GLCM Vectorizing through NumPy

John Chang
Nanyang Technological University

2020
October

1 Introduction

This is written on the assumption that the reader understands GLCM.

As we understand, the Gray Level Co-occurrence Matrix is a separately generated matrix to calculate statistics such as **Contrast, Correlation and Entropy**.

However, from a programming perspective, it's computationally expensive and redundant.

In this, we'll discuss how to vectorize the operations for **Contrast and Correlation**, then discuss issues with vectorizing **Entropy**.

1.1 Preamble

The main issue with creating a GLCM is due to NumPy vectorizing not being able to work around it.

Consider the following:

```
a = [1,2,1,4,5]
b = [1,3,1,5,5]

c = np.vstack([a,b])

print(np.unique(c, return_counts=True, axis=1))
```

Returns the following:

```
(array([[1, 2, 4, 5],
        [1, 3, 5, 5]]),
 array([2, 1, 1, 1], dtype=int64))
```

This is a suitably fast enough operation if we needed the `counts`. However, `np.unique` is a large bottleneck on larger scale data.

Can we calculate all 3 statistics *(Contrast, Correlation and Entropy)* without calling `np.unique`?

Vanilla Conventions

These conventions are used in the GLCM context, whereever it's generated.

i, j Represents the labels on the 2 sides of the GLCM.

$n_{i,j}$ Represents the GLCM count for index i, j

$$\sum_{i=i_0, j=j_0}^{i=i, j=j} f(i, j, n_{i,j})$$

Means to loop through all cells, i_0 and j_0 can be arbitrarily defined. However it is paramount that it loops through all cells once.

$f(i, j, n_{i,j})$ is defined as the function that will occur on every i, j pair before summation.

Offset Arrays

GLCM generates itself by looking at a neighbour at a fixed offset. Instead of creating the GLCM right away, we can generate 2 arrays.

1. Original Array a
2. Target Array b

a, b Represents the original and target matrices.

Doesn't matter which is called the original or target.

Making Pairs

One important aspect of this is that we make 2 matrices with a single one by overlaying one with another. The shift is manually defined, in this we will shift diagonally by 1.

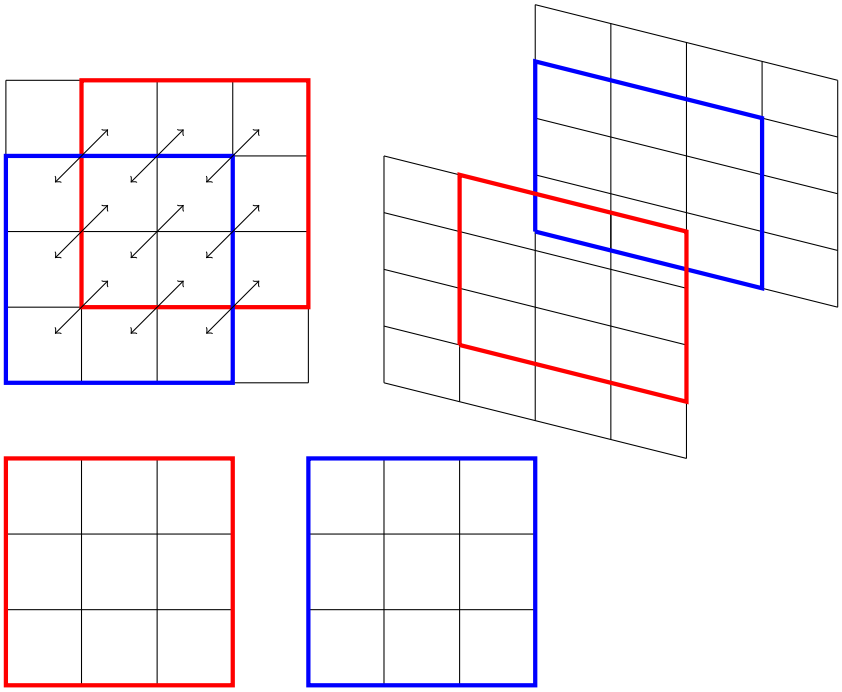

Figure 1: a, b from a single 2-D matrix

Kernel Convolution

GLCM works in local kernels, hence we need to partition it somehow.
It should iterate through all available windows, however, it's time consuming to do the following:

1. Split data to sliding windows
2. Operate on each window
3. Merge all data together

Hence, we would want to retain the original data as much as possible while mimicking that concept.

Example

Let's say we wanted just the sum of all 3x3 kernel windows on i or j.
For each 3x3 window, we calculate the following

$$\sum i$$

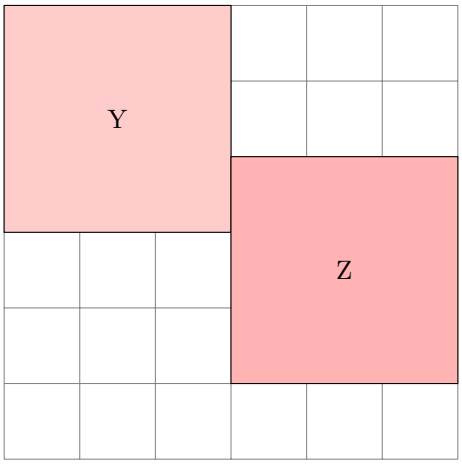

Figure 2: Y and Z Represents possible 3x3 windows

This can be solved using a convolution library.

1	1	1			
1	1	1			
1	1	1			

Figure 3: A 3x3 Ones Convolution Kernel

Using this 3x3 kernel, we can do a sliding window summation for all **valid** cells.

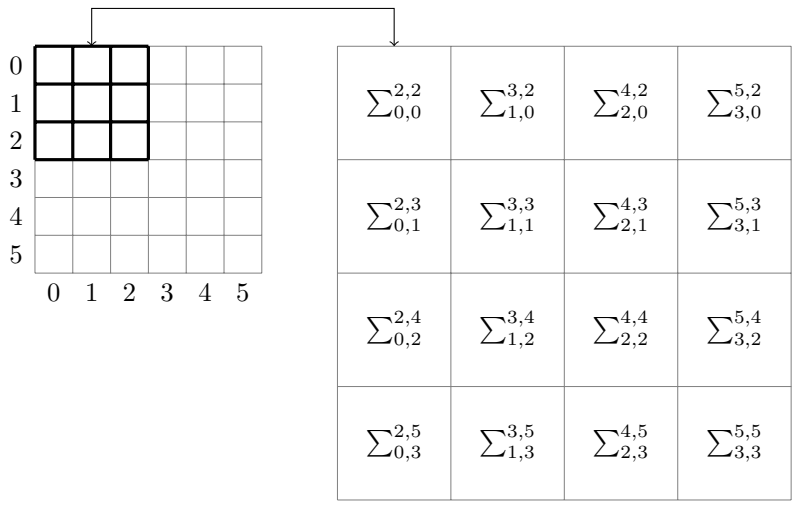

Figure 4: 3x3 Convolution Summation

A more detailed example utilizing this concept can be found in the section **Contrast** below.

1.2 Contrast

Contrast is defined as such

$$\sum_{i=i_0,j=j_0}^{i=i,j=j} (i-j)^2 * n_{i,j}$$

Let's say we have the following image data. We will do a 1-step diagonal GLCM.

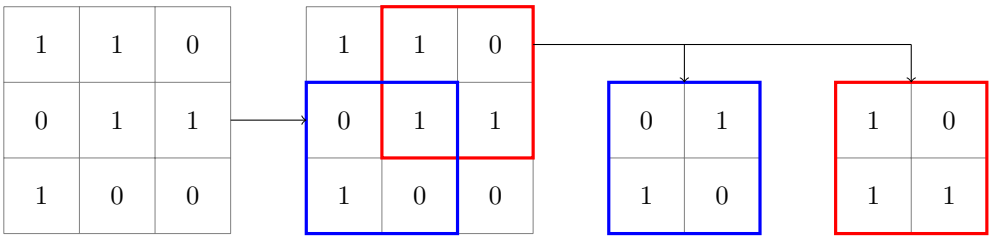

Figure 5: Example GLCM Pair Making

If we were to calculate the GLCM, it'll look something like this

	0	1
0	0	2
1	1	1

$$\begin{aligned}\sum (i-j)^2 * n &= (0-1)^2 * 2 + (1-0)^2 + (1-1)^2 \\ &= (0-1)^2 + (0-1)^2 + (1-0)^2 + (1-1)^2 \\ &= 3\end{aligned}$$

Notice how we expanded n, so we can vectorize it.
Let's say we don't generate the GLCM, instead we have the data as so, right after the **pair making**.

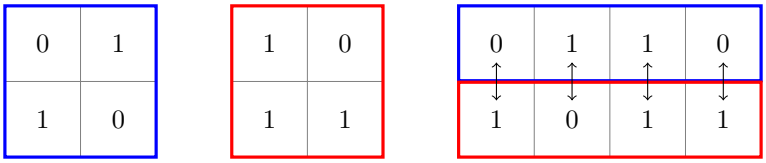

Figure 6: Flattening for Vectorization

$$\begin{aligned}\sum (i-j)^2 * n &= (0-1)^2 + (1-0)^2 + (1-0)^2 + (1-1)^2 \\ &= 3\end{aligned}$$

As you can see, we managed to avoid GLCM entirely because of the linear property of a summation.
We will use the same idea for **Correlation** below.

1.3 Correlation

Correlation is defined as such

$$\sum_{i=i_0,j=j_0}^{i=i,j=j} \frac{(i*j)*n_{i,j} - (\mu_x - \mu_y)}{\sigma_x * \sigma_y}$$

Where μ_x, μ_y are defined as the mean of all i and j values respectively.
σ_x, σ_y are defined as the standard deviation of i and j values respectively.
To make this formula simpler and relevant for our **Making Pairs** method, we reword it as such

$$\sum_{w=w_0}^{w=w} \sum_{a=a_0,b=b_0}^{a=a,b=b} \frac{(a*b) - (\mu_a - \mu_b)}{\sigma_a * \sigma_b}$$

In simpler terms

```
for a and b in each window pair:
(
sum all (a * b) cells
minus mean of a
plus mean of b # Note the sign!
)
all divided by
(
standard deviation of a
multiplied by standard deviation of b
)
```

Notice how we need to calculate the mean and standard deviation **within** each window pair. This raises a problem where we need to **Convolute** within the **Window Convolution**.
This is referred to as the *Inner Convolution*.

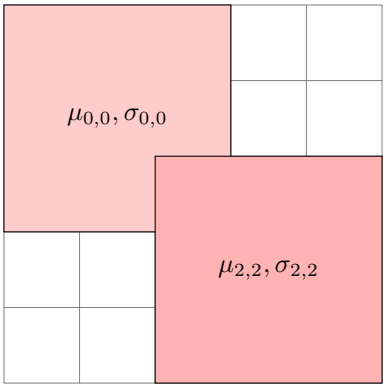

Figure 7: Correlation Inner Convolution 3x3

Inner Convolution Mean

Let's deal with μ, the mean first.

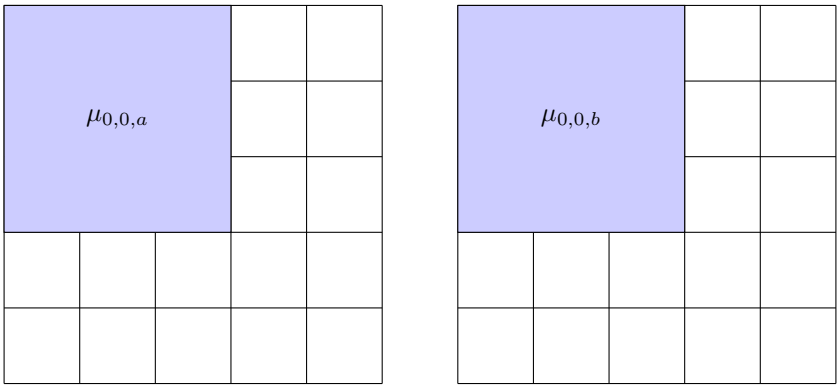

Figure 8: Correlation Inner Convolution Mean

In order to integrate this with a convolution, we just need to create a **ones kernel** then divide by the size or $1/size$ **kernel** and convolute that window across the data.

Inner Convolution Standard Deviation

We'll look at σ, which causes a bit of an issue.

The standard deviation function cannot be attached to the convolution **directly**. However, thankfully, we have an alternative formula.

Note: We only depict a here

$$Var = E(A^2) - E(A)^2$$
$$\sigma = \sqrt{E(A^2) - E(A)^2}$$

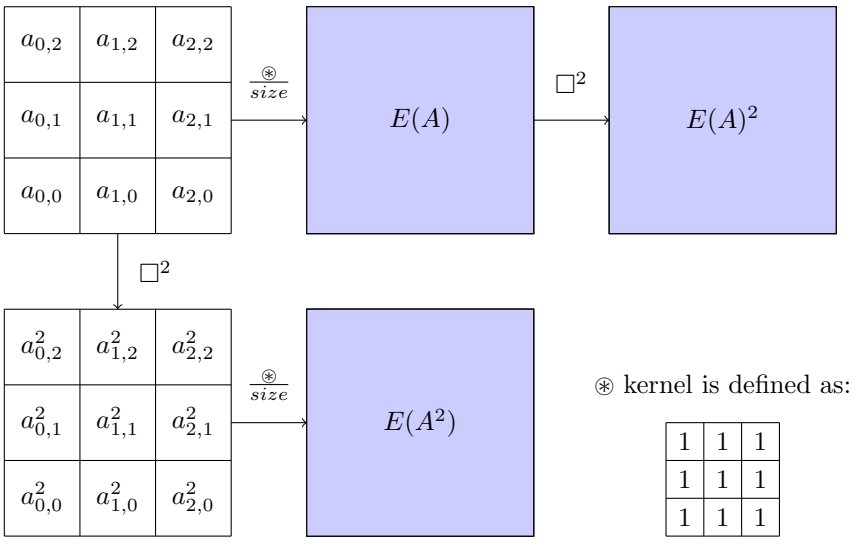

Figure 9: Correlation Inner Convolution Standard Deviation

Using this flow-chart, we can get our standard deviation locally for each kernel.

The rest of the steps are similar to **Contrast**, so I won't repeat it. The following shows a summary on how it's done in python.

```
kernel = np.ones(shape=[radius * 2 + 1, radius * 2 + 1, 1])

conv_ab = fftconvolve(a * b, kernel, mode='valid')

# E(A)
conv_a = fftconvolve(ai, kernel, mode='valid')
conv_ae = conv_a / kernel.size

# E(A^2)
conv_a2 = fftconvolve(a ** 2, kernel, mode='valid')
conv_ae2 = conv_i2 / kernel.size

# E(A)^2
conv_ae_2 = conv_ae ** 2

# Stdev(A)
conv_stda = np.sqrt(np.abs(conv_ae2 - conv_ae_2))

cor = (conv_ab - (conv_ae - conv_be)) / conv_stda * conv_stdb
```

Figure 10: Correlation Algorithm Summary. Self-explanatory variables are removed.

1.4 Entropy

This is one of the statistics that is not possible to vectorize efficiently.
Entropy is defined as such

$$\sum_{i=i_0,j=j_0}^{i=i,j=j} n^2$$

In other words, every occurrence is squared, then summed. Due to the squaring on the occurrences, we cannot simplify it for convolution vectorization.

Note that we'll use the GLCM convention i, j here instead of a, b.

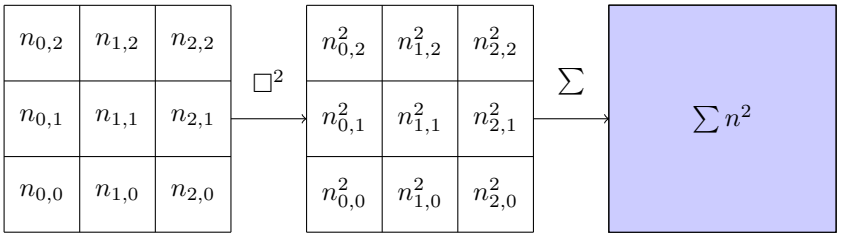

Figure 11: Entropy Calculation

Brute Force Approach

The current approach is simple

1. Multiply one by 256 and add both together
2. Partition summed data with an $n * n$ sliding window
3. For each window, we calculate local entropy

NumPy Unique and Bin Count

There's an issue with `np.unique` and `np.bin_count`, where it doesn't operate on multiple axes correctly.
Hence we require both i, j to be represented as a singular value.

$$\begin{aligned}(i, j) &\in \mathbb{Z} \\ max(i, j) &= (255, 255) \\ f(i, j) &= i * 256 + j\end{aligned}$$

For all (i, j) pairs, there's a unique $f(i, j)$ value.
Hence, we can call `np.unique` or `np.bin_count` on a the single dimension.

Performance of NumPy Unique and Bin Count

Tested it out with a small 200 x 200 image, `np.bin_count` seems to be the faster alternative.

Sliding Window

Instead of using **Convolution** as per the previous statistics, we have to use a sliding window as `np.bin_count` can't be vectorized.

Entropy Calculation

For each local sliding window, we can simply call `np.bin_count` on it then square each occurrence, then sum it.

```
entropy = np.sum(np.bincount(w) ** 2)
```

COO Sparse Matrix Approach

This is an alternative approach, where it's significantly slower, however it may be improved with low-level tweaking.

COO is a Co-ordinate based sparse matrix.

A **Sparse Matrix** is any matrix that is expected to have large amounts of *null values*, like 0.

By making i, j coordinates and feeding it into `scipy.sparse.coo_matrix`, it implicitly sums up occurrences. Hence, we can extract them as a `np.ndarray` and square sum it.

However, because it doesn't integrate well with multiple channels, discussed later, it is slower than **Brute Force**.

```
coo_r = coo_matrix((cd, (ca[..., 0], cb[..., 0])),
                   shape=(MAX_RGB, MAX_RGB))
                   .tocsr(copy=False).power(2).sum()
coo_g = coo_matrix((cd, (ca[..., 1], cb[..., 1])),
                   shape=(MAX_RGB, MAX_RGB))
                   .tocsr(copy=False).power(2).sum()
coo_b = coo_matrix((cd, (ca[..., 2], cb[..., 2])),
                   shape=(MAX_RGB, MAX_RGB))
                   .tocsr(copy=False).power(2).sum()
```

Notice how each channel must run their own `coo_matrix`, I speculate that its performance decreases here.

Multi Processing

In the library, I included an option to use multiple processors to loop on separate threads.

This is possible because it's a for-loop, this makes the entropy looping much faster.

1.5 Multiple Channels

One major consideration for the above algorithms is that it must work with multiple channels as we're looking for the following

	Con	Cor	Ent
R	ConR	CorR	EntR
G	ConG	CorG	EntG
B	ConB	CorB	EntB

Making a poor non-vectorized algorithm will result in heavy delays as they are processed linearly.

As most operations work with `np.ndarray`, most of it works fine with additional axes.

Channel Dimension

Those unfamiliar on how my data is original represented may find this useful.

An example of my original data would be `[3, 4, 3]`.

This means, 3 rows, 4 columns, 3 channels. If they were RGB channels, then it'll look like the following figures.

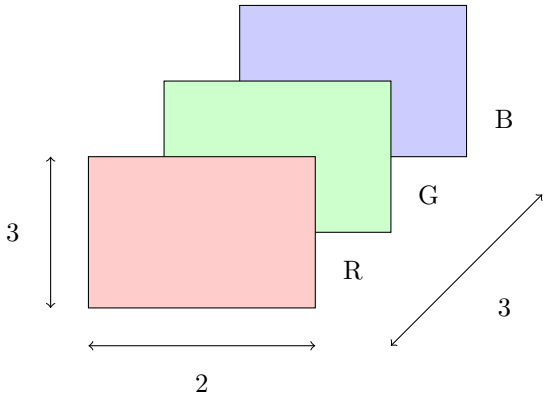

Figure 12: Channel Dimension Example 1

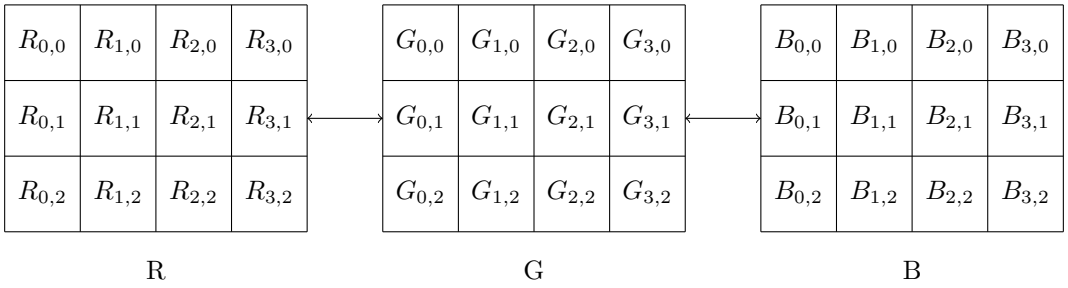

Figure 13: Channel Dimension Example 2

The last dimensions is always the *Channel Dimension*.
Hence if I would want to call operations on separate channels, I'd just have to specify to split on the last axis.

1.6 Other Channels

As of now GLCM works only with RGB, but can be adapted to other channels with tweaking.

- Entropy assumes RGB, hence it may need to multiply by other values other than 256 if changed
- Entropy assumes integer inputs, `np.bin_count` only works with integers, `np.unique` can be used but with a decrease in performance

2 Code Annex

2.1 Contrast

```
ar = (rgb_a - rgb_b) ** 2
     return fftconvolve(ar,
                        np.ones(shape=[radius * 2 + 1,
                                       radius * 2 + 1, 1]),
                        mode='valid')
```

2.2 Correlation

```
kernel = np.ones(shape=[radius * 2 + 1, radius * 2 + 1, 1])

conv_ab = fftconvolve(rgb_a * rgb_b, kernel, mode='valid')

conv_a = fftconvolve(rgb_a, kernel, mode='valid')
conv_b = fftconvolve(rgb_b, kernel, mode='valid')

# E(A) & E(B)
conv_ae = conv_a / kernel.size
conv_be = conv_b / kernel.size

conv_a2 = fftconvolve(rgb_a ** 2, kernel, mode='valid')
conv_b2 = fftconvolve(rgb_b ** 2, kernel, mode='valid')

# E(A^2) & E(B^2)
conv_ae2 = conv_a2 / kernel.size
conv_be2 = conv_b2 / kernel.size

# E(A)^2 & E(B)^2
conv_ae_2 = conv_ae ** 2
conv_be_2 = conv_be ** 2

conv_stda = np.sqrt(np.abs(conv_ae2 - conv_ae_2))
conv_stdb = np.sqrt(np.abs(conv_be2 - conv_be_2))

with np.errstate(divide='ignore', invalid='ignore'):
    cor = (conv_ab - (conv_ae - conv_be)) / conv_stda * conv_stdb
    return np.nan_to_num(cor, copy=False, nan=0, neginf=-1, posinf=1)
```

2.3 Entropy

```
rgb_a = rgb_a.astype(np.uint16)
rgb_b = rgb_b.astype(np.uint16)

cells = view_as_windows(self.init(rgb_a * (MAX_RGB + 1) + rgb_b).data,
                        [radius * 2 + 1, radius * 2 + 1,3],
                        step=1).squeeze()

out = np.zeros((rgb_a.shape[0] - radius * 2,
                rgb_a.shape[1] - radius * 2,
                3))  # RGB count

for row, _ in enumerate(cells):
    for col, cell in enumerate(_):
        c = cell.reshape([-1, cell.shape[-1]])
        entropy = np.asarray([np.sum(np.bincount(g) ** 2)
                              for g in c.swapaxes(0, 1)])
        out[row, col, :] = entropy
return out
```

2.4 Entropy Multiprocessing

```
p = Pool(procs) if procs else Pool()

for i, _ in enumerate(p.imap_unordered(self._get_glcm_entropy_mp_loop, cells)):
    out[i, :, :] = _

return out
```

2.5 Entropy COO (Deprecated)

Some functions may be deprecated.

```
w_a = self.init(rgb_a).slide_xy(by=radius * 2 + 1, stride=1)
w_b = self.init(rgb_b).slide_xy(by=radius * 2 + 1, stride=1)

out = np.zeros((rgb_a.shape[0] - radius * 2,
                rgb_a.shape[1] - radius * 2,
                3))  # RGB * Channel count

for col, (_a, _b) in enumerate(zip(w_a, w_b)):
    if verbose: print(f"GLCM Entropy: {col} / {len(w_a)}")
    for row, (ca, cb) in enumerate(zip(_a, _b)):
        # We flatten the x and y axis first.
        ca = ca.data.reshape([-1, ca.shape[-1]])
        cb = cb.data.reshape([-1, cb.shape[-1]])
        cd = np.ones(ca.shape[0])

        coo_r = coo_matrix((cd, (ca[..., 0], cb[..., 0])),
                           shape=(MAX_RGB, MAX_RGB))
                           .tocsr(copy=False).power(2).sum()
        coo_g = coo_matrix((cd, (ca[..., 1], cb[..., 1]))
                           shape=(MAX_RGB, MAX_RGB))
                           .tocsr(copy=False).power(2).sum()
        coo_b = coo_matrix((cd, (ca[..., 2], cb[..., 2]))
                           shape=(MAX_RGB, MAX_RGB))
                           .tocsr(copy=False).power(2).sum()

        out[row, col, :] = [coo_r, coo_g, coo_b]

return out
```