A Universal Tensor Abstraction and its Application to and Implementation within Block-Based Compression

by

Jessica Morgan Ray

B.S., University of Massachusetts Amherst (2012)
S.M., Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

September 2023

Authored by: Jessica Morgan Ray
Department of Electrical Engineering and Computer Science
June 15, 2023

Certified by: Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

A Universal Tensor Abstraction and its Application to and Implementation within Block-Based Compression

by

Jessica Morgan Ray

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy

ABSTRACT

Data with spatial relationships are often represented in modern programs using multidimensional arrays or other tensor structures, along with the associated metadata necessary to track the location of each piece of data. For example, a program can represent a video as a multidimensional array and a frame within the video as another multidimensional array with a timestamp that gives the location relative to the start of the video. Current programs cannot natively associate this location-based metadata with the arrays, causing the burden of tracking location to fall on to the user. It quickly becomes an arduous task within domains that have numerous arrays with spatial relationships spread across them. The task becomes further complicated when domains have multiple ways to represent the data, such as using projections, permutations, refinement, and coarsening. One such domain, block-based compression, has this type of heterogeneous, spatial data all throughout it, leading to overly complex implementations.

Block-based compression forms the core of many common image and video standards such as JPEG, H.264, H.265, and H.266. The fundamental data unit in block-based compression, the block, represents everything from a video down to an individual pixel, all of which need to maintain their location relative to other blocks in a program. Due to the lack of support for this spatial data, each implementation largely starts from scratch, leading to inconsistencies in data representation and data access.

This dissertation provides a critical look at the association between location and tensors, and defines a core abstraction called the Universal Tensor abstraction (UniTe). UniTe mathematically describes what it means to associate tensors with location and quantify spatial relationships across multiple tensors in a single program.

While UniTe itself is not tied to a particular domain, this dissertation also provides a practical look at implementing UniTe in the context of block-based compression. An initial library implementation highlights the overhead incurred from UniTe due to computing spatial relationships and underlying array indices at the innermost level of computations.

To combat this overhead, this dissertation also presents two different domain-specific languages, CoLa (Compression Language) and SHiM (Staged Hierarchical Multidimensional arrays), and their accompanying compilers built around UniTe. CoLa and SHiM show that it is possible to remove the overhead and achieve performance parity with hand-implemented C code, while also providing users with an intuitive way to represent and utilize spatial data.

Thesis supervisor: Saman Amarasinghe

Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

First and foremost, I thank my advisor, Saman Amarasinghe. Saman was my advisor for my entire time in grad school, and I could not have asked for a better advisor to guide me through the rigors of a PhD. I remember sitting in Saman's office before I started at MIT and bluntly asking him if I could be his student, to which he quickly responded yes, even though I had next to no background in compilers. He continually supported me throughout and pushed me to be a better researcher, and even though I definitely became frustrated at times, I look back fondly at our conversations (and arguments!) that led me to where I am today. I may not have gone down the academic route that he originally encouraged me to follow (sorry!), but he helped me discover my true interests in the field. It's hard to believe my time as a grad student is coming to an end, but I look forward to continuing to collaborate with Saman in the future.

Many thanks to my committee members Vivienne Sze and Albert Reuther as well. I was fortunate to have my committee in place several years before I actually defended, and this work would not have been possible without their continued support. During the times when my research wasn't going the way I had hoped, they were there to offer helpful advice and assure me that the work I was doing was useful and exciting, even if there were some roadblocks along the way. I would also like to thank the members and collaborators of the COMMIT group who have helped me in so many ways throughout the years. This work would not have been possible without their help. Particular shout-outs go to the developers of Codon, Ariya Shajii, Ibrahim Numanagić, and Gabriel Ramirez, as well as the developer of BuildIt, Ajay Brahmakshatriya. Special thanks as well to Teodoro Collin who helped guide the description of the UniTe framework.

I also want to thank Janet Fischer, Leslie Kolodziejski and the entire grad office for their support throughout my time here. I have many fond memories of events run in conjunction with the grad office (sunset cruises, EECS visit days and orientation, softball), and I always enjoyed just stopping by to say "hi!". Thanks also to Mary McDavitt for helping with all the administrative details that come with being a grad student, and just being a friend. I'll miss chatting and having lunch with Mary (and the other admins!), and I look forward to stopping by in the future to join in another lunch time.

I was also fortunate to have several great sources of funding throughout my time at MIT. My first year was covered by the Irwin Mark Jacobs and Joan Klein Jacobs MIT Presidential Fellowship and my second year was funded by Toyota Research. The remainder of my time was funded by Lincoln Laboratory, with the majority of the funding coming from the Lincoln Scholars program, which allowed me to remain affiliated with Lincoln Laboratory while completing my studies on campus.

Throughout my time at MIT, I was also affiliated with Lincoln Laboratory, which is where I worked before beginning grad school. In my senior year of undergrad, I remember telling Prof. Hanna Wallach (who I was doing an independent study with at the time) that I was interested in working at Lincoln, and she immediately reached out to people she knew there, which eventually led me to an interview with Wade Shen. Wade was a staff member in the Human Language Technology group at the time and took me under his wing and made me realize I loved doing research, which lead to me eventually applying for grad school. Both of them were the ones who got the ball rolling for me professionally and academically, and I will be forever grateful for that.

While my affiliation with Lincoln has been, for lack of a better phrase, all over the place (I was a full-time employee, then a student, then a Lincoln Scholar, and everything in between), so many people there have helped me sort everything all out so that I could continue to be a member of Lincoln Laboratory while also completing my PhD. A special shout-out to Doug Stetson and Katie Lannin in my group office for getting me back to campus to defend and hand in my dissertation. It literally would not have been possible without them!

My co-workers at Lincoln Laboratory are some of the best around, and have helped me get to where I am today. I love working at Lincoln Laboratory and have made many friends there over the years, and I look forward to returning full-time. A huge amount of thanks goes to Chad Meiners, who acted as my unofficial mentor when I returned full-time to Lincoln Laboratory during grad school. Chad is always there to lend a hand, or just listen when I need to rant about something.

Whenever Chad walks in my office, I know I'm going to learn something new (or try out a new fountain pen), and I look forward to continuing to push cutting-edge compiler research with Chad at Lincoln.

I've been lucky to make (and continue!) so many great friendships throughout my time at MIT as well. I especially want to thank Whitney Young, Liz Salesky, and Lizzy Godoy who were always there to lend an ear when I needed someone to talk to. And many thanks to Julie Heislein, who I have literally known my entire life, but was also my roommate during my initial years at MIT. Living with a great friend made the transition back to school so much easier (and fun!). I also want to give thanks to all my UMass friends, softball friends, trivia friends, and the friends I made in grad school (I'll miss GSB!). You all helped me keep my sanity throughout all these years.

Finally, I want thank my family. I'm originally from Massachusetts, so am lucky to have had them close to me throughout my whole life. My parents, Kelly and Jack, provided me with unconditional love and support, just like they have always done. Even through the times when I just wanted to say "I quit!", they pushed me to continue because they had confidence in me. Now that I'll have more free time, I'll undoubtedly be calling them constantly to ask for help on the various projects around my house that I've been putting off while finishing up school.

And to my boyfriend, David–it has meant so much to have you by my side over these last (nearly) six years. Even through the times where I was grumpy, overwhelmed, and stressed out of my mind, you were always there to help and listen. Get ready for me to bother you constantly with all the free time I'll have!

To my parents, David, and the rest of my family and friends, I love you guys!

Contents

List of Figures

List of Tables

Introduction 1

Data compression is a ubiquitous component in nearly every application that involves storing, transmitting, or processing large amounts of data. Beginning in the late 1940s with Shannon's seminal work on information theory [1], data compression has continued to advance to meet the needs of fast-growing data and evolving data sources.

While data compression itself is a very broad term, several categories of compression exist within it, each focused on different data sources and different algorithms. For example, dictionary encoders, such as LZ77 [2] and LZ78 [3], maintain a record of previously seen strings in a dictionary structure and aim to replace repeatedly occurring strings with an index into the dictionary. Run-length encoding schemes replace sequences of repeated values with a single value, helping for data such as sparse data which contains many sequences of zeros throughout.

Other categories of compression are collections of related algorithms that together achieve compression. *Block-based compression*, which is a major component of this dissertation, divides input data into blocks and processes each through a series of stages that progressively transform the data into a form more amenable for compression. It then applies a more traditional type of compression, such as Huffman encoding [4]. Block-based compression forms the core of many types of image and video compression, and is an active area of research, with new standards coming roughly every 10 years[1]. These new standards continually improve various facets of compression, such as compression rates and compression quality. They also target new data sources which have different structure and requirements. For example, streaming video data to a laptop should be compressed differently than a video stored on the laptop as it needs to take into account the transmission of data across a network.

With the improvement in compression that comes with these new standards, an unfortunate side effect is the increased complexity of the standards, which in turn increases the complexity of their implementations. Table 1.1 on Page 16 gives the lines of code for several implementations for block-based compression for image and video. At a minimum, tens of thousands of lines of code are required for an implementation[2]. Each time a new standard comes out, implementations start over from scratch and hand-roll their own ad-hoc data structures, which leads to incompatibility between different implementations, even though they fundamentally all follow the same structure[3]. Usually, an application with this amount of complexity, yet shared core structure, becomes an instant target for programming language support through libraries and domain-specific languages. However, there is a conspicuous lack of such support for block-based compression. To the best of our knowledge, there is only a single library, QccPack [5], targeting encoders, and a single framework, RMC [6, 7] targeting decoders. QccPack provides implementations for several kernels used across different types of block-based encoders, but does not provide any abstractions for implementing the kernels themselves. RMC abstracts the overall structure of decoders

1: Typically, new standards begin development when it becomes possible to achieve 2× improvement in compression when compared to the prior standard. This is about 10 years.

2: Compression is one of the few domains that still rely on hand-coded assembly as well!

3: Sometimes a single implementation does not even agree on the data representation.

Table 1.1: Lines of code for various block-based compression implementations that utilize C/C++ (grouped by standard). Note that two different libraries for JPEG with the name libjpeg exist, thus IJG is used to differentiate (libjpeg-turbo is based on IJG libjpeg).

Software	Standard	Type	C/C++	Assembly
JM [8]	H.264	Video	120,000	N/A
x264 [9]	H.264	Video	68,000	37,000
openh264 [10]	H.264	Video	98,000	33,000
HM [11]	H.265	Video	60,000	N/A
x265 [12]	H.265	Video	96,000	179,000
VTM [13]	H.266	Video	134,000	N/A
libvpx [14]	VP8/VP9	Video	326,000	23,000
libwebp [14]	WebP	Image	74,000	N/A
libjpeg [15]	JPEG	Image	37,000	N/A
IJG libjpeg [16]	JPEG	Image	31,000	N/A
libjpeg-turbo [17]	JPEG	Image	57,000	31,000

using a dataflow framework, but like QccPack, does not provide abstractions for implementing the kernels (which would represent the nodes within the dataflow graph).

1.1 Tensors and Spatial Relationships

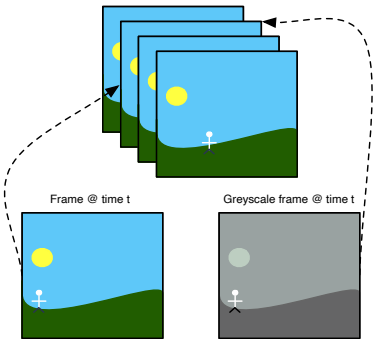


Figure 1.1: Two frames corresponding to time t within a video. Here, the color frame references a location within the memory of the video, while the greyscale frame references the same location, but has its own separate region of memory.

A core part of any programming language support for a domain is the underlying data representation, and block-based compression is no exception. Blocks are the main unit of computation in block-based compression, so a natural data representation is the multidimensional array (e.g. a tensor), originally introduced in Fortran [18]. In block-based compression, this tensor can represent anything from a video down to single pixel within a video. However, these tensors also have *spatial relationships* among them, which requires being able to represent and reason about their *location* with respect to one another.

Consider a simple visual example in Figure 1.1. This example depicts a color video with a frame extracted at time t, which is then converted to a greyscale frame (with the video and both frames being tensors). Both the color frame and greyscale frame point to different memory, but the key is that they both refer to the same location: each represents data that represents a frame at time t. While straightforward in this example, this type of relationship between tensors occurs all throughout the different stages of block-based compression, with numerous tensors created at every level of computation. Current methods are ad-hoc, representing tensors as plain multidimensional arrays and manually tracking all the metadata necessary to capture the location; no library or language provides the necessary semantics to accurately represent this type of data. In isolated, local contexts, these ad-hoc approaches may be sufficient. However, in domains such as block-based compression, issues arise due to both the breadth and depth of spatial relationships across tensors. Breadth refers to the fact that computations often operate on data across many tensors simultaneously, which requires finding the data in each tensor that corresponds to some specific location. Depth means that the tensors and their location propagate throughout implementations, which requires tightly integrating tensors with their location so they can be used at any point within a program.

Consider the top example shown in Figure 1.2 on Page 18, which shows a snippet from the H.264 standard describing an operation called intra-prediction[4]. At a minimum, intra-prediction requires two different tensor representations: one representing a frame of raw pixels and another representing a frame of reconstructed pixels (more on this in Chapter 2). Intra-prediction takes a location within the raw pixel frame and must find the corresponding location within the reconstructed frame. In this particular example, intra-prediction must also access the data in the row above and column to the left of the current location within the reconstructed frame, highlighting the breadth of this operation. Furthermore, the locations tied to the inputs and outputs of this operation propagate through the rest of the pipeline, showing that the tensors and locations exhibit significant depth.

4: This particular mode is called 16x16 plane.

These ad-hoc implementations also lead to inconsistent and unintuitive interfaces for accessing data. The code in the middle of Figure 1.2 shows an example of one of these ad-hoc implementations for the same type of intra-prediction, where the access semantics do not match that of the standard. The bottom shows another implementation using the `SHiM` domain-specific language (introduced in Chapter 9), which provides a nearly one-to-one correspondence with the accesses in the standard. Again, maintaining this type of representation across tens to hundreds of thousands of lines of code quickly becomes unwieldy and opens the door for indexing bugs.

1.2 A Universal Tensor Abstraction

Spatial relationships among tensors also span beyond just block-based compression. For example, adaptive mesh refinement (AMR) requires representing data in a hierarchical structure, tying each layer back to its location in the prior layer. However, AMR also refines the data in each successive layer, adding another level of complexity as individual data points across each layer no longer have a one-to-one correspondence with one another. Another example, geographical information systems (GIS), filters out data points within a grid based on their location, and then computes statistics across all the points sharing the same location[5].

5: Known as *spatial statistics*.

In order to reason about spatial relationships in a program, this dissertation develops a core theory in Chapter 4 called the *Universal Tensor* abstraction (UniTe), which provides well-defined ways to represent tensors with location and compare those locations. UniTe supports multiple different representations of locations with tensors, including the ability to vary the dimensionality, axes orientation, origins, and point density[6]. Across these different representations, UniTe defines locations so that they remain invariant to the variations, thus two tensors, regardless of their representation, can always have their locations compared to one another.

6: Refinement *increases* the point density in a given location, while coarsening *decreases* the density.

This work also introduces a more data-aware extension of UniTe called *UniTe eXtended* (UniTeX) in Chapter 5. UniTeX defines the *block* and *view* tensors, which own and share their data, respectively, modeling how a program could likely implement such a tensor. Unlike existing uses of blocks and views that exist in modern programs (such as with NumPy, Julia, and other array-centric systems),

The values of the prediction samples $pred_L[\,x, y\,]$, with $x, y = 0..15$, are derived by

$$pred_L[\,x, y\,] = Clip1_Y(\,(\,a + b * (\,x - 7\,) + c * (\,y - 7\,) + 16\,) >> 5\,),\ \text{with}\ x, y = 0..15, \tag{8-122}$$

where

$$a = 16 * (\,p[\,-1, 15\,] + p[\,15, -1\,]\,) \tag{8-123}$$

$$b = (\,5 * H + 32\,) >> 6 \tag{8-124}$$

$$c = (\,5 * V + 32\,) >> 6 \tag{8-125}$$

and H and V are specified as

$$H = \sum_{x'=0}^{7} (\,x'+1\,) * (\,p[\,8 + x', -1\,] - p[\,6 - x', -1\,]) \tag{8-126}$$

$$V = \sum_{y'=0}^{7} (\,y'+1\,) * (\,p[\,-1, 8 + y'\,] - p[\,-1, 6 - y'\,]) \tag{8-127}$$

```
static inline void get_i16x16_plane(imgpel **cur_pred, imgpel *PredPel,
                                    int max_imgpel_value) {
  int i, j;
  // plane prediction
  int ih=0, iv=0;
  int ib, ic, iaa;
  imgpel *t_pred = &PredPel[25];
  imgpel *u_pred = &PredPel[8];
  imgpel *b_pred = &PredPel[23];
  for (i = 1; i < 8;++i) {
    ih += i*(*(u_pred + i) - *(u_pred - i));
    iv += i*(*t_pred++ - *b_pred--);
  }
  ih += 8*(*(u_pred + 8) - PredPel[0]);
  iv += 8*(*t_pred++ - PredPel[0]);
  ib = (5 * ih + 32) >> 6;
  ic = (5 * iv + 32) >> 6;
  iaa=16 * (PredPel[16] + PredPel[32]);
  for (j=0;j< MB_BLOCK_SIZE;++j) {
    for (i=0;i< MB_BLOCK_SIZE;++i) {
      cur_pred[j][i] =
        (imgpel) iClip1( max_imgpel_value, ((iaa+(i-7)*ib+(j-7)*ic+16)>>5));
    }
  }
}
```

```
template <typename Pred, typename Ref>
static void get_16x16_plane(Pred &pred, Ref &ref) {
  auto p = ref[pred].vpermute({1,0});
  dint H = 0;
  dint V = 0;
  for (dint q = 0; q < 8; q=q+1) {
    H += (q+1)*(p(8+q,-1) - p(6-q,-1));
    V += (q+1)*(p(-1,8+q) - p(-1,6-q));
  }
  dint a = 16 * (p(-1,15) + p(15,-1));
  dint b = ((dint)(5 * H + 32) >> 6);
  dint c = ((dint)(5 * V + 32) >> 6);
  pred[y][x] = CLIP1Y((a+b*(x-7)+c*(y-7)+16) >> 5);
}
```

Figure 1.2: A comparison between the description of 16x16 plane intra-prediction from the H.264 standard [19] (top), the code from an existing manual implementation [8] (middle), and code from SHiM (bottom). As shown from the standard, this particular operation requires accessing the data in the row above and column to the left of the tensor p, which the standard shows using negative indices. The SHiM code (see Chapter 9) provides an intuitive multidimensional representation of the data, and provides access semantics that support the negative indices of the standard. On the other hand, the manual code requires accessing a linear array, and also uses a mix of array accesses and pointer arithmetic, resulting in code that obscures what data is actually being accessed.

blocks and views in UniTeX both have location[7]. This allows UniTe and UniTeX to define operations that can exploit spatial relationships. For example, Chapter 5 defines an operation called colocation, which provides well-defined semantics for what it means to access data in a tensor at the same location as another tensor. This operation is only possible thanks to the underlying representations of UniTe and UniTeX.

7: Views in other languages have a minimal notion of spatial relationships as they need to know where they point to in their backing array (a block) in order to correctly access data. However, this "location" is typically not accessible to the user, nor is it exploited for any operations based on spatial relationships.

1.2.1 Implementing UniTe and UniTeX

While UniTe and UniTeX provide an intuitive way to represent location-aware tensors, like most abstractions, they suffer from performance overheads if implemented naively. In particular, these abstractions add additional arithmetic overhead when computing the underlying array indices due to the different representations allowed for locations. In block-based compression, these index operations also happen at the innermost levels of the loop nests, causing the overhead to accumulate quickly. In fact, compared to hand-optimized code, a library implementation for UniTeX can run up to 65× slower due to this overhead[8]. The second half of this dissertation focuses on these performance overheads by implementing the abstractions and applying them to block-based compression through the use of two domain-specific languages (DSLs), `CoLa` (Compression Language) and `SHiM` (Staged Hierarchical Multidimensional arrays). `CoLa` (Chapter 8) and `SHiM` (Chapter 9) are able to achieve performance parity through the use of various optimization passes (in `CoLa`) and staging (in `SHiM`). This work explores the reasons for the overhead in the context of compression, and also considers the various design aspects that should be taken into account when designing a DSL for block-based compression.

8: Which is likely why there is a lack of library implementations for block-based compression.

1.3 Contributions

This dissertation provides three categories of contributions focused on the theory, implementation, and application of tensors with spatial relationships. The theory covers how to define an abstraction that captures and makes it possible to reason about spatial relationships across tensors (and arbitrary sets of points). The implementation provides practical approaches on how to remove the runtime overhead associated with the abstraction through the use of domain-specific languages. Finally, the application shows how the abstraction and DSLs can be used to simplify the implementation of various kernels within block-based compression, which heavily utilize spatial tensors throughout. To the best of our knowledge, this dissertation is the first to provide an abstraction and implementation of tensors with spatial relationships, as well as the first to provide language support for block-based compression encoders. In particular, the contributions of this dissertation are:

A mathematical framework, UniTe, that provides programs with the ability to capture and reason about spatial relationships in multidimensional array or tensor data structures. UniTe makes it possible to define the location of tensors across different frames of reference that allow variations in the

dimensionality, axes ordering, origin, refinement, and coarsening of points within the tensors.

A breakdown of tensors into blocks and views, which connects data to the tensors, where blocks correspond to new multidimensional arrays and views reference a portion of an existing block. Unlike traditional uses of views that largely focus on preventing data copies, the views in this dissertation also provide the ability to represent data with location in the different frames of reference defined by UniTe.

Novel tensor operations exploiting relationships that users can utilize to create new tensors from one another and access them. This dissertation defines several such operations and provides particular emphasis on colocation and locality access operations which are only possible due to the underlying structure provided by UniTe.

The DSL `CoLa`, which is a Pythonic DSL embedded in Codon [20] that removes the overhead of UniTe. `CoLa` is able to bring performance from nearly 65× slower than existing hand-optimized C code down to parity through the use of domain-specific compiler passes targeting index computations and view creation.

The DSL `SHiM`, which is a DSL embedded in C++ that removes the overhead of UniTe through the use of staging via the BuildIt [21] library. Like `CoLa`, `SHiM` is able to achieve performance parity with existing hand-optimized C code.

A demonstration of UniTe applied to JPEG and H.264 that shows how tensors with spatial relationships can be utilized to simplify the representation of and access to data in image (JPEG) and video (H.264) compression. JPEG and H.264 also serve as benchmarks for `CoLa` and `SHiM`.

1.4 Dissertation Overview

The rest of this dissertation is structured as follows:

Chapter 2-The Design of Encoders gives a primer on block-based compression, focusing on the stages within exemplar pipelines for JPEG and H.264 encoders and pointing out various parts of stages that can be aided by abstractions.

Chapter 3-Understanding Spatial Relationships provides intuition for reasoning about spatial relationships across tensors and what goes into representing location.

Chapter 4-Formalizing UniTe introduces the core UniTe abstraction, incrementally building up a mathematical framework that captures spatial relationships for both arbitrary sets of points and tensors.

Chapter 5-Extending UniTe: UniTeX expands UniTe with the addition of block and view tensors, which open the door for new tensor creation and access operations.

Chapter 6-Applying UniTeX to JPEG and H.264 provides pseudocode and accompanying descriptions that highlight how to apply the abstractions within several stages from JPEG and H.264 encoders.

Chapter 7-Implementation Considerations motivates the use of DSLs for implementing UniTe and UniTeX, and also discusses various considerations that should be taken into account when designing a DSL for block-based compression.

Chapter 8-`CoLa` introduces the DSL `CoLa` and describes its integration within Codon [20], along with a look at its performance and the optimizations necessary to achieve such performance.

Chapter 9-`SHiM` introduces the DSL `SHiM` and describes its integration within C++ and BuildIt [21], and also provides a look at how it is able to achieve the same optimizations of `CoLa` without the need for explicit compiler passes.

Chapter 10-Related Works highlights the lack of prior work specifically related to language support for block-based compression, while also providing a look at more general purpose array-based languages and briefly discussing the related domains of adaptive mesh refinement and geographic information systems.

Chapter 11-Conclusion and Future Work summarizes the dissertation and provides some possible paths for future work based on the concepts presented.

The Design of Encoders 2

Block-based compression is a relatively broad term that encompasses data compression that partition inputs into smaller blocks of data, and operates on those blocks individually. Specific classes of block-based compression are typically associated with *standards*, and this chapter explores two such standards: JPEG [22] for image compression and H.264 [19] for video compression. Standards describe the operation of the *decoder*, which takes in a compressed bitstream and reproduces either an exact version, or approximation of, the original input. These standards detail the structure that a compressed bitstream must conform to, along with other requirements that must be met. However, they do not specify the operation of the *encoder*, which produces the compressed bitstream. This allows for a great deal of variety and creativity across encoder design, opening the door for tuning the implementation in many ways, but also increasing the complexity of such a system. This work focuses on the encoder side, as encoders encompass most of the operations of the decoder as well.

Despite the variations across encoders, the majority of encoders follow the same high-level structure, composed of a series of stages that either decompose the data into smaller blocks, and/or operate on the data in those blocks. In both the JPEG and H.264 encoders discussed in this chapter, the blocks of data are primarily blocks of pixels, though smaller units, such as partial pixels, are also allowed (see Subsection 2.2.6). Key components of these blocks are the spatial relationships amongst one another. Many operations occur across multiple blocks simultaneously and access data at related locations within each. These blocks and their location propagate through nearly every stage of an encoder, which requires significant amounts of metadata in order to track the locations throughout a program[1]. The main abstraction of this dissertation, UniTe, focuses on providing an intuitive representation for this type of data that simplifies accesses on and across different data blocks, all while maintaining their locations relative to one another. This chapter serves to provide an overview of not just what the stages of the encoder do, but also highlights where the interesting and difficult parts of data representation and data access occur within each stage. The focus will be on the main stages within JPEG and H.264 encoders, which provide exemplars of the operations and overall structure for this class of encoders.

1: One C implementation for H.264 called JM [8] (discussed more throughout the rest of this dissertation) defines nearly 130 different variables for tracking block location in just one header file.

While not all of the stages within these pipelines fit into the abstractions presented here, this chapter still provides a look at these stages because they are useful in regards to implementing the encoders. For example, the last stage of the encoder handles bitstream output, which requires various bit-level operations such as packing bits together. This is outside the scope of this work, but still important for the overall implementation since it is required for correctness, thus is still introduced in this chapter.

Note that use of the term "block" in this chapter is mainly used as a general term encompassing a rectangular region of data as opposed to representing a block versus a view. When the distinction is necessary, it will be explicitly mentioned.

2.1 JPEG

2: A *lossless* encoder produces output that a decoder can use to perfectly reconstruct the original input. A *lossy* encoder produces output that a decoder cannot use to perfectly reconstruct the original input.

The JPEG standard [22] defines several different lossy[2] modes of image compression, along with a single lossless mode, all of which lead to varying degrees of compression and quality (in the lossy case) depending on the data and use case. The pipeline in Figure 2.1 models the lossy *baseline sequential mode*, which is the mode typically referred to when "JPEG" is used without additional qualification. This mode compresses the entire input image in a single pass, going top-to-bottom, left-to-right through the image. Other lossy modes include *progressive mode*, which compresses the image using multiple passes (including a different part of the image in each pass), and *hierarchical mode*, which compresses the same image multiple times using different resolutions. The lossless version takes a similar top-to-bottom, left-to-right approach, but with a different core algorithm. The next sections focus on each stage within baseline JPEG.

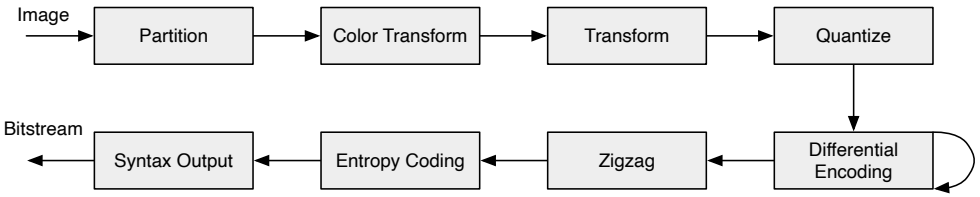


Figure 2.1: A possible encoder pipeline for the baseline sequential mode in JPEG. This highlights the primary stages that should be found in any given implementation for this type of JPEG encoder.

2.1.1 Image Input formats

The raw image data for JPEG typically comes in RGB format, with pixels stored in either an *interleaved* or *planar* fashion. Together, an R, a G, and a B value (each is a *component*) define a single pixel. The interleaved format stores the components of a pixel sequentially (RGBRGB...RGB), while planar mode stores all R components, then G, then B (RRR...RRRGGG...GGGBBB...BBB). Figure 2.2 on Page 25 shows the format of interleaved and planar data, respectively.

Considerations for an abstraction

As conceptually simple as these two representations are, the fact that there can be multiple representations automatically adds complexity to an implementation. In this case, the two different layouts would be indexed differently, thus an implementation that handles both layouts would need separate logic (or potentially separate functions) depending on the layout. This opens the door for indexing bugs, such as incorrectly accessing interleaved data as planar. An abstraction should be able to hide the underlying details of the data format and provide a uniform representation to the user.

2.1.2 Partition

The *partition* stage takes the input and transforms it into the separate planar components, and then splits each plane into 8x8 blocks. The rest of the pipeline

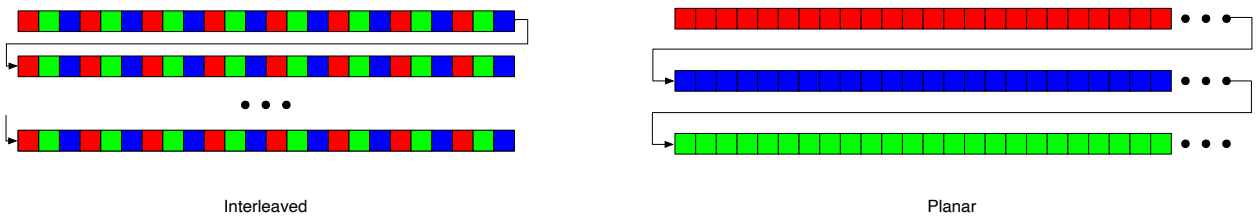


Figure 2.2: Comparing the memory layout of interleaved and planar data formats for JPEG.

operates on these 8x8 blocks, running the same operations on each, and eventually combining the results back together before outputting the final bitstream.

Considerations for an Abstraction

Being able to partition a block of data is a fundamental operation within compression because it allows representing the data in a more intuitive format, thus an abstraction needs to support this. For example, with the 8x8 blocks in JPEG, it is much simpler to access pixel components within those blocks with indices that are relative to the 8x8 block, rather than relative to the entire image. Standards also define data accesses relative to these partitioned blocks, so the ability to partition makes it easier to follow the standard.

From a practical perspective, a user likely does not want to copy data from one block to another whenever performing a partition, so it is sometimes necessary to represent partitioned data as views (e.g. aliases or references) pointing to existing data. Whether a block or alias to one, each of the 8x8 regions needs to know its location (i.e. X and Y position) within the input image, as well as its location within the physical memory it references.

Like with data formats, an abstraction should not only hide these indexing details, but also present a uniform representation to the user such that they do not care if the data they receive is the raw data itself, or some view upon the data.

2.1.3 Color Transformation and Chroma Subsampling

Color transformation operations convert the input pixels into another color space more suitable for compression. A common transformation converts from the RGB color space to the YCbCr [23] color space. With data in the YCbCr color space, systems will often reduce the resolution of the Cb and Cr (*chroma*) components through *chroma subsampling*. While this inherently introduces loss into the compressed bitstream, the chroma components have less of an effect on human perception as compared to the Y (*luma*) component, thus the overall loss in quality becomes largely unnoticeable [24].

3: These transformations are shown as this function is referred back to several times.

Converting RGB to YCbCr applies the linear transformations[3] shown below to each component in the 8x8 block [24]:

$$
\begin{aligned}
RGB \rightarrow YCbCr \\
Y &= \frac{77}{256}R + \frac{150}{256}G + \frac{29}{256}B \\
Cb &= -\frac{44}{256}R - \frac{87}{256}G + \frac{131}{256}B + 128 \\
Cr &= \frac{131}{256}R - \frac{110}{256}G - \frac{21}{256}B + 128
\end{aligned}
$$

4: 4:2:0 is the most widely used subsampling format.

JPEG supports several subsampling formats, namely 4:4:4, 4:2:2, 4:1:1, and 4:2:0, which define the ratio of luma-to-chroma resolution[4]. 4:4:4 performs no subsampling, keeping all components. 4:2:2 and 4:1:1 halve and quarter the chroma horizontal resolution, respectively, while 4:2:0 halves both the horizontal and vertical chroma resolution. Figure 2.3 visualizes each type of subsampling, showing which Cb and Cr components are kept relative to the Y components for a 2x4 block of pixels.

Considerations for an Abstraction

The color transformation operation provides an example of an elementwise transform across a block of data, which is an extremely common pattern of computation within compression. While applying a transformation like this is relatively simple, depending on how much partitioning has happened and what the underlying layout of the data is, computing the correct indices into the underlying data can become arbitrarily complex. Again, an abstraction should hide these details.

Subsampling also provides another important aspect of location where data may have a different parameterization, yet still represent the same location.

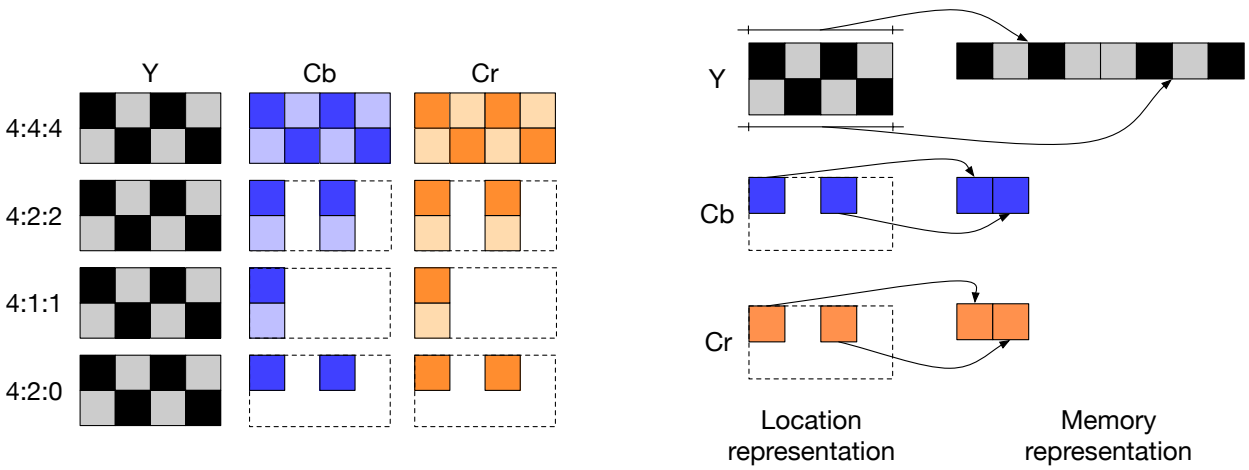


Figure 2.3: Chroma subsampling in JPEG. The left side shows how the different subsamplings for Cb and Cr are spatially aligned with respect to the unsampled Y components. The right side shows how the 4:2:0 components would map to adjacent locations in memory (which would be similar to the other subsampling variations as well). The dashed boxes emphasize that the subsampled versions still represent the same location as the Y samples.

For example, despite subsampling effectively removing or skipping over data in the Cb and Cr planes (except for 4:4:4), the region of data that results still represents the *same part* of the image as the Y plane. In other words, it still has the same location.

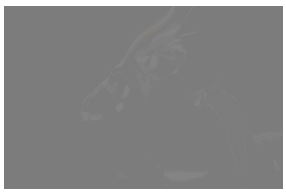
Another way to look at subsampling is as if Cb and Cr represent strided versions of the Y data. For example, 4:2:2 has a stride of one in the vertical dimension since it preserves all the data, and a stride of two in the horizontal dimension since it skips every other pixel. While not as important within this particular example, many stages of compression require having mappings between different values based on location. These mappings may be one-to-one, many-to-one, or one-to-many, which introduce additional logic when computing the underlying array indices. For one-to-one mappings, accesses across different pieces of data are straightforward. However, for anything more than a small number of local mappings that are not one-to-one, having to manually track all of the necessary metadata for location (and then having to factor in views or different data formats) can quickly get out of hand, leading to unnecessary code obscurity[5] and leaving the door open for indexing bugs.

5: The low-level operations on individual pixels tend to be quite simple, but additional computations for computing the actual index into the arrays overshadow these operations, leading to code obscurity.

2.1.4 Transform and Quantization

The *transform* stage applies an 8x8 discrete cosine transform [25] (DCT) to each of the blocks. For JPEG, this transform can be implemented as two separate 1-dimensional transforms, the first applied to each row in the 8x8 block, and the second applied to each column of the transformed rows. These types of Fourier transforms are an integral part of many block-based compression systems and serve to separate the values into their lower-frequency and higher-frequency components (both referred to as *coefficients*). Lower-frequency coefficients carry more "useful" information than high-frequency coefficients, and get compacted into the upper left-hand corner of the 8x8 block. The following *quantization* stage maps the coefficients in the block to a different set of values with a smaller range, often by dividing the coefficients by some (potentially different) quantization value. This brings many of the coefficients down to zero.

Quantization provides the main tradeoff between compression ratio and compression quality. The larger the divisors used in quantization, the more coefficients brought to zero, which ultimately provides better compression. However, this also introduces more loss, potentially leading to a worse quality image from decoding. On the other hand, smaller divisors lead to less compression, but can improve compression quality. The appropriate setting for the quantization factor ultimately depends on the use case. Figure 2.4 shows the result of using different quality factors in a JPEG encoder. Note the diminishing returns in quality between 50 and 95, despite the nearly 10× increase in size of the compressed image.



(a) Quality=1, compressed size=288KB

(b) Quality=50, compressed size=965KB

(c) Quality=95, compressed size=8.9MB

Figure 2.4: Differences in quality and size of an image compressed with various amounts of quantization. (a) has the best compression ratio, but is completely unrecognizable except for faint lines outlining the deer. (b) and (c) are nearly indistinguishable, but have very different compression ratios.

Considerations for an Abstraction

The DCT and quantization provide additional examples of using partitioning and elementwise operations. The DCT is particularly interesting since it is usually implemented as a separable transform, meaning it uses a one-dimensional DCT across each of the rows of the data, followed by another 1D

DCT across each of the columns. The computation on each row and column are largely the same (except for scaling factors), thus an abstraction should be able to represent a one-dimensional transform invariant to the particular row/column layout.

2.1.5 Differential Encoding, Zigzag, Entropy Coding, and Syntax Output

Differential encoding subtracts the upper-left coefficient of each 8x8 block (called the *DC* coefficient) from the DC coefficient of the prior block. The dynamic range of values taken on by the difference is often smaller than the raw value itself, which lends itself to better compression in the later entropy coding phase. Next, the zigzag phase iterates through each of the remaining 63 coefficients of the block (called the *AC* coefficients) in a zigzag order and performs run-length encoding. Here, run-length encoding gathers sequences of adjacent zeros in the block and represents them with two different values: the number of zeros in the run, and the non-zero value occurring after the run. This reduces the number of individual values to compress, which can improve the overall compression ratio. While JPEG utilizes the zigzag ordering here, other orders exist, such as Hilbert and Morton orderings [26]. Depending on the common patterns of zeros in blocks, different types of compression can benefit from alternate orderings. Figure 2.5 gives an example of zigzag run-length encoding.

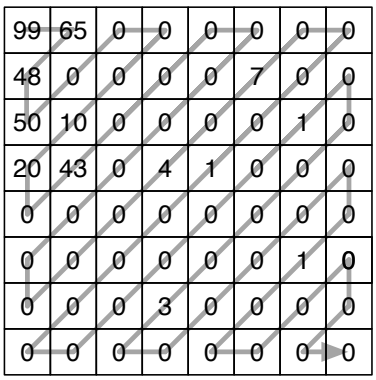


Figure 2.5: Run length encoding with a zigzag. Runs are encoded as (⟨#zeros⟩, ⟨value after last zero⟩).

Entropy coding replaces the run-length encoded values with codewords, mapping more frequent values to shorter code words and vice-versa. Most JPEG encoders utilize *Huffman encoding* [4], though the standard also allows another form of entropy coding known as *arithmetic coding* [27]. Entropy coding outputs all the codewords for the Y run-length values, Cb run-length values, and Cr run-length values at the 8x8 block level, so the final bitstream utilizes a hybrid interleaved-planar format (interleaved across blocks, planar within blocks). Generating the bitstream from entropy coding requires the standard for the *syntax output*, which defines the bit-by-bit structure of the bitstream through syntax elements, which define a related group of values. For example, JPEG includes a frame syntax element, which specifies image-wide values necessary for decoding, such as the original image size and chroma subsampling factors.

Considerations for an Implementation

Unlike the other stages, the operations presented here are very different from those introduced thus far. While there are some data accesses, there is not much in the way of partitioning or any type of location. However, these differences are what make these stages important. Any domain-specific language for compression should strive to support full end-to-end encoders and include the necessary features to support these additional operations. Otherwise, users would have to jump back and forth between different languages, which can reduce the overall effectiveness and usability of such a language.

2.2 H.264

While the JPEG encoder represented a fairly flat data hierarchy (except for the initial 8x8 partition and splitting up rows/columns for the DCT), H.264 [19] introduces a deeper and wider data hierarchy with several forms of partitioning, and more spatial operations within and across blocks.

The H.264 standard describes several *profiles* that provide constraints on the types of features that a decoder must support, thus the output of an encoder that says it conforms to profile X should be decodable by any decoder that also supports X. In order to give a broad look at what makes up an H.264 encoder, the description in this section does not focus on any particular profile and instead presents a pipeline of stages generally found across the different profiles. Figure 2.6 shows such a pipeline.

2.2.1 Partition

The encoder begins in a similar fashion to the JPEG encoder with the *partitioning* stage, taking in input video frames one at a time and partitioning them into planes of pixel components, and then partitioning each plane into 16x16 blocks of pixels called *macroblocks*. H.264 also operates on pixels with luma and chroma components, such as YCbCr.

Considerations for an Abstraction

Partitioning here has a similar flavor to JPEG, except H.264 includes more levels of partitioning, leading to a deeper data hierarchy. As will be seen shortly, H.264 requires more interactions across data blocks, where interactions range from further partitioning to accessing surrounding data relative to a block. All together, this increases the metadata necessary to track the locations of everything, adding further complexity to an implementation.

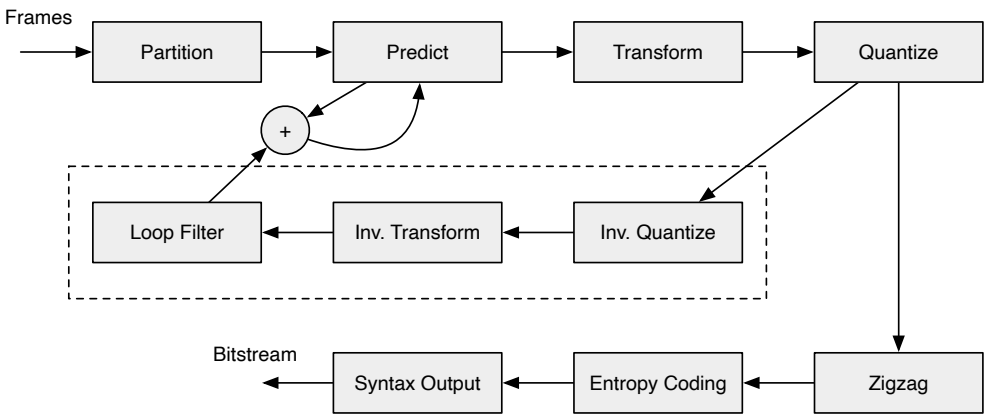


Figure 2.6: An encoder pipeline with several stages common to the various profiles of H.264. The stages in the dashed box are decoder stages used within the encoder.

2.2.2 Prediction (a First Glance)

The *prediction* stage is arguably the most important part of the pipeline. It attempts to represent the values in a macroblock as a function of other values in the frame or other frames. The encoder computes the difference between the original values and the predicted values (called the *residual*) and sends that through the rest of the pipeline, rather than the predicted values themselves. Before diving further into prediction, we will skip ahead to the rest of the pipeline, and then return to prediction in Subsection 2.2.6.

2.2.3 Transform, Quantization, Zigzag

The *transformation*, *quantization*, *zigzag* stages also operate similarly to the JPEG encoder, just with different computations. The transformation stage utilizes DCT-like transforms applied to each 4x4 block within the 16x16 macroblock, and also applies additional Hadamard transforms to part of the macroblock in some cases. The zigzag stage also operates on those 4x4 blocks.

These present the same considerations as for JPEG, so they will not be discussed further.

2.2.4 Entropy Coding and Syntax Output

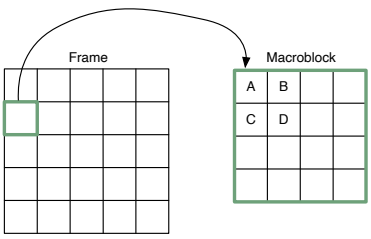


Figure 2.7: Availability of neighbors for 4x4 submacroblocks (A, B, C, D) within a macroblock.

One form of *entropy coding* used in H.264, called context-adaptive variable length coding (CAVLC), utilizes inter-block data accesses that have not been encountered yet. Without going into details of CAVLC itself, one step involves taking a 4x4 block of coefficients (the output of the transform and quantization stages) and checking if a 4x4 block exists above and to the left of it, and if they exist, checking how many zeros exist in those blocks[6]. For example, Figure 2.7 shows a frame and highlights one of the 16x16 macroblocks within it, which is broken down into its constituent 4x4 submacroblocks. The table below categorizes each of the highlighted 4x4 submacroblocks (A, B, C, and D) based on whether they have a left and/or upper neighbor, and if so, whether that neighbor is within the same macroblock or a different macroblock.

Submacroblock	Left Neighbor	Upper Neighbor
A	None	Other macroblock
B	Same macroblock	Other macroblock
C	None	Same macroblock
D	Same macroblock	Same macroblock

Syntax output operates like in JPEG, though H.264 includes a larger number of syntax elements than JPEG, and has a significantly more complex bitstream format.

Considerations for an Abstraction

From a computational point-of-view, this particular operation for CAVLC is easy since it just requires counting the number of zeros in the neighbors.

6: CAVLC uses the information to better compute the output codewords that ultimately generate the compressed data.

However, there are several interesting considerations around how to represent location in this context. First off, the non-zeros are likely held in a completely different array than the current data within the actual macroblock, so there needs to be a way to access the data containing the non-zeros based on the location of the current macroblock (and ideally the current submacroblock).

Once a reference to the correct data within the non-zeros is obtained, the next step has to access the surrounding data. This requires knowing the location of that surrounding data as well. To complicate issues, it is highly likely that the granularity of the data for the non-zeros is different from the data represented by the macroblock and submacroblock[7]. This difference in granularity adds another variable to the indexing that needs to be taken into account and presents another opportunity for simplification with an abstraction.

7: This is because only one non-zero is needed per 4x4 submacroblock, while all 16 values in the submacroblock are required for the actual step that generates the codewords.

2.2.5 Decoder Stages

The three stages in the dashed boxes in the earlier Figure 2.6 represent a series of decoder stages that operate within the encoder and serve to reconstruct the pixels in the input frame (though imperfectly since the encoder is lossy). The *inverse quantize* and *inverse transform* stages undo the quantization and transformation operations, respectively. These are similar to their forward (e.g. encoder side) counterparts. Due to the blocked structure of these encoders, the decoded blocks may have blocking effects around the edge, which introduce distortion, so the *loop filter* stage applies a filter around each block, smoothing the edges.

After loop filtering, the next step adds the residual from prediction with the decoded values, which produces the *reconstructed* pixels. The reconstructed pixels are what a decoder would ultimately generate from a compressed bitstream, and in the encoder, the reconstructed pixels are used for predicting the next macroblock. Thus, the encoding pipeline has a sequential aspect to it, where the next macroblock (or submacroblock in some cases) cannot be predicted until the reconstruction of the prior macroblock completes.

Considerations for an Abstraction

While these stages are similar to their encoder-side counterparts, the reconstructed pixels represent another scenario with location. This time, completely separate data (the input pixels and the reconstructed pixels) refer to the same location. Prediction requires being able to go back and forth between these two different pieces of data, and accessing the data that refers to specific locations between each.

2.2.6 Prediction (an In-Depth Look)

Now let us return to prediction. Figure 2.8 on Page 32 provides a closer look at the prediction stage from Figure 2.6. H.264 supports two types of prediction: intra-prediction, which performs prediction within a frame, and inter-prediction, which performs prediction across frames. Respecting the constraints of the profile, each type of prediction also contains several modes of prediction which (along with

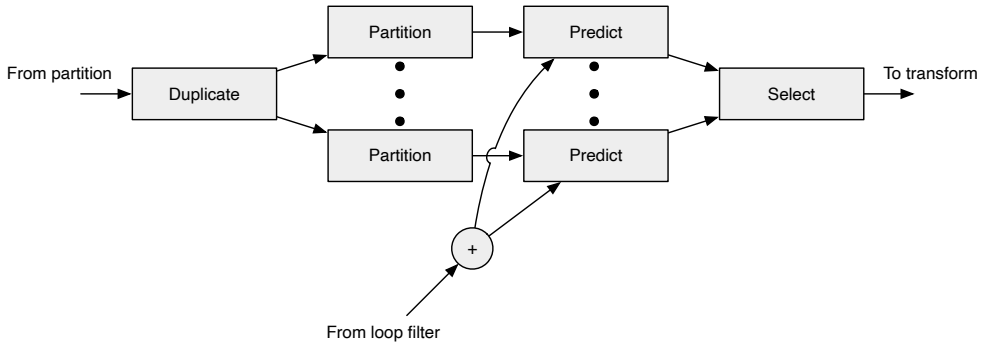


Figure 2.8: A closer look at the H.264 prediction pipeline used within Figure 2.6.

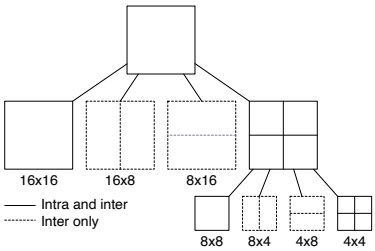


Figure 2.9: Possible partitions for intra- and inter-prediction in H.264.

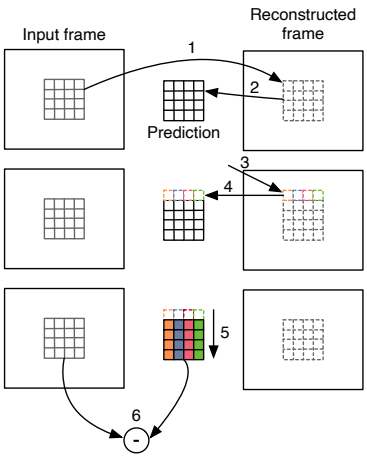


Figure 2.10: Step-by-step breakdown for intra-prediction with 4x4 vertical mode in H.264. See the main text to the side for a complete description.

a few other parameters) define the values that can be used for prediction. The encoder can try out any type of prediction it wants, so the first stage *duplicates* the current macroblock (while maintaining the location with it) and passes those off to the various modes of prediction. Next, the encoder performs an optional second *partition* stage based on the type of prediction.

Intra-prediction can operate on full 16x16 macroblocks, as well as 8x8 and 4x4 submacroblocks. Inter-prediction can also operate on sizes of 16x16, 8x8, 4x4, 8x16, 16x8, 4x8, and 8x4. Figure 2.9 shows the partition options for intra- and inter-prediction.

The actual prediction stage varies based on whether the encoder performs intra- or inter-prediction. Figure 2.10 shows an example of the steps involved in intra-prediction for H.264 using a particular 4x4 mode referred to as vertical prediction. As the name suggests, this mode operates on a 4x4 submacroblock and creates the prediction by copying values vertically. Recall that prediction utilizes reconstructed values (i.e. the values resulting from the loop filter stage). The steps for vertical prediction in Figure 2.10 are as follows:

1. Take the current submacroblock which points to data in the input frame and find its corresponding location in the reconstructed frame.
2. Make a copy of the 4x4 submacroblock with the same location. This will hold the resulting prediction.
3. Access the row of pixel values above the submacroblock within the reconstructed frame.
4. Translate the row of pixels over to the submacroblock copy.
5. Fill the copy with the row of pixels by copying them vertically.
6. Compute the residual by subtracting the values in the copy from the corresponding data in the input frame.

In total, H.264 supports nine intra-prediction modes for 4x4 prediction, nine modes for 8x8 prediction, and four modes for 16x16 prediction. All of them follow this same general procedure.

Inter-prediction operates across frames, trying to find the best match for the current block in either a prior or future frame of reconstructed values. In H.264, inter-prediction takes the form of *motion estimation*, which attempts to compute how far objects move between these prior and future frames, and encodes the prediction as a *motion vector* that captures the horizontal and vertical offsets of

the object. Figure 2.11 shows an example of using motion estimation with the following steps:

1. Start with frame t and t+1 and highlight the region of interest to predict (the dashed box in frame t).
2. Compare the data at the location in reconstructed frame t+1 with the region in frame t.
3. Compare the data at the location in reconstructed frame t+1 with the region in frame t.
4. Compare the data at the location in reconstructed frame t+1 with the region in frame t. This is the best option since it most closely matches.
5. Compute the motion vector between the two selections by computing the Y-X offsets.
6. Compute the residual by subtracting the values in the predictioj from the corresponding data in the input frame.

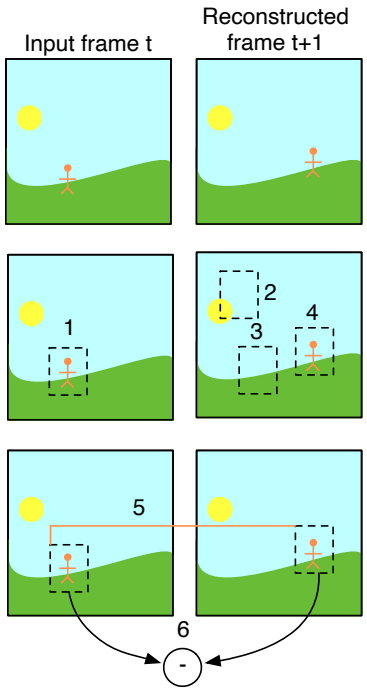


Figure 2.11: Step-by-step breakdown for inter-prediction between two frames in H.264. See the main text to the side for a complete description.

The most important part of inter-prediction centers around determining which block of data within a prior or future reconstructed frame to use for computing the motion vectors. There are two components to this selection process, namely 1) how to search the frame, and 2) how to compare blocks during the search. This search and compare is one of the most expensive parts of H.264 encoding, and numerous papers have been devoted to improving the search from both an algorithmic and implementation perspective, such as hexagonal search in [28–31].

At a high-level, the search moves a stencil across the reconstructed frame, comparing the pixel components at the stencil points with the corresponding pixel components in the current block being predicted. How to do the comparison between the pixel components varies, but a simple cost function such as sum-of-absolute differences can be used. Once the cost hits some threshold, the encoder uses the selected block in the reconstructed frame as the prediction and computes the motion vector from that.

If the pixel components are used directly in the comparison, this is referred to as integer motion estimation since the motion vectors computed will have integral values. However, H.264 also supports $1/4$-pixel motion estimation[8], which means the motion vectors will have $1/4$-pixel resolution. This requires interpolating the pixel components within the reconstructed and the current block, which creates values "in between" the actual pixel components. H.264 defines several filters used in series to compute the interpolated pixel values.

8: $1/4$-for the luma, and $1/8$ for chroma in 4:2:0 chroma subsampling.

Since the encoder can potentially attempt many different ways of predicting a macroblock, at some point the encoder needs to commit to one form of prediction, which is where the *select* stage comes into play. Here, the encoder picks the best type of prediction and passes that information along through the rest of the pipeline with the macroblock.

Considerations for an Abstraction

Prediction utilizes all the operations discussed thus far, but to an even further degree. There are a variety of different partitions, many different copies of blocks referring to the same location (potentially with different granularities),

but holding different data (e.g. holding the results of prediction, holding the results of the different modes of prediction). There are also accesses adjacent to regions of data, and mappings from one location to another.

2.3 Beyond JPEG and H.264

- H.261: see [32]
- MPEG-1: see [33]
- JPEG: see [22]
- MPEG-2: see [34]
- H.263: see [35]
- MPEG-4: see [36]
- JPEG2000: see [37]
- H.264: see [19]
- VP8: see [38]
- JPEGXR: see [39]
- VP9: see [40]
- H.265: see [41]
- JPEGXS: see [42]
- H.266: see [43]
- JPEGXL: see [44]

JPEG and H.264 are not the only forms of block-based compression, and numerous standards have been developed since the 1980s utilizing block-based compression for image and video. However many of them build on top of one another, leading to the common structure discussed in the prior sections. Figure 2.12 shows a timeline of some of the major image and video standards introduced in this area.

Newer standards extend the compression capabilities of prior ones in many ways, such as supporting new color spaces, allowing more block sizes, introducing new prediction modes, and so on. New standards also introduce non-compression improvements as well, such as removing dependencies between data within or across frames in order to increase the amount of parallelism possible. Other standards change the fundamental transforms used, such as JPEG2000 and JPEGXS, which utilize wavelet transforms instead of DCTs[9].

9: While not considered here, wavelet transforms create a blocked-hierarchical structure, thus can utilize the abstractions presented later.

Future-proofing an abstraction

While we expect much of the structure to remain the same as new standards become available, it is important to maintain a degree of flexibility in the abstraction to support any new features that might arise. Based on variations across the current set of block-based compression, we believe the most important part of the abstraction would be the ability to compare locations across blocks that live in different "frames of reference." For example, the input stage of JPEG showed that image data that could be stored in an interleaved or planar format. These two formats could be considered different frames of reference because they represent the *same* data at the *same* location, but just have different access semantics. Other operations like interpolation add on another frame of reference that increases the point density for two sets of data representing the same location. The abstraction must be able to define locations invariant to these frames of reference, otherwise it would not be able to compute meaningful relationships between locations.

The UniTe abstraction presented in Chapter 4 does exactly this and provides a mathematical way to connect together these different frames of reference in order to be able to transform locations between them. While UniTe represents

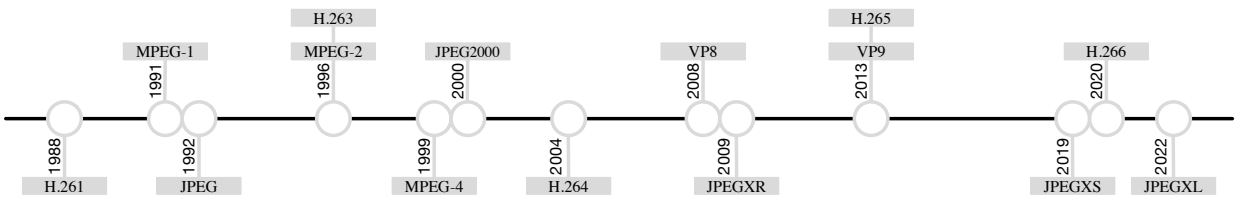


Figure 2.12: Selection of major image and video compression standards.

a specific set of ways to parameterize these frames of reference, it is possible that other parameterizations would be useful for future forms of block-based compression. UniTe focuses on providing the foundations of *how* to represent these frames as opposed to unnecessarily constraining the abstraction to one particular set of representations. This makes it possible to extend UniTe to meet future needs. For example, it is reasonable to assume that compressing sparse data could become a part of block-based compression standards in the future. Computing the data elements that correspond to a particular location in sparse data is different than computing it for dense data; "sparse" would become a new frame of reference that requires a transform specifying how to go between sparse and dense locations. UniTe would be able to support this, provided that the transforms (and the other necessary components that UniTe requires) are specified.

2.4 Summary

This chapter provided a high-level overview of what block-based compression is and the different stages that make up an encoder. It shows two exemplar encoder pipelines, namely JPEG for image compression and H.264 for video compression, and discussed the overall control flow and primary stages in each. JPEG provides a gentle overview of the basic operations required in block-based compression, while H.264 provides a more advanced set of operations requiring various spatial operations between blocks of data.

Despite the variety of ways to implement encoders and the sheer number of standards for compression that exist, these encoders do share a lot of structure. Since the standards often build off of the prior ones, future block-based compression standards will undoubtedly continue to follow the same basic structure, meaning encoders will also continue to share the same structure. However, the complexity of both the standards and associated implementations will undoubtedly continue to increase with each new standard as well, making an abstraction all the more necessary.

Chapters 3 to 5 capture the abstractions alluded to in this chapter. These abstractions are more broadly applied than just compression, thus they are introduced in isolation of compression. Later, Chapters 6 to 9 bring compression back into the picture and tie together the abstractions with block-based compression.

Understanding Spatial Relationships

3

The last chapter showed several different ways of representing and utilizing blocks of data within the stages of block-based encoders, sometimes performing operations within blocks, and other times performing operations across blocks. The two key features are that the blocks display *spatial relationships* amongst one another and also require different *frames of reference* to describe their location. Spatial relationships here refer to the fact that blocks need to know and reason about their location relative to other blocks. For example, in video compression it is necessary to associate a timestamp with a frame that says where it belongs relative to its containing video. Frame of reference refers to the different ways blocks are parameterized, such as some having different layouts (e.g. interleaved vs. planar), different sizes (e.g. macroblocks, submacroblocks), and so on.

The next chapter describes the UniTe (Universal Tensor) abstraction, which provides the necessary foundations to capture spatial relationships across tensors that have different representations (i.e. live in different frames of reference). That chapter includes all the concrete mathematical components of the UniTe abstraction, with a particular focus on how to define the different frames of reference (referred to as *reference spaces* later on). However, this chapter provides some insight on what is meant by spatial relationships and what needs to go into an abstraction that can capture them. The first scenario presented looks at how to describe the location of a single object (myself) relative to different physical landmarks. The second scenario looks at frames within a video and explores the spatial relationship between those frames.

3.1 Scenario 1: Describing My Location

Assume I want to describe my location within my office in CSAIL[1] to someone. The simplest way to describe this location is to just describe my location locally within CSAIL using my office number, G740. Figure 3.1 on Page 38 shows this first description. For someone who knows CSAIL, this number likely provides enough information to find my office; however, if someone does not know where CSAIL is located, this does not really help. So, I can give more information and define my location relative to the inside of the Stata Center and instead say that I am in office G740 in CSAIL, which is on the 7th floor of the Gates tower of the Stata Center. Figure 3.2 on Page 38 shows this second description.

1: CSAIL is the Computer Science and Artificial Intelligence Laboratory at MIT and is housed within the Stata Center.

Now, if the person does not know where Stata is, I can again provide more context and describe my location relative to the Kendall Sq. area around campus and say that I am in office G740 in CSAIL on the 7th floor of the Gates tower of the Stata Center, which is 0.3mi from the Kendall Sq. T stop. In this case, I have effectively represented my location using three different frames of reference, each of which builds on the prior (i.e. Stata's location is relative to the T, my office's location is relative to Stata). The object whose location is being described (myself) does

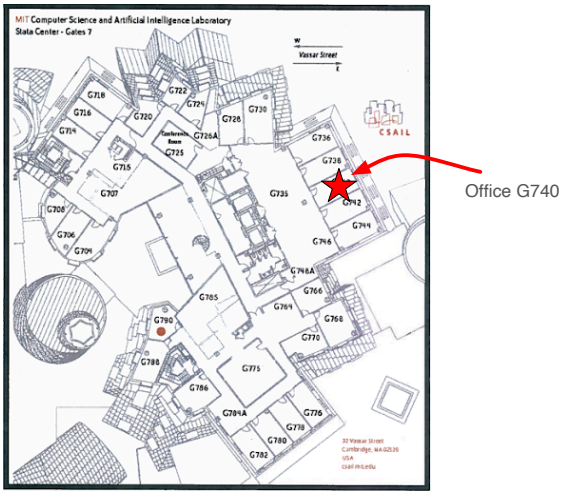


Figure 3.1: Describing my office location in CSAIL using a floor map and my office number.

not change; rather, the description of the location does. Figure 3.3 on Page 39 shows this third representation.

Now, say I want to go somewhere else, so instead I head from my office over to the Muddy Charles Pub on the other side of campus. The object is still the same; it is me. But now I have a totally different location that I need to describe. I can use the same strategy as before and define my location relative to the same Kendall Sq. area and say that I am at the Muddy Charles Pub which is 0.4mi from the T stop[2]. Using the same Kendall Sq. frame of reference, I could also compute a distance between the Stata Center and the Muddy Charles Pub, thus despite the different frames of reference and physical locations, it is possible to compute a spatial relationship between myself in these two spots. Figure 3.4 on Page 39 shows this extension.

2: The T stop is used for the frame of reference here as it does not make sense to describe the location of something external to the Stata Center or CSAIL using a frame of reference within the Stata Center or CSAIL.

I can continue to arbitrarily expand this scenario by adding new frames of reference or moving myself to different physical locations. For example, if I decide to head home and describe my location relative to that same T stop, it is no longer sufficient to just represent my location within Kendall Sq. since I do not live near there. Instead I would need to take another step back and use another frame of reference for Massachusetts, and then can compute the distance that way. The

Figure 3.2: Describing my office location in CSAIL using a floor map (Figure 3.1) and relative description within the Stata Center.

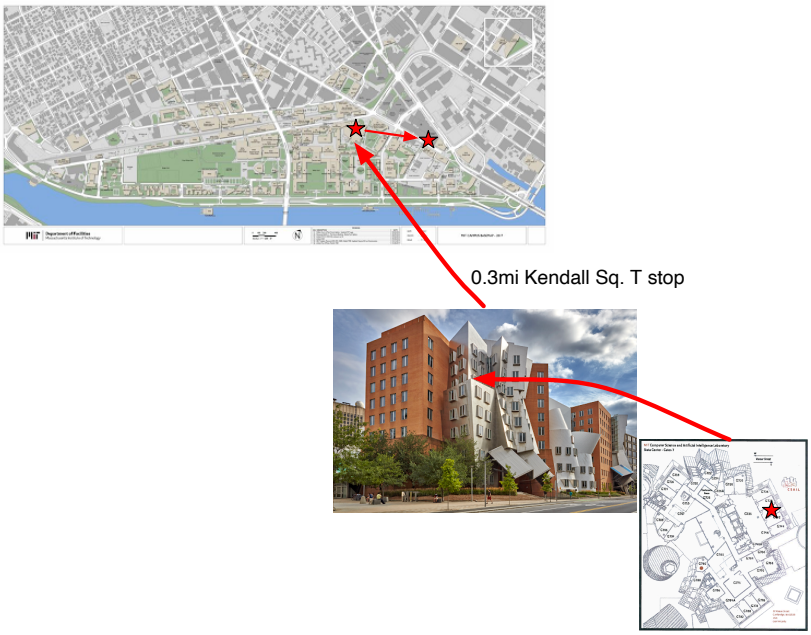


Figure 3.3: Describing my office location in CSAIL using a floor map (Figure 3.1), relative description within the Stata Center (Figure 3.2), and distance from the Kendall Sq. T stop.

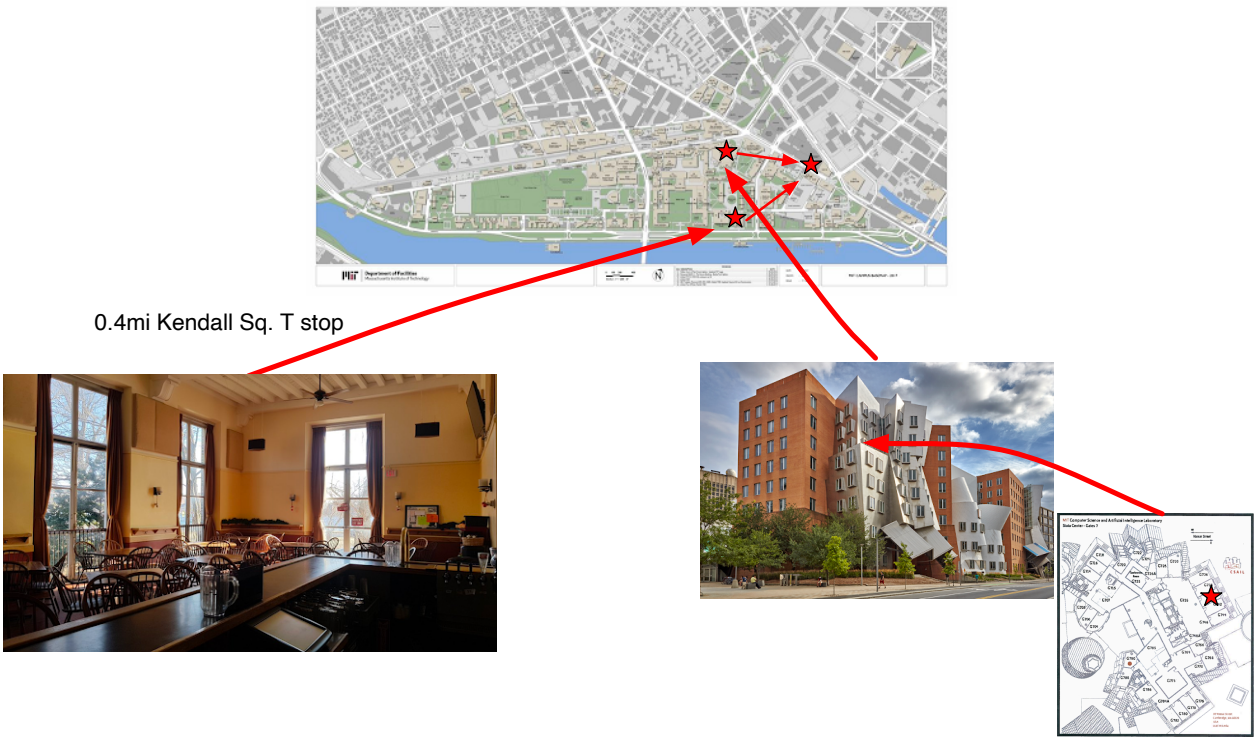


Figure 3.4: Describing my location at the Muddy Charles Pub relative to the Kendall Sq. T stop.

top of Figure 3.5 on Page 40 shows this final extension.

Regardless of how I change the example, the key is having the continued ability to define my location and compare it across different frames of reference. The bottom of Figure 3.5 shows a trie representing this scenario, with the root

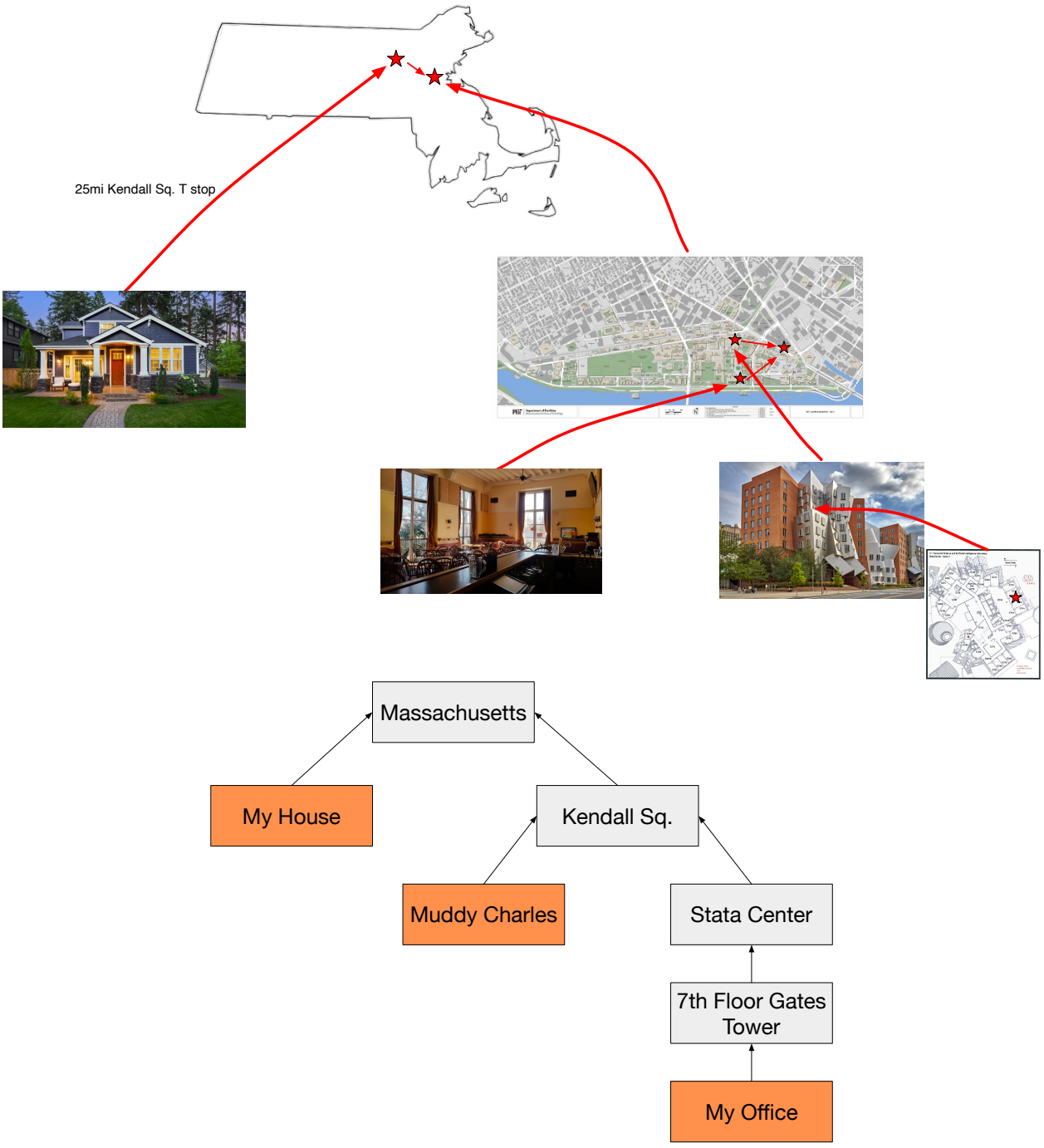


Figure 3.5: Describing the location at my house relative to the Kendall Sq. T stop using Massachusetts as a frame of reference. The bottom trie shows the relative relationship between the different frames of reference described within the example (grey) and the different locations (orange).

representing a *global* frame of reference with respect to all the other frames of reference within the root's *universe*. Global means that it is possible to characterize any location within the trie with respect to the root itself, while universe means that only locations captured within the trie can be compared to each other[3].

3: For example, I could have the Massachusetts universe, and then another New Hampshire universe. Each can represent all the locations within the respective state, but I cannot use one to represent a location within the other because it does not logically make sense.

3.2 Scenario 2: Locations Across a Video

Consider a video like the one represented in Figure 3.6. The video can be broken up into individual frames, each of which has a location associated with a particular timestamp within the video. The bottom left of the image shows a single frame extracted at time t within the video, and this frame contains the same data as the video at time t. Now look at the frame on the bottom right. This frame holds a greyscale version of the frame on the left. Despite the fact that it represents different data now (it is not in color), it still logically represents the same location at time t within the initial video.

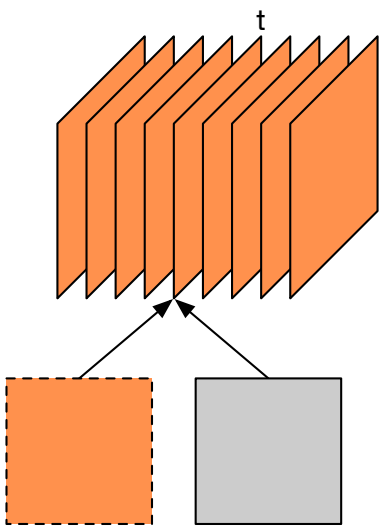


Figure 3.6: Two frames at time t within a video. Both have different data, but still logically represent the same location (any differences in frame resolution are ignored for this example).

Figure 3.7 on Page 42 shows this example, but using a different frame of reference to represent the grey frame. Here, the grey frame still represents the same location, except it skips over elements in the horizontal and vertical dimension (for example, it could be subsampled). Now, there is combination of different data and different frames of reference used to describe the same location; however, it is still necessary to be able to compare and reason about the locations regardless of these differences.

With this example, the first important concept is that location is a property of the logical container (e.g. the video, a frame, a row in the frame), not the data itself. The second important concept (shared with the office location example) is that the video represents a common ancestor for describing the location of all the other objects. This means location is essentially invariant to the different frames of reference, different data, etc. All the different pieces, as long as they have locations within the video *universe*, can have their locations compared, where the comparison can be something like checking if two frames occur at the same timestamp, or computing the offset between the times of two different frames.

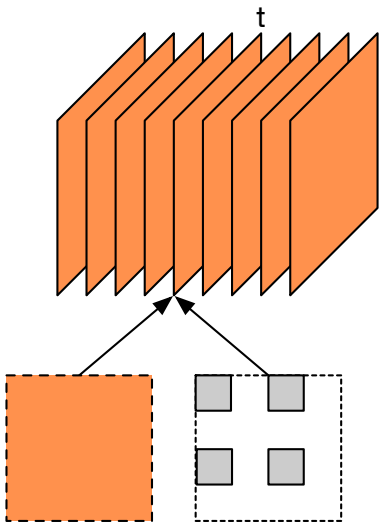


Figure 3.7: Two frames at time t within a video that each live within a different frame of reference. The finely dashed box around the grey blocks emphasizes that the grey blocks still represent the same location (time t) within the video as the orange frame.

3.3 Representing Location Manually

Modern programs lack the abstractions necessary to represent these types of spatial relationships among objects, shifting the burden of doing so to the user. As a simple example, consider the code in Figure 3.8. This is NumPy code that models computing the various parts of the video and frame example in the last section. The issues arise from the NumPy view, `orange_frame`, on Line 2 and ndarray[4], `grey_frame`, on Line 4. Even though these logically both correspond to the same time within the video, NumPy has no way to encode that information, thus the user would have to associate a timestamp with the objects to track the location. Internally, NumPy has to maintain a mapping from `orange_frame` to `video` since `orange_frame` is a view on `video`[5]. However, NumPy does not expose this information to the user, so it is not helpful for tracking spatial relationships. In a simple localized context such as this, tracking location is manageable. But with more realistic domains containing numerous different representations of data spread across entire programs, the required bookkeeping quickly becomes overwhelming.

4: This is a multidimensional array that allocates new memory.

5: A view being a reference to an existing ndarray, such that accessing data in a view propagates to the ndarray and gets the data from there.

Looking ahead to block-based compression presents a more realistic view of the difficulties in manually tracking location as well. The struct shown in Figure 3.9 on Page 43 is from the H.264 reference code, JM [8], and contains several parameters for describing the location of just a single macroblock. There are approximately 130 total parameters dealing with location in the file that contains this struct definition. These parameters are necessary to describe the macroblock and pixels within it relative to various input and output frames, as well as other macroblocks. The multidimensional arrays used in block-based compression all have intrinsic location in them that describes their location relative to an image

```
1 video = ndarray([3,nframes,H,W])
2 orange_frame = video[:,t,:,:]
3 # "subsampled" version of orange_frame
4 grey_frame = ndarray([H/2,W/2])
5 orange_to_grey(orange_frame, grey_frame)
```

Figure 3.8: NumPy code roughly corresponding to the scenario in Figure 3.7. NumPy has no intrinsic notion of location, thus it views `orange_frame` and `grey_frame` completely independently of one another, even though they logically represent the same location.

```
//! Macroblock
typedef struct macroblock_enc
{
  ...
  short mb_x;
  short mb_y;
  short block_x;
  short block_y;
  short pix_x;
  short pix_y;
  short pix_c_x;
  short pix_c_y;
  short opix_y;
  short opix_c_y;
  short subblock_x;
  short subblock_y;
  ...
```

Figure 3.9: Partial definition of the macroblock struct in [8].

(for image compression) or a video (for video compression). Everything from the image or video down to a pixel has a location that is used at some point during compression. However, these arrays exist in different frames of reference, sometimes having different storage formats (interleaved vs planar), different densities (interpolation vs subsampling), different dimensionalities (2D frame vs. 3D video), and so on. Having to manually deal with these different parameters across tens of thousands of lines of code quickly becomes a monumental task that opens the door for data access bugs. It also leads to inconsistencies in the design of the necessary data structures, which may be designed by several different developers, even within a single implementation. Chapters 6 and 7 explore this in more detail.

3.4 Summary

This chapter provides intuition for what it means to have data with location and how to describe the spatial relationships across the data. The key to capturing and reasoning about these spatial relationships is being able to describe the location of data invariant to its actual representation, which can vary in several ways. For example, the office example described my location relative to different physical locations, while the video example described the same location, but varied the actual data associated with the location.

The next chapter introduces a mathematical framework called UniTe that captures these spatial relationships and provides fundamental operations for defining frames of reference, defining location, moving between frames, and comparing locations. UniTe provides the necessary basis for defining higher-order representations and operations that exploit location (Chapter 5), which can be used to simplify domains containing spatial relationships, such as block-based compression (Chapter 6).

Formalizing UniTe 4

The last chapters provided some initial motivation for the association of location with different objects, namely tensors. Those examples highlight several different cases where tensors can represent the same data and the same location, or some combination of different data and different location. Regardless of the combination, the key is that one should be able to easily represent and compute the location of a given tensor relative to the other tensors where necessary. However, to complicate the issue, the tensors themselves may have different parameterizations, such as different dimensionalities, sizes, and origins, leading to arbitrarily complex ways to capture the location.

This chapter presents a mathematical framework for capturing and reasoning about these spatial relationships, starting from a point-based representation and culminating in the tensor-based representation, UniTe (Universal Tensor abstraction).

4.1 Building up an Abstraction

Before diving into the formal definitions for capturing spatial relationships, it is worthwhile to take a step back and understand what makes up the core of UniTe. Figures 4.1 and 4.2 on Pages 46 and 47, respectively, provide a working example for this section that highlights how each component builds off of the prior one.

The first necessary component for such an abstraction is a set of "things" that have a location. Since UniTe opts for a mathematical representation of location, it simply uses points (top left of Figure 4.1). By themselves, these points do not offer much, thus the next necessary component is a way to define the local location of a point. This is where the frames of reference from the prior chapter come into play. UniTe defines these as mathematical spaces called *reference spaces*, which have an attached orthogonal coordinate system (top right of Figure 4.1). These reference spaces and orthogonal coordinates make it possible to represent the local location of points and can be used to define local spatial relationships. However, by themselves, they do not help define relationships across different spaces. Rather, they require a way to define relative relationships between the different reference spaces and points across them. UniTe breaks this into two parts: reference space mappings and point-to-point mappings.

Reference space mappings provide a well-defined way to quantify how two different reference spaces differ from one another. In particular, these mappings define the relative change between a *parent* reference space and a *child* reference space. The bottom left of Figure 4.1 shows this with two different child reference spaces that differ from the parent through a permutation (left child) and slice (right child). UniTe itself defines five different possible modifications between parent and child reference spaces that support changing the dimensionality, axes layout, origin, refinement, and coarsening.

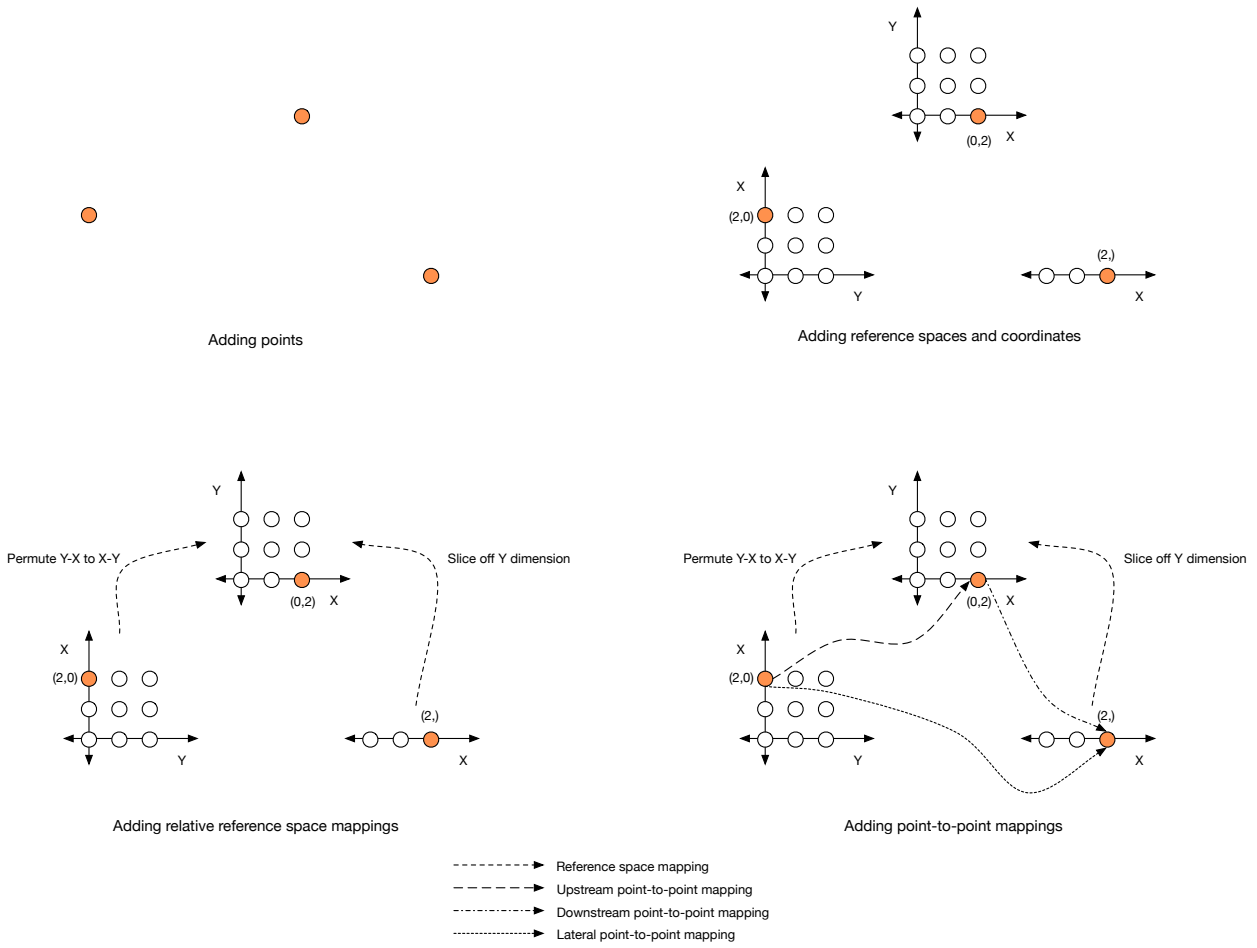


Figure 4.1: Points (top left), reference spaces (top right), reference space mappings (bottom left), and point-to-point mappings (bottom right). These four components, along with the trie shown in Figure 4.2, provide the building blocks for UniTe that make it possible to describe spatial relationships across points (and ultimately tensors).

Point-to-point mappings define how to map a point from a parent reference space to a child reference space, and vice versa. For example, the bottom right of Figure 4.1 shows different mappings between the child and parent reference spaces, as well as a lateral mapping between siblings. These mappings make it possible to reason about spatial relationships since points can be compared across the different reference spaces.

The final core component of UniTe is a way to *globally* represent spatial relationships; or in other words, provide a way to capture spatial relationships across points in reference spaces that are not necessarily connected as parent and child. UniTe defines all its reference space mappings such that they can be recursively applied to generate a *trie*. The root of this trie represents a global location within which all the points in all the descendant reference spaces can be compared. Similarly, it is always valid to compare points within a common ancestor reference space. All the point-to-point mappings in UniTe can also be recursively applied in order to move points beyond just parent-to-child or child-to-parent. In fact, it is possible to map a point from any reference space to any other reference space (assuming they are in the same trie) by repeatedly applying a combination of parent-to-child and child-to-parent mappings, even when the two reference

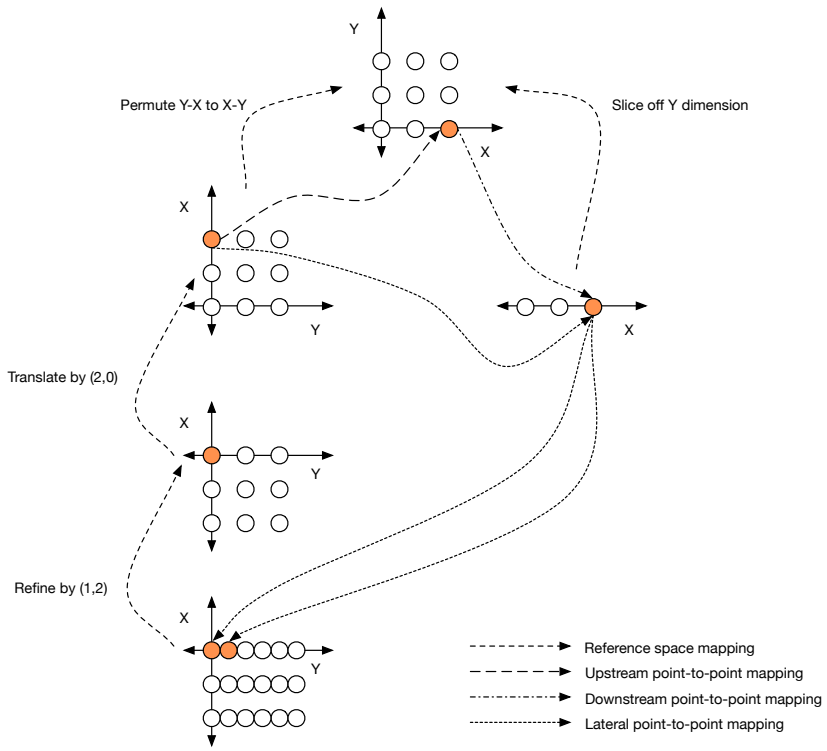


Figure 4.2: Extending Figure 4.1 into a trie by adding on additional reference spaces with their mappings. Various point-to-point mappings are included to show that points can be mapped across arbitrary reference spaces in the trie.

spaces are in different paths of the trie[1]. This is possible due to the fact that common ancestors represent a common frame of reference. Figure 4.2 shows one example of such a trie built off of Figure 4.1.

1: Though, the mapping itself may not produce a point in all cases. See Subsection 4.3.1, which discusses cases with projections.

The next sections look at these mappings in more detail, starting with reference space mappings, followed by the point-to-point mappings.

UniTe as an abstraction framework

While UniTe defines these five particular types of mappings, the purpose of the abstraction is not to constrain users to just these mappings. The ones presented in this chapter are ultimately useful for the examples of block-based compression given in later chapters, but it is expected that other domains would likely have additional or different mapping requirements. Thus, it is more useful to think of UniTe as an abstraction framework which can contain many types of mappings, as long as they conform to the constraints defined in this chapter. For example, the mappings must be able to be defined between parent and child reference spaces, but also must be able to be defined recursively in order to create the trie structure.

4.2 Reference Spaces

UniTe defines a single type of reference space that captures possible changes in the dimensionality, axes layout, origin, refinement, and coarsening between a child reference space and its parent. This section focuses on how these different

parameters modify the space itself, while the next section defines the point-to-point mappings associated with the different parameters. Definition 4.2.1 gives the definition for a reference space.

Definition 4.2.1 (Reference space) *A reference space, $\mathbb{R}_c^N$, is an N-dimensional space defined by the 5-tuple $(\mathbb{R}_p^M, \vec{p}, \vec{o}, \vec{r}, \vec{c})$, where*

$\mathbb{R}_p^M$ *is the parent reference space of* $\mathbb{R}_c^N$ $(N \leq M)$,
$\vec{p} = (p_0, \ldots, p_{N-1})$ *represents the permutation of* $\mathbb{R}_c^N$ *with respect to* $\mathbb{R}_p^M$ $(0 \leq p_i < M - 1$ *and each* p_i *has a unique value)*,
$\vec{o} = (o_0, \ldots, o_{M-1})$ *represents the origin of* $\mathbb{R}_c^N$ *with respect to* $\mathbb{R}_p^M$ $(o_i \in \mathbb{Z})$,
$\vec{r} = (r_0, \ldots, r_{N-1})$ *represents the refinement factor of* $\mathbb{R}_c^N$ *with respect to* $\mathbb{R}_p^M$ $(r_i \geq 1)$, *and*
$\vec{c} = (c_0, \ldots, c_{N-1})$ *represents the coarsening factor of* $\mathbb{R}_c^N$ *with respect to* $\mathbb{R}_p^M$ $(c_i \geq 1)$.

In addition, a dimension i cannot be both refined and coarsened (where $r_i > 1$ indicates a refinement and $c_i > 1$ indicates a coarsening), thus r_i and c_i cannot both be greater than one simultaneously.

The root of the trie, $\mathbb{U}^L$, is simply represented by its dimensionality L since it does not have a parent. This definition for reference spaces can be used to construct the trie representation since it defines each child reference space relative to its parent, thus it provides the ability to define location using different frames of reference throughout.

The first change supported by the reference space is a change in dimensionality, which allows a child reference space (dimension N) to drop dimensions from the parent (dimension M). The left diagram of Figure 4.3 shows an example of this. The only constraint is that the dimensionality must monotonically decrease from root to leaves, meaning it is not possible to add back on a dimension once it is dropped as this does not logically make sense from a spatial point-of-view.

The next change is the permutation, $\vec{p}$, which reorders the dimensions of a child reference space relative to the parent. The permutation vector $\vec{p}$ ultimately

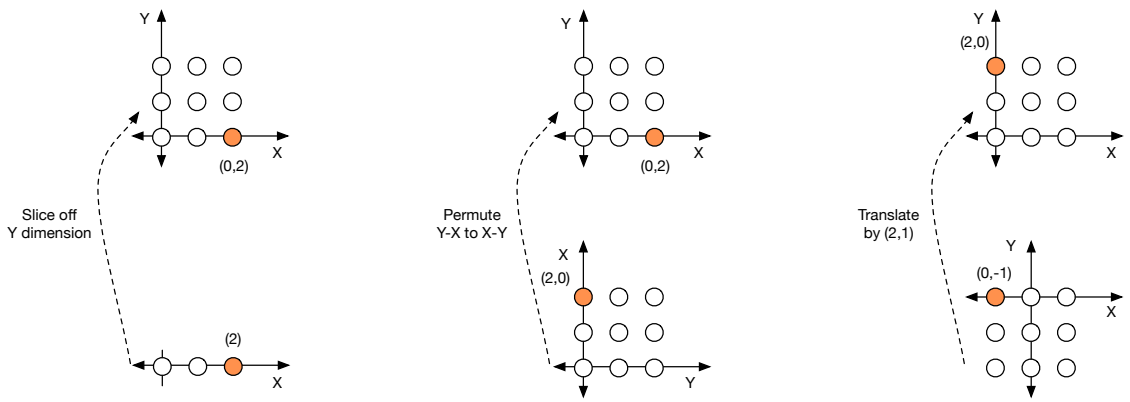


Figure 4.3: Examples of a slice (left), permutation (middle), and translation (right) operation between parent and child reference spaces. The slice operation takes the row at Y= 0 and drops then drops the outermost (Y) dimension. The permutation operation swaps the Y and X axes. Finally, the translation operation shifts the origin by $(2, 1)$.

defines a partial permutation matrix, P, which is defined in the next section in Definition 4.3.1. When the parent and child have the same dimensionality, P is square, otherwise it is rectangular. The middle of Figure 4.3 gives an example of a permutation.

A change in the origin, $\vec{o}$, represents a translation between the parent and child space. The right of Figure 4.3 gives an example of a translation.

Finally, refinement and coarsening change the density of points in a reference space[2]. Figure 4.4 gives an example of refinement and coarsening. Note that refinement introduces a one-to-many mapping from the parent to the child, while coarsening introduces a one-to-many mapping from the child to the parent. The mapping functions in the next section take this into account and ultimately work on sets of points when refinement and coarsening are used, as opposed to single points.

2: A value of one for a given dimension in both $\vec{r}$ and $\vec{c}$ indicates that no refinement or coarsening occurs in that dimension.

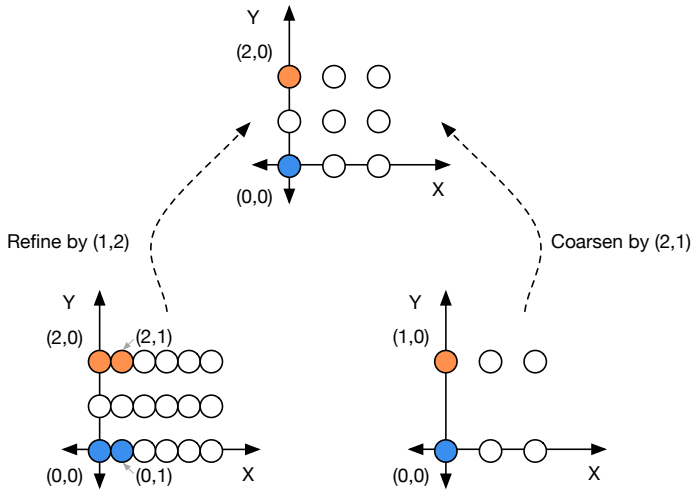


Figure 4.4: Refining and coarsening a 2D reference space.

4.3 Point Mappings

While the last section defined the different parameters for reference spaces, this section defines the various mappings necessary to map points throughout the trie. Like the reference spaces themselves, these mappings are defined between a parent and child, but can be recursively applied to map points between any two reference spaces. An *upstream mapping* ($\phi^{\uparrow}$) moves a set of points from child to parent, while a *downstream mapping* ($\phi^{\downarrow}$) moves them from parent to child. This section presents a single set of mappings that can take into account changes in all the parameters of reference spaces.

4.3.1 Mapping Types

Before giving the definitions for the mapping, this section builds up some intuition and looks at the types of mappings that arise depending on how the parameters between parent and child reference spaces change. In particular, there are three categories of mappings that naturally arise out of the various parameters:

bijective mappings, embedded mappings, and R-C mappings (where R stands for refinement and C stands for coarsening).

Bijective mappings perform affine mappings between points in the parent and child, where every point in the child maps to a unique point in the parent, and vice versa (hence the bijective part) using a combination of permutations and translations. These are the most basic type of mapping and occur when a parent and child have the same dimensionality and point density (the refinement and coarsening vectors are unit vectors). Figure 4.5 shows an example of a bijective mapping.

Embedded mappings occur when the parent and child have a different dimensionality, but still have the same point density. To represent these types of mappings, UniTe introduces projections, which map multiple points in a parent to a single point in the child. The left part of Figure 4.6 on Page 51 shows an example of a projection for an embedded mapping which projects the three colored points down to the single orange point in the child[3].

3: Technically, the reference spaces are infinite, so an infinite number of points would project to a single point in the child.

UniTe uses projections to simplify the definitions for the mappings given in Subsection 4.3.2. However, these projections can introduce mappings that are not spatially sound. This issue is depicted in Figure 4.6. The left side shows three points in the parent reference space that map to the same point in the child due to the projection. However, from a spatial point-of-view, this does not logically make sense because the top two points do not share the same location as the bottom point, thus should not be able to map to the same point in the child. This requires that embedded mappings be able to produce "one-to-none" mappings which can map an input point to the empty set, signifying that it does not have a valid location in the child. The right side of Figure 4.6 depicts this.

The trie structure used to represent reference spaces makes it easy to check for these one-to-none mappings. In particular, the root of the trie represents a global location for all points within the trie. Checking for an invalid spatial projection between a parent and child involves mapping the point in the parent to the root, then mapping the projected point to the root, and finally comparing their locations. If the locations are the same, then the projection is spatially valid; if not, then

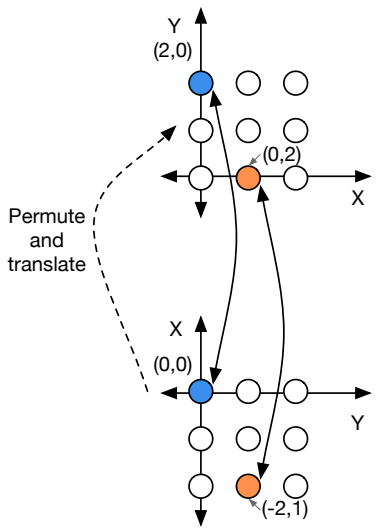


Figure 4.5: Bijective mappings between a parent and child that differ in their axes layout and origin.

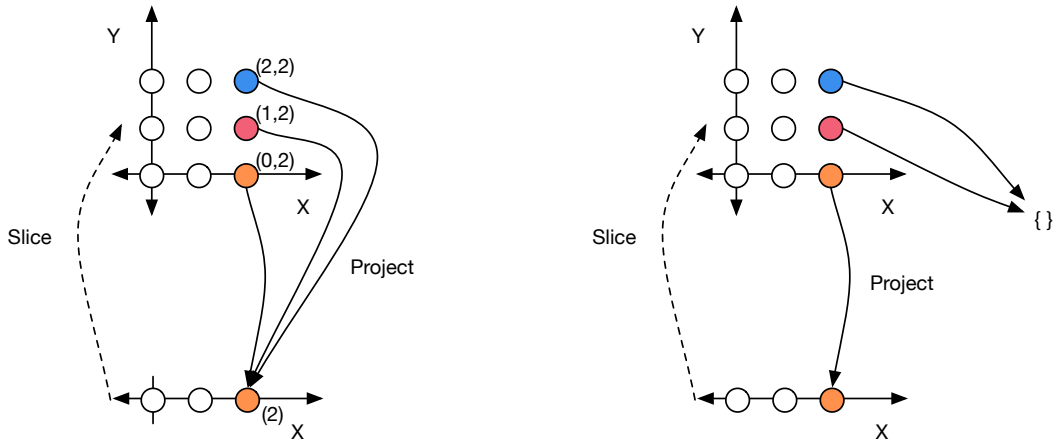


Figure 4.6: Invalid (left) and valid (right) embedded mappings arising from projections. The blue and red points represent a different location than the orange point, thus cannot project to the orange point in the 1D space.

the original points map to the empty set. Figure 4.7 on Page 52 depicts this procedure.

Finally, R-C mappings allow any of the parameters to vary and introduce one-to-many mappings in the upstream direction (when coarsening is applied) and many-to-one mappings in the downstream direction (when refinement is applied). Figure 4.8 on Page 53 depicts these types of mappings. Note that R-C mappings also allow projections.

4.3.2 Mapping Functions

The various mapping functions and related components are described in Definitions 4.3.1 to 4.3.5, starting with the partial permutation matrix [45] in Definition 4.3.1. These take into account all of the possible parameter changes, thus fully support R-C mappings, but can be simplified to support bijective and embedded mappings as well. Further discussion follows each definition.

Definition 4.3.1 (Partial permutation matrix) *Let P be an $N \times M$ partial permutation matrix defined for a child reference space, $\mathbb{R}_c^N$, with definition $\mathbb{R}_c^N = (\mathbb{R}_p^M, \vec{p}, \vec{o}, \vec{r}, \vec{c})$. Derive P from the permutation vector $\vec{p}$ as follows:*

$$P = \begin{bmatrix} P_{0,0} & \cdots & P_{0,M-1} \\ \vdots & \ddots & \vdots \\ P_{N-1,0} & \cdots & P_{N-1,M-1} \end{bmatrix}$$

$$P_{i,j} = \begin{cases} 1 & \text{If } p_i = j \\ 0 & \text{Otherwise.} \end{cases}$$

The partial permutation matrix combines together any permutations and projections defined on the axes between the parent and the child. The following

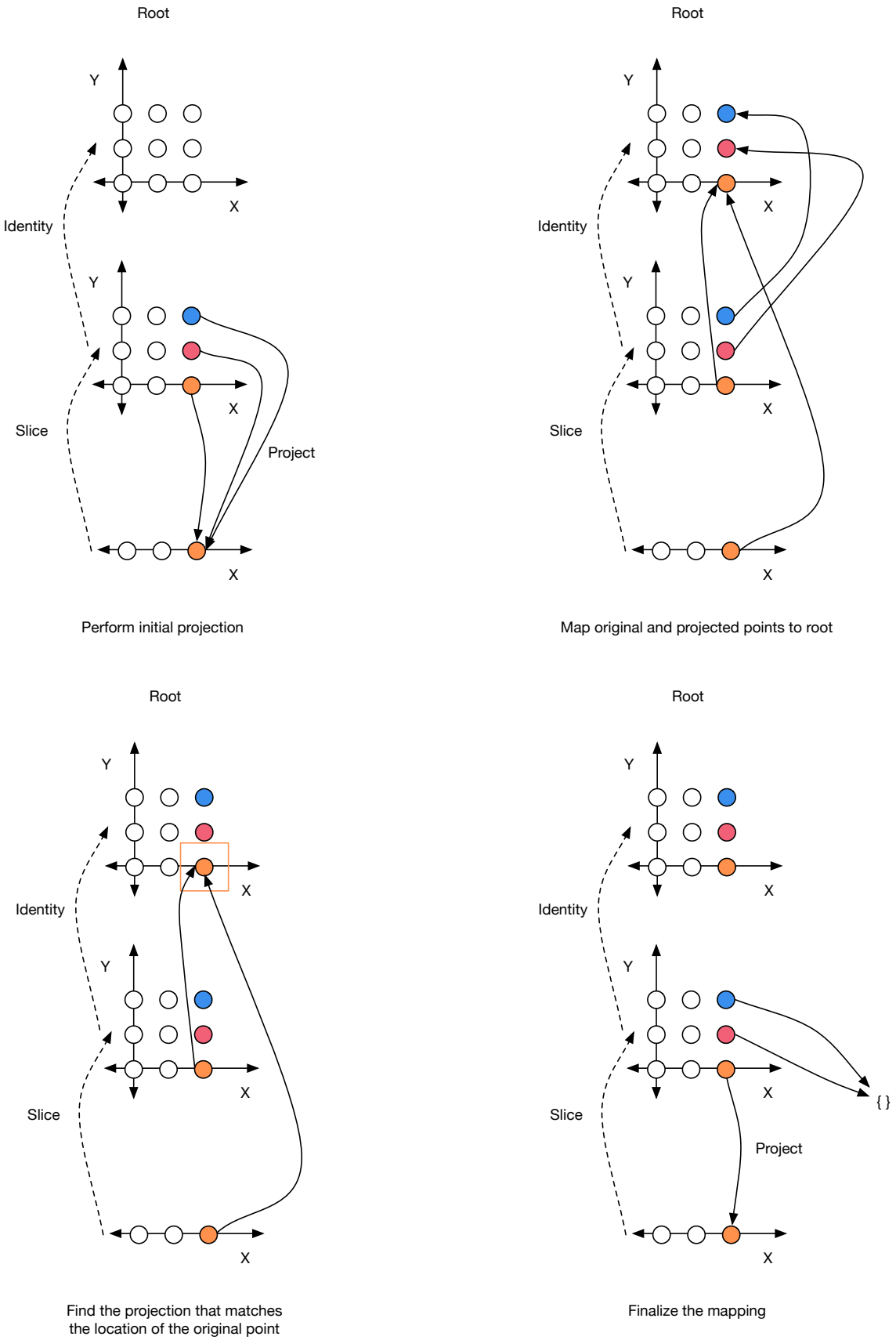


Figure 4.7: The procedure for finding valid projections with embedded mappings. The top left performs the initial projection for all points, regardless of whether the mapping is spatially valid or not. The top right maps the original points and projected point up to their corresponding locations in the root. The bottom left finds the point in the root that shares a location with both the original orange point and the projected orange point. Since the original orange point and projected orange point both map to the same location in the root, that would be a valid projection. The bottom right finalizes the mapping and allows the projection for orange, while mapping blue and red to the empty set.

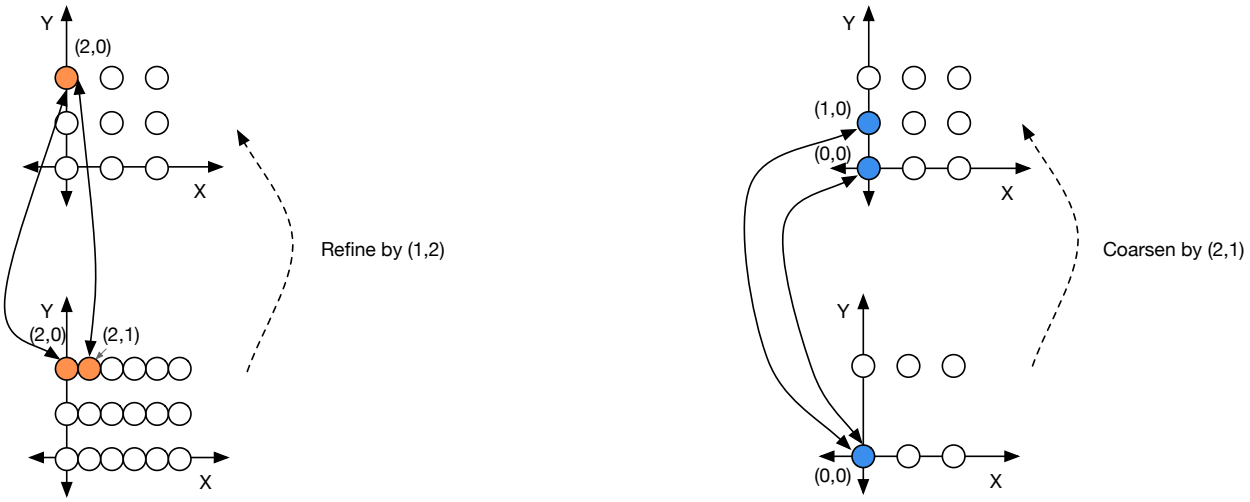


Figure 4.8: R-C mappings from refinement (left) and coarsening (right). Mappings with refinement map a single point in the parent to multiple points in the child, while mappings with coarsening map a single point in the child to multiple points in the parent.

example shows applying the matrix to do a simple mapping from parent-to-child and child-to-parent[4].

4: This example is like an embedded mapping, but with no translation. The definitions in Subsection 4.3.2 take into account all the parameters that may change.

Partial permutation example

Let the dimensionality of a parent be $M = 4$ and the child be $N = 3$ with permutation vector $\vec{p} = [3, 1, 2]$. The resulting partial permutation matrix is:

$$P = \begin{bmatrix} P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} \\ P_{1,0} & P_{1,1} & P_{1,2} & P_{1,3} \\ P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Let $\vec{x} = [x_0, x_1, x_2, x_3]^T$ be a point in the parent. To permute and project $\vec{x}$ from the parent to the child, compute $\vec{x}' = P\vec{x}$ as shown below:

$$\begin{aligned} \vec{x}' = P\vec{x} &= \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \\ &= \begin{bmatrix} x_0 \times 0 + x_1 \times 0 + x_2 \times 0 + x_3 \times 1 \\ x_0 \times 0 + x_1 \times 1 + x_2 \times 0 + x_3 \times 0 \\ x_0 \times 0 + x_1 \times 0 + x_2 \times 1 + x_3 \times 0 \end{bmatrix} = \begin{bmatrix} x_3 \\ x_1 \\ x_2 \end{bmatrix}. \end{aligned}$$

Let $\vec{y} = [y_0, y_1, y_2]^T$ be a point in the child. To undo a projection and apply a permutation from child to parent, compute $\vec{y}' = P^T\vec{y}$ as shown below:

$$\vec{y}' = P^T\vec{y} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} y_0 \times 0 + y_1 \times 0 + y_2 \times 0 \\ y_0 \times 0 + y_1 \times 1 + y_2 \times 0 \\ y_0 \times 0 + y_1 \times 0 + y_2 \times 1 \\ y_0 \times 1 + y_1 \times 0 + y_2 \times 0 \end{bmatrix} = \begin{bmatrix} 0 \\ y_1 \\ y_2 \\ y_0 \end{bmatrix}.$$

The remaining definitions for upstream and downstream mappings integrate this matrix with the remaining factors for translation, refinement, and coarsening as well. Definition 4.3.2 gives the upstream mapping function.

Definition 4.3.2 (Upstream mapping) *Let $\mathbb{R}_c^N$ be a reference space with definition $\mathbb{R}_c^N = (\mathbb{R}_p^M, \vec{p}, \vec{o}, \vec{r}, \vec{c})$. Let $S_{\vec{x}}$ be a set of points where for each $\vec{x} \in S_{\vec{x}}$, $\vec{x}$ is a point within $\mathbb{R}_c^N$. Let P be a partial permutation matrix defined on $\mathbb{R}_c^N$ and $\mathbb{R}_p^M$.*
An upstream mapping $\phi^{\uparrow}(S_{\vec{x}})$ maps the set $S_{\vec{x}}$ from $\mathbb{R}_c^N$ to $\mathbb{R}_p^M$. Compute the upstream mapping as follows:

$$\phi^{\uparrow}(S_{\vec{x}}) = \bigcup_{\vec{x} \in S_{\vec{x}}} \left\{ P^T(\vec{d} + \vec{i}) + \vec{o} \mid \textit{where for each } j \in [0, N),\ d_j = \left\lfloor \frac{c_j x_j}{r_j} \right\rfloor \textit{ and } i_j \in [0, c_j) \right\}.$$

The upstream mapping function maps a point from the child to parent by accounting for all the possible changes in parameters between the reference spaces. For upstream mappings, a coarsening between the parent and child creates the scenario where a single point in the child maps to multiple points in the parent. In particular, the child point would map to an interval of points within the parent. For a single point $\vec{x} \in S_{\vec{x}}$, the upstream mapping definition contains several key components to deal with this within the term $P^T(\vec{d} + \vec{i}) + \vec{o}$. The P^T, $\vec{d}$, and $\vec{o}$ compute the "base" point in the parent. If there is a coarsening, this base point represents the first point in the interval mapped to in the parent. The $\vec{i}$ part, along with the associated range $i_j \in [0, c_j)$ computes the rest of the points in the interval by accounting for coarsening in all possible dimensions. If multiple dimensions have coarsening applied, this amounts to a cross-product between the intervals to which they are mapped in each dimension.

This mapping is not always this complex, however. For example, in the case of bijective and embedded mappings, this equation can be simplified to omit refinement and coarsening, which results in the following:

$$\phi^{\uparrow}(S_{\vec{x}}) = \bigcup_{\vec{x} \in S_{\vec{x}}} P^T \vec{x} + \vec{o}.$$

For a bijective mapping, P is square since the dimensionality of the parent and child is the same. For an embedded mapping, it would be rectangular.

To map a point from a child to an arbitrary ancestor in the trie, it is possible to recursively apply the upstream mapping until reaching the necessary ancestor. The spatial intersection function described in Definition 4.3.3 uses such a recursive mapping.

Definition 4.3.3 (Spatial intersection function) *Let $\mathbb{R}_x^N$ and $\mathbb{R}_y^M$ be two reference spaces in the same trie with root $\mathbb{U}^L$. Let $S_{\vec{x}}$ and $S_{\vec{y}}$ be two sets of points where for each $\vec{x} \in S_{\vec{x}}$ and $\vec{y} \in S_{\vec{y}}$, $\vec{x}$ is a point in $\mathbb{R}_x^N$ and $\vec{y}$ is a point in $\mathbb{R}_y^M$.*

Let $\Phi^{\uparrow}(S_{\vec{x}})$ represent the recursive application of $\phi^{\uparrow}(S_{\vec{x}})$ from $\mathbb{R}_x^N$ to $\mathbb{U}^L$ and $\Phi^{\uparrow}(S_{\vec{y}})$ represent the recursive application of $\phi^{\uparrow}(S_{\vec{y}})$ from $\mathbb{R}_y^M$ to $\mathbb{U}^L$. The spatial intersection function $S_{\vec{x}} \cap^s S_{\vec{y}}$ computes the intersection of the sets $S_{\vec{x}}$

and $S_{\vec{y}}$ with respect to $\mathbb{U}^L$. Compute the spatial intersection as follows:

$$S'_{\vec{x}} = \Phi^{\uparrow}(S_{\vec{x}})$$
$$S'_{\vec{y}} = \Phi^{\uparrow}(S_{\vec{y}})$$
$$S_{\vec{x}} \cap^s S_{\vec{y}} = S'_{\vec{x}} \cap S'_{\vec{y}}.$$

The spatial intersection function computes a normal set intersection on two sets of points, but first maps both sets to the root of a trie. Since the root represents a global location that all points have a location within, this function effectively keeps the points in each set that overlap in their location. Figure 4.9 depicts this operation. Other related operations like spatial union are possible, but this intersection function is required for the downstream functions given next in order to account for the possibility of projections.

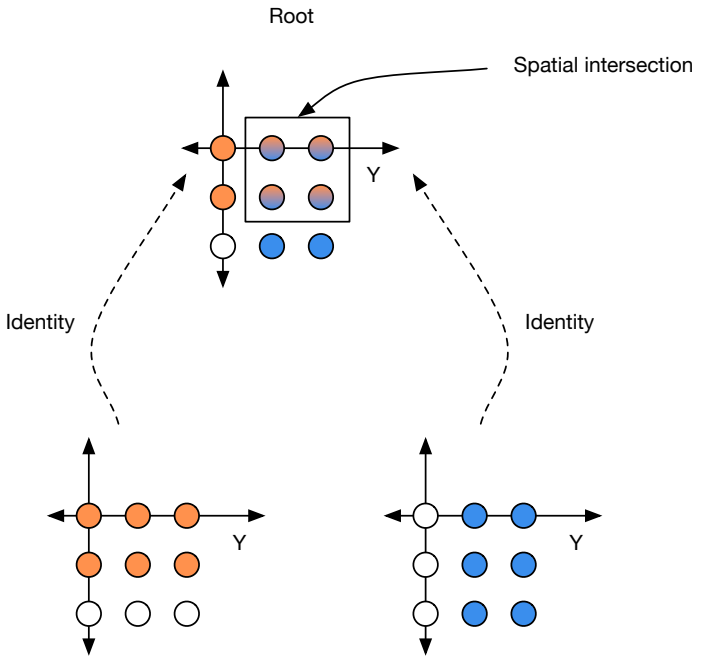


Figure 4.9: A spatial intersection. The combined blue/orange points in the root represent the intersection of the points mapped to from the child reference spaces.

The downstream mapping function requires two different definitions, starting with the downstream hull given in Definition 4.3.4.

Definition 4.3.4 (Downstream hull) *Let $\mathbb{R}^N_c$ be a reference space with definition $\mathbb{R}^N_c = (\mathbb{R}^M_p, \vec{p}, \vec{o}, \vec{r}, \vec{c})$. Let $S_{\vec{x}}$ be a set of points where for each $\vec{x} \in S_{\vec{x}}$, $\vec{x}$ is a point in $\mathbb{R}^M_p$. Let P be a partial permutation matrix defined on $\mathbb{R}^N_c$ and $\mathbb{R}^M_p$.*
A downstream hull $\delta^{\downarrow}(S_{\vec{x}})$ maps the set $S_{\vec{x}}$ from $\mathbb{R}^M_p$ to $\mathbb{R}^N_c$. Compute the downstream hull as follows:

$$\delta^{\downarrow}(S_{\vec{x}}) = \bigcup_{\vec{x} \in S_{\vec{x}}} \left\{ \vec{d} + \vec{i} \mid \text{where for each } j \in [0, N),\ d_j = \left\lfloor \frac{t_j r_j}{c_j} \right\rfloor \text{ and } \vec{t} = P(\vec{x} - \vec{o}) \text{ and } i_j \in [0, r_j) \right\}.$$

The downstream hull is analogous to the upstream mapping function given in Definition 4.3.2, except that the interval comes from refinement instead of coarsening. This function is referred to as a hull because it computes a superset of the possible points since it does not account for the issue with projections

(meaning it will project points to a single point in the child even if the projection is spatially unsound). The downstream mapping function given in Definition 4.3.5 includes the logic necessary to adjust for this scenario.

Definition 4.3.5 (Downstream mapping) *Let $\mathbb{R}_c^N$ be a reference space with definition $\mathbb{R}_c^N = (\mathbb{R}_p^M, \vec{p}, \vec{o}, \vec{r}, \vec{c})$ in a trie with root $\mathbb{U}^L$ and $S_{\vec{x}}$ be a set of points where for each $\vec{x} \in S_{\vec{x}}$, $\vec{x} \in \mathbb{R}_p^M$.*

Let $\Delta^{\downarrow}(S)$ represent the recursive application of $\delta^{\downarrow}(S)$ from $\mathbb{U}^L$ to $\mathbb{R}_c^N$. A downstream mapping $\phi^{\downarrow}(S_{\vec{x}})$ maps the set $S_{\vec{x}}$ from $\mathbb{R}_p^M$ to $\mathbb{R}_c^N$ while accounting for invalid spatial projections. Compute the downstream mapping as follows:

$$\phi^{\downarrow}(S_{\vec{x}}) = \Delta^{\downarrow}(\delta^{\downarrow}(S_{\vec{x}}) \cap^s S_{\vec{x}}).$$

The downstream mapping function in Definition 4.3.5 utilizes the spatial intersection and downstream hull functions in order to map only those points from parent-to-child that have a spatially valid projection. The term $\delta^{\downarrow}(S_{\vec{x}})$ first computes the downstream hull assuming that all points in $S_{\vec{x}}$ have valid locations in the child. The intersection $\delta^{\downarrow}(S_{\vec{x}}) \cap^s S_{\vec{x}}$ intersects the location of those mapped points with the original points in $S_{\vec{x}}$. The points that intersect in the root have locations that overlap, meaning they have a valid location in the child reference space thus can be mapped back down from the root, which is what the final $\Delta^{\downarrow}$ application performs.

For bijective mappings, no projections can occur, so the downstream hull and downstream mapping functions are the same and can be written as follows:

$$\phi^{\downarrow}(S_{\vec{x}}) = \delta^{\downarrow}(S_{\vec{x}}) = \bigcup_{\vec{x} \in S_{\vec{x}}} P(\vec{x} - \vec{o}).$$

For embedded mappings, the downstream hull and downstream mapping become:

$$\delta^{\downarrow}(S_{\vec{x}}) = \bigcup_{\vec{x} \in S_{\vec{x}}} P(\vec{x} - \vec{o})$$
$$\phi^{\uparrow}(S_{\vec{x}}) = \Delta^{\downarrow}(\delta^{\downarrow}(S_{\vec{x}}) \cap^s S_{\vec{x}}).$$

Together, these definitions provide all the core functionality necessary to move points all throughout the trie (up, down, and laterally) and compare locations.

4.4 Adding Tensors

A tensor reference space provides the final component necessary for UniTe and simply bounds a set of points within a reference space, which defines the *tensor*. The bounds can be represented by an extent in each dimension corresponding to a simple linear constraint that creates a hyperrectangular shape. The primary effect of bounding the points is that tensor reference spaces are finite, thus the operations defined earlier only apply to points within the bounds of the tensor[5].

5: The next chapter abuses this slightly with the introduction of locality accesses, but gives a well-defined result for what it means to access points (data) outside the bounds of a tensor.

Definition 4.4.1 gives the definition of a tensor reference space, which builds off of the reference space definition from 4.2.1. Tensors are built out more in the next chapter with the introduction of blocks and views.

Definition 4.4.1 (Tensor reference space) *A tensor reference space, $\mathbb{T}^N$, is an N-dimensional reference space with an associated extent defined by the tuple $(\mathbb{R}^N, \vec{e})$, where*

$\mathbb{R}^N$ is the associated reference space of $\mathbb{T}^N$ and
$\vec{e} = (e_0, \ldots, e_{N-1})$ represents the extent of $\mathbb{T}^N$ with respect to $\mathbb{R}^N$ ($e_i \geq 1$).

Ample prior work exists in the area of tensors, particularly in how to represent them within a program. UniTe's representation sits at a lower-level than much of the work since it largely deals with unconstrained sets of points, but the addition of bounds for tensors moves UniTe towards other representations such as the polyhedral model [46–48]. While UniTe does not consider these models directly (as it is not necessary for the implementations described later in this dissertation), they could prove useful when extending UniTe for other domains that have more complex usages of tensors (or have more shapes than just hypperectangular–for example, more general rectilinear shapes).

4.5 Summary

This chapter introduced the fundamental abstraction for capturing and reasoning about spatial relationships, UniTe. At its core, UniTe defines reference spaces which create different frames of reference for describing the location of points. Child reference spaces are defined relative to their parent, creating a trie structure where reference spaces can differ in their dimensionality, axes layout, origin, refinement, and coarsening. UniTe provides reference space mappings to quantify the relationships between the parent and child reference spaces, as well as mappings between points in difference spaces, ultimately making it possible to compare the location of points or sets of points (including tensors) across different spaces in the trie.

The next chapter builds some higher-order components on top of UniTe and begins to look at how to map these tensors to data. Chapter 6 shows how to apply UniTe to various kernels within JPEG and H.264, while Chapters 7 to 9 describe different approaches to implementing the abstraction.

Extending UniTe: UniTeX 5

While the last chapter provided the necessary foundations for connecting location with sets of points and tensors, it provides relatively low-level operations that do not take into account the actual data tied to a tensor. This chapter extends UniTe with a set of higher-order tensors and operations that focus on the data aspect, providing ways to create, share, and access the underlying data. The implementations `CoLa` and `SHiM` (introduced in Chapters 8 and 9, respectively) build off of this extension, referred to as UniTe eXtended, or UniTeX. While these later chapters use UniTeX in the context of block-based compression, the material in this chapter is still general enough to support more than just that, so it is presented independent of any particular domain.

5.1 Extending Tensors

UniTeX splits the tensor reference space of UniTe into two separate categories: block tensor reference spaces (*blocks*) and view tensor reference spaces (*views*). From an implementation point-of-view, blocks correspond to multidimensional arrays that have their own data storage, while views represent lightweight tensors that reference existing blocks. Thus a block can exist on its own, but a view cannot. This is a necessary detail to avoid unnecessary copying of data in an implementation, but also provides a more intuitive data representation. This chapter does not provide any specific operation for creating an initial block from scratch and instead leaves it to the implementation.

Both views and blocks maintain mappings to their parents as with the tensor spaces of UniTe, but views also maintain an implicit mapping to the block that they reference, which is referred to as the *nearest block*. A view and its nearest block can be arbitrarily far apart, but will always maintain an ancestor-descendant relationship, with the block representing the ancestor and the view representing the descendant. Sticking with the trie representation, this can be shown with additional upwards links in the trie as in Figure 5.1 on Page 60.

By themselves, blocks and views are not a novel construct and appear in many existing languages and libraries such as Julia and NumPy (see Chapter 10 for more examples of view-centric systems). However, UniTeX defines several higher-order operations on top of blocks and views that are either novel (such as colocation) or easier to reason about using UniTe (such as locality access). The next section defines a selection of these operations, and the next chapter looks at how to utilize many of these operations within block-based compression.

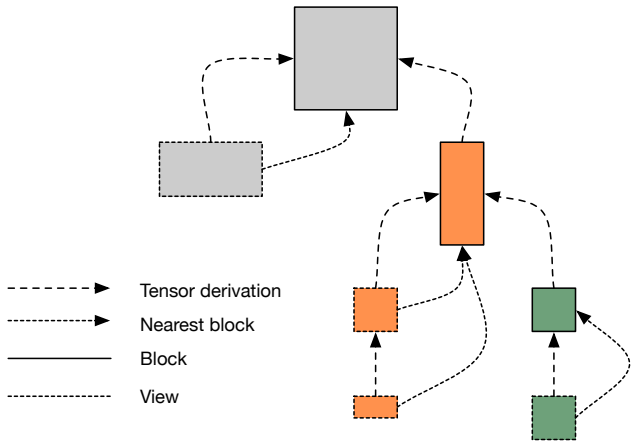


Figure 5.1: A trie containing blocks and views. Both blocks and views point to their parent as with the trie representations for reference spaces. However, views also point to their nearest block, which contains the data they reference.

5.2 Extending UniTe Operations

Using blocks and views opens the door for a variety of other operations based on UniTe that can be used to access data within these tensors, as well as to derive new blocks and views. Technically, any operations that conform to UniTe are allowed, however this section focuses on 10 particular operations listed in Table 5.1.

Table 5.1: Operations defined in UniTeX. b and v represent a block and view, respectively, while T represents a tensor which can be a block or a view.

Operation	Syntax	UniTeX Section
Block copy	`b = block_copy(T)`	5.2.1
Virtual permute	`v = permute(T)`	5.2.2
Virtual slice	`v = T / k`	5.2.2
Virtual refine	`v = refine(T,F)`	5.2.2
Virtual coarsen	`v = coarsen(T,F)`	5.2.2
Partition	`v = T[x0:y0:z0,...,zN:yN:zN]`	5.2.3
Colocation	`v = T0[T1]`	5.2.4
Coverage	`v = cov(x,T0,T1)`	5.2.5
Read	`val = T(i0,...,iN)`	5.2.6
Write	`T(i0,...,iN) = val`	5.2.6

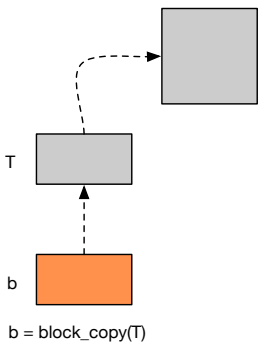


Figure 5.2: A block copy operation. Block copy creates a new block (orange) from an existing tensor (grey), with the existing tensor becoming the parent and the new block becoming the child. The new block has the same absolute parameterization as the parent.

5.2.1 Block Copy

Syntax: `b = block_copy(T)`

Block copy is the simplest operation and creates a new block, `b`, with the same parameterization as tensor `T`. `b` becomes a child of `T` as well. This operation provides a simple way to create a new block with the same location as another block, but pointing to a different set of data. For example, `b` and `T` could both represent the same frame within a video, but one storing the color version and the other storing the greyscale version.

5.2.2 Virtual Permute, Slice, Refine, and Coarsen

Syntax: `v = permute(T,F)`
Syntax: `v = T / K`
Syntax: `v = refine(T,F)`
Syntax: `v = coarsen(T,F)`

These operations function just like their counterparts in UniTe, but return new views from each. This allows changing the representation of underlying data without having to actually copy it. Figure 5.3 highlights each of these operations. If a new block is required after one of these operations, a block copy can simply be used on the view to create a new block.

5.2.3 Partition

Syntax: `v = T[x0:y0:z0,...,xN:yN:zN]`

The partition operation combines the UniTe translate and coarsen operations together, allowing a user to extract a segment of a tensor given a new origin (the `x` parameters), a stopping point in each dimension (the `y` parameters), and a coarsening factor (the `z` parameters). This operation returns a view and functions like array slice operations in existing tools like Python[1]. The left side of Figure 5.4 on Page 62 shows an example of a partition operation.

1: When only coarsening is used, it is effectively the same as striding, which is the term more commonly associated with array slicing.

Note that the view created from partitioning does not necessarily have to reference data within the bounds of the original partitioned tensor. A concern within an actual implementation would be making sure that accesses to the underlying array are in-bounds, however, a partition does not actually access any data. Thus, a partition can be used to create arbitrary views over any location, regardless of whether data actually exists for that view, such as shown on the right side of Figure 5.4. Of course, accessing such a view requires handling cases that may result in an out-of-bounds access on an underlying array. This is discussed

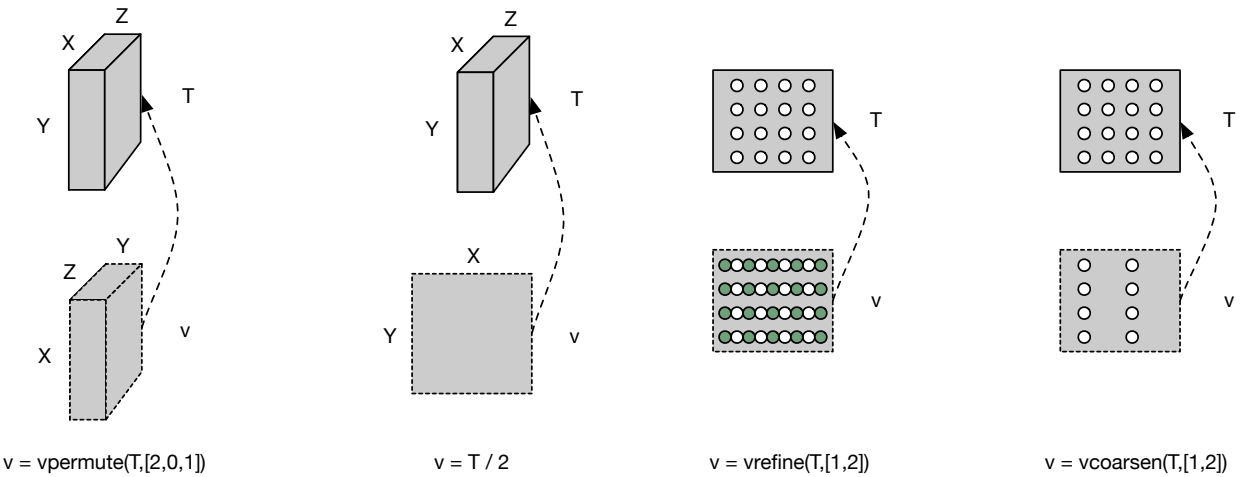


Figure 5.3: From left-to-right, a permutation, slice, virtual refinement, and virtual coarsening operation. Each creates a new view from an existing tensor, with the existing tensor becoming the parent and the new view becoming the child.

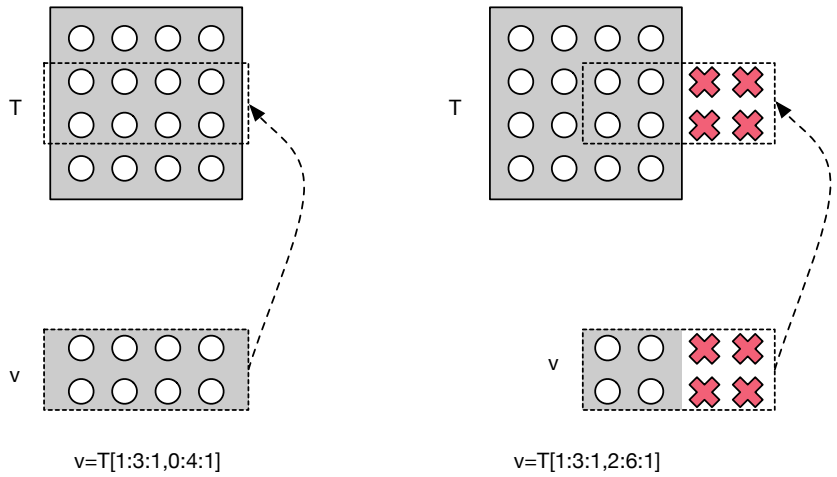


Figure 5.4: Partition operations on tensors. The left side shows a partition that points to a location fully within the parent tensor. The right side shows a partition that points outside the parent tensor. In this case, the red Xs may or may not correspond to valid data (see locality access in Subsection 5.2.6), but since a view does not access data, this is perfectly valid.

shortly in the context of locality access, which differentiates between data and location with regards to what "out-of-bounds" means.

5.2.4 Colocation

Syntax: `v = T0[T1]`

The colocation operation maps a tensor from one tensor space (the source) to another (the destination). This creates a view relative to the destination corresponding to the location of the source[2]. There are many possible ways to parameterize the resulting view in regards to whether the parameters of the source or destination should be used. For example, say the source and destination define different permutations on their axes. Some domains may prefer the new view preserve the source permutation, while others may want the destination permutation. UniTeX itself does not constrain how to combine the parameters[3], however, the implementations introduced in Chapters 8 and 9 use the source parameterization for the new view. Figure 5.5 on Page 63 shows two examples of colocation, one using the source's parameterization and the other using the destination's parameterization.

Like with partitions, a view created from colocation does not have to reference data within the bounds of the destination tensor since it does not actually access the underlying data.

2: Another way to think about colocation is that it lets you index a tensor with another tensor.

3: The only constraint is that the views are valid with respect to UniTe. For example, colocation cannot produce a view that has greater dimensionality than the parent, which would violate the semantics of reference spaces.

5.2.5 Coverage

Syntax: `v = cov(x,T0,T1)`

At a high-level, the coverage operation returns a view that contains all the points in a tensor `T0` that overlap the location of the point `x` with respect to another tensor `T1`. This operation is useful when refinement and/or coarsening are applied since they create multiple points sharing the same location. However, depending on the amount of refinement and/or coarsening applied, coverage can return different views, hence the need for the addition tensor `T1`, which specifies how "far" to walk

4: `T1` can be anywhere in the trie; it does not necessarily need to be in the same path as `T0`.

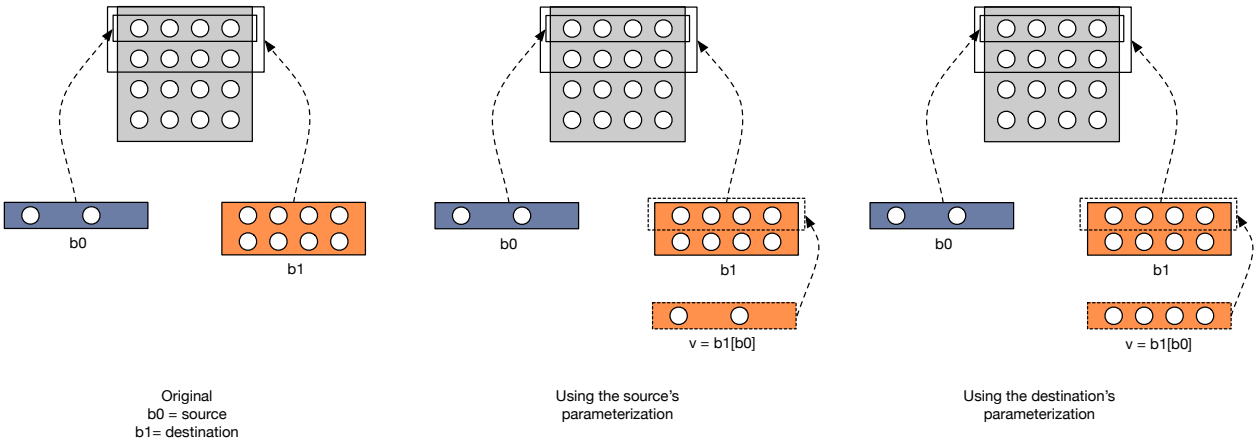


Figure 5.5: Two possible versions of colocation using the source's parameterization (middle) and the destination's (right). The left gives the initial structure with the left block pointing to the first row of the parent and applying a horizontal coarsening by two. The right block simply points to the first two rows of the parent. Using the source's parameterization effectively creates a virtual coarsening on `b1`. Note that `b0`, `b1`, and the parent could also be views; this example just sticks with blocks to reduce clutter in the image.

the trie in order to compute the coverage[4]. Figure 5.6 shows various examples of how varying `T1` relative to `T0` impacts the returned view for a series of 1D reference spaces.

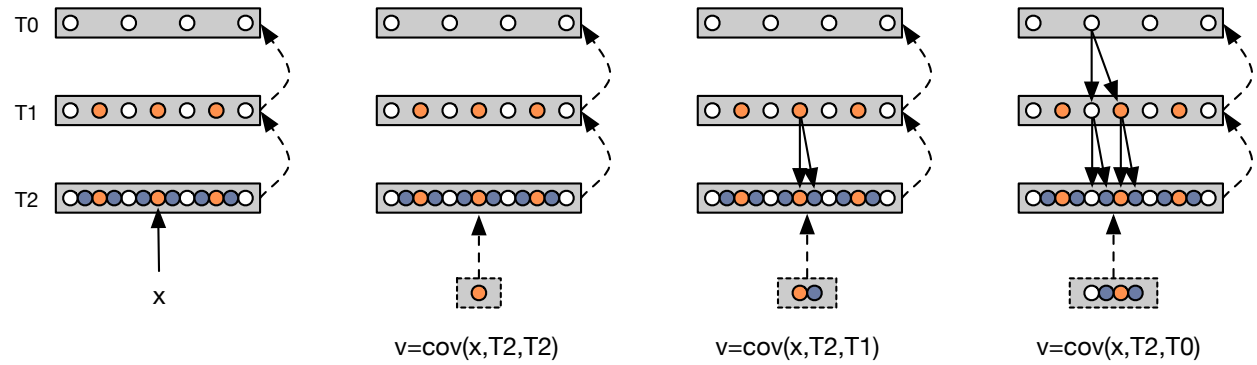


Figure 5.6: Different versions of the coverage operation showing how context effects the resulting view. The left figure gives the initial state and the others show coverage with respect to different reference spaces (in general, they do not need to be ancestors). Since these reference spaces are refined relative to one another, more points are included in the resulting view as coverage moves up the trie. The downward arrows show how each point within each reference space maps from the parent to the child.

5.2.6 Data Access

Syntax: `val = T(i0,...,iN)` (read)
Syntax: `T(i0,...,iN) = val` (write)

The data access operations are necessary for reading from and writing to data in tensors[5]. When the accessed location maps to an index that is in-bounds with respect to the tensor and corresponding underlying memory, the read and write operations are straightforward and just directly access the data at that index. In cases where the locations accessed are outside the bounds of the tensor, or even outside the bounds of the memory, UniTeX differentiates between location out-of-bounds and memory out-of-bounds.

5: Reading from or writing to a view propagates to the view's nearest block.

First, consider a simple example with a single 4x4 block as shown in Figure 5.7. In an implementation, this block would correspond to an underlying memory allocation with enough space to hold at least 16 data elements. A read (or write) specified with 2D indices (Y, X) points to valid data in the memory if the indices are anywhere in the range $(Y, X) : 0 \leq Y < 4$ and $0 \leq X < 4$. The access in the diagram at (2,1) is valid since it points to a valid memory location, while the index at (-1,1) is invalid. This is hardly surprising, but the key with this is how implementations handle such an "invalid" access on an array. Implementations range the entire spectrum for this, with languages like C not performing any checks (leaving the door open for segmentation faults), and others like NumPy letting the users choose from a variety of options.

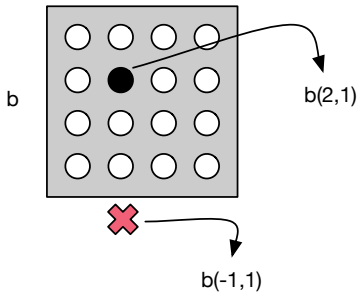


Figure 5.7: In-bounds and out-of-bounds accesses on a block.

For example, in NumPy, users can choose from throwing an error, clipping the index, or wrapping the index. Other libraries, such as ones for image processing like JuliaImages [49] or Halide [50], also provide options to return default values for out-of-bounds accesses, which are helpful for dealing with accesses at the edge of an image. This is all perfectly reasonable for accesses on independent tensors/arrays; however, UniTeX exploits spatial relationships across tensors to to provide a larger set of semantics (referred to as *locality access* semantics) that handle (and make it easier to reason about) these "out-of-bounds" accesses.

The left side of Figure 5.8 shows a straightforward example of a locality access on a view where the access refers to data in the row below the view. However, this particular data has a corresponding location within the block that the view points to, thus UniTeX supports returning that point in the block. The key in this example is that there is a difference between the index itself being out-of-bounds, and the location being out-of-bounds, meaning an out-of-bounds index can still point to an in-bounds location.

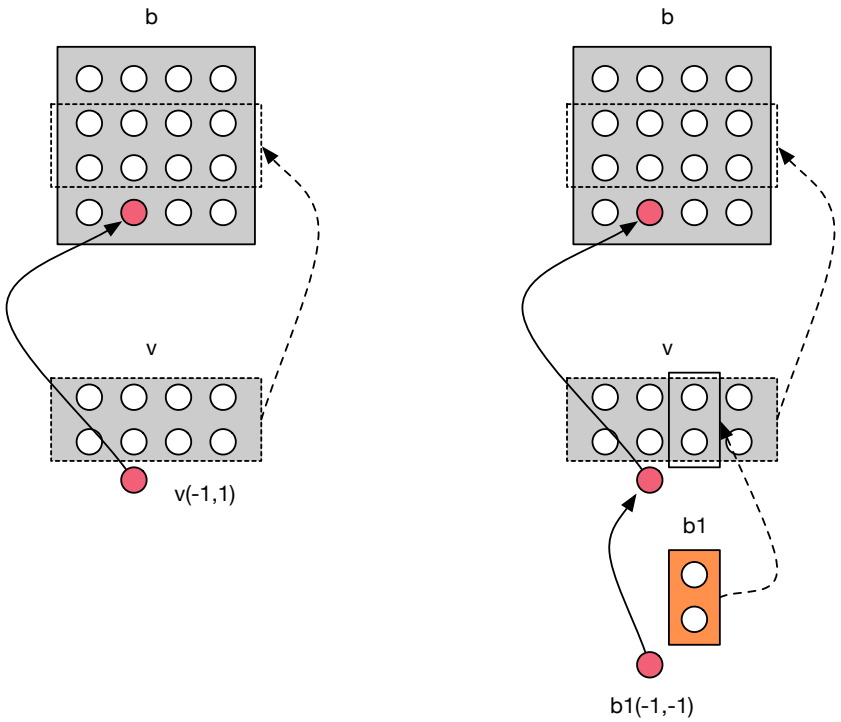


Figure 5.8: Locality accesses on a view (left) and block (right). Both support walking up the trie until finding data that corresponds to the initial out-of-bounds location accessed.

At some point, a locality access may result in accessing a location that is outside of a block as well. In this scenario, two options are available: 1) utilize an existing approach such as throwing an error or clipping the index, 2) continue walking up the trie. The first option is easy and just requires checking whether the final index is within the bounds of the block or not. The second option treats the block like a view and walks up the trie until an access within the bounds of a block is encountered[6]. The right side of Figure 5.8 shows an example of locality access on a block. This type of access is only possible due to the fact that UniTe can represent the location of both blocks *and* views.

6: Of course, if an index results in a truly out-of-bounds location that does not exist in any block, then an implementation would have to handle that and either throw an error or adjust the index accordingly.

5.3 Summary

This chapter gave a look at higher-order structures and operations that can be defined by extending the core principles of UniTe. This extension, UniTeX, separates tensors into blocks and views, where blocks correspond to a new multidimensional array and views reference an existing block. While blocks and views themselves are not a new idea, UniTeX builds several operations on top of them that exploit location information. In particular, UniTeX defines the colocation and locality access operations, which are only possible due to the fact that all tensors have locations that can be related to one another within a single universe.

The next chapter shifts the focus to block-based compression and looks at how UniTe and UniTeX provide intuitive ways to represent and access data within several kernels from JPEG and H.264. Then Chapters 7 to 9 look at implementing UniTe and UniTeX through the use of domain-specific languages.

Applying UniTeX to JPEG and H.264

6

UniTe and its extension UniTeX define themselves independent from any particular application, presenting a broad and flexible set of components that can be used in many domains. This chapter provides a concrete look at how these components apply in the context of block-based compression, utilizing kernels from various stages in JPEG and H.264 as introduced in Chapter 2. The examples provided in this chapter highlight use cases of UniTeX as applied to block-based compression, and also serve to highlight some various features of these kernels that are not directly related to the application, but are important for designing an implementation (discussed more in Chapters 7 to 9).

Note that the kernels described in this chapter represent an "idealized" view of an encoder, leaving out some finer algorithmic details that are not relevant to the abstractions. Also ignored are details such as how to implement the underlying data structures and operations. Chapters 8 and 9 dive more into these implementation details, as well as provide concrete examples of how the pseudocode in this chapter translates to an actual program. However, this chapter does point out some details that should be considered when choosing an implementation. Chapter 7 explores these considerations in more detail.

6.1 Syntax

All the UniTeX operations used in the pseudocode of this chapter follow from the syntax introduced in the prior chapter, which is reintroduced in Table 6.1. In addition, all the lines in the pseudocode that correspond to UniTeX operations have their line numbers highlighted in red.

Operation	Syntax	UniTeX Section
Block copy	`b = block_copy(T)`	5.2.1
Virtual permutation	`v = permute(T)`	5.2.2
Slice	`v = T / k`	5.2.2
Virtual refinement	`v = refine(T,F)`	5.2.2
Virtual coarsening	`v = coarsen(T,F)`	5.2.2
Partitioning	`v = T[x0:y0:z0,...,zN:yN:zN]`	5.2.3
Colocation	`v = T0[T1]`	5.2.4
Read	`val = T(i0,...,iN)`	5.2.6
Write	`T(i0,...,iN) = val`	5.2.6

Table 6.1: Pseudocode syntax used in the examples of this chapter. b and v represent a block and view, respectively, while T represents a tensor which can be a block or a view.

6.2 JPEG

This section focuses on four different kernels within the JPEG encoder: the primary encoding loop (Subsection 6.2.1), color transformation (Subsection 6.2.2),

the DCT (Subsection 6.2.3), and entropy coding (Subsection 6.2.4). Together, these serve as a gentle introduction to some basic usages UniTeX.

6.2.1 Primary Encoding Loop

The pseudocode and diagram in Figure 6.1 show the structure of the main encoding loop in JPEG. At a high-level, these two functions shown are responsible for getting the initial input data (Lines 1 to 7) and dispatching it throughout the various stages for compression. Line 12 initially partitions the input into 8x8 views and creates a block (Line 13) with the same location as the 8x8 view. This block (`ycbcr`) holds the result of converting the input from RGB to YCbCr format (Line 14). Then the function partitions `ycbcr` into views representing its three constituent color planes (Lines 15 to 17), which are passed to the remaining stages of compression. The read accesses on Lines 27 to 29 store the upper-left value of each processed 8x8 region, which is required for entropy coding.

Why this example

While straightforward, this example highlights an important benefit of data layout transformations through permutations. The initial `start` function begins by taking interleaved data and immediately applying a virtual permutation

```
1  def start(H,W)
2    if input_data is interleaved then
3      image = <read interleaved into block>
4      jpeg(permute(image,(2,0,1)))
5    else
6      image = <read planar into block>
7      jpeg(image)
8  def jpeg(image: t)
9    last_Y,last_Cb,last_Cr = 0
10   for y = 0 to h by 8 do
11     for x = 0 to w by 8 do
12       rgb = image[y:y+8:1,x:x+8:1,0:3:1]
13       ycbcr = block_copy(rgb)
14       colorTransform(rgb,ycbcr)
15       Y  = ycbcr[0:1:1,0:8:1,0:8:1]
16       Cb = ycbcr[1:2:1,0:8:1,0:8:1]
17       Cr = ycbcr[2:3:1,0:8:1,0:8:1]
18       dct(Y)
19       dct(Cb)
20       dct(Cr)
21       quantize(Y, luma_quant_table)
22       quantize(Cb, chroma_quant_table)
23       quantize(Cr, chroma_quant_table)
24       entropy(Y, last_Y)
25       entropy(Cb, last_Cb)
26       entropy(Cr, last_Cr)
27       last_Y = Y(0,0,0)
28       last_Cb = Cb(0,0,0)
29       last_Cr = Cr(0,0,0)
```

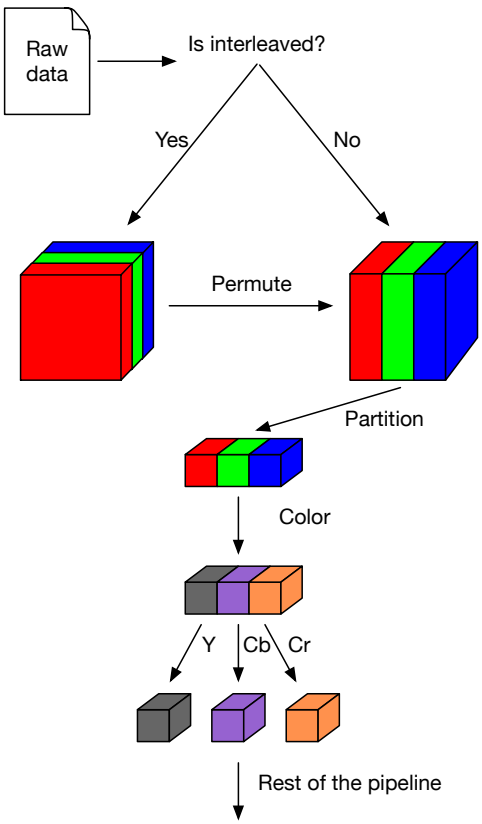


Figure 6.1: The main JPEG encoding loop. The `start` function takes in the raw data in either interleaved or planar format and then converts it to planar format so that the `jpeg` function can logically access data in planar format. The diagram depicts the high-level flow of these two functions.

to make it look like planar data. This makes it so the `jpeg` function only needs to consider planar input, as opposed to having to deal with both planar and interleaved. The permutation functions allow abstracting this sort of data layout from the get-go, such that only the top-level functions have to actually worry about handling the different layouts. The same idea applies with operations like coarsening and refinement. The further these operations are pushed upwards, the less specialized the inner functions needs to be, which helps simplify the code.

6.2.2 Color Transformation

Figure 6.2 shows the pseudocode for color conversion. Following from the prior code, this code expects that `rgb` and `ycbcr` are both logically represented in planar format. The main operations used here are basic in-bounds accesses for reads and writes (Lines 5, 7 and 9).

Why this example

The reads and writes in the color conversion code all represent elementwise accesses on tensors, which are extremely prevalent throughout many of the stages of block-based compression. The implementations introduced in Chapter 8 and Chapter 9 provide a simple syntax for performing elementwise operations across tensors without the need for explicit loop nests.

```
1 def colorTransform(rgb: t, ycbcr: t)
2   for y = 0 to 8 do
3     for x = 0 to 8 do
4       # Compute Y component
5       ycbcr(0,y,x) = rgb(0,y,x)*0.299+rgb(1,y,x)*0.587+rgb(2,y,x)*0.114;
6       # Compute Cb component
7       ycbcr(1,y,x) = rgb(0,y,x)*-0.168736+rgb(1,y,x)*-0.33126+rgb(2,y,x)*0.50000+128;
8       # Compute Cr component
9       ycbcr(2,y,x) = rgb(0,y,x)*0.5+rgb(1,y,x)*-0.418688+rgb(2,y,x)*-0.081312+128;
```

Figure 6.2: Pseudocode for the RGB to YCbCr color transformation stage. Like the `jpeg` function shown prior in Figure 6.1, this function assumes that the inputs are logically in planar format, even though the underlying data may actually have an interleaved layout.

6.2.3 DCT

Figure 6.3 on Page 70 shows parts of two different versions of DCT functions, which mainly involve partitioning, reads, and writes. Both functions apply the 2D DCT as separate 1D transforms (Lines 11 and 12 on the left and Lines 20 and 21 on the right), where the first transform computes the 1D DCT across the rows, and the second across the columns. Both apply slicing as well (Line 2 on the left and Line 11 on the right) in order to access plane as a 2D object instead of a 3D object[1]. However, the version on the left applies a permutation that transforms the columns to logically appear as rows, making it so the same 1D DCT kernel can be used for both the row and column transform[2]. The version on the right does not use a permutation, thus requires two separate functions, one for the rows and one for the columns, which only differ in the order of the Y

1: The left one could technically slice `plane` to 1D since the Y index is always zero.

2: In reality, there are some differences between scaling in the row and column 1D transforms. This can easily be handled by passing in the scaling factor as a parameter to the transform.

and X coordinate (compare Lines 5, 6, 8 and 9 on the left against Lines 14, 15, 17 and 18 on the right).

Why this example

Here, the use of a permutation helps abstract the algorithm (as opposed to just the data), making it invariant to whether the DCT is on a row or a column.

The slicing operation allows the plane to be accessed as a 2D tensor (which is what it logically represents), even though it is derived from a 3D tensor originally. This further separates the physical layout from the logical data layout.

```
1  def dct1d(plane: t)
2    plane2D = plane/2
3    for r = 0 to 8 do
4      vec = plane2D[r:r+1:1,0:8:1]
5      tmp0 = vec(0,0) + vec(0,7)
6      tmp7 = vec(0,0) - vec(0,7)
7      ...
8      vec(0,3) = scale(tmp6+z2+z3, SCALE)
9      vec(0,1) = scale(tmp7+z1+z4, SCALE)
10 def dctA(plane: t)
11   dct1d(plane)
12   dct1d(permute(plane,(0,2,1)))
```

```
1  def dct_row(plane: t)
2    plane2D = plane/2
3    for r = 0 to 8 do
4      vec = plane2D[r:r+1:1,0:8:1]
5      tmp0 = vec(0,0) + vec(0,7)
6      tmp7 = vec(0,0) - vec(0,7)
7      ...
8      vec(0,3) = scale(tmp6+z2+z3, SCALE)
9      vec(0,1) = scale(tmp7+z1+z4, SCALE)
10 def dct_col(plane: t)
11   plane2D = plane/2
12   for c = 0 to 8 do
13     vec = plane2D[0:8:1,c:c+1:1]
14     tmp0 = vec(0,0) + vec(7,0)
15     tmp7 = vec(0,0) - vec(7,0)
16     ...
17     vec(3,0) = scale(tmp6+z2+z3, SCALE)
18     vec(1,0) = scale(tmp7+z1+z4, SCALE)
19 def dctB(plane: t)
20   dct_row(plane)
21   dct_col(plane)
```

Figure 6.3: Pseudocode for part of the DCT transformation. The left and right versions of the DCT presented here use two individual 1D transformations that first go over the rows of the input, then the columns. The left DCT version uses permutations and slicing, making it possible to use the same 1D DCT code for both rows and columns by permuting the columns and making them accessible as rows. The right DCT shows the code without a permutation, which requires two separate functions, even though the accesses only differ slightly.

6.2.4 Entropy Coding

Figure 6.4 on Page 71 shows the high-level structure of Huffman entropy coding, focusing on the interleaving of the control flow structure and reads and writes on `plane` (Lines 2 and 8). JPEG utilizes a zigzag traversal for accessing the AC values, where the call to `zigzag` represents some operation (or table lookup) that transforms linear coordinates into the appropriate zigzag coordinate.

Why this example

While this part of the encoder is outside the scope of UniTeX, the implementations in Chapter 8 and Chapter 9 take special care to ensure that non-UniTeX operations easily interleave with UniTeX operations, ensuring that users are provided with a wide variety of features necessary to implement full encoders.

```
1  def entropy(plane: bv)
2    DC = plane(0,0,0)
3    if (...) then
4    while (...) do
5    if (...) then
6    for (i = ...) do
7      for (j = ...) do
8        AC = plane((0,zigzag(i,j)))
9        if (...) then
10       else
11         while (...) do
12         if (...) then
13         while (...) do
14   if (...) then
```

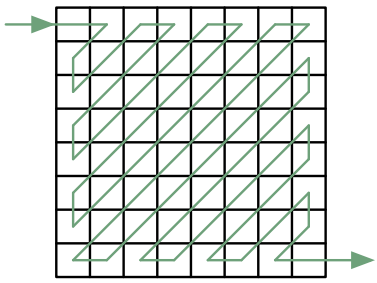


Figure 6.4: High-level structure of Huffman entropy coding in JPEG. This function interleaves tensor reads on `plane` with several pieces of other control flow. The zigzag traversal iterates through the plane using the ordering shown on the right.

This is necessary here to support all the control flow, especially the non-standard zigzag traversals. Most implementations represent the zigzag via an indirect lookup on a matrix that stores the zigzag coordinate at each row/column index corresponding to the linear coordinate.

6.3 H.264

The JPEG example showed several straightforward uses of operations like partition, slice, and tensor access. H.264 utilizes all of these operations, along with the rest of the UniTeX operations. Rather than include stages similar to JPEG, this section highlights a few examples in H.264 related to intra-prediction which have very different use cases from JPEG. This provides a broader view of how UniTeX is useful. There are many ways to define the underlying algorithms for these stages, however, this section opts for straightforward versions for presentation purposes, so this section may show some operations that would not necessarily occur in a real implementation (such as making extra copies of data).

6.3.1 Intra-Prediction Loop

Figure 6.5 on Page 72 shows the main control loop for 4x4 intra-prediction. H.264 also contains 16x16 and 8x8 forms of intra-prediction, but the overall idea is the same. This code can be broken down into four main segments: frame coarsening/refinement, frame partitioning, mode verification, and prediction. Lines 3 and 4 create a new tensor representing a coarsened and refined version of the original frame. This will be explained shortly. The partitions on Lines 10 and 15 extract macroblocks from the frame, then submacroblocks from the macroblocks, respectively. Prediction happens on each of these submacroblocks, and each submacroblock can be predicted independently. Since there are nine possible modes of prediction, Lines 17 to 19 creates nine blocks with the same location of the submacroblock that will store the individual predictions.

```
1  def predict(frame: t, rframe: t, H, W)
2    # block copy
3    coarsenedFrame = block_copy(coarsen(frame, (16,16)))
4    usedIntraPred = refine(coarsenedFrame, (16,16))
5    ...
6    intraLoop(frame, rframe, usedIntraPred, H, W)
7  def intraLoop(frame: t, rframe: t, usedIntraPred: t, H, W)
8    for r = 0 to H by 16
9      for c = 0 to W by 16
10       mblk = frame[r:r+16,c:c+16]
11       predMblk = block_copy(mblk)
12       # Partition into 4x4 submacroblocks
13       for r2 = 0 to 16 by 4
14         for c2 = 0 to 16 by 4
15           smblk = mblk[r2:r2+4:1,c2:c2+4:1]
16           # There are 9 modes
17           pred0 = block_copy(smblk)
18           ... other pred blocks ...
19           pred8 = block_copy(smblk)
20           # Initialize state
21           bestCost = MAX
22           bestMode = -1
23           bestPred = ∅
24           # Try each mode if possible
25           if (canRunMode0(frame,mblk,pred0,usedIntraPred)) then
26             mode0(pred0, rframe)
27             cost = sad(pred0,smblk)
28             if cost < bestCost then
29               bestCost = cost
30               bestMode = 0
31               bestPred = pred0
32           ... other modes ...
33           if (canRunMode8(frame,mblk,pred8,usedIntraPred)) then
34             mode8(pred8, rframe)
35             cost = sad(pred8,smblk)
36             if cost < bestCost then
37               bestCost = cost
38               bestMode = 8
39               bestPred = pred8
40           # Write the best prediction
41           for i = 0 to 4 do
42             for j = 0 to 4 do
43               predMblk[bestPred](i,j) = bestPred(i,j)
44   return bestCost,predMblk
```

Figure 6.5: The structure of the main control flow for 4x4 intra-prediction. This begins by creating a special coarsened version of the input frame (see Subsection 6.3.2 for more info). The outer loops create macroblocks and submacroblocks. The code at the inner level generates the necessary blocks for holding each possible type of prediction, then checks which modes can execute, and then runs the modes. It finishes by copying the best prediction back into another macroblock.

Not all nine modes of prediction are valid for every submacroblock, and depend on factors such as whether the submacroblock is at a frame edge or not. Lines 25 and 33 verify whether a mode is valid[3] through the `canRunModeX` functions, and then performs the actual prediction (`modeX`). Both of these will be explained shortly. Once the prediction is complete for the submacroblock (i.e. the best mode was picked), the final loop writes the best prediction for the submacroblock back to its corresponding location in the prediction macroblock via a colocation and write on Line 43.

3: These operations would exist for all nine modes even though only two are shown.

Why this example

Intra-prediction in H.264 is a very involved process, and this example serves to show all the different pieces that must come together in order to compute a single prediction on a submacroblock. The remaining H.264 examples in this section highlight some of the kernels within this main loop, so having this initial example provides some necessary context for those.

This example also highlights how deep tensors can go within an encoder. For example, `frame`, `rframe`, and `usedIntraPred` are created at the outer levels of the code, but passed all the way through down into the innermost prediction functions. Without UniTeX, this adds another layer of complexity to an implementation as locations cannot just be handled locally; often, they need to be tracked through all levels of the program. The depth of the tensors (submacroblocks in this case) also impacts performance as many tensors need to be created at the innermost levels of the program. The implementations given in Chapter 8 and Chapter 9 take this into account in their designs, ensuring that creating tensors does not introduce additional overhead within user code.

6.3.2 Frame Coarsening/Refinement

Lines 3 and 4 of Figure 6.5 on Page 72 use coarsening and refinement operators, along with a block copy, to generate a block (`coarsenedFrame`) that has 1/16 the size of the original frame (thus represents data at the macroblock level), but can be indexed at the pixel level (using `usedIntraPred`). Figure 6.6 provides a visualization and in-depth description of what this operation does. But to understand the purpose of this, it is first necessary to understand how the `usedIntraPred` view is utilized within the prediction.

Recall that H.264 prediction uses pixels from a reconstructed frame for the prediction, as opposed to the raw frame pixels. This reconstruction represents data that was already encoded (and decoded), which means it used either intra-prediction or inter-prediction. H.264 includes an option that constrains intra-prediction to only use reconstructed pixels that were also predicted via intra-prediction. If a particular intra-prediction mode accesses reconstructed data predicted with inter-prediction, that mode would be invalid and would not be used. As a result, determining what type of prediction was used requires being able to associate a flag with each pixel in the reconstructed data, so that intra-prediction can check that the pixels it needs are valid. The `usedIntraPred` tensor in this example is what stores those flags.

The most intuitive way to represent all the flags would be to create a tensor that is the same size as the frame, such that every point within it contains the flag. However, this is very wasteful as every pixel within a macroblock is predicted the same way. The better way to approach this is to store one flag per macroblock. This is optimal in terms of storage, but complicates accessing the data as it can no longer be accessed at the pixel level; it must be accessed at the macroblock level. However, being able to access at the pixel level is much more intuitive and fits in better with how this tensor is actually utilized. This is exactly what the code originally shown in Figure 6.5 does: it creates a block with macroblock-level

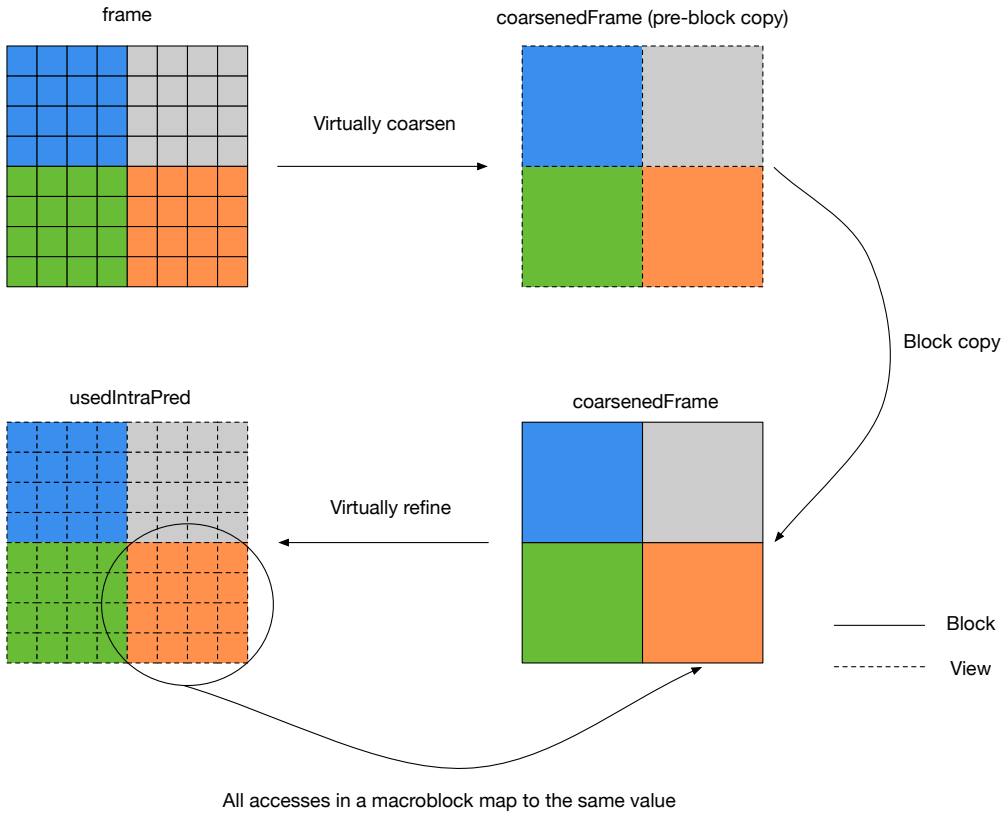


Figure 6.6: Applying virtual coarsening and refinement to `frame` from Figure 6.5. This example simplifies the representation slightly in order to fit within the diagram, and assumes the frame itself is 8x8, and macroblocks are 4x4. The colors indicate which parts of the tensors share the same location. The virtual coarsening operation creates a view over the 8x8 frame, coarsening it to the macroblock-level instead of the pixel level. Then the copy operation creates a new block. This new block effectively stores one value per macroblock as opposed to one value per pixel like the original frame. Taking a step further, virtual refinement allows making the coarsened block "look like" the original block in terms of how many points can be indexed within the refined view. Thus, it can be indexed as if it were an 8x8 frame, even though underneath it only stores 4x4 values (all the accesses map back to a single value in `usedIntraPred`, as shown by the curved left-to-right arrow).

granularity (Line 3), then creates a refined view (Line 4), which can be accessed at pixel-level granularity. The next example shows how to actually access the `usedIntraPred` tensor.

Why this example

Even though this example is small in terms of code, it shows a very practical use case that requires combining multiple UniTeX operations together (virtual coarsening, block copy, and virtual refinement) and maintaining the correct locations throughout. This general paradigm shows up constantly within encoders as well–this particular flag shown here is not the only use case. Without the abstraction, users would be required to manually handle all the mappings and parameters between the various tensors created.

Location of Reads and Writes for Coarsened and Refined View Access

For the purposes of compression, any reads or writes to a coarsened view map to a single point in the corresponding block. Other domains may want to map

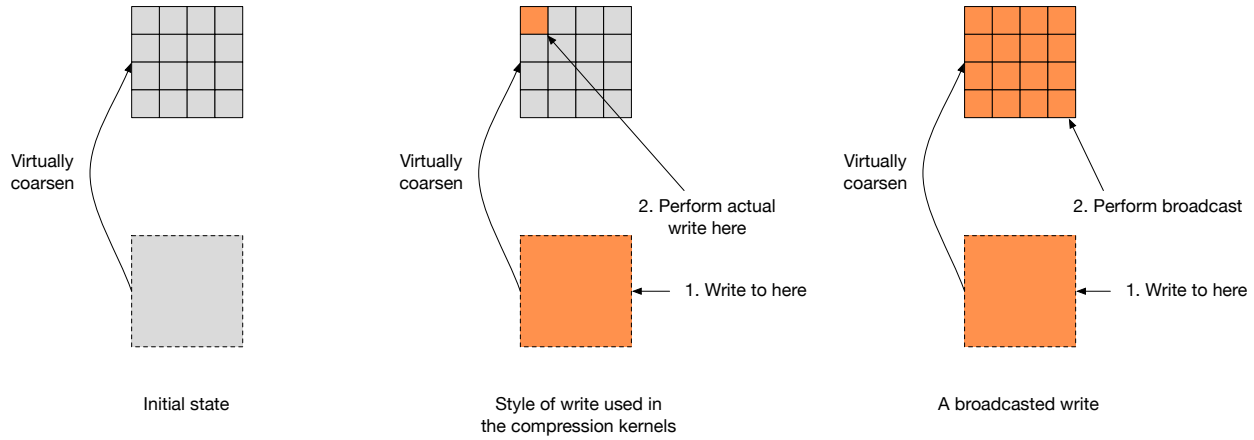


Figure 6.7: Writing to a coarsened view and accessing the corresponding data in the corresponding block. The left shows the initial state. The middle depicts how the kernels presented in this chapter (and the rest of the dissertation) assume that a write to a coarsened region maps to the upper-left pixel. The right shows another possible case where the write is broadcast to all pixels corresponding to the location of the coarsened point. Reads would operate in an analogous way, but could also support returning more complex results such as taking the average of all the points in the coarsened region (or min, max, etc.).

to different points, which can be done with UniTe and UniTeX since operations like coverage can be used to access all the points with the same location. This makes it possible to do something like take a write to a coarsened point in a view and broadcast it all the points that it covers with respect to the parent block. Figure 6.7 shows both cases for a write (a read would operate in a similar way). Refinement is a little different since refined points in the view all map to a single point in the corresponding block. However, coverage can again be used to have more control over which points in the block are accessed, but it is not necessary for these examples.

6.3.3 Mode Verification

To ensure that a particular intra-prediction mode is valid, an encoder needs to check that 1) the data accessed actually exists and 2) only reconstructed pixels predicted with intra-prediction are used. The first requirement involves checking whether or not the current submacroblock exists at the edge of a frame, which can be done by checking its origin relative to the frame. For example, Figure 6.8 on Page 76 shows code for verifying the vertical right mode of intra-prediction, which requires the row above and column to the left of the current submacroblock (`pred`). Thus, this mode cannot be used for a submacroblock that exists on the top or left frame edge. Lines 4 and 5 performs a locality access to get the row and column, then gets the origins[4] relative to the frame using colocation on Lines 6 and 7.

4: This also assumes the existence of some origin function which returns the origin parameter.

If the data does exist (Line 8), the next step checks for intra-prediction using the `usedIntraPred` tensor described earlier, using colocation to get a view representing the prediction submacroblock's location within `usedIntraPred`. Depending on the location of the submacroblock, up to three different macroblocks are adjacent to it, meaning three different points need to be checked (see the right side of Figure 6.8). Lines 11 to 13 check these three points, accessing `usedIntraPred`

```
1  def canDoModeVR(frame: t, mblk: t, pred: t,
2                  usedIntraPred: t)
3    # Make sure data exists
4    rowUp = pred[-1:0:1,0:4:1]
5    colLeft = pred[0:4:1,-1:0:1]
6    rowY,rowX = origin(frame[rowUp])
7    colY,colX = origin(frame[colLeft])
8    if rowY>0 && colX>0 then
9      # Check for intra-prediction
10     flags = usedIntraPred[pred]
11     upleft = flags(-1,-1) == INTRA
12     up = flags(-1,0) == INTRA
13     left = flags(0,-1) == INTRA
14     return left && up && left
15   else
16     return false
```

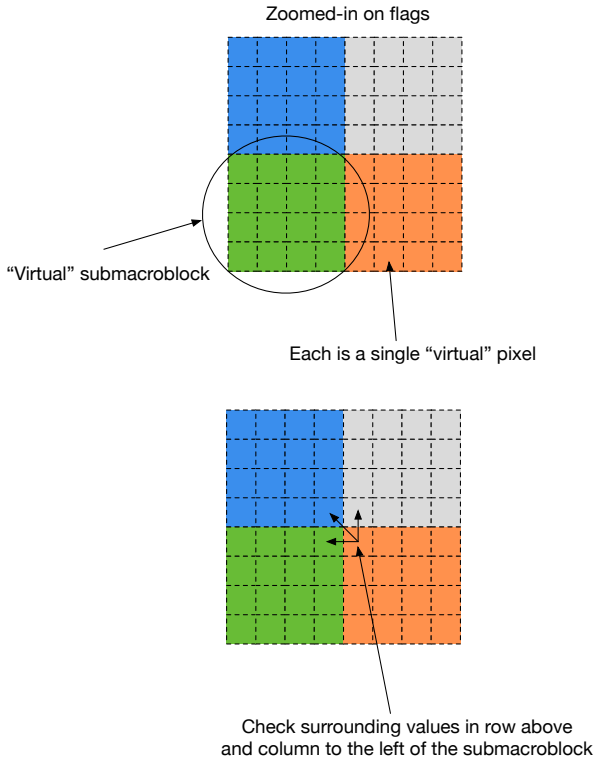


Figure 6.8: Checking that the reconstructed data needed for performing 4x4 vertical right intra-prediction exists and was also predicted via intra-prediction. The right side shows a zoomed-in view of `flags` (Line 10), showing colored regions that correspond to "virtual" submacroblocks and the individual squares representing "virtual" pixels (virtual meaning there are not actually values per submacroblock/pixel since this is virtually refined). With virtual refinement, it is possible to check the flag values at the pixel level, even though `flags` physically contains one value per macroblock, not per pixel. The abstraction hides all the low-level indexing details.

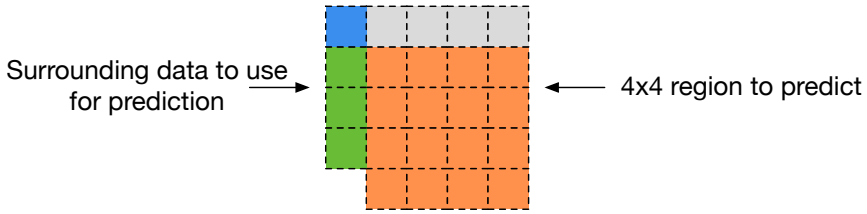
as if it had pixel-level granularity. If all points are predicted via intra-prediction, this mode can be used.

Why this example

While the earlier example showed how to construct the `usedIntraPred` tensor using virtual coarsening and refinement, this example shows one way to actually access it. The colocation operation abstracts away the mappings needed to find the location of the submacroblock, while still maintaining the virtual refinement of `usedIntraPred`.

6.3.4 Vertical Right Intra-Prediction

Figure 6.9 on Page 77 shows one mode of 4x4 intra-prediction referred to as vertical right intra-prediction. This code starts by using colocation (Line 4) to get the location in the reconstructed data (`rframe`). In this case, `pred` is a 4x4 block representing the current submacroblock, and is also used to hold the resulting prediction. The rest of the code relies on locality accesses to get the data in the row above and the column to the left of the submacroblock.




```
1  # Intra_4x4_Vertical_Right
2  def modeVR(pred: t, rframe: t)
3    # Map pred into rframe
4    p = rframe[pred]
5    for y = 0 to 4 do
6      for x = 0 to 4 do
7        zVR = 2*x-y
8        # Compute the prediction
9        if zVR == 0 || zVR == 2 || zVR == 4 || zVR == 6 then
10         pred(y,x) = (p(-1,x-(y>>1))+p(-1,x-(y>>1))+1)>>1
11       else if zVR == 1 || zVR == 3 || zVR == 5 then
12         pred(y,x) = (p(-1,x-(y>>1)-2)+2*p(-1,x-(y>>1)-1)+p(-1,x-(y>>1))+2)>>2
13       else if zVR == -1 then
14         pred(y,x) = (p(0,-1)+2*p(-1,-1)+p(-1,0)+2)>>2
15       else
16         pred(y,x) = (p(y-1,-1)+2*p(y-2,-1)+p(y-3,-1)+2)>>2
```

Figure 6.9: Vertical right 4x4 intra-prediction as defined in the H.264 standard [19]. The top image shows which of the pixels (green, blue, grey) are accessed relative to `pred` (orange). The code utilizes colocation (Line 4) and locality accesses (Lines 10, 12, 14 and 16) in order to perform the prediction, which accesses the points relative to the submacroblock `pred`.

Why this example

This example highlights the benefit of the indexing scheme used in UniTeX. Rather than follow indexing like in other programs such as NumPy (which may do something like wrap or clip an out-of-bounds index), the locality access semantics provide a very intuitive correspondence between how the standards define an operation and how to actually implement it.

6.4 Summary

The tensors and operations defined in UniTeX apply to many stages throughout JPEG and H.264, and this chapter provided a look at several such examples. While these examples are not the only usages of spatial relationships within JPEG and H.264, they serve as exemplars of each operation and provide a broad look at how UniTeX interleaves within the other processing for compression. It also pointed out different considerations to take into account when implementing these abstractions.

The next chapter shifts to the implementation side and discusses the pros and cons of different approaches to implementing UniTeX at a high level. Following that chapter are the two implementations, `CoLa` and `SHiM`, which provide a concrete look at how to effectively implement UniTeX for use with compression.

Implementation Considerations 7

The last chapters described the theory behind UniTe and UniTeX and looked at their application within several examples from JPEG and H.264, focusing on how the abstraction helps simplify various operations. However, it also alluded to some implementation considerations for both practicality and performance that are taken into account by the `CoLa` and `SHiM` implementations introduced in the next chapters. Both of these implementations are designed as domain-specific languages (DSL) to provide performance parity with existing implementations. Furthermore, they are also designed with flexibility in mind and provide users with access to a variety of features necessary for implementing the parts of compression that fall outside the domain of UniTeX. This chapter provides a bridge between the discussion of application and implementation and focuses on the reasons for selecting DSLs as opposed to another implementation strategy such as a library. In particular, it provides examples of some of the difficulties that come with current manual (Section 7.1) and library (Section 7.2) implementations, and then discusses how a DSL (Section 7.3) provides a middle-ground combining the benefits of a manual and library approach.

7.1 A Manual Approach

Due to the lack of existing work related to language support for encoders, implementations for block-based compression encoders are largely implemented from scratch, where each implementation chooses their own representation for tensors[1]. While this may be fine for small programs, these end-to-end encoders can contain hundreds of thousands of lines of code. Any additional implementation complexity on top of the complexity of the algorithms themselves opens up the door for bugs, particularly indexing bugs, and leads to unnecessary code obscurity.

1: Not just tensors, but really all of the data structures.

Take the code on the top of Figure 7.1 on Page 80 as an example, which comes from the reference implementation for H.264 (JM [8]). At first glance, it appears that this code is doing some rather complicated operation between the pointer objects due to the mixture of array indexing and pointer arithmetic. In reality, all this code is actually doing is 1) computing the residual (i.e. subtraction) between `cur_img` and `prd_img` and storing in `m7`, and 2) copying the prediction from `prd_img` to `cur_prd`. There is no reason this code needs to be this obscure for such a simple operation[2]. This highlights the need for some higher-level abstraction to provide a uniform way to access the data and avoid code like this, which is what the code on the bottom of Figure 7.1 shows. This is pseudocode using UniTeX operations, which hides the underlying indexing details and provides a clean interface for accessing the data in a logical 2D fashion.

2: A second copy of this function also exists, but with a different value for `BLOCK_SIZE`. Also, the `memcpy` is useless as the calling context for these functions does the copy itself, so this just duplicates unnecessary work.

Tensor representations also vary within a given implementation, sometimes using multiple representations within a single function. Figure 7.2 on Page 80 shows a collection of function signatures for Hadamard and DCT transforms in JM.

```
1  void generate_pred_error_4x4(imgpel **cur_img, imgpel **prd_img,
2                               imgpel **cur_prd, int **m7, int pic_opix_x,
3                               int block_x) {
4    int j, i, *m7_line;
5    imgpel *cur_line, *prd_line;
6    for (j = 0; j < BLOCK_SIZE; j++) {
7      m7_line = &m7[j][block_x];
8      cur_line = &cur_img[j][pic_opix_x];
9      prd_line = prd_img[j];
10     memcpy(&cur_prd[j][block_x], prd_line, BLOCK_SIZE * sizeof(imgpel));
11     for (i = 0; i < BLOCK_SIZE; i++) {
12       *m7_line++ = (int) (*cur_line++ - *prd_line++);
13     }
14   }
15 }
```

```
1  def generate_pred_error_unitex(cur_img: tensor, pred_img: tensor,
2                                 cur_pred: tensor, m7: tensor)
3    for i = 0 to BLOCK_SIZE do
4      for j = 0 to BLOCK_SIZE do
5        cur_prd[prd_img](i,j) = prd_img(i,j)
6        m7[prd_img](i,j) = cur_img[prd_img](i,j) - prd_img(i,j)
```

Figure 7.1: Code for computing the residual between `cur_img` and `prd_img`, as well as copying `prd_img` to its location in `cur_prd`. The code on the top comes from the H.264 reference implementation, JM [8], while the bottom shows the same operation using UniTeX primitives (see Section 6.1 for pseudocode syntax). The UniTeX operations makes the data access clear with its use of 2D tensors and simple indexing, while the JM code obscures the underlying access through 1D arrays and a mixture of array and pointer arithmetic.

```
1  void hadamard4x4(int **block, int **tblock);
2  void hadamard2x2(int **block, int tblock[4]);
3  void ihadamard2x2(int tblock[4], int block[4]);
4  void forward8x8(int **block, int **tblock, int pos_y, int pos_x);
```

Figure 7.2: Hadamard and transform function signatures in JM. Each of these operations functions on a 2D region of data, though JM varies in how it stores the data. In some cases, it stores the data as a pointer-to-pointer and in others it stores it as a 1D array. In the first three functions, the blocks store exactly the data needed for the function, while the last one stores the data within a larger block (like a view), hence the need for the offsets `pos_y` and `pos_x`.

This shows various ways the implementation represents tensors, all of which require a different way of accessing them. Having these different representations requires maintaining different location information for each, leading to additional complexity that should instead be captured by an abstraction.

Figure 7.3 on Page 81 provides one final example, this time from openh264 [10]. This code implements the 4x4 vertical right intra-prediction mode originally shown in Figure 6.9. This code is particularly interesting because parts of it mimic the locality access operation introduced in Subsection 5.2.6 (see the negative indices on Lines 9 to 16) for reads on the input data. However, these accesses are on a 1D array and do not match up to the standard in any intuitive manner since the standard defines the accesses on a 2D array.

Together, these different examples, along with the earlier ones shown throughout this dissertation, serve to give a taste of some of the issues that come with having to manually implement tensors and location within encoders. Complexity pops up in several different ways, from lack of uniform representations for the tensors themselves to variations in how they are accessed. Even though these issues may not pose an issue locally, when spread across an entire encoder, they introduce a considerable amount of bookkeeping to the implementation in order

```
#define LD64(a) (*((uint64_t*)(a)))
#define ST64(a, b) *((uint64_t*)(a)) = (b)
static inline void WelsFillingPred8x2to16_c (uint8_t* pPred, uint8_t* pSrc) {
  ST64 (pPred  , LD64 (pSrc));
  ST64 (pPred + 8, LD64 (pSrc + 8));
}
void WelsI4x4LumaPredVR_c (uint8_t* pPred, uint8_t* pRef, const int32_t kiStride) {
  const int32_t kiStridex2  = kiStride << 1;
  const uint8_t kuiLT = pRef[-kiStride - 1];
  const uint8_t kuiL0 = pRef[-1];
  const uint8_t kuiL1 = pRef[kiStride - 1];
  const uint8_t kuiL2 = pRef[kiStridex2 - 1];
  const uint8_t kuiT0 = pRef[-kiStride];
  const uint8_t kuiT1 = pRef[1 - kiStride];
  const uint8_t kuiT2 = pRef[2 - kiStride];
  const uint8_t kuiT3 = pRef[3 - kiStride];
  const uint8_t kuiVR0 = (1 + kuiLT + kuiT0) >> 1;
  const uint8_t kuiVR1 = (1 + kuiT0 + kuiT1) >> 1;
  const uint8_t kuiVR2 = (1 + kuiT1 + kuiT2) >> 1;
  const uint8_t kuiVR3 = (1 + kuiT2 + kuiT3) >> 1;
  const uint8_t kuiVR4 = (2 + kuiL0 + (kuiLT << 1) + kuiT0)>>2;
  const uint8_t kuiVR5 = (2 + kuiLT + (kuiT0 << 1) + kuiT1)>>2;
  const uint8_t kuiVR6 = (2 + kuiT0 + (kuiT1 << 1) + kuiT2)>>2;
  const uint8_t kuiVR7 = (2 + kuiT1 + (kuiT2 << 1) + kuiT3)>>2;
  const uint8_t kuiVR8 = (2 + kuiLT + (kuiL0 << 1) + kuiL1)>>2;
  const uint8_t kuiVR9 = (2 + kuiL0 + (kuiL1 << 1) + kuiL2)>>2;
  uiSrc[0] = uiSrc[9] = kuiVR0;
  uiSrc[1] = uiSrc[10] = kuiVR1;
  uiSrc[2] = uiSrc[11] = kuiVR2;
  uiSrc[3] = kuiVR3;
  uiSrc[4] = uiSrc[13] = kuiVR4;
  uiSrc[5] = uiSrc[14] = kuiVR5;
  uiSrc[6] = uiSrc[15] = kuiVR6;
  uiSrc[7] = kuiVR7;
  uiSrc[8] = kuiVR8;
  uiSrc[12] = kuiVR9;
  WelsFillingPred8x2to16 (pPred, uiSrc);
}
```

Let the variable zVR be set equal to 2 * x − y.

The values of the prediction samples $pred4x4_L[x, y]$, with x, y = 0..3, are derived as follows:

– If zVR is equal to 0, 2, 4, or 6,

$$pred4x4_L[x, y] = (p[x - (y >> 1) - 1, -1] + p[x - (y >> 1), -1] + 1) >> 1 \quad (8\text{-}57)$$

– Otherwise, if zVR is equal to 1, 3, or 5,

$$pred4x4_L[x, y] = (p[x - (y >> 1) - 2, -1] + 2 * p[x - (y >> 1) - 1, -1] + p[x - (y >> 1), -1] + 2) >> 2 \quad (8\text{-}58)$$

– Otherwise, if zVR is equal to −1,

$$pred4x4_L[x, y] = (p[-1, 0] + 2 * p[-1, -1] + p[0, -1] + 2) >> 2 \quad (8\text{-}59)$$

– Otherwise (zVR is equal to −2 or −3),

$$pred4x4_L[x, y] = (p[-1, y - 1] + 2 * p[-1, y - 2] + p[-1, y - 3] + 2) >> 2 \quad (8\text{-}60)$$

Figure 7.3: Code from openh264 for performing vertical right intra-prediction originally as shown in the standard (bottom). This code uses negative indexing, however it uses a 1D representation instead of 2D, so the accesses still do not match up with the standard.

to maintain the correct locations, which is why abstractions like UniTe and UniTeX become necessary. A library representation provides a straightforward way to implement the abstraction and provides users with a drop-in solution. However, the next section discusses the performance implications that come with a library, making it an impractical approach for implementing such an abstraction.

7.2 A Library Approach

UniTeX lends itself well to a library implementation that can provide blocks and views and expose a simple API with all the necessary operations. Many different languages can support such an implementation, making a library as close to a drop-in replacement for existing implementations as possible. Figure 7.4 shows an example of the color conversion operation from Subsection 2.1.3 using a simple C++ library called libUnite. While libUnite provides a straightforward interface that provides similar syntax to the pseudocode examples of UniTeX, it comes with severe performance overheads, rendering it effectively useless for any practical purpose.

Table 7.1 on Page 83 shows some results from implementing the vertical right mode of intra-prediction using libUnite, and a manual version, also in C++ (Figure 7.5 shows the same data in graph form). The manual version operates like the existing implementations shown already, directly passing around a multidimensional array and performing all the indexing directly using only the properties needed. libUnite implements blocks and views as C++ objects with methods for each operation, and stores all the property values in `std::array` objects. It does not do any specialization based on property values, thus relies on the backend compiler (clang 12 in this case) to attempt optimizations (such as removing the division by one in the event that no coarsening is required). As the table shows, clang largely fails to optimize the library version as it runs about 12× slower than the manual version in the best case.

There are two primary factors leading to this slowdown: 1) the additional indexing computations, and 2) where the indexing computations occur. In this particular

```
template <typename RGB_T, typename YCbCr_T>
void color(RGB_T &RGB, YCbCr_T &YCbCr) {
  for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 8; j++) {
      YCbCr[{0,i,j}] = (int)((double)(RGB(i,j,0))*0.299 +
                             (double)(RGB(i,j,1))*0.587 +
                             (double)(RGB(i,j,2))*0.114));
      YCbCr[{1,i,j}] = (int)((double)(RGB(i,j,0))*-0.168736 +
                             (double)(RGB(i,j,1))*-0.33126 +
                             (double)(RGB(i,j,2))*0.500002)+128);
      YCbCr[{2,i,j}] = (int)((double)(RGB(i,j,0))*0.5 +
                             (double)(RGB(i,j,1))*-0.418688 +
                             (double)(RGB(i,j,2))*-0.081312)+128);
    }
  }
}
```

Figure 7.4: A look at a C++ library implementation with libUnite for the color conversion function from JPEG. Utilizing operator overloading in C++ makes it simple to provide an intuitive interface for data access that hides all the low-level indexing details.

Clang opt	Manual (s)	libUnite (s)	libMin (s)
-O3	0.27	5.51	0.46
-O2	0.45	5.34	0.78
-O1	0.72	75.22	27.79

Table 7.1: Performance comparison between a manual and library implementation of prediction. The last column shows performance for a minimal (and largely impractical) library implementation that only tracks and computes with the extent and origin of the blocks and views.

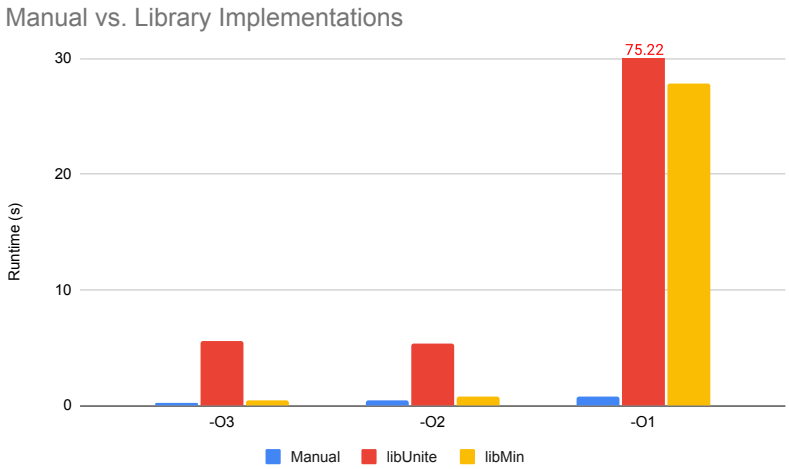


Figure 7.5: Chart for the raw runtime performance shown in Table 7.1.

instance, only the origin and the extent properties are really necessary for computing the indices. However, the library does not have knowledge of that, thus must take them into account in all of its indexing computations. Even though the values for coarsening and refinement are both one in all dimensions (the default value), clang fails to exploit this in its optimizations, leaving expensive divisions and multiplications by coarsening and refinement in the compiled code.

If the indexing operations occurred at the outer loop levels of the program, then some additional overhead could be handled as it would be amortized within the rest of the program runtime. However, many of these tensor operations happen at the inner levels of the loop nest, executing across every macroblock and submacroblock. This causes any additional indexing overhead to quickly add up.

The other library shown in the data, libMin, represents a minimal library implementation that only includes extent and origin within its computation, thus does not have to worry about the divisions and multiplications with coarsening and refinement, nor any permutations. This provides a sort of upper bound on the performance that can be achieved with a library of UniTeX since it only requires the minimal amount of computation. However, it still suffers some overhead, coming in at about 1.7× slower than the manual implementation. clang fails to remove some usages of the block and view objects themselves, leading to extra overhead in just accessing the underlying data through the object when compared to the manual implementation. All together, these results lead to the conclusion that a library version of UniTeX is not a suitable method of implementation.

7.3 A DSL Approach

With the issues stemming from a library implementation, a DSL offers a way to provide a library-like implementation for working with these tensor objects, while also providing the ability to remove the overheads associated with indexing. While the two DSLs introduced in the next chapters (`CoLa` and `SHiM`) differ in their style of implementation, they both focus on three different key requirements listed below:

- Provide an intuitive API for creating and operating on tensor data structures.
- Support control flow and other high-level non-UniTeX features.
- Remove the overhead associated with creating and indexing tensor objects.

Next discusses each requirement in more detail and provides the motivation behind why each is important.

7.3.1 Designing Tensor Data Structures

To provide as seamless a transition between existing implementations and UniTeX, both the `CoLa` and `SHiM` implementations strive to present a clear and intuitive syntax and API for building blocks and views and performing any operations on them.

Why this matters

In terms of syntax, users in other domains that already contain DSLs may be more willing to deal with new types of syntax and such. But, with compression, there are not any other DSLs out there for designing any part of encoders. Trying to convince developers to switch from using plain C/C++ or Python to some brand new language with unfamiliar syntax would ultimately be a losing battle. `CoLa` and `SHiM` build off of Python and C++, respectively, thus provide a familiar syntax to the user.

One glaring issue in existing implementations is the wide variety of ways the implementations represent and operate on multidimensional data due to the lack of a unified representation. A plain C or C++ representation offers the maximum flexibility since it just has an array, but then leads to this issue of multiple representations. On the other hand, a representation that only supports one type of tensor (i.e., only fixed size, only on the heap, etc.) creates a very specific representation, but cannot be tuned appropriately to the use case. `CoLa` and `SHiM` strike a balance between the two, providing several ways to define blocks and views based on use cases observed in implementations, but limiting how they can be accessed.

7.3.2 Control Flow and Non-UniTeX Features

As discussed in prior chapters, implementations for compression require other language features beyond just creating blocks and views and operating on them.

This requires a careful balance between choosing what should and what should not go into the DSL.

Why this matters

As shown in examples so far, many features outside of UniTeX are needed to express the various stages of compression. Whether it be something simple like maintaining an integer tracking the best cost of prediction, or more complicated like entropy coding requiring a lot of control flow, these non-UniTeX features are intimately interweaved with UniTeX features. As before, `CoLa` and `SHiM` strike a balance in the features they offer, as too few features hamper their usefulness, while too many features make them more like general purpose languages, ultimately making it difficult to provide targeted optimizations.

A DSL for UniTeX needs to be able to express some degree of control flow and external operations within it. However, this limits the choice for DSLs. For example, consider an embedded DSL within C++ such as Halide [50]. At a high-level, Halide (and most C++-embedded DSLs) utilizes staging, where the users write their program in C++ using Halide constructs, which creates an AST (abstract syntax tree) of the user program. From the AST, Halide transforms and optimizes it, and then generates new code which the user can link into an external program. Since C++ does not support any type of control flow overloading, any C++ control flow used within the user program will *not* be present in the generated code as Halide has no knowledge of it. For example, the Halide code below presents a simple blur stencil[3] that will produce a two-level loop nest in the generated code that traverses a single image.

3: Taken from https://github.com/halide/Halide/blob/master/apps/blur/halide_blur_generator.cpp.

```
Func blur_x("blur_x");
Var x("x"), y("y"), xi("xi"), yi("yi");
blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3;
blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3;
```

If a user wanted to run this blur across multiple images, they cannot just wrap the Halide code in a loop nest like below because it will execute during staging and will not be present in the generated code.

```
for (auto image : images) {
  blur_x(x,y) = (input(x,y)+input(x+1,y)+input(x+2,y))/3;
  blur_y(x,y) = (blur_x(x,y)+blur_x(x,y+1)+blur_x(x,y+2))/3;
}
```

Instead, they would need to call blur within a loop nest in some external program that links in the generated code for the blur.

Now, in Halide this is not really a problem for the stencil operations since they can capture the necessary program semantics with this declarative structure. But, this will not cut it for compression. `CoLa` achieves this goal by implementing itself as a sort of hybrid standalone-embedded DSL. It implements its UniTeX specific features in a standalone fashion, but also embeds itself within the Pythonic language Codon [20], so it exposes all the existing Codon features, including control flow, and more Pythonic control flow like generators. `SHiM` takes a different approach and embeds itself within C++, but utilizes the BuildIt [21] staging library which captures C++ control flow and other arithmetic operators.

7.3.3 Ability to Optimize

This final requirement is simple: the DSL must remove the overhead incurred by the UniTeX abstraction. Any DSL for UniTeX would be useless if it did not provide any way to optimize, as that would essentially produce another slow library.

Why this matters

By now, it should be obvious why performance matters, especially with how much of a slowdown a library implementation of UniTeX incurs. In general, this requires that DSL includes UniTeX-specific compiler optimizations with knowledge of the UniTeX primitives. As discussed previously, the performance overhead of the library largely stems from the extraneous computations performed during indexing, for example, dividing by coarsening and refinement even if they are equal to one in a given dimension. Such a compiler should be able to perform some type of static analysis to determine when blocks and views have constant values for these expensive properties, and then perform transformations exploiting those constant values, such as constant folding/propagation and function specialization.

`CoLa` follows a more traditional path for optimization, utilizing the compiler framework exposed by Codon. `CoLa` inserts UniTeX-specific passes within Codon's compiler that perform these exact analyses and transformations, allowing it to remove the indexing overhead, and also address some other necessary transformations specific to using Codon as the host language.

`SHiM` also has the ability to insert traditional compiler passes that operate on the AST constructed via BuildIt. However, `SHiM` largely takes a different approach, implementing its data structures and operations so that they are inlined within the generated code from BuildIt, which exposes all the constant values. With this, `SHiM` can rely on the backend compiler (e.g. clang) to perform the necessary transformations, allowing `SHiM` to essentially get the optimizations for "free"[4].

4: "Free" meaning `SHiM` itself does not need to do special static analysis or constant folding/propagation. But it stills need to be careful with implementation to ensure the generated code can actually be optimized by the backend compiler.

7.4 Summary

This chapter provided an overview of various considerations that need to be taken into account when designing an implementation for UniTeX. It discussed the pros and cons of a library implementation, which provides a simple, drop-in use of the tensors and operations for users, but incurs overheads due to the added arithmetic needed for data access. Then it discussed how domain-specific languages provide the ability to remove this overhead through the use of domain-specific optimizations, though at the cost of a (potentially) more complicated implementation of the language itself. It also discussed some important compression-specific considerations that must be taken into account for any domain-specific language for compression, including the need for more general purpose language features such as control flow.

The next chapters discuss two implementations, `CoLa` (Chapter 8) and `SHiM` (Chapter 9), which follow the advice of this section and implement UniTe and UniTeX as domain-specific languages. Though each follows a very different style

of implementation, both focus on removing the overhead of the abstraction while also providing users a library-like feel that supports language features outside of the abstraction itself.

CoLa 8

This chapter introduces one of two implementations of UniTeX for compression, CoLa (Compression Language), which is a hybrid standalone-embedded DSL built around the Pythonic language and compiler Codon [20]. CoLa provides users with UniTeX-specific data structures and functions through customizations made in Codon, while also preserving Codon's Pythonic high-level features. This way, users get the best of both worlds, allowing them to use UniTeX when necessary, and Python for everything else, making it possible to implement end-to-end encoders all within a single language.

Through Codon's extensible compiler framework, CoLa inserts UniTeX-specific passes focused on removing the overhead introduced by the abstraction, bringing average case performance from 20.5×, 48.8×, and 6.5× slower than reference down to 1.2×, 1.0×, and 1.5× faster than that of reference for a series of H.264, JPEG baseline, and JPEG lossless benchmarks, respectively. The rest of this chapter provides examples of CoLa, explains the various additions made to Codon to support the UniTeX-abstraction, and discusses the various optimizations performed by CoLa to reach parity with existing implementations.

8.1 A Taste of CoLa

This section begins with a discussion of the UniTeX components implemented in CoLa and then offers a look at the syntax of CoLa through a description of the API and code examples for different compression kernels introduced in Chapters 2 and 6. Each example provides a mix of the Pythonic syntax of Codon with the UniTeX-specific features of CoLa.

8.1.1 CoLa and UniTeX

CoLa implements a subset of the UniTeX operations introduced in Chapter 5. In particular, CoLa implements block copy, partitions, colocation, and locality access on views (both reads and writes), along with various ways to create new blocks manually. Through partitions, CoLa supports virtual coarsening. However, CoLa refers to coarsening as *striding* instead, which more closely aligns with how implementations describe this type of partition operation[1].

In CoLa's implementation, blocks and views store their absolute location instead of relative locations. This means blocks and views update their absolute location on-the-fly and maintain a flattened representation of location rather than storing the trie structure used to describe UniTe and UniTeX. Views also store a reference to their nearest block, which is still necessary for data access operations.

1: See https://www.w3schools.com/python/numpy/numpy_array_slicing.asp. Note that "slicing" in UniTeX refers to a slightly different operation that drops a dimension. Thus slicing in NumPy is equivalent to partitioning in UniTeX terminology.

8.1.2 API

CoLa supports all the Pythonic features provided by Codon (see Section 8.2) and implements UniTeX features using Codon-proper. In particular, CoLa heavily relies on magic-method overloading (see Subsection 8.3.1) for many of the UniTeX operations. The segments below highlight the main operations related to UniTeX.

`Block.make(origin,extents,stride,init,elem_type)` creates a new block from a set of manually-supplied parameters. CoLa provides various versions of this function that take in different combinations of the parameters, each of which are tuples of the same length (where the length is the dimensionality). Any omitted parameters are given default values[2]. At a minimum, the type parameter is required, as it specifies the underlying element type stored within the Block. CoLa supports all primitive Codon types as the element type. If the init parameter is specified (which can be a Codon list), CoLa copies the data from init into the memory allocated for the block.

2: Zero in all dimensions for origin and one in all dimensions for extents and stride.

`Block.make(tensor)` performs a block copy, copying the parameters of tensor and allocating new memory for the resulting block. CoLa does not copy the data from the tensor in this case.

`tensor[x0:y0:z0,...,xN:yN:zN]` partitions a tensor (see Subsection 5.2.3), producing a new view. This overloads the `__getitem__` magic method, and operates using the familiar slice syntax of Python. Here, the `x` parameters represent the start, `y` represents the stop, and `z` represents the stride. In place of the start, stop, slice triples, a single integer value can be used instead. If the number of parameters is less than the dimensionality, CoLa pads the left with zeros. The start and stop may point out-of-bounds with respect to the tensor and perform a locality partition.

`tensor0[tensor1]` performs colocation (see Subsection 5.2.4), producing a new view. This also overloads the `__getitem__` magic method.

`tensor(c0,...,cN)` reads a value from a tensor (see Subsection 5.2.6) at the coordinate specified by the `c` indices, which are integers. This overloads the `__call__` magic method. If the number of parameters is less than the dimensionality, CoLa pads the left with zeros. The start and stop may point out-of-bounds with respect to the tensor and perform a locality access.

`tensor[c0,...,cN] = val` writes a value to tensor (see Subsection 5.2.6) at the coordinate specified by the `c` indices, which are integers. This overloads the `__setitem__` magic method. If the number of parameters is less than the dimensionality, CoLa pads the left with zeros. The start and stop may point out-of-bounds with respect to the tensor and perform a locality access.

`tensor[!x] = tensor2(!x)` creates an implicit loop nest around the write statement and performs an elementwise write. See Subsections 8.3.2 and 8.3.4 for more information.

8.1.3 Code Examples

The first piece of code shown in Figure 8.1 gives the implementation for the color conversion operation in JPEG, which was presented earlier in both Subsections 2.1.3 and 6.2.2. This code relies on reads and writes to access the two tensors, `YCbCr` and `RGB`. Blocks and views have the same operations, thus users do not need to worry about whether `YCbCr` and `RGB` are blocks or views (Codon's type system handles type inference at compile time). It also highlights `CoLa`'s custom elementwise iterator syntax, given by the bang notation (i.e., `!i`). Elementwise iterators define an implicit loop nest around a write statement, applying the same operation to each tensor in the statement, thus removing the need for explicit loops. This is discussed more in Subsection 8.3.2.

The next piece of code given in Figure 8.2 shows part of `CoLa`'s implementation for the DCT, originally introduced in Subsections 2.1.4 and 6.2.3. The remainder of the DCT looks the same as what is shown, but just using different indices. This code uses reads and writes throughout, along with `CoLa`'s partition syntax (Lines 3 and 11).

Figure 8.3 on Page 92 gives `CoLa` code for part of the Huffman entropy coding kernel of JPEG, mentioned prior in Subsections 2.1.5 and 6.2.4. This code just highlights that `CoLa` preserves the Pythonic operations provided by Codon and supports seamlessly interleaving `CoLa` operations (see the reads on Lines 3 and 21). It also shows the function `pack_and_stuff`, which is part of a bitstream packing library in `CoLa`.

```
1 def RGB2YCbCr(RGB, YCbCr):
2   YCbCr[0,!i,!j] = \
3     RGB(!i,!j,0)*0.299 + RGB(!i,!j,1)*0.587 + RGB(!i,!j,2)*0.114
4   YCbCr[1,!i,!j] = \
5     RGB(!i,!j,0)*-0.168736 + RGB(!i,!j,1)*-0.33126 + RGB(!i,!j,2)*0.500002+128
6   YCbCr[2,!i,!j] = \
7     RGB(!i,!j,0)*0.50000 + RGB(!i,!j,1)*-0.418688 + RGB(!i,!j,2)*-0.081312+128
```

Figure 8.1: `CoLa` code for color conversion in JPEG. This code utilizes `CoLa`'s custom elementwise iterator syntax in conjunction with reads and writes on the tensors `YCbCr` and `RGB`.

```
1  def dct(obj):
2    for r in range(8):
3      row = obj[r,:]
4      tmp0 = int(row(0) + row(7))
5      tmp7 = int(row(0) - row(7))
6      ...
7      row[7] = descale(tmp4 + z1 + z3, 11)
8      row[5] = descale(tmp5 + z2 + z4, 11)
9      ...
10   for c in range(8):
11     col = obj[:,c]
12     tmp0 = int(col(0,0) + col(7,0))
13     tmp7 = int(col(0,0) - col(7,0))
14     ...
15     col[7,0] = descale(tmp4 + z1 + z3, 15)
16     col[5,0] = descale(tmp5 + z2 + z4, 15)
17     ...
```

Figure 8.2: `CoLa` code showing part of a DCT. This code utilizes reads, writes, and partitions. It also uses `CoLa`'s implicit index padding, which adds 0s to the front of any coordinates that contain less than N indices, where N is the dimensionality of the tensor. This mimics the slice operation of UniTe.

```
def huffman_encode_block(blk, last, bits, zigzag, huff_codes: HuffmanCodes):
  # DC
  dc = blk(0)
  temp = dc - last
  temp2 = temp
  if temp < 0:
    temp = -temp
    temp2 -= 1
  nbits = 0
  while temp > 0:
    nbits += 1
    temp >>= 1
  pack_and_stuff(bits, huff_codes.dc_ehufco[nbits], huff_codes.dc_ehufsz[nbits])
  if nbits != 0:
    pack_and_stuff(bits, temp2, nbits)
  # AC
  run = 0
  ziter = iter(zigzag)
  next(ziter) # skip the DC
  for zcoord in ziter:
    ac = blk(*zcoord)
    ...
```

Figure 8.3: CoLa code showing part of the Huffman entropy coding kernel. This code does not use many UniTeX-specific features, but serves to show that CoLa preserves the Pythonic features of Codon. It also uses some library functions for bit packing (`pack_and_stuff`) provided by CoLa.

The final sample, given in Figure 8.4, gives CoLa code for computing the 4x4 vertical right intra-prediction mode. This code highlights several reads/writes/locality accesses (Lines 7, 9, 11 and 13), and a colocation operation (Line 2) which gets the region of pixels in the reconstructed frame (represented by `ref`) based on the location of the predication (`pred`) submacroblock.

```
def intra_4x4_vr(pred, ref):
  p = ref[pred]
  for y in range(4):
    for x in range(4):
      zVR = 2*x-y
      if zVR % 2 == 0 and zVR > 0:
        pred[y,x] = (p(-1,x-(y>>1)-1)+p(-1,x-(y>>1))+1)>>1
      elif zVR > 0:
        pred[y,x] = (p(-1,x-(y>>1)-2)+2*p(-1,x-(y>>1)-1)+p(-1,x-(y>>1))+2)>>2
      elif zVR == -1:
        pred[y,x] = (p(0,-1)+2*p(-1,-1)+p(-1,0)+2)>>2
      else:
        pred[y,x] = (p(y-1,-1)+2*p(y-2,-1)+p(y-3,-1)+2)>>2
```

Figure 8.4: CoLa code for computing the vertical right mode of 4x4 intra-prediction. This code utilizes elementwise reads and writes, locality accesses, and colocation.

8.2 Overview of the Codon Language

At its core, Codon [20] is a high-level general purpose language that implements the syntax of Python, along with many of Python's features and semantics. Compared to Python, Codon differs in two primary ways: 1) it enforces static type checking and requires strong types, and 2) it utilizes ahead-of-time compilation. Unlike other extensions to Python, such as MyPy [51] and PyType [52] for

optional static type checking, Codon does not utilize the Python interpreter (or any parts of the standard Python implementation) at any stage of compilation or execution. Rather, Codon implements its own full-featured compilation framework that supports parsing, AST/IR generation, typechecking, optimization, and code generation with LLVM. Figure 8.5 provides a breakdown of Codon into separate layers and briefly describes each below. The next section will explore how `CoLa` modifies these layers.

Layer 1: User API Represents the user-facing frontend where users implement their programs. This exposes Pythonic data structures, operators, and libraries.

Layer 2: Parser Specifies Codon's grammar as a parsing expression grammar [53] (PEG) derived from Python 3's PEG[3] (and adds type annotations). This also generates Codon's AST.

3: `https://docs.python.org/3/reference/grammar.html`

Layer 3: AST Transformations/Typechecking Performs various canonicalization and simplification passes on the AST. It also runs type inference and type checking, assigning static types for every object and operation.

Layer 4: AST Lowering Generates Codon IR from the AST, which is a bidirectional graph IR that can interact with the typechecker in order to generate new fully-type IR nodes.

Layer 5: IR Transformations Includes various analysis and transformation passes. Codon includes a default set of passes that can be run on any program.

Layer 6: LLVM Code Generation Generates LLVM code and includes an additional Codon-specific LLVM pass focused on coroutines (which are used to implement Python generators in Codon).

Using Codon to implement `CoLa` gives users access to Python primitive data types (numerical types, sets, lists, dictionaries, etc.), Python language constructs (classes, generators, magic methods, etc.), and the most common Python modules (math, file I/O, exception handling, etc.), all of which have been reimplemented to work with Codon[4]. Developers using Codon as a host language for a DSL implementation (as done with `CoLa`) get access to the full AST, type-

4: Codon supports about 95% of the features in Python, but due to its use of static and strong typechecking, Codon does not support truly dynamic features such as dynamic polymorphism.

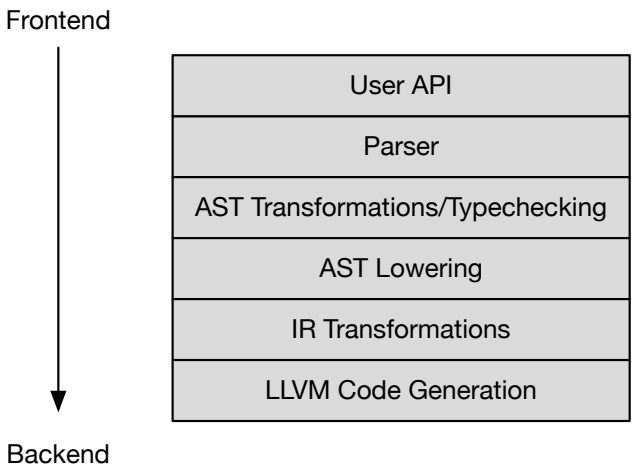


Figure 8.5: Codon's framework, which supports all the stages of compilation from frontend parsing down to code generation with LLVM. `CoLa` inserts UniTeX-specific functionality in nearly every layer (except LLVM Code Generation), but still preserves the core functionality of Codon, which provides users with a combination of UniTeX and Python features.

checking, and IR framework, as well as a set of standard compiler passes for analysis (control flow graph construction, reaching definitions analysis, side-effect analysis, etc.) and transformations (dead code elimination, constant folding, global demotion, etc.). CoLa utilizes many of these existing features, and also inserts customizations at nearly every level of the Codon framework.

8.3 Implementation

CoLa's implementation with Codon adds in UniTeX-specific data structures and methods and custom compiler passes operating on those data structures. This section goes through each layer in Codon's framework (Figure 8.5) and points out any modifications that CoLa makes to the layer. The frontend levels mainly deal with the API presented to users, and the backend levels focus on optimizations.

8.3.1 Layer 1: User API

CoLa defines its API using Codon-proper, effectively defining all the data structures and operations as a library, and then applies CoLa-specific compiler passes later to optimize it. Rather than implementing UniTeX with separate tensors and reference spaces, CoLa ties the reference space parameters directly with tensors. It also utilizes a flat representation such that tensors always maintain their absolute parameter values, as opposed to relative values like in the description of UniTeX. The user API includes all the type information and logic necessary to generate blocks and views and perform all operations on them, reducing the amount of changes required in the typechecking stage for CoLa-specific objects.

CoLa utilizes Codon classes to implement blocks and views and heavily relies on magic method[5] overloading to provide users with intuitive syntax for interacting with blocks and views. In all, the UniTeX frontend of CoLa is approximately 750 lines of Codon code[6]. Figure 8.6 provides an example of the implementation for `tensor(c0,...,cN)`, showing that CoLa just utilizes the normal Pythonic syntax of Codon to implement its methods.

5: *Magic methods* allow Python (or Codon) objects to exploit built-in functions and operators within Python (or Codon). CoLa utilizes them to exploit operators on its classes, such as overloading the square bracket syntax to perform colocation on blocks and views.

6: The full frontend of CoLa is approximately 1,250 lines of code, with the other 500 lines of code implementing a bit-level library for packing and outputting the compressed bitstreams.

```
def __call__(self, *idxs) -> E:
  if staticlen(idxs) == 0:
    compile_error('Idxs is an empty tuple. Did you use * with an integer argument?')
  padded = pad_zeros(idxs, staticlen(self.dims()))
  for p in padded:
    if not isinstance(p, int):
      compile_error('Must be integer index. Did you forget *?')
  coord = compute_astarts(self.astarts(), self.astrides(), padded)
  # now do relative to the block
  bcoord = tup_fdiv(tup_sub(coord, self._block.astarts()), self._block.astrides())
  lidx = linearize(bcoord, self._block.dims())
  return self._block._buff[lidx]
```

Figure 8.6: An example of an overloaded magic method in CoLa for performing a read on a view. CoLa uses plain Codon to implement its functions in the frontend, thus adding new operations for blocks and views is straightforward.

8.3.2 Layer 2: Grammar

In this layer, CoLa defines syntactic sugar for the elementwise iterators. This operator is indicated by the use of the bang (!) symbol and defines an implicit loop nest around writes to a tensor. The top of Figure 8.7 shows an example using two of these iterators. Here, `!y` and `!x` on the left-hand side define a two-level loop nest that would iterate over the two dimensions of `T0`. The use of `!y` on the right-hand side operates like a normal loop index and can be used wherever an integer[7] can be used. The bottom of Figure 8.7 shows the corresponding loop nest code.

```
T0[!y,!x] = T1(!y,-1)
```

```
for y in range(T0.dims(0)):
  for x in range(T0.dims(1)):
    T0[y,x] = T1(y,-1)
```

Figure 8.7: Elementwise iterators (top) and the loop nests they correspond to (bottom).

7: Specifically, a 64-bit integer.

To provide this operator to the user, CoLa augments Codon's parsing expression grammar specification with the simple syntax shown in Figure 8.8. This parses the elementwise iterators like a normal expression, and then the later layers check for valid usages of the iterators. Subsection 8.3.4 discusses the transforms necessary to convert writes with elementwise iterators into their corresponding loop nest form.

```
/ '!' NAME {
  return ast<IdExpr>(...);
}
```

Figure 8.8: PEG rule added to Codon to support element iterators in CoLa.

CoLa provides this syntactic sugar to users as elementwise writes are an extremely common operation on the innermost tensors created within the encoder, and this gives a straightforward way to perform the operation while avoiding bugs such as selecting the wrong loop extent. For block-based compression, a simple iterator such as this is satisfactory; other types of index notation like einsum are unnecessary[8]. Despite the simplicity of the operation[9], this syntax defers from typical implementations of elementwise operations, which typically would utilize magic method overloads to implement the elementwise operations instead. For example, NumPy implements an elementwise addition by overloading `__add__` for ndarrays[10]. CoLa could implement the elementwise operations this way, but avoids doing so for two primary issues: 1) creation of temporaries and 2) ambiguity.

8: CoLa's usage is similar to that in Halide [50], which also defines simple elementwise iterators for use in stencil computations.

9: While this provides the necessary level of flexibility for the purposes of compression, the only real disadvantage of this notation is the fact that it requires modifying the compiler, which increases the complexity of CoLa itself.

10: And also provides a longer form `add` function: https://numpy.org/doc/stable/reference/generated/numpy.add.html.

Temporaries A common issue in libraries that implement these types of operations using methods (whether magic or not) is the introduction of temporaries. For example, if CoLa provided the `__add__` method between tensors, and a user wrote the expression `t0 = t1 + t2 + t3`, this would result in creating a temporary to hold `t1+t2`, then adding the temporary to `t3`, creating yet another temporary. Then the result in that second temporary would be written to `t0`. With CoLa's syntax, this could just be written as something like `t0[!i] = t1[!i] + t2[!i] + t3[!i]`, which does not create any temporaries and writes directly to the result. With Codon (or Python), the only way to avoid this would be to warn the user that they should not write the code this way and instead write a loop nest to perform the operation. C++-based libraries can utilize expression templates [54] to solve this issue by lazily building up the expression; however, that requires the use of templates, so it is not relevant to Codon nor Python.

Ambiguity Ambiguity in an expression like `t0 = t1 + t2 + t3` arises due to the location associated with tensors. Ignoring the issue of temporaries from above, a statement like this requires that CoLa implicitly create a new block tensor for `t0` that holds the resulting computation. However, it is ambiguous which location should be used for `t0` since each tensor on the right-hand side

could have a different location. The elementwise notation `t0[!i] = t1[!i] + t2[!i] + t3[!i]` would require that `t0` already exists, so no ambiguity arises.

8.3.3 Layers 3 and 4: AST Transformations/Typechecking/AST Lowering

Here, CoLa primarily includes the code necessary to propagate through the use of the elementwise iterators as they are not lowered until the IR transform stage. This involves wrapping the iterators in CoLa-specific AST nodes, and assigning integral types to them so the typechecker can appropriately typecheck any expressions and statements that utilize them. This is similar to what Codon does for magic methods, where it looks for the special syntax associated with the magic method and converts it to the appropriate method call.

8.3.4 Layer 5: IR Transforms

All of Codon's optimizations and analysis passes live within this layer, and it is also where CoLa inserts its custom compiler passes targeting blocks, views, and the operations on them. Through Codon, the IR at this level is fully typechecked, so it is possible to detect uses of CoLa-specific objects by just checking different type information attached to the IR nodes. The main transformations in CoLa fall into two primary categories: lowering passes and optimization passes. The lowering passes are necessary to ensure correct code generation, while the optimization passes are necessary to remove the performance overhead that comes from the UniTeX features (see Section 8.4). This section discusses three main passes: elementwise lowering, collapsing, and parameter propagation. As the name suggests, elementwise lowering is a lowering pass, while the other two are optimization passes. Each pass is actually a collection of several smaller passes, but to simplify the presentation, they are described as if they are singular passes.

Elementwise Lowering

The elementwise lowering pass is a straightforward pass that adds explicit loop nests around write statements using elementwise iterators. This pass first verifies that all usages of elementwise iterators are valid, which includes checking that iterators on the left-hand side are unadorned, iterators are unique, and right-hand side iterators exist on the left-hand side. For example, the following usages are invalid:

```
pred[!y,!x] = p(!y,!z)   # INVALID USAGE
pred[!y,!x+1] = p(!y,!x) # INVALID USAGE
pred[!y,!y] = p(!y,!y)   # INVALID USAGE
```

The first one is invalid because `!z` does not occur on the left-hand side. The second is invalid as `!x` is adorned on the left-hand side (it can be adorned on the right-hand side). Finally, the last one is invalid because `!y` occurs in two positions on the left-hand side.

The next step infers the extents of the loop nest based on their location on the left-hand side. For example, consider the following elementwise write below:

```
obj[!i,0,!j] = p(!j+1,-1)+p2(0,0)+p3(!i,!j)
```

This write contains two iterators, `!i` and `!j`, thus corresponds to a two-level loop nest. CoLa infers the extent based on which dimension on the left-hand side contains the iterator. In this case, `!i` corresponds to dimension 0 of `obj`, and `!j` corresponds to dimension 2. Thus, the outer loop will have the extent corresponding to dimension 0, and the inner loop will have the extent of dimension 2. CoLa wraps the write in the loop nest, and converts the elementwise iterators into simple integer variables (on both the left-hand side and right-hand side), resulting in the code below:

```
for i in range(obj.dims()[0]):
  for j in range(obj.dims()[2]):
    obj[i,0,j] = p(j+1,-1)+p2(0,0)+p3(i,j)
```

For multidimensional loops, CoLa opts for a simple transformation that nests the loops in the order that the iterators are used on the left-hand side (going outermost-to-innermost). CoLa also places the write at the innermost loop level, even if it could be hoisted; it relies on the LLVM backend for performing loop-based transforms and optimizations.

Collapsing

The collapsing optimization pass aims to reduce the number of intermediate views created from partitioning by looking for reads, writes, and partitions on views and propagating those operations as high up the partition hierarchy as possible, effectively collapsing successive views together. Views are often created from partitions at the innermost levels of loop nests, so reducing the number of views both reduces the number of operations needed to create new objects and the number of heap allocations needed, as Codon allocates all created objects on the heap. Figure 8.9 on Page 95 shows an example of a before and after that collapses two successive partitions such that they can be removed.

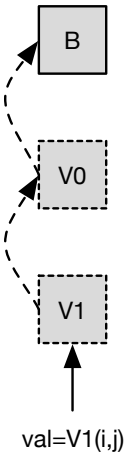
Collapsing proceeds in a bottom-up fashion, iteratively repeating until no more collapsing can occur. It runs two main steps that 1) identify candidate reads/writes and 2) lift the reads/writes up to the parent of the view. To find the candidates, CoLa checks for any reads or writes on a view, and then looks for the definition of the view itself. CoLa utilizes Codon's built-in reaching definitions analysis for this part. If the definition of the view is the result of a partition operation, this can be collapsed[11]. In the code on the top of Figure 8.9, there is a read on `V1`, where `V1` is produced by a partition on `V0`, thus this can be collapsed. In this case, the read is referred to as the *downstream operation*, `V1` is the *downstream view*, while the partition on `V0` is the *upstream operation* and `V0` is the *upstream view*[12].

Once a candidate has been identified, the next step combines the indices of the downstream operation and upstream operation together, and moves the read or write on the downstream view to the upstream view. Looking now at the middle

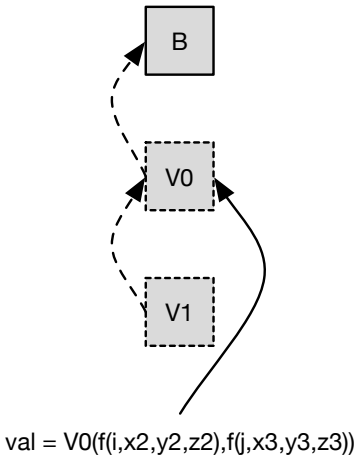
11: This pass is intra-procedural, so partitions across function boundaries are not considered.

12: Collapsing only proceeds when a use has a single reaching definition. For the encoders implemented in CoLa, this was not an issue. However, it would be possible to expand the collapsing pass and perform specialization with multiple reaching definitions if necessary.

```
1 i,j = <some integer values>
2 # partition
3 V0  = B[x0:y0:z0,x1:y1:z1]
4 # partition
5 V1  = V0[x2:y2:z2,x3:y3:z3]
6 # read
7 val = V1(i,j)
```




```
1 i,j = <some integer values>
2 # partition
3 V0  = B[x0:y0:z0,x1:y1:z1]
4 # partition
5 V1  = V0[x2:y2:z2,x3:y3:z3]
6 # read
7 val = V0(f(i,x2,y2,z2),f(j,x3,y3,z3))
```




```
1 i,j = <some integer values>
2 # partition
3 V0  = B[x0:y0:z0,x1:y1:z1]
4 # partition
5 V1  = V0[x2:y2:z2,x3:y3:z3]
6 # read
7 val = B(f(f(i,x2,y2,z2),x0,y0,z0),
8         f(f(j,x3,y3,z3),x1,y1,z1))
```

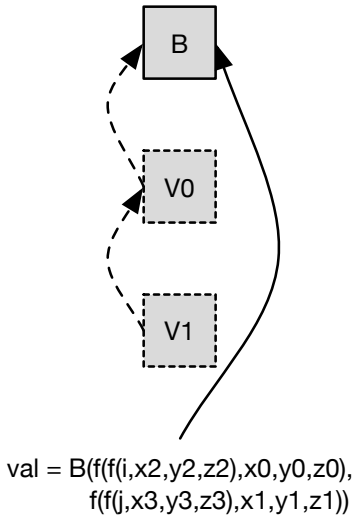


Figure 8.9: Before and after collapsing. The top code snippet does two partitions, creating two views, and then performs a read on the final partition `V1`. The top diagram gives the initial state. The middle example shows the result of lifting the read on `V1` and collapsing it with the partition on `V0`, rendering `V1` dead (the function `f` is a placeholder for a function inserted by the compiler that combines the indices of the read and the partition). The bottom example repeats this process, lifting the newly formed read on `V0` and collapsing it with the partition on `B`. Now `V0` is dead and also can be removed.

of Figure 8.9, this has combined the original indices `i` and `j` in the downstream operation with the partition parameters `x2`, `y2`, `z2`, `x3`, `y3`, `z3` in the upstream operation (with the combination represented by the black box function `f`). It also moved the read to be on the upstream view. At this point, the downstream view `V1` is no longer required and can be removed by dead code elimination (also provided by Codon). In this particular example, it is possible to collapse one more time, resulting in the code on the bottom of Figure 8.9.

Location Propagation

Location propagation performs an aggressive form of constant propagation and function specialization across an entire CoLa program, propagating through constant values for the origin, stride, and extent parameters of blocks and views. This pass ultimately helps expose opportunities for constant folding, which helps remove certain expensive indexing operations, such as division by stride.

Consider the example in Figure 8.10. This shows the high-level steps performed in order to infer that the value for `d0` (a) is equivalent to the value 20 (d). At a high-level, this pass operates top-down and looks for uses of origin, stride, and extent, which are accessible by the user through corresponding getter functions (which are called `origin()`, `strides()`, and `dims()`). When CoLa finds one of the getter functions, it attempts to infer any constant values within the returned tuple by checking whether the corresponding tensor contains constant values.

This pass runs independently for each type of parameter, but each follows the same basic structure containing two (unordered) phases: 1) inference and 2) propagation. Each phase is applied on-the-fly rather than doing all inference in one pass, then all propagation in one pass.

```
1 block = Block.make(dims=(10,20))
2 d0 = block.dims()[1]
```

(a) Initial state before location propagation.

```
1 block = Block.make(dims=(10,20), origin=(0,0), stride=(1,1))
2 d0 = block.dims()[1]
```

(b) Expanding builder functions.

```
1 block = Block.make(dims=(10,20), origin=(0,0), stride=(1,1))
2 d0 = (10,20)[1]
```

(c) Propagating dimensions.

```
1 block = Block.make(dims=(10,20), origin=(0,0), stride=(1,1))
2 d0 = 20
```

(d) Tuple extraction.

Figure 8.10: Simple example of location propagation in CoLa. In this code, CoLa can infer that the value of `block.dims()[1]` in (a) is 20 based on the parameters originally used to create `block`. It does this by first expanding the `make` function to include any missing parameters in (b), then propagating the appropriate parameter tuple in place of `block.dims()` in (c). Finally, CoLa simplifies the resulting code with tuple extraction, which removes the second element of the tuple and results in the value of 20 in (d). CoLa implements a series of passes that can be used to infer the values of the origin, extents, and strides of tensors at various points throughout a program.

The first phase, inference, runs whenever CoLa finds a definition for a tensor. For example, in Figure 8.10, CoLa finds the `Block.make` function. CoLa infers any missing parameters for the function (origin and stride in this case) by inserting default values (b). CoLa performs this general process when encountering such a definition[13]. In the event that the parameters provided to the definition are not constants, CoLa attempts normal constant propagation and folding to see if the parameters correspond to constants. Once all the constants are found, CoLa now knows the full location the block (or view).

13: CoLa also looks for views created from partitioning, and can compute the location of the view based on the partition parameters and the parameters of the tensor being partitioned. However, most views are removed in the collapsing phase, so this is not as common.

The propagation phase looks for any usages of the getter functions and checks to see if the tensor that the getter is called on has constants for that particular value. In the example of Figure 8.10, the getter function for the extents (`block.dims()`) can be replaced by `(10,20)` in (c) since `block` has constant values for that parameter[14]. To find the tensor that the getter is called on, CoLa relies on reaching definitions analysis provided in Codon. Finally, CoLa simplifies the result, which gives the code in (d).

14: CoLa also supports propagation in cases where the parameter only has constant values in some of the dimensions.

CoLa runs this pass inter-procedurally, performing function specialization if a tensor is passed into a function and the function calls a getter on it. For example, consider the code in Figure 8.11a, which reconfigures the code in Figure 8.10 to use a function call. When CoLa finds a getter function (`block.dims()` in `foo`) and the tensor definition points to a function argument, CoLa goes to the calling site and performs the same type of reaching definitions/constant propagation as before, except moving it across the function boundary, effectively specializing the function with the constant values. CoLa also memoizes the specialized function

```
1 def foo(block):
2   d0 = block.dims()[1]
3 block = Block.make(dims=(10,20))
4 foo(block)
```

(a) Initial state before location propagation.

```
1 def foo(block):
2   d0 = block.dims()[1]
3 block = Block.make(dims=(10,20), origin=(0,0), stride=(1,1))
4 foo(block)
```

(b) Expanding builder functions.

```
1 def foo_special(block):
2   d0 = (10,20)[1]
3 block = Block.make(dims=(10,20), origin=(0,0), stride=(1,1))
4 foo_special(block)
```

(c) Specializing `foo`.

```
1 def foo_special(block):
2   d0 = 20
3 block = Block.make(dims=(10,20), origin=(0,0), stride=(1,1))
4 foo_special(block)
```

(d) Tuple extraction.

Figure 8.11: Simple example of function specialization within location propagation. CoLa can infer the constant tuple for `block.dims()` in the `foo` function (see (a)) using the same techniques as for intra-procedural location propagation (see Figure 8.10), and then specializing the function body with the constants.

in the case that multiple specializations all produce the same function.

8.3.5 Layer 6: LLVM Code Generation

By this point, all CoLa-specific features have been lowered to standard Codon IR nodes, thus CoLa does not modify the code generation layer.

8.4 Evaluation

The optimizations performed in CoLa allow it to both reduce the number of views created (collapsing pass) and specialize location information for tensors (location propagation pass), which impact the performance in different ways. This section discusses CoLa's performance on three different encoder benchmarks.

8.4.1 Benchmarks

Performance is measured across three different benchmarks, two for JPEG [22] and one for H.264 [19], and compared against the reference implementations for each standard. For JPEG, implementations for both the sequential baseline (JPEGS) and lossless version (JPEGL) are included. JPEGS represents the version of JPEG used as an example throughout this dissertation, and follows the pipeline given in Chapter 2. JPEGL, despite having JPEG in the name, follows a different encoder pipeline. Briefly, JPEGL uses similar entropy coding and syntax output to JPEGS, but utilizes prediction instead for the core of the encoder (as opposed to the DCT and quantization in JPEGS). It supports eight different prediction modes which utilize up to three surrounding pixel values. Figure 8.12 shows the pixel positions available for prediction and Table 8.1 gives the actual predictions. From the prediction, it computes the residual and sends that to the entropy coding stage. Like H.264, JPEGL uses reconstructed pixels for computing the prediction as opposed to the raw values.

For H.264, CoLa implements an encoder utilizing a subset of the pipeline given in Chapter 2.

The performance of CoLa is compared to ijg [16], libjpeg [55], and JM [8] for JPEGS, JPEGL, and H.264 benchmarks, respectively. These existing systems provide a suitable comparison as they implement straightforward versions of algorithms used within the compression pipeline, and include many options for tuning what features are used. This provides a fairer comparison than with other implementations that utilize numerous algorithmic modifications for performance. These types of algorithmic changes are outside the scope of this work.

8.4.2 Experimental Setup

CoLa's performance is evaluated across each of the benchmarks discussed above, first applying only the lowering passes required for correctness, which effectively runs CoLa as a library. Then, collapsing is added ("+ Collapse"),

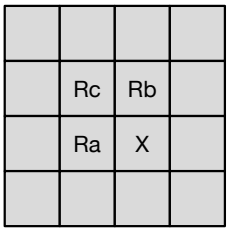


Figure 8.12: Valid reconstructed pixel component positions for use in predicting x with JPEG lossless.

Mode	Prediction
0	None
1	R_a
2	R_b
3	R_c
4	$R_a + R_b - R_c$
5	$R_a + (R_b - R_c)/2$
6	$R_b + (R_a - R_c)/2$
7	$(R_a + R_b)/2$

Table 8.1: Prediction modes for JPEG lossless using the neighborhood shown in Figure 8.12.

Image	Resolution
Rose	227×149
Cat	640×960
Poster	2160×2880
Painting	2653×3307
Satellite	5181×4828
Overcast	7200×5400
Overcast Big	7200×10800
Map	10315×7049
Map Big	20630×7049

Table 8.2: Resolution of test images.

Video	Resolution
Carphone	QCIF
Coastguard	QCIF
Foreman	CIF
Akiyo	CIF
Football	CIF

Table 8.3: Resolution of test videos.

followed by location propagation for each of the individual properties ("+ Prop (origin)", "+ Prop (strides)", "+ Prop (extents)"). The JPEG benchmarks use the test images given in Table 8.2, while the H.264 benchmarks use the videos given in Table 8.3. All H.264 benchmarks are evaluated across 200 frames.

All runtimes in Subsection 8.4.3 are given in seconds (H.264 also provides an additional frames per second version) and represent the average of 20 iterations. Through Codon, all CoLa code is generated and compiled through llvm version 12.0.0 with the default -O3 optimizations. All reference codes were compiled with clang 12.0.0 (all references are written in C) with -O3, along with the flags included in the Makefiles provided with each reference. Each experiment was run on systems with Intel Xeon E5-2695 v2 cores running at 2.40GHz. All times are for single threaded performance. Allocation and instruction counts were collected with valgrind.

8.4.3 Runtime Performance

Figures 8.13 to 8.15 on Pages 104 and 105 show the runtime results for JPEGS, JPEGL, and H.264, respectively. All of these charts show the speedup of CoLa relative to the corresponding reference implementation (represented by the value one on the vertical axis) such that bars greater than one represent a speedup. Tables 8.4 to 8.6 on Page 105 provide the raw runtimes of each, and Table 8.7 on Page 106 also gives the performance for H.264 in frames per second (which is a more standard way of representing performance for video compression applications). With the combination of these optimizations, CoLa achieves speedups of 1.0×, 1.5×, and 1.2× relative to the reference implementations for JPEGS, JPEGL, and H.264, respectively.

Unsurprisingly, the performance of CoLa without any optimizations is far worse than compared to the reference C implementations, running nearly 65× slower in the worst case (JPEGS-overcast). Similar to the library implementation in the previous chapter, there are two main culprits for the performance degradation: 1) an excess of views, and 2) additional indexing computations. Views are often created at innermost loop levels, and tensors are often indexed at those inner levels as well. Thus, any additional overhead proves catastrophic for the runtime. This is discussed next.

Location Propagation

In the context of additional indexing overhead, stride incurs an extremely costly overhead due to divisions and multiplications that happen for many of the operations. However, in most cases, the stride is one, making these operations unnecessary. Even when the stride is greater than one, it is usually a power of two, so the multiplication and divisions can be converted into less costly shift operations. With CoLa's passes, it effectively exposes these opportunities for constant folding and strength reduction to the underlying llvm framework, providing the necessary reduction in overhead.

Collapsing

With view creation, performance is impacted partially due to the fact that Codon allocates objects on the heap, as well as due to the need for computing the new parameter values for a view whenever one is created. With the collapsing pass, many of these views are eliminated. Using the JPEGL benchmark, Table 8.8 on Page 106 shows the total number of instructions, as well as the total number of `mallocs` before and after this pass. For instruction counts, `CoLa` is able to reduce the number of instructions by up to a factor of ten, largely due to the fact that removing views also removes the need to recompute a new parameterization for each of them.

Collapsing also significantly impacts the number of `mallocs` in the generated code as shown in the bottom of Table 8.8. Again, this is due to collapsing being able to remove many of the views itself. To support Python's pass-by-object-reference semantics, Codon allocates all objects on the heap (and performs automatic memory management). As a result, all blocks and views in `CoLa` are allocated on the heap as well. This can lead to a bottleneck, as views are often created at the innermost loop levels of encoders, requiring a large amount of small allocations, especially when the loop extents are proportional to the input image size (as they are here). As seen in the table, JPEGL can allocate up to nearly 500 million objects. However, after removing a majority of the views created after hierarchy collapsing, `CoLa` reduces the number of `mallocs` by up to a factor of 14,120 times! In the end, adding parameter propagation reduces the number of `mallocs` even further. In the case of JPEGL, all the view allocations happen within the innermost levels of the encoder, and `CoLa` is able to remove all of them, leaving just the allocations for the initial block objects and some other non-`CoLa` objects, which all happen outside the main loop nest. As a result, the program ultimately only requires a constant amount of allocations, making allocations invariant to the initial image size.

8.5 Summary

This chapter introduced one of two implementations for UniTeX, `CoLa`. `CoLa` is a hybrid embedded-standalone domain-specific language embedded within the language Codon, which provides the high-level syntax of Python along with a type system and full compiler framework. `CoLa` also extends some of the syntax of Codon and inserts UniTeX-specific passes within Codon's compiler framework, allowing it to optimize usages of blocks and views and remove the overhead associated with a library implementation. In particular, `CoLa` implements two primary optimizations passes called view collapsing and location propagation, which target overheads from creating views and additional indexing, respectively. On a set of JPEG and H.264 benchmarks comparing `CoLa` to reference hand-optimized C implementations, `CoLa` is able to bring performance from nearly 65× slower than reference down to parity.

The next chapter introduces the `SHiM` implementation, which takes a very different implementation approach from `CoLa`, but achieves the same performance goals while still providing a simple API for users.

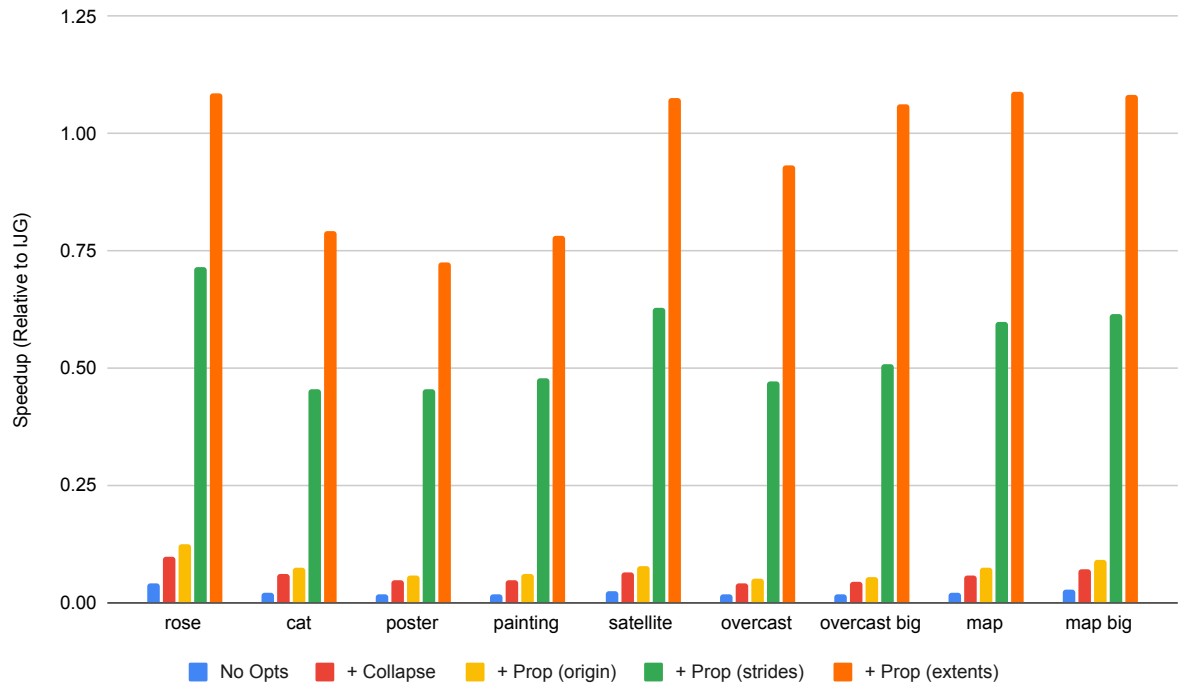


Figure 8.13: CoLa performance on JPEGS relative to IJG.

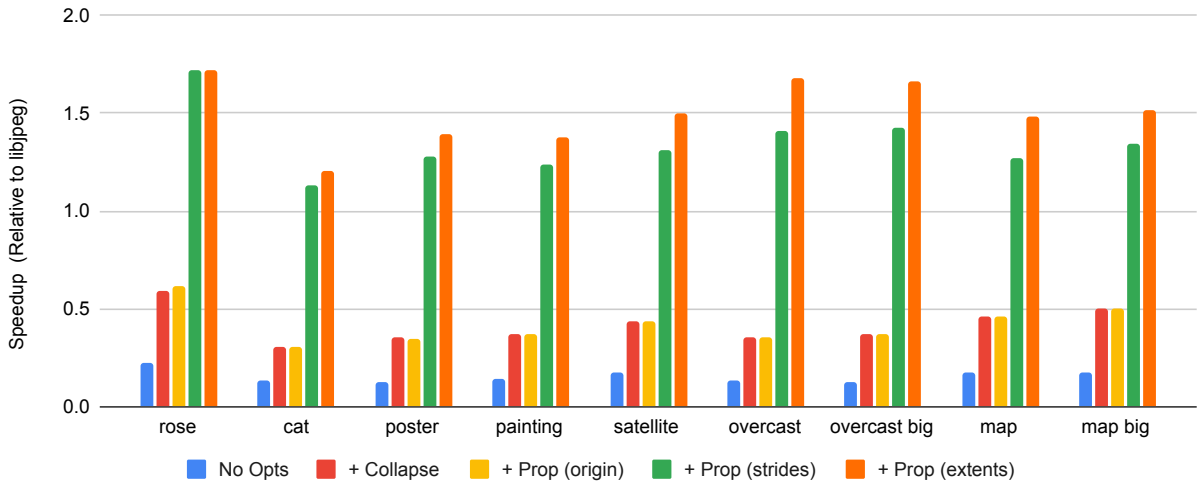


Figure 8.14: CoLa performance on JPEGL relative to libjpeg.

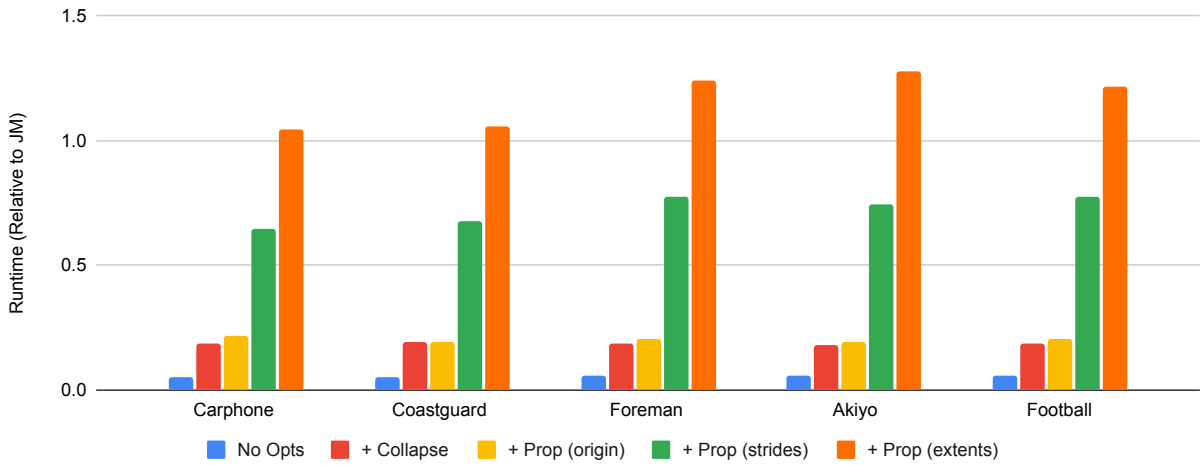


Figure 8.15: CoLa performance on H.264 relative to JM.

Table 8.4: Raw runtimes for CoLa and IJG for JPEGS (in seconds).

	No Opts	+ Collapse	+ Prop (origin)	+ Prop (strides)	+ Prop (extents)	IJG
rose	0.108	0.043	0.034	0.006	0.004	0.004
cat	1.338	0.473	0.392	0.064	0.037	0.029
poster	10.891	4.075	3.262	0.407	0.256	0.186
painting	14.990	5.766	4.627	0.583	0.356	0.278
satellite	42.081	16.502	13.173	1.638	0.958	1.030
overcast	65.482	25.041	19.940	2.132	1.080	1.008
overcast big	138.041	50.078	39.851	4.290	2.051	2.181
map	131.930	47.590	38.058	4.593	2.521	2.750
map big	273.892	97.894	78.274	11.462	6.505	7.054

Table 8.5: Raw runtimes for CoLa and libjpeg for JPEGL (in seconds).

	No Opts	+ Collapse	+ Prop (origin)	+ Prop (strides)	+ Prop (extents)	libjpeg
rose	0.045	0.017	0.016	0.006	0.006	0.010
cat	0.502	0.216	0.215	0.059	0.055	0.066
poster	3.896	1.402	1.418	0.383	0.352	0.491
painting	5.563	2.122	2.108	0.633	0.567	0.783
satellite	15.785	6.284	6.325	2.095	1.833	2.752
overcast	23.545	8.706	8.719	2.182	1.840	3.084
overcast big	49.384	16.394	16.441	4.267	3.655	6.086
map	48.108	17.807	17.833	6.478	5.544	8.217
map big	100.332	34.572	34.766	12.895	11.403	17.317

Table 8.6: Raw runtimes for CoLa and JM for H.264 (in seconds).

	No Opts	+ Collapse	+ Prop (origin)	+ Prop (strides)	+ Prop (extents)	JM
Carphone	23.741	6.143	5.341	1.778	1.093	1.144
Coastguard	23.990	6.195	6.203	1.756	1.125	1.186
Foreman	76.648	23.112	21.535	5.587	3.488	4.333
Akiyo	76.177	22.586	21.409	5.508	3.195	4.085
Football	76.841	23.278	21.524	5.585	3.561	4.340

Table 8.7: Raw runtimes for CoLa and JM for H.264 (in frames per second).

	No Opts	+ Collapse	+ Prop (origin)	+ Prop (strides)	+ Prop (extents)	JM
Carphone	8	33	37	112	183	175
Coastguard	8	32	32	114	178	169
Foreman	3	9	9	36	57	46
Akiyo	3	9	9	36	63	49
Football	3	9	9	36	56	46

Table 8.8: Instruction and `malloc` counts for JPEGL with and without CoLa optimizations.

	# Instructions		
Image	**Pure Library**	**+ Collapse**	**+ Prop (all)**
Rose	120,955,600	29,311,077	20,980,925
Cat	1,787,985,652	344,059,715	199,419,046
Poster	16,192,330,494	3,112,528,526	1,665,259,234
Painting	23,307,655,048	5,107,986,425	3,068,869,577
Satellite	66,622,488,297	15,574,441,545	9,765,728,553
Overcast	100,498,604,280	21,404,924,936	12,379,880,192

	# `mallocs`		
Image	**Pure Library**	**+ Collapse**	**+ Prop (all)**
Rose	405,920	1534	642
Cat	7,369,600	6,400	642
Poster	74,638,720	17,920	642
Painting	105,269,064	20,482	642
Satellite	300,147,744	29,608	642
Overcast	466,539,040	33,040	642

9 SHiM

This chapter looks at a second approach to the implementation of UniTeX, SHiM (Staged Hierarchical Multidimensional arrays), which is an embedded DSL within C++ that utilizes the BuildIt [21] staging library. Like CoLa, SHiM provides UniTeX-specific data structures and functions, but does so without any modifications to the host language. Unlike other DSLs embedded within C++, SHiM supports control flow through BuildIt, allowing users to interleave UniTeX structures with normal C++ control flow, making SHiM capable of implementing end-to-end encoders as with CoLa. SHiM is also helpful for generating individual compression kernels, which can be linked into existing implementations. This makes it possible to incrementally update existing implementations if a user does not want to write an entire end-to-end encoder from scratch.

Like CoLa, SHiM focuses on removing the overhead of the abstraction. However, SHiM takes a very different approach than CoLa, taking advantage of BuildIt's staging capabilities to provide the necessary optimizations "for free." All SHiM-generated programs have these optimizations applied, and the results (Section 9.4) show that the performance of the generated code does not add any additional overhead. The rest of this chapter provides examples of SHiM, discusses the design of SHiM within BuildIt, and finally looks at SHiM's performance on H.264 and JPEG benchmarks.

9.1 Inserting a SHiM

This section highlights using SHiM for a selection of the kernels presented earlier in Chapter 6. These examples highlight the UniTeX operations, as well as the interaction between the non-UniTeX features and normal C++ code (and BuildIt). In particular, these examples use SHiM to write kernels that can generate code that plug into existing implementations (though SHiM can implement end-to-end encoders as well). Since SHiM is embedded within C++, it provides familiar syntax and does not require any changes to the C++ syntax.

9.1.1 SHiM and UniTeX

SHiM implements the full UniTeX abstraction, except for coverage. Like CoLa, blocks and views in SHiM store their absolute location rather than a trie representing the relative location, and views also maintain a reference to their nearest block.

9.1.2 API

Users of SHiM get access to all the features provided by BuildIt in addition to the UniTeX features. The segments below highlight the main operations related to UniTeX.

`Block<type,N>::heap(origin,extents,refinement,coarsening,permutation)` creates a new heap-allocated N-dimensional block (see Subsection 9.3.4) with the specified location parameters. SHiM includes overloads of this function taking different combinations of the parameters.

`Block<type,N>::stack<extents>(origin,refinement,coarsening,permutation)` creates a new stack-allocated N-dimensional block (see Subsection 9.3.4) with the specified location parameters. SHiM includes overloads of this function taking different combinations of the parameters. Extents are specified as template parameters to ensure that the generated code generates an array with constant parameters, since C++ does not support variable length arrays.

`Block<type,N>::external(origin,extents,refinement,coarsening,permutation)` creates a new N-dimensional block pointing to a user-allocated region of memory (see Subsection 9.3.4) with the specified location parameters. SHiM includes overloads of this function taking different combinations of the parameters.

`tensor.partition(range(x0,y0,z0),...,range(xN,yN,zN))` partitions a tensor (see Subsection 5.2.3), producing a new view. Here, the `x` parameters represent the start, `y` represents the stop, and `z` represents the stride. In place of the triples, a single integer or BuildIt `dyn_var` or `static_var` can be used. If the number of parameters is less than the dimensionality, SHiM pads the left with zeros. The start and stop may point out-of-bounds with respect to the tensor as well.

`tensor0.coloc(tensor1)` performs a colocation (see Subsection 5.2.4), producing a new view.

`tensor(c0,...,cN)` reads from a tensor at the specified indices (see Subsection 5.2.6). The indices can be integers or BuildIt `dyn_var` or `static_var` objects, but cannot be `Iter` objects (see Subsection 9.3.3). If the number of parameters is less than the dimensionality, SHiM pads the left with zeros. The start and stop may point out-of-bounds with respect to the tensor and perform a locality access.

`tensor[c0][...][cN]` reads from a tensor at the specified indices (see Subsection 5.2.6). This is similar to the read function above, but also supports `Iter` objects for indices (see Subsection 9.3.3).

`tensor[c0][...][cN] = val` writes to tensor at the specified indices (see Subsection 5.2.6). The indices can be integers, BuildIt `dyn_var` or `static_var` objects, or `Iter` objects (see Subsection 9.3.3). If the number of parameters is less than the dimensionality, SHiM pads the left with zeros. The start and stop may point out-of-bounds with respect to the tensor and perform a locality access.

`tensor.vrefine(r0,...,rN)` performs a refinement in each dimension (see Subsection 5.2.2), producing a new view. The refinement factors `r0,...,rN` can be integers or BuildIt `dyn_var` or `static_var` objects.

`tensor.vcoarsen(c0,...,cN)` performs a coarsening in each dimension (see Subsection 5.2.2), producing a new view. The coarsening factors `c0,...,cN` can be integers or BuildIt `dyn_var` or `static_var` objects.

`tensor.vpermute(p0,...,pN)` performs a permutation (see Subsection 5.2.2), producing a new view. The permutation factors `p0,...,pN` must be integers.

`Iter` defines an elementwise iterator (see Subsection 9.3.3), modeling those used in `CoLa`.

Note that `SHiM` does not include an explicit slice operation. Instead, `SHiM` automatically applies padding to read, write, partition, and colocation operations that take into account any differences in dimensionality (which is sufficient for the uses of slicing in the encoders described here).

9.1.3 Code Examples

Figure 9.1 gives an example of the JPEG color conversion kernel in `SHiM`, taking advantage of elementwise iterators (like `CoLa`'s), which are defined with the `Iter` objects and shown on Lines 11 to 13. It also uses permutations to abstract the underlying data layout (Line 4).

This code also highlights the interface between an external user program that would call the code generated from this kernel. In particular, on Line 2, the provided `SHiM` function wraps some user-defined pointer that represents the image data (wrapped in a `dyn_var` in order to make it work with BuildIt). Now this external pointer can be used like a regular `SHiM` tensor.

`SHiM` can also generate the 1D DCT code from Figure 6.3 using permutations as shown in Figure 9.2 on Page 110. Here, `SHiM` also inserts padding into the accesses on `vec` (Lines 5 to 9), which implements the functionality of the UniTe slice operation.

The example in Figure 9.3 on Page 110 performs the necessary checks for the vertical right mode of intra-prediction originally discussed in Subsection 6.3.3. This code utilizes `SHiM` partitions (Lines 9 and 10), colocations (Lines 11 and 12),

```
1  void jpeg(dyn_var<uint8_t*> image) {
2    auto RGB = Block<uint8_t,3>::external({H, W, 3}, image);
3    ...
4    color(RGB.vpermute({2,0,1}), YCbCr);
5    ...
6  }
7  template <typename RGB_T, typename YCbCr_T>
8  void color(RGB_T &RGB, YCbCr_T &YCbCr) {
9    Iter<'i'> i;
10   Iter<'j'> j;
11   YCbCr[0][i][j] = RGB[0][i][j]*0.299+RGB[1][i][j]*0.587+RGB[2][i][j]*0.114;
12   YCbCr[1][i][j] = RGB[0][i][j]*-0.168736+RGB[1][i][j]*-0.33126+RGB[2][i][j]*0.500002)+128;
13   YCbCr[2][i][j] = RGB[0][i][j]*0.5+RGB[1][i][j]*-0.418688+RGB[2][i][j]*-0.081312)+128;
14 }
```

Figure 9.1: `SHiM` code for a color conversion kernel in JPEG (See Subsections 2.1.3 and 6.2.2), and initial construction of the RGB image. This code originally builds up the RGB image based on an external allocation provided by the user (allowing this code to be inserted into an existing codebase), then permutes it to convert it to planar format. The actual conversion functions utilizes elementwise iterators (`Iter`) to create implicit loops, and then performs basic tensor accesses.

```
template <typename Plane_T>
void dct1d(Plane_T &plane, dyn_var<int> f) {
  for (dyn_var<int> r = 0; r < 8; r=r+1) {
    auto vec = plane.partition(range(r,r+1,1),range(0,8,1));
    auto tmp0 = vec(0) + vec(7);
    auto tmp7 = vec(0) - vec(7);
    ...
    vec[3] = scale(tmp6+z2+z3, f);
    vec[1] = scale(tmp7+z1+z4, f);
  }
}
template <typename Plane_T>
void dct(Plane_T &plane, dyn_var<int> scale_row, dyn_var<int> scale_col) {
  dct1d(plane, scale_row);
  dct1d(plane.vpermute({0,2,1}), scale_col);
}
```

Figure 9.2: SHiM code for a DCT (see Subsections 2.1.4 and 6.2.3). This code relies on permutations to implement a single 1D DCT kernel that can compute on either rows or columns. Rather than providing an explicit slice operation, SHiM automatically inserts zero padding for the outermost indices (`vec(7)` becomes `vec(0,0,7)`), which provides the same functionality.

```
template <typename Frame_T>
void predict(Frame_T &frame) {
  auto coarsened_frame = frame.vcoarsen({16,16}).to_block();
  auto used_intra_pred = coarsened_frame.vrefine({16,16});
  ...
}
template <typename Frame_T, typename MBlk_T mblk, typename Pred_T pred, typename Used_T>
void can_do_mode_VR(Frame_T &frame, MBlk_T mblk, Pred_T &pred, Used_T &used_intra_pred) {
  auto row_up = pred.partition(range(-1,0,1),range(0,4,1));
  auto col_left = pred.partition(range(0,4,1),range(-1,0,1));
  auto row_origin = row_up.coloc(frame).vorigin();
  auto col_origin = col_left.coloc(frame).vorigin();
  if (row_origin[0] > 0 && row_origin[1] > 0) {
    auto flags = used_intra_pred[pred];
    dyn_var<bool> up_left = flags(-1,-1) == INTRA;
    dyn_var<bool> up = flags(-1,0) == INTRA;
    dyn_var<bool> left = flags(0,-1) == INTRA;
    return up_left && up && left;
  } else {
    return false;
  }
}
```

Figure 9.3: Utilizing coarsening and refinement in SHiM to determine whether a mode of prediction is possible (see Subsection 6.3.3). SHiM provides a straightforward translation of the pseudocode in Figure 6.8 into an implementation, making it possible to access `used_intra_pred` at pixel-level granularity (even though it stores data at macroblock-level granularity).

and locality accesses (Lines 15 to 17). The final example in Figure 9.4 on Page 111 shows code for the vertical right mode itself. Utilizing a combination of permutations (Lines 5 and 6), elementwise iterators (Lines 10, 12, 14 and 15), and special elementwise conditionals (Lines 9, 11 and 13), SHiM is able to provide a near one-to-one mapping with the standard for this operation.

```
template <typename Pred, typename Ref>
void get_4x4_vert_right(Pred &predr, Ref &ref) {
  Iter<'x'> x;
  Iter<'y'> y;
  auto p = ref.coloc(predr).vpermute(1,0);
  auto pred = predr.vpermute(1,0);
  auto zVR = 2*x-y;
  pred[x][y] =
    select(zVR == Or(0,2,4,6),
           (p[x-(y>>1)-1][-1]+p[x-(y>>1)][-1]+1)>>1,
           select(zVR == Or(1,3,5),
                  (p[x-(y>>1)-2][-1]+2*p[x-(y>>1)-1][-1]+p[x-(y>>1)][-1]+2)>>2,
                  select(zVR == -1,
                         (p[-1][0]+2*p[-1][-1]+p[0][-1]+2)>>2, (p[-1][y-1]+2*p[-1][y-2]+
                          p[-1][y-3]+2)>>2)));
}
```

Figure 9.4: SHiM code for computing the vertical right mode of 4x4 intra-prediction. This code utilizes a combination of permutations, elementwise reads/writes, colocation, and locality accesses.

9.2 Overview of the BuildIt Library

BuildIt [21] is a C++ library for multi-stage programming that can capture control flow without the need for any changes to C++ or the C++ compiler. Since C++ does not offer any type of reflection nor control flow overloading capabilities, embedded DSLs within C++ traditionally have to take a more declarative approach and implicitly represent control flow (such as loop nests) through other means such as lambdas. If control flow represents a large portion of a given program in a DSL, this approach can lead to compensating for control flow with overly complicated code.

During execution of a staged program, BuildIt constructs an AST that captures all operations using BuildIt types, with operations ranging from simple arithmetic expressions to control flow through for/while loops and conditionals. BuildIt introduces three data types, `dyn_var<T>`, `static_var<T>`, and `dyn_arr<T,N>`, which are wrappers to the underlying `T` types (and `T[N]` for `dyn_arr<T,N>`). In the context of staging, `dyn_var<T>` objects get passed through to the generated code, while `static_var<T>` objects are evaluated during staging. The individual elements in `dyn_arr<T,N>` objects are treated as `dyn_var<T>` objects, and the actual array does not live through to code generation. In the context of control flow, BuildIt utilizes implicit conversions and operator overloading on `dyn_var<T>` and `static_var<T>` objects to infer their use within a program. Figure 9.5 on Page 112 shows a toy example of some code utilizing BuildIt and the resulting generated code, which fully unrolls the `static_var`, but leaves the loop and arithmetic expressions with `dyn_var`s. Note that BuildIt does not actually *execute* the loops with `dyn_var` or `static_var` loop induction variables (i.e., it is not running the for loop with the `static_var` five times); it only *generates* the code representing the loop.

```
dyn_var<int> func(dyn_var<int**> obj,
                  dyn_var<int> Y) {
  static_var<int> X = 5;
  dyn_var<int> res = 0;
  dyn_var<int> y = 0;
  static_var<int> x = 0;
  for (y; y < Y; y=y+1) {
    for (x; x < X; x=x+1) {
      res += obj[y][x];
    }
  }
  return res;
}
```

```
int func(int** arg0, int arg1) {
  int var2 = 0;
  for(int var3=0; var3<arg1; var3=var3+1) {
    var2 = var2 + arg0[var3][0];
    var2 = var2 + arg0[var3][1];
    var2 = var2 + arg0[var3][2];
    var2 = var2 + arg0[var3][3];
    var2 = var2 + arg0[var3][4];
  }
  return var2;
}
```

Figure 9.5: Code written using BuildIt (left), and the code ultimately generated from BuildIt's AST after staging (right). BuildIt unrolls the inner loop due to the use of a `static_var`, but leaves the outer loop and expressions using `dyn_var<int>`.

9.3 Implementation

SHiM takes a very different implementation approach to UniTeX as compared to CoLa, but the goals of the implementation are the same: provide a straightforward API to UniTeX and remove the overhead incurred by the abstractions. SHiM provides additional features such as elementwise writes (like in CoLa) and memory allocation abstractions. For transformations and optimizations, SHiM exploits the staging framework provided by BuildIt and applies many of its transformations *during* AST construction as opposed to transforming the AST itself. In particular, SHiM uses a two-stage pipeline, where the first stage represents the user's program written with SHiM and BuildIt, and the second stage represents the generated code corresponding to the user's program. Figure 9.6 gives the high-level pipeline for staging code written with SHiM.

As discussed in Chapter 7, a C++ library suffers from unacceptable performance overheads, thus SHiM needs to ensure that the generated code removes the overhead as well. Through careful integration with BuildIt, a program written with SHiM captures all the UniTeX functionality specified in the user code, as well as any user control flow[1] that may (or may not) encapsulate the UniTeX operations. Since BuildIt constructs an AST, SHiM could technically proceed in a similar fashion to CoLa and apply all of its transformations on the AST using full-fledged compiler passes (e.g. collapsing and propagation). However, SHiM

1: Assuming users used the correct BuildIt types for their control flow

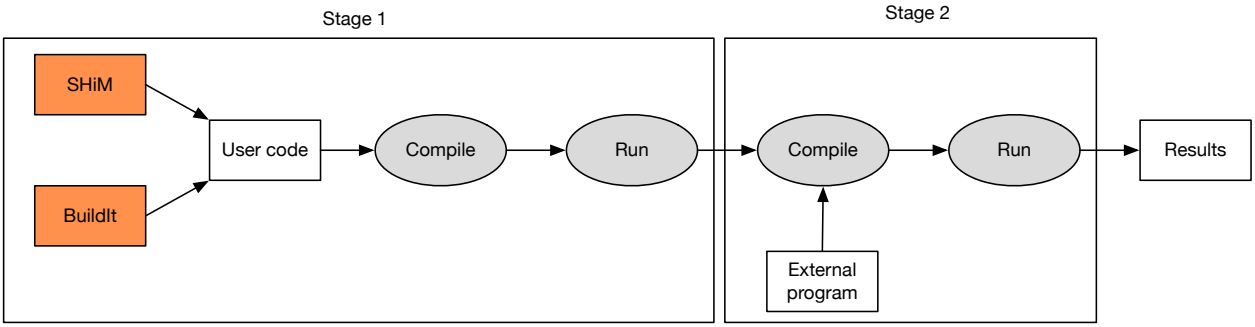


Figure 9.6: The SHiM execution pipeline, which utilizes two stages. The first stage contains the user's C++ code, where users can interleave SHiM and BuildIt types and operations. Then the stage generates code from the BuildIt AST. This generated code is free of SHiM and BuildIt types. The second stage is where the user would actually run their program (and optionally link the generated code into a larger program).

builds on BuildIt's datatypes such that all SHiM data structures and operations are fully inlined within the produced AST. This results in the generated code from staging also being fully inlined, which makes it trivial for another compiler (e.g. clang) to aggressively apply constant propagation, constant folding, and strength reduction on the generated code. Of course, it is still possible to run passes on the AST itself for any SHiM-specific transformations. Thus, it is useful to differentiate between *AST construction transformations*, which modify how the AST is initially constructed (covering the majority of SHiM's operation), and more traditional *AST transformations*, which modify the AST after construction. This section primarily looks at AST construction transformations, though SHiM provides the necessary tooling for performing AST transformations as well[2].

2: SHiM performs some AST transformations, but these are mostly for inserting some function wrappers that are needed in the generated code. These types of transformations do not add anything to the discussion, thus will be omitted.

With this design, SHiM sits somewhere between a fully templated C++ library approach and a runtime C++ library approach. A fully templated library (where the templates store constant property values) could theoretically provide all the necessary constant folding, propagation, etc., but would be unable to adapt to any values only known at runtime. A runtime library, again, supports both constant and dynamic values, but then has the issues with overhead. SHiM gives the best of both worlds without requiring any differences in how users specify constant or dynamic values.

9.3.1 Code Generation Example

Before getting into the details on the internals of SHiM, this section gives a preview of what code generated using SHiM's implementation of UniTeX looks like. The top of Figure 9.7 on Page 114 shows a simple example of SHiM code that creates a heap allocated block (Line 5) and writes to it (Line 7) using a mixture of an explicit loop (which BuildIt captures in its AST) and implicit elementwise loops (see Subsections 9.3.3 and 9.3.4), and a permutation. The bottom of Figure 9.7 shows the relevant portions of the code generated by SHiM[3]. Note that the while loops in the generated code are from the expansion of the elementwise writes; BuildIt does not necessarily guarantee that the generated code will use the same control flow structures as in the user code, but guarantees that the semantics will be the same.

3: Much of the generated code is omitted as it contains many redundant variable initializations. These are easily optimized away.

There are several important parts of the generated code to notice in this example:

Tensor parameters are inlined as shown on Lines 3 to 5, which represent the extents of the block created. This exposes constant parameters for propagation and folding, which as discussed in CoLa, are necessary for improving performance.

Operations are inlined and all accesses are reduced to direct accesses on the underlying array, as shown by the write on Line 22. Combined with the inlined property values, this exposes further opportunities for constant propagation, as well as constant folding. In addition, none of the tensor objects pass through to code generation. They exist only in the frontend.

SHiM generates necessary allocation code as shown on Line 7. SHiM supports several different types of allocations and generates the appropriate code for each type.

SHiM generates implicit loops for the elementwise operations, which are shown by the while loops.

As mentioned earlier, SHiM effectively achieves the same type of optimizations as CoLa in terms of exposing the necessary code for further optimization by an existing compiler, hence the aggressive inlining on the generated code.

```
// SHiM code
void func() {
  Iter<'j'> j;
  Iter<'k'> k;
  auto block = Block<int,3>::heap({5,10,15});
  for (dyn_var<int> i = 0; i < 5; i=i+1) {
    block.vpermute(2,0,1)[i][j][k] = i;
  }
}
```

```
// Generated code
...
int var48 = 5;
int var49 = 10;
int var50 = 15;
...
shim::HeapArray<int32_t> var85 = shim::build_heaparr<int32_t>(var72);
for (int var86 = 0; var86 < 5; var86 = var86 + 1) {
  while (1) {
    if (var341 < var100) {
      int var364 = 0;
      while (1) {
        if (var364 < var102) {
          ...
          var436 = ((var391 * var106) + var103) / var109;
          var437 = ((var392 * var107) + var104) / var110;
          var438 = ((var393 * var108) + var105) / var111;
          ...
          var457 = (var445 - var3) / var6;
          var458 = (var446 - var4) / var7;
          var459 = (var447 - var5) / var8;
          var85[var459 + (var62 * (var458 + (var61 * var457)))] = var86;
          var364 = var364 + 1;
        } else {
          break;
        }
...
```

Figure 9.7: SHiM UniTeX operations before (top) and after code generation (bottom). The generated code shows the allocation, loop structures, and writes to the block. Note that the code is fully inlined, with no reference to the block object.

9.3.2 Design of SHiM Structures

To generate the inlined code shown previously, SHiM heavily utilizes the BuildIt types to represent the block and view data structures and all of their parameterizations. Figure 9.8 on Page 115 gives a peek at the layout of SHiM's Properties data structure, which corresponds to the parameters defined for UniTe in Chapter 4. This structure forms the core of SHiM's tensors and stores all the tensor space parameters in dyn_arr objects, except for permutations, which are stored in C++ std:array. The dyn_arr objects do not live through to code generation–they exist to store arrays of dyn_var objects during staging. This means any

```
template <unsigned long Rank>
struct Properties {
  dyn_arr<int,Rank> _extents;
  dyn_arr<int,Rank> _origin;
  dyn_arr<int,Rank> _strides;
  dyn_arr<int,Rank> _refinement;
  array<int,Rank> _permutations;
};
```

Figure 9.8: Representation of the core data structure in SHiM that contains all the parameters for a tensor space. All parameters are stored in `dyn_arr` objects (except for permutations, which use a regular `std:array`–this is just an implementation quirk). Note that `_strides` is a legacy name and operates the same as coarsening.

operations or accesses to a `dyn_arr` instead generate code corresponding to the `dyn_var` value they hold, which effectively amounts to inlining all accesses on `dyn_arr`. No explicit passes on the AST are necessary.

With this structure, it is possible to implement the operations for UniTeX in the frontend just like a library. Figure 9.9 shows the implementation for the refinement operation, which updates the various parameters as necessary. The implementation is straightforward, and through the structure of SHiM, generates the inlined code without the need for any special optimizations.

```
template <unsigned long Rank>
Properties<Rank> Properties<Rank>::refine(Property<Rank> refinement) {
  Property<Rank> new_extents;
  Property<Rank> new_origin;
  Property<Rank> new_refinement;
  for (static_var<int> i = 0; i < Rank; i=i+1) {
    new_extents[i] = _extents[i] * refinement[i];
    new_origin[i] = _origin[i] * refinement[i];
    new_refinement[i] = _refinement[i] * refinement[i];
  }
  return {new_extents, new_origin, _strides, new_refinement, this->
    _permutations};
}
```

Figure 9.9: Implementation for the refinement operation in SHiM. All UniTeX components in SHiM have a straightforward library implementation and do not require any special AST passes for optimization.

9.3.3 Elementwise Writes

Like CoLa, SHiM provides the ability to generate writes to tensors through the use of elementwise iterators[4]. However, unlike CoLa, SHiM does not need any explicit compiler passes to perform the transformation; instead, it uses an AST construction transformation, generating the correct loops during execution of the first stage. Figure 9.10 on Page 116 gives a working example for this section and shows a case with two iterators.

4: See Subsection 8.3.2 for a discussion on the purpose of elementwise iterators with respect to ambiguity.

Since SHiM does not change any of the syntax of C++, it cannot use the iterator style of CoLa (the `!x` syntax). Instead it introduces `Iter` objects (similar to `Var` objects in Halide [50]). Semantically, these are the same as the CoLa iterators such that they represent the current iteration of the implicit loop, and have integer type so that they can be used in any expression that takes an integer. SHiM also places the same constraints on the iterators: any iterator on the right-hand side

```
Iter<'i'> i;
Iter<'j'> j;
tensorA[i][2][j] = tensorB[i+j] + tensorC[i];
```

(a) Two elementwise writes using `Iter` objects.

```
// First peel
for (dyn_var<int> i = 0; i < tensorA.vextents()[0]; i=i+1) {
}
// Second peel
for (dyn_var<int> i = 0; i < tensorA.vextents()[0]; i=i+1) {
  dyn_var<int> c = 2;
}
// Third peel
for (dyn_var<int> i = 0; i < tensorA.vextents()[0]; i=i+1) {
  dyn_var<int> c = 2;
  for (dyn_var<int> j = 0; j < tensorA.vextents()[2]; j=j+1) {
  }
}
```

(b) Peeling off indices to generate loop nests from (a).

```
for (dyn_var<int> i = 0; i < tensorA.vextents()[0]; i=i+1) {
  dyn_var<int> c = 2;
  for (dyn_var<int> j = 0; j < tensorA.vextents()[2]; j=j+1) {
    tensorA.write({i,2,j}, tensorB(i+j) + tensorC(i));
  }
}
```

(c) Adding in the write statement from (a) into (b).

Figure 9.10: Example of the process of generating loops from `Iter` objects in `SHiM`. See the text in Subsection 9.3.3 for a full discussion of each step.

must be present on the left-hand side, left-hand side iterators must be unadorned, and iterators on the left-hand side must be unique.

`SHiM` constructs the right-hand side of an elementwise statement lazily using expression templates [54, 56–58]. Expression templates are traditionally used for lazy evaluation in order to reduce the number of temporaries necessary. For example, if `d=a+b+c` represents the addition of three matrices, eager evaluation of this would evaluate `tmp=a+b`, then `d=tmp+c`. This issue of temporaries is not so much of an issue with `SHiM`'s particular use case, but expression templates provide a straightforward way for `SHiM` to generate the full loop nest. Rather than focus on the explicit use of expression templates, the rest of this description for elementwise writes will focus on how `SHiM` generates the loop nests from the constructed statement.

Once `SHiM` has verified that an expression contains a valid use of elementwise iterators, it performs a two-step procedure until all loops have been generated for the expression. The first step generates the loop nest, while the second step executes the actual statement. All of this happens during the first stage, and the transformations happen within the write functions; no explicit AST transformations are necessary.

To generate the loop nest, `SHiM` "peels" off indices from the left-hand side and replaces them one-by-one with their corresponding loop nest (or a constant if not using an iterator). Similar to `CoLa`, `SHiM` does this by determining which dimension the loop corresponds to and uses that for the extent of the loop nest.

From Figure 9.10a, there are three different "peels": one for `i`, constant index `2`, and `j`. Since `2` is just a constant, that does not produce a loop nest and instead just creates a `dyn_var` representing the value. Applying this process produces the code in Figure 9.10b.

Once SHiM has generated the loops, it unwraps the right-hand side expression. This involves applying all the terms captured within the expression template and replacing uses of the `Iter` objects with their corresponding loop iterators. This generates the code in Figure 9.10c. By default, SHiM places the full right-hand side expression within the innermost level of the loop nest even if parts of the expression can be hoisted.

9.3.4 Memory Allocation Abstractions

SHiM provides several different ways to generate code for the underlying block allocations. Users can have SHiM generate code for performing the allocation automatically, or can instruct it to use an existing memory allocation. The latter provides a necessary degree of flexibility that makes it easy to integrate kernels generated with SHiM into existing implementations because the kernels can utilize memory allocated by the existing implementation.

SHiM defines four primary ways to specify the type of allocation used[5], which are shown in Figure 9.11 on Page 118. A heap allocation, shown on Line 5, generates code for a reference-counted array, and a stack allocation, shown on Line 6, generates code for a plain stack array.

5: Note that SHiM does not actually allocate any of this memory itself during staging. Rather, it generates the code necessary to perform the allocation.

Lines 7 and 8 show two ways to specify that SHiM should *not* perform an allocation itself and instead utilize an existing allocation (represented by the two `dyn_var` pointers passed into the function). This abstraction not only provides a way to utilize already-allocated data, but also provides a solution to cases where an existing implementation is not consistent with the way it represents blocks. For example, in this case, one 2D block has type `int*` and the other has `int**`. It is possible to wrap each in a 2D block, allowing both to be intuitively accessed with 2D indices, even though the former is stored linearly. The type of allocation does not affect the user's view of the data, and the API for accessing and operating on blocks (and corresponding views) does not change.

9.4 Evaluation

The fully inlined nature of the generated code from SHiM allows it to be very easily optimized by an external compiler, thus SHiM, like CoLa, is able to generate code that ultimately removes the overhead associated with the UniTeX abstraction. This section highlights SHiM's performance on two different encoder benchmarks.

9.4.1 Benchmarks

The evaluation for SHiM includes the same sequential JPEG example discussed in CoLa's evaluation in Section 8.4 (JPEGS) and compares against the same reference (IJG). It also includes H.264 and compares against JM. However, instead

```
1  dyn_var<int> func(dyn_var<int*> ext, dyn_var<int**> ext2,
2                    dyn_var<int> Y, dyn_var<int> X) {
3    Iter<'y'> y;
4    Iter<'x'> x;
5    auto block_heap = Block<int,2>::heap({10,10});
6    auto block_stack = Block<int,2>::stack<10,10>();
7    auto block_ext = Block<int,2>::external({Y,X}, ext);
8    auto block_ext2 = Block<int,2,true>::external({Y,X}, ext2);
9    block_heap[y][x] = 1;
10   block_stack[y][x] = 1;
11   block_ext[y][x] = 1;
12   block_ext2[y][x] = 1;
13   return 0;
14 }
```

Figure 9.11: Four different ways of allocating memory with SHiM. The heap allocation on Line 5 generates code allocating a reference-counted array, while Line 6 generates code for a simple stack-allocated array. On Lines 7 and 8, SHiM takes in externally-allocated arrays rather than generating allocations itself. The former expects a single pointer, while the latter supports a pointer-to-pointer allocation (and more generally, takes in an N-level pointer, where N corresponds to the number of dimensions specified).

of implementing the full encoder for comparison, individual kernels are implemented in SHiM, and the generated code is linked into JM[6], replacing the original functions. In particular, the JM kernels related to 4x4 and 16x16 intra-prediction are replaced with SHiM kernels. The evaluation setup (Subsection 8.4.2) and data (Tables 8.2 and 8.3) are the same as presented in CoLa as well. Unlike CoLa, SHiM's optimizations cannot be disabled, so SHiM produces a single result per individual benchmark that just represents the optimized runtime.

6: Note that this usage of JM uses different parameters than the JM benchmark for CoLa, hence the overall performance difference when compared to the runtime for JM in the CoLa chapter.

9.4.2 Runtime Performance

Figures 9.12 and 9.13 on Page 119 show the performance of CoLa relative to the IJG and JM reference, respectively, and Table 9.1 on Page 120 gives the corresponding raw runtimes. Overall, the results are similar to what was shown for CoLa, with SHiM reaching parity with the reference implementations. This performance shows that SHiM effectively exposes all the underlying constant property values, allowing the external compiler (clang in this case) to optimize any unnecessary operations/simplify others.

9.5 Summary

This chapter introduced SHiM, which implements the full UniTeX abstraction as an embedded C++ domain-specific language utilizing staging through the BuildIt library. Through BuildIt, SHiM provides the ability to interleave C++ control flow with SHiM-specific data structures and operations, which is not typically supported with other C++-embedded domain-specific languages. Unlike CoLa, SHiM is able to perform many of its transformations and optimizations without the need for separate compiler passes.

While SHiM supports the ability to implement end-to-end encoders, SHiM also makes it easy to implement compression kernels that take in external data allocations and wrap them in SHiM block data structures. These kernels can be seamlessly integrated with existing code, allowing users to replace certain parts

of encoders rather than having to design an encoder from scratch. Regardless of whether SHiM is used for an end-to-end encoder, or for generating kernels, SHiM is able to achieve runtime parity with existing implementations.

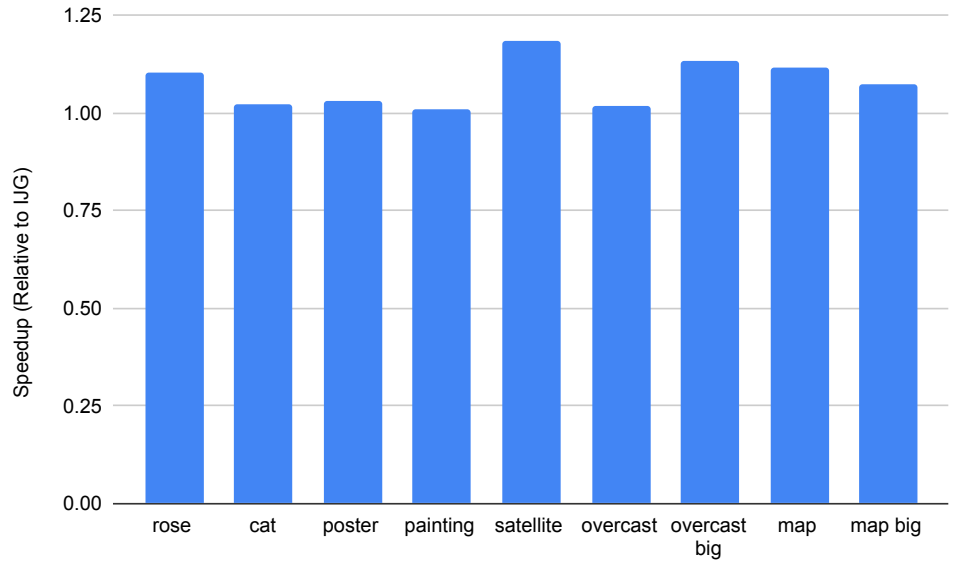


Figure 9.12: SHiM performance on JPEGS relative to IJG.

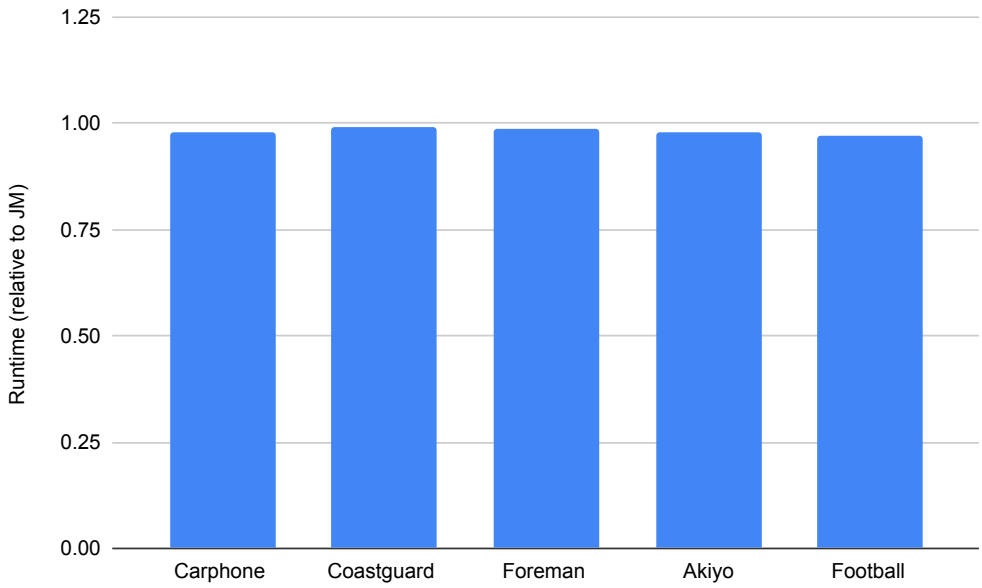


Figure 9.13: SHiM performance H.264 relative to JM.

Table 9.1: Raw runtimes for SHiM.

	SHiM	IJG
rose	0.004	0.004
cat	0.028	0.029
poster	0.196	0.186
painting	0.277	0.278
satellite	0.875	1.030
overcast	1.014	1.008
overcast big	1.905	2.181
map	2.466	2.750
map big	6.497	7.054

(a) SHiM vs IJG for JPEG (seconds).

	SHiM	JM
Carphone	2.670	2.617
Coastguard	3.169	3.146
Foreman	9.674	9.553
Akiyo	8.597	8.419
Football	10.554	10.231

(b) SHiM vs JM for H.264 (seconds).

	SHiM	JM
Carphone	75	76
Coastguard	63	64
Foreman	21	21
Akiyo	23	24
Football	19	20

(c) SHiM vs JM for H.264 (frames per second..

Related Works 10

Works related to this dissertation come from both application-specific research, as well as more general research focused on programming with arrays. This chapter first explores related work (or lack thereof) for block-based compression (Section 10.1), then moves on to array programming (Section 10.2). The array programming section looks at both more traditional uses of array programming, which largely deal with applying operations across arrays, as well as view-based programming (Subsection 10.2.1), which focuses on different ways to capture both structured and arbitrary data within arrays. Finally, this chapter concludes with a look at the domains of adaptive mesh refinement (Subsection 10.3.1) and geographic information systems (Subsection 10.3.2), which share aspects of spatial relationships with block-based compression.

10.1 Block-Based Compression

Decades of research for block-based compression have led to numerous implementations for end-to-end systems from both open source developers and companies alike. However, the majority of work on encoders focuses on the algorithmic side, as opposed to providing language support.

10.1.1 End-to-End Implementations

Some of the most prevalent open-source implementations include the reference software released with the standards, such as JM [8] (H.264), HM [11] (H.265), VTC [13] (H.266), and libjpeg [55] (JPEG), libvpx [14] (VP8/VP9). Many other implementations are based on these, thus share similar structure, but are typically implemented from scratch every time. While it is relatively easy to find highly optimized end-to-end encoders[1], such as nvJPEG [59], libjpeg-turbo [17], x264 [9], x265 [12], and x266 [60]), it is nearly impossible to find any language support to help build these implementations.

1: There tends to be a tradeoff in the implementation space between performance and how many features are implemented, with more performance corresponding to less features and vice versa. Thus being able to try different algorithms and such will likely require hand-implementation of the algorithm, or potentially even the whole system if the algorithm does not fit into the existing structure.

10.1.2 Decoder Support

Even language support for decoders, which are generally more straightforward to implement than encoders, is scarce. Some abstractions for MPEG decoders appear in the work on Flavor [61–63], XFlavor [64], BFlavor [65, 66], MPEG21-BDSL [67], StreamIt [68], StreamJIT [69], and the Reconfigurable Media Codec (RMC) [6, 7]. Flavor and its variants, along with the work on MPEG21-BDSL, focus on providing abstractions that simplify parsing a compressed MPEG bitstream, which is the first step in any decoder. Flavor provides a DSL in the style of C++ for declaratively describing the structure of the bitstream, while the others utilize primarily XML-based descriptions that act as input to RMC. RMC provides a more general dataflow model for decoders and provides users with a control flow graph

where they can mix-and-match different computational kernels (called functional units) throughout. StreamIt and StreamJIT model the MPEG2 video decoder with structured streams. These dataflow platforms largely focus on describing the overall pipeline of the decoder as opposed to the internals of stages.

10.2 Array Programming

Array programming has a much richer history of language support, beginning with the introduction of multidimensional arrays in Fortran [18]. A vast amount of libraries and languages have been developed over that span of time, with many focused on numerical and scientific computing. Irrespective of the application domain, languages and libraries for array programming constantly provide new ways to represent, transform, compute on, and optimize arrays. Several prior works capture various parts of UniTe and UniTeX, such as using views, but none fully encapsulate the semantics in this dissertation nor provide an underlying framework for spatial relationships among data.

Many languages are array-based or have wide ranging support for multidimensional arrays, with more popular ones including Julia [70], APL [71], Ada [72], MATLAB [73], SaC [74], and J [75]. Several libraries also include support such as NumPy [76], Eigen [77], Blaze [78], Blitz++ [79], Armadillo [80], and FTensor [81]. These systems provide a variety of tools for manipulating arrays and performing arithmetic operations on them, so have operations that can be useful for compression, but none provide abstractions for fully capturing spatial relationships. For example, NumPy and Julia provide native support for views and various types of manipulation (slicing, partitioning, etc.), but do not expose any of the location information for views related to their blocks, nor do they provide a global concept of location, which limits their usefulness for an application like compression[2].

2: The JuliaImages [49] and PaddedViews.jl [82] libraries in Julia provide some related work on accessing surrounding data (though not referred to by any specific name). For example, the libraries use negative indices to define relative offsets from an origin, but in this case, an index outside the bounds of a view returns a padding value as opposed to the data in the surrounding block. This could provide an extension to locality access on a block in UniTeX where it defines boundary conditions that dictate what to return on a truly out-of-bounds access. These types of boundary condition semantics are commonly found in image processing languages, such as Halide [50].

In a sense, this dissertation does not provide an abstraction for a traditional array-based programming, as that typically refers to applying operations on all the elements of the arrays at once. For example, `A*B` in many of these languages performs a matrix multiplication. These types of operations are orthogonal to UniTe as UniTe focuses on the relationships across these multidimensional arrays instead of the computations on them. However, both `CoLa` and `SHiM` can easily support this type of functionality through overloading magic methods (for `CoLa`), operator overloading (for `SHiM`), or elementwise iterators.

10.2.1 View Programming

Other systems with multidimensional arrays focus on the view/hierarchical aspect, which is more closely aligned with this work. Marray [83] provides a hybrid templated/runtime library implementation in C++ that treats strided views as a core primitive. Velociraptor [84] provides a framework for building JIT compilers for multidimensional tensors and views, providing an array-based IR. The work on Flexible Data Views [85] utilizes templates for C++ and dynamic staging through Lightweight Modular Staging [86] for Scala and focuses on providing views that cover disjoint regions of data. Hierarchically tiled arrays (HTAs) [87–89] provide a library approach to partitioning blocks and strided views for the purpose of

parallelism. Taichi [90] focuses on the hierarchical structure of disjoint, sparse data. Finally, the View Template Library [91] represents views in a more functional manner, utilizing smart iterators [92] which are C++ iterators with attached predicates. This provides the most flexible representation of a view as it can arbitrarily select data for a view depending on the evaluation of the predicate across each element[3]. However, for domains that have more constrained representations of views, this type of flexibility would lead to significant overheads. Across all of these, Marray is the only work that provides a theoretical foundation for the types of structure underlying their arrays and the operations on them.

3: For example, an iterator-based view (not in any particular language) would look something like `view=block.keep_if(fun x -> x > 10)`, where `view` represents only the elements of `block` that are larger than 10.

Like the general array-programming works, none of these capture all the semantics and functionality described in this dissertation. There is also a fundamental mismatch between how many of these prior works view the hierarchical structure of the data and the way this work views it. To understand the mismatch, consider the HTA library introduced earlier. HTA stores blocks and views in a hierarchy and provides various ways to partition the data. However, their focus is more on local, structured decomposition of the data, such as performing a quadtree decomposition on an array into tiles and then immediately executing all the tiles in parallel[4]. As a result, all of the operations are more focused on providing users with a simple way to decompose their data in this way and optimizing these localized usages. While that is extremely helpful for many applications, it is not as useful within compression.

4: HTAs provide support for computing and transferring boundary values as well.

Compression has elements of more structured decomposition in it (for example, partitioning macroblocks into submacroblocks in H.264 follows a quadtree decomposition), but the usages of the individual partitions are largely spread out across various stages, and also mixed with more ad-hoc partitions (like accessing a single row above a submacroblock to see if it exists), making it hard to extract any sort of simple, localized representation. While UniTe can already be used to extract this localized representation, it does not necessarily mean the tensors have the ability to be parallelized. With HTAs, there is an underlying assumption that the partitions are done for the purposes of parallelization, not necessarily for providing a more intuitive view of the data like with UniTe. In addition, many of the dependencies cannot be captured with simple boundary conditions. For example, conditions such as the one in H.264 checking whether a macroblock used intra-prediction or inter-prediction, introduce other complex dependencies between data that are actually based on runtime data decisions, not just the location. However, in the case that this type of hierarchical structure proves beneficial, UniTeX already provides the necessary semantics to capture the local structure, determine dependencies, etc.

10.3 Related Domains: Adaptive Mesh Refinement and Geographic Information Systems

Although this dissertation largely focuses on compression, the underlying abstractions span beyond that area and can be used as a foundation for representing data in related domains. Two particularly relevant domains are that of adaptive mesh refinement and geographic information systems, as UniTeX can represent the core data in each, as well as many of the operations.

10.3.1 Adaptive Mesh Refinement

Adaptive mesh refinement (AMR), particularly that of block structured adaptive mesh refinement, represents a hierarchy of grids derived from one another, where grids at the lower levels have greater resolution (refinement) and grids at the higher levels have less resolution (coarsening). At a high-level, AMR iteratively applies refinement to the grid until some stopping criterion is met. However, an entire grid is not usually refined, rather smaller rectangular regions within each are refined instead, leading to rectilinear patches of different refinement. A layer in a grid is the combination of all the patches having the same refinement, and AMR requires being able to globally represent all of these patches within the grid. Existing libraries for structured AMR, such as SAMRAI [93], Paramesh [94], and BoxLib/AMReX [95, 96] each utilize their own hand-rolled representation for these data structures, even though each effectively needs the same representation.

UniTeX provides a natural connection to this type of AMR and can serve as a foundation for representing the grid and patches within it. It has the necessary semantics for refinement, and could easily determine all the patches in a layer by computing the refinement of each tensor space with respect to the root of the trie. Since UniTe also supports arbitrary sets of points, it is possible to use UniTe to describe more than just tensors, thus it can also represent these rectilinear patches[5].

5: It is possible to take the representation a step further and represent non-rectilinear shapes, such as triangles. Chapter 11 discusses this more when looking at potential future work.

10.3.2 Geographic Information Systems

Geographic information systems (GIS) are a broad class of tools that store, visualize, and analyze geospatial data, with ArcGis [97] and QGIS [98] being the most widely used implementations of GIS. A core class of analyses used in GIS, known as spatial statistics, compute various statistics (such as min/max) across different pieces of data that have some spatial connection between them, such as proximity. For a particular example, consider one category of spatial statistics known as *zonal statistics*[6].

6: See https://pro.arcgis.com/en/pro-app/latest/tool-reference/spatial-analyst/how-zonal-statistics-works.htm

A zonal statistic computes some value on a *value raster* (the input data) based on a *zonal raster*, which effectively acts as a mask that indicates which data in the value raster to use in computing the statistic. These rasters are made up of cells, and the zonal raster groups cells into *zones*, where different zones each have the statistic computed independently on them. As with AMR, the zones can define rectilinear regions, but are not strictly limited to that shape. Figure 10.1 on Page 125 gives an example of the setup for computing a sum zonal statistic.

Like AMR, UniTeX contains the necessary semantics to represent this data. However, unlike AMR, these zonal statistics also have an element of colocation in them, as they need to determine the data in the value raster corresponding to the location of the zonal raster. This would be possible with UniTe by extending it in a similar way to UniTeX and defining colocation on arbitrary sets of points.

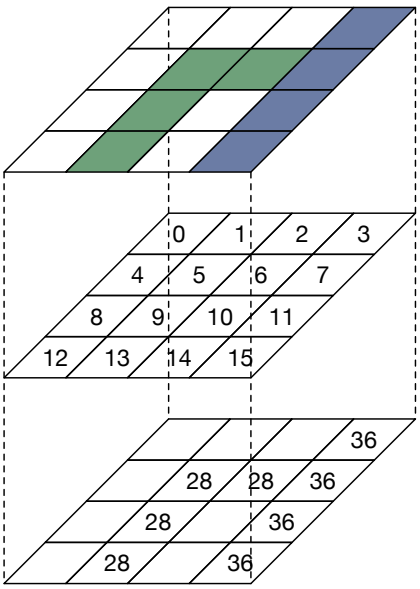


Figure 10.1: A zonal statistic for calculating the sum of elements in each zone. The top raster defines the individual zones, which each have the statistic (sum) computed independently on them. The middle raster defines the values to use for the statistic, and the bottom layer represents the result of summing the data corresponding to each zone.

11 Conclusion and Future Work

This dissertation showed how to capture, implement, and apply spatial relationships across tensors within a program. It introduced a mathematical framework, UniTe, which provides a way to capture and reason about location on arbitrary sets of points, as well as tensors, which represent bounded sets of points. UniTe makes it possible to capture the location of data with different representations using reference spaces, where the representations can vary in their axes orientation, dimensionality, origin, and point density (refinement and coarsening). Together with a set of definitions that perform mappings across reference spaces, UniTe provides well-defined ways to map data between reference spaces and compute spatial relationships across them.

In addition to UniTe, this dissertation introduced UniTeX, which defines higher-order operations on top of UniTe that exploit spatial relationships. These operations specify how to both create new reference spaces (e.g. copy, permute, slice, refine, coarsen, partition, colocation) and access data (e.g. locality access, coverage). UniTe and UniTeX aim to provide a concrete framework for defining and utilizing location, but are flexible enough that they can be extended to support new features that may be necessary with certain domains. The goal of these abstractions is not to constrain tensors to a particular set of parameters, but rather provide the structure necessary to fit parameters within the abstraction (i.e. requiring mappings between reference spaces).

To demonstrate the use of the abstraction, this work applied the abstraction to block-based compression, which heavily relies on spatial relationships across tensors all throughout encoders. Block-based compression largely lacks any sort of programming language support, but UniTe and UniTeX provide the necessary data representations that can be used to build such support. However, this work showed that a library implementation can incur up to a 65× slowdown compared to hand-optimized C code due to the overhead of indexing and creating views.

To remove this overhead, this dissertation presented two different domain-specific languages, `CoLa` and `SHiM`, which are built off of UniTe and UniTeX. Both of these provide an intuitive interface for users while also removing the overhead and bringing performance down to parity with the existing implementations; however, both take very different approaches in doing so. In particular, `CoLa` follows a more traditional path, using domain-specific compiler passes to remove the overhead, while `SHiM` utilizes staging, which generates code that an existing compiler (such as clang) can instead successfully optimize.

This work is ultimately a necessary first step in enabling programming language support for block-based compression, and there are many paths for continuing research in both the abstraction and block-based compression. Some potential avenues include:

Structured encoder pipelines Chapter 2 introduced the high-level pipelines and stages found in encoders for JPEG and H.264 and discussed both the underlying data representations and operations, as well as how the stages connect to one another. The work in this dissertation targets simplifying the

implementation of the stages themselves, but does not address how the stages connect to one another. Another source of complexity in existing encoder implementations comes from hooking the stages together, as processing data throughout an encoder pipeline is largely non-linear (for example, reconstructed data must be fed back into the pipeline for use with the next block of data).

For this, there is potential in the area of structured stream processing where the tensors defined in UniTe can be used to define the data elements consumed and produced by nodes in the stream graph. This structure could simplify building an encoder by providing a sort of "plug-and-play" framework. While the difficulty lies in the dynamism that comes with compression, there should be ways to leverage the overall structure within the compression pipelines to make it fit within such a framework. Existing structured stream processing works such as that with StreamIt [68], StreamJIT [69], and the Reconfigurable Media Codec [6, 7] would be ideal starting points for this type of work as they define many primitive node types that would be useful for encoders.

Bitstream abstractions A somewhat unexpected difficulty encountered when writing implementations for compression was correctly formatting the output bitstreams. The sheer number of parameters that have to be written to the bitstream quickly gets overwhelming, as many dependencies exist across the parameters, and can only be figured out by jumping back and forth between pages of the standard. A useful DSL would be one that could be used to encode the structure of a bitstream in a library, where the library provides functions representing bitstream primitives that continuously validate the produced bitstream (e.g. check for valid values, valid orderings, etc.). Having such a tool, even just for the purposes of debugging, would undoubtedly reduce the implementation effort for the encoder itself.

Expanding into other domains As discussed in Chapter 10, the abstractions presented in this dissertation also apply to domains such as adaptive mesh refinement and geographic information systems, which utilize spatial relationships across different reference spaces. Understanding the differences between these domains and the current version of UniTe can help expand it with new types of reference spaces and operations exploiting spatial relationships, ultimately adding to the power of the representation.

Expanding UniTe and UniTeX Both UniTe and UniTeX are ultimately tied to tensors, which can be used to define both rectangular and rectilinear regions of data[1]. However, other domains (including adaptive mesh refinement and geographic information systems defined earlier) utilize non-rectangular regions of data, such as triangles. Since the foundations of UniTe just operate on sets of points, it would be possible to extend UniTe to these other types of shapes by just providing different bounds on the sets of points themselves. Similarly, other domains may require different types of parameterizations and mappings such as rotations and non-linear mappings. The reference spaces and point mappings of UniTe can be extended with these other features, as long as the core of UniTe is maintained (namely, reference spaces are defined relative to one another, points can be mapped between any reference space, etc.). The definition of UniTe used in this dissertation was picked to primarily support block-based compression, and adding other features would have added unnecessary overhead that would

1: Rectilinear regions can just be represented with a collection of tensors.

still need to be accounted for in the compilers. However, other domains can mix-and-match the necessary components to define their own spatial abstraction.

Adding spatial structures and operators to existing languages Spatial operations appear in many different applications, not just block-based compression, adaptive mesh refinement, and geographic information systems. In fact, spatial data structures such as quadtrees [99], octtrees [100], k-d trees [101], and R-trees [102] have been studied for decades and have numerous variants derived from each of them. While these spatial structures are all different, they still have a notion of location at their core and require maintaining relative locations within each. To the best of our knowledge, existing general purpose languages do not natively include any spatial primitives for building these structures. UniTe and UniTeX provide the necessary components for defining these primitives in a language, and the various considerations described in Chapters 7 to 9 provide some initial insight into the implementation details of such primitives.

Admittedly, adding new features to an existing general-purpose language can be a difficult path, however, other paths such as implementing UniTe and UniTeX into a library could prove to be a useful starting point. While Chapter 7 discussed the performance issues that come with a naive library implementation, it could be possible to use staging techniques to integrate UniTe efficiently into existing libraries such as NumPy[2].

2: NumPy kernels are written in C, thus could be specialized using staging.

As these domains that utilize spatial relationships continue to evolve and become more complex, there is an immediate need for better programming language support. This dissertation has demonstrated that it is possible to develop a flexible abstraction for capturing and reasoning about spatial relationships within a program. From this work, the hope is that it inspires others to take a serious look at ways to improve language support for both block-based compression and other related domains.

Bibliography

[1] Claude E Shannon. 'A mathematical theory of communication'. In: *The Bell system technical journal* 27.3 (1948), pp. 379–423 (cited on page 15).

[2] Jacob Ziv and Abraham Lempel. 'A universal algorithm for sequential data compression'. In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343 (cited on page 15).

[3] Jacob Ziv and Abraham Lempel. 'Compression of individual sequences via variable-rate coding'. In: *IEEE transactions on Information Theory* 24.5 (1978), pp. 530–536 (cited on page 15).

[4] David A Huffman. 'A method for the construction of minimum-redundancy codes'. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101 (cited on pages 15, 28).

[5] James E Fowler. 'QccPack: An open-source software library for quantization, compression, and coding'. In: *Applications of digital image processing xxiii*. Vol. 4115. SPIE. 2000, pp. 294–301 (cited on page 15).

[6] Christophe Lucarz et al. 'Reconfigurable media coding: A new specification model for multimedia coders'. In: *2007 IEEE workshop on signal processing systems*. IEEE. 2007, pp. 481–486 (cited on pages 15, 121, 128).

[7] Joseph Thomas-Kerr et al. 'Reconfigurable media coding: Self-Describing multimedia bitstreams'. In: *2007 IEEE Workshop on Signal Processing Systems*. IEEE. 2007, pp. 319–324 (cited on pages 15, 121, 128).

[8] Joint Video Team. *JM software (v19.0)*. 2022. URL: http://iphome.hhi.de/suehring/ (cited on pages 16, 18, 23, 42, 43, 79, 80, 101, 121).

[9] multicoreware. *x264*. 2022. URL: https://multicorewareinc.com/video/ (cited on pages 16, 121).

[10] cisco. *OpenH264*. 2023. URL: https://github.com/cisco/openh264 (cited on pages 16, 80).

[11] Joint Collaborative Team on Video Coding. *HM software (v16.22)*. 2022. URL: https://vcgit.hhi.fraunhofer.de/jct-vc/HM (cited on pages 16, 121).

[12] multicoreware. *x265*. 2022. URL: https://multicorewareinc.com/video/ (cited on pages 16, 121).

[13] Joint Video Experts Team. *VTM software (v10.2)*. 2022. URL: https://vcgit.hhi.fraunhofer.de/jvet/VVCSoftware_VTM (cited on pages 16, 121).

[14] webmproject. *The WebM Project*. 2022. URL: %7Bhttps://www.webmproject.org/code/%7D (cited on pages 16, 121).

[15] Richter, Thomas. *libjpeg*. 2023. URL: %7Bhttps://github.com/thorfdbg/libjpeg%7D (cited on page 16).

[16] Independent JPEG Group. *JPEG software*. 2022. URL: https://ijg.org/ (cited on pages 16, 101).

[17] libjpeg-turbo. *libjpeg-turbo*. 2023. URL: https://libjpeg-turbo.org/ (cited on pages 16, 121).

[18] John Backus. 'The history of Fortran I, II, and III'. In: *ACM Sigplan Notices* 13.8 (1978), pp. 165–180 (cited on pages 16, 122).

[19] ITU-T. *Advanced Video Coding for Generic Audiovisual Services*. Tech. rep. H.264. Geneva, Switzerland: International Telecommunication Union, 2021 (cited on pages 18, 23, 29, 34, 77, 101).

[20] Ariya Shajii et al. 'Codon: A Compiler for High-Performance Pythonic Applications and DSLs'. In: *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. 2023, pp. 191–202 (cited on pages 20, 21, 85, 89, 92).

[21] Ajay Brahmakshatriya and Saman Amarasinghe. 'BuildIt: A type-based multi-stage programming framework for code generation in C++'. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 39–51 (cited on pages 20, 21, 85, 107, 111).

[22] CCITT. *Information Technology–Digital Compression and Coding of Continuous-Tone Still Images–Requirements and Guidelines*. Tech. rep. T.81. Geneva, Switzerland: International Telecommunication Union, 1992 (cited on pages 23, 24, 34, 101).

[23] Radiocommunication Sector of ITU. *Studio encoding parameters of digital television for standard 4:3 and widescreen 16:9 aspect ratios*. BT Series ITU-R BT.601-7. Geneva, Switzerland: International Telecommunication Union, 2011 (cited on page 25).

[24] David Salomon. *Data compression: the complete reference*. Springer Science & Business Media, 2004 (cited on pages 25, 26).

[25] Nasir Ahmed, T_ Natarajan, and Kamisetty R Rao. 'Discrete cosine transform'. In: *IEEE transactions on Computers* 100.1 (1974), pp. 90–93 (cited on page 27).

[26] Jan-Yie Liang et al. 'Lossless compression of medical images using Hilbert space-filling curves'. In: *Computerized Medical Imaging and Graphics* 32.3 (2008), pp. 174–182 (cited on page 28).

[27] Jorma J Rissanen. 'Generalized Kraft inequality and arithmetic coding'. In: *IBM Journal of research and development* 20.3 (1976), pp. 198–203 (cited on page 28).

[28] Ce Zhu, Xiao Lin, and Lap-Pui Chau. 'Hexagon-based search pattern for fast block motion estimation'. In: *IEEE transactions on circuits and systems for video technology* 12.5 (2002), pp. 349–355 (cited on page 33).

[29] Zhibo Chen et al. 'Fast integer-pel and fractional-pel motion estimation for H. 264/AVC'. In: *Journal of visual communication and image representation* 17.2 (2006), pp. 264–290 (cited on page 33).

[30] Shan Zhu and Kai-Kuang Ma. 'A new diamond search algorithm for fast block-matching motion estimation'. In: *IEEE transactions on Image Processing* 9.2 (2000), pp. 287–290 (cited on page 33).

[31] Jo Yew Tham et al. 'A novel unrestricted center-biased diamond search algorithm for block motion estimation'. In: *IEEE transactions on Circuits and Systems for Video Technology* 8.4 (1998), pp. 369–377 (cited on page 33).

[32] ITU-T. *Video Codec for Audiovisual Services at p×64 kbits*. Tech. rep. H.261. Geneva, Switzerland: International Telecommunication Union, 1993 (cited on page 34).

[33] JTC-1. *Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s*. Tech. rep. ISO/IEC 11172. Washington, DC: ISO/IEC, 1991 (cited on page 34).

[34] JTC-1. *The generic coding of moving pictures and associated audio information*. Tech. rep. ISO/IEC 13818. Washington, DC: ISO/IEC, 1996 (cited on page 34).

[35] ITU-T. *Video Ciding for Low Bit Rate Commnunication*. Tech. rep. H.263. Geneva, Switzerland: International Telecommunication Union, 2005 (cited on page 34).

[36] JTC-1. *Coding of Audio-Visual Objects*. Tech. rep. ISO/IEC 14496. Washington, DC: ISO/IEC, 1999 (cited on page 34).

[37] ITU-T. *Still-image compression – JPEG 2000*. Tech. rep. ISO/IEC 15444-1. Geneva, Switzerland: International Telecommunication Union, 2019 (cited on page 34).

[38] J Bankoski et al. *VP8 Data Format and Decoding Guide*. Tech. rep. RFC 6386. ISO/IEC, 2011 (cited on page 34).

[39] ITU-T. *JPEG Extended Range*. Tech. rep. ISO/IEC 29199-2:2020. Geneva, Switzerland: International Telecommunication Union, 2020 (cited on page 34).

[40] Google. *VP9*. Tech. rep. 2013 (cited on page 34).

[41] ITU-T. *High Efficient Video Coding*. Tech. rep. H.265. Geneva, Switzerland: International Telecommunication Union, 2021 (cited on page 34).

[42] ITU-T. *JPEG XS*. Tech. rep. ISO/IEC 21122. Geneva, Switzerland: International Telecommunication Union, 2019 (cited on page 34).

[43] ITU-T. *Versatile Video Coding*. Tech. rep. H.266. Geneva, Switzerland: International Telecommunication Union, 2022 (cited on page 34).

[44] ITU-T. *JPEG XL*. Tech. rep. ISO/IEC 18181. Geneva, Switzerland: International Telecommunication Union, 2022 (cited on page 34).

[45] Dylan Heuer and Jessica Striker. ‘Partial permutation and alternating sign matrix polytopes’. In: *SIAM Journal on Discrete Mathematics* 36.4 (2022), pp. 2863–2888 (cited on page 51).

[46] Saman P Amarasinghe and Monica S Lam. ‘Communication optimization and code generation for distributed memory machines’. In: *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. 1993, pp. 126–138 (cited on page 57).

[47] Corinne Ancourt and François Irigoin. ‘Scanning polyhedra with DO loops’. In: *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*. 1991, pp. 39–50 (cited on page 57).

[48] Martine Ancourt. ‘Generation automatique de codes de transfert pour multiprocesseurs a memoires locales’. PhD thesis. Paris 6, 1991 (cited on page 57).

[49] JuliaImages. *JuliaImages*. 2023. URL: %7B%22https://github.com/JuliaImages%22%7D (cited on pages 64, 122).

[50] Jonathan Ragan-Kelley et al. ‘Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines’. In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530 (cited on pages 64, 85, 95, 115, 122).

[51] the mypy project. *the mypy project*. 2023. URL: https://mypy-lang.org/ (cited on page 92).

[52] google. *A static type analyzer for Python code*. 2023. URL: https://github.com/google/pytype (cited on page 92).

[53] Bryan Ford. ‘Parsing expression grammars: a recognition-based syntactic foundation’. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2004, pp. 111–122 (cited on page 93).

[54] Todd Veldhuizen. ‘Expression templates’. In: *C++ Report* 7.5 (1995), pp. 26–31 (cited on pages 95, 116).

[55] Richter, Thomas. *libjpeg*. 2022. URL: https://github.com/thorfdbg/libjpeg (cited on pages 101, 121).

[56] Jochen Härdtlein, Alexander Linke, and Christoph Pflaum. ‘Fast expression templates’. In: *Computational Science–ICCS 2005: 5th International Conference, Atlanta, GA, USA, May 22-25, 2005. Proceedings, Part II 5*. Springer. 2005, pp. 1055–1063 (cited on page 116).

[57] Jochen Härdtlein et al. ‘Advanced expression templates programming’. In: *Computing and Visualization in Science* 13 (2010), pp. 59–68 (cited on page 116).

[58] Klaus Iglberger et al. ‘Expression templates revisited: a performance analysis of current methodologies’. In: *SIAM Journal on Scientific Computing* 34.2 (2012), pp. C42–C69 (cited on page 116).

[59] NVIDIA. *nvJPEG Libraries*. 2023. URL: https://developer.nvidia.com/nvjpeg (cited on page 121).

[60] multicoreware. *x266-VVC Encoder*. 2023. URL: https://multicorewareinc.com/what-we-do/audio-video-solutions/x266-vvc-encoder/ (cited on page 121).

[61] Alexandros Eleftheriadis. ‘The MPEG-4 system and description languages: from practice to theory’. In: *1997 IEEE International Symposium on Circuits and Systems (ISCAS)*. Vol. 2. IEEE. 1997, pp. 1480–1483 (cited on page 121).

[62] Alexandros Eleftheriadis and Danny Hong. ‘Flavor: a formal language for audio-visual object representation’. In: *Proceedings of the 12th annual ACM international conference on Multimedia*. 2004, pp. 816–819 (cited on page 121).

[63] Alexandros Eleftheriadis. ‘Flavor: a language for media representation’. In: *Proceedings of the fifth ACM international conference on Multimedia*. 1997, pp. 1–9 (cited on page 121).

[64] Danny Hong and Alexandros Eleftheriadis. ‘XFlavor: bridging bits and objects in media representation’. In: *Proceedings. IEEE International Conference on Multimedia and Expo*. Vol. 1. IEEE. 2002, pp. 773–776 (cited on page 121).

[65] Wesley De Neve et al. ‘BFlavor: a harmonized approach to media resource adaptation, inspired by MPEG-21 BSDL and XFlavor’. In: *Signal Processing: Image Communication* 21.10 (2006), pp. 862–889 (cited on page 121).

[66] Davy Van Deursen et al. 'BFlavor: an optimized XML-based framework for multimedia content customization'. In: *Proceedings of the 25th Picture Coding Symposium*. 2006 (cited on page 121).

[67] Gabriel Panis et al. 'Bitstream syntax description: a tool for multimedia resource adaptation within MPEG-21'. In: *Signal Processing: Image Communication* 18.8 (2003), pp. 721–747 (cited on page 121).

[68] William Thies, Michal Karczmarek, and Saman Amarasinghe. 'StreamIt: A language for streaming applications'. In: *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings 11*. Springer. 2002, pp. 179–196 (cited on pages 121, 128).

[69] Jeffrey Bosboom et al. 'StreamJIT: A commensal compiler for high-performance stream programming'. In: *ACM SIGPLAN Notices* 49.10 (2014), pp. 177–195 (cited on pages 121, 128).

[70] Jeff Bezanson et al. 'Julia: A fresh approach to numerical computing'. In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671 (cited on page 122).

[71] Kenneth E Iverson. 'A programming language'. In: *Proceedings of the May 1-3, 1962, spring joint computer conference*. 1962, pp. 345–351 (cited on page 122).

[72] John G Barnes. *Programming in ADA*. Addison-Wesley Longman Publishing Co., Inc., 1984 (cited on page 122).

[73] Desmond J Higham and Nicholas J Higham. *MATLAB guide*. SIAM, 2016 (cited on page 122).

[74] Sven-Bodo Scholz. 'Single Assignment C: efficient support for high-level array operations in a functional setting'. In: *Journal of functional programming* 13.6 (2003), pp. 1005–1059 (cited on page 122).

[75] Kenneth E Iverson. 'A personal view of APL'. In: *ACM SIGAPL APL Quote Quad* 30.3 (2000), pp. 4–13 (cited on page 122).

[76] Charles R. Harris et al. 'Array programming with NumPy'. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2 (cited on page 122).

[77] Gaël Guennebaud, Benoit Jacob, et al. 'Eigen'. In: *URl: http://eigen. tuxfamily. org* 3 (2010) (cited on page 122).

[78] Blaze. *blaze-lib*. 2023. URL: %7B%22https://bitbucket.org/blaze-lib/blaze/src/master/%22%7D (cited on page 122).

[79] blitzpp. *blitz*. 2023. URL: %7B%22https://github.com/blitzpp/blitz%22%7D (cited on page 122).

[80] Conrad Sanderson and Ryan Curtin. 'Armadillo: a template-based C++ library for linear algebra'. In: *Journal of Open Source Software* 1.2 (2016), p. 26 (cited on page 122).

[81] Walter Landry. *ftensor*. 2023. URL: %7B%22https://wlandry.net/Projects/FTensor/%22%7D (cited on page 122).

[82] JuliaArrays. *PaddedViews.jl*. 2023. URL: %7B%22https://github.com/JuliaArrays/PaddedViews.jl%22%7D (cited on page 122).

[83] Bjoern Andres et al. 'Runtime-flexible multi-dimensional arrays and views for C++ 98 and C++ 0x'. In: *arXiv preprint arXiv:1008.2909* (2010) (cited on page 122).

[84] Rahul Garg and Laurie Hendren. 'Velociraptor: An embedded compiler toolkit for numerical programs targeting cpus and gpus'. In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 317–330 (cited on page 122).

[85] Leo Osvald and Tiark Rompf. 'Flexible data views: design and implementation'. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. 2017, pp. 25–32 (cited on page 122).

[86] Tiark Rompf and Martin Odersky. 'Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs'. In: *Proceedings of the ninth international conference on Generative programming and component engineering*. 2010, pp. 127–136 (cited on page 122).

[87] Basilio B Fraguela et al. ‘The hierarchically tiled arrays programming approach’. In: *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*. 2004, pp. 1–12 (cited on page 122).

[88] Ganesh Bikshandi et al. ‘Programming for parallelism and locality with hierarchically tiled arrays’. In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2006, pp. 48–57 (cited on page 122).

[89] Ganesh Bikshandi et al. ‘Design and use of htalib–a library for hierarchically tiled arrays’. In: *Languages and Compilers for Parallel Computing: 19th International Workshop, LCPC 2006, New Orleans, LA, USA, November 2-4, 2006. Revised Papers 19*. Springer. 2007, pp. 17–32 (cited on page 122).

[90] Yuanming Hu et al. ‘Taichi: a language for high-performance computation on spatially sparse data structures’. In: *ACM Transactions on Graphics (TOG)* 38.6 (2019), pp. 1–16 (cited on page 123).

[91] Gary Powell and Martin Weiser. ‘Views, A New Form of Container Adaptors’. In: *C/C++ Users Journal* (Apr. 1998) (cited on page 123).

[92] Thomas Becker. ‘Smart iterators and stl’. In: *C/C++ Users Journal* 16.9 (1998), pp. 39–45 (cited on page 123).

[93] LLNL. *SAMRAI: Structured Adaptive Mesh Refinement Application Infrastructure*. 2023. URL: https://computing.llnl.gov/projects/samrai (cited on page 124).

[94] Peter MacNeice et al. ‘PARAMESH: A parallel adaptive mesh refinement community toolkit’. In: *Computer physics communications* 126.3 (2000), pp. 330–354 (cited on page 124).

[95] Weiqun Zhang et al. ‘Boxlib with tiling: An adaptive mesh refinement software framework’. In: *SIAM Journal on Scientific Computing* 38.5 (2016), S156–S172 (cited on page 124).

[96] LBNL, NREL, and ANL. *AMReX-Codes*. 2023. URL: https://amrex-codes.github.io/amrex/ (cited on page 124).

[97] ESRI. *ArcGis*. 2023. URL: https://www.esri.com/en-us/landing-page/product/2019/arcgis-online/overview (cited on page 124).

[98] QGIS. *QGIS*. 2023. URL: https://qgis.org/en/site/ (cited on page 124).

[99] Raphael A Finkel and Jon Louis Bentley. ‘Quad trees a data structure for retrieval on composite keys’. In: *Acta informatica* 4 (1974), pp. 1–9 (cited on page 129).

[100] Donald Meagher. ‘Geometric modeling using octree encoding’. In: *Computer graphics and image processing* 19.2 (1982), pp. 129–147 (cited on page 129).

[101] Jon Louis Bentley. ‘Multidimensional binary search trees used for associative searching’. In: *Communications of the ACM* 18.9 (1975), pp. 509–517 (cited on page 129).

[102] Antonin Guttman. ‘R-trees: A dynamic index structure for spatial searching’. In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 1984, pp. 47–57 (cited on page 129).