CoolerSpace: A Language for Physically Correct and Computationally Efficient Color Programming

ETHAN CHEN, University of Rochester, USA
JIWON CHANG, University of Rochester, USA
YUHAO ZHU, University of Rochester, USA

Color programmers manipulate lights, materials, and the resulting colors from light-material interactions. Existing libraries for color programming provide only a thin layer of abstraction around matrix operations. Color programs are, thus, vulnerable to bugs arising from mathematically permissible but physically meaningless matrix computations. Correct implementations are difficult to write and optimize. We introduce CoolerSpace to facilitate physically correct and computationally efficient color programming. CoolerSpace raises the level of abstraction of color programming by allowing programmers to focus on describing the logic of color physics. Correctness and efficiency are handled by CoolerSpace. The type system in CoolerSpace assigns physical meaning and dimensions to user-defined objects. The typing rules permit only legal computations informed by color physics and perception. Along with type checking, CoolerSpace also generates performance-optimized programs using equality saturation. CoolerSpace is implemented as a Python library and compiles to ONNX, a common intermediate representation for tensor computations. CoolerSpace not only prevents common errors in color programming, but also does so without run-time overhead: even unoptimized CoolerSpace programs out-perform existing Python-based color programming systems by up to 5.7 times; our optimizations provide up to an additional 1.4 times speed-up.

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Computing methodologies → Computer graphics**.

Additional Key Words and Phrases: language design, color science, type systems

ACM Reference Format:
Ethan Chen, Jiwon Chang, and Yuhao Zhu. 2024. CoolerSpace: A Language for Physically Correct and Computationally Efficient Color Programming. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 301 (October 2024), 30 pages. https://doi.org/10.1145/3689741

1 Introduction

Color programming broadly refers to the programmatic manipulation of lights, materials (e.g. pigments), and the resulting color of light-material interactions. Color programming is fundamental to almost every domain of art, science, and engineering. Imaging and display technologies are, in essence, about capturing and reproducing colors [Miller and Spicer 2019; Rowlands 2017; Sharma 2017]; computer graphics simulate light-material interaction and color capturing in cameras [Pharr et al. 2023]; artists use vibrant palettes of colors, both real and digital, to create their works [Sochorová and Jamriška 2021], while art conservators analyze and preserve the original pigments in historical pieces [Berns 2016; Johnston-Feller 2001].

Authors' Contact Information: Ethan Chen, University of Rochester, Rochester, USA, echen48@ur.rochester.edu; Jiwon Chang, University of Rochester, Rochester, USA, jchang38@ur.rochester.edu; Yuhao Zhu, University of Rochester, Rochester, USA, yzhu@rochester.edu.

ACM 2475-1421/2024/10-ART301
https://doi.org/10.1145/3689741

Color programmers must follow the rules of physics governing light-material interaction and the standards of different color encodings (Sec. 2). The languages (e.g., Python) and libraries (NumPy [Harris et al. 2020] and OpenCV [Bradski 2000]) they use, however, are physics-agnostic: they, in large part, provide only a thin wrapper around raw tensor operations. The physical meanings of objects (e.g., color, light power spectrum, material scattering spectrum) are not tracked. Thus, programmers are prone to accidentally writing mathematically permissible but physically meaningless or incorrect code. Physically correct code can be time consuming to implement and are not always computationally efficient (Sec. 3).

We propose CoolerSpace to facilitate physically correct and computationally efficient color programming (Sec. 4). The core of CoolerSpace is a type system (Sec. 5), which raises the level of abstraction of color programming from tensors to physical objects, such as lights, materials, and colors. The domain-specific typing rules, which are statically checked, permit only physically-based or perceptually accurate computations.

In addition to avoiding common errors, the higher level of programming abstraction also frees programmers from the burden of efficiently implementing color science algorithms. Instead, a CoolerSpace program is translated to a semantically-equivalent set of tensor algebra operations (Sec. 6), which are then optimized using equality saturation [Tate et al. 2009; Yang et al. 2021] (Sec. 8). Translation is guided by formal translational semantics that are provably type sound.

CoolerSpace assists any programmers working with color and light, and can be particularly useful for a significant scientific community that might not intersect with the conventional CS community: color scientists and vision scientists. These are domain experts who write programs to directly manipulate light and color data. They must do so in a physically accurate way. However, researchers in these fields are less familiar with modern programming techniques (e.g., type checking, performance optimizations). They are exactly the population CoolerSpace can help.

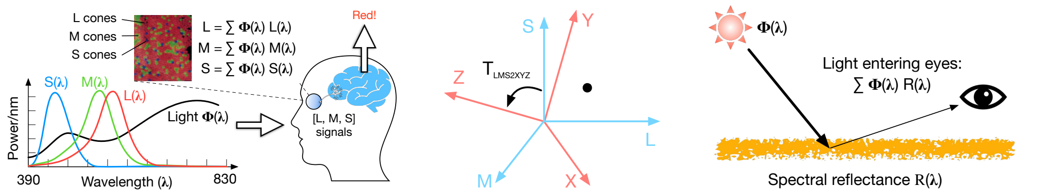
We implement CoolerSpace as a Python library, as color and vision scientists predominantly use high-level languages like Python and MATLAB. The Python program is compiled to ONNX [Onnx 2018], an intermediate representation for tensor algebra. The ONNX program is then executed using ONNX Runtime [Developers 2021]. We show that CoolerSpace can express common algorithms in color programming and prevent common errors without runtime overhead. Unoptimized CoolerSpace programs out-perform existing Python color programming systems by up to 5.6 times. CoolerSpace is capable of further optimizing its programs by up to 1.4 times.

The entire CoolerSpace system, along with programs developed using CoolerSpace, will be made open-source. Our specific contributions are as follows.

(1) We demonstrate a class of bugs in color programs where programmers write mathematically permissible but physically incorrect computations.
(2) We design a type system specific to color programming. The type system codifies and enforces the fundamental principles of color physics.
(3) We introduce CoolerSpace, which implements the type system and automatically generates efficient color programs through tensor algebra optimizations.
(4) We experimentally show that CoolerSpace prevents common bugs in color programming while providing up to 5.6 × speed-ups against existing Python color programming libraries.
(5) We also experimentally show that CoolerSpace's optimizer provides an additional 1.4 × speed-up to our compiled programs.

2 Background: Lights, Colors, and Materials

Color programming involves manipulating lights, materials, and different representations of color. This section provides the necessary scientific background.



(a) Color perception.

(b) Color encodings are different in different color spaces.

(c) Material color depends on light spectrum and material reflectance.

Fig. 1. (a): A light with a Spectral Power Distribution (SPD) $\Phi(\lambda)$ gets transformed into a triplet [L, M, S] on the retina, which represents the total responses of the Long, Medium, and Short cone photoreceptor cells. The brain interprets a combination of [L, M, S] as a color. (b): Geometrically, a color is a point in a 3D color space. We can change the basis of the coordinate system to derive a new color space. The same color is encoded differently in different color spaces. (c): Material color depends on both the spectrum of the incident light and the spectral reflectance of the material.

Lights. In the realm of color science, a light is physically represented by a Spectral Power Distribution (SPD) function, which describes the power distribution of the light over wavelengths. The $\Phi(\lambda)$ function in Fig. 1a illustrates one example. The SPD is defined over the visible spectrum, usually between 390 nm and 830 nm, and quantized into discrete intervals for ease of computation.

Colors. Humans perceive colors from lights because photons are absorbed by cone cells on the retina, which in turn generate neural responses. These responses are interpreted by the brain as a particular color. A cone's behavior is described by its Spectral Sensitivity Function (SSF) [Wandell 1995], which represents the neural response generated per unit power at a particular wavelength.

There are three kinds of cone cells that are responsible for color vision, each with a unique SSF that peaks at long, medium, and short wavelengths, respectively; these cone cells are thus called the L, M, and S cones. $L(\lambda)$, $M(\lambda)$, and $S(\lambda)$ in Fig. 1a show the SSFs of the three cone cells.

An incident light's power, after retinal processing, gets converted into three numbers, i.e., the total L, M, and S cone responses stimulated by the light. Mathematically, this is:[1]

$$[L, M, S] = [\sum_{\lambda=390}^{830} \Phi(\lambda)L(\lambda), \sum_{\lambda=390}^{830} \Phi(\lambda)M(\lambda), \sum_{\lambda=390}^{830} \Phi(\lambda)S(\lambda)] \tag{1}$$

where $\Phi(\lambda)$ is the SPD of the incident light. This can be understood as weighting the light SPD by the cone sensitivity per wavelength and summing the weighted responses over the visible spectrum.

Our brain, over time, learns to associate an [L, M, S] triplet with a color. Thus, a color can be represented as a point in a 3D (LMS) space. This is the fundamental reason why human color perception is trichromatic.

In addition to the LMS space, there are many other color spaces in which a color can be represented. Geometrically, this amounts to changing the basis of a coordinate system and re-expressing the same color in the new space. For instance, the most commonly used color space in color science is the CIE 1931 XYZ color space [Brainard and Stockman 2010], which is a new coordinate system that is a linear transformation ($T_{LMS2XYZ}$) away from the LMS space. This is illustrated in Fig. 1b.

[1]The summation is sometimes also written as an integration [Marschner and Shirley 2021]. We choose summation to reflect the actual computation performed in programs.

A color in the LMS space $[L_c, M_c, S_c]$ can be re-expressed as $[X_c, Y_c, Z_c]$ in the XYZ space by:

$$[X_c, Y_c, Z_c]^T = T_{LMS2XYZ} \times [L_c, M_c, S_c]^T \tag{2}$$

Gamma. The LMS space is a *linear* color space, in that the channel values are proportional to light power. For instance, if the light power is doubled across the spectrum, the resulting channel values will simply double accordingly. However, color spaces used to encode digital images are usually *non-linear*. Common examples include the popular sRGB and opRGB color spaces. In these color spaces, channel values are proportional to *perceived brightness*, which is non-linearly related to light power. The non-linear transformation between light power and brightness is called Gamma correction/encoding [Poynton 2012]. Gamma correction is governed by a single value γ[2]:

$$\alpha = \beta^{\frac{1}{\gamma}} \tag{3}$$

α represents the gamma-encoded color space, and β represents its linear counterpart.

Materials. Much of the light entering our eyes is reflected off materials. The apparent color of a material depends on the light striking the material and the material's physical properties. The simplest phenomenological model of a material is the spectral reflection function $R(\lambda)$, which describes how much light is reflected back at a given wavelength λ[3]. Given an incident light with an SPD $\Phi(\lambda)$, the SPD of the reflected light is $\sum_{\lambda=390}^{830} \Phi(\lambda)R(\lambda)$, as illustrated in Fig. 1c.

Physically, the reason a light gets reflected back is the complicated interaction of photons being absorbed and/or scattered by particles inside the material. The absorption and scattering behavior of a material is modeled by the spectral absorption function $K(\lambda)$ and spectral scattering function $S(\lambda)$. The reflectance spectrum $R(\lambda)$ is related to the absorption and scattering spectra through the Kubelka-Munk model [Kubelka 1948; Kubelka and Munk 1931]. The model is expressed in Equ. 4.

$$R(\lambda) = 1 + \frac{K(\lambda)}{S(\lambda)} - \sqrt{\frac{K(\lambda)^2}{S(\lambda)^2} + 2\frac{K(\lambda)}{S(\lambda)}} \tag{4}$$

All functions are spectra, indicating that the scattering and absorption capability of a material (and thus the reflection) depend on the wavelength of the incident light. The advantage of modeling materials using scattering and absorption spectra is that it allows us to easily simulate the color of pigment mixing. The absorption and scattering coefficients of homogeneous mixtures are modeled as a weighted average of that of the constituent materials [Duncan 1940]. Specifically, when mixing N materials, each with a spectral absorption and scattering function of $K_i(\lambda)$ and $S_i(\lambda)$, respectively, the resulting mixture has a spectral absorption and scattering function of:

$$K_{mix}(\lambda) = \frac{1}{C}\sum_{i}^{N} K_i(\lambda) \times C_i,\ S_{mix}(\lambda) = \frac{1}{C}\sum_{i}^{N} S_i(\lambda) \times C_i \tag{5}$$

where C_i denotes the weight concentration of the i^{th} material. C is the sum of all individual concentrations. The models are the scientific bases of simulating material mixing in CoolerSpace.

3 Motivations

Color programs are often written using libraries like Colour-Science [Developers 2015] and NumPy [Harris et al. 2020]. These libraries provide only a thin wrapper around raw matrix operations. The physical meanings of objects (e.g., color, light power spectrum, material scattering

[2]There also exist piece-wise Gamma functions.

[3]For simplicity we assume the surface is diffuse, where reflection is angle insensitive; phenomenological models such as BRDF and BSSDF [Pharr et al. 2023] used for modeling non-diffuse surfaces can be similarly supported.

spectrum) are not tracked by these tools [Stefan 2017]. Programmers are, thus, responsible for keeping track of the physical meanings manually. This burden can lead to a variety of subtle bugs.

The consequence of such bugs is often "silent data corruption", as physically incorrect operations do not usually lead to program crashes. These bugs are tricky to catch, motivating the need to type-check color programs. Users complain when they observe an undesirable output, as discussed in a Google IO talk [Guy 2017].

Physically Meaningless Operations. Without meaningful type information, programmers are prone to defining arithmetic operations that are physically meaningless but mathematically permissible. For instance, common libraries provide code for color space conversion, but do not check whether such conversion is performed on the correct color space. The code snippet in Prog. 1, written using the popular Colour-Science Python library [Developers 2015], executes without complaint but is incorrect.

```
1  image = open_image('srgb_image.png')  # sRGB image
2  colour.XYZ_to_Lab(image) # should use sRGB_to_XYZ prior to XYZ_to_Lab
```

Program 1. Incorrect Translation from `sRGB` to `LAB` space.

In this example, a programmer intends to convert an image encoded in the `sRGB` color space to the CIELAB color space, but the `sRGB` color data is inadvertently, and incorrectly, treated as data encoded in the CIEXYZ color space on line 2. Nothing prevents programmers from making this mistake. Needless to say, translating an `sRGB` image to a `LAB` image as if the former were encoded in the `XYZ` space is not physically meaningful. The program still executes without complaint, as the operation is mathematically permissible.

Incorrect color space handling is a common issue reported by programmers [McCurdy 2022]. For instance, a PyTorch programmer says, "*there is no way for the library to know if the input tensor that you are passing is indeed in rgb colorspace. So you can silently get wrong results if you are not careful*" [Massa 2021]. Similarly, a scikit-image user says, "*scikit-image operates on numpy arrays exclusively, so we have no way of knowing metadata about the image*" [Stefan 2017].

One might also want to simulate light reflection off a surface using the light spectrum and the material reflectance spectrum as shown in Fig. 1c, but may accidentally multiply two light spectra, as they have the same data dimension. Such a multiplication makes no physical sense.

Incorrect Understanding of Color Science. Another class of bugs arises from incorrect understanding of color science, ranging from a lack of understanding of radiometry (physics) vs. photometry (human perception) [Pharr et al. 2023], to mistakenly equating mixing lights with mixing materials [Sochorová and Jamriška 2021].

Consider a scenario where one wants to estimate the color of mixing two lights represented in the `sRGB` color space. A naive programmer may simply add the `sRGB` values, as seen in Prog. 2.

```
# Assuming both images are encoded in sRGB
image1 = open_image('image1.png')
image2 = open_image('image2.png')

# Physically meaningless but permitted operation
mixed_img = image1 + image2
```

Program 2. Incorrect `sRGB` light addition.

This code is incorrect, because, as discussed in Sec. 2, `sRGB` is a non-linear color space: `sRGB` channel values are not proportional to light power. To accurately simulate the mixing of two colored lights, the addition must take place in a linear color space. The code is shown in Prog. 3.

```
# Convert sRGB image to linear sRGB
```

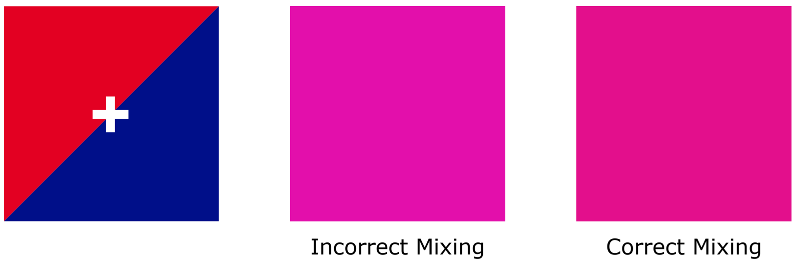


Fig. 2. Mixing of red (`sRGB` [227, 0, 34]) and blue (`sRGB` [0, 15, 137]) lights in the non-linear sRGB space (incorrect) vs. in a linear color space (correct).

```
image1_linear = (image1 / 255) ** 2.2
image2_linear = (image2 / 255) ** 2.2

# Add in linear space
mixed_linear = image1_linear + image2_linear

# Re-apply gamma
mixed = (mixed_linear ** (1 / 2.2)) * 255
```

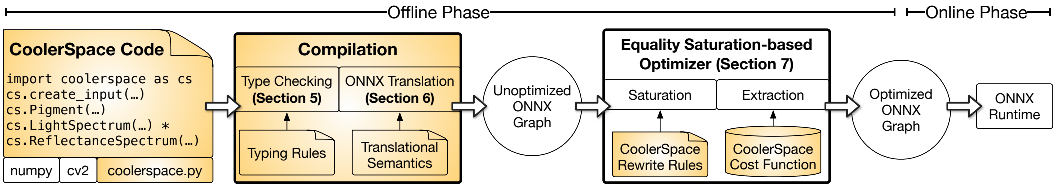
Program 3. Correct `sRGB` light addition.

Incorrectly performing linear physics operations in a non-linear color space is a common issue of color programming in the wild [yet 2011; wro 2014, 2016; thi 2021]. Programmers usually have to manually track whether data are encoded in a physically linear space. For instance, three.js warns programmers that "*it's important that the working color space be linear and the output color space be nonlinear*" [McCurdy 2022], without providing any checks.

"Silent data corruptions" from incorrect color manipulations are often too subtle to catch. Fig. 2 compares the outputs of the naive interpolation and the correct interpolation: the results are visually similar. It can be hard to distinguish the output of unprincipled color programs from properly written color programs. Bugs can easily pass human scrutiny.

Implementation Details and Speed Concerns. The correct program to add two lights represented in the `sRGB` color space, shown in Prog. 3, requires a non-trivial amount of complexity. However, the program is semantically simple — the program mixes two colored lights. The complexity of physically accurate color manipulation can and should be abstracted from programmers.

Isolating the logic of color physics from its implementation has another key advantage: it frees programmers from optimizing for performance. Color programs operate on large datasets. For reference, an uncompressed one minute full HD (1920 × 1080 pixels) video filmed at 60 frames per second constitutes over 22 GB of data. Due to the immense dataset sizes, color programmers are sensitive to slow run-times. The absence of optimization can be a deal breaker. One user of the Colour-Science python library writes, "*this library is incredible but a lot of functions seem to be really slow*" [Ebenezer 2021]. Similar sentiments have been expressed by other developers [adriahf 2016; Lavrov 2020]. Operating on such a large dataset means that even small optimizations can yield sizable execution time reductions. Manually writing optimal code is not always obvious, and should be left to a compiler.



```
import coolerspace as cs
cs.create_input(…)
cs.Pigment(…)
cs.LightSpectrum(…) *
cs.ReflectanceSpectrum(…)
```

Fig. 3. CoolerSpace overview. The colored components are introduced by CoolerSpace whereas the rest are existing tools. CoolerSpace is a Python-based meta-programming system: the Python program gets compiled and optimized into another program, an ONNX graph, which executes on the ONNX Runtime [Developers 2021]. Compilation and optimization are done once (offline phase), so they introduce only a one-time cost.

```
import coolerspace as cs
light1 = cs.create_input(
    shape=[1920,1080],
    colorspace=cs.LightSpectrum
)
light2 = cs.create_input(
    shape=[1920,1080],
    colorspace=cs.LightSpectrum
)
color1 = cs.sRGB(light1)
color2 = cs.sRGB(light2)
color3 = color1 + color2
cs.create_output(color3)
```

Program 4. A CoolerSpace program, where we create two sets of light spectra, which are cast to the `sRGB` type. The two sets of colored lights are then mixed. Notice how the programmer has to consciously unify the color space but is free from writing and optimizing the actual color mixing code.

4 CoolerSpace System Overview

CoolerSpace is a Python-based meta-programming system. The key goal of CoolerSpace is to allow programmers to focus on the logic of color physics while relying on the programming system to guarantee correctness and to optimize for efficiency. Fig. 3 provides an overview of the system. This section walks through the pipeline at a high level and highlights our major design decisions.

Language. From a programmer's perspective, CoolerSpace is a Python library. Prog. 4 shows a simple program written in CoolerSpace. We choose Python as our host language as it is the lingua franca of color programmers, who make use of libraries such as OpenCV [Bradski 2000], Colour-Science [Developers 2015], and NumPy [Harris et al. 2020].

Compilation. The crux of CoolerSpace is a type system, which assigns types to user-defined objects. The types store both the physical meaning and dimension of the data. For example, the type of a full HD `sRGB` image would store the dimensions of the image (1920×1080) and the color encoding (`sRGB`). Types are used to enforce physical and dimensional correctness through static type checking. The type system is formalized in Sec. 5.

During compilation, the Python program is "executed", and any operations between CoolerSpace objects are intercepted, type checked, but *not* evaluated. If a program type checks, CoolerSpace creates a corresponding ONNX program that is equivalent to the original CoolerSpace program. For instance, the second last line in Fig. 4, `color1+color2`, does not actually mix `sRGB` colors. Instead, the '+' operator is overloaded to type check that both colors are in the same color space

Table 1. Simplified CoolerSpace abstract syntax. We omit trivial operations such as indexing color channels. A complete syntax is defined in the supplemental material.

Arrays	$a \in$ floating point arrays
Variable Names	$x \in$ variable names
Tristimulus Color Types	$\tau_{\text{tristimulus}} ::= \tau_{\text{XYZ}} \vert \tau_{\text{LMS}} \vert \tau_{\text{sRGB}} \vert \tau_{\text{opRGB}}$
Perceptual Color Types	$\tau_{\text{perceptual}} ::= \tau_{\text{HSL}} \vert \tau_{\text{LAB}}$
Color Types	$\tau_{\text{color}} ::= \tau_{\text{tristimulus}} \vert \tau_{\text{perceptual}}$
Spectral Types	$\tau_{\text{spectrum}} ::= \tau_{\text{Light}} \vert \tau_{\text{Reflectance}} \vert \tau_{\text{Scattering}} \vert \tau_{\text{Absorption}} \vert \tau_{\text{Pigment}}$
Physical Types	$\tau ::= \tau_{\text{color}} \vert \tau_{\text{spectrum}} \vert \tau_{\text{Chromaticity}} \vert \tau_{\text{Matrix}}$
Dimension Types	$d ::= \mathbb{N} \vert d \times d$
Shaped Types	$s ::= (\tau, d)$
Values	$v ::= x \vert \tau(a) \vert \tau(v) \vert \tau(v, v) \vert \text{mix}(v, v, v, v) \vert v + v \vert v - v \vert v / v \vert$ $v \times v \vert \text{matmul}(v, v)$
Expressions	$e ::= x = v$
Programs	$P ::= e; P \vert e$

and of the same dimension. Afterwards, ONNX code is generated with the arithmetic for `sRGB` color mixing. The exact translation strategy is defined in Sec. 6.

We have chosen ONNX as a compilation target, becasue ONNX is a popular format for tensor algebra. There exists a vibrant community and ecosystem that provides cross-platform ONNX support [Developers 2021]. Given that color programs naturally manipulate tensors, mapping CoolerSpace programs to ONNX allows us to benefit from ONNX's existing ecosystem.

Optimization. The ONNX graph produced by the compiler faithfully reproduces the semantics of the original program but might not be optimal. Our optimizer then converts the unoptimized ONNX graph into an optimized one.

The particular optimization strategy we use is based on equality saturation [Tate et al. 2009; Willsey et al. 2021], which has been shown to be effective for optimizing tensor computations [Jia et al. 2019; Yang et al. 2021]. The technique is broadly split into two phases: saturation and extraction. During the saturation phase, rewrite rules are used to generate a set of equivalent programs, each with a different cost governed by a cost function. Then, the extraction phase extracts the cheapest program. While using equality saturation for tensor optimizations is well established, our contribution lies in specifying rewrite rules and a cost function tailored to color programming. These are described in Sec. 8.

Execution. The optimizer outputs an optimized ONNX program, which is executed using ONNX Runtime [Developers 2021]. Since CoolerSpace generates a reusable ONNX file, the compilation and optimization costs only have to be paid once per program.

5 CoolerSpace Type System

After discussing the general principles behind our type system (Sec. 5.1), we will walk through the syntax by first describing the supported types (Sec. 5.2), followed by describing the permissible operations and the typing rules that govern these operations (Sec. 5.3).

5.1 Overview

We show the abstract syntax of CoolerSpace in Tbl. 1. A program in CoolerSpace consists of a set of expressions, each of which represents a physical operation that manipulates colors and/or spectra. Critically, each value in an expression is typed, permitting static type checking and promoting physics-aware color programming.

CoolerSpace's type system is designed to capture common computations used in color programming. CoolerSpace's types capture the underlying physical qualities of lights and materials, digital encodings of color, and models of human color perception. CoolerSpace allows users to operate on lights, materials, and the colors that result from light-material interactions.

We acknowledge that our type system, and arguably any type system, is inherently opinionated, as it stipulates as set of concrete rules. CoolerSpace's type system is designed to follow the rules of fundamental physics, or wherever applicable, of standards defined by bodies such as the International Commission on Illumination (CIE). There may exist scenarios where programmers would prefer to break our rules in favor of other considerations like speed. In these scenarios, we provide users with an escape from our type system (see the usage of the `Matrix` type in Sec. 5.3).

5.2 Types in CoolerSpace

The type system is the most important component of CoolerSpace. All type-checked values in CoolerSpace are of a Shaped Type (τ, d), which is a product type of a Physical Type τ and a Dimension Type d. Physical Types represent physical properties, e.g., colors, light, and material; we will describe them in detail later. Dimension Types represent the tensor dimension of a value/object. The Dimension Types are expressed as a product of natural numbers, representing the shape of a matrix. For example, a full HD image (1920×1080 in resolution) encoded in the `sRGB` color space would have the Shaped Type $(\tau_{\text{sRGB}}, 1920 \times 1080)$.

Both Physical Types and Dimension Types are important because permissible computations in CoolerSpace should be performed on inputs that are physically meaningful and of correct dimension. For instance, adding lights with colors is physically meaningless, and adding two color objects with mismatching dimensions is mathematically meaningless.

The Physical Types supported by CoolerSpace can be largely split into three broad categories: Tristimulus Color Types, Perceptual Color Types, and Spectral Types.

Tristimulus Color Types. Tristimulus Color Types, $\tau_{\text{tristimulus}}$, represent color spaces in which any color is represented by three channels, i.e., the tristimulus values. These color spaces are defined by the choice of three primary colors and a white color. A color of these types is internally encoded as a linear combination of the three primaries. Thus, a specific color can be encoded differently across different tristimulus color spaces.

CoolerSpace currently supports four Tristimulus Color Types: `opRGB`, `sRGB`, `LMS`, and `XYZ`. Extending to other color spaces is straightforward. The `LMS` and `XYZ` spaces are linear color spaces, in the sense that a color encoded in these spaces has channel values proportional to the *power* of the corresponding light. In contrast, `sRGB` and `opRGB` color spaces are non-linear. Channel values are proportional to *perceived brightness*, as discussed in Sec. 2.

Linear and non-linear color spaces have different uses in color programming and are both important to support in CoolerSpace. Linear color spaces are usually used when colors are initially captured or produced because of its direct relationship to light power. By contrast, non-linear color spaces are usually used when encoding and storing colors; in fact, most image file formats encode colors in the `sRGB` color space by default. A classic workflow in the graphics pipeline is to render pixel colors in a linear space and store the image in a non-linear space.

Perceptual Color Types. CoolerSpace also supports a set of Perceptual Color Types, $\tau_{\text{perceptual}}$, each corresponding to a perceptual color space. Unlike tristimulus color spaces, perceptual color spaces do not represent colors as a mixture of primary colors; instead, they represent colors by modeling how humans subjectively perceive colors.

For example, the CIELAB color space (abbreviated as `LAB`) models the opponent process of the human visual system [Stockman and Brainard 2010]. `LAB` represents a color by lightness (perceived brightness), red-green opponency, and yellow-blue opponency. The `HSV` (also called HSL or HSB) color space represents a color as its lightness, hue, and saturation.

CoolerSpace provides the perceptual color spaces to support use-cases where subjective assessments of colors are involved. For instance, both the `HSV` and `LAB` color spaces are commonly used to compare colors; in fact, `LAB` is the most common color space to quantify color differences (known as the CIE Delta E metric [Sharma 2017]), a key task in color programming. The `HSV` color space is commonly used to design color pickers in digital applications.

Spectral Types. A Spectral Type, τ_{spectrum}, represents a physical property that is dependent on wavelength. For instance, the `Reflectance` type represents the reflectance of a surface/material over wavelength; similarly, the `Light` type represents the spectral power distribution of lights.

Two other important Spectral Types are the `Scattering` and `Absorption` types, which represent the spectral scattering and spectral absorption functions of a surface/material, respectively. The `Pigment` type, which represents materials (e.g., pigments like phthalo blue), is a product type between `Scattering` and `Absorption`. This is because both scattering and absorption spectra are required to accurately model the mixture of two materials [Kubelka and Munk 1931].

Internally, spectral types are represented as histograms across the visible spectrum, defined between 390 nm and 830 nm in CoolerSpace. We uniformly quantize this visible spectrum into 89 unique bands (i.e., the Spectral Type has a channel count of 89) at a 5 nm interval, but more fine-grained quantization schemes can be trivially implemented.

Matrix Type. The `Matrix` type allows programmers to specify a numerical tensor in CoolerSpace. These tensors are usually used for geometrically or arithmetically manipulating colors and spectra. For instance, a tristimulus color can be seen as a point in a 3D Euclidean space, and a color science programmer might want to project the color to a plane, e.g., when simulating color vision deficiency [Brettel et al. 1997]. Projection (and in fact any linear transformation) is mathematically a matrix multiplication, hence the need for a `Matrix` type.

5.3 Typing Rules

The typing rules allows only physically meaningful arithmetic operations. Each class of the typing rules is, thus, designed to allow expressing a particular set of physical operations. Our description below focuses on how CoolerSpace helps implement physically correct color programs.

Mixing Lights. The first class of typing rules expresses mixing two lights, which is perhaps the single most widely used operation in color science where one, for instance, mixes multiple lights in order to produce a target color.

Light mixing can be done in either the spectral or tristimulus domains. Both are expressed by the '+' operator in the syntax. When mixing two lights in the spectral space, the intention is to calculate the spectrum of the resulting light. This is expressed by the Rule 5.1, which stipulates that the result of mixing two set of light spectra is another set of light spectra.

$$\frac{\Gamma \vdash v_1 : (\tau_{\text{Light}}, d) \qquad \Gamma \vdash v_2 : (\tau_{\text{Light}}, d)}{\Gamma \vdash v_1 + v_2 : (\tau_{\text{Light}}, d)}\ \textsc{LightAdd}$$

Rule 5.1. Light addition rule.

Mixing lights using their colors is also permitted. Programmers can mix tristimulus colors additively with the intention to calculate the color of the mixture of the original lights. This is represented in Rule 5.2, which stipulates that both input colors must belong to the same tristimulus color space. Then, the resulting color of the light will be presented in the same color space. Note both rules also enforce that the dimensions of the two operands must match.

$$\frac{\Gamma \vdash v_1 : (\tau_{\text{tristimulus}}, d) \qquad \Gamma \vdash v_2 : (\tau_{\text{tristimulus}}, d)}{\Gamma \vdash v_1 + v_2 : (\tau_{\text{tristimulus}}, d)} \textsc{TristimulusAdd}$$

Rule 5.2. Tristimulus addition rule.

Mixing Perceptual Colors. Color mixing can be done in either a physically uniform or perceptually uniform manner. The rules above are intended for the physically uniform mixture of colors – the linearity of addition in the spectral power domain is preserved. However, the human visual system perceives color non-uniformly. For example, a linear increase in LMS cone responses does not correspond to a linear increase in perceived brightness. Perceptually uniform color spaces, represented by $\tau_{\text{perceptual}}$, are color spaces designed to represent the range of human-perceivable colors uniformly. The distance between any two colors in a perceptually uniform color spaces like `LAB` are representative of the perceived difference between the two colors [Sharma and Bala 2017].

Programmers may want to mix colors in a perceptually uniform color space when creating gradients or when conducting psychophysical experiments [Fairchild and Reniff 1995]. CoolerSpace allows programmers to add colors in a perceptually uniform manner, provided that the inputs to the addition operation are of a perceptual color space. This is represented in Rule 5.3. Like in Rule 5.2, the addition is only permitted if the operands are of the same specific perceptual type.

$$\frac{\Gamma \vdash v_1 : (\tau_{\text{perceptual}}, d) \qquad \Gamma \vdash v_2 : (\tau_{\text{perceptual}}, d)}{\Gamma \vdash v_1 + v_2 : (\tau_{\text{perceptual}}, d)} \textsc{PerceptualAdd}$$

Rule 5.3. Perceptual addition rule.

Light Reflection. CoolerSpace also allows expressing reflecting light off of a surface. This operation allows programmers to calculate the color of an object under a particular illuminant. Physically, such calculation must be done in the spectral space, where light SPD and material reflectance are defined (Fig. 1c). Syntactically, reflection is expressed with '×'. Rule 5.4 describes the corresponding typing rule.

$$\frac{\Gamma \vdash v_1 : (\tau_{\text{Light}}, d) \qquad \Gamma \vdash v_2 : (\tau_{\text{Reflectance}}, d)}{\Gamma \vdash v_1 \times v_2 : (\tau_{\text{Light}}, d)} \textsc{Reflect}$$

Rule 5.4. Type rule for reflection.

Mixing Materials. Color programming also involves mixing materials, e.g., pigments. For instance, painters routinely mix their primary paints to produce new colors. This process must be faithfully implemented in any digital painting software [Sochorová and Jamriška 2021]. Print and dye industries also investigate how to properly mix inks and dyes to produce the target material quality [Broadbent 2001]. Syntactically, mixing material is expressed by mix(·), which takes four parameters: two pigment objects and their corresponding concentrations.

Mixing pigments is both physically and mathematically different from mixing lights and warrants its own typing rules. Two rules in CoolerSpace govern pigment-related operations. First, we allow initializing a pigment type from an absorption and a scattering type, as enshrined by the PgmtInit rule in Rule 5.5. This reflects the fact that a `Pigment` is internally described by the absorption and scattering spectra of the material. Second, the PgmtMix rule in Rule 5.5 expresses the mixing the of two pigment objects. The rule states that the output of mixing two `Pigment` objects of matching dimensions is another `Pigment` object of the same dimension.

$$\frac{\Gamma \vdash v_1 : (\tau_{\text{Absorption}}, d) \qquad \Gamma \vdash v_2 : (\tau_{\text{Scattering}}, d)}{\Gamma \vdash \tau_{\text{Pigment}}(v_1, v_2) : (\tau_{\text{Pigment}}, d)} \textsc{PgmtInit}$$

$$\frac{\Gamma \vdash v_1, v_2 : (\tau_{\text{Pigment}}, d) \qquad \Gamma \vdash v_3, v_4 : (\tau_{\text{Matrix}}, d)}{\Gamma \vdash \mathsf{mix}(v_3, v_1, v_4, v_2) : (\tau_{\text{Pigment}}, d)} \textsc{PgmtMix}$$

Rule 5.5. `Pigment` rules.

Transforming Colors. Programmers can scale each channel of a tristimulus color through element-wise multiplication. Syntactically, these operations are expressed as $v_{rgb} \times v_{matrix}$. Rule 5.6 governs this operation.

$$\frac{\Gamma \vdash v_1 : (\tau_{\text{tristimulus}}, d) \qquad \Gamma \vdash v_2 : (\tau_{\text{Matrix}}, 3)}{\Gamma \vdash v_1 \times v_2 : (\tau_{\text{tristimulus}}, d)} \textsc{TriScale}$$

Rule 5.6. Elementwise scaling of a tristimulus color.

Type Casting. If one wants to mix, for instance, an `sRGB` color with a `XYZ` color, one must cast the `sRGB` type to the `XYZ` type (or vice versa). Syntactically, casting is expressed by $\tau(v)$, where τ is the target type and v is the object to cast.

Principles in color science dictate the set of legal castings, which is illustrated in Fig. 4. A casting between any origin and destination type is allowed if there exists a path from the former to the latter in Fig. 4. The legal castings are expressed in Rule 5.7. The path_exists(τ_1, τ_2) function type checks if there is a path from type τ_1 to type τ_2 in Fig. 4.

$$\frac{\Gamma \vdash v : (\tau_1, d) \qquad \text{path\_exists}(\tau_1, \tau_2)}{\Gamma \vdash \tau_2(v) : (\tau_2, d)} \textsc{Cast}$$

Rule 5.7. Casting rule

Our casting rules prevent mathematically ill-posed castings. For instance, casting a `Light` type to an `LMS` type is permitted, but casting an `LMS` type to a `Light` type is not. This is because converting a light spectrum to an LMS color is a dimensionality reduction (Equ. 1), so the inversion is mathematically ill-posed. There are many light spectra that correspond to the same LMS color.

The `Pigment` type casts to the `Scattering` type or the `Absorption` type, because the former is defined as a product type of the latter two; these two castings are lossy and cannot be reversed. Importantly, we allow the `Pigment` type to cast to the `Reflectance` type. This represents the physical reality that the reflectance spectrum of a material can be derived from the material's scattering and absorption spectra, which are carried in the `Pigment` type.

Using the Matrix Type. We allow casting to and from the `Matrix` type from most other types. By casting objects to the `Matrix` type, programmers can escape our strict type system and define arithmetic operations that would not normally be permissible. For example, programmers can define arbitrary addition operations using Rule 5.8. If one wishes to arithmetically add two `sRGB` colors, as seen in Prog. 2 of Sec. 3, they may do so by first casting the objects to the `Matrix` type before the addition operation.

Other Rules. CoolerSpace also defines other operations and their associated typing rules. For instance, programmers can retrieve individual channels of an object through the syntax $v.c$, where c represents the channel to be accessed. We also allow the application of transformation

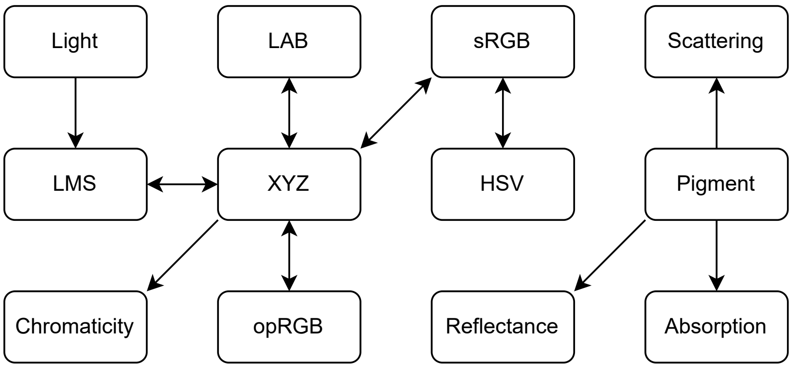


Fig. 4. The graph of all permissible castings. CoolerSpace currently only supports a small number of commonly used color spaces. This is not an inherent limitation of CoolerSpace. With more engineering effort, CoolerSpace's interface can be extended to support other color spaces.

$$\frac{\Gamma \vdash v_1 : (\tau_{\text{Matrix}}, d) \qquad \Gamma \vdash v_2 : (\tau_{\text{Matrix}}, d)}{\Gamma \vdash v_1 + v_2 : (\tau_{\text{Matrix}}, d)} \textsc{MatrixAdd}$$

Rule 5.8. Simplified matrix type addition.

Table 2. Subset of ONNX syntax.

Natural Numbers	$\mathbb{N} \in$ natural numbers
Arrays	$a \in$ floating point arrays
Dimension Types	$d ::= \mathbb{N} \vert d \times d$
Values	$u ::= a \vert \mathsf{add}(u,u) \vert \mathsf{div}(u,u) \vert \mathsf{mul}(u,u)$ $\vert \mathsf{sub}(u,u) \vert \mathsf{matmul}(u,u) \vert \mathsf{pow}(u,u)$

matrices to tristimulus colors using $\mathsf{matmul}(v_{rgb}, v_{matrix})$. The supplemental material contains a comprehensive list of CoolerSpace's typing rules.

6 CoolerSpace to ONNX Translation

Programs written in CoolerSpace are translated into ONNX. The decision to use ONNX as a compilation target is justified in Sec. 4. We introduce our translation strategy with formal translational semantics in Sec. 6.1. We then prove that our translation is sound in Sec. 6.2: any type checked value in CoolerSpace translates to a type checked value in ONNX. Finally, in Sec. 6.3, we show that CoolerSpace is type sound in addition to being translationally sound.

6.1 Translational Semantics

The translation process is guided by our translational semantics. The translational semantics must be understood in conjunction with the abstract syntax of ONNX, our target language. To our best knowledge, there is no formal syntax of ONNX. Completely formalizing ONNX is out of our scope. We do, however, formalize a subset of ONNX pertaining to CoolerSpace, which is shown in Tbl. 2.

Unlike CoolerSpace, ONNX does not assign Physical Types to values, because ONNX is designed to express tensor algebra. Therefore, values in ONNX are assigned only Dimension Types. add, div, mul, sub, and pow perform element-wise operations between two tensors; matmul is matrix multiplication. All expressions support dimension broadcasting, which is discussed in Sec. 6.2

Table 3. Subset of CoolerSpace to ONNX translational semantics. M_1, M_2, and M_3 represent the LMS Cone Fundamentals, XYZ to LMS transformation matrix, and XYZ to RGB transformation matrix, respectively. They are constant matrices that can be found in standard color science texts [Wyszecki and Stiles 2000], and are omitted here.

$$
\begin{array}{ll}
[\![\mathbb{N}]\!] \triangleq \mathbb{N} & \textbf{T-Nat} \\
[\![v_1 + v_2]\!] \triangleq \Phi(+, \tau_1, \tau_2)(v_1, v_2), \quad v_1 : (\tau_1, d_1), v_2 : (\tau_2, d_2) & \textbf{T-Add} \\
[\![a]\!] \triangleq a & \textbf{T-Array} \\
[\![v_1 \times v_2]\!] \triangleq \Phi(\times, \tau_1, \tau_2)(v_1, v_2), \quad v_1 : (\tau_1, d_1), v_2 : (\tau_2, d_2) & \textbf{T-Mul} \\
[\![\tau(a)]\!] \triangleq a & \textbf{T-Init} \\
[\![\tau_d(v_o)]\!] \triangleq \Psi(\tau_o, \tau_d)(v_o), \quad v_o : (\tau_o, d_o) & \textbf{T-Cast} \\
[\![d]\!] \triangleq d & \textbf{T-Dim} \\
[\![(\tau, d)]\!] \triangleq d \times \mathsf{channel\_count}(\tau) & \textbf{T-Type} \\
\Phi(+, \tau_{\text{XYZ}}, \tau_{\text{XYZ}})(v_1, v_2) \triangleq \mathsf{add}\left([\![v_1]\!], [\![v_2]\!]\right) & \\
\Phi(+, \tau_{\text{LMS}}, \tau_{\text{LMS}})(v_1, v_2) \triangleq \mathsf{add}\left([\![v_1]\!], [\![v_2]\!]\right) & \\
\Phi(+, \tau_{\text{sRGB}}, \tau_{\text{sRGB}})(v_1, v_2) \triangleq \mathsf{mul}(\mathsf{pow}(\mathsf{add}(\mathsf{pow}(\mathsf{div}([\![v_1]\!], \left[255\right]), \left[2.2\right]), & \\
\qquad\qquad \mathsf{div}(\mathsf{pow}([\![v_2]\!], \left[255\right]) \left[2.2\right])), \left[0.455\right]), \left[255\right]) & \\
\Phi(\times, \tau_{\text{Light}}, \tau_{\text{Reflectance}})(v_1, v_2) \triangleq \mathsf{mul}([\![v_1]\!], [\![v_2]\!]) & \\
\Psi(\tau_{\text{Light}}, \tau_{\text{LMS}})(v) \triangleq \mathsf{matmul}\left([\![v]\!], \mathrm{M}_1\right) & \\
\Psi(\tau_{\text{LMS}}, \tau_{\text{XYZ}})(v) \triangleq \mathsf{matmul}\left([\![v]\!], \mathrm{M}_2\right) & \\
\Psi(\tau_{\text{XYZ}}, \tau_{\text{sRGB}})(v) \triangleq \mathsf{pow}\left(\mathsf{mul}\left(\mathsf{matmul}\left([\![v]\!], \mathrm{M}_3\right), \left[255\right]\right), \left[2.2\right]\right) &
\end{array}
$$

Given the abstract syntax of ONNX, Tbl. 3 shows the formal translational semantics for converting a subset of the CoolerSpace abstract syntax to ONNX's abstract syntax. Due to space constraints, we show only a subset of the translational semantics, focusing on those that highlight important properties of this translation.

Immediately clear is that the Physical Type information in CoolerSpace is lost during translation. As shown by the **T-Type** rule, the Physical Type τ in CoolerSpace gets reduced to only its dimension information in ONNX. For instance, after translating to ONNX, objects of the same dimension in `LMS` and `sRGB` space are no longer differentiated, as both `LMS` and `sRGB` have three channels.

The translations of the actual expressions are encoded in the Φ and Ψ lookup tables, which represent, respectively, the ONNX implementation of each operation and casting in CoolerSpace. For instance, $\Phi(+, \tau_{\text{sRGB}}, \tau_{\text{sRGB}})(v_1, v_2)$ encodes the ONNX implementation of `sRGB+sRGB`, and $\Psi(\tau_{\text{Light}}, \tau_{\text{LMS}})(v)$ encodes how we cast a `Light` spectrum to a color in the `LMS` color space.

Two interesting properties of Φ and Ψ are worth discussing. First, arithmetic operations type checked by the same type rule do not necessarily translate to the same set of ONNX operations in Φ. For instance, adding two `XYZ` objects and adding two `sRGB` objects are both expressed with the '+' syntax and are type-checked by Rule 5.2. However, `sRGB` addition is translated to a much more complicated sequence of ONNX operations, as seen in the corresponding entry in Φ (Tbl. 3). The reason is that the `sRGB` color space has a defined gamma curve, which must be removed before addition and reapplied after addition.

Second, casting operations between types that are not adjacent in Fig. 4 are intentionally *not* defined in Ψ. Casting operations between non-adjacent types are expanded into a series of individual casting operations representing the shortest path between the origin type and the destination type. For example, `LMS` and `sRGB` are not connected by an edge. The casting between the two, $\Psi(\tau_{\text{LMS}}, \tau_{\text{sRGB}})(v)$, is converted to $\Psi(\tau_{\text{XYZ}}, \tau_{\text{sRGB}})(\Psi(\tau_{\text{LMS}}, \tau_{\text{XYZ}})(v))$. This cascaded translation is sub-optimal: it involves multiplying two constant matrices. We rely upon equality saturation to optimize these inefficient translations (Sec. 8) while maintaining a small Φ. Note that there is no

$$\frac{}{\text{broadcastable}(d, d)}\ \textsc{TrivialBroadcast}$$

$$\frac{}{\text{broadcastable}(1, d)}\ \textsc{ScalarBroadcast}$$

$$\frac{d_1 = d_2 \times d_3}{\text{broadcastable}(d_3, d_1)}\ \textsc{SubsetBroadcast}$$

$$\frac{u_1 : d_1 \qquad u_2 : d_2 \qquad \text{broadcastable}(d_1, d_2)}{\text{add}(u_1, u_2) : d_2}\ \textsc{OnnxAddR}$$

Fig. 5. Subset of ONNX typing rules.

ambiguity in non-adjacent casting: there exists at most one path between any pair of types, because the casting graph in Fig. 4 is a directed forest.

6.2 Translational Soundness

We now prove translational soundness: any well-typed value in CoolerSpace remains well-typed after being translated to ONNX. Translational soundness indicates that our translation preserves typeability: CoolerSpace's type safety is as strong as that of ONNX. This strategy is inspired by Gator [Geisler et al. 2020]. However, proving ONNX's type safety is beyond the scope of this paper.

ONNX Typing Rules. The translational soundness proof depends on the formal typing rules of ONNX. Fig. 5 shows a set of key typing rules for ONNX. The rules ensure that the tensor dimensions of operands are valid. This is complicated because ONNX, like many tensor libraries (e.g., NumPy), permits flexible dimension *broadcasting*. Arithmetic operations are valid even if the Dimension Types of two input tensors do not match, so long as the dimension of the smaller tensor can be "broadcast", or expanded, to be compatible with that of a larger tensor.

Many CoolerSpace expressions rely on broadcasting in ONNX to implement. For instance, removing and applying gamma from a tensor of `sRGB` objects requires broadcasting a scalar (gamma) to an entire tensor of `sRGB` colors. Therefore, the pow function in ONNX, which is used to implement gamma, must broadcast a scalar (2.2) to every element in a tensor in an element-wise manner. See the $\Phi(+, \tau_{\text{sRGB}}, \tau_{\text{sRGB}})(v_1, v_2)$ entry in the translational semantics (Tbl. 3) for an example.

To our best knowledge, broadcasting has not been formally specified in ONNX. We formalize a subset of broadcasting rules that are relevant to CoolerSpace. These rules are defined in the rules TrivialBroadcast, ScalarBroadcast, and SubsetBroadcast. Here, broadcastable(d_1, d_2) indicates that a dimension d_1 can be broadcast to a dimension d_2. TrivialBroadcast is the usual case where both inputs have the same dimensions. ScalarBroadcast allows a scalar input to be broadcast to a tensor in an element-wise manner. SubsetBroadcast allows a smaller tensor dimension d_1 to be broadcast to d_2 if d_1 is a *right-aligned* subset of d_2 (i.e. 1080×3 is broadcastable to $1920 \times 1080 \times 3$, but 1920×1080 is not).

The rest of the rules codify legal tensor operations using the broadcast rules. For instance, OnnxAddR specifies that adding two tensors is allowed so long as the first tensor dimension can be broadcast to that of the second. Other rules governing sub, mul, div, pow, and matmul are omitted here, but can be found in Section 2.2 of the Supplementary Material.

Theorem. Formally, translational soundness states:

$$\frac{[\![v : (\tau, d)]\!]}{[\![v]\!] : [\![(\tau, d)]\!]}$$

$$\dfrac{\dfrac{\dfrac{\llbracket v_1 + v_2 : (\tau_{\mathrm{XYZ}}, d)\rrbracket}{\llbracket v_1, v_2 : (\tau_{\mathrm{XYZ}}, d)\rrbracket}\,\textsc{Conv-TriAdd} \qquad \llbracket v_1, v_2 \subset v_1 + v_2\rrbracket}{\llbracket v_1, v_2\rrbracket : \llbracket(\tau_{\mathrm{XYZ}}, d)\rrbracket}\,\textsc{IndHyp} \qquad \dfrac{}{\mathsf{broadcastable}(\llbracket(\tau_{\mathrm{XYZ}}, d)\rrbracket, \llbracket(\tau_{\mathrm{XYZ}}, d)\rrbracket)}\,\textsc{TrivBroadcast}}{\mathsf{add}(\llbracket v_1\rrbracket, \llbracket v_2\rrbracket) : \llbracket(\tau_{\mathrm{XYZ}}, d)\rrbracket}\,\textsc{OnnxAddR}$$

$$\dfrac{\mathsf{add}(\llbracket v_1\rrbracket, \llbracket v_2\rrbracket) : \llbracket(\tau_{\mathrm{XYZ}}, d)\rrbracket \qquad \dfrac{\llbracket v_1, v_2 : (\tau_{\mathrm{XYZ}}, d)\rrbracket}{\llbracket v_1 + v_2\rrbracket \triangleq \mathsf{add}(\llbracket v_1\rrbracket, \llbracket v_2\rrbracket)}\,\textsc{T-Add}}{\llbracket v_1 + v_2\rrbracket : \llbracket(\tau_{\mathrm{XYZ}}, d)\rrbracket}\,\textsc{Subst}$$

Fig. 6. Proof of translational soundness for `XYZ` + `XYZ` under ColorAdd

The Inductive Hypothesis. We use structural induction to prove translational soundness. The inductive hypothesis is given below. The inductive hypothesis states that all sub-values of a well-typed value in CoolerSpace translate to well-typed values in ONNX. We use $v_i \subset v$ to mean that v_i is an immediate sub-value of v.

$$\dfrac{\llbracket v_i \subset v\rrbracket \qquad \llbracket v_i : (\tau_i, d_i)\rrbracket \qquad \llbracket v : (\tau, d)\rrbracket}{\llbracket v_i\rrbracket : \llbracket(\tau_i, d_i)\rrbracket}\,\textsc{IndHyp}$$

Proof. We prove translational soundness by covering all cases of well-typed values. We show a representative case, where the value v is of the form $v_1 + v_2$ and belongs to the `XYZ` type. Other cases are similar in form. See Section 2 of the Supplementary Material for the comprehensive proof.

Fig. 6 shows the proof tree. If the value $v_1 + v_2$ has the type `XYZ`, from the typing rules (Rule 5.2 TristimulusAdd) we know that v_1 and v_2 are both of type `XYZ`; this is represented by Conv-TriAdd in the proof tree. By the inductive hypothesis, v_1 and v_2 are typed as $\llbracket(\tau_{\mathrm{XYZ}}, d)\rrbracket$ in ONNX after translation. From OnnxAddR, we know that $\mathsf{add}(\llbracket v_1\rrbracket, \llbracket v_2\rrbracket)$ must also be typed as $\llbracket(\tau_{\mathrm{XYZ}}, d)\rrbracket$.

From the translational semantics T-Add, we know that $\llbracket v_1 + v_2\rrbracket$ translates to $\mathsf{add}(\llbracket v_1\rrbracket, \llbracket v_2\rrbracket)$. Given that $\mathsf{add}(\llbracket v_1\rrbracket, \llbracket v_2\rrbracket)$ is typed as $\llbracket(\tau_{\mathrm{XYZ}}, d)\rrbracket$ in ONNX, using substitution we can conclude that $\llbracket v_1 + v_2\rrbracket$ is typed as $\llbracket(\tau_{\mathrm{XYZ}}, d)\rrbracket$ in ONNX. Therefore, the translational soundness theorem is satisfied for ColorAdd when $v_1 + v_2$ is of type `XYZ`.

6.3 Type Soundness

Our proof of translational soundness demonstrates that any well-typed value in CoolerSpace is guaranteed to translate to a well-typed ONNX value. However, translational soundness does not imply type safety. For example, translational soundness cannot ensure that CoolerSpace types are preserved after evaluation. Consider the following hypothetical and faulty reflection type rule for reflection operations:

$$\dfrac{\Gamma \vdash v_1 : (\tau_{\mathrm{Light}}, d) \qquad \Gamma \vdash v_2 : (\tau_{\mathrm{Reflectance}}, d)}{\Gamma \vdash v_1 \times v_2 : (\tau_{\mathrm{Reflectance}}, d)}\,\textsc{WrongReflect}$$

Rule 6.1. A faulty type rule for reflection operations. The correct type rule is Rule 5.4 (Reflect).

$$\tau_{\mathrm{Light}}(a_1) \times \tau_{\mathrm{Reflectance}}(a_2) \longrightarrow \tau_{\mathrm{Light}}((a_1) \times (a_2)) \quad \textsc{E-Reflect}$$

Rule 6.2. The evaluation rule for reflection operations. The corresponding type rule is Rule 5.4 (Reflect).

In Rule 6.1, reflection operations are mistakenly given the type `Reflectance`. The evaluation rule E-Reflect (Rule 6.2) evaluates the physical type of $\tau_{\mathrm{Light}}(a_1) \times \tau_{\mathrm{Reflectance}}(a_2)$ to `Light`. Thus, the type of the expression $\tau_{\text{Light}}(a_1) \times \tau_{\text{Reflectance}}(a_2)$ changes after evaluation. Preservation is violated. Importantly, the translational soundness proof would not be able to catch this faulty type rule. $v_1 \times v_2$ is translated to $\mathsf{mul}(\llbracket v_1 \rrbracket, \llbracket v_2 \rrbracket)$, according to the translational semantics in Tbl. 3. $\mathsf{mul}(\llbracket v_1 \rrbracket, \llbracket v_2 \rrbracket)$ would still type check in ONNX, as the dimension types of v_1 and v_2 match.

We prove type soundness of CoolerSpace in addition to translational soundness. Our approach is a straightforward proof of progress and preservation for each rule. We have defined a set of evaluation rules to aid the proof. The entire type soundness proof can be found in Sections 3 and 4 of the Supplementary Material.

7 Type System Design Decisions

There were several considerations that informed our design decisions for CoolerSpace. We detail them in this section.

Handling of Non-linear Tristimulus Color Spaces. The `sRGB` and `opRGB` color spaces are non-linear tristimulus color spaces (**Gamma** paragraph of Sec. 2). In CoolerSpace, interpolations between two `sRGB` or two `opRGB` objects are done in linear space (see the $\Phi(+, \tau_{\text{sRGB}}, \tau_{\text{sRGB}})(v_1, v_2)$ entry in Tbl. 3). We convert `sRGB` and `opRGB` values to linear space prior to interpolation as interpolation in linear spaces is uniform with respect to physical luminance.

An alternative is to perform non-linear tristimulus interpolation in a perceptual space instead. This is because non-linear tristimulus color spaces are roughly uniform in *perceived brightness*; programmers may expect interpolation between non-linear tristimulus colors to be perceptually uniform. However, non-linear tristimulus color spaces are not uniform in chromaticity. Principled perceptual interpolation must be done in a perceptually uniform space like `LAB`.

CoolerSpace could convert `sRGB` values to `LAB` values prior to interpolation. However, there are many competing models of uniform color perception like CIELUV [Sharma and Bala 2017], CIELAB [Sharma and Bala 2017], and CAM16 [Li et al. 2017]. Interpolation operations done in different perceptually uniform color spaces will yield different results. We do not want to make any assumptions on what model of perceptual uniformity the programmer prefers. We therefore reject this design.

Tristimulus Addition Restrictions. Rule 5.2 (TristimulusAdd) stipulates that the operands of a tristimulus addition must be of the same tristimulus type. For example, addition between two `LMS` objects is valid, but addition between an `LMS` and an `XYZ` object is invalid. However, there is nothing wrong, in principle, with interpolating two tristimulus values of different spaces. One can imagine a set of translational semantics that, given an addition between an `LMS` or an `XYZ` object, first converts the `LMS` value to `XYZ` space or converts the `XYZ` value to `LMS` space. However, the output type of an operation between operands of differing spaces would be unspecified.

To address that issue we could ideally use bidirectional type checking [Chlipala et al. 2005] to derive the expected color space of the tristimulus addition operation. For example, in Prog. 5, the output of the operation between the `XYZ` and `LMS` is assigned to the `mixed` variable. The `mixed` variable is annotated with the Python type hint `: cs.XYZ`, indicating that the programmer expects the `mixed` variable to be of the `XYZ` type. Using the `mixed` variable's type hint, CoolerSpace could infer, through bidirectional type checking, that the programmer expects `cs.XYZ(...) + cs.LMS(...)` to output an object of type `XYZ`.

```
mixed: cs.XYZ = cs.LMS(...) + cs.XYZ(...)
```

Program 5. Bidirectional typing example. The cs.LMS(...) + cs.XYZ(...) operation is inferred to have the output type of cs.XYZ, as the `mixed` object that the operation output is assigned to has the `XYZ` type.

Unfortunately, implementing bidirectional type checking in a Python library is not feasible. Python is a dynamically typed interpreter language. It is impossible for CoolerSpace to obtain the type hint of the `mixed` variable prior to variable assignment. Therefore, the output type of `cs.LMS(...) + cs.XYZ(...)` cannot be known when evaluating the addition operation.

It's important to note that while bidirectional typing would alleviate the stringent type restrictions for tristimulus space addition operations, perceptual addition (Rule 5.3) would remain the same. This is because the outputs of arithmetic operations performed in different perceptual color spaces are not equivalent. The user must still specify the perceptual color space used for interpolation.

Casting. Readers familiar with physical unit types (measurement types) literature [Allen et al. 2004; Dreiheller et al. 1986; Karr and Loveman 1978] will notice some parallels between our type system and those of previous works in the field of physical unit types. However, not all assumptions of physical unit type systems are applicable to CoolerSpace. This difference informs the design of our casting rule (Rule 5.7).

In physical unit types literature, units can be easily converted into other units of measurement representing the same dimension[4], but not to units of different dimensions. Dimensions indicate the physical quantity being measured, and units indicate the standard of measurement [Allen et al. 2004; Varkor 2018]. Examples of dimensions include time, length, and mass. Corresponding examples of units include seconds, meters, and grams. A programmer can convert a Fahrenheit measurement to a Celsius measurement using the syntax of [Allen et al. 2004] in Prog. 6. The conversion from Fahrenheit to Celsius is possible as both units share the same dimension type — temperature. However, conversion from Fahrenheit to, say, meters is not permitted: the dimension types are mismatched.

```
fahrenheitValue.inUnit<CelsiusDegrees>()
```

Program 6. Conversion from Fahrenheit to Celsius in [Allen et al. 2004].

There are seven dimensions represented in CoolerSpace: reflectance spectra, absorption spectra, scattering spectra, pigments, light spectra, color, and chromaticity. Unlike measurement types, units in CoolerSpace can be converted to units of other dimensions. For example, `Light` is a unit of the light spectra dimension, and `XYZ` is a unit of the color dimension. `Light` values can be coerced into `XYZ` values. However, the reverse is not true. `XYZ` values cannot be coerced into `Light` values. This is because the operation is under-determined; there exist multiple light spectra that correspond to a single color. A similar relationship exists between units of the color dimension and the `Chromaticity` unit of the chromaticity dimension.

Since casting is not always bidirectional in CoolerSpace, we designed the `path_exists` function in the Cast rule (Rule 5.7) to enable the type checker to automatically determine if a casting is possible from one physical type to another. Additionally, the edges of the graph in Fig. 4 represent implemented casting algorithms. Therefore, if the type checker confirms that an object is cast-able into another type, there must exist a corresponding series of castings that convert the object into the desired type. The `path_exists` rule and the casting graph are also designed to facilitate the addition of new types into CoolerSpace. New types can be inserted as nodes into Fig. 4, and edges can be defined that correspond to implemented casting algorithms. No modifications to the casting type rule needs to be made.

Tristimulus Pigment Mixing. The `mix` function is utilized to simulate the mixture of pigments, as specified in Rule 5.5 (PgmtMix). `mix` is only applicable to `Pigment` objects. We had originally planned for the `mix` operator to simulate pigment mixing for colors encoded in tristimulus color spaces as well. However, this problem is ill-posed — there exists an infinite number of pigment mixtures and ambient lighting conditions that are able to generate any given color. There are

[4]The dimension terminology here is not to be confused with dimension types in CoolerSpace. Dimensions in CoolerSpace refer to matrix dimensions.

algorithms of pigment mixing that operate on tristimulus colors like MixBox [Sochorová and Jamriška 2021]. These algorithms make several assumptions about the constraints of pigment mixture and the ambient lighting. The outputs are not principled, even though they approximate artists' expectations. We cannot use these algorithms in a physically rigorous programming language like CoolerSpace.

8 Optimizing CoolerSpace Programs

The translated ONNX program is optimized using equality saturation [Tate et al. 2009; Willsey et al. 2021]. The technique consists of a saturation phase, where a set of equivalent programs are enumerated using *rewrite rules*, each of which replaces an expression with a semantically equivalent one; the two expressions might have different run-time costs. Then, in the extraction phase, the program with the cheapest cost is chosen.

The rationale of using such an optimization strategy is discussed in Sec. 4. While equality saturation as an optimization technique is established, this section focuses on describing the specific design decisions we made in applying the technique to optimize CoolerSpace programs.

Rewrite Rules. We design a small set of tensor algebra rewrite rules. The rules are designed for the ONNX operations used in the translational semantics (Tbl. 3). A list of rewrite rules can be found in the supplemental material.

Part of the rules are adapted from TASO [Jia et al. 2019], which investigates rewrite rules for tensor algebra containing up to four operators. Those rules are not only overkill for our purposes (because CoolerSpace uses a subset of tensor algebra), but also do not support operations that are unique to color programming. Specifically, we include rewrite rules for the tensor exponentiation operator, which is important for implementing non-linear color space transformations.

Cost Function. In the extraction phase we need to compare the different programs yielded by the saturation phase. We implement a cost function that estimates the run-time cost of a given program; we do so empirically by calculating the total number of operations the program performs. We will show in Sec. 9 that even with these coarse-grain estimates we can still get statistically significant speedups. Better estimates will further improve performance.

Our cost function also enables constant propagation, which color programs commonly benefit from. Although the popular ONNX Runtime library [Developers 2021] performs constant propagation, it does so only when an *entire* sub-expression consists of only computations on constants. This is not always the form un-optimized ONNX programs are in.

We adjust our cost function to accommodate ONNX Runtime's constant propagation requirements. During equality saturation, we flag each tensor in the input program to indicate whether it is a constant. For instance, M_1, M_2, and M_3 in Tbl. 3 are constant matrices, and will be flagged as such. During extraction, our cost function assigns a cost of 0 to any sub-expression that can be pre-computed using only constant values. The cheapest programs will be those that have sub-expressions consisting of only computations on constants. This matches ONNX Runtime's requirements for constant propagation, and enables more aggressive constant propagation.

Implementation. We implement our optimizer using egg [Willsey et al. 2021], an e-graph-based equality saturation tool. We extend egg's interface to implement the cost function and rewrite rules, which are heavily modified from TENSAT's [Yang et al. 2021] usage of egg. We also implement converters between egg's LISP-like string input and ONNX programs.

9 Experimental Setup

Benchmarking Programs. There is no standard benchmark for evaluating CoolerSpace, so we design six programs that are commonly seen in color programming to assess the overhead and optimization capabilities of CoolerSpace. These programs are also tailored to exercise the entirety of CoolerSpace's type system. We briefly describe how they exercise different syntactic features, typing rules, translational semantics, and optimization cases in CoolerSpace. The supplemental material contains the source code of all the programs.

Color Space Conversion (SpaceConv) is a program that converts an input image in `sRGB` space to an image in `opRGB` space. CoolerSpace abstracts away the complexity of applying and removing gamma from `sRGB` and `opRGB`. `sRGB` and `opRGB` are non-adjacent in Fig. 4; therefore, this casting requires an intermediate step in ONNX that can be eliminated by our optimizer.

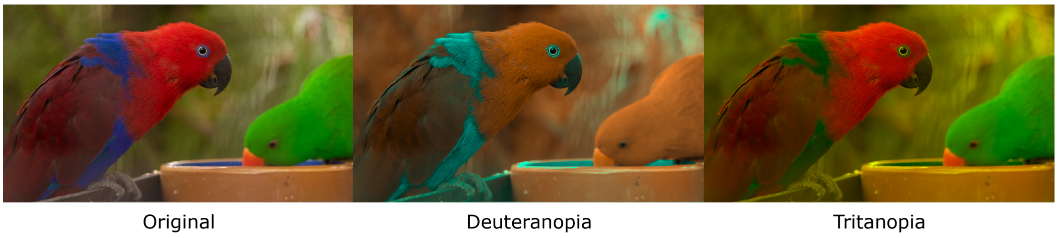


Fig. 7. Color blindness simulation. Original image courtesy of Simon Amarasingham [Amarasingham 2019].

Color Blindness Simulation (ColorBlindness) simulates dichromatic color vision. Example outputs of the program can be found in Fig. 7. While most images are originally encoded in the `sRGB` space, principled color blindness simulation must be done in the `LMS` space. CoolerSpace automatically handles the implementation logic of casting from `sRGB` to `LMS` and back. In the LMS space, the image is projected using a transformation matrix corresponding to a particular color blindness type [Viénot et al. 1999]. The program also demonstrates CoolerSpace's ability to treat colors as geometric objects and to cast them with a transformation matrix of the `Matrix` type.

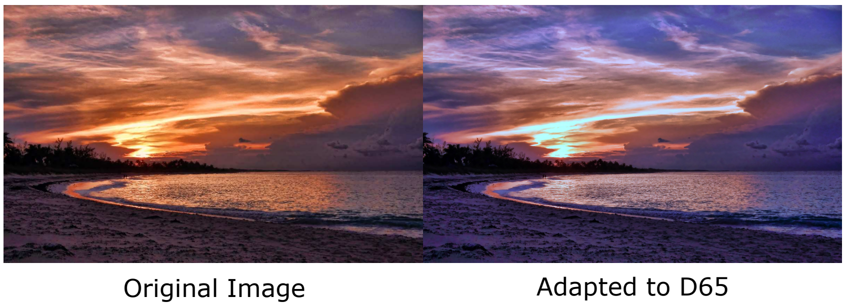


Fig. 8. Chromatic adaptation simulation. Original image courtesy of Trish Hartman [Hartmann 2012].

Chromatic Adaptation (Adaptation) is a program that simulates how the visual system adapts to the illuminant of a scene and preserves constant color perception across different illuminants [Stockman and Brainard 2010]. Chromatic adaptation is the basis of white balancing in the camera image processing pipeline [Rowlands 2020].

Our implementation takes as input two `Light` spectra representing the original and target illuminants, as well as an input image in `sRGB` space. It then applies the classic von Kries transformation matrix [Rowlands 2020] in `LMS` space to calculate the adapted image. This program exercises the casting between the `Light` type and the Tristimulus Color Types. The output of the program is shown in Fig. 8, where the original image, captured under the CIE Standard Illuminant D35 (estimated), is adapted to the CIE Standard Illuminant D65 (typical daylight).

Color Interpolation (Interpolation) is a program that linearly interpolates the colors of two different `sRGB` images. As mentioned in Sec. 3, programmers often attempt to interpolate colors in the `sRGB` color space, which is neither perceptually nor physically linear. This program demonstrates that CoolerSpace performs arithmetic operations on non-linear and non-perceptual color spaces in linear space for physical accuracy.

Pigment Mixing (Mixing) uses the `Pigment` type to simulate mixing two sets of pigments under typical daylight [Henderson and Hodgkiss 1963]. The mixing algorithm follows the Kubelka-Munk model [Kubelka 1948; Kubelka and Munk 1931] introduced in Sec. 2. Pigment mixing is commonly simulated in digital painting apps [Sochorová and Jamriška 2021]. Pigment mixing is a complex phenomenon to accurately model. CoolerSpace abstracts away the complexity of the K-M model and allows the programmer to simulate pigment mixing through the `mix`$(\cdot)$ function and the `Pigment` type. The `mix`$(\cdot)$ function and the `Pigment` type ensure that the pigment mixing simulation is done in the spectral space, with the correct spectral types as inputs.

LAB to HSV Conversion (LAB2HSV) converts an image in `LAB` space to an image in `HSV` space. `LAB` and `HSV` are complex, perceptual color spaces. Translating images to and from these spaces involve multiple expensive and non-linear operations [Lindbloom 2017]. This program demonstrates CoolerSpace's ability to handle complicated programs.

Experimental Environment. The programs are compiled and run on two machines. Machine 1 has two Nvidia GeForce RTX 2080 GPU (8 GB VRAM each), an Intel Xeon Silver 4114 CPU, and 64 GB DRAM. Machine 2 is equipped with two Nvidia GeForce RTX 4090 GPUs (24 GB VRAM each), an AMD Ryzen 9 7900X3D 12-Core Processor CPU, and 128 GB of DRAM. While both machines have two GPUs, only one is utilized during testing.

Python 3.11, ONNX Runtime 1.16, CUDA 11.8, and egg 0.9.4 are used during execution. The compiler and optimizer are run on a single core of the CPU. The optimized ONNX programs are run on either the CPU or the GPU, depending on the exact comparison being made.

To get statistically meaningful results, we compile, optimize, and run each CoolerSpace program 100 times. Both the unoptimized and optimized ONNX files are executed. The one-tailed t-test [Lakens 2017] is used to test the statistical significance of our speed-ups.

Comparison Against Existing Solutions. We also benchmark CoolerSpace against existing Python solutions. CoolerSpace's performance across the six benchmark programs is compared to equivalent programs written with the Colour-Science library [Developers 2015] and Numba [Lam et al. 2015]. Colour-Science 0.4.4, Numba 0.59, and NumPy 1.25.2 are used.

Colour-Science is chosen as a benchmarking target as it is a commonly used Python library in the color science community. However, it naturally comes with the Python interpreter overhead. To construct a stronger baseline, we also choose to benchmark CoolerSpace against Numba, a compiler capable of translating Python and Numpy code into machine code [Lam et al. 2015]. Numba lacks the overhead of the Python interpreter.

10 Results

We perform an empirical analysis on two CoolerSpace programs to demonstrate how CoolerSpace's type system enforces correctness (Sec. 10.1). CoolerSpace is capable of type checking, compiling, and optimizing a program with a minimal amount of overhead (Sec. 10.2). Even unoptimized CoolerSpace programs are faster than existing python solutions (Sec. 10.3), and the optimizations bring up to a further 1.4 $\times$ speed-up (Sec. 10.4).

10.1 Case Study on the Type System

We provide an empirical study of two CoolerSpace programs. We show CoolerSpace statically prevents programmers from specifying physically or perceptually unprincipled operations; CoolerSpace is also able to use type information to accurately translate user code.

ColorBlindness. Prog. 8 shows the source code for the ColorBlindness program. Prog. 7 is the source code for the corresponding program written in Colour-Science and NumPy. In the Colour-Science program, the `colorblind` function takes as input an image and a colorblind matrix of the NumPy array type. The user makes the assumption that the input image is of the `sRGB` type, and manually invokes specific operations to translate the image to the XYZ space (line 3), then to the LMS space (line 5). The operations on lines 3 and 5 take raw NumPy arrays as input. No information on the color space of the input image is recorded. As a result, Colour-Science and NumPy are unable to validate the encoding of their input.

```
1  def colorblind(image: np.ndarray, colorblind_matrix: np.ndarray):
2      # Convert image from sRGB to XYZ
3      xyz_image = colour.sRGB_to_XYZ(image)
4      # Convert image from XYZ to LMS
5      lms_image = xyz_image @ xyz_to_lms
6      # Apply single-plane color blindness transformation
7      lms_image_modulated = lms_image @ colorblind_matrix
8      # Convert image back to sRGB and return
9      xyz_image_modulated = lms_image_modulated @ lms_to_xyz
10     return colour.XYZ_to_sRGB(xyz_image_modulated)
```

Program 7. Colour-Science ColorBlindness code.

On lines 2 and 3 of the CoolerSpace program (Prog. 8), the programmer specifies the color space and dimensions of the input data (`sRGB`); on line 5, the programmer also specifies an `sRGB` to `LMS` casting, which CoolerSpace types check to confirm that the `image` is indeed in the `sRGB` space. This transformation is then implemented correctly behind the scenes.

```
1  # Inputs
2  image = cs.create_input("image", [1080, 1920], cs.sRGB)
3  colorblind_matrix = cs.create_input("colorblind_matrix", [3, 3], cs.Matrix)
4  # Convert image to LMS
5  image_lms = cs.LMS(image)
6  # Apply single-plane color blindness transformation
7  colorblind_image_lms = cs.matmul(image_lms, colorblind_matrix)
8  # Convert back
9  colorblind_image = cs.sRGB(colorblind_image_lms)
```

Program 8. CoolerSpace ColorBlindness code.

In line 7 of Prog. 7 and Prog. 8, the programmers apply the single-plane color blindness transformation matrix to the `LMS` image. In the Colour-Science implementation, the matrix multiplication has no guarantee that the user-input `colorblind_matrix` is 3×3. The NumPy program may crash during execution if an incorrect array is provided. Since the dimension of `colorblind_matrix` is specified in line 3 of Prog. 8, the CoolerSpace program is guaranteed to run successfully.

Interpolation. Prog. 9 is the CoolerSpace implementation of the `sRGB` interpolation. It is similar in function to the `sRGB` light addition program (Prog. 3) from Sec. 3. The operations in line 5 of Prog. 9 are interpreted as physical operations, as `sRGB` is not a perceptual color space. As a result, CoolerSpace performs the addition and multiplication operations in a linear, gamma-removed space behind the scenes.

```python
# Inputs
image1 = cs.create_input("image1", [1080, 1920], cs.sRGB)
image2 = cs.create_input("image2", [1080, 1920], cs.sRGB)
# Interpolate between the two images 50/50
mixed = image1 * 0.5 + image2 * 0.5
```

Program 9. CoolerSpace interpolation code.

To achieve an equivalent program in NumPy, as seen in Prog. 10, considerably more code is required. The additional code is error-prone. The programmer needs to manually specify the gamma removal and application procedures in lines 3, 4, and 9. Gamma values vary by color space. In NumPy, there is no guarantee that the input images are represented in `sRGB` space. There is not even a guarantee that the two input images are of the same encoding. The end result would be an interpolated image that is physically and perceptually inaccurate.

```python
1 def interpolate(image1: np.ndarray, image2: np.ndarray):
2     # Remove gamma of sRGB color space
3     image1_linear = (image1 / 255) ** 2.2
4     image2_linear = (image2 / 255) ** 2.2
5     # Interpolate 50/50
6     image_avg = image1_linear * 0.5 + image2_linear * 0.5
7     # Re-apply gamma
8     return image_avg ** (1 / 2.2) * 255
```

Program 10. NumPy interpolation code.

10.2 Compilation and Optimization Time

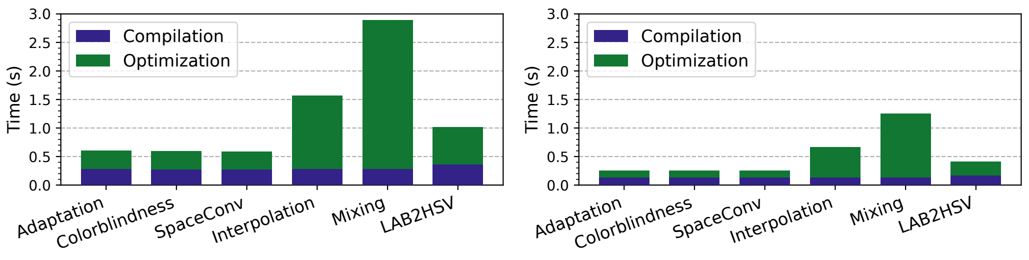
Compilation and optimization are both one-time costs. Still, we show that these one-times costs are minimal. Fig. 9 compares the average compilation and optimization times for each program, which are all less than 3 seconds with low variance one machine 1. The standard deviations of the two processes are below 0.014 ms and 0.077 ms, respectively. On machine 2, compilation and optimization are faster. The average compilation time is less than 1.5 seconds with low variance. The standard deviations of compilation and optimization are 0.001 ms and 0.23 ms respectively.

The LAB2HSV program has a notably higher compilation time in comparison to the other five programs on both machines. This is because the compiled ONNX program has a significantly higher ONNX operation count: 67, as opposed to about 15 in others.

The optimization time is generally longer than compilation time, but still below 3 seconds even for the worst test case scenario. Mixing and Interpolation are more expensive to optimize as they have a higher number of e-nodes and e-classes in their saturated e-graphs. Note that equality saturation is a worst-case exponential time algorithm, and a timeout is usually used to limit the optimization time. In our experiments, we place no such limits.

10.3 Comparison with Existing Libraries

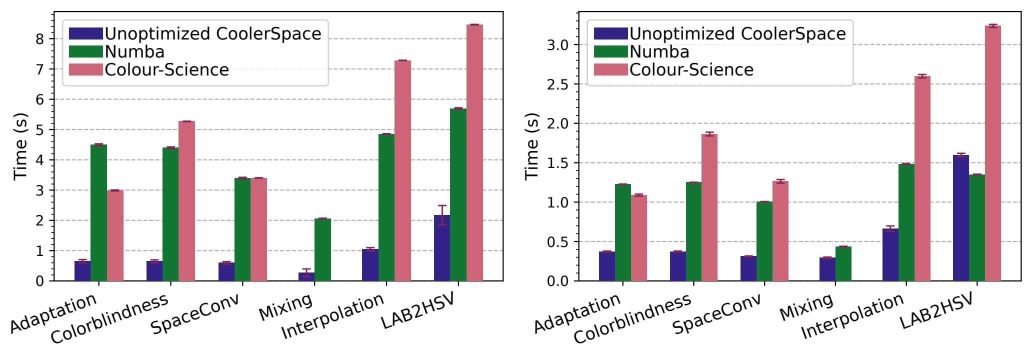
Even unoptimized CoolerSpace programs are faster than programs written using existing libraries by several times. Fig. 10 compares the performance of CoolerSpace's unoptimized ONNX executables with Numba and Colour-Science. We report the CPU comparison results here since both baseline implementations are CPU-based. CoolerSpace's has a 4.4× geomean speed-up over Numba, and a 5.7× geomean speed-up over Colour-Science on machine 1. On machine 2, CoolerSpace has 3.4× and 2.1× geomean speedups over Colour-Science and Numba. All speed-ups are significant ($p < 0.01$), with the exception of `LAB2HSV` on machine 2.



(a) Machine 1

(b) Machine 2

Fig. 9. Compilation and optimization time by program.



(a) Machine 1

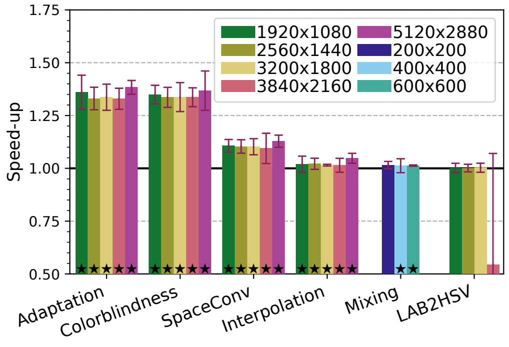
(b) Machine 2

Fig. 10. CoolerSpace average program execution time on CPU benchmarked against other Python solutions. Adaptation, Colorblindness, SpaceConv, Interpolation, and LAB2HSV were run on 5K images. Mixing was run on input sizes of 600 × 600 × 89. The error bars are 3 times the standard deviation of each set of tests. Colour-Science does not have a mixing program benchmark, as the library does not implement the Kubelka-Munk model. All Numba programs were compiled following the best practices recommended by Numba (e.g., with the `nopython` flag) for best performance. The JIT compile time costs of Numba are excluded from the time measurements.

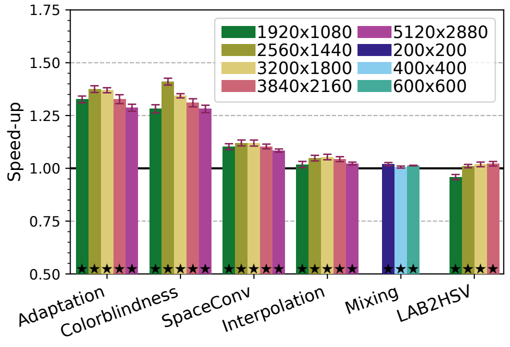
These results do not demonstrate CoolerSpace's optimization capabilities. Rather, the performance gains are a product of the differences between the libraries' software stacks. This evaluation is meant to show that, compared to existing color programming systems, CoolerSpace not only provides type safety guarantees but also does so without additional run-time overhead – in fact, we provide a performance improvement.

10.4 Optimization Effects

We have also benchmarked the GPU execution times of both our optimized and unoptimized ONNX files. Fig. 11 shows the speed-up of each GPU-run program under five different image resolutions. The only exceptions are Mixing and LAB2HSV, which use a smaller set of resolutions to prevent out-of-memory issues during the run time. Our optimization yields a speed-up of about 12% (geometric mean) across all programs and all resolutions on machine 1. The speed-up is about 15% on machine 2. A star in Fig. 11 indicates that the corresponding speedup is statistically



(a) Machine 1



(b) Machine 2

Fig. 11. Speed-ups by program under different resolutions. The error bars represent one standard deviation. A star above the bar indicates that the corresponding difference in runtime is statistically significant ($p < 0.01$).

significant. All programs show statistically significant speedups on some resolutions except for LAB2HSV on machine 1.

The speed-ups for ColorBlindness and Adaptation are attributed to the reduction in operations. This is because the original algorithms operate in the `LMS` space whereas the input images are in the `sRGB` space, so a color space transformation is necessary; our optimization identifies a reformulation of the arithmetics to operate directly in the `sRGB` space. The speed-ups for SpaceConv, Mixing, and Interpolation can be mostly attributed to constant folding. SpaceConv's execution time is dominated by a sequence of constant matrix multiplications, which are optimized away.

LAB2HSV at the highest resolution on machine 1 shows a significant slowdown with high variance after optimization. Further observation reveals a patch of 20 consecutive abnormally high run-times in the optimized LAB2HSV results. Transient memory management issues likely are the culprit, as this slowdown is not present on Machine 2. Machine 2 has 24 GB of VRAM, as opposed to machine 1's 8 GB of VRAM.

Further Optimizations. We choose to compile to ONNX because of practical considerations: ONNX is a convenient IR that is also cross-platform. However, ONNX and ONNX Runtime are not necessarily the most speed-efficient options. Our experiments show that equivalent CuPy [Okuta et al. 2017] code is 1.2× faster than corresponding CoolerSpace programs. The difference in execution time is a product of the different implementations of CuPy and ONNX Runtime and, potentially, the GPU code generated by CuPy and ONNX Runtime. In principle, we could directly compile to a more efficient target such as CuPy (or further translate from ONNX IR to that). However, such backend-specific and/or device-specific optimizations are out of the scope of the current paper, which focuses on enforcing color program correctness without additional run-time overhead.

11 Related Works

Color Programing Libraries. Commonly used Python libraries by color scientists include numpy [Harris et al. 2020], OpenCV [Bradski 2000], Pillow [Clark 2023], and Colour-Science [Developers 2015]. Unlike CoolerSpace, however, Numpy, OpenCV, and Colour-Science provide only a wrapper for tensor operations and color science algorithms without (type) checking physical correctness. Pillow has a small set of informal "types" (referred to as "modes") to track image representation and to validate operations, but is much weaker than CoolerSpace. For instance, Pillow can distinguish between `RGB` and `RGBA`, but not between actual color spaces: `opRGB` and `sRGB` images are treated identically. Pillow also lacks types for other important physical objects such as light and material properties. Finally, unlike all existing libraries, CoolerSpace is a meta-programming library, which compiles a Python program into an optimized ONNX file. Other Python libraries perform no performance optimization.

Domain-Specific Languages. CoolerSpace is a programming system for color science. Several domain-specific languages exist for the field of visual computing. All raise the level of programming abstraction. Gator [Geisler et al. 2020] introduces a type system for expressing coordinate systems in rendering. Simit [Kjolstad et al. 2016] is a language for physics simulation. Scenic [Fremont et al. 2019] is a language for probabilistically modeling a virtual scene. None of these domain specific languages target color programming. CoolerSpace uses tensor shape information to check the validity of operations. Similar static type checking systems for tensor shape are seen in array programming languages [Joisha and Banerjee 2006; Slepak et al. 2014].

Tensor Representations and Optimizations. CoolerSpace compiles user programs into tensor algebra represented by ONNX [Onnx 2018]. We chose ONNX because it is cross-platform and has a vibrant user community [Danopoulos et al. 2021; Jin et al. 2020].

CoolerSpace uses equality saturation to optimize tensor algebra [Tate et al. 2009]. Other tensor optimization techniques are in principle applicable too [Chen et al. 2018; Kjolstad et al. 2017; Susungi et al. 2018; Vasilache et al. 2018]. Our rewrite rules borrow from previous works on tensor optimization [Jia et al. 2019; Yang et al. 2021] but include color specific rules. Our cost function is based on a first-order estimation of operation counts; while empirically effective, future work can consider integrating hardware-aware models [Ahrens et al. 2022; Anderson et al. 2021; Liu et al. 2022]. Our implementation is based on the egg library [Willsey et al. 2021] with an extension to support constant propagation. While it is possible to use egg to implement constant propagation, it requires serializing a large amount of constant values, which might increase memory usage and optimization time. Our approach, by contrast, is symbolic.

Physical Unit Types. Researchers have previously explored the application of type theory to physical units and dimensions [Allen et al. 2004; Dreiheller et al. 1986; Karr and Loveman 1978]. This line of research encodes both physical units (i.e., meters, liters, kilograms) and dimensions[5] (i.e., length, volume, mass) as types. In physical unit types literature (also known as measurement types), a measurement can generally be converted to units of the same dimension, but not to units of other dimensions. For example, 1 minute can be converted to 60 seconds, as minutes and seconds are both units of the time dimension. 1 minute cannot be converted to grams. Such a conversion is nonsensical, as units of the time dimension cannot be converted to units of the mass dimension. CoolerSpace also contains multiple dimensions, but unlike measurement types, CoolerSpace allows conversion between units of different dimensions. This topic is discussed further in the **Casting** paragraph of Sec. 7.

Approximate Data Types and Information Flow Types. Both approximate [Sampson et al. 2011] and information flow [Myers 1999; Sabelfeld and Myers 2003] types enforce unidirectional information flow. In EnerJ [Sampson et al. 2011], precise to approximate data flow is allowed, but the reverse is prohibited. Similarly, information flow types prevent confidential data from affecting non-confidential outputs. Non-confidential data can affect confidential data. CoolerSpace also restricts data flow in a unidirectional manner. `Light` values can be coerced to `sRGB` values, but `sRGB` values can't be coerced to `Light` values.

In approximate data types and information flow type literature, the one directional information flow restrictions are designed to enforce best practices: it is feasible, if inadvisable, to openly share

[5]The dimension terminology here is not to be confused with dimension types in CoolerSpace. Dimensions in CoolerSpace refer to matrix dimensions.

password hashes. By contrast, the restrictions on information flow in CoolerSpace are informed by mathematics and physics: it is mathematically impossible to derive light spectrum data from an `sRGB` value because there are infinitely many physical light spectra that correspond to the same `sRGB` color. This topic is discussed further in the **Casting** paragraph of Sec. 7.

12 Conclusion

CoolerSpace's type system prevents mathematically permissible but physically meaningless or incorrect computations. CoolerSpace also automatically generates performance-optimized color science programs using equality saturation. We see CoolerSpace as the first step, rather than the final work, in raising the level of programming abstraction for physical sciences. Languages should empower domain experts to express the physical meaning of their programs. Correctness guarantees and performance optimizations should be left to the compiler and run-time system.

13 Acknowledgements

We would like to thank Professor Sreepathi Pai of the University of Rochester's Computer Science department for his feedback on our type system and soundness proofs. We would also like to thank the anonymous reviewers of ASPLOS'24 and OOPLSA'24 for their constructive criticism and helpful insight. The project is partially funded by NSF Award #2225860.

14 Data-Availability Statement

CoolerSpace has four artifacts: the CoolerSpace library [Chen 2024a], the ONNX optimizer [Chen and Chang 2024], a wrapper for the equality saturation rust library egg [Chang and Chen 2024; Willsey et al. 2021], and a set of benchmarking programs [Chen 2024b]. The programs are open-source and freely available on GitHub. Additionally, the all artifacts are available in Zenodo. The artifacts are also available on Zenodo [Chen et al. 2024].

References

2011. Answer to "adding/mixing colors in HSV Space". https://stackoverflow.com/a/7388476.

2014. Answer to "Interpolate from one color to another". https://stackoverflow.com/a/21010385.

2016. Weird interpolation between colors in hsv? https://stackoverflow.com/q/37471461.

2021. How to calculate (a physical) ratio of colors to achieve a target color? https://math.stackexchange.com/q/4335003.

adriahf. 2016. Increase the velocity of the calculations. · Issue #302 · colour-science/colour. https://github.com/colour-science/colour/issues/302

Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 269–285. https://doi.org/10.1145/3519939.3523442

Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele. 2004. Object-oriented units of measurement. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2004-10-01) *(Oopsla '04)*. Association for Computing Machinery, 384–403. https://doi.org/10.1145/1028976.1029008

Simon Amarasingham. 2019. Red and green Eclectus Parrots. https://www.flickr.com/photos/22896868@N05/49257147352/

Luke Anderson, Andrew Adams, Karima Ma, Tzu-Mao Li, Tian Jin, and Jonathan Ragan-Kelley. 2021. Efficient automatic scheduling of imaging and vision pipelines for the GPU. *Proceedings of the ACM on Programming Languages* 5, Oopsla (2021), 1–28. https://doi.org/10.1145/3485486

Roy S Berns. 2016. *Color science and the visual arts: a guide for conservators, curators, and the curious*. Getty Publications. https://doi.org/10.17613/d08z-ga34

G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

David H Brainard and Andrew Stockman. 2010. Colorimetry. *The Optical Society of America Handbook of Optics* 3 (2010), 10–1.

Hans Brettel, Françoise Viénot, and John D Mollon. 1997. Computerized simulation of color appearance for dichromats. *Josa a* 14, 10 (1997), 2647–2655. https://doi.org/10.1364/josaa.14.002647

Arthur D Broadbent. 2001. *Basic principles of textile coloration*. Vol. 132. Society of Dyers and Colorists Bradford.

Jiwon Chang and Ethan Chen. 2024. eggwrap. https://github.com/horizon-research/eggwrap

Ethan Chen. 2024a. CoolerSpace. https://github.com/horizon-research/CoolerSpace

Ethan Chen. 2024b. CoolerSpace Benchmarker. https://github.com/horizon-research/CoolerSpaceBenchmarker

Ethan Chen and Jiwon Chang. 2024. onneggs.

Ethan Chen, Jiwon Chang, and Yuhao Zhu. 2024. *CoolerSpace Artifacts*. https://doi.org/10.5281/zenodo.13621721

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594. https://doi.org/10.5555/3291168.3291211

Adam Chlipala, Leaf Petersen, and Robert Harper. 2005. Strict bidirectional type checking. In *Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (Long Beach, California, USA) *(Tldi '05)*. Association for Computing Machinery, New York, NY, USA, 71–78. https://doi.org/10.1145/1040294.1040301

Jeffrey Clark. 2023. Pillow. https://github.com/python-pillow/Pillow.

Dimitrios Danopoulos, Christoforos Kachris, and Dimitrios Soudris. 2021. Utilizing cloud FPGAs towards the open neural network standard. *Sustainable Computing: Informatics and Systems* 30 (2021), 100520. https://doi.org/10.1016/j.suscom.2021.100520

Colour Developers. 2015. Colour Science for Python. https://www.colour-science.org/.

ONNX Runtime Developers. 2021. ONNX Runtime. https://onnxruntime.ai/. Version: 1.16.

A Dreiheller, B Mohr, and M Moerschbacher. 1986. Programming pascal with physical units. *ACM SIGPLAN Notices* 21, 12 (Dec. 1986), 114–123. https://doi.org/10.1145/15042.15048

DR Duncan. 1940. The colour of pigment mixtures. *Proceedings of the Physical Society* 52, 3 (1940), 390.

Joshua Ebenezer. 2021. Computational speed for conversions · Issue #788 · colour-science/colour. https://github.com/colour-science/colour/issues/788

Mark D Fairchild and Lisa Reniff. 1995. Time course of chromatic adaptation for color-appearance judgments. *Josa A* 12, 5 (1995), 824–833. https://doi.org/10.1016/s0042-6989(00)00050-x

Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. 2019. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 63–78. https://doi.org/10.1145/3314221.3314633

Dietrich Geisler, Irene Yoon, Aditi Kabra, Horace He, Yinnon Sanders, and Adrian Sampson. 2020. Geometry types for graphics programming. *Proceedings of the ACM on Programming Languages* 4, Oopsla (2020), 1–25. https://doi.org/10.1145/3428241

Romain Guy. 2017. Understanding color. https://www.youtube.com/watch?v=r8NeG0wmFXM

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Trish Hartmann. 2012. Eleuthera Sunset. https://openverse.org/image/d693e7e7-d7aa-4801-96c5-56674e5715c6

ST Henderson and D Hodgkiss. 1963. The spectral energy distribution of daylight. *British Journal of Applied Physics* 14, 3 (1963), 125. https://doi.org/10.1088/0508-3443/15/8/310

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62. https://doi.org/0.1145/3341301.3359630

Tian Jin, Gheorghe-Teodor Bercea, Tung D Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, et al. 2020. Compiling onnx neural network models using mlir. *arXiv preprint arXiv:2008.08272* (2020). https://doi.org/10.48550/arXiv.2008.08272

Ruth Johnston-Feller. 2001. *Color science in the examination of museum objects: nondestructive procedures.* Getty Publications. https://doi.org/10.1002/col.10107

Pramod G Joisha and Prithviraj Banerjee. 2006. An algebraic array shape inference system for MATLAB®. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 5 (2006), 848–907. https://doi.org/10.1145/1152649.1152651

Michael Karr and David B. Loveman. 1978. Incorporation of units into programming languages. 21, 5 (1978), 385–391. https://doi.org/10.1145/359488.359501

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, Oopsla (2017), 1–29. https://doi.org/10.1145/3133901

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, et al. 2016. Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–21. https://doi.org/10.1145/2866569

Paul Kubelka. 1948. New contributions to the optics of intensely light-scattering materials. Part I. *Josa* 38, 5 (1948), 448–457.

Paul Kubelka and Franz Munk. 1931. An article on optics of paint layers. *Z. Tech. Phys* 12, 593-601 (1931), 259–274.

Daniël Lakens. 2017. Equivalence tests: A practical primer for t tests, correlations, and meta-analyses. *Social psychological and personality science* 8, 4 (2017), 355–362. https://doi.org/10.1177/1948550617697177

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(Llvm '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

Dmitry Lavrov. 2020. Reading 65^3 Iridas 3D LUT is incredibly slow. · Issue #573 · colour-science/colour. https://github.com/colour-science/colour/issues/573

Changjun Li, Zhiqiang Li, Zhifeng Wang, Yang Xu, Ming Ronnier Luo, Guihua Cui, Manuel Melgosa, Michael H Brill, and Michael Pointer. 2017. Comprehensive color solutions: CAM16, CAT16, and CAM16-UCS. *Color Research & Application* 42, 6 (2017), 703–718. https://doi.org/10.1002/col.22131

Bruce Lindbloom. 2017. XYZ to LAB. http://www.brucelindbloom.com/index.html?Eqn%5FXYZ%5Fto%5FLab.html

Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified tensor-program optimization via high-level scheduling rewrites. *Proceedings of the ACM on Programming Languages* 6, Popl (2022), 1–28. https://doi.org/10.1145/3498717

Steve Marschner and Peter Shirley. 2021. Chapter 14.6.1 Spectral Energy. In *Fundamentals of Computer Graphics*. AK Peters/CRC Press, 357–382. https://doi.org/10.1201/9781439865521

Francisco Massa. 2021. [feature request] rgb2lab / rgb2hsv / rgb2gray and other color space conversions (maybe upstream from kornia? or colorsys python core module?) · Issue #4029 · pytorch/vision. https://github.com/pytorch/vision/issues/4029

Don McCurdy. 2022. Color management. https://threejs.org/docs/#manual/en/introduction/Color-management

Michael E Miller and Spicer. 2019. *Color in Electronic Display Systems*. Springer. https://doi.org/10.1007/978-3-030-02834-3

Andrew C. Myers. 1999. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(Popl '99)*. Association for Computing Machinery, New York, NY, USA, 228–241. https://doi.org/10.1145/292540.292561

Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*.

Onnx. 2018. Open Neural Network Exchange. https://github.com/onnx/onnx.

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2023. *Physically based rendering: From theory to implementation*. MIT Press.

Charles Poynton. 2012. *Digital video and HD: Algorithms and Interfaces*. Elsevier. https://doi.org/10.1016/B978-0-12-391926-7.50059-X

Andy Rowlands. 2017. *Physics of digital photography*. IOP Publishing. https://doi.org/10.1088/978-0-7503-2558-5

D Andrew Rowlands. 2020. Color conversion matrices in digital cameras: a tutorial. *Optical Engineering* 59, 11 (2020), 110801. https://doi.org/10.1117/1.oe.59.11.110801

A. Sabelfeld and A.C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. https://doi.org/10.1109/jsac.2002.806121

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: approximate data types for safe and general low-power computation. *SIGPLAN Not.* 46, 6 (jun 2011), 164–174. https://doi.org/10.1145/1993316.1993518

Gaurav Sharma. 2017. Color fundamentals for digital imaging. In *Digital color imaging handbook*. CRC press, 1–114.

Gaurav Sharma and Raja Bala. 2017. *Digital color imaging handbook*. CRC press.

Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An array-oriented language with static rank polymorphism. In *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 23*. Springer, 27–46. https://doi.org/10.1007/978-3-642-54833-8_3

Šárka Sochorová and Ondřej Jamriška. 2021. Practical pigment mixing for digital painting. *ACM Transactions on Graphics (TOG)* 40, 6 (2021), 1–11. https://doi.org/10.1145/3478513.3480549

van der Walt Stefan. 2017. Finding color space information about image · Issue #2175 · scikit-image/scikit-image. https://github.com/scikit-image/scikit-image/issues/2175

Andrew Stockman and David H Brainard. 2010. Color vision mechanisms. *The Optical Society of America Handbook of Optics* 3 (2010), 11–1.

Adilla Susungi, Norman A Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. 2018. Meta-programming for cross-domain tensor optimizations. *ACM SIGPLAN Notices* 53, 9 (2018), 79–92. https://doi.org/10.1145/3278122.3278131

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276. https://doi.org/10.1145/1480881.1480915

Varkor. 2018. Types for units of measure. https://varkor.github.io/blog/2018/07/30/types-for-units-of-measure.html

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018). https://doi.org/10.48550/arXiv.1802.04730

Françoise Viénot, Hans Brettel, and John D Mollon. 1999. Digital video colourmaps for checking the legibility of displays by dichromats. *Color Research & Application* 24, 4 (1999), 243–252. https://doi.org/10.1002/(SICI)1520-6378(199908)24:4<243::AID-COL5>3.0.CO;2-3

Brian A Wandell. 1995. *Foundations of vision.* sinauer Associates.

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, Popl (2021), 1–29. https://doi.org/10.1145/3434304

Günther Wyszecki and Walter Stanley Stiles. 2000. *Color science: concepts and methods, quantitative data and formulae.* Vol. 40. John wiley & sons.

Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems* 3 (2021), 255–268.

Received 2024-04-05; accepted 2024-08-18