightarrow{\%}\, T' <^{\downarrow} W')$,
Then $I = \bot \vee (L \leq I <^{\downarrow} U \wedge (I - L) \overrightarrow{\%}\, T <^{\downarrow} W)$, with $I = tf^i_{\leftarrow}(I')$ should hold.

Note that this automatically holds for $I' = \bot$, because by definition, $tf^i_{\leftarrow}(\bot) = \bot$ for any $tf^i_{\leftarrow}$. Additionally, we can use the simplifications from lemma 2.5 to make simpler proofs for codomains in $\mathbb{S}^n_n$ and $\mathbb{S}^n_d$:

- *For $tf^i_{\leftarrow} : S^{n\prime}_g \to S^n_g$:*

  Given I' with $L' \leq I' <^{\downarrow} U' \wedge (I' - L') \overrightarrow{\%}\, T' <^{\downarrow} W'$,
  Then $I = \bot \vee (L \leq I <^{\downarrow} U \wedge (I - L) \overrightarrow{\%}\, T <^{\downarrow} W)$, with $I = tf^i_{\leftarrow}(I')$ should hold.

- *For $tf^i_{\leftarrow} : S^{n\prime}_g \to S^n_n$:*

  Given I' with $L' \leq I' <^{\downarrow} U' \wedge (I' - L') \overrightarrow{\%}\, T' <^{\downarrow} W'$,
  Then $I = \bot \vee (I <^{\downarrow} U \wedge (I - L) \overrightarrow{\%}\, T <^{\downarrow} W)$, with $I = tf^i_{\leftarrow}(I')$ should hold.

- *For $tf^i_{\leftarrow} : S^{n\prime}_g \to S^n_n$:*

  Given I' with $L' \leq I' <^{\downarrow} U' \wedge (I' - L') \overrightarrow{\%}\, T' <^{\downarrow} W'$,
  Then $I = \bot \vee I <^{\downarrow} U$, with $I = tf^i_{\leftarrow}(I')$ should hold.

Operational bijectivity of $tf^i_{\leftarrow}$

Lemma 2.10 (Operational bijectivity of $tf^i_{\leftarrow}$)**.** *(tfi-opbi)*

For any well-defined $tf^i_{\leftarrow} : S' \to S$ we need prove that $tf^i_{\leftarrow} : \mathcal{O}_{tf}(S) \to \mathcal{O}(S)$ is a bijection. We do this by showing $\textcircled{A}$ surjectivity and $\textcircled{B}$ injectivity.

$\textcircled{A}$ For surjectivity, we need to prove that for each element I in $\mathcal{O}(S)$, there is an element I' in $\mathcal{O}_{tf}(S)$ with $tf^i_{\leftarrow}(I') = I$:

Given an $I \in \mathcal{O}(S)$:
$\exists I' \in \mathcal{O}_{tf}(S), tf^i_{\leftarrow}(I') = I$

However, by definition, we know that $\mathcal{O}_{tf}(S)$ is exactly the subset of S' with $tf^i_{\leftarrow}(I') \in S$. Because of this, it is equivalent to prove\$:

Given an $I \in \mathcal{O}(S)$:
$\exists I' \in S', tf^i_{\leftarrow}(I') = I$

$\textcircled{B}$ For injectivity, we need to prove that if two indexes $I', I'' \in \mathcal{O}_{tf}(S)$ both map to the same element I in S, then they are the same element:

Given an $I', I'' \in \mathcal{O}_{tf}(S)$:
$tf^i_{\leftarrow}(I') = tf^i_{\leftarrow}(I'') \to I' = I''$

Again, we can make use of the definition of $\mathcal{O}_{tf}(S)$ and rewrite the proposition as:

Given an $I \in \mathcal{O}(S)$ and $I', I'' \in S'$:
$tf^i_{\leftarrow}(I') = I \wedge tf^i_{\leftarrow}(I'') = I \to I' = I''$

Separating dimensions

In many cases, it is easier to talk about or prove something for one single dimension instead of for all dimensions at once. This is possible when the dimensions are independent from eachother in the current context. To enable us to easily talk about those individual dimensions, we may *separate dimensions.*

Definition 2.11 (Separating dimensions)**.** (sep-dim)

When separating dimensions, we prove or show a property on a single abstract dimension. If we can prove it for any dimension, we know it has to be true for all dimensions.

Whenever we separate dimensions, we will implicitly introduce the following variables:

- The variable d contains the current abstract dimension we are talking about
- The variables $i, l, u, t, w, i', l', u', t', w'$ are the values of their captital counterparts in dimension d
- $tf^i_{\leftarrow d}(I)$ are the values of $tf^i_{\leftarrow}(I)$ in dimension d,

- $tf^i_{\leftarrow d}(i)$ are the values of $tf^i_{\leftarrow}(I)$ in dimension d, iff this function is only dependent on the values on dimension d,

Definition 2.12 (Dimension dependency)**.** (dim-dep)

In the context of a certain mapping, two dimensions d and d' with $d \neq d'$ of their index spaces can be either dependent or independent. Which one of them is applicable can be derived from the definitions of $tf^s_{\rightarrow}$ and $tf^i_{\leftarrow}$.

For some dimension d holds:

- If the definition of $tf^s_{\rightarrow}$ or $tf^i_{\leftarrow}$ for dimension d *possibly* refers to some $L_{d'}, U_{d'}, T_{d'}, W_{d'}, I_{d'}$ with $d \neq d'$, then dimensions d and d' are dependent.
- The opposite is automatically also true: if the defiition of $tf^s_{\rightarrow}$ or $tf^i_{\leftarrow}$ for some dimension d' *possibly* refers to L_d, U_d, T_d, W_d, I_d with $d \neq d'$, dimensions d and d' are dependent.
- If dimensions d and d' are not dependent, they are dependent.
- If there is no d' so that d and d' are dependent, we speak of d as being independent.

Lemma 2.13 (Proofs with separated dimensions)**.** *(sep-dim-prf)*

In the proofs outlined by lemma's 2.8, 2.9 and **??***, it is always possible to separate dimensions as in definition 2.11. The properties in question are:*

- *Well definedness of $tf^s_{\rightarrow}$*
- *Well definedness of $tf^i_{\leftarrow}$*
- *Operational injectivity of $tf^i_{\leftarrow}$*
- *Operational surjectivity of $tf^i_{\leftarrow}$*

Proof. For the well definedness of $tf^s_{\rightarrow}$ and $tf^i_{\leftarrow}$, this is trivial. Both the definition of S^n_g and $I \in S^n_g$ are defined so that the dimensions are completely independent of eachother.

For the operational injectivity and surjectivity of $tf^i_{\leftarrow}$, this follows from how $I \in S^n_g$ is defined. It is defined as $\forall d \in [0..n-1] : I_d \in S^n_g{}_d \iff I \in S^n_g$. This means that if you can prove for all dimensions that they are surjective and injective, you can prove that $tf^i_{\leftarrow}$ is surjective and injective. Of course, some extra steps may be required if the dimensions are not independent of eachother in $tf^i_{\leftarrow}$. ∎

(Partially) Identity mappings

In some cases, mappings leave certain dimensions completely intact, behaving like an identity mapping for those dimensions. To avoid repeating the same proof over and over again, we will prove mapping equivalence for such an identity dimension.

Lemma 2.14 (Mapping equivalence for identity dimensions)**.** *(tf-idd)*

For any dimension d that is separated using definition 2.11 and $tf^s_{\rightarrow d}((l,u,s,w)) = (l,u,s,w), tf^i_{\leftarrow d}(i) = i$, tf is mapping equivalent.

Proof. Using definition 2.7 we now have to prove Ⓐ $tf^s_{\rightarrow}$ is well defined, Ⓑ $tf^i_{\leftarrow}$ is well defined and Ⓒ $tf^i_{\leftarrow}$ is operationally bijective.

Ⓐ We know that $l, u, s, w \in \mathbb{N}$. By definition, $tf^s_{\rightarrow}((l,u,s,w)) = (l,u,s,w)$. Using lemma \$2.8, we know that $tf^s_{\rightarrow}$ is well defined.

Ⓑ We know that $tf^s_{\rightarrow}({}^*S^n_g) = {}^*S^n_g$. We also know that $tf^i_{\leftarrow}(i) = i$. This makes $tf^i_{\leftarrow}$ an identity function, which is well defined.

Ⓒ Using lemma 2.10, we have to prove (C.1) $\exists i' \in S^n_g, tf^i_{\leftarrow d}(i') = i$ for any $i \in S^n_g indimensiond$, and (C.2) $tf^i_{\leftarrow}(i') = i \wedge tf^i_{\leftarrow}(i'') = i \rightarrow i' = i''$ for any $i, i', i'' \in S^n_g$.

(C.1) By definition, if $i' = i$ then $tf^i_{\leftarrow d}(i') = i$.

(C.2) By definition, $tf^i_{\leftarrow d}(i') = i \iff i' = i$. This also holds for i''. Now we can prove $i' = i = i'' \implies i' = i''$. ∎

2.4 ShiftLB

ShiftLB removes the lowerbound of an index space by shifting the index space so that the lowerbound becomes 0. This means that the mapping transforms any grid index space to a normalized index space.

Definition 2.15 (*ShiftLB*)**.** (def-shiftlb)

- $ShiftLB^s_{\rightarrow} : {}^*\mathbb{S}^n_g \rightarrow {}^*\mathbb{S}^n_n$

  $ShiftLB^s_{\rightarrow}((L,U,T,W)) = (U', T', W')$ with:

  - $U' = U - L$
  - $T' = T$
  - $W' = W$

- $ShiftLB^i_{\leftarrow} : S^{n\prime}_n \rightarrow S^n_g$

  $ShiftLB^i_{\leftarrow}(I) = I + L$

Lemma 2.16 ($ShiftLB^s_{\rightarrow}$ is well-defined)**.** *(shift-wds)*

Given any ${}^*S^n_g \in {}^*\mathbb{S}^n_g$. We now have to prove that $ShiftLB^s_{\rightarrow}({}^*S^n_g) \in \mathbb{S}^n_n$.

Proof. Using lemma 2.8 for ${}^*S^n_n$, we have to prove Ⓐ $U', T', W' \in \mathbb{N}$, Ⓑ $W' \leq T'$.

Ⓐ As $L \leq U$ holds, we know that $U - L$ is a vector of natural numbers ≥ 0. This means that $U' \in Nn$.

$T' \in \mathbb{N}^n$ and $W' \in \mathbb{N}^n$ holds too, as $T \in \mathbb{N}^n$ and $W \in \mathbb{N}^n$.

Ⓑ $W' \leq T'$ holds, as $W \leq T$ holds. ■

Lemma 2.17 (*$ShiftLB^i_{\leftarrow}$* is well-defined)**.** *(shift-wdi)*

Given any $I' \in {S^n_n}'$. We now have to prove that $ShiftLB^i_{\leftarrow}(I') \in S^n_g$.

Proof. Using lemma 2.9 for codomain S^n_g, we can now prove for any given I' and $I = ShiftLB^i_{\leftarrow}(I')$:

$L' \leq I' <^{\downarrow} U' \wedge (I' - L') \;\%\; T' <^{\downarrow} W' \rightarrow$
$I = \perp \vee (L \leq I <^{\downarrow} U \wedge (I - L) \;\%\; T <^{\downarrow} W)$

$\iff$

$0 \leq I' <^{\downarrow} U - L \wedge (I' - 0) \;\%\; T <^{\downarrow} W \rightarrow$
$L \leq (I' + L) <^{\downarrow} U \wedge (I' + L - L) \;\%\; T <^{\downarrow} W$
Inserted definitions of shiftLB, pruned left or clause

$\iff$

$0 \leq I' <^{\downarrow} U - L \wedge I' \;\%\; T <^{\downarrow} W \rightarrow$
$L \leq (I' + L) <^{\downarrow} U \wedge I' \;\%\; T <^{\downarrow} W$
Some simplifications of formulas

$\iff$

$0 \leq I' <^{\downarrow} U - L \wedge I' \;\%\; T <^{\downarrow} W \rightarrow$
$L \leq (I' + L) <^{\downarrow} U$
Right 'and' clause is true from our assumption, substracted L from formula

$\iff$

$0 \leq I' <^{\downarrow} U - L \wedge I' \;\%\; T <^{\downarrow} W \rightarrow$
$0 \leq I' <^{\downarrow} U - L$

■

Lemma 2.18 (*$ShiftLB^i_{\leftarrow}$* is operational bijective)**.** *(shift-opbi)*

Take the operational parts of S^n_g : $\mathcal{O}(S^n_g)$ and ${S^n_n}'$: $\mathcal{O}_{ShiftLB^i_{\leftarrow}}(S^n_g)$). We now have to prove that $ShiftLB^i_{\leftarrow} : \mathcal{O}_{ShiftLB^i_{\leftarrow}}(S^n_g) \rightarrow \mathcal{O}(S^n_g)$ is bijective.

Proof. Using lemma 2.10, we now have to prove Ⓐ surjectivity and Ⓑ injectivity.

Ⓐ Given $I \in \mathcal{O}(S^n_g)$. We now have to prove that there is some $I' \in {S^n_n}'$ with $tfi(I') = I$.

We define this I' as $I - L$. We now have to prove that ① $tf^i_{\leftarrow}(I') = I$ and ② $I' \in {S^n_n}'$.

① $tf^i_{\leftarrow}(I')$
$= I' + L$
$= I - L + L$
$= I$

② We know that $I \in \mathcal{O}(S^n_g)$. We can now prove:

$L \leq I <^{\downarrow} U \wedge (I - L) \;\%\; T <^{\downarrow} W \rightarrow$
$L' \leq I' <^{\downarrow} U' \wedge (I' - L') \;\%\; T' <^{\downarrow} W'$

$\iff$

$L \leq I <^{\downarrow} U \wedge (I - L) \;\%\; T <^{\downarrow} W \rightarrow$
$0 \leq (I - L) <^{\downarrow} (U - L) \wedge (I - L - 0) \;\%\; T <^{\downarrow} W$
Inserted definitions of prime values

$\iff$

$L \leq I <^{\downarrow} U \wedge (I - L) \;\%\; T <^{\downarrow} W \rightarrow$
$L \leq I <^{\downarrow} U \wedge (I - L) \;\%\; T <^{\downarrow} W$
Right 'and' clause is true from our assumtion, added L to formula

Ⓑ Given $I \in \mathcal{O}(S^n_g)$ and $I', I'' \in {S^n_n}'$. We now have to prove that if $tf^i_{\leftarrow}(I') = I$ and $tf^i_{\leftarrow}(I'') = I$, then $I' = I''$. We prove this by contradiction. Assume there are two $I' \neq I''$ with $tf^i_{\leftarrow}(I') = I$ and $tf^i_{\leftarrow}(I'') = I$:

$I' \neq I'' \wedge tf^i_{\leftarrow}(I') = I \wedge tf^i_{\leftarrow}(I'') = I$

$\iff$

$I' \neq I'' \wedge I' + L = I \wedge I'' + L = I$

$\iff$

$I' \neq I'' \wedge I' = I - L \wedge I'' = I - L$

$\iff$

$I' \neq I'' \wedge I' = I''$ ↯ ■

Theorem 2.19 (ShiftLB is mapping-equivalent)**.** *(shift-me)*

By definition 2.7, ShiftLB is mapping equivalent iff:

- Ⓐ *$ShiftLB^s_{\rightarrow}$ is well defined*
- Ⓑ *$ShiftLB^i_{\leftarrow}$ is well defined*
- Ⓒ *$ShiftLB^i_{\leftarrow}$ is operationally bijective*

Proof. Ⓐ is proven in lemma 2.16
Ⓑ is proven in lemma 2.17
Ⓒ is proven in lemma 2.18 ■

2.5 CompressGrid

CompressGrid removes the step and width from a normalized index space by removing the space between the indexes in the index space. This means that the mapping transforms any normalized index space into a dense index space.

CompressGrid takes an extra argument $C \in \mathbb{B}^n$, defining which dimensions should be compressed.

Definition 2.20 (*CompressGrid*)**.** (def-compress)

- $CompressGrid^s_\rightarrow : \mathbb{B}^n \rightarrow {}^*\mathbb{S}^n_n \rightarrow {}^*\mathbb{S}^n_n$

  $CompressGrid^s_\rightarrow(C)(U, T, W) = (U', T', W')$:

  For definitions of U', T', W' we separate dimensions as in definition 2.11.

$$u' = \begin{cases} \left\lfloor \frac{u}{t} \right\rfloor * w + min(u \ \% \ t, w) & | \ c = \top \\ u & | \ c = \bot \end{cases}$$

$$t' = \begin{cases} 1 & | \ c = \top \\ t & | \ c = \bot \end{cases}$$

$$w' = \begin{cases} 1 & | \ c = \top \\ w & | \ c = \bot \end{cases}$$

- $CompressGrid^i_\leftarrow : S^{n\prime}_d \rightarrow S^n_n$

  $CompressGrid^i_\leftarrow(I') = I$:

  For the definition of I we separate dimensions as in definition 2.11.

$$i = \begin{cases} \left\lfloor \frac{i'}{w} \right\rfloor * t + i' \ \% \ w & | \ c = \top \\ i' & | \ c = \bot \end{cases}$$

Lemma 2.21 ($CompressGrid^s_\rightarrow$ is well defined)**.** *(compress-wds)*

Given any $^*S^n_n \in {}^*\mathbb{S}^n_n$. *We now have to prove that* $CompressGrid^s_\rightarrow({}^*S^n_n) \in {}^*\mathbb{S}^n_d$.

Proof. Using lemma 2.8 for $^*S^n_d$, we now only have to prove that $U' \in \mathbb{N}^n$. We separate dimensions as in definition 2.11. We also distinct two cases: $c = \top$ and $c = \bot$.

- $c = \top$:

  $u, t, w \in \mathbb{N} \rightarrow \left\lfloor \frac{u}{t} \right\rfloor * w + min(u \ \% \ t, w) \in \mathbb{N}$

  $\Longleftrightarrow$

  $u, t, w \in \mathbb{N} \rightarrow \left\lfloor \frac{u}{t} \right\rfloor, u, t, w \in \mathbb{N}$

  For u, t, w, this immediately holds from the assumption. For $\left\lfloor \frac{u}{t} \right\rfloor$ this holds as well, as the floor makes the result an integer again. As u and t are at least zero, $\frac{u}{t}$ is at least zero as well.

- $c = \bot$:

  We know $u, t, w \in \mathbb{N}$. We also know $u' = u, t' = t, w' = w$. So, we can conclude that $u', t', w' \in \mathbb{N}$ as well.

∎

Lemma 2.22 ($CompressGrid^i_\leftarrow$ is well defined)**.** *(compress-wdi)*

Given any $I' \in S^{n\prime}_d$. *We now have to prove that* $CompressGrid^i_\leftarrow(I') \in S^n_n$.

Proof. Using lemma **??** for codomain S^n_n, we can now prove for any given I' and $I = CompressGrid^i_\leftarrow(I')$: $I' \in S^{n\prime}_n \rightarrow I \in S^n_n$. We separate dimensions as in definition 2.11.

$i' < u' \land i' \ \% \ t' < w' \rightarrow$
$i = \bot \lor (i < u \land i \ \% \ t < w)$

$\Longleftarrow$

$i' < u' \land i' \ \% \ t' < w' \rightarrow$
$i < u \land i \ \% \ t < w$
Prune left hand side of the 'or'

We now distinct two cases: $c = \top$ and $c = \bot$.

- $c = \top$:

  $i' < u' \land i' \ \% \ t' < w' \rightarrow$
  $i < u \land i \ \% \ t < w$

  $\Longleftrightarrow$

  $i' < (\left\lfloor \frac{u}{t} \right\rfloor * w + min(u \ \% \ t, w)) \land i' \ \% \ 1 < 1 \rightarrow$
  $(\left\lfloor \frac{i'}{w} \right\rfloor * t + i' \ \% \ w) < u \land$
  $(\left\lfloor \frac{i'}{w} \right\rfloor * t + i' \ \% \ w) \ \% \ t < w$

  We now split the proof into the left hand side and the right hand side of the 'and.'

  - Left hand side:

    $i' < (\left\lfloor \frac{u}{t} \right\rfloor * w + min(u \ \% \ t, w)) \rightarrow$
    $(\left\lfloor \frac{i'}{w} \right\rfloor * t + i' \ \% \ w) < u$

    $\Longleftrightarrow$

    $i' < (\left\lfloor \frac{u - u \ \% \ t}{t} \right\rfloor * w + min(u \ \% \ t, w)) \rightarrow$
    $\left\lfloor \frac{i'}{w} \right\rfloor * t \leq u - u \ \% \ t \land i' \ \% \ w < u \ \% \ t$

    Because $\left\lfloor \frac{u}{t} \right\rfloor = \left\lfloor \frac{u - u \ \% \ t}{t} \right\rfloor$ *and* $ab < c + d \leftarrow a \leq c \land b < d$

    $\Longleftrightarrow$

$i' - i' \% w \leq \left\lfloor \frac{u - u \% t}{t} \right\rfloor * w \wedge i' \% w < min(u \% t, w) \rightarrow$
$\left\lfloor \frac{i' - i' \% w}{w} \right\rfloor * t \leq u - u \% t \wedge i' \% w < u \% t$
Similarly, because $\left\lfloor \frac{i'}{t} \right\rfloor = \left\lfloor \frac{i' - i' \% t}{t} \right\rfloor$ *and* $ab < c + d \leftarrow a \leq c \wedge b < d$

Now, we can split this proof in two:

* $i' - i' \% w \leq \left\lfloor \frac{u - u \% t}{t} \right\rfloor * w \rightarrow$
  $\left\lfloor \frac{i' - i' \% w}{w} \right\rfloor * t \leq u - u \% t$

  $\iff$

  $i' - i' \% w \leq \left\lfloor \frac{u - u \% t}{t} \right\rfloor * w \rightarrow$
  $i' - i' \% w \leq \frac{u - u \% t}{t} * w$
  Divide by t, multiply by w

  $\iff$

  $i' - i' \% w \leq \left\lfloor \frac{u - u \% t}{t} \right\rfloor * w \rightarrow$
  $\left\lfloor \frac{i' - i' \% w}{w} \right\rfloor \leq \frac{u - u \% t}{t}$
  Divide by t

  This holds, as $\frac{u - u \% t}{t} = \left\lfloor \frac{u - u \% t}{t} \right\rfloor$.

* $i' \% w < min(u \% t, w) \rightarrow$
  $i' \% w < u \% t$

  $\iff$

  $i' \% w < u \% t \rightarrow$
  $i' \% w < u \% t$

– Right hand side:

$i' < (\left\lfloor \frac{u}{t} \right\rfloor * w + min(u \% t, w)) \wedge i' \% 1 < 1 \rightarrow$
$(\left\lfloor \frac{i'}{w} \right\rfloor * t + i' \% w) \% t < w$

$\Longleftarrow$

$i' < (\left\lfloor \frac{u}{t} \right\rfloor * w + min(u \% t, w)) \wedge i' \% 1 < 1 \rightarrow$
$(\left\lfloor \frac{i'}{w} \right\rfloor * t) \% t + (i' \% w) \% t < w$
Because $a \% t + b \% t < w \rightarrow (a + b) \% t < w$

$\iff$

$i' < (\left\lfloor \frac{u}{t} \right\rfloor * w + min(u \% t, w)) \wedge i' \% 1 < 1 \rightarrow$
$0 + (i' \% w) \% t < w$
Because $(x * t) \% t = 0$*, if* $x \in \mathbb{N}$

This holds, by definition of $\%$.

• $c = \bot$:

$i' < u' \wedge i' \% t' < w' \rightarrow$
$i < u \wedge i \% t < w$

$\iff$

$i' < u \wedge i' \% t < w \rightarrow$
$i' < u \wedge i' \% t < w$
Expand definitions of $CompressGrid^s_{\rightarrow}$ *for* u', t', w' *and* $CompressGrid^i_{\leftarrow}$ *for* i

■

Lemma 2.23 ($CompressGrid^i_{\leftarrow}$ is operational bijective)**.** *(compress-opbi)*

Take the operational parts of $S^n_n : \mathcal{O}(S^n_n)$ *and* $S^{n\prime}_n : \mathcal{O}_{CompressGrid^i_{\leftarrow}}(S^n_n)$*. We now have to prove that* $CompressGrid^i_{\leftarrow} : \mathcal{O}_{CompressGrid^i_{\leftarrow}}(S^n_n) \rightarrow \mathcal{O}(S^n_n)$ *is bijective.*

Proof. We separate dimensions as in definition 2.11. If $c = \bot$, we have an identity mapping and we can use lemma 2.14. For the rest of the proof, we can now assume that $c = \top$.

Using lemma 2.10, we now have to prove $\textcircled{A}$ surjectivity and $\textcircled{B}$ injectivity.

$\textcircled{A}$ Given an $i \in \mathcal{O}(S^n_n)_d$ and an $i' = \left\lfloor \frac{i}{t} \right\rfloor * w + i \% t$. We now have to prove that $\textcircled{A.1}$ $CompressGrid^i_{\leftarrow d}(i') = i$ and $\textcircled{A.2}$ $i' \in S^{n\prime}_{d\ d}$.

$\textcircled{A.1}$ $CompressGrid^i_{\leftarrow d}(i')$

$= \left\lfloor \frac{\left\lfloor \frac{i}{t} \right\rfloor * w + i \% t}{w} \right\rfloor * t + i' \% w$
Expanded definitions of $CompressGrid^i_{\leftarrow}$ *and* i'

$= \left\lfloor \frac{\left\lfloor \frac{i}{t} \right\rfloor * w}{w} \right\rfloor * t + i' \% w$
$\lfloor x \rfloor * w$ *is a multiple of* w*, and by definition of* $i \in S^n_n$*,* $i \% t < w$*, so we can remove the* $i \% t$*.*

$= \left\lfloor \frac{i}{t} \right\rfloor * t + i' \% w$

$= \left\lfloor \frac{i}{t} \right\rfloor * t + (\left\lfloor \frac{i}{t} \right\rfloor * w + i \ \% \ t) \ \% \ w$
Expanded definition of i'

$= \left\lfloor \frac{i}{t} \right\rfloor * t + (i \ \% \ t) \ \% \ w$
$\lfloor x \rfloor * w$ is a multiple of w

$= \left\lfloor \frac{i}{t} \right\rfloor * t + i \ \% \ t$
By definition of $i \in S_n^n$, $i \ \% \ t < w$

$= i$

Ⓐ.2 To prove:

$i < u \wedge i \ \% \ t < w \rightarrow i' < u' \wedge i' \ \% \ t' < w'$

We will now assume $i < u \wedge i \ \% \ t < w$ to be true, and prove the right hand side of the implication in two parts: $i' < u'$, and $i' \ \% \ t' < w'$.

- $i' < u'$

  $\iff$

  $\left\lfloor \frac{i}{t} \right\rfloor * w + i \ \% \ t < \left\lfloor \frac{u}{t} \right\rfloor * w + min(u \ \% \ t, w)$
  Expand defintions of i' and u'

  $\Longleftarrow$

  We will split into three cases:

  - $\left\lfloor \frac{i}{t} \right\rfloor * w > \left\lfloor \frac{u}{t} \right\rfloor * w$

    This case is not possible, as by definition $i < u$.
  - $\left\lfloor \frac{i}{t} \right\rfloor * w = \left\lfloor \frac{u}{t} \right\rfloor * w$

    Because of this equality and the fact that $i < u$, we know that $i \ \% \ t < u \ \% \ t$. We also know that $i \ \% \ t < w$. So we can conclude that $i \ \% \ t < min(u \ \% \ t, w)$, and thus that:

    $\left\lfloor \frac{i}{t} \right\rfloor * w + i \ \% \ t < \left\lfloor \frac{u}{t} \right\rfloor * w + min(u \ \% \ t, w)$
  - $\left\lfloor \frac{i}{t} \right\rfloor * w < \left\lfloor \frac{u}{t} \right\rfloor * w$

    Because of this equality, we know that the difference between the two is at least w, so it is sufficient to prove that:

    $i \ \% \ t < min(u \ \% \ t, w) + w$

    This is trivially true, as by definition $i \ \% \ t < w$ is true.

- $i' \ \% \ t' < w'$

  By definition, $t' = 1$ and $w' = 1$, so this true for any i'.

Ⓑ Given $I \in S_n^n$. We now have to prove that if $CompressGrid^i_{\leftarrow d}(i') = i$ and $CompressGrid^i_{\leftarrow d}(i'') = i$, then $i = i''$. We prove this by contradiction. Assume there are two $i' \neq i''$ with $CompressGrid^i_{\leftarrow d}(i') = i$ and $CompressGrid^i_{\leftarrow d}(i'') = i$:

$i' \neq i'' \wedge compressgridi_d(i') = i \wedge CompressGrid^i_{\leftarrow d}(i'') = i$

$\iff$

$i' \neq i'' \wedge \left\lfloor \frac{i'}{w} \right\rfloor * t + i' \ \% \ w = i \wedge \left\lfloor \frac{i''}{w} \right\rfloor * t + i'' \ \% \ w = i$

Let's try to take $i' = i''$, and try to add/substract to i' so that $i' \neq i''$:

- If you want the left hand side of the addition to stay the same, the maximum you can add/substract to i' is $w - 1$
- If you want the right hand side of the addition to stay the same, you'd have to add/substract a multiple of t. Remember that $t \geq w$.

This means that it's impossible to change the value of i' with either the left hand side or the right hand side of the addition to change. Furthermore, it is also impossible to change both sides and get them to even out:

- The left hand side of the addition only changes with multiples of w
- The right hand side of the addition changes with a maximum of $w - 1$, as by definition $i \ \% \ t < w$.

■

Theorem 2.24 (CompressGrid is mapping-equivalent). *(compress-me)*

By definition 2.7, CompressGrid is mapping equivalent iff:

- *Ⓐ $CompressGrid^s_{\rightarrow}$ is well defined*
- *Ⓑ $CompressGrid^i_{\leftarrow}$ is well defined*
- *Ⓒ $CompressGrid^i_{\leftarrow}$ is operationally bijective*

Proof. Ⓐ is proven in lemma 2.21
Ⓑ is proven in lemma 2.22
Ⓒ is proven in lemma 2.23 ■

2.6 PruneGrid

PruneGrid removes the step and width from a normalized index space by mapping all indexes outside of the grid to $\perp$. This means that the mapping transforms any normalized index space into a dense index space.

Definition 2.25 (*PruneGrid*)**.** (def-prune) ■

- $\mathit{PruneGrid}^s_{\rightarrow} : {}^*\mathbb{S}^n_n \rightarrow {}^*\mathbb{S}^n_d$

  $\mathit{PruneGrid}^s_{\rightarrow}(U, T, W) = (U')$ with:

  - $U' = U$

- $\mathit{PruneGrid}^i_{\leftarrow} : S^{n}_d{}' \rightarrow S^n_n$

$$\mathit{PruneGrid}^i_{\leftarrow}(I') = \begin{cases} I' & | \quad I' \,\%\, T <^{\downarrow} W \\ \bot & | \quad I' \,\%\, T \not<^{\downarrow} W \end{cases}$$

Lemma 2.26 ($\mathit{PruneGrid}^s_{\rightarrow}$ is well defined)**.** *(prune-wds)*

Given any ${}^*S^n_n \in {}^*\mathbb{S}^n_n$. We now have to prove that $\mathit{PruneLB}^s_{\rightarrow}({}^*S^n_n) \in {}^*S^n_d$.

Proof. Using lemma 2.8, we just have to prove $U' \in \mathbb{N}^n$. We know $U \in \mathbb{N}^n$. We also know $U' = U$. So, we can conclude that $U' \in \mathbb{N}^n$ as well. ■

Lemma 2.27 ($\mathit{PruneGrid}^i_{\leftarrow}$ is well defined)**.** *(prune-wdi)*

Given any $I \in S^{n}_d{}'$. We now have to prove that $\mathit{PruneGrid}^i_{\leftarrow}(I') \in S^n_n$.

Proof. Using lemma 2.9, we can now prove for any given I' and $I = \mathit{PruneGrid}^i_{\leftarrow}(I')$:

$I' <^{\downarrow} U' \wedge I' \overrightarrow{\%}\, T' <^{\downarrow} W' \rightarrow$
$I = \bot \vee (I <^{\downarrow} U \wedge I \overrightarrow{\%}\, T <^{\downarrow} W)$

We now distinct two cases:

- $I' \,\%\, T <^{\downarrow} W$:

  $I' <^{\downarrow} U \wedge I' \overrightarrow{\%}\, T <^{\downarrow} W \rightarrow$
  $I' = \bot \vee (I' <^{\downarrow} U \wedge I' \overrightarrow{\%}\, T <^{\downarrow} W)$
  Insert definitions of $\mathit{PruneGrid}^s_{\rightarrow}$ and $\mathit{PruneGrid}^i_{\leftarrow}$

  $\iff$

  $I' <^{\downarrow} U \wedge I' \overrightarrow{\%}\, T <^{\downarrow} W \rightarrow$
  $I' <^{\downarrow} U \wedge I' \overrightarrow{\%}\, T <^{\downarrow} W$
  Choose right 'or' clause

- $I' \,\%\, T \not<^{\downarrow} W$:

  $\bot = \bot \vee (\bot <^{\downarrow} U \wedge \bot \overrightarrow{\%}\, T <^{\downarrow} W)$
  Insert definition of $\mathit{PruneGrid}^i_{\leftarrow}$, prune assumption

  $\iff$

  $\bot = \bot$
  Choose left 'or' clause

Lemma 2.28 ($\mathit{PruneGrid}^i_{\leftarrow}$ is operational bijective)**.** *(prune-opbi)*

Take the operational parts of $S^n_n : \mathcal{O}(S^n_n)$ and $S^{n}_d{}' : \mathcal{O}_{\mathit{PruneGrid}^i_{\leftarrow}}(S^n_n)$. We now have to prove that $\mathit{PruneGrid}^i_{\leftarrow} : \mathcal{O}_{\mathit{PruneGrid}^i_{\leftarrow}}(S^n_n) \rightarrow \mathcal{O}(S^n_n)$ is bijective.

Proof. Using lemma 2.10, we now have to prove $\textcircled{A}$ surjectivity and $\textcircled{B}$ injectivity.

$\textcircled{A}$ Given an $I \in \mathcal{O}(S^n_n)$. We take $I' = I$. We now have to prove that $\textcircled{A.1}$ $\mathit{PruneGrid}^i_{\leftarrow}(I') = I$ and $\textcircled{A.2}$ $I' \in S^{n}_d{}'$.

We know that $I \in \mathcal{O}(S^n_n)$. This means, by definition, that $I \leq U \wedge I \,\%\, T <^{\downarrow} W$ holds. From this, we can conclude that:

$\textcircled{A.1}$ $\mathit{PruneGrid}^i_{\leftarrow}(I') = I' = I$

$\textcircled{A.2}$ $U' = U \implies I' <^{\downarrow} U' \iff I' \in S^{n}_d{}'$

$\textcircled{B}$ Given $I \in \mathcal{O}(S^n_g)$ and $I', I'' \in S^{n}_n{}'$. We now have to prove that if $\mathit{PruneGrid}^i_{\leftarrow}(I') = I$ and $\mathit{PruneGrid}^i_{\leftarrow}(I'') = I$, then $I' = I''$. We prove this by contradiction. Assume there are two $I' \neq I''$ with $\text{PruneGrid}(I') = I$ and $\mathit{PruneGrid}^i_{\leftarrow}(I'') = I$:

$I' \neq I'' \wedge \text{PruneGrid}(I') = I \wedge \mathit{PruneGrid}^i_{\leftarrow}(I'') = I$

$\iff$

$I' \neq I'' \wedge I' = I \wedge I'' = I$ *By definition of $\mathit{PruneGrid}^i_{\leftarrow}$. $\mathit{PruneGrid}^i_{\leftarrow}(\bot) = \bot$, and $I \in \mathcal{O}(S^n_g)$, so I' and I'' cannot be $\bot$.*

This is a contradiction by transitivity of $=$. ↯ ■

Theorem 2.29 (PruneGrid is mapping-equivalent)**.** *(prune-me)*

By definition 2.7, PruneGrid is mapping equivalent iff:

- $\textcircled{A}$ *$\mathit{PruneGrid}^s_{\rightarrow}$ is well defined*
- $\textcircled{B}$ *$\mathit{PruneGrid}^i_{\leftarrow}$ is well defined*
- $\textcircled{C}$ *$\mathit{PruneGrid}^i_{\leftarrow}$ is operationally bijective*

Proof. $\textcircled{A}$ is proven in lemma 2.26
$\textcircled{B}$ is proven in lemma 2.27
$\textcircled{C}$ is proven in lemma 2.28 ■

2.7 SplitLast

SplitLast increases the dimensionality by splitting the last dimension in two. This means that the product of the lenghts of the two new dimensions is equal to the length of the original dimension.

SplitLast takes an extra argument $l \in \mathbb{N}$. We assume U_M to be a multiple of l and $l > 0$.

Definition 2.30 (*SplitLast*)**.** (def-split)

- We define the major and minor dimensions to be: $M = n - 1$ the major dimension $m = n$ the minor dimension
- $SplitLast^s_{\rightarrow} : \mathbb{N} \rightarrow {}^*S^n_d \rightarrow {}^*S^{n+1\prime}_d$

  $SplitLast^s_{\rightarrow}(l)(U) = U'$:

  For the definition of U' we separate dimensions as in definition 2.11.

$$u' = \begin{cases} u & \mid d < M \\ \dfrac{U_M}{l} & \mid d = M \\ l & \mid d = m \end{cases}$$

- $SplitLast^i_{\leftarrow} : S^{n+1\prime}_d \rightarrow S^n_d$

  $SplitLast^i_{\leftarrow}(l)(I') = I$:

  For the definition of I we separate dimensions as in definition 2.11.

$$i = \begin{cases} i' & \mid d < M \\ l * I'_M + I'_m & \mid d = M \end{cases}$$

Lemma 2.31 ($SplitLast^s_{\rightarrow}$ is well defined)**.** *(split-wds)*

Given any ${}^*S^n_d \in \mathbb{S}^n_d$. We now have to prove that $SplitLast^s_{\rightarrow}({}^*S^n_d) \in {}^*\mathbb{S}^{n+1\prime}_d$.

Proof. Using lemma 2.8 for S^n_d, we now only have to prove that $U' \in \mathbb{N}$. We will separate dimensions as in definition 2.11.

We distinct three cases:

- $d < M$

  In this case, $u' = u$, and we know that $u \in \mathbb{N}$, as ${}^*S^n_d \in \mathbb{S}^n_d$.
- $d = M$

  In this case, $u' = \dfrac{U_M}{l}$. By definition, U_M is a multiple of l, so $\dfrac{U_M}{l} \in \mathbb{N}$.
- $d = m$

  In this case, $u' = l$, which by definition comes from $\mathbb{N}$.

■

Lemma 2.32 ($SplitLast^i_{\leftarrow}$ is well defined)**.** *(split-wdi)*

Given any $I' \in {}^*S^{n+1\prime}_d$. We now have to prove that $SplitLast^i_{\leftarrow}(I') \in S^n_d$.

Proof. Using lemma 2.9 for codomain S^n_d, we can now prove for any given I' and $I = SplitLast^i_{\leftarrow}(I')$: $I' \in S^{n+1\prime}_d \rightarrow I \in S^n_d$. We separate dimensions as in definition 2.11$. For $d < M$, we have an identity mapping and we can use lemma 2.14$. For the rest of the proof, we can now assume that $d = M$.

$I' < U' \rightarrow i < u$

$\Longleftarrow$

$I'_M < U'_M \wedge I'_m < U'_m \rightarrow i < u$
We are only interested in dimensions M and m

$\Longleftrightarrow$

$I'_M < \dfrac{u}{l} \wedge I'_m < l \rightarrow l * I'_M + I'_m < u$
Expand definitions of i and U'

$\Longleftrightarrow$

$I'_M \leq \dfrac{u}{l} - 1 \wedge I'_m < l \rightarrow l * I'_M + I'_m < u$
As all parts are integers, we can rewrite the leftmost comparison

$\Longleftrightarrow$

$l * I'_M \leq u - l \wedge I'_m < l \rightarrow l * I'_M + I'_m < u$
Multiply leftmost comparison by l

This is true, as $l * I'_M + I'_m$ combined is less then $u - l + l$. ■

Lemma 2.33 ($SplitLast^i_{\leftarrow}$ is operational bijective)**.** *(split-opbi)*

Take the operational parts of S^n_d : $\mathcal{O}(S^n_d)$ and $S^{n+1\prime}_d$: $\mathcal{O}_{SplitLast^i_{\leftarrow}}(S^n_d)$. We now have to prove that $SplitLast^i_{\leftarrow} : \mathcal{O}_{SplitLast^i_{\leftarrow}}(S^n_d) \rightarrow \mathcal{O}(S^n_d)$ is bijective.

Proof. We separate dimensions as in definition 2.11. If $d < M$, we have an identity mapping and we will use lemma 2.14. For the rest of this proof, we can now assume that $d = M$.

Using lemma **??**, we now have to prove Ⓐ surjectivity and Ⓑ injectivity.

$\textcircled{A}$ Given an $i \in \mathcal{O}(S_d^n)_M$, $I'_M = \left\lfloor \frac{i}{l} \right\rfloor$, $I'_m = i \; \% \; l$ and $I'_d = I_d$ for any $d < M$. We now have to prove that $\textcircled{A.1}$ $SplitLast^i_{\leftarrow M}(I') = i$ and $\textcircled{A.2}$ $I' \in S_d^{n+1\prime}$.

$\textcircled{A.1}$ $SplitLast^i_{\leftarrow M}(I')$

$= l * \left\lfloor \frac{i}{l} \right\rfloor + i \; \% \; l$
Expanded definitions of $SplitLast^i_{\leftarrow}$ and I'

$= (i - i \; \% \; l) + i \; \% \; l$
Cancelled out the multiplication and division of l, reimplemented the flooring as a modulo operation

$= i$

$\textcircled{A.2}$ For dimensions $d < M$ this is trivial, as this is the identity mapping. We now only have to discuss dimensions M and m.

- Dimension M: $I_M < U_M \to I'_M < U'_M$

  We assume $I_M < U_M$ to be true, and prove $I'_M < U'_M$.

  $I'_M < U'_M$

  $\iff$

  $\left\lfloor \frac{I_M}{l} \right\rfloor < \frac{U_M}{l}$
  Expanded definitions of I'_M and U'_M

  $\iff$

  $\frac{I_M}{l} < \frac{U_M}{l}$
  Because $\left\lfloor \frac{I_M}{l} \right\rfloor < \frac{I_M}{l}$

  $I_M < U_M$
  As $l > 0$

$\textcircled{B}$ Given an $I \in S_d^n$ and $I', I'' \in S_d^{n+1\prime}$. We now have to prove that if $SplitLast^i_{\leftarrow d}(I') = i$ and $SplitLast^i_{\leftarrow d}(I'') = i$, then $I'_M = I''_M$ and $I'_m = I''_m$. We prove this by contradiction. Assume there are I', I'' with $I'_M neq I''_M$ or $I'_m \neq I''_m$, $SplitLast^i_{\leftarrow d}(I') = i$ and $SplitLast^i_{\leftarrow d}(I'') = i$:

$(I'_M \neq I''_M \lor I'_m \neq I''_m) \land SplitLast^i_{\leftarrow d}(I') = i \land SplitLast^i_{\leftarrow d}(I'') = i$

$\iff$

$(I'_M \neq I''_M \lor I'_m \neq I''_m) \land l * I'_M + I'_m = i \land l * I''_M + I''_m = i$
Expanded definition of $SplitLast^i_{\leftarrow}$

Let's compute I''_M and I''_m from I'_M and I'_M. If we want to have $l * I''_M + I''_m$ retain it's value, we have to either increase $l * I''_M$ and decrease I''_m by the same amount, or the other way around. However, by definition, $0 \leq I''_m < l$, and $l * I''_M$ can only be increased or decreased by multiples of l. Because of this, $I''_M = I'_M$ and $I''_m = I'_m$. $\lightning$ ■

Theorem 2.34 (SplitLast is mapping-equivalent)**.** *(split-me)*

By definition 2.7, SplitLast is mapping equivalent iff:

- $\textcircled{A}$ *$SplitLast^s_{\rightarrow}$ is well defined*
- $\textcircled{B}$ *$SplitLast^i_{\leftarrow}$ is well defined*
- $\textcircled{C}$ *$SplitLast^i_{\leftarrow}$ is operationally bijective*

Proof. $\textcircled{A}$ is proven in lemma 2.31
$\textcircled{B}$ is proven in lemma 2.32
$\textcircled{C}$ is proven in lemma 2.33 ■

2.8 FoldLast2

FoldLast2 is the inverse of *SplitLast*. It decreases the dimensionality by folding the last two dimensions together. This means that the length of the new dimension is equal to the product of the lengths of the last two original dimensions.

Definition 2.35 (*FoldLast2*)**.** (def-fold)

- We define the major and minor dimensions to be: $M = n - 2$ the major dimension and $m = n - 1$ the minor dimension.
- $FoldLast2^s_{\rightarrow} : {}^*S_d^n \to {}^*S_d^{n-1\prime}$

  $FoldLast2^s_{\rightarrow}(U) = U'$:

  For the definition of U' we separate dimensions as in definition 2.11.

$$u' = \begin{cases} u & \mid d < M \\ U_M * U_m & \mid d = M \end{cases}$$

- $FoldLast2^i_{\leftarrow} : S_d^{n-1\prime} \to S_d^n$

  $FoldLast2^i_{\leftarrow}(I') = I$:

  For the definition of I we separate dimensions as in definition 2.11.

$$i = \begin{cases} i' & \mid d < M \\ \left\lfloor \frac{i'}{U_m} \right\rfloor & \mid d = M \\ i' \; \% \; U_m & \mid d = m \end{cases}$$

Lemma 2.36 ($FoldLast2^s_\rightarrow$ is well defined)**.** *(fold-wds)*

Given any $^*S^n_d \in \mathbb{S}^n_d$*. We now have to prove that* $SplitLast^s_\rightarrow(^*S^n_d) \in {}^*\mathbb{S}^{n-1\prime}_d$*.*

Proof. Using lemma 2.8 for S^n_d, we now only have to prove that $U' \in \mathbb{N}$. We will separate dimensions as in definition 2.11.

We distinct two cases:

- $d < M$

  In this case, $u' = u$, and we know that $u \in \mathbb{N}$, as $^*S^n_d \in \mathbb{S}^n_d$.

- $d = M$

  In this case, $u' = U_M * U_m$. As both $U_M, U_m \in \mathbb{N}$, we know $u' \in \mathbb{N}$.

■

Lemma 2.37 ($FoldLast2^i_\leftarrow$ is well defined)**.** *(fold-wdi)*

Given any $I' \in {}^*S^{n-1\prime}_d$*. We now have to prove that* $FoldLast2^i_\leftarrow(I') \in S^n_d$*.*

Proof. Using lemma 2.9 for codomain S^n_d, we can now prove for any given I' and $I = FoldLast2^i_\leftarrow(I')$: $I' \in S^{n-1\prime}_d \rightarrow I \in S^n_d$. We separate dimensions as in definition 2.11\$. For $d < M$, we have an identity mapping and we can use lemma 2.14\$. For the rest of the proof, we only have to prove the cases Ⓐ $d = M$ and Ⓑ $d = m$.

Ⓐ $d = M$

$i' < u' \rightarrow i < u$

$\iff$

$i' < U_M * U_m \rightarrow \left\lfloor \frac{i'}{U_m} \right\rfloor < U_M$
Expand definitions of i and u'

$\iff$

$\frac{i'}{U_M} < U_m \rightarrow \left\lfloor \frac{i'}{U_m} \right\rfloor < U_M$
Divided left side of the implication by U_M

This holds, as $\lfloor X \rfloor \leq X$ for any X.

Ⓑ $d = m$

$I'_M < U'_M \rightarrow i < u$

$\iff$

$I'_M < U'_M \rightarrow i'_M \ \% \ U_m < U_m$
Expand definition of i

This holds, as $X \ \% \ Y < Y$ holds for any $X, Y \in N$. ■

Lemma 2.38 ($FoldLast2^i_\leftarrow$ is operational bijective)**.** *(fold-opbi)*

Take the operational parts of S^n_d : $\mathcal{O}(S^n_d)$ *and* $S^{n-1\prime}_d$: $\mathcal{O}_{FoldLast2^i_\leftarrow}(S^n_d)$*. We now have to prove that* $FoldLast2^i_\leftarrow : \mathcal{O}_{FoldLast2^i_\leftarrow}(S^n_d) \rightarrow \mathcal{O}(S^n_d)$ *is bijective.*

Proof. We separate dimensions as in definition 2.11. If $d < M$, we have an identity mapping and we will use lemma 2.14. For the rest of this proof, we can assume that $d = M$ or $d = m$.

Using lemma **??**, we now have to prove Ⓐ surjectivity and Ⓑ injectivity.

Ⓐ Given $I \in \mathcal{O}(S^n_d)$, $I'_M = U_m * I_M + I_m$ and $I'_d = I_d$ for any $d < M$. We now have to prove that $\boxed{A.1}$ $FoldLast2^i_\leftarrow(I') = I$ and $\boxed{A.2}$ $I' \in S^{n-1\prime}_d$.

$\boxed{A.1}$ We distinct three cases: $d < M$, $d = M$ and $d = m$.

- $d < M$

  In this case we have an identity mapping, and we can use lemma 2.14.

- $d = M$

  $FoldLast2^i_{\leftarrow M}(I')$

  $= \left\lfloor \frac{I'_M}{U_m} \right\rfloor$
  Expand definition of $FoldLast2^i_\leftarrow$

  $= \left\lfloor \frac{U_m * I_M + I_m}{U_m} \right\rfloor$
  Expand definition of I'_M

  $= \left\lfloor \frac{U_m * I_M}{U_m} \right\rfloor$
  As $U_m * I_M$ is a multiple of U_m and $I_m < U_m$, we can remove I_m from the equasion

  $= I_M$

- $d = m$

  $FoldLast2^i_{\leftarrow m}(I')$

  $= I'_M \ \% \ U_m$
  Expand definition of $FoldLast2^i_\leftarrow$

  $= (U_m * I_M + I_m) \ \% \ U_m$
  Expand definition of I'_M

  $= I_m \ \% \ U_m$
  As $U_m * I_M \ \% \ U_m = 0$, we can remove it

  $= I_m$
  As $I_m < U_m$ by definition of $I \in S^n_d$

$\textcircled{A.2}$ For dimensions $d < M$ this is trivial, as this is the identity mapping. We now only have to discuss dimension M.

$I_M < U_M \land I_m < U_m \to I'_M < U'_M$

$\iff$

$I_M < U_M \land I_m < U_m \to U_m * I_M + I_m < U_M * U_m$
Expand definitions of I'_M and U'_M

$\Longleftarrow$

$I_M < U_M \land I_m < U_m \to$
$U_m * I_M \leq U_M * U_m - U_m \land I_m < U_m$
Split the smaller then into two smaller smaller then's

$\iff$

$I_M < U_M \land I_m < U_m \to I_M \leq U_M - 1 \land I_m < U_m$
Divide by U_m

This is trivially true.

Ⓑ Given an $I \in S^n_d$ and $I', I'' \in S^{n-1\prime}_d$. We now have to prove that if $FoldLast2^i_{\leftarrow M}(I') = I_M$, $FoldLast2^i_{\leftarrow M}(I'') = I_M$, $FoldLast2^i_{\leftarrow m}(I') = I_m$ and $FoldLast2^i_{\leftarrow m}(I'') = I_m$, then $I'_M = I''_M$. We prove this by contradiction. Assume there are I', I'' with $I'_M \neq I''_M$ for which this holds:

$I'_M \neq I''_M \land$
$FoldLast2^i_{\leftarrow M}(I') = I_M \land FoldLast2^i_{\leftarrow M}(I'') = I_M \land$
$FoldLast2^i_{\leftarrow m}(I') = I_m \land FoldLast2^i_{\leftarrow m}(I'') = I_m$

$\iff$

$I'_M \neq I''_M \land$
$\left\lfloor \frac{I'_M}{U_m} \right\rfloor = I_M \land \left\lfloor \frac{I''_M}{U_m} \right\rfloor = I_M \land$
$I'_M \,\%\, U_m = I_m \land I''_M \,\%\, U_m = I_m$
Expand definition of $FoldLast2^i_{\leftarrow}$

For $\left\lfloor \frac{I'_M}{U_m} \right\rfloor = I_M \land \left\lfloor \frac{I''_M}{U_m} \right\rfloor = I_M$ to hold, I'_M and I''_M can differ at most $U_m - 1$.

For $I'_M \,\%\, U_m = I_m \land I''_M \,\%\, U_m = I_m$ to hold, I'_M and I''_M must differ an exact multiple of U_m.

Combined, this ensures that $I'_M = I''_M$, which contradicts with $I'_M \neq I''_M$. ↯

■

Theorem 2.39 (FoldLast2 is mapping-equivalent)**.** *(fold-me)*

By definition 2.7, FoldLast2 is mapping equivalent iff:

- Ⓐ *$FoldLast2^s_{\rightarrow}$ is well defined*
- Ⓑ *$FoldLast2^i_{\leftarrow}$ is well defined*
- Ⓒ *$FoldLast2^i_{\leftarrow}$ is operationally bijective*

Proof. Ⓐ is proven in lemma 2.31
Ⓑ is proven in lemma 2.32
Ⓒ is proven in lemma 2.33 ■

2.9 Permute

Permute changes the order of dimensions, preparing it for a mapping operating on specific dimensions, such as *FoldLast*2 or *PadLast*. The mapping does not change the dimensions themselves.

Permute takes an extra argument $P \in \mathbb{Z}^n$. We assume P to be a valid permuatation of $0..[n-1]$.

Definition 2.40 (*Permute*)**.** (def-permute)

- $Permute^s_{\rightarrow} : \mathbb{Z}^n \to {}^*\mathbb{S}^n_g \to {}^*\mathbb{S}^n_g$

  $Permute^s_{\rightarrow}(P)(L, U, T, W) = (L', U', T', W')$:

  For definitions of L', U', T', W', we separate dimensions as in definition 2.11.

  - $l' = L_{p'}$
  - $u' = U_{p'}$
  - $t' = T_{p'}$
  - $w' = W_{p'}$

- $Permute^i_{\leftarrow} : S^{n\prime}_d \to S^n_n$

  $Permute^i_{\leftarrow}(P)(I') = I$:

  For definition of I, we separate dimensions as in definition 2.11.

  - let $p' \in \mathbb{Z}$ with $P_{p'} = d$
    $i = I'_{p'}$

Lemma 2.41 ($Permute^s_{\rightarrow}$ is well defined)**.** *(permute-wds)*

Given any ${}^*S^n_g \in {}^*\mathbb{S}^n_g$. We now have to prove that $Permute^i_{\leftarrow}({}^*S^n_g) \in {}^*\mathbb{S}^n_g$.

Proof. Using lemma 2.8 for S^n_g, we now have to prove that Ⓐ $L', U', T', W' \in \mathbb{Z}^n$ and Ⓑ $L \leq U$ and $W \leq T$. We separate dimensions as in definition 2.11.

Ⓐ We know that $l = L_p$. By definition, $L_p \in \mathbb{Z}$. So, we can conclude that $l \in \mathbb{Z}$ also holds. Similarly, $u, t, w \in \mathbb{Z}$ also hold.

Ⓑ We know that $l = L_p$ and $u = U_p$. By definition, $L_p \leq U_p$. So, we can conclude that $l \leq u$ also holds. Similarly, $w \leq t$ also holds. ■

Lemma 2.42 ($Permute^i_{\leftarrow}$ is well defined)**.** *(permute-wdi)*

Given any $I' \in S^{n\prime}_g$. We now have to prove that $Permute^i_{\leftarrow}(I') \in S^n_g$.

Proof. Using lemma 2.9, we can now prove for any given I' and $I = PruneGrid^i_{\leftarrow}(I')$. We split separate dimensions for I.

$L' \leq I' <^{\downarrow} U' \wedge (I' - L') \overrightarrow{\%\, T'} <^{\downarrow} W' \rightarrow$
$i = \bot \vee (l \leq i <^{\downarrow} u \wedge (i-l) \overrightarrow{\%\, t} <^{\downarrow} w)$

$\iff$

$L' \leq I' <^{\downarrow} U' \wedge (I' - L') \overrightarrow{\%\, T'} <^{\downarrow} W' \rightarrow$
$I'_{p'} = \bot \vee (L'_{p'} \leq I'_{p'} <^{\downarrow} U'_{p'} \wedge (I'_{p'} - L'_{p'}) \overrightarrow{\%\, T'_{p'}} <^{\downarrow} W'_{p'})$
By definition of $Permute^i_{\leftarrow}$, with $P_{p'} = d$. Because P is a permutation, we know such a p' exists.

This holds, as the right hand side of the implication is equal to the left hand side in a single dimension. ■

Lemma 2.43 ($Permute^i_{\leftarrow}$ is operational bijective)**.** *(permute-opbi)*

Take the operational parts of S^n_g : $\mathcal{O}(S^n_g)$ and $S^{n\prime}_g$: $\mathcal{O}_{Permute^i_{\leftarrow}}(S^n_g)$. We now have to prove that $Permute^i_{\leftarrow} : \mathcal{O}_{Permute^i_{\leftarrow}}(S^n_g) \rightarrow \mathcal{O}(S^n_g)$ is bijective.

Proof. Using lemma 2.10, we now have to prove $\textcircled{A}$ surjectivity and $\textcircled{B}$ injectivity.

$\textcircled{A}$ Given an $I \in \mathcal{O}(S^n_g)$. For each $d' \in [0..n-1]$, we take I' with $\textcircled{1}$ (tfi-opbi-Ip) $I'_{d'} = I_{P_{d'}}$. We now have to prove that $\textcircled{A.1}$ $Permute^i_{\leftarrow}(I') = I$ and $\textcircled{A.2}$ $I' \in S^{n\prime}_g$.

$\textcircled{A.1}$ By definition, for each $d \in [0..n-1]$ we have a $p' \in \mathbb{Z}$ with $\textcircled{2}$ (tfi-opbi-Pp) $P_{p'} = d$. We have to prove that $I_d = I'_{p'}$.

I_d
$= I'_{p'}$
$= I_{P_{p'}}$ $\textcircled{1}$
$= I_d$ $\textcircled{2}$

$\textcircled{A.2}$ We separate dimensions using definition 2.11 for I'. By definition, we now need to prove that:

$i' = \bot \vee (l' \leq i' < u' \wedge (i' - l') \ \%\ t' < w')$

$\iff$

$L_{p'} \leq I_{P_d} < U_{p'} \wedge (I_{p'} - L_{p'}) \ \%\ T_{p'} < W_{p'}$
Truncated left hand side of the 'or,' expanded definition of $Permute^s_{\rightarrow}$ for l', u', t', w', expanded definition of i' $\textcircled{1}$

$\iff$

$L_{p'} \leq I_{p'} < U_{p'} \wedge (I_{p'} - L_{p'}) \ \%\ T_{p'} < W_{p'}$

This holds, as it holds for any dimension $p' \in [0..n-1]$, and $p' \in [0..1]$ as P is a permutation of $[0..n-1]$.

$\textcircled{B}$ Given $I \in \mathcal{O}(S^n_g)$ and $I', I'' \in S^{n\prime}_g$. We now have to prove that if $Permute^i_{\leftarrow}(I') = I$ and $Permute^i_{\leftarrow}(I'') = I$, then $I' = I''$. We prove this by contradiction. Assume there are two $I' \neq I''$ with $Permute^i_{\leftarrow}(I') = I$ and $Permute^i_{\leftarrow}(I'') = I$. We separate dimensions for I' and I'' using definition 2.11\$

$i' \neq i'' \wedge Permute^i_{\leftarrow}(i') = i \wedge Permute^i_{\leftarrow}(i'') = i$

$\iff$

$i' \neq i'' \wedge P_{p'} = d \wedge I'_{p'} = i \wedge I''_{p'} = i$

$\iff$

$I' \neq I'' \wedge P_{p'} = d \wedge I' = I \wedge I'' = I$
As P is a permutation, the above holds for all $p' \in [0..n-1]$.

This is a contradiction by transitivity of $=$ ↯ ■

Theorem 2.44 (Permute is mapping-equivalent)**.** *(permute-me)*

By definition 2.7, Permute is mapping equivalent iff:

- *$\textcircled{A}$ $Permute^s_{\rightarrow}$ is well defined*
- *$\textcircled{B}$ $Permute^i_{\leftarrow}$ is well defined*
- *$\textcircled{C}$ $Permute^i_{\leftarrow}$ is operationally bijective*

Proof. $\textcircled{A}$ is proven in lemma 2.41
$\textcircled{B}$ is proven in lemma 2.42
$\textcircled{C}$ is proven in lemma 2.43 ■

2.10 Padlast

PadLast extends the length of the last dimension to a multiple of a given integer. This is to prepare it for the GPU warp size, or to be split using *SplitLast*.

PadLast takes an extra argument $p \in \mathbb{Z}$ to indicate the size of the padding.

Definition 2.45 (*PadLast*)**.** (def-pad)

- $PadLast^s_{\rightarrow} : \mathbb{Z} \rightarrow {}^*\mathbb{S}^n_g \rightarrow {}^*\mathbb{S}^n_g$

  $PadLast^s_{\rightarrow}(p)(L, U, T, W) = (L', U', T', W')$:

  - $L' = L$
  - $T' = T$
  - $W' = W$
  - For the definition of U', we separate dimensions as in definition 2.11.

$$u' = \begin{cases} u & | \quad d \neq n-1 \\ \left\lceil \dfrac{u}{p} \right\rceil * p & | \quad d = n-1 \end{cases}$$

- $PadLast^i_{\leftarrow} : S^{n\prime}_g \to S^n_g$

  $PadLast^i_{\leftarrow}(p)(I') = I$:

  –

  $$I = \left\{ \begin{array}{ccc} I' & | & I' <^{\downarrow} U \\ \perp & | & I' \not<^{\downarrow} U \end{array} \right.$$

Lemma 2.46 ($PadLast^s_{\rightarrow}$ is well defined)**.** *(pad-wds)*

Given any ${}^*S^n_g \in {}^*\mathbb{S}^n_g$. We now have to prove that $PadLast^s_{\rightarrow}({}^*S^n_g) \in {}^*S^n_g$.

Proof. Using lemma 2.8 for S^n_g, we now have to prove that $\textcircled{A}$ $L', U', T', W' \in \mathbb{Z}^n$ and $\textcircled{B}$ $L \leq U$ and $W \leq T$. We separate dimensions as in definition 2.11.

$\textcircled{A}$ By definition of $PadLast^s_{\rightarrow}$: $L' = L, T' = T, W' = W$. Because $L, T, W \in \mathbb{Z}$, $L', T', W' \in \mathbb{Z}$ as well. Similarly, $U' = \left\lceil \dfrac{U}{p} \right\rceil * 32$, so $U' \in \mathbb{Z}$ as well.

$\textcircled{B}$ We know that $L' = L$, $U' \geq U, T' = T, W' = W$. We also know that $L \leq U$ and $W \leq T$. Together, this proves $L' \leq U'$ and $W' \leq T'$. ■

Lemma 2.47 ($PadLast^i_{\leftarrow}$ is well defined)**.** *(pad-wdi)*

Given any $I' \in S^{n\prime}_g$. We now have to prove that $PadLast^i_{\leftarrow}(I') \in S^n_g$.

Proof. Using lemma 2.9, we can now prove for any given I' and $I = PadLast^i_{\leftarrow}(I')$. We distinct two cases:

- $I' <^{\downarrow} U$

  $L' \leq I' <^{\downarrow} U' \wedge (I' - L') \overrightarrow{\%} T' <^{\downarrow} W' \to$
  $I = \perp \vee (L \leq I <^{\downarrow} U \wedge (I - L) \overrightarrow{\%} T <^{\downarrow} W)$

  $\iff$

  $L \leq I' <^{\downarrow} U \wedge (I' - L) \overrightarrow{\%} T <^{\downarrow} W \to$
  $L \leq I' <^{\downarrow} U \wedge (I' - L) \overrightarrow{\%} T <^{\downarrow} W$

  – *Expand definition of $PadLast^s_{\rightarrow}$ for L', T', W'.*
  – *Replace $I' <^{\downarrow} U'$ with distinction assumption $I' <^{\downarrow} U$.*
  – *Prune left hand side of the 'or'*
  – *Expand definition of $PadLast^i_{\leftarrow}$ for I using $I' <^{\downarrow} U'$*

- $I' \not<^{\downarrow} U$

  $L' \leq I' <^{\downarrow} U' \wedge (I' - L') \overrightarrow{\%} T' <^{\downarrow} W' \to$
  $I = \perp \vee (L \leq I <^{\downarrow} U \wedge (I - L) \overrightarrow{\%} T <^{\downarrow} W)$

  $\iff$

  $L' \leq I' <^{\downarrow} U' \wedge (I' - L') \overrightarrow{\%} T' <^{\downarrow} W' \to$
  $\perp = \perp$
  Expand definition of $PadLast^i_{\leftarrow}$ for I using $I' \not<^{\downarrow} U'$, prune right hand side of the 'or'

■

Lemma 2.48 ($PadLast^i_{\leftarrow}$ is operational bijective)**.** *(pad-opbi)*

Take the operational parts of $S^n_g : \mathcal{O}(S^n_g)$ and $S^{n\prime}_g : \mathcal{O}_{PadLast^i_{\leftarrow}}(S^n_g)$. We now have to prove that $PadLast^i_{\leftarrow} : \mathcal{O}_{PadLast^i_{\leftarrow}}(S^n_g) \to \mathcal{O}(S^n_g)$ is bijective.

Proof. Using lemma 2.10, we now have to prove $\textcircled{A}$ surjectivity and $\textcircled{B}$ injectivity.

$\textcircled{A}$ Given an $I \in \mathcal{O}(S^n_g)$ and $I' \in S^n_g$ with $I' = I$. We now have to prove that $\textcircled{A.1}$ $PadLast^i_{\leftarrow}(I') = I$ and $\textcircled{A.2}$ $I' \in S^{n\prime}_g$.

$\textcircled{A.1}$ By definition 2.45 of $PadLast^i_{\leftarrow}$, $PadLast^i_{\leftarrow}(I') = I \iff I' <^{\downarrow} U$. We know that I' = I, and by definition of $I \in \mathcal{O}(S^n_g)$, $I <^{\downarrow} U$. So, we can conclude that $PadLast^i_{\leftarrow}(I') = I$ holds.

$\textcircled{A.2}$

$L \leq I <^{\downarrow} U \wedge (I - L) \ \% \ T <^{\downarrow} W \to$
$L' \leq I' <^{\downarrow} U' \wedge (I' - L') \ \% \ T' <^{\downarrow} W'$

$\iff$

$L' \leq I <^{\downarrow} U \wedge (I - L') \ \% \ T' <^{\downarrow} W' \to$
$L' \leq I <^{\downarrow} U' \wedge (I - L') \ \% \ T' <^{\downarrow} W'$
By expanding the definition of $PadLast^s_{\rightarrow}$ for L, T, W and the definition of I'

This holds, as U can only be bigger then U'.

$\textcircled{B}$ Given $I \in \mathcal{O}(S^n_g)$ and $I', I'' \in S^{n\prime}_g$. We now have to prove that if $PadLast^i_{\leftarrow}(I') = I$ and $PadLast^i_{\leftarrow}(I'') = I$, then $I' = I''$. We prove this by contradiction. Assume there are two $I' \neq I''$ with $PadLast^i_{\leftarrow}(I') = I$ and $PadLast^i_{\leftarrow}(I'') = I$.

$I' \neq I'' \wedge PadLast^i_{\leftarrow}(I') = I \wedge PadLast^i_{\leftarrow}(I'') = I$

$\iff$

$I' \neq I'' \wedge I' = I \wedge I'' = I$
$PadLast^i_{\leftarrow}(I')$ cannot be $\perp$, as $\perp \notin \mathcal{O}(S^n_g)$. Similarly for I''.

This is a contradiction by transitivity of = $\lightning$ ■

Theorem 2.49 (PadLast is mapping-equivalent)**.** *(pad-me)*

By definition 2.7, PadLast is mapping equivalent iff:

- *$\textcircled{A}$ $PadLast^s_{\rightarrow}$ is well defined*

- $\textcircled{B}$ *$PadLast^{i}_{\leftarrow}$ is well defined*
- $\textcircled{C}$ *$PadLast^{i}_{\leftarrow}$ is operationally bijective*

Proof. $\textcircled{A}$ is proven in lemma 2.46
$\textcircled{B}$ is proven in lemma 2.47
$\textcircled{C}$ is proven in lemma 2.48 ∎

Chapter 3

Implementation

The implementation of these mappings was done in the SAC compiler [1], which is implemented in C. The compiler takes a SAC program as input, generates C code, and uses a C or cuda C compiler to create an executable. We will distinct the implementation of the SAC compiler in three different parts: The existing compiler, the implementation of the mappings as defined in chapter **??**, and the mechanisms that determine what mappings will be executed, in what order, and with what parameters. We will discuss each part in it's own section below.

3.1 The SAC compiler

First of all, there is the existing compiler. The part we're mainly interested in is the part with Jing's implementation for executing SAC with-loops on the GPU. This implementation is fairly limited and has many cases where a with-loop cannot be executed. If such a case occurs, it either results in a compiler or runtime error, while the program would have been valid in the normal sequential compiler backend.

AST Traversals

When the compiler compiles a SAC program for the CUDA backend, it will do so using many different AST traversals. We will discuss the traversals relevant to the execution of with-loops on the GPU using a small code example in figure **??** 3.1. Note that some of the code examples use a bit of pseudo code to illustrate internal AST representations. Also note that this example will probably not work like this when compiled, as some optimizations may take out the loop altogether.

```
int[.,.] plusone(int[.,.] a) {
  b = with {
    (0 <= iv < shape(a)) : a[iv] + 1;
  } : genarray (shape(a), 0);
  return b;
}
```
(trav-ex)

code example 3.1: Example SAC program

- Annotate cudarizable loops: This compiler phase determines whether a with-loop is eligible for execution on a GPU. Examples of with-loops not eligible are very small loops, nested with-loops, or with loops with non-cudarizable function calls in their bodies. This compiler phase sets a boolean flag in with-loop AST nodes. Note that the dimensionality is always known. When a cudarizable function has an unknown dimensionality, e.g. it can be used with any dimensionality, a separate instance of this function is created for each used dimensionality.

- Insert primitives to prepare for GPU execution: These are multiple phases responsible for transfering variables, constants and arrays to the GPU before execution, and the generated array back to the CPU after execution. Figure 3.2 displays the state of the example program after this phase.

- Create CUDA kernels: Create a CUDA kernel function for each partition in the with loop. Note that multiple partitions can exist within a with-loop. Each kernel function will contain the body of the partition it was created for. Information as the lowerbound, upperbound, step and width will be passed in as an argument, as well as GPU pointers to the arrays. The current iv can be derived from the variables threadIdx and blockIdx, which both contain the properties x, y and z. Figure 3.3 displays the state of the example program after this phase.

- Generate C code: The actual generated C code does not contain any pseudocode anymore, but should be compilable by the C-compiler.

```
int[.,.] plusone(int[.,.] a) {
  a_d = cudaMemCpyHost2Dev(a);
  b_d = cudaMemCreate(...);

  b_d = with {
    (0 <= iv < shape(a_d)) : a_d[iv]+1;
  } : genarray (shape(a_d), 0);

  b = cudaMemCpyDev2Host(a_d);
  cudaMemFree(a_d);
  cudaMemFree(b_d);

  return b;
}
```
(trav-mem)

code example 3.2: Example SAC program after memory transfers have been inserted

Grid and block

In figure 3.3, there are a few lines of code left undefined. The most important of these are the lines annotated with the comments *Compute CUDA grid and block sizes* and *Recompute iv from threadIdx and blockIdx*. The original implementation is pretty straightforward. Let us define the lengths of the dimensions as $d_0 \ldots d_n$. Now we make a case distincion for the first five dimensionalities, and map them as described in table 3.4. For 6 or more dimensions, the current implementation gives a compiler error. Note that the block size has a maximum 1024 threads (may vary per GPU), so $d_0 * d_1$ can be at most 1024. If it is not, a runtime error is thrown.

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. For iv_0, we do this as shown in code example 3.5. iv_1 is computed similarly. Note that in the generated C code, these values do not exist as an array. Instead, all values for each dimensions have their own variables.

Lowerbound, upperbound, step and width

In table 3.4, we used dimension lengths $d_0 \ldots d_n$. However, in our C program we have $lb_0 \ldots lb_n$, $ub_0 \ldots ub_n$, $step_0 \ldots step_n$ and $width_0 \ldots width_n$. This means we

```
int[.,.] plusone(int[.,.] a) {
  a_d = cudaMemCpyHost2Dev(a);
  b_d = cudaMemCreate(...);

  // Compute CUDA grid and block sizes
  grid = ...
  block = ...

  plusone_kernel<grid, block>(
      lb_0, lb_1, ub_0, ub_1, a_d, b_d);

  b = cudaMemCpyDev2Host(a_d);
  cudaMemFree(a_d);
  cudaMemFree(b_d);

  return b;
}

void plusone_kernel_0 (
    lb_0, lb_1, ub_0, ub_1, a_d, b_d) {
  // Recompute iv from
  // threadIdx and blockIdx
  iv = ...

  b_d[iv] = a_d[iv] + 1;
}
```
(trav-kernel)

code example 3.3: Example SAC program after kernels have been generated

have to make a few slight changes still:

$\forall i < dims$:

- We take the upperbound as the dimension lengths

```
    // Computing grid and block sizes
    d_i = ub_i
```

- We substract the lowerbound from the dimension length. Note that we have to add it again inside the kernels.

```
    // Computing grid and block sizes
    d_i = d_i - lb_i

    // Recomputing the index vector
    iv_i = iv_i + lb_i
```

- The step and width do not influence the lengths of the dimensions. Instead, we will check inside of the kernel whether iv is inside the grid:

```
    // Recomputing the index vector
    if (iv_i % step_i > width_i)
        return;
```

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

(jing-mapping)

table 3.4: Case distinction for mapping n-dimensional index spaces onto the GPU, using Jing's heuristics

```
iv_0 = blockIdx.x * 32 + threadIdx.x;
if (iv_0 >= d_0) return;
```

(trav-kernel-iv)

code example 3.5: Recomputation of iv_0 in the kernel, for dimensionalities 1 and 2. iv_1 is computed similarly.

If we combine all this together, we get code example 3.6 as output of the compiler:

Optimizations

As SAC is a functional language, the SAC compiler can do many different optimizations that are unique to functional languages. For example, the SAC compiler will, in this case, probably decide that the memory of variable a can be reused. This means that no b or b_dev have to be allocated. Furthermore, the SAC compiler may just omit the with-loop altogether, and replace it with an accessor that adds one to the original array value. The code fragments we have been looking at merely serve as an example, so there are no guarantees that the SAC compiler will, in this case, behave exactly as illustrated here.

3.2 Mapping execution

The mappings introduced in **??** change how the grid and block spaces are generated, and how the iv is computed from the grid and block coordinates. This also means that the mapping execution implementation only changes the generated code responsible for those two tasks. In code example 3.3, this would be lines 5-7 and 21-23.

Differences from theoretical model

Because C works a bit different then theoretical maths, the C implementation differs a bit from the theoretical model we implemented in **??**. Besides these implementation details, there are also a few conceptual differences in the implementation.

Grid index spaces

The representation of the grid index spaces in the compiler are very similar to the definition of a grid index space as defined in section 3.2. There are, however, a few small differences.

- The arrays of I, L, U, T and W variables only exist in the compiler. In the generated code, the values are represented either in individual variables or, if possible, as constants.
- The filter function f does not exist in the runtime. The check is immediately executed alongside of the mappings, and if it fails we either return immediately in branching mode, or we set an "outside of grid" variable in branchless mode. More on branching/branchless modes in section TODO.

3.3 Mapping generation using strategies

```
int[.,.] plusone(int[.,.] a) {
  // Assume all lb_i, ub_i, step_i
  // and width_i variables exist

  a_d = cudaMemCpyHost2Dev(a);
  b_d = cudaMemCreate(...);

  // Compute CUDA grid and block sizes
  grid.x = (ub_0 - lb_0) / 32;
  grid.y = (ub_1 - lb_1) / 32;
  block.x = 32;
  block.y = 32;

  plusone_kernel<grid, block>(
      lb_0, lb_1, ub_0, ub_1, a_d, b_d);

  b = cudaMemCpyDev2Host(a_d);
  cudaMemFree(a_d);
  cudaMemFree(b_d);

  return b;
}

void plusone_kernel_0 (
    lb_0, lb_1, ub_0, ub_1, a_d, b_d) {
  // Recompute iv from
  // threadIdx and blockIdx
  iv_0 = blockIdx.x * 32 + threadIdx.x;
  iv_0 = iv_0 + lb_0
  if (iv_0 >= ub_0) return;
  // No step or width, so check omitted

  iv_1 = blockIdx.y * 32 + threadIdx.y;
  iv_1 = iv_1 + lb_1
  if (iv_1 >= ub_1) return;
  // No step or width, so check omitted

  iv = {iv_1, iv_0};

  b_d[iv] = a_d[iv] + 1;
}
```
(trav-complete)

code example 3.6: Example SAC program after it has been compiled

Chapter 4

Listings

List of figures

List of tables

List of code examples

List of definitions

List of lemmas

List of theorems

References

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

[2] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. “Breaking the GPU programming barrier with the auto-parallelising SAC compiler”. In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. 2011, pp. 15–24.

[3] Hee Sik Kim J. Neggers. *Basic posets*, pp. 64–78. ISBN: 9789810235895. URL: `https://books.google.nl/books?id=-ip3-wejeR8C&pg=PA64&redir_esc=y#v=onepage&q&f=false`.