Domain-Specific Language Abstractions for Compression

Jessica Ray*, Ajay Brahmakshatriya*, Richard Wang*, Shoaib Kamil†,
Albert Reuther‡, Vivienne Sze*, Saman Amarasinghe*

*MIT CSAIL jray@csail.mit.edu, {ajaybr,rwang99}@mit.edu {sze,samana}@mit.edu

†Adobe Research kamil@adobe.com

‡MIT Lincoln Laboratory reuther@ll.mit.edu

Abstract

While block-based data compression is a critical component of many compression algorithms, little attention has been given towards language support for compression. Compression algorithms share many high-level patterns related to data representation, data partitioning, and data traversals, but these features are not easily captured in popular low-level languages used for compression, such as C. While higher-level libraries and languages such as NumPy, Julia, and MATLAB can provide more intuitive support for parts like array-based data access, they lack native support for expressing the intricate spatial/temporal data dependencies, partitions, and traversals that come with compression.

This paper introduces a set of appropriate high-level abstractions for block-based compression that supports data representation, data partitioning, and data traversals. We demonstrate the power and flexibility of these constructs using a prototype implementation in Python. We believe these abstractions will simplify the implementation complexity of compression algorithms, allowing researchers to focus on the algorithms rather than the implementation details. Additionally, these abstractions can enable the development of a domain-specific language and optimizing compiler that can automatically generate high-performance compression code.

Introduction

Data compression is a fundamental component in much of today's technology driven world, and new algorithms are constantly being developed to keep up with the ever increasing amount of data being generated. Block-based compression has prevailed as a major component of many of the algorithms and forms the core of many popular compression standards. Despite the prevalence and longevity of block-based compression, implementing such algorithms remains an arduous task.

C and C++ are popular languages for implementing compression algorithms due to their superior performance, but their low-level imperative design is a mismatch for the design of compression algorithms, leading to overly complex software systems. This complexity can be seen in the recently released VTM [1] software for the VVC/H.266 standard, where the 134,000 lines of code in the reference software more than doubles the amount of code in the HM [2] software of its predecessor, HEVC/H.265. This is largely due to the data representations, complex spatial/temporal data accesses, and intricate data traversals common to many compression algorithms, none of which are easily captured by the primitives in a language like C. Nor does there exist any high-level language or even library of compression primitives that is able to succinctly capture these components.

Current hand-crafted systems are exceedingly complex, non-malleable, and non-portable. Implementing these systems often requires the efforts of many people focusing on specific areas, making it challenging for any single person to get a global

understanding of the system. It also presents a high barrier of entry for academic researchers wanting to try out new compression techniques, where their implementation options are two equally unappealing choices: implement from scratch or hack into one of the existing systems.

The structure of block-based compression algorithms lends itself well to the design methodology behind domain-specific languages (DSLs). As their name implies, a DSL is designed with a specific domain in mind, providing programming abstractions that capture the high-level characteristics of that domain. For example, while a language such as C would provide low-level data types like `int` and generic data structures like arrays, a DSL for compression would provide a block type that can capture not just the data, but also useful high-level information such as the rectangular layout and spatial dependencies between different pieces of data. DSLs provide a more intuitive programming interface, making it easier to implement existing algorithms and explore new ones, as evidenced by the success of DSLs such as MATLAB for matrix operations, TensorFlow and TACO [3] for tensor operations, Halide [4] for image processing, and StreamIt [5] for stream programs. With a DSL, domain experts can focus their efforts on developing algorithms, rather than having to deal with low-level implementation details.

Developing the appropriate abstractions for compression opens the door for further research into language support for compression. Not only do these abstractions enable DSL creation, but they also enable the development of compression-specific compiler techniques that can automatically generate high-performance code. In this paper, we present a set of abstractions that we believe capture the high-level structure of block-based compression and are appropriate for development of a high-performance DSL. We have implemented our abstractions in Python[1] as part of a small run-time system which allows us to demonstrate the efficacy of our abstractions on real-world compression algorithms. We refer the reader to `https://jray.mit.edu/block-based-compression` where we provide more in-depth descriptions of the run-time system, as well as explicit comparisons to existing compression implementations.

Characteristics of Block-Based Compression

Even with the diversity of compression techniques, many share common characteristics related to data representation, data partitioning, and data traversals. Capturing these characteristics with the right abstractions makes implementation simpler and provides the necessary building blocks for a DSL. We summarize each characteristic below.

Data Representation

Operations in block-based compression are typically applied to multidimensional rectangular regions of data that have many spatial relationships between regions (we will refer to these regions as *blocks*). An appropriate abstraction must capture both the data and these spatial components. For example, a 4:2:0 subsampled block will have half the rows and columns of its unsampled counterpart, with each row and column

[1]Some examples include a pseudo-Python syntax which we preprocess to convert to legal syntax.

in the subsampled version corresponding to every other row and column of the unsampled one. While an array in C might be fine for storing the underlying data here, it does not provide any intuitive description of the subsampled block's layout, nor the spatial mapping between the subsampled and unsampled block. We will often refer to the *location* of one block relative to another block, which means that we can compute a mapping from the elements of one block to another.

Data Partitioning
The spatial relationships previously mentioned are commonly induced by data partitioning. Here, blocks of data are decomposed into smaller sub-blocks, which may in turn be decomposed into even smaller sub-blocks, and so on. For example, in AVC/H.264, a frame is split into macroblocks, and in HEVC/H.265, a CTU can be split into CUs, and CUs into PUs. This decomposition creates a data hierarchy, where sub-blocks can be mapped back to their location in containing blocks. Some of these hierarchies can be very large and specify hundreds or possibly thousands of different ways to decompose a single block. An appropriate abstraction must be able to provide a compact and intuitive notation for specifying the possible decompositions, as well as capture the hierarchical structure of the blocks after partitioning.

Data Traversals
Many compression algorithms specify explicit orderings for traversing both blocks and values within blocks, such as the z-scan and zigzag ordering. Implementing new and existing traversals is not straightforward as they do not tend to fit easily into loop nests, so many systems will manually enumerate the points of the traversal, which is both tedious and inflexible. However, many of the traversals exhibit highly recursive patterns that can be built from simple linear traversals and rotations. These patterns can be used to form the basis of a more compact and flexible representation for traversals, making it easier to explore different orderings.

Compression Abstractions

The abstractions we developed fit into three categories: data representation, data partitioning, and data traversals. To demonstrate our abstractions, we include several code examples selected from our encoder implementations of JPEG, AVC/H.264, and HEVC/H.265. The examples show our abstractions in the context of data initialization, subsampling, entropy coding, intra-prediction, and traversals. We believe these examples provide a good variety of contexts and use cases where our abstractions can make implementation easier and more intuitive.

Data Representation
We introduce five different data representations for use in compression: `Stream`s, `BitStream`s, `Block`s, `View`s, and `Unit`s. Each of these representations was created to better capture the inherent structure of compression data, such as the layout of blocks and the spatial relationships between blocks.

Streams and BitStreams: A `Stream` models the input data for compression and represents it as a finite (or infinite) sequence of data items. The `Stream` itself can represent a structure such as an image or a video sequence, and individual data

items of the stream would typically represent values like pixels. A `BitStream` is used for representing arbitrary length sequences of bits, such as individual code words or the compressed output of an encoder.

Blocks: A `Block` represents a location that has its own underlying data buffer, which allows multiple blocks of data to exist at the same location. Reads and writes to a `Block` access only the data in that `Block`, even if other `Block`s at the same location exist. `Block`s are useful for maintaining independent representations of a region, such as storing different predictions for a single block of data.

Views and Units: `View`s represent a lightweight version of a `Block` and do not have their own data buffers. Rather, they contain a reference to an existing buffer within another `Block` (or `Stream`). A `Unit` is a special case of a `View` and represents a `View` of size 1. `View`s are useful for defining specific blocks of data without having to copy any data, as reads and writes to a `View` propagate through to the underlying buffer that the `View` references.

```
1  rgb_data = Block(...) # external data
2  rgb_stream = Stream((width,height,3)) # (x,y,z)
3  rgb_stream[:,:,0] = rgb_data[::3] # R values
4  rgb_stream[:,:,1] = rgb_data[1::3] # G values
5  rgb_stream[:,:,2] = rgb_data[2::3] # B values
6  R = rgb_stream[:,:,0].elide(2) # WxH View
7  G = rgb_stream[:,:,1].elide(2) # WxH View
8  B = rgb_stream[:,:,2].elide(2) # WxH View
9  Y =  Block(R) # WxH Block
10 Cb = Block(G) # WxH Block
11 Cr = Block(B) # WxH Block
12 Y[:,:]  =  0.299*R[:,:]+0.587*G[:,:]+0.114*B[:,:]
13 Cb[:,:] = -0.169*R[:,:]+0.331*G[:,:]+0.500*B[:,:]
14 Cr[:,:] =  0.500*R[:,:]-0.419*G[:,:]-0.081*B[:,:]
```

Listing 1: Converting interleaved RGB to planar YCbCr format.

Listing 1 gives a simple example from our JPEG implementation and demonstrates reading in interleaved RGB data, representing it in a planar format, and then converting to the YCbCr color space. Line 2 creates a three-dimensional finite `Stream`, `rgb_stream`, with dimensions `width`×`height`×3. Lines 3, 4, and 5 read from the `Block` `rgb_data` containing interleaved RGB data, and write to `rgb_stream` in planar format. Here, we utilize Python's slice syntax to provide our data access abstraction. Each line reads every third element, starting from indices 0, 1, and 2, which corresponds to grouping every `R` value together, then every `G` value, and then every `B` value. Then, all the `R` values are written into `stream` at locations with indices `(x,y,0)`. Similarly, `G` values are written to `(x,y,1)` and `B` to `(x,y,2)`.

A `View` is created whenever a `Stream`, `Block`, or another `View` is sliced, as can be seen on Lines 6, 7, and 8. Here, we slice out three `View`s of size `width`×`height`×1[2] with origins `(0,0,0)`, `(0,0,1)`, and `(0,0,2)` relative to `stream`. These represent the `R`, `G`, and `B` planes, respectively. Since these are `View`s on `stream`, any reads and writes to them propagate through to `stream`.

`Block`s are created by copying size and location information from another `Stream`, `Block`, or `View`. For example, Lines 9, 10, and 11 create new `Block`s with empty buffers covering the same locations as `R`, `G`, and `B`, respectively. These `Block`s hold the result of doing the color space conversion shown on Lines 12, 13, and 14.

`View`s can also map to non-contiguous spaces by specifying non-unit strides. This makes it very simple to represent operations like subsampling in software, such as that used with the chroma components for image and video compression.

[2]Technically the regions are of size `width`×`height` because we use the elide function to drop the last dimension. A dimension with size 1 can be elided so that it does not need to be written.

```
1 # chroma can be a Block, View, or Stream
2 chroma_420 = chroma[::2,::2] # W/2xH/2 View
3 chroma_411 = chroma[::4,:]   # W/4xH View
4 chroma_410 = chroma[::4,::2] # W/4xH/2 View
```

Listing 2: Subsampling using `Views`.

Line 2 in Listing 2 performs 4:2:0 subsampling on `chroma` by specifying a stride of two over the columns and rows. This means that every row of `chroma_420` maps to every other row of `chroma`, as well as every column mapping to every other column. Lines 3 and 4 show 4:1:1 and 4:1:0 subsampling, respectively.

Listing 3 gives a look at using `BitStream`s and `Unit`s in the context of encoding AC coefficients for JPEG. We start by creating `BitStream` `encoded` on line 1 to hold our result, as well as two `BitStream`s on Lines 2 and 3 representing the special run-length encoding symbols `EOB` and `ZRL`. Here, Line 13 iterates through each `8x8` block of `Y` (where `blocked` represents the collection of `8x8` blocks), and Line 19 iterates through each AC coefficient of the `8x8` block in zigzag order (we will discuss iteration and the `scan` construct in more detail later).

```
1  encoded = BitStream()
2  luma_EOB = BitStream(0b1010, 4)
3  luma_ZRL = BitStream(0b11111111001, 11)
4  blocked = PTree()
5  pt_root(Y,blocked):
6    pt_and:
7      for i in range(0, Y.size[1],8):
8        row = Y[:,i:(i+8)] # Wx8 View
9        for j in range(0, Y.size[0],8):
10         col = row[j:(j+8),0:8] # 8x8 View
11         pt_leaf(col)
12 # blk_8x8: 8x8 View
13 for blk_8x8 in scan(blocked):
14   prev_blk = scan_offset(-1) # 8x8 View
15   DC_coeff = blk_8x8[0,0] # Unit
16   ... encode DC coefficient ...
17   nzeros = 0
18   # AC_coeff: Unit
19   for AC_coeff in scan(zigzag(blk_8x8))[1:]:
20     if AC_coeff == 0:
21       nzeros += 1
22     else:
23       for i in range(nzeros//16):
24         encoded.append_bits(luma_ZRL)
25       cat = floor(log2(abs(AC_coeff)))+1
26       # AC_codes: lookup table
27       # cat_code: BitStream
28       cat_code = AC_codes[nzero_rem%16][cat-1]
29       encoded.append_bits(cat_code)
30       if coeff > 0:
31         code = BitStream(AC_coeff,cat)
32         encoded.append_bits(code)
33       else:
34         code = BitStream(~abs(AC_coeff),cat)
35         encoded.append_bits(code)
36       nzeros = 0
37   if nzeros > 0:
38     encoded.append_bits(luma_EOB)
```

Listing 3: Encoding luma AC coefficients in JPEG.

Each AC coefficient is a `Unit`, and if it is non-zero, the appropriate Huffman codes are computed (Lines 25, 28, 31, and 34) and appended to `encoded` on Lines 29, 32, and 35. Runs of 16 or more zeros, or trailing zeros, are appended to `encoded` on Lines 24 and 38 using the `BitStream`s that represent the `ZRL` and `EOB` codes.

In each example above, our abstraction makes it easy to read and write arbitrary sub-blocks of data and hides the low-level tasks of manually computing access indices and mappings between blocks. While slicing itself is not a new concept, the ability to map between block locations is a unique feature of our abstraction.

Data Partitioning

Data partitioning represents the decomposition of a block of data into smaller sub-blocks, typically of many different sizes. For example, in AVC/H.264 we decompose a frame into macroblocks, and then decompose the macroblocks into sub-macroblocks for inter-prediction and intra-prediction. In HEVC/H.265, CTUs are decomposed into CUs, and CUs into PUs. Additionally, these sub-blocks often have specific coding orders, such as coding CUs in depth-first order. Our partitioning abstraction provides a compact way to specify all the different possible decompositions of a block, specify the traversal order across the sub-blocks, and maintain spatial mappings between sub-blocks and their containing blocks.

To express possible partitions, we use a tree representation called a `PTree` which is inspired by and-or trees [6]. Consider the `PTree` shown in Figure 1 which represents a quad-tree decomposition of a `32x32` CTU in HEVC/H.265 down into CUs of size

32x32, 16x16, or 8x8. Each AND defines a set of sub-blocks that must occur together in a decomposition. For example, if we split a 32x32 CTU, it must contain four 16x16 CUs. An OR defines options for a decomposition. That same 32x32 CTU has two options: do not split, or split into the four 16x16 CUs. Each intermediate and leaf CU is also annotated with its origin (@(x,y)) relative to the immediate containing block. For example, the four 16x16 interior nodes have origins (0,0), (16,0), (0,16), and (16,16) with respect to the root, and the 8x8 leaves have origins (0,0), (8,0), (0,8), and (8,8), respectively, with respect to their parent 16x16 node. A depth-first scan of this tree produces a z-scan coding order due to the relative positions of each leaf sub-block in the tree. The order can easily be changed by rearranging the subtrees and leaves.

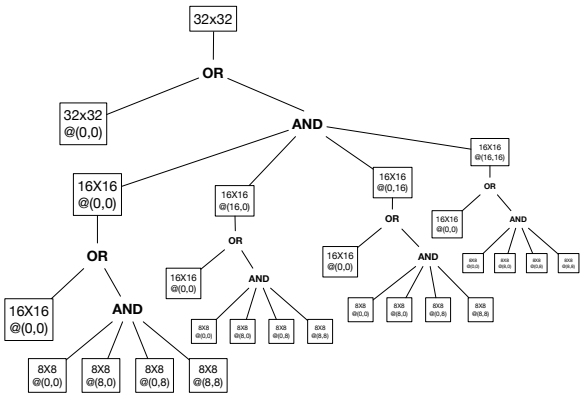


Figure 1: PTree for partitioning a 32x32 CTU into CUs in HEVC/H.265.

Listing 4 shows an implementation of the PTree in Figure 1. Each component of the tree in the figure has a direct counterpart in the code: the root is created with pt_root, ORs with pt_or, ANDs with pt_and, and leaves with pt_leaf. This function recursively creates PTree subtrees, where each subtree either splits the current block into four equal sized quadrants (lines 9-12), or keeps the current sub-block as is (line 6). pt_subtree on Lines 13 through 16 adds the root of the subtree as the child of the pt_and node on Line 7. Splitting continues until the size of block is 8x8, which can no longer be split. This is captured by the condition M>8 on the pt_and.

```
1  def CTU_to_CUs(ctu): # ctu: MxM View
2    M = ctu.size[0]
3    ptree = PTree()
4    pt_root(ctu,ptree):
5      pt_or:
6        pt_leaf(ctu)
7        pt_and(M>8):
8          # q0, q1, q2, q3: M/2xM/2 Views
9          q0 = ctu[:M/2,:M/2]
10         q1 = ctu[M/2:,:M/2]
11         q2 = ctu[:M/2,M/2:]
12         q3 = ctu[M/2:,M/2:]
13         pt_subtree(CTU_to_CUs(q0))
14         pt_subtree(CTU_to_CUs(q1))
15         pt_subtree(CTU_to_CUs(q2))
16         pt_subtree(CTU_to_CUs(q3))
17   return ptree
18 ctu = View(...) # MxM
19 # The partition separates into 3 PTrees:
20 # no split, 16x16 split, or 8x8 split
21 for ptree in separate(CTU_to_CUs(ctu)):
22   for cu in scan(ptree):
23     # cu is either a 32x32, 16x16, or 8x8
       View, depending on ptree
```

Listing 4: PTree for partitioning CTUs into CUs in HEVC/H.265.

PTrees provide a compact representation for all the possible decompositions of a block and are most commonly applied in conjunction with scan constructs to iterate over the sub-blocks in the decomposition. We present the scan in the next section on data traversals, but the important point for now is that a scan gathers all the leaf nodes and returns a single leaf per iteration. The leaves of an AND-only PTree (no OR nodes) can be collected with a simple pre-order depth-first traversal (DFS) since this tree would express only a single decomposition. A tree with OR nodes represents multiple decompositions, so it first needs to be separated into individual AND-only PTrees representing each unique decomposition. Once it is separated, the individual PTrees can be iterated over. Listing 4 shows an example of separating a PTree on Line 21, iterating through each resulting PTree, and then iterating through the individual CUs for each unique PTree on Line 22.

Data Traversals

Traversals define ordered iterations over both blocks (*block traversals*) and individual values of the blocks (*unit traversals*), adding another layer of complexity to our set of spatial relationships. We have seen examples of block traversals already with our `PTree` abstraction: the ordering of the leaves defines the iteration order of the blocks. However, unit traversals are more complex and require a different abstraction for specifying orderings

We introduce a recursive notation for unit traversals inspired by parametric Lindenmayer Systems [7] (L-systems) used to model plant growth in biology. Conceptually, L-systems operate over a grid and generate an ordered enumeration of all the points to be visited in the grid.

```
1  zz = ScanOrder(blk)
2  size = 8
3  ST = Step()
4  i = Var()
5  A,B = NTerm([i]), NTerm([i])
6  zz.axiom([A(1)])
7  zz.rule(A(i),[ST,135,ST*i,315,B(i+1),45,ST*i,225,ST],i<size-1)
8  zz.rule(B(i),[ST,225,ST*i,45,A(i+1),315,ST*i,135,ST],i<size-1)
9  zz.rule(A(i),[ST,135,ST*(size-1),225,ST],i==size-1)
10 zz.rule(B(i),[ST,225,ST*(size-1),135,ST],i==size-1)
```

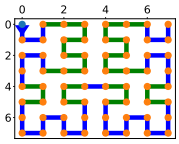
Listing 5: A zigzag unit traversal.

Listing 5 defines a zigzag traversal over an `8x8` block using our recursive notation. It is characterized by three main components: terminals, non-terminals, and production rules. A `Step` terminal (Line 3) defines moving in a straight line, and a rotation terminal (the integers on Lines 7 through 10) changes the orientation of that line. Non-terminals (`A` and `B`) represent symbols that can be replaced with a production rule.

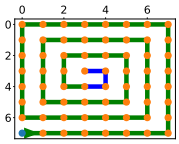
At a high level, the points of the zigzag are enumerated starting from the axiom on Line 6. `A(1)` is expanded into a string using a matching rule, which is the rule on Line 7 in this case. A rule can only be applied if its condition is met, which is `i<size-1` here. The expanded string now contains the other non-terminal `B`, which is expanded using the rule for `B` on Line 8. This process of expanding continues until no more production rules can be applied.

Traversals over `Stream`s, `Block`s, and `View`s are realized using the `scan` construct, which is our abstraction of an iterator with knowledge of block and unit traversals. If a block or unit traversal is specified, a `scan` returns each sub-block/unit according to the order of the traversal. Line 13 in Listing 3 splits `Y` into `8x8` sub-blocks according to the partition defined on Lines 4 to 11 and then iterates through the `8x8` sub-blocks. The `scan` on Line 19 performs a unit traversal through the AC coefficients of one of the `8x8` blocks in zigzag order (defined in Listing 5), skipping the first value of the block which is the DC coefficient. `scan`s also support temporal accesses, e.g. accessing other blocks or units with respect to the current iteration. This is shown on line 14 of Listing 3 with `scan_offset(-1)`, which returns the `8x8` block from the prior iteration (if it exists). This is necessary for computing the differences between DC coefficients.

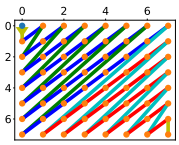
This notation is much more flexible than the alternative options of implementing the ordering as a loop nest or manually enumerating the individual points yourself. It can easily represent other complex traversals, like the ones shown in Figure 2.



```
order=3
hilbert = ScanOrder(blk)
hilbert.axiom([A(0)])
hilbert.rule(A(i),[90,B(i+1),ST,270,A(i+1),ST,A(i+1),270,ST,B(i+1),90],i<order)
hilbert.rule(B(i),[270,A(i+1),ST,90,B(i+1),ST,B(i+1),90,ST,A(i+1),270],i<order)
```



```
perimeter = ScanOrder(blk)
perimeter.axiom([A(N-1)])
perimeter.rule(A(i),[ST,270,ST,270,ST],i==1)
perimeter.rule(A(i),[ST*i,270,ST*i,270,ST*i,270,ST*(i-1),270,ST,A(i-2)],i>1)
```



```
diagonal = ScanOrder(blk)
diagonal.axiom([90,ST,A(1,N-1),135,ST])
diagonal.rule(A(i,j),[225,ST*i,B(i+1,j)],i<N)
diagonal.rule(B(i,j),[225,SK*(i-1),270,SK*i,A(i,j)],i<N)
diagonal.rule(A(i,j),[225,ST*(j-1),B(i,j-1)],(i==N)&(j>1))
diagonal.rule(B(i,j),[225,SK*(j-1),270,SK*j,A(i,j)],(i==N)&(j>1))
```

Figure 2: Hilbert and perimeter traversals from [8], and a diagonal traversal from HEVC/H.265. `SK` is a `Skip` and behaves like a `Step`, but the points it moves between are excluded from the final enumeration.

Complex Spatial Dependencies

So far, we have seen various code snippets that show our abstractions in different contexts. However, the spatial accesses have been straightforward, accessing only data within one's own block. Here, we provide a more complete example modeled after `4x4` luma intra-prediction in the AVC/H.264 encoder. This particular example requires accessing data in another block relative to one's own block, which is a subtle, yet important difference from the accesses we have seen. We begin by examining the parts of the algorithm that lead up to the accesses.

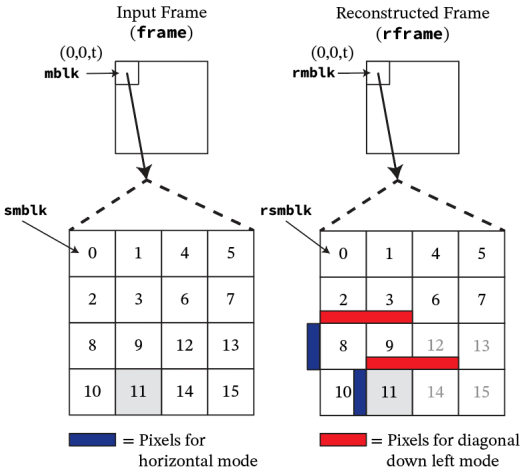


Figure 3: Block order and spatial references for two luma intra-prediction modes in AVC/H.264.

Listing 6 implements prediction on `4x4` sub-macroblocks using the horizontal and diagonal-down-left modes of AVC/H.264. In Listing 6 on Lines 1 through 12, we specify a `PTree` that partitions a macroblock into `4x4` sub-macroblocks, producing the z-scan ordering annotated on the sub-macroblocks in Figure 3. Lines 13 through 18 define a raster unit traversal through a `4x4` `View`.

We assume there is a `View` on a frame of pixels, `frame`, and create a reconstructed version of the frame, `rframe`, at the same location (Lines 19 and 20). On Line 22, both `frame` and `rframe` are decomposed into macroblocks (`mblk` and `rmblk`), which we iterate through with the `scan`. We assume `to_mblk` is a `PTree` similar to that in Listing 3, but creates `16x16` blocks rather than `8x8` blocks. Next, we decompose the macroblocks into `4x4` sub-macroblocks, `smblk` and `rsmblk`, and iterate through them on Line 24. Figure 3 also shows the relationship between the locations of macroblocks and sub-macroblocks in `frame` and `rframe`.

On Lines 25 and 26, we create two `Block` copies of `smblk` to hold the result of

prediction and Lines 28 and 30 access the reference pixels that are needed for each prediction mode, where the reference pixels are taken from `rframe`. Horizontal mode needs the column of four pixels to the left of `smblk`, and diagonal-down-left mode needs the row of eight pixels above, which is shown for `smblk`s `8` and `11` in Figure 3. `smblk 8` cannot be predicted with horizontal mode since it is on the left-edge of the frame boundary, meaning there is no data available on the left. This is the reason for the check on Line 31. `smblk 11` is not on the edge, so it can perform horizontal prediction shown on Lines 32 through 35 using values from `rsmblk 10`.

```
1  def zscan_grid(blk):
2    ptree = PTree()
3    M = blk.size[0]
4    pt_root(mblk,ptree):
5      pt_and(M>4):
6        pt_subtree(zscan_grid(blk[0:M/2,0:M/2]))
7        pt_subtree(zscan_grid(blk[M/2:,0:M/2]))
8        pt_subtree(zscan_grid(blk[0:M/2,M/2:]))
9        pt_subtree(zscan_grid(blk[M/2:,M/2:]))
10     pt_and(M==4):
11       pt_leaf(blk)
12   return ptree
13 def raster_unit(blk): # blk: 4x4 View
14     raster = ScanOrder(blk)
15     raster.axiom([A(0)])
16     raster.rule(A(i),[ST*3,90,SK,90,SK*3,180,
17         A(i+1)],i<4)
18     return raster
19 frame = View(...) # a frame of raw pixels
20 rframe = Block(frame) # reconstructed version
21 # mblk,rmblk: 16x16 Views
22 for mblk,rmblk in scan(to_mblk(mblk), to_mblk(
       rmblk)):
23   # smblk,rsmblk: 4x4 Views
24   for smblk,rsmblk in scan(zscan_grid(mblk),
       zscan_grid(rmblk)):
25     hor_pred = Block(smblk) # 4x4 Block
26     ddl_pred = Block(smblk) # 4x4 Block
27     # hor_ref: 1x4 View
28     hor_ref = rframe[smblk[-1,:]]
29     # ddl_ref: 8x1 View
30     ddl_ref = rframe[smblk[0:8,-1]].wrt_scan()
31     if hor_ref != None:
32       hor_pred[0,:] = hor_ref[:,:]
33       hor_pred[1,:] = hor_ref[:,:]
34       hor_pred[2,:] = hor_ref[:,:]
35       hor_pred[3,:] = hor_ref[:,:]
36     if ddl_ref != None:
37       ... do prediction ...
38     ... select a final prediction ...
39     recons = ... reconstruct rsmblk ...
40     # r,p: Units
41     for r,p in scan(raster_unit(rsmblk),
       raster_unit(recons)):
42       r.write(p)
```

Listing 6: Modified AVC/H.264 4x4 intra-prediction.

Diagonal-down-left mode for `smblk 8` is straightforward and uses values from `rsmblk 2` and `3`. However, assume we are predicting `smblk 11`, which means `rsmblk`s `12-15` are not available yet. `smblk 11` needs values from `rsmblk 12`, which does not exist, so we cannot do the prediction in this case. To enforce this, we provide `wrt_scan()` on Line 30 which indicates that values should not be returned if they violate the innermost scan order (which is the z-scan across sub-macroblocks in this case). Since we track location information in our data abstraction, we can automatically determine invalid accesses like this rather than the user having to manually check where `smblk` is relative to the other blocks.

The remaining parts of the code assume one of the modes is selected and eventually reconstructed. Lines 41 and 42 write the reconstructed values into `rmsblk` unit-by-unit.

Related Work

Countless implementations of individual compression algorithms exist, but we have not seen any explicit language support for compression, whether it be through the use of DSLs or libraries of compression primitives. We focused primarily on the reference software for standards, such as JM [9], HM [2], and VTM [1] for the AVC/H.264, HEVC/H.265, and VVC/H.266 standards, respectively, as well as IJG's JPEG software [10]. While the structure of each is highly intertwined with the individual algorithms, these pieces of software provided us with several insights into the common features in compression implementations.

Our abstractions were initially modeled after those in streaming languages such as [5]. However, the one-dimensional nature of those streams was not able to capture the hierarchical block structure of compression data. Array-OL [11] is a dataflow specification language for expressing data dependencies in infinite multidimensional

streams. They present a tile object that can be moved around a grid, similar to our blocks and traversals. However, Array-OL is a specification language, so it does not provide any of the control-flow that we rely on to express our abstractions (e.g. conditionals within `PTree`s and traversals). It does provide an interesting look at non-rectangular shapes, which could be useful for providing language support for compression algorithms that fall outside of the scope of block-based compression.

Conceptually, our approach to compression abstractions are similar to Scanner [12] and Halide [4], which provide abstractions for video analysis and image processing, respectively. Both provide abstractions focusing on efficient and intuitive storage representations, pixel-level spatial access, and high-performance pixel-level optimizations. While they lack the abstractions needed for compression, such as partitioning and traversals, the design and implementation of their high-performance compilers may provide useful insights for developing such a compiler for our abstractions.

Conclusion

Imperative languages like C are poorly suited to the implementation of block-based compression algorithms as their low-level design cannot easily model the structure of these algorithms. In this paper, we presented a set of language abstractions that capture the main characteristics of these algorithms, namely data representation, partitioning, and traversals. These abstractions provide a much more intuitive framework for implementing compression algorithms, allowing researchers to focus on the actual algorithm rather than implementation details. We believe these abstractions are well-suited for a high-performance domain-specific compression language.

References

[1] Joint Video Experts Team, "VTM software (v10.2)," 2020.

[2] Joint Collaborative Team on Video Coding, "HM software (v16.22)," 2016.

[3] Fredrik Kjolstad et al., "Taco: A tool to generate tensor algebra kernels," in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017, pp. 943–948.

[4] Jonathan Ragan-Kelley et al., "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," Acm Sigplan Notices, vol. 48, no. 6, pp. 519–530, 2013.

[5] William Thies et al., "StreamIt: A language for streaming applications," in International Conference on Compiler Construction. Springer, 2002, pp. 179–196.

[6] Brigitte Chauvin et al., "And/or trees revisited," Combinatorics Probability and Computing, vol. 13, no. 4-5, pp. 475–497, 2004.

[7] Przemyslaw Prusinkiewicz and James Hanan, Lindenmayer systems, fractals, and plants, vol. 79, Springer Science & Business Media, 2013.

[8] Erdoğan Aldemir et al., "Binary medical image compression using the volumetric run-length approach," The Imaging Science Journal, vol. 67, no. 3, pp. 123–135, 2019.

[9] Joint Video Team of ISO/IEC MPEG & ITU-T VCEG, "JM software (v19.0)," 2015.

[10] Independent JPEG Group, "jpeg software (v9d)," .

[11] Pierre Boulet, Array-OL revisited, multidimensional intensive signal processing specification, Ph.D. thesis, INRIA, 2007.

[12] Alex Poms et al., "Scanner: Efficient video analysis at scale," ACM Transactions on Graphics (TOG), vol. 37, no. 4, pp. 1–13, 2018.