Domain-Specific Language Abstractions for Compression

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

MIT CSAIL, Adobe Research, MIT Lincoln Laboratory

The Case for a Compression DSL

DSL = domain-specific language

- **Motivation: Experts should be able to focus on the compression and not have to worry about implementation details**
  - Overall lack of programming language support
  - Current hand-crafted compression systems are exceedingly complex, non-malleable, and non-portable
  - Implementation often requires efforts of many people
- **Problem: Primitives in C/C++ are a poorly matched to the structure of compression algorithms**
  - Lack multidimensional data structures, cannot natively capture the complex data dependencies, mismatched control flow, require manual optimization
  - Higher-level languages (Julia, Python/Numpy, etc) provide more multidimensional support
    - Still lack ability to capture the dependencies, control flow, optimizations
- **DSL Benefit: DSLs provide a more intuitive programming interface**
  - Data structures that model the multidimensional structure, data access, dependencies
  - Control flow designed for iterating through the data structures
  - Provides concise, high-level syntax
- **Problem: Optimizations**
  - Optimizations are necessary for getting practical run-time performance
  - Currently, optimizations are hand-coded, often in assembly--significantly increases the complexity factor
- **DSL Benefit: DSLs enable development of an optimizing compiler**
  - Can automatically generate high-performance code
  - Has intrinsic knowledge of the domain-specific abstractions, opening the door for compression-specific optimizations
    - Ex: wavefront parallelism, non-SIMD vectorization
- **Primary goals**
  1. (This work) Develop domain-specific abstractions for block-based compression
     - The abstractions are the main building blocks of DSLs
     - Focusing on image and video: JPEG, WebP, AVC/H.264, HEVC/H.265, VVC/H.266
  2. Develop an optimizing compiler based around the abstractions
  3. Extend the abstractions for other categories of compression

```
INIT_YMM avx2
cglobal pixel_sad_64x64, 4,7,6
    xorps           m0, m0
    xorps           m5, m5
    mov             r4d, 8
    lea             r5, [r1 * 3]
    lea             r6, [r3 * 3]
.loop:
    movu           m1, [r0]
    movu           m2, [r2]
    movu           m3, [r0 + 32]
    movu           m4, [r2 + 32]
    psadbw         m1, m2
    psadbw         m3, m4
    paddd          m0, m1
    paddd          m5, m3
    movu           m1, [r0 + r1]
    movu           m2, [r2 + r3]
    movu           m3, [r0 + 32 + r1]
    movu           m4, [r2 + 32 + r3]
    psadbw         m1, m2
    psadbw         m3, m4
    paddd          m0, m1
    paddd          m5, m3
    movu           m1, [r0 + 2 * r1]
    movu           m2, [r2 + 2 * r3]
    movu           m3, [r0 + 2 * r1 + 32]
    movu           m4, [r2 + 2 * r3 + 32]
    psadbw         m1, m2
    psadbw         m3, m4
    paddd          m0, m1
    paddd          m5, m3
    movu           m1, [r0 + r5]
    movu           m2, [r2 + r6]
    movu           m3, [r0 + 32 + r5]
    movu           m4, [r2 + 32 + r6]
    psadbw         m1, m2
    psadbw         m3, m4
    paddd          m0, m1
    paddd          m5, m3
    lea     r2,     [r2 + 4 * r3]
    lea     r0,     [r0 + 4 * r1]
    movu           m1, [r0]
    movu           m2, [r2]
    movu           m3, [r0 + 32]
    movu           m4, [r2 + 32]
    psadbw         m1, m2
    psadbw         m3, m4
    paddd          m0, m1
    paddd          m5, m3
    movu           m1, [r0 + r1]
    movu           m2, [r2 + r3]
    movu           m3, [r0 + 32 + r1]
    movu           m4, [r2 + 32 + r3]
    psadbw         m1, m2
    psadbw         m3, m4
    paddd          m0, m1
    paddd          m5, m3
    movu           m1, [r0 + 2 * r1]
    movu           m2, [r2 + 2 * r3]
    movu           m3, [r0 + 2 * r1 + 32]
    movu           m4, [r2 + 2 * r3 + 32]
    psadbw         m1, m2
    psadbw         m3, m4
    paddd          m0, m1
    paddd          m5, m3
    movu           m1, [r0 + r5]
    movu           m2, [r2 + r6]
    movu           m3, [r0 + 32 + r5]
    movu           m4, [r2 + 32 + r6]
    psadbw         m1, m2
    psadbw         m3, m4
    paddd          m0, m1
    paddd          m5, m3
    lea     r2,     [r2 + 4 * r3]
    lea     r0,     [r0 + 4 * r1]
    dec         r4d
    jnz         .loop
    paddd          m0, m5
    vextracti128   xm1, m0, 1
    paddd          xm0, xm1
    pshufd         xm1, xm0, 2
    paddd          xm0,xm1
    movd            eax, xm0
    RET
```

Hand-coded assembly for the simple sum-of-absolute-differences operation (from x265 software)

Framework Code Size

JM (H.264 reference)
> 120,000 lines of C/C++

x264
> 68,000 lines of assembly
> 37,000 lines of C/C++

libvpx (vp8)
> 47,000 lines of assembly
> 5,000 lines of C/C++

HM (H.265 reference)
> 60,000 lines of C/C++

x265
> 179,000 lines of assembly
> 96,000 lines of C/C++

aom (AV1 reference)
> 215,000 lines of C/C++

VTM (H.266 reference)
> 134,000 lines of C/C++

kvazaar (H.265 encoder)
> 2,800 lines of assembly
> 32,000 lines of C/C++

WebP
> 50,000 lines of C/C++

Compression Abstractions

Data Representation

- Goals
  - Provide multidimensional data structures
  - Capture spatial information
  - Provide intuitive data access
  - Support elementwise and reduction operations
- Our abstractions
  - `Block`, `View`, `Stream`, `BitStream`

Data Partitioning

- Goals
  - Capture partition options in single tree structure
  - Provide high-level syntax for creating the tree
  - Support tree iteration
  - Preserve spatial information
- Our abstractions
  - `PTree, AndTree`

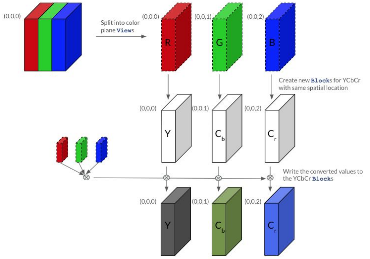
Data Traversals

- Goals
  - Capture temporal dependencies that imply data orderings
  - Provide high-level syntax for representing dependencies
  - Support iteration with the orderings
- Our abstractions
  - `UnitTraversal`

Compression Abstractions

Data Representation

- Goals
  - Provide multidimensional data structures
  - Capture spatial information
  - Provide intuitive data access
  - Support elementwise and reduction operations
- Our abstractions
  - `Block`, `View`, `Stream`, `BitStream`



Data Partitioning

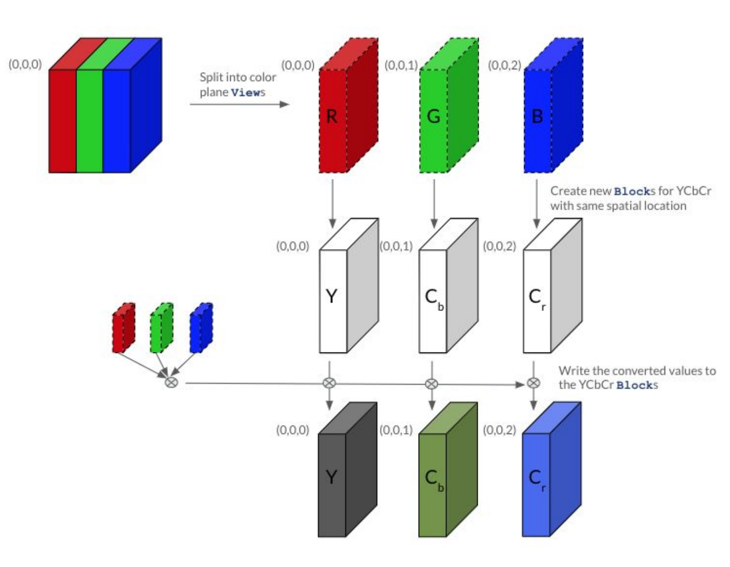
- Goals
  - Capture partition options in single tree structure
  - Provide high-level syntax for creating the tree
  - Support tree iteration
  - Preserve spatial information
- Our abstractions
  - `PTree, AndTree`

Data Traversals

- Goals
  - Capture temporal dependencies that imply data orderings
  - Provide high-level syntax for representing dependencies
  - Support iteration with the orderings
- Our abstractions
  - `UnitTraversal`

Data Representation

- Abstraction goals
  - Provide multidimensional data structures
  - Capture the spatial information in regions of data
  - Hide the underlying data storage format
  - Provide more intuitive access to data
  - Prevent unnecessary data copies
  - Support elementwise and reduction operations on data
- `Block` abstraction
  - A multidimensional region of data with an underlying buffer
- `View` abstraction
  - Represents a shared copy of some part of a `Block`'s buffer
  - Reads/writes automatically propagated to the underlying `Block`
  - Allows a for a different data representation without copying
  - Has a flexible slicing syntax for extracting arbitrary regions of data with varying strides
- `Stream/BitStream` abstraction
  - Models the input/output as a possibly infinite series of data elements
  - Bit-level version provides support for common bit operations such as slicing, masking, packing, shuffling, etc
- Existing implementations
  - Rely on low-level types like arrays which do not provide any spatial information
  - Programmer has to track all dependencies manually



Visualization of color conversion from RGB->YCbCr

RGB->YCbCr Color Conversion

```python
# Color conversion function
def rgb_to_ycc(R_plane, G_plane, B_plane, rescale, rgb_ycc):
    Y_plane = Block(R_plane)
    Cb_plane = Block(G_plane)
    Cr_plane = Block(B_plane)
    Domain i,j
    Y_plane[i,j] = (rgb_ycc[rescale[R_plane[i,j]],0] +
         rgb_ycc[rescale[G_plane[i,j]],1] +
         rgb_ycc[rescale[B_plane[i,j]],2]) >> 16 - 128
    Cb_plane[i,j] = (rgb_ycc[rescale[R_plane[i,j]],3] +
         rgb_ycc[rescale[G_plane[i,j]],4] +
         rgb_ycc[rescale[B_plane[i,j]],5]) >> 16 - 128
    Cr_plane[i,j] = (rgb_ycc[rescale[R_plane[i,j]],6] +
         rgb_ycc[rescale[G_plane[i,j]],7] +
         rgb_ycc[rescale[B_plane[i,j]],8]) >> 16 - 128
    return Y_plane,Cb_plane,Cr_plane

image = Stream(w,h,3)
# Separate image into separate color planes
R = image[:,:,0] # View
G = image[:,:,1] # View
B = image[:,:,2] # View
# Color conversion
Y, Cb, Cr = rgb_to_ycc(R, G, B, 3, rescale_values, rgb_ycc_table)
# Signal the type of padding to use
Y_padded = (8 * round(Y.dims[0] / 8), 8 * round(Y.dims[1] / 8))
Y = Block(Y, Y_padded, padding_type=Block.EXTEND)
Cb_padded = (8 * round(Cb.dims[0] / 8), 8 * round(Cb.dims[1] / 8))
Cb = Block(Cb, Cb_padded, padding_type=Block.EXTEND)
Cr_padded = (8 * round(Cr.dims[0] / 8), 8 * round(Cr.dims[1] / 8))
Cr = Block(Cr, Cr_padded, padding_type=Block.EXTEND)
```

With our abstractions: concise, supports multidimensional representation, intuitive access

```c
// Call color conversion, downsampling (which does right edge padding),
// and bottom edge padding
METHODDEF(void)
pre_process_data (j_compress_ptr cinfo,
                  JSAMPARRAY input_buf, JDIMENSION *in_row_ctr,
                  JDIMENSION in_rows_avail,
                  JSAMPIMAGE output_buf, JDIMENSION *out_row_group_ctr,
                  JDIMENSION out_row_groups_avail) {
    my_prep_ptr prep = (my_prep_ptr) cinfo->prep;
    int numrows, ci;
    JDIMENSION inrows;
    jpeg_component_info * compptr;
    // Iterate through the image rows
    while (*in_row_ctr < in_rows_avail &&
           *out_row_group_ctr < out_row_groups_avail) {
        // Color conversion
        inrows = in_rows_avail - *in_row_ctr;
        numrows = cinfo->max_v_samp_factor - prep->next_buf_row;
        numrows = (int) MIN((JDIMENSION) numrows, inrows);
        // Calls rgb_ycc_convert(...)
        (*cinfo->cconvert->color_convert) (cinfo, input_buf + *in_row_ctr,
                                          prep->color_buf,
                                          (JDIMENSION) prep->next_buf_row,
                                          numrows);
        *in_row_ctr += numrows;
        prep->next_buf_row += numrows;
        prep->rows_to_go -= numrows;
        if (prep->next_buf_row == cinfo->max_v_samp_factor) {
            // Calls fullsize_downsample(...)
            (*cinfo->downsample->downsample) (cinfo,
                                              prep->color_buf, (JDIMENSION) 0,
                                              output_buf, *out_row_group_ctr);
            prep->next_buf_row = 0;
            (*out_row_group_ctr)++;
        }
        // Pad the bottom
        if (prep->rows_to_go == 0 &&
            *out_row_group_ctr < out_row_groups_avail) {
            for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
                ci++, compptr++) {
                numrows = (compptr->v_samp_factor * compptr->DCT_v_scaled_size) /
                    cinfo->min_DCT_v_scaled_size;
                expand_bottom_edge(output_buf[ci],
                                   compptr->width_in_blocks * compptr->DCT_h_scaled_size,
                                   (int) (*out_row_group_ctr * numrows),
                                   (int) (out_row_groups_avail * numrows));
            }
            *out_row_group_ctr = out_row_groups_avail;
            break;
        }
    }
}
// Pad right edge
LOCAL(void)
expand_right_edge (JSAMPARRAY image_data, int num_rows,
                   JDIMENSION input_cols, JDIMENSION output_cols) {
    register JSAMPROW ptr;
    register JSAMPLE pixval;
    register int count;
    int row;
    int numcols = (int) (output_cols - input_cols);
    if (numcols > 0) {
        for (row = 0; row < num_rows; row++) {
            ptr = image_data[row] + input_cols;
            pixval = ptr[-1];
            for (count = numcols; count > 0; count--)
                *ptr++ = pixval;
        }
    }
}
// Pad left edge
LOCAL(void)
expand_bottom_edge (JSAMPARRAY image_data, JDIMENSION num_cols,
                    int input_rows, int output_rows) {
    register int row;
    fprintf(stderr, "In expand %d->%d, %d\n", input_rows, output_rows, num_cols);
    for (row = input_rows; row < output_rows; row++) {
        jcopy_sample_rows(image_data, input_rows-1, image_data, row,
                          1, num_cols);
    }
}
// Copy data and pad the right edge
METHODDEF(void)
fullsize_downsample (j_compress_ptr cinfo, jpeg_component_info * compptr,
                     JSAMPARRAY input_data, JSAMPARRAY output_data) {
    jcopy_sample_rows(input_data, 0, output_data, 0,
                      cinfo->max_v_samp_factor, cinfo->image_width);
    expand_right_edge(output_data, cinfo->max_v_samp_factor, cinfo->image_width,
                      compptr->width_in_blocks * compptr->DCT_h_scaled_size);
}

// Perform the actual color conversion
METHODDEF(void)
rgb_ycc_convert (j_compress_ptr cinfo,
                 JSAMPARRAY input_buf, JSAMPIMAGE output_buf,
                 JDIMENSION output_row, int num_rows) {
    my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
    register int r, g, b;
    // This is what is generated by rgb_ycc_start
    register INT32 * ctab = cconvert->rgb_ycc_tab;
    register JSAMPROW inptr;
    register JSAMPROW outptr0, outptr1, outptr2;
    register JDIMENSION col;
    JDIMENSION num_cols = cinfo->image_width;
    // Iterate through each row of the current part of the image
    while (--num_rows >= 0) {
        // Pull out the sample for each color component
        inptr = *input_buf++;
        outptr0 = output_buf[0][output_row];
        outptr1 = output_buf[1][output_row];
        outptr2 = output_buf[2][output_row];
        output_row++;
        // Iterate through the columns of the row
        for (col = 0; col < num_cols; col++) {
            // Get the individual color samples from each row
            r = GETJSAMPLE(inptr[RGB_RED]);
            g = GETJSAMPLE(inptr[RGB_GREEN]);
            b = GETJSAMPLE(inptr[RGB_BLUE]);
            inptr += RGB_PIXELSIZE;
            // Do the actual conversion
            // Y
            outptr0[col] = (JSAMPLE)
                ((ctab[r+R_Y_OFF] + ctab[g+G_Y_OFF] + ctab[b+B_Y_OFF])
                >> SCALEBITS);
            // Cb
            outptr1[col] = (JSAMPLE)
                ((ctab[r+R_CB_OFF] + ctab[g+G_CB_OFF] + ctab[b+B_CB_OFF])
                >> SCALEBITS);
            // Cr
            outptr2[col] = (JSAMPLE)
                ((ctab[r+R_CR_OFF] + ctab[g+G_CR_OFF] + ctab[b+B_CR_OFF])
                >> SCALEBITS);
        }
    }
}
```

IJG implementation: long, most code dedicated to computing appropriate 1D indices

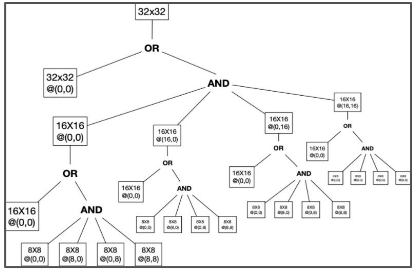
Compression Abstractions

Data Representation

- Goals
  - Provide multidimensional data structures
  - Capture spatial information
  - Provide intuitive data access
  - Support elementwise and reduction operations
- Our abstractions
  - `Block`, `View`, `Stream`, `BitStream`

Data Partitioning

- Goals
  - Capture partition options in single tree structure
  - Provide high-level syntax for creating the tree
  - Support tree iteration
  - Preserve spatial information
- Our abstractions
  - `PTree, AndTree`

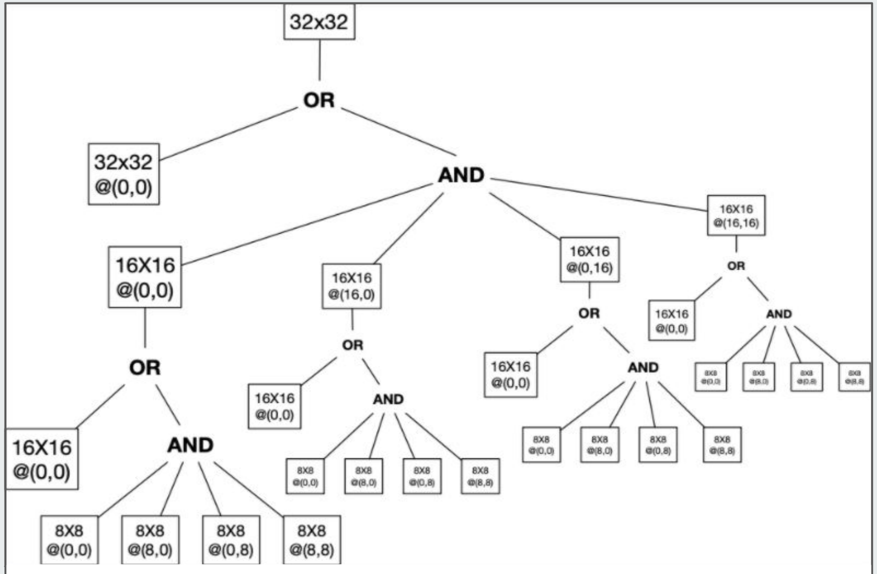


Data Traversals

- Goals
  - Capture temporal dependencies that imply data orderings
  - Provide high-level syntax for representing dependencies
  - Support iteration with the orderings
- Our abstractions
  - `UnitTraversal`

Data Partitioning

- Abstraction goals
    - Provide tree data structure that compactly represents all the possible ways to partition a region into sub-regions
    - Provide compact syntax for creating the tree
    - Support iteration through all sub-regions
        - Relates to data traversal abstraction described later
    - Preserve all spatial information needed for data representation
- `PTree` abstraction
    - An And-Or tree representing the partition options
        - And node: defines one way to partition a region
        - Or node: defines options for partitioning a region
    - Iteration returns each unique partition as an `AndTree`
- `AndTree` abstraction
    - Represents one unique partition of the `PTree` root
    - Iteration returns the leaf sub-regions
- Existing implementations
    - May not explicitly express the partition
    - Wrapped into index calculations, utilize lookup tables
        - Inflexible!



`PTree` visualization for partitioning a 32x32 CTU in CUs.
@(x,y) represents the origin of a region relative to the parent region.

CU and PU Partitions in HEVC/H.265

```
# define PTree for CTU->CU
def CTU_to_CUs(ctu):
  M = ctu.dims[0]
  ptree = PTree()
  pt_root(ctu,ptree):
    pt_or:
      pt_leaf(ctu)
      if M > 8:
        pt_and:
          q0 = ctu[:M/2,:M/2]
          q1 = ctu[M/2:,:M/2]
          q2 = ctu[:M/2,M/2:]
          q3 = ctu[M/2:,M/2:]
          pt_leaf(CTU_to_CUs(q0))
          pt_leaf(CTU_to_CUs(q1))
          pt_leaf(CTU_to_CUs(q2))
          pt_leaf(CTU_to_CUs(q3))
  return ptree

CTU = …
# partition CTU into CUs
cu_ptree = CTU_to_CUs(CTU)
# iterate through each option (AndTree)
for cu_andtree in scan(cu_ptree):
  # iterate through each CU
  for CU in scan(cu_andtree):
    # partition CU into PUs
    pu_tree = CU_to_PUs(CU)
    # iterate through each PU
    for PU in scan(pu_tree):
      # get the data to the above and right
      # using location relative to the PU
      above_right = data[PU[-1,right_range]]
```

```
# define PTree for CU->PU
def CU_to_PUs(cu):
  M = cu.dims[0]
  ptree = PTree()
  pt_root(cu,ptree):
    pt_or:
      pt_leaf(cu)
      pt_and:
        pt_leaf(cu[:,:M/2])
        pt_leaf(cu[:,M/2:])
      pt_and:
        pt_leaf(cu[:M/2,:])
        pt_leaf(cu[M/2:,:])
      if M == min_CU_size:
        pt_and:
          pt_leaf(cu[:M/2,:M/2])
          pt_leaf(cu[:M/2,M/2:])
          pt_leaf(cu[M/2:,:M/2])
          pt_leaf(cu[M/2:,M/2:])
      if M > 8:
        pt_and:
          pt_leaf(cu[:,:M/4])
          pt_leaf(cu[:,M/4:])
        pt_and:
          pt_leaf(cu[:,:3M/4])
          pt_leaf(cu[:,3M/4:])
        pt_and:
          pt_leaf(cu[:M/4,:])
          pt_leaf(cu[M/4:,:])
        pt_and:
          pt_leaf(cu[:3M/4,:])
          pt_leaf(cu[3M/4:,:])
  return ptree
```

With our abstractions: high-level and compact recursive notation, easy to modify tree, straightforward iteration

```
m_cuAbove = (m_cuAddr >= widthInCU) && !m_bFirstRowInSlice ?
m_encData->getPicCTU(m_cuAddr - widthInCU) : NULL;
m_cuAboveRight = (m_cuAbove && ((m_cuAddr % widthInCU) < (widthInCU - 1))) ?
m_encData->getPicCTU(m_cuAddr - widthInCU + 1) : NULL;

const CUData* CUData::getPUAboveRight(uint32_t& arPartUnitIdx, uint32_t
curPartUnitIdx) const {
 if ((m_encData->getPicCTU(m_cuAddr)->m_cuPelX + g_zscanToPelX[curPartUnitIdx] +
UNIT_SIZE) >= m_slice->m_sps->picWidthInLumaSamples)
   return NULL;
 uint32_t absPartIdxRT = g_zscanToRaster[curPartUnitIdx];
 if (lessThanCol(absPartIdxRT, s_numPartInCUSize - 1)) {
   if (!isZeroRow(absPartIdxRT)) {
     if (curPartUnitIdx > g_rasterToZscan[absPartIdxRT - RASTER_SIZE + 1]) {
       uint32_t absZorderCUIdx = g_zscanToRaster[m_absIdxInCTU] + (1 <<
(m_log2CUSize[0] - LOG2_UNIT_SIZE)) - 1;
       arPartUnitIdx = g_rasterToZscan[absPartIdxRT - RASTER_SIZE + 1];
       if (isEqualRowOrCol(absPartIdxRT, absZorderCUIdx))
         return m_encData->getPicCTU(m_cuAddr);
       else {
         arPartUnitIdx -= m_absIdxInCTU;
         return this;
       }
     }
     return NULL;
   }
   arPartUnitIdx = g_rasterToZscan[absPartIdxRT + ((s_numPartInCUSize - 1) <<
LOG2_RASTER_SIZE) + 1];
   return m_cuAbove;
 }
 if (!isZeroRow(absPartIdxRT))
   return NULL;
 arPartUnitIdx = g_rasterToZscan[(s_numPartInCUSize - 1) << LOG2_RASTER_SIZE];
 return m_cuAboveRight;
}
inline bool isEqualCol(int addrA, int addrB) {
    return ((addrA ^ addrB) & (RASTER_SIZE - 1)) == 0;
}

inline bool isEqualRow(int addrA, int addrB) {
    return ((addrA ^ addrB) < RASTER_SIZE);
}

inline bool isEqualRowOrCol(int addrA, int addrB) {
    return isEqualCol(addrA, addrB) | isEqualRow(addrA, addrB);
}

inline bool isZeroRow(int addr) {
    return (addr < RASTER_SIZE);
}

inline bool lessThanCol(int addr, int val) {
    return (addr & (RASTER_SIZE - 1)) < val;
}
```

```
const uint32_t g_zscanToRaster[MAX_NUM_PARTITIONS] =
{
    0x00, 0x01, 0x10, 0x11, 0x02, 0x03, 0x12, 0x13, 0x20, 0x21, 0x30, 0x31, 0x22, 0x23, 0x32, 0x33,
    0x04, 0x05, 0x14, 0x15, 0x06, 0x07, 0x16, 0x17, 0x24, 0x25, 0x34, 0x35, 0x26, 0x27, 0x36, 0x37,
    0x40, 0x41, 0x50, 0x51, 0x42, 0x43, 0x52, 0x53, 0x60, 0x61, 0x70, 0x71, 0x62, 0x63, 0x72, 0x73,
    0x44, 0x45, 0x54, 0x55, 0x46, 0x47, 0x56, 0x57, 0x64, 0x65, 0x74, 0x75, 0x66, 0x67, 0x76, 0x77,
    0x08, 0x09, 0x18, 0x19, 0x0A, 0x0B, 0x1A, 0x1B, 0x28, 0x29, 0x38, 0x39, 0x2A, 0x2B, 0x3A, 0x3B,
    0x0C, 0x0D, 0x1C, 0x1D, 0x0E, 0x0F, 0x1E, 0x1F, 0x2C, 0x2D, 0x3C, 0x3D, 0x2E, 0x2F, 0x3E, 0x3F,
    0x48, 0x49, 0x58, 0x59, 0x4A, 0x4B, 0x5A, 0x5B, 0x68, 0x69, 0x78, 0x79, 0x6A, 0x6B, 0x7A, 0x7B,
    0x4C, 0x4D, 0x5C, 0x5D, 0x4E, 0x4F, 0x5E, 0x5F, 0x6C, 0x6D, 0x7C, 0x7D, 0x6E, 0x6F, 0x7E, 0x7F,
    0x80, 0x81, 0x90, 0x91, 0x82, 0x83, 0x92, 0x93, 0xA0, 0xA1, 0xB0, 0xB1, 0xA2, 0xA3, 0xB2, 0xB3,
    0x84, 0x85, 0x94, 0x95, 0x86, 0x87, 0x96, 0x97, 0xA4, 0xA5, 0xB4, 0xB5, 0xA6, 0xA7, 0xB6, 0xB7,
    0xC0, 0xC1, 0xD0, 0xD1, 0xC2, 0xC3, 0xD2, 0xD3, 0xE0, 0xE1, 0xF0, 0xF1, 0xE2, 0xE3, 0xF2, 0xF3,
    0xC4, 0xC5, 0xD4, 0xD5, 0xC6, 0xC7, 0xD6, 0xD7, 0xE4, 0xE5, 0xF4, 0xF5, 0xE6, 0xE7, 0xF6, 0xF7,
    0x88, 0x89, 0x98, 0x99, 0x8A, 0x8B, 0x9A, 0x9B, 0xA8, 0xA9, 0xB8, 0xB9, 0xAA, 0xAB, 0xBA, 0xBB,
    0x8C, 0x8D, 0x9C, 0x9D, 0x8E, 0x8F, 0x9E, 0x9F, 0xAC, 0xAD, 0xBC, 0xBD, 0xAE, 0xAF, 0xBE, 0xBF,
    0xC8, 0xC9, 0xD8, 0xD9, 0xCA, 0xCB, 0xDA, 0xDB, 0xE8, 0xE9, 0xF8, 0xF9, 0xEA, 0xEB, 0xFA, 0xFB,
    0xCC, 0xCD, 0xDC, 0xDD, 0xCE, 0xCF, 0xDE, 0xDF, 0xEC, 0xED, 0xFC, 0xFD, 0xEE, 0xEF, 0xFE, 0xFF
};
const uint32_t g_rasterToZscan[MAX_NUM_PARTITIONS] =
{
    0x00, 0x01, 0x04, 0x05, 0x10, 0x11, 0x14, 0x15, 0x40, 0x41, 0x44, 0x45, 0x50, 0x51, 0x54, 0x55,
    0x02, 0x03, 0x06, 0x07, 0x12, 0x13, 0x16, 0x17, 0x42, 0x43, 0x46, 0x47, 0x52, 0x53, 0x56, 0x57,
    0x08, 0x09, 0x0C, 0x0D, 0x18, 0x19, 0x1C, 0x1D, 0x48, 0x49, 0x4C, 0x4D, 0x58, 0x59, 0x5C, 0x5D,
    0x0A, 0x0B, 0x0E, 0x0F, 0x1A, 0x1B, 0x1E, 0x1F, 0x4A, 0x4B, 0x4E, 0x4F, 0x5A, 0x5B, 0x5E, 0x5F,
    0x20, 0x21, 0x24, 0x25, 0x30, 0x31, 0x34, 0x35, 0x60, 0x61, 0x64, 0x65, 0x70, 0x71, 0x74, 0x75,
    0x22, 0x23, 0x26, 0x27, 0x32, 0x33, 0x36, 0x37, 0x62, 0x63, 0x66, 0x67, 0x72, 0x73, 0x76, 0x77,
    0x28, 0x29, 0x2C, 0x2D, 0x38, 0x39, 0x3C, 0x3D, 0x68, 0x69, 0x6C, 0x6D, 0x78, 0x79, 0x7C, 0x7D,
    0x2A, 0x2B, 0x2E, 0x2F, 0x3A, 0x3B, 0x3E, 0x3F, 0x6A, 0x6B, 0x6E, 0x6F, 0x7A, 0x7B, 0x7E, 0x7F,
    0x80, 0x81, 0x84, 0x85, 0x90, 0x91, 0x94, 0x95, 0xC0, 0xC1, 0xC4, 0xC5, 0xD0, 0xD1, 0xD4, 0xD5,
    0x82, 0x83, 0x86, 0x87, 0x92, 0x93, 0x96, 0x97, 0xC2, 0xC3, 0xC6, 0xC7, 0xD2, 0xD3, 0xD6, 0xD7,
    0x88, 0x89, 0x8C, 0x8D, 0x98, 0x99, 0x9C, 0x9D, 0xC8, 0xC9, 0xCC, 0xCD, 0xD8, 0xD9, 0xDC, 0xDD,
    0x8A, 0x8B, 0x8E, 0x8F, 0x9A, 0x9B, 0x9E, 0x9F, 0xCA, 0xCB, 0xCE, 0xCF, 0xDA, 0xDB, 0xDE, 0xDF,
    0xA0, 0xA1, 0xA4, 0xA5, 0xB0, 0xB1, 0xB4, 0xB5, 0xE0, 0xE1, 0xE4, 0xE5, 0xF0, 0xF1, 0xF4, 0xF5,
    0xA2, 0xA3, 0xA6, 0xA7, 0xB2, 0xB3, 0xB6, 0xB7, 0xE2, 0xE3, 0xE6, 0xE7, 0xF2, 0xF3, 0xF6, 0xF7,
    0xA8, 0xA9, 0xAC, 0xAD, 0xB8, 0xB9, 0xBC, 0xBD, 0xE8, 0xE9, 0xEC, 0xED, 0xF8, 0xF9, 0xFC, 0xFD,
    0xAA, 0xAB, 0xAE, 0xAF, 0xBA, 0xBB, 0xBE, 0xBF, 0xEA, 0xEB, 0xEE, 0xEF, 0xFA, 0xFB, 0xFE, 0xFF
};
const uint8_t g_zscanToPelX[MAX_NUM_PARTITIONS] =
{
    0, 4, 0, 4, 8, 12, 8, 12, 0, 4, 0, 4, 8, 12, 8, 12,
    16, 20, 16, 20, 24, 28, 24, 28, 16, 20, 16, 20, 24, 28, 24, 28,
    0, 4, 0, 4, 8, 12, 8, 12, 0, 4, 0, 4, 8, 12, 8, 12,
    16, 20, 16, 20, 24, 28, 24, 28, 16, 20, 16, 20, 24, 28, 24, 28,
    32, 36, 32, 36, 40, 44, 40, 44, 32, 36, 32, 36, 40, 44, 40, 44,
    48, 52, 48, 52, 56, 60, 56, 60, 48, 52, 48, 52, 56, 60, 56, 60,
    32, 36, 32, 36, 40, 44, 40, 44, 32, 36, 32, 36, 40, 44, 40, 44,
    48, 52, 48, 52, 56, 60, 56, 60, 48, 52, 48, 52, 56, 60, 56, 60,
    0, 4, 0, 4, 8, 12, 8, 12, 0, 4, 0, 4, 8, 12, 8, 12,
    16, 20, 16, 20, 24, 28, 24, 28, 16, 20, 16, 20, 24, 28, 24, 28,
    0, 4, 0, 4, 8, 12, 8, 12, 0, 4, 0, 4, 8, 12, 8, 12,
    16, 20, 16, 20, 24, 28, 24, 28, 16, 20, 16, 20, 24, 28, 24, 28,
    32, 36, 32, 36, 40, 44, 40, 44, 32, 36, 32, 36, 40, 44, 40, 44,
    48, 52, 48, 52, 56, 60, 56, 60, 48, 52, 48, 52, 56, 60, 56, 60,
    32, 36, 32, 36, 40, 44, 40, 44, 32, 36, 32, 36, 40, 44, 40, 44,
    48, 52, 48, 52, 56, 60, 56, 60, 48, 52, 48, 52, 56, 60, 56, 60
};
```

x265 implementation: imperative with lots of conditionals determining location, iteration order manually specified in lookup tables

Compression Abstractions

Data Representation

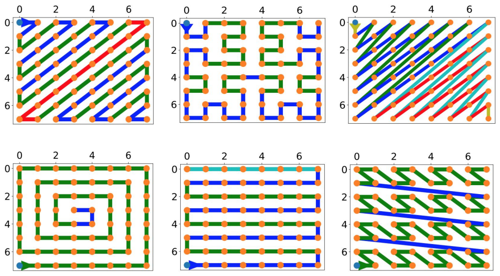
- Goals
  - Provide multidimensional data structures
  - Capture spatial information
  - Provide intuitive data access
  - Support elementwise and reduction operations
- Our abstractions
  - `Block`, `View`, `Stream`, `BitStream`

Data Partitioning

- Goals
  - Capture partition options in single tree structure
  - Provide high-level syntax for creating the tree
  - Support tree iteration
  - Preserve spatial information
- Our abstractions
  - `PTree, AndTree`

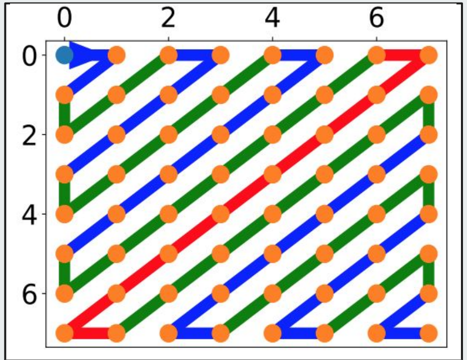
Data Traversals

- Goals
  - Capture temporal dependencies that imply data orderings
  - Provide high-level syntax for representing dependencies
  - Support iteration with the orderings
- Our abstractions
  - `UnitTraversal`

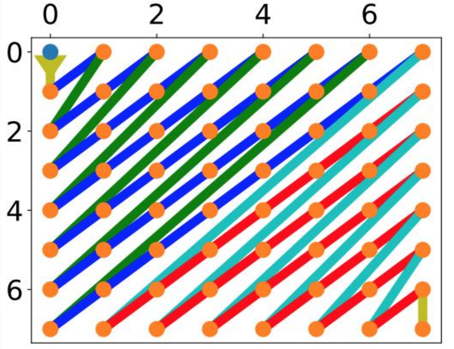


Data Traversal

- Abstraction goals
  - Capture the intricate temporal dependencies that define a data ordering
    - Orderings can be within or across multidimensional data regions
  - Provide syntax for succinctly representing these temporal dependencies
  - Support iteration across `Block`s, `View`s, `Stream`s, and `BitStream`s using a specified traversal
- `AndTree` abstraction
  - Same `AndTree` from data partitioning
  - The order of the leaves imposes an iteration order across regions
  - Changing the traversal order just requires changing the leaves
- `UnitTraversal` abstraction
  - Defines an iteration order within a region
  - We utilize a recursive notation based on parametric Lindenmayer systems[1] to compactly define the order
- Existing implementations
  - Manually enumerate each coordinate or require loop nests with many conditionals
    - Manual enumerations introduce indirect data accesses
  - Lack a common syntax



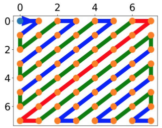
Visualization of 8x8 zigzag traversal



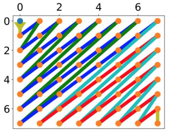
Visualization of 8x8 diagonal traversal

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

Zigzag and Diagonal Traversals

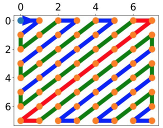



```
N = ...region size...
zz = UnitTraversal(blk)
zz.axiom([A(1)])
zz.rule(A(i),[ST,135,ST*i,315,B(i+1),45,ST*i,225,ST],i<N)
zz.rule(B(i),[ST,225,ST*i,45,A(i+1),315,ST*i,135,ST],i<N)
zz.rule(A(i),[ST,135,ST*(size-1),225,ST],i==N)
zz.rule(B(i),[ST,225,ST*(size-1),135,ST],i==N)
```

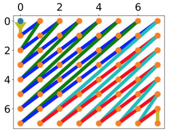



```
N = ...region size...
diagonal = UnitTraversal(blk)
diagonal.axiom([90,ST,A(1,N-1),135,ST])
diagonal.rule(A(i,j),[225,ST*i,B(i+1,j)],i<N)
diagonal.rule(B(i,j),[225,SK*(i-1),270,SK*i,A(i,j)],i<N)
diagonal.rule(A(i,j),[225,ST*(j-1),B(i,j-1)],(i==N)&(j>1))
diagonal.rule(B(i,j),[225,SK*(j-1),270,SK*j,A(i,j)],(i==N)&(j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

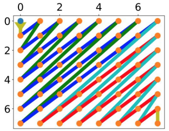
With our abstraction: high-level recursive notation, defines traversal as series of steps and rotations, easy to modify




```
// IJG implementation
const int jpeg_natural_order[DCTSIZE2+16] = {
0, 1, 8, 16, 9, 2, 3, 10,
17, 24, 32, 25, 18, 11, 4, 5,
12, 19, 26, 33, 40, 48, 41, 34,
27, 20, 13, 6, 7, 14, 21, 28,
35, 42, 49, 56, 57, 50, 43, 36,
29, 22, 15, 23, 30, 37, 44, 51,
58, 59, 52, 45, 38, 31, 39, 46,
53, 60, 61, 54, 47, 55, 62, 63};
```




```
// HM implementation
if ((m_column == (m_blockWidth - 1)) || (m_line == 0)) {
  m_line    += m_column + 1;
  m_column   = 0;
  if (m_line >= m_blockHeight) {
    m_column += m_line - (m_blockHeight - 1);
    m_line    = m_blockHeight - 1;
  }
} else {
  m_column++;
  m_line--;
}
```




```
// x265 implementation
const uint16_t g_scan8x8diag[8 * 8] =
{
    0,  8,  1, 16,  9,  2, 24, 17,
    10,  3, 32, 25, 18, 11,  4, 40,
    33, 26, 19, 12,  5, 48, 41, 34,
    27, 20, 13,  6, 56, 49, 42, 35,
    28, 21, 14,  7, 57, 50, 43, 36,
    29, 22, 15, 58, 51, 44, 37, 30,
    23, 59, 52, 45, 38, 31, 60, 53,
    46, 39, 61, 54, 47, 62, 55, 63
};

const uint16_t g_scan16x16[16 * 16] =
{
    0, 16, 1, 32, 17, 2, 48, 33, 18, 3, 49, 34, 19, 50, 35, 51,
    64, 80, 65, 96, 81, 66, 112, 97, 82, 67, 113, 98, 83, 114, 99, 115,
    4, 20, 5, 36, 21, 6, 52, 37, 22, 7, 53, 38, 23, 54, 39, 55,
    128, 144, 129, 160, 145, 130, 176, 161, 146, 131, 177, 162, 147, 178, 163, 179,
    68, 84, 69, 100, 85, 70, 116, 101, 86, 71, 117, 102, 87, 118, 103, 119,
    8, 24, 9, 40, 25, 10, 56, 41, 26, 11, 57, 42, 27, 58, 43, 59,
    192,208, 193,224,209, 194,240,225,210, 195,241,226,211,242,227,243,
    132, 148, 133, 164, 149, 134, 180, 165, 150, 135, 181, 166, 151, 182, 167, 183,
    72, 88, 73, 104, 89, 74, 120, 105, 90, 75, 121, 106, 91, 122, 107, 123,
    12, 28, 13, 44, 29, 14, 60, 45, 30, 15, 61, 46, 31, 62, 47, 63,
    196,212, 197,228,213, 198,244,229,214, 199,245,230,215,246,231,247,
    136, 152, 137, 168, 153, 138, 184, 169, 154, 139, 185, 170, 155, 186, 171, 187,
    76, 92, 77, 108, 93, 78, 124, 109, 94, 79, 125, 110, 95, 126, 111, 127,
    200,216,201,232,217,202,248,233,218,203,249,234,219,250,235,251,
    140, 156, 141, 172, 157, 142, 188, 173, 158, 143, 189, 174, 159, 190, 175, 191,
    204,220,205,236,221,206,252,237,222,207,253,238,223,254,239,255
};
```

```
const uint16_t g_scan32x32[32 * 32] =
{
    0,32,1,64,33,2,96,65,34,3,97,66,35,98,67,99,128,160,129,192,161,130,224,193,162,131,225,194,163,226,195,227,
    4,36,5,68,37,6,100,69,38,7,101,70,39,102,71,103,256,288,257,320,289,258,352,321,290,259,353,322,291,354,323,355,
    132,164,133,196,165,134,228,197,166,135,229,198,167,230,199,231,8,40,9,72,41,10,104,73,42,11,105,74,43,106,75,107,
    384,416,385,448,417,386,480,449,418,387,481,450,419,482,451,483,260,292,261,324,293,262,356,325,294,263,357,326,295,358,327,359,
    136,168,137,200,169,138,232,201,170,139,233,202,171,234,203,235,12,44,13,76,45,14,108,77,46,15,109,78,47,110,79,111,
    512,544,513,576,545,514,608,577,546,515,609,578,547,610,579,611,388,420,389,452,421,390,484,453,422,391,485,454,423,486,455,487,
    264,296,265,328,297,266,360,329,298,267,361,330,299,362,331,363,140,172,141,204,173,142,236,205,174,143,237,206,175,238,207,239,
    16,48,17,80,49,18,112,81,50,19,113,82,51,114,83,115,640,672,641,704,673,642,736,705,674,643,737,706,675,738,707,739,
    516,548,517,580,549,518,612,581,550,519,613,582,551,614,583,615,392,424,393,456,425,394,488,457,426,395,489,458,427,490,459,491,
    268,300,269,332,301,270,364,333,302,271,365,334,303,366,335,367,144,176,145,208,177,146,240,209,178,147,241,210,179,242,211,243,
    20,52,21,84,53,22,116,85,54,23,117,86,55,118,87,119,768,800,769,832,801,770,864,833,802,771,865,834,803,866,835,867,
    644,676,645,708,677,646,740,709,678,647,741,710,679,742,711,743,520,552,521,584,553,522,616,585,554,523,617,586,555,618,587,619,
    396,428,397,460,429,398,492,461,430,399,493,462,431,494,463,495,272,304,273,336,305,274,368,337,306,275,369,338,307,370,339,371,
    148,180,149,212,181,150,244,213,182,151,245,214,183,246,215,247,24,56,25,88,57,26,120,89,58,27,121,90,59,122,91,123,
    896,928,897,960,929,898,992,961,930,899,993,962,931,994,963,995,772,804,773,836,805,774,868,837,806,775,869,838,807,870,839,871,
    648,680,649,712,681,650,744,713,682,651,745,714,683,746,715,747,524,556,525,588,557,526,620,589,558,527,621,590,559,622,591,623,
    400,432,401,464,433,402,496,465,434,403,497,466,435,498,467,499,276,308,277,340,309,278,372,341,310,279,373,342,311,374,343,375,
    152,184,153,216,185,154,248,217,186,155,249,218,187,250,219,251,28,60,29,92,61,30,124,93,62,31,125,94,63,126,95,127,
    900,932,901,964,933,902,996,965,934,903,997,966,935,998,967,999,776,808,777,840,809,778,872,841,810,779,873,842,811,874,843,875,
    652,684,653,716,685,654,748,717,686,655,749,718,687,750,719,751,528,560,529,592,561,530,624,593,562,531,625,594,563,626,595,627,
    404,436,405,468,437,406,500,469,438,407,501,470,439,502,471,503,280,312,281,344,313,282,376,345,314,283,377,346,315,378,347,379,
    156,188,157,220,189,158,252,221,190,159,253,222,191,254,223,255,904,936,905,968,937,906,1000,969,938,907,1001,970,939,1002,971,1003,
    780,812,781,844,813,782,876,845,814,783,877,846,815,878,847,879,656,688,657,720,689,658,752,721,690,659,753,722,691,754,723,755,
    532,564,533,596,565,534,628,597,566,535,629,598,567,630,599,631,408,440,409,472,441,410,504,473,442,411,505,474,443,506,475,507,
    284,316,285,348,317,286,380,349,318,287,381,350,319,382,351,383,908,940,909,972,941,910,1004,973,942,911,1005,974,943,1006,975,1007,
    784,816,785,848,817,786,880,849,818,787,881,850,819,882,851,883,660,692,661,724,693,662,756,725,694,663,757,726,695,758,727,759,
    536,568,537,600,569,538,632,601,570,539,633,602,571,634,603,635,412,444,413,476,445,414,508,477,446,415,509,478,447,510,479,511,
    912,944,913,976,945,914,1008,977,946,915,1009,978,947,1010,979,1011,788,820,789,852,821,790,884,853,822,791,885,854,823,886,855,887,
    664,696,665,728,697,666,760,729,698,667,761,730,699,762,731,763,540,572,541,604,573,542,636,605,574,543,637,606,575,638,607,639,
    916,948,917,980,949,918,1012,981,950,919,1013,982,951,1014,983,1015,792,824,793,856,825,794,888,857,826,795,889,858,827,890,859,891,
    668,700,669,732,701,670,764,733,702,671,765,734,703,766,735,767,920,952,921,984,953,922,1016,985,954,923,1017,986,955,1018,987,1019,
    796,828,797,860,829,798,892,861,830,799,893,862,831,894,863,895,924,956,925,988,957,926,1020,989,958,927,1021,990,959,1022,991,1023
};
```

IJG, HM, x265 implementations: manual enumerations (for each block size) or conditional blocks

Looking Ahead: DSL Compiler Support

- **An optimizing compiler for a DSL can automatically generate high-performance code**
    - Optimization by hand is hard, especially when there are multiple possible optimizations that can be combined
    - The DSL compiler is designed around the set of abstractions
        - Allows it to perform optimizations beyond what you would get with a general compiler
- **Three categories of optimizations we are interested in: parallelization, vectorization, speculation**
    1. Parallelization
        - Wavefront, frame/tile/slice, mode selection, etc
    2. Vectorization
        - Generate vector instructions for expensive computation kernels such as transforms
            - Provide support for both SIMD and non-SIMD vectorization
        - Reduce the reliance on hand-coded vector instructions
    3. Speculation
        - Relieve sequential bottlenecks by speculatively performing later computations

Put the Focus on the Algorithm!

- Experts should be able to focus on the compression and not have to worry about implementation details
- Benefits of a DSL
  - More intuitive programming interface
  - Primitive data structures, control flow, operations, etc. that capture the structure of compression algorithms
  - Automatic generation of high-performance code
- Primary goals
  1. (This work) Develop domain-specific abstractions for block-based compression
     - The abstractions are the main building blocks of DSLs
     - Focusing on image and video: JPEG, WebP, AVC/H.264, HEVC/H.265, VVC/H.266
  2. Develop an optimizing compiler based around the abstractions
  3. Extend the abstractions for other categories of compression
- **Faced your own compression-related implementation challenges?**
  - We'd love to hear from you!
- **Contact Jessica Ray: jray@csail.mit.edu**
- **Comparisons with existing frameworks: http://jray.mit.edu/block-based-compression**