Generating SIMD Instructions for Cerebras CS-1 using Polyhedral Compilation Techniques

Sven Verdoolaege
Cerebras Systems
sven@cerebras.net

Manjunath Kudlur
Cerebras Systems
manjunath@cerebras.net

Rob Schreiber
Cerebras Systems
rob.schreiber@cerebras.net

Harinath Kamepalli
Cerebras Systems
harinath@cerebras.net

Abstract

The Cerebras CS-1 is a computing system based on a wafer-scale processor having nearly 400,000 compute cores. It is intended for training of and inference on deep neural networks. The architecture has several features specifically designed for this and related fields. One of these is a sophisticated SIMD engine that can mimic a rectangular loop nest of depth at most four. In order to achieve optimal performance, it is crucial to use SIMD instructions as much as possible.

This paper describes a high-level polyhedral compiler that takes a high-level algorithm description that can be written manually or extracted from a TensorFlow computation graph and generates input to the low-level C-based compiler. In this intermediate code, the use of SIMD instructions is made explicit. The main focus of the paper is the generation of these CS-1 SIMD instructions for convolution style algorithms. What complicates the task is that the set of computation instances that need to be performed may not at first sight look like they form a rectangular loop nest. The basis of the compilation is formed by an effective combination of relatively well-known, but more specialized polyhedral operations.

1 Introduction

Deep learning networks are well recognized as a prime target for polyhedral compilation techniques (Baghdadi et al. 2019; Lattner and Pienaar 2019; Pradelle et al. 2019; Vasilache et al. 2019; Zerrell and Bruestle 2019). It is then no surprise that Cerebras is also developing a polyhedral compiler called `dtg_codegen` that takes a high-level description of part of a deep learning network as input and produces low-level C code for implementing that part on a subset of the PEs (Processing Elements). A different part of the compilation flow, not discussed in this paper, is responsible for breaking up the network into these parts and for selecting the subsets of PEs on which to implement them, thus providing the main inputs to `dtg_codegen`.

Besides a description of the algorithm, `dtg_codegen` also takes information about where and in what order the input tensor elements arrive and, similarly, where and how the output tensor elements should be produced. In the current state of development, the PE where each computation instance needs to be executed is also given as input to `dtg_codegen`. The tool is then responsible for routing the data to the PEs that need them and for generating the code for processing the data on each PE.

One of the challenges in this code generation process is detecting opportunities for generating SIMD (Single Instruction Multiple Data) instructions. Note that the process of generating SIMD instructions for Cerebras CS-1 is quite different from that for CPUs (Feld et al. 2013; Henretty et al. 2013; Kong et al. 2013; Sharma et al. 2015; Hallou et al. 2017), as there is no need to ensure stride-0 or stride-1 accesses, to compute tile sizes or to perform data layout transformations. Instead, a CS-1 SIMD instruction can mimic a rectangular loop nest performing arbitrary affine accesses. It is therefore important to try and represent the set of computation instances as a rectangular domain. As will be explained below, the description of the input algorithm only allows rectangular domains, so this may seem like a trivial task. However, the instances that need to be executed when a piece of data arrives at a PE form a slice of the total set of instances and the orientation of this slice is not necessarily in an orthogonal direction, especially for convolution style algorithms. The way the instances are mapped to the PEs may be a further cause of complications. Since these SIMD instructions replace a rectangular loop nest, their detection conceptually bears some similarities with the replacement of part of the code by library calls (Alias and Barthou 2005; Lu et al. 2012; Iooss 2016; Vasilache et al. 2019), but the way the detection is performed is quite different.

2 Polyhedral Compilation Background

For its core polyhedral compilation operations, `dtg_codegen` relies on `isl` (Verdoolaege 2010), a library for manipulating sets of integer tuples described by Presburger formulas. Recall that a *Presburger formula* is a first order logic formula involving affine expressions, equality (=) and the less-than-or-equal relation (≤). An *affine expression* is a linear expression plus a constant term. In the case of `isl`, the affine expressions may involve the tuple variables, existentially quantified variables, as well as symbolic constants. Such symbolic constants have a fixed but unknown value. For example,

$$\{\, \mathrm{S}[i,j] : \exists \alpha : i,j \geq 0 \wedge i = 2\alpha + 1 \wedge i + j \leq \mathrm{N} \,\} \tag{1}$$

describes elements in a two-dimensional S-space where both coordinates are non-negative, the first coordinate is odd and the sum is less than or equal to some symbolic constant N. Besides sets, `isl` also supports binary relations on integer tuples and explicit functions on sets. These functions may be piecewise quasi-affine. Quasi-affine means that the affine expressions may involve integer divisions of other (quasi-)affine expressions. Piecewise means that the domain may be subdivided into disjoint pieces each with its own associated (quasi-)affine expression. An example of a binary relation is

$$\{\, [x] \to [y] : 0 \leq y < x \leq 10 \,\}, \tag{2}$$

relating elements from an interval between 0 and 10 to elements in the same interval that are smaller. An example of a piecewise quasi-affine function is

$$\{\, [i] \to [i] : i \geq 0 \,\} \cup \{\, [i] \to [-i] : i < 0 \,\}, \tag{3}$$

representing the absolute value. Note that → is used to separate the domain of a function from its value (3), but also to separate the tuples in a binary relation (2).

Verdoolaege (2016) provides a detailed description of the `isl` notations used in this paper and of operations that can be performed on `isl` objects. There are two notations that are not yet covered by this tutorial:

- Inside a tuple, the value of a variable can be described as *lower*:*upper*, meaning that this variable has a value between *lower* and *upper* (inclusive). For example, $\{\, [x = 0{:}10] \,\}$ is equivalent to $\{\, [x] : 0 \leq x \leq 10 \,\}$ This notation is borrowed from `Omega` (Kelly et al. 1996). In the `isl` implementation, however, the lower and/or upper bound may be omitted.
- The `//` operator represents integer division. For example, `x // 10` means $\lfloor x/10 \rfloor$. This notation is borrowed from `Python` (Rossum 1995).

Some operations used in this paper are not described in the tutorial, either because they were added to `isl` more recently or because they are of a more heuristic nature and are only useful in specific circumstances. The new operation is that of *binding* a tuple to a sequence of symbolic constants, which fixes the tuple dimensions to the corresponding symbolic constants. The tuple itself is removed in this operation. For example, binding the range tuple of a binary relation results in a set corresponding to the domain of the binary relation. Binding the range or the relation in (2) to the (single element) sequence [Y] yields the set

$$\{\, [x] : 0 \leq \mathrm{Y} < x \leq 10 \,\}, \tag{4}$$

which is equal to the interval $\{\, [\mathrm{Y} + 1{:}10] \,\}$ if the symbolic constant Y has a value between 0 and 9 and is equal to the empty set if Y has some other value. This operation only changes the interpretation of the input object and does not require any computations.

Variable compression (Meister 2004) exploits the equality constraints in the description of a set to obtain a set with the same number of points, but of a lower dimensionality. In particular, the points in the original set can be obtained from the "compressed" set by applying an affine function that is constructed during the computation of the compression. The original description is formulated in terms of polyhedra. In `isl`, the input set may be described by a generic Presburger formula, which is converted internally into a disjunction of conjunctions. The equality constraints used for compression are those shared by all disjuncts. If these equality constraints involve any existentially quantified variables, then these are treated as set variables for the purpose of compression in the `isl` implementation. The dimensionality of the compressed set may therefore in theory be greater than that of the input set.

The fixed-size *box hull* operation examines the constraints of a binary relation to find an overapproximation where the range can be described as a rectangular box with fixed size and an offset that depends on the domain variables and the symbolic constants. Moreover, the offset is restricted to being purely affine. This operation was described by Verdoolaege, Juega, et al. (2013, Section 7.3) for obtaining a suitable mapping to shared memory in `PPCG` (Verdoolaege, Juega, et al. 2013) and is also used by `Tensor Comprehensions` (Vasilache et al. 2019). In these applications, the memory size is required to be fixed (and in particular not depend on outer loops or block identifiers) and the offset should be purely affine to avoid the introduction of piecewise of quasi-affine expressions in the accesses to shared memory. Since this operation is based on the constraints of a particular representation of a set, it is not guaranteed to produce the same results on different representations of the same set. The operation may also fail to produce any result, either because the range cannot be approximated by a fixed-sized box or because no suitable fixed-sized box with purely affine offset could be found.

3 Target Architecture

The target architecture is an MPPA (Massively Parallel Processor Array), consisting of a 2-dimensional grid of PEs that communicate with their nearest neighbors in the four cardinal directions. At the time of writing, only few details have been made public (Lie 2019). Most relevant for the present paper is that memory is distributed over the PEs and that the hardware performs dataflow scheduling. In particular, the data that is communicated between PEs can be either a floating point number or a pair of a floating point number (of smaller size) and an integer value. In this second case, the integer value is typically an indication of the position of the communicated piece of data inside a tensor, called its *index*. Positions that do not appear in any such pair are assumed to have a zero value, meaning that zero values do not need to be communicated. Tensors communicated in this way are called *sparse*. There is also a mechanism for marking the end of a sequence of value-position pairs, but the details are not important here.

On start-up, each PE executes its `main` function, which typically performs some configuration before the actual computation is initiated. The computation itself is performed by *tasks* that can be invoked by other tasks (including `main`) or by the hardware in response to some event such as the arrival of tensor data.

A feature of the architecture that is especially relevant to the present paper is that it supports powerful SIMD instructions that mimic a loop nest of depth at most four. In particular, each invocation of such an instruction performs a sequence of operations defined by a rectangular set of instances of dimension at most four. The rectangular set is described by a sequence of fixed loop sizes. That is, each instance is represented by a sequence of integers between 0 (inclusive) and the corresponding size (exclusive). Furthermore, each access performed by the instruction needs to be equal to some offset plus a linear function of this sequence of integers. In particular, apart from the constant term, they are not allowed to be quasi-affine or piecewise. A SIMD instruction can not only perform a certain number of operations per cycle, but it also removes a lot of overhead compared to an explicit loop since the software management of the loop is replaced by a hardware mechanism. Note that the exact number of operations that can be performed in one cycle has not been made public yet.

4 Code Generation Overview

The inputs to `dtg_codegen` are

1. an algorithm description in a high-level Cerebras specific intermediate representation called LAIR (Linear Algebra Intermediate Representation);
2. the size of a rectangular block of PEs that should perform this computation;
3. information about how tensors arrive and should leave this block.

The output is C-like code that should be run on each PE.

4.1 Input

The input to `dtg_codegen` consists of two major pieces, a description of the algorithm in LAIR and information about how this algorithm should be mapped to a block or PEs.

```
lair ff<T=float16>(M, N):
    T W[M][N], T x[N] -> T y[M]
{
    all (i, j) in (M, N)
        y[i] += W[i][j] * x[j]
}
```

Listing 1. Excerpt from Fully Connected LAIR input

4.1.1 LAIR

LAIR is a DSL (Domain Specific Language) for describing ML (Machine Learning) layers. It can be written by hand or it can be extracted from a description in an ML framework such as `TensorFlow` (Abadi et al. 2016). Listing 1 shows an excerpt from a LAIR description for a fully connected layer. In particular, it shows a node called `ff` containing the matrix-vector multiplication for the forward pass.

Each `lair` node describes how a set of input tensors is transformed to one or more output tensors. Some interesting features of this description are:

- each statement has a rectangular set of instances, with lower bounds zero and (exclusive) upper bounds specified in the `in`-clause of the `all`-construct.
- LAIR is a single assignment language in the sense that a tensor is defined by a single statement. In a reduction, a given tensor element may be defined by multiple instances of this statement. However, such a reduction statement always initializes the reduction and never updates the tensor with respect to some previous value. Since all reductions are initializing, no special notation is needed for marking initializing reductions, such as the '`!`' of `Tensor Comprehensions`.
- All accesses are (strictly and totally) affine. In particular, quasi-affine expressions or piecewise affine expressions are not allowed.
- Each tensor in a statement is accessed through a single affine index expression. In particular, if the same tensor appears multiple times in the same statement, then it is accessed in exactly the same way in all appearances.

This restricted representation should be adequate for most deep learning applications.

Other nodes in the LAIR input combine and/or specialize `lair` nodes to form a complete ML layer description. In particular, while the `lair` nodes may have parametric tensor sizes such as `M` and `N` in Listing 1, the input to `dtg_codegen` is always fully specialized to fixed integer tensor sizes. The rest of this paper will take $M = 32$ and $N = 16$.

4.1.2 LAIR mapping

The LAIR mapping provides information to `dtg_codegen` about how it should map the LAIR input to a rectangle of PEs. This information is specified in `isl` syntax.

```
{
  ff[i,j] -> PE[j//4, 1 + i//8];
  fg[i,j] -> PE[j//4, 1 + i//8];
  fd[j,i] -> PE[j//4, 1 + i//8];
  update[i,j] -> PE[j//4, 1 + i//8]
}
```

Listing 2. Placement for Fully Connected LAIR input

The first bit of information is the size of the target rectangle of PEs, e.g., `{ PE[4, 5] }` representing the set of PEs $\{ [0{:}3, 0{:}4] \}$. The second bit describes the communication of the input and output tensors. In particular, for each input tensor element, it specifies from which external PE(s) the element arrives at the target rectangle and in what order compared to the other elements that arrive from the same external PE. Note that these external PEs need to be adjacent to the target rectangle. For example, the `x` input of Listing 1 may have the following specification:

```
{ x[i = 0:15] -> [PE[0, -1] -> index[i]] }
```

This means that the entire `x` tensor arrives at PE $(0, 0)$ from the north (i.e., $(0, -1)$) and that the index value that comes along with the data (assuming it is sparse) is equal to the position in the vector. If the `index` space is multi-dimensional, then it is only the innermost dimension that specifies the index value, while the outer dimensions count the number of value-position sequences. These are called the *outer index values.* If the tensor is not sparse, then the `index` space specifies the (lexicographic) order in which tensor elements arrive. For example, assuming the `y` output is not sparse, the specification

```
{ y[i = 0:31] -> [PE[4, 1 + i//8] ->
                    index[i mod 8]] }
```

means that the PEs $(4, 1)$ to $(4, 4)$ are each expecting a chunk of `y` of size 8 from the target rectangle and that within each chunk the elements are sent in order of increasing vector position. Note that while any `isl` supported function could be specified for these tensor mappings, in practice they are limited to a combination of interchange, reversal and chunking.

Finally, the LAIR mapping specifies which PEs should perform which LAIR compute node instances. For example, Listing 2 shows a complete placement for a fully connected LAIR input, including the `ff`-node of Listing 1. The LAIR code for the other nodes (the gradient `fg`, the backpropagation `fd` and the weight update `update`) is not shown here. Note that the computations are mapped to the set of PEs $\{ [0{:}3, 1{:}4] \}$, which is a strict subset of the target rectangle. As explained in Section 4.2 below, the top row of PEs will be used to spread data to the computation PEs. It is also possible to only specify the placement direction (here, (j, i)) and to let `dtg_codegen` determine the appropriate tiling and offset onto the rectangle of PEs. In the future, some suitable mapping may be (optionally) computed automatically.

The LAIR mapping also contains additional information about how tensors are communicated (such as whether they are sparse) and about delays between different parts of the LAIR graph, but this is beyond the scope of this paper.

4.2 Task Graph Construction

During task graph construction, the computation performed by the input LAIR graph is broken up into communication and computation tasks. The communication tasks send data available on some PEs to other PEs, e.g., spreading input tensors to all PEs that need them or collecting the results of a reduction. Computation tasks perform local computations. A distinction is made between tasks that react to the arrival of an element from a sparse tensor and those that read in an entire tensor or that operate on local memory. A special class of computation tasks that operate on local memory is formed by the initialization tasks. These initialize local memory to some fixed value, typically zero when preparing for storing sparse data or the neutral element when preparing for a reduction.

During this construction process, various assumptions are made on the internal and external communication. For example, within the rectangle of PEs performing the computation, all data is assumed to flow either horizontally or vertically. If these conditions are not met, then the set of PEs that perform the core computations is reduced such that the peeled-off border PEs can make the required adjustments. For example, according to the specification shown in Section 4.1.2, all `x` elements arrive on a single PE. These elements are spread over the columns that need them by adapters introduced in the top row of the original target rectangle of PEs. The target rectangle and the LAIR mapping for the computation PEs are adjusted accordingly. In particular, the target rectangle of computation PEs is reduced to size 4×4 and from the perspective of these computation PEs, the `x` tensor now arrives as

$$\{ \mathrm{x}[i = 0{:}15] \to [\mathrm{PE}[\lfloor i/4 \rfloor, -1] \to \mathrm{index}[i \bmod 4]] \}. \quad (5)$$

Similarly, `y` is now sent out as

$$\{ \mathrm{y}[i = 0{:}31] \to [\mathrm{PE}[4, \lfloor i/8 \rfloor] \to \mathrm{index}[i \bmod 8]] \} \quad (6)$$

and also the placement of Listing 2 is shifted up by one because the `PE`-space is shifted down by one.

4.3 Local Memory Allocation

Some input tensors are processed as they arrive. Other input tensors need to be stored in memory, e.g., because they are used in multiple operations or because an operation takes multiple input tensors. Similarly, for output tensors that are the result of a reduction, the partial results computed by each PE may need to be stored in memory. Any internal tensor (i.e., not an input or output tensor) also needs to be stored in memory.

The allocation of local memory for these tensors constitutes a first application of the box hull operation. In this case, the fixed size that this operation produces is not critical since the size of a local array could in theory depend on the PE coordinates. The purely affine offset is also not strictly required. Even though the mapping from global tensors to local tensors needs to be taken into account in the generation of the accesses of SIMD instructions that access local tensors, these accesses may have an arbitrary constant term. While the box hull operation is not guaranteed to produce a result, given the structured nature of the input it always does here in practice.

The detailed computation is illustrated on the output tensor y of the code in Listing 1. Combining the access relation $\{ \text{ff}[i = 0{:}31, 0{:}15] \to \text{y}[i] \}$ and the placement of Listing 2, adjusted according to Section 4.2, yields the set of tensor elements accessed per PE:

$$\{ \text{PE}[0{:}3, i_1 = 0{:}3] \to \text{y}[i = 0{:}31] : 8i_1 \leq i \leq 7 + 8i_1 \}. \quad (7)$$

The constraints $8i_1 \leq i \leq 7 + 8i_1$ can be used to obtain a suitable box of size $\{ \text{y}[8] \}$ at offset $\{ \text{PE}[i_0, i_1] \to \text{y}[8i_1] \}$. The (PE-dependent) mapping from global tensor elements to local memory is obtained by subtracting this offset from an identity mapping on y, resulting in

$$\{ [\text{PE}[i_0, i_1] \to \text{y}[i]] \to \text{y\_local}[-8i_1 + i] \}, \quad (8)$$

or

$$\{ \text{y}[i] \to \text{y\_local}[-8\text{PE}_\text{Y} + i] \} \quad (9)$$

after binding to symbolic constants representing the PE coordinates.

The initial local memory allocation described here may get adjusted later to accommodate extra computation instances as described in Section 5.3 below.

5 SIMD Code Generation

In some cases, the generation of SIMD instructions is fairly trivial. For example, initialization tasks initialize an entire (rectangular) local tensor (or a rectangular subset if the local tensor gets extended as described in Section 5.3 below). In this case, the rectangular domain of the generated SIMD instruction corresponds to the local tensor and the linear access is simply an identity function.

In other cases, the generation is more involved. In particular, this section will focus on the generation of SIMD instructions for tasks that react to the arrival of an element from a sparse tensor and that therefore only need to perform the operations that correspond to the given index value(s). A SIMD instruction is only generated for such a task if the following conditions are met.

1. The computation of the task consists of a single operation for which SIMD support is available.
2. The set of instances is a (dense) rectangular domain with a fixed size.
3. The dimensionality of this domain is greater than zero.
4. All accesses are linear (ignoring the constant term).

Item 3 is imposed because otherwise it is simply not useful to generate a SIMD instruction. The other conditions are required by the target architecture as explained in Section 3. The fixed-size nature of the domain in Item 2 needs some clarification. The size only needs to be fixed at the point in the code where the SIMD instruction is configured. This means the size can depend on the PE coordinates since those are fixed on each PE. In theory, the size could also depend on the index of the arriving sparse tensor. However, this would mean that the SIMD instruction needs to be configured before every call. Since this configuration can itself be fairly expensive, this is not currently considered. In fact, the configuration is currently performed once at the start of the execution, meaning that the size is also not allowed to depend on the outer index values.

5.1 Base Case

The analysis starts by checking if the performed computation is one for which a SIMD instruction is available. In the base case, no checks need to be performed on the accesses since they come directly from the LAIR specification, which requires accesses to be purely affine.

The instance set of the task is computed by first binding the range of the placement (intersected with the instance set of the computation) to symbolic constants representing the PE coordinates and then intersecting the result with the constraints on the symbolic constants representing the index values. These constraints are obtained by binding the range of the order of the incoming tensor to the index symbolic constants (relating these symbolic constants to the tensor indices) and applying the inverse access relation.

Example 5.1. The task corresponding to the matrix-vector multiplication of Listing 1 performs an add-multiple, for which a SIMD instruction is available. The original placement of Listing 2 was adjusted to

$$\{ \text{ff}[i, j] \to \text{PE}[\lfloor j/4 \rfloor, \lfloor i/8 \rfloor] \} \quad (10)$$

in Section 4.2. Binding the range of this relation to the PE coordinates and taking into account the instance set results in

$$\{ \text{ff}[i = 0{:}31, j = 0{:}15] : 8\text{PE}_\text{Y} \leq i \leq 7 + 8\text{PE}_\text{Y} \land 4\text{PE}_\text{X} \leq j \leq 3 + 4\text{PE}_\text{X} \}. \quad (11)$$

Assuming the task is activated on the arrival of an element of a sparse x, its order is $\{ \text{x}[i = 0{:}15] \to \text{index}[i] \}$ as specified in Section 4.1.2. Binding the range of this relation to the (here) single index symbolic constant, results in $\{ \text{x}[\text{index}] : 0 \leq \text{index} \leq 15 \}$. Applying the inverse of the access relation $\{ \text{ff}[i_0, i_1] \to \text{x}[i_1] \}$ yields $\{ \text{ff}[i_0, \text{index}] : 0 \leq \text{index} \leq 15 \}$.

```
lair C() : float16 x[16],
  float16 W[2][3] -> float16 y[2][14]
{
  all (k, w, rw) in (2, 16 - 3 + 1, 3)
    y[k][w] += x[w + rw] * W[k][rw]
}
```

Listing 3. Excerpt from 1D convolution LAIR input

Intersecting the domain (11) with these constraints gives

$$\begin{aligned}\{\, \mathrm{ff}[i = 0{:}31, j = \mathrm{index}] : 0 \le \mathrm{index} \le 15 \wedge{} & \\ \mathrm{PE_X} = \lfloor(\mathrm{index})/4\rfloor \wedge{} & \\ 8\mathrm{PE_Y} \le i \le 7 + 8\mathrm{PE_Y} \,\}. & \end{aligned} \tag{12}$$

The resulting instance set is then examined to see if it forms a dense rectangular domain of fixed size (on each PE). This can be determined by independently computing the minimal and maximal values of the set dimensions using PILP (Parametric Integer Linear Programming) (Feautrier 1988), constructing a set with these bounds and then checking if this set is a subset of the original set. The difference between maximal and minimal values (plus 1) yields the size and this size needs to be fixed. As explained above, this means that the size should not depend on any symbolic constants representing index values.

Example 5.2. Continuing from Example 5.1, the minimal and maximal values of the set (12) are

$$\{\, \mathrm{ff}[8\mathrm{PE_Y}, \mathrm{index}] : \mathrm{K} \,\} \tag{13}$$

and

$$\{\, \mathrm{ff}[7 + 8\mathrm{PE_Y}, \mathrm{index}] : \mathrm{K} \,\}, \tag{14}$$

with K the known constraints on the symbolic constants:

$$\mathrm{K} : 0 \le \mathrm{index} \le 15 \wedge \mathrm{PE_X} = \lfloor \mathrm{index}/4 \rfloor \wedge 0 \le \mathrm{PE_Y} \le 3. \tag{15}$$

Building a set with these bounds yields exactly the set (12), meaning that it forms a dense rectangular set. The resulting size is

$$\{\, \mathrm{ff}[8, 1] : \mathrm{K} \,\}, \tag{16}$$

which simplifies to $\{\, \mathrm{ff}[8, 1] \,\}$ when simplifying with respect to the known constraints (15). This size is independent of the symbolic constants representing index values and the computation can therefore be performed by a SIMD instruction.

5.2 Compression

In some cases, the instances that need to be performed at the arrival of a tensor element do not form a (dense) rectangular set in the original space of node instances, but they can still be transformed to form a set of this shape in some other space. A prime class of examples is formed by convolution style algorithms.

Example 5.3. Consider the excerpt for a small 1D convolution in Listing 3, with placement $\{\, \mathrm{C}[k, w, rw] \to \mathrm{PE}[0, k] \,\}$ and sparse tensor order $\{\, \mathrm{x}[w = 0{:}15] \to \mathrm{index}[w] \,\}$. Computing the set of instances of the single computation that need to be performed on each invocation of the task as in Example 5.1 yields

$$\begin{aligned}&\{\, \mathrm{C}[k = \mathrm{PE_Y}, w = 0{:}13, rw = \mathrm{index} - w] : \\ &\quad \mathrm{PE_X} = 0 \wedge 0 \le \mathrm{index} \le 15 \wedge 0 \le \mathrm{PE_Y} \le 1 \wedge \\ &\quad -2 + \mathrm{index} \le w \le \mathrm{index} \,\}.\end{aligned} \tag{17}$$

Note in particular that the symbolic constant index is equal to the *sum* of two tuple dimensions. This means that the instances that need to be run on an invocation lie on a diagonal inside the full set of instances. Computing minimal and maximal values in each direction results in a rectangular box containing this diagonal and this box is clearly not equal to the diagonal itself.

The main property that prevents these sets of instances from being recognized as a (dense) rectangular box is that they live in a higher-dimensional space. Variable compression can be used to reduce the dimensionality of the ambient space, thereby increasing the chance of obtaining a rectangular box. Note that there is no guarantee that the result will be a rectangular box, even if the set of instances could still be transformed into a rectangular box using a further affine transformation. In particular, a unimodular transformation may need to be applied, e.g., the one that is used for factoring polyhedra (Verdoolaege, Seghir, et al. 2007, Section 7). However, such a unimodular transformation is not currently considered by `dtg_codegen`. The variable compression itself is applied indiscriminately. This means it is also applied when the original instance set can already be identified as a rectangular box.

Example 5.4. Consider first the instance set (12) from Example 5.1, which was already determined to be a rectangular box in Example 5.2. Applying variable compression to this set produces the compression

$$\{\, [i_0] \to \mathrm{ff}[i_0, \mathrm{index}] \,\} \tag{18}$$

and the compressed instance set

$$\begin{aligned}\{\, [i_0 = 0{:}31] : 0 \le \mathrm{index} \le 15 \wedge \mathrm{PE_X} = \lfloor(\mathrm{index})/4\rfloor \wedge \\ 8\mathrm{PE_Y} \le i_0 \le 7 + 8\mathrm{PE_Y} \,\}.\end{aligned} \tag{19}$$

Since this is a one-dimensional set, it is clearly still a rectangular set. The minimal and maximal values of this compressed instance set are

$$\{\, [8\mathrm{PE_Y}] : \mathrm{K} \,\} \quad \text{and} \quad \{\, [7 + 8\mathrm{PE_Y}] : \mathrm{K} \,\}, \tag{20}$$

with K the known constraints (15). The corresponding size is

$$\{\, [8] : \mathrm{K} \,\}. \tag{21}$$

```
lair C() : float16 x[16][16],
  float16 W[2][3][3] -> float16 y[2][14][14]
{
  all (k, w, h, rw, rh) in (2, 14, 14, 3, 3)
    y[k][w][h] +=
      x[w + rw][h + rh] * W[k][rw][rh]
}
```

Listing 4. Excerpt from 2D convolution LAIR input

Note that the variable compression changes the representation of the set of computation instances, requiring a modification to the access functions. However, since both the original access functions and the compression are purely affine, so is their composition and the modified access function are therefore also purely affine.

Example 5.5. Continuing from Example 5.3, applying variable compression to the set (17) yields the compression

$$\{ [i_0] \to \mathrm{C}[\mathrm{PE_Y}, \mathrm{index} - i_0, i_0] \} \tag{22}$$

and the compressed instance set

$$\{ [i_0 = 0{:}2] : \mathrm{PE_X} = 0 \wedge 0 \le \mathrm{index} \le 15 \wedge \\ 0 \le \mathrm{PE_Y} \le 1 \wedge -13 + \mathrm{index} \le i_0 \le \mathrm{index} \}. \tag{23}$$

Since this is a one-dimensional set, it is clearly a rectangular set. Since no existentially quantified variables are involved in the description, the set is also dense. However, the size of the interval is *not* constant. In particular, simplified with respect to the known constraints on the symbolic constants, the size is of the form

$$\{ [16 - \mathrm{index}] : \mathrm{index} \ge 13 \} \cup \\ \{ [3] : 2 \le \mathrm{index} \le 12 \} \cup \\ \{ [1 + \mathrm{index}] : \mathrm{index} \le 1 \}. \tag{24}$$

That is, the size starts out at 1 for index equal to 0, increases to 3, remains at 3 for a number of index values and then drops back down to 1. This non-constant size issue will be addressed in Section 5.3 and illustrated in Example 5.9.

The original access functions for accessing the other tensors are

$$\{ \mathrm{C}[i_0, i_1, i_2] \to \mathrm{W}[i_0, i_2] \} \quad \text{and} \quad \{ \mathrm{C}[i_0, i_1, i_2] \to \mathrm{y}[i_0, i_1] \}. \tag{25}$$

Combined with the compression (22), these become

$$\{ [i_0] \to \mathrm{W}[\mathrm{PE_Y}, i_0] \} \quad \text{and} \quad \{ [i_0] \to \mathrm{y}[\mathrm{PE_Y}, \mathrm{index} - i_0] \}. \tag{26}$$

Example 5.6. Similar results can be obtained for the 2D convolution in Listing 4, with placement $\{ \mathrm{C}[k, w, h, rw, rh] \to [0, k] \}$ and sparse tensor order $\{ \mathrm{x}[w = 0{:}15, h = 0{:}15] \to \mathrm{index}[w, h] \}$. The original instance set is

$$\{ \mathrm{C}[k = \mathrm{PE_Y}, w = 0{:}13, h = 0{:}13, \\ rw = \mathrm{index}_0 - w, rh = \mathrm{index} - h] : \\ \mathrm{PE_X} = 0 \wedge 0 \le \mathrm{index}_0 \le 15 \wedge 0 \le \mathrm{index} \le 15 \wedge \\ 0 \le \mathrm{PE_Y} \le 1 \wedge -2 + \mathrm{index}_0 \le w \le \mathrm{index}_0 \wedge \\ -2 + \mathrm{index} \le h \le \mathrm{index} \}. \tag{27}$$

The compression is

$$\{ [i_0, i_1] \to \mathrm{C}[\mathrm{PE_Y}, \mathrm{index}_0 - i_0, \mathrm{index} - i_1, i_0, i_1] \} \tag{28}$$

and the compressed instance set is

$$\{ [i_0 = 0{:}2, i_1 = 0{:}2] : \\ \mathrm{PE_X} = 0 \wedge 0 \le \mathrm{index}_0 \le 15 \wedge 0 \le \mathrm{index} \le 15 \wedge \\ 0 \le \mathrm{PE_Y} \le 1 \wedge -13 + \mathrm{index}_0 \le i_0 \le \mathrm{index}_0 \wedge \\ -13 + \mathrm{index} \le i_1 \le \mathrm{index} \}. \tag{29}$$

This is again a rectangular box of non-constant size.

```
lair C() : float16 x[8][8][4],
  float16 filter[4][4] -> float16 y[8][8][4]
{
  all (k, h, w, c) in (4, 8, 8, 4)
    y[h][w][k] += x[h][w][c] * filter[c][k]
}
```

Listing 5. Excerpt from degenerate 2D convolution LAIR input

Example 5.7. To illustrate that variable compression, or at least the `isl` implementation, can in some case also be used to turn a strided domain into a dense domain, consider the (degenerate) 2D convolution in Listing 5, with placement

$$\{ \mathrm{C}[k, h, w, c] \to \mathrm{PE}[\lfloor k/2 \rfloor + 2\lfloor w/4 \rfloor, 2\lfloor h/4 \rfloor + \lfloor c/2 \rfloor] \} \tag{30}$$

and sparse tensor order

$$\{ \mathrm{x}[h = 0{:}7, w = 0{:}7, c = 0{:}3] \to \mathrm{index}[h \bmod 4, w, c \bmod 2] \}. \tag{31}$$

The original instance set is

$$\{ \mathrm{C}[k = 0{:}3, h = 0{:}7, w = \mathrm{index}_1, \\ c = \mathrm{index}_0 + \mathrm{index} + 2\mathrm{PE_Y} - h] : \\ (\mathrm{index}_0 - h) \bmod 4 = 0 \wedge 0 \le \mathrm{index}_0 \le 3 \wedge \\ 0 \le \mathrm{index}_1 \le 7 \wedge 0 \le \mathrm{index} \le 1 \wedge \\ h \le \mathrm{index}_0 + \mathrm{index} + 2\mathrm{PE_Y} \le h + 3 \wedge \\ 2\mathrm{PE_X} - k \le 4\lfloor(\mathrm{index}_1)/4\rfloor \le 1 + 2\mathrm{PE_X} - k \}. \tag{32}$$

The compression is

$$\{ [i_0, i_1] \to \mathrm{C}[i_0, \mathrm{index}_0 - 4i_1, \mathrm{index}_1, \mathrm{index} + 2\mathrm{PE_Y} + 4i_1] \} \tag{33}$$

and the compressed instance set is

$$\begin{aligned}&\{\,[i_0 = 0{:}3, i_1 = \lfloor(3 - \mathrm{index} - 2\mathrm{PE_Y})/4\rfloor] : \\ &\quad 0 \le \mathrm{index}_0 \le 3 \wedge 0 \le \mathrm{index}_1 \le 7 \wedge \\ &\quad 0 \le \mathrm{index} \le 1 \wedge -7 + \mathrm{index}_0 \le 4i_1 \le \mathrm{index}_0 \wedge \\ &\quad 2\mathrm{PE_X} - i_0 \le 4\lfloor(\mathrm{index}_1)/4\rfloor \le 1 + 2\mathrm{PE_X} - i_0\,\}.\end{aligned} \tag{34}$$

This too is a rectangular box of non-constant size. Note that the second tuple dimension has a fixed (symbolic) value, meaning that variable compression failed to exploit all equality constraints among the variables, mainly because the remaining equality constraint is not explicitly available in the description of the input set. Still, the compressed set has the form of a rectangular box, be it of size 1 in one dimension.

5.3 Approximation

When performing a convolution, the number of operations that need to be performed at the border is typically less than in the center. This causes a variation in the size of the instance set, which would require the SIMD configuration to be performed multiple times, rather than just once at the start of the entire execution. It is, however, possible in some cases to still perform a single SIMD configuration by letting the SIMD instruction perform some *extra* operations for those invocation where it would have to do fewer operations. The results of these extra operations are then simply discarded. Since these extra operations are only performed for invocations at the border, the extra cost is small compared to having to perform a reconfiguration.

The extra instances are obtained by computing a fixed-size box hull of the (compressed) set of instances. Obviously, these extra instances should not be allowed to interfere with the proper instances. In particular, this means that the tensor elements written by the extra instances should be disjoint from those written by the proper instances. That is, for any PE, the sets should be disjoint over all invocations of the task. This means that the symbolic constants representing index values should be projected out before checking for disjointness. The disjoint accesses typically correspond to out-of-bounds tensor accesses. The mapping of the tensor to local memory therefore needs to be adjusted accordingly. The current implementation does not allow the extra instances to perform any out-of-bounds *reads*, although this could easily be relaxed since it does not matter which values are used in a computation of which the results will be discarded.

Note that `dtg_codegen` computes a box hull approximation first and only if this fails to produce a suitable rectangular domain does it try to compute the actual minima and maxima as described in Section 5.1. This means that this hull may be used even in cases where the actual (compressed) instance set already is a rectangular box. The set of extra instances will simply be empty in such cases and there is then therefore no way that they could interfere with the proper instances.

Example 5.8. For the compressed instance set (19) from Example 5.4, a fixed-size box hull is computed with offset $\{\,[8\mathrm{PE_Y}]\,\}$ and size $\{\,[8]\,\}$, which in this case is the same as what would be obtained from computing the minimal and maximal values. There are then also no extra instances involved.

Example 5.9. The compressed instance set (23) from Example 5.5 has a non-constant size (24), but it can be approximated by a fixed-size box hull with offset $\{\,[0]\,\}$ and size $\{\,[3]\,\}$. That is, the approximated (compressed) instance set is

$$\{\,[0{:}2] : \mathrm{PE_X} = 0 \wedge 0 \le \mathrm{PE_Y} \le 1 \wedge 0 \le \mathrm{index} \le 15\,\}. \tag{35}$$

The extra instances are therefore

$$\{\,[1 + \mathrm{index}{:}2]\,\} \cup \{\,[0{:}{-14} + \mathrm{index}]\,\}, \tag{36}$$

when simplified with respect to the known constraints on the symbolic constants.

The read access relation (26) in terms of the compressed instance set becomes

$$\{\,[i_0] \to \mathrm{W\_local}[0, i_0]\,\} \tag{37}$$

when composed with the mapping to local memory. The local memory elements read by the extra instances are therefore (simplified)

$$\{\,\mathrm{W\_local}[0, 1 + \mathrm{index}{:}2]\,\} \cup \{\,\mathrm{W\_local}[0, 0{:}{-14} + \mathrm{index}]\,\}, \tag{38}$$

which falls within the bounds of the allocated memory.

The write access relation (26) in terms of the compressed instance set becomes

$$\{\,[i_0] \to \mathrm{y\_local}[0, \mathrm{index} - i_0]\,\} \tag{39}$$

when composed with the mapping to local memory. The local memory elements written by the proper instances (over all index values) are therefore

$$\{\,\mathrm{y\_local}[0, 0{:}13] : \mathrm{PE_X} = 0 \wedge 0 \le \mathrm{PE_Y} \le 1\,\}, \tag{40}$$

while those written by the extra instances are

$$\begin{aligned}&\{\,\mathrm{y\_local}[0, 14{:}15] : \mathrm{PE_X} = 0 \wedge 0 \le \mathrm{PE_Y} \le 1\,\} \cup \\ &\{\,\mathrm{y\_local}[0, -2{:}{-1}] : \mathrm{PE_X} = 0 \wedge 0 \le \mathrm{PE_Y} \le 1\,\}.\end{aligned} \tag{41}$$

These sets of memory elements are clearly disjoint, so the extra instances do not interfere with the proper instances. The mapping to local memory does need to be adjusted, though. In particular, the allocation size is extended to become $\{\,\mathrm{y\_local}[1, 18]\,\}$ and the accesses need to be shifted over an offset $\{\,\mathrm{y\_local}[0, -2]\,\}$.

5.4 Size Enumeration

An alternative to extending the set of instances to a fixed-size box hull in case it is not of fixed size already is to set up different SIMD configurations, one for each possible size. This allows SIMD instructions to be used when the extra instances introduced by the box hull conflict with the other instances, with only a limited overhead. In particular, the SIMD configurations are performed at the start of the execution and the right configuration is selected before each invocation of the SIMD instruction. The down-side is that only a limited number of SIMD configurations can be active at any given time, meaning that this approach can only be taken if the total number of possible sizes is small.

Note that, in theory, the sizes may not only depend on the symbolic constants corresponding to the index values, but also on those corresponding to the PE coordinates. In order for the set of sizes to be enumerated, all symbolic constants need to be projected out. However, each PE should ideally only perform SIMD configurations for sizes that it needs. The set of sizes is therefore first approximated by a fixed-size box hull and the offset is subtracted from the elements in the actual set of sizes.

Example 5.10. Consider once more the 1D convolution from Example 5.5. While this input can be handled by the fixed-size box hull of the computation instances without conflicts, it is instructive for illustrating the difference. Projecting out the symbolic constants representing the index values from the set of sizes (24) yields

$$\{\,[1{:}3] : \mathrm{PE_X} = 0 \wedge 0 \le \mathrm{PE_Y} \le 1\,\}. \tag{42}$$

The fixed-size box hull has offset $\{\,[1]\,\}$ and size $\{\,[3]\,\}$. Note that the offset does not depend on the PE coordinates here, but it could in theory. Subtracting the offset and projecting out the symbolic constants corresponding to the PE coordinates yields the set

$$\{\,[0{:}2]\,\}. \tag{43}$$

The elements in this set are used as identifiers for the different SIMD configurations. Adding back the offset that was subtracted before produces the size that corresponds to an identifier.

6 Experimental Evaluation

Some preliminary experimental results show the effectiveness of using SIMD instructions. Unfortunately, the actual speed-ups obtained cannot be included in this paper as that would reveal the number of operations that can be performed in one cycle. The number of cycles performed with and without the use of SIMD instructions are obtained from a simulator of the hardware. In order to obtain a fixed number of cycles for each invocation of the network layer, the number of zero entries was set to zero in these experiments. The tensor sizes and the number of PEs used in the experiments are also kept very small to reduce the running time of the simulator. This means that the advantage of SIMD code over explicit loops is somewhat exaggerated as there is little steady state in the loops and therefore comparably a lot of overhead.

For some test cases no SIMD code can or needs to be generated. These show no difference when SIMD code generation is enabled. Some test cases show a speed-up below the number of operations that can be executed in a single cycle by the SIMD engine. This can be explained by the fact that the number of iterations performed by the SIMD instruction is relatively low, meaning that the SIMD instructions do not completely dominate the number of cycles. Many of the test cases with lower speed-up use the size enumeration of Section 5.4, which has some overhead in selecting the SIMD configuration. Some test cases show a speed-up that is much larger than the SIMD width. This can be explained by the fact that the overhead of the loops simulated by the SIMD instruction is completely removed. This overhead includes not only the updates of the loop iterators and the address calculations, but also the computation of the bounds on inner loops.

7 Related Work

A LAIR specification is essentially a restricted form of a SARE (System of Affine Recurrence Equations) augmented with reduction operators (Lavenier et al. 1999) with an implicit time. There is a plethora of research on mapping such systems to various forms of parallel hardware. In `dtg_codegen`, this mapping is currently taken as input and the main purpose is the generation of PE code for performing the required communications and computations. Note that the way data is communicated shares some similarity with the treatment of communication on SAREs, but this is not the focus of the present paper.

Feld et al. (2013) focus on finding good tile sizes taking into account SIMD units and cache sizes. Kong et al. (2013) focus on constructing schedules that favor stride-0 and stride-1 accesses and then mapping the resulting vectorizable "codelets" to efficient target specific SIMD code. Henretty et al. (2013) focus on stencil computations and combine a data layout transformation interleaving vector segments with split tiling. Sharma et al. (2015) perform more general array interleaving for SIMD architectures. Hallou et al. (2017) operate on binary code and consider the conversion of SSE to AVX SIMD instructions as well as runtime vectorization using polyhedral compilation techniques.

The instance set compression of Section 5.2 is similar to the array compaction of Schreiber and Cronquist (2004). Both this array compaction and the variable compression of Meister (2004) used in Section 5.2 rely on the Hermite Normal Form.

Alias and Barthou (2005) decide where to call performance libraries by matching code against library templates. Iooss (2016) extends this to templates where inputs can occur multiple times and also considers semantic properties such as distributivity, associativity and commutativity. Lu et al. (2012) break up tensor contraction expressions into components that are mapped to matrix multiplications with efficient library implementations. As part of mapping their DSL to

CUDA, Vasilache et al. (2019, Section 3.7) also detect reductions that can be handled by CUB.

8 Conclusion

This paper has shown that an effective combination of two relatively well-known polyhedral operations, variable compression and fixed-size box hull, can be used to expose rectangular sets of operations that can be mapped to the advanced SIMD operations of the Cerebras CS-1 architecture.

References

Abadi, Martín, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng (Nov. 2016). “TensorFlow: A System for Large-Scale Machine Learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, pp. 265–283.

Alias, Christophe and Denis Barthou (2005). “Deciding Where to Call Performance Libraries”. In: *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings*. Ed. by José C. Cunha and Pedro D. Medeiros. Vol. 3648. Lecture Notes in Computer Science. Springer, pp. 336–345.

Baghdadi, Riyadh, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe (2019). “Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code”. In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO 2019. Washington, DC, USA: IEEE Press, pp. 193–205. DOI: 10.1109/CGO.2019.8661197.

Feautrier, Paul (1988). “Parametric Integer Programming”. In: *RAIRO Recherche Opérationnelle* 22.3, pp. 243–268.

Feld, Dustin, Thomas Soddemann, Michael Jünger, and Sven Mallach (Jan. 2013). “Facilitate SIMD-Code-Generation in the polyhedral model by hardware-aware automatic code-transformation”. In: *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*, pp. 45–54.

Hallou, Nabil, Erven Rohou, and Philippe Clauss (Dec. 2017). “Runtime Vectorization Transformations of Binary Code”. In: *International Journal of Parallel Programming* 45.6, pp. 1536–1565. DOI: 10.1007/s10766-016-0480-z.

Henretty, Tom, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan (2013). “A stencil compiler for short-vector SIMD architectures”. In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ICS ’13. Eugene, Oregon, USA: ACM, pp. 13–24. DOI: 10.1145/2464996.2467268.

Iooss, Guillaume (2016). “Detection of linear algebra operations in polyhedral programs”. PhD thesis. University of Lyon, France.

Kelly, Wayne, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott (Nov. 1996). *The Omega Library*. Tech. rep. University of Maryland.

Kong, Martin, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan (2013). “When polyhedral transformations meet SIMD code generation”. In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. PLDI ’13. Seattle, Washington, USA: ACM, pp. 127–138. DOI: 10.1145/2491956.2462187.

Lattner, Chris and Jacques Pienaar (2019). *MLIR Primer: A Compiler Infrastructure for the End of Moore’s Law*.

Lavenier, Dominique, Patrice Quinton, and Sanjay Rajopadhye (1999). “Advanced Systolic Design”. In: *Digital Signal Processing for Multimedia Systems*. Ed. by Keshab K. Parhi and Takao Nishitani. New York, NY, USA: Marcel Dekker, Inc.

Lie, Sean (Aug. 2019). *Wafer-Scale Deep Learning*.

Lu, Qingda, Xiaoyang Gao, Sriram Krishnamoorthy, Gerald Baumgartner, J Ramanujam, and Ponnuswamy Sadayappan (2012). “Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions”. In: *Journal of Parallel and Distributed Computing* 72.3, pp. 338–352. DOI: 10.1016/j.jpdc.2011.09.006.

Meister, Benoît (Dec. 2004). “Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization”. PhD thesis. Université Louis Pasteur.

Pradelle, Benoît, Benoît Meister, Muthu Baskaran, Jonathan Springer, and Richard Lethin (2019). “Polyhedral Optimization of TensorFlow Computation Graphs”. In: *Programming and Performance Visualization Tools*. Ed. by Abhinav Bhatele, David Boehme, Joshua A. Levine, Allen D. Malony, and Martin Schulz. Cham: Springer International Publishing, pp. 74–89. DOI: 10.1007/978-3-030-17872-7_5.

Rossum, Guido (1995). *Python Reference Manual*. Tech. rep. Amsterdam, The Netherlands, The Netherlands.

Schreiber, Robert and Darren C. Cronquist (Feb. 2004). *Near-Optimal Allocation of Local Memory Arrays*. Tech. rep. HPL-2004-24. HP Laboratories.

Sharma, Namita, Preeti Ranjan Panda, Francky Catthoor, Praveen Raghavan, and Tom Vander Aa (June 2015). “Array Interleaving – An Energy-Efficient Data Layout Transformation”. In: *ACM Trans. Des. Autom. Electron. Syst.* 20.3, 44:1–44:26. DOI: 10.1145/2747875.

Vasilache, Nicolas, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen (Oct. 2019).

"The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically". In: *ACM Trans. Archit. Code Optim.* 16.4, 38:1–38:26. DOI: 10.1145/3355606.

Verdoolaege, Sven (2010). "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software - ICMS 2010.* Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49.

Verdoolaege, Sven (2016). *Presburger Formulas and Polyhedral Compilation.* DOI: 10.13140/RG.2.1.1174.6323.

Verdoolaege, Sven, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor (2013). "Polyhedral parallel code generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4, p. 54. DOI: 10.1145/2400682.2400713.

Verdoolaege, Sven, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe (June 2007). "Counting integer points in parametric polytopes using Barvinok's rational functions". In: *Algorithmica* 48.1, pp. 37–66. DOI: 10.1007/s00453-006-1231-0.

Zerrell, Tim and Jeremy Bruestle (2019). *Stripe: Tensor Compilation via the Nested Polyhedral Model.* arXiv: 1903.06498. URL: http://arxiv.org/abs/1903.06498.