


# Using your GPU with CuPy

Last updated on 2023-11-11 | [Edit this page](#) 

## OVERVIEW

### Questions

- "How can I increase the performance of code that uses NumPy?"
- "How can I copy NumPy arrays to the GPU?"

### Objectives

- "Be able to indicate if an array, represented by a variable in an iPython shell, is stored in host or device memory."
- "Be able to copy the contents of this array from host to device memory and vice versa."
- "Be able to select the appropriate function to either convolve an image using either CPU or GPU compute power."
- "Be able to quickly estimate the speed benefits for a simple calculation by moving it from the CPU to the GPU."

## Introduction to CuPy

CuPy is a GPU array library that implements a subset of the NumPy and SciPy interfaces. This makes it a very convenient tool to use the compute power of GPUs for people that have some experience with NumPy, without the need to write code in a GPU programming language such as CUDA, OpenCL, or HIP.

## Convolution in Python

We start by generating an artificial "image" on the host using Python and NumPy; the host is the CPU on the laptop, desktop, or cluster node you are using right now, and from now on we may use *host* to refer to the CPU and *device* to refer to the GPU. The image will be all zeros, except for isolated pixels with value one, on a regular grid. The plan is to convolve it with a Gaussian and inspect the result. We will also record the time it takes to execute this convolution on the host.

We can interactively write and execute the code in an iPython shell or a Jupyter notebook.

```
import numpy as np

# Construct an image with repeated delta functions
deltas = np.zeros((2048, 2048))
deltas[8::16,8::16] = 1
```

PYTHON < >

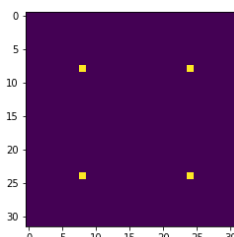
To get a feeling of how the whole image looks like, we can display the top-left corner of it.

```
import pylab as pyl
# Necessary command to render a matplotlib image in a Jupyter notebook.
%matplotlib inline

# Display the image
# You can zoom in using the menu in the window that will appear
pyl.imshow(deltas[0:32, 0:32])
pyl.show()
```

PYTHON < >

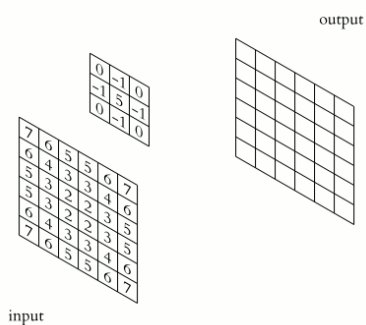
After executing the code, you should see the following image.



Deltas array

## GPU Programming: Using your GPU with CuPy

The computation we want to perform on this image is a convolution, once on the host and once on the device so we can compare the results and execution times. In computer vision applications, convolutions are often used to filter images and if you want to know more about them, we encourage you to check out [this github repository](#) by Vincent Dumoulin and Francesco Visin with some great animations. We have already seen that we can think of an image as a matrix of color values, when we convolve that image with a particular filter, we generate a new matrix with different color values. An example of convolution can be seen in the figure below (illustration by Michael Plotke, CC BY-SA 3.0, via Wikimedia Commons).



Example of animated convolution

In our example, we will convolve our image with a 2D Gaussian function shown below:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right),$$

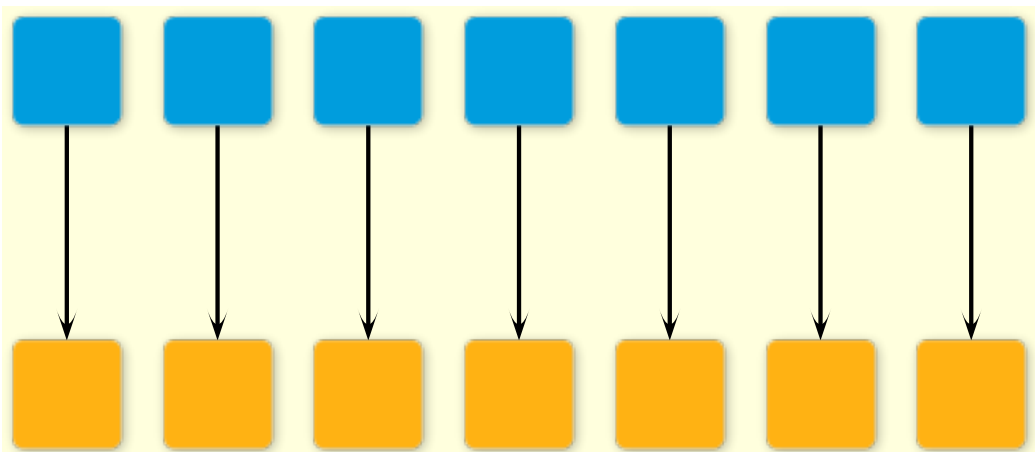
where  $x$  and  $y$  are the "coordinates in our matrix, i.e. our row and columns, and

$\sigma$

controls the width of the Gaussian distribution. Convolving images with 2D Gaussian functions will change the value of each pixel to be a weighted average of the pixels around it, thereby "smoothing" the image. Convolving images with a Gaussian function reduces the noise in the image, which is often required in [edge-detection](#) since most algorithms to do this are sensitive to noise.

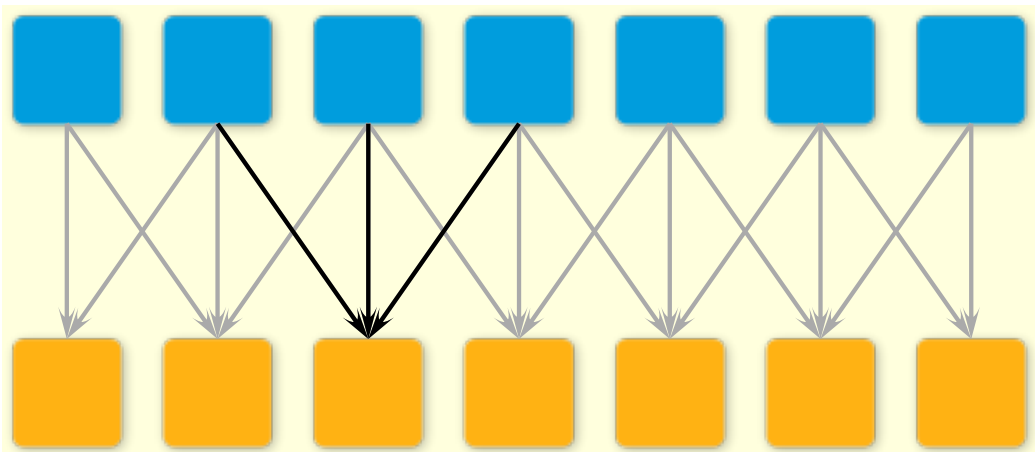
CALLOUT

It is often useful to identify the dataflow inherent in a problem. Say, if we want to square a list of numbers, all the operations are independent. The dataflow of a one-to-one operation is called a *map*.



Data flow of a map operation

A convolution is slightly more complicated. Here we have a many-to-one data flow, which is also known as a stencil.



Data flow of a stencil operation

GPU's are exceptionally well suited to compute algorithms that follow one of these patterns.

## Convolution on the CPU Using SciPy

Let us first construct the Gaussian, and then display it. Remember that at this point we are still doing everything with standard Python, and not using the GPU yet.

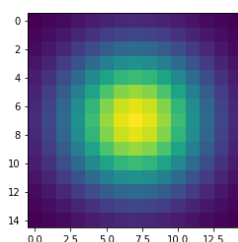
```

x, y = np.meshgrid(np.linspace(-2, 2, 15), np.linspace(-2, 2, 15))
dst = np.sqrt(x*x + y*y)
sigma = 1
muu = 0.000
gauss = np.exp(-((dst-muu)**2)/(2.0 * sigma**2))
pyl.imshow(gauss)
pyl.show()

```

PYTHON &lt; &gt;

This should show you a symmetrical two-dimensional Gaussian.



Two-dimensional Gaussian

## GPU Programming: Using your GPU with CuPy

Now we are ready to do the convolution on the host. We do not have to write this convolution function ourselves, as it is very conveniently provided by SciPy. Let us also record the time it takes to perform this convolution and inspect the top left corner of the convolved image.

```
from scipy.signal import convolve2d as convolve2d_cpu

convolved_image_using_CPU = convolve2d_cpu(deltas, gauss)
pyl.imshow(convolved_image_using_CPU[0:32, 0:32])
pyl.show()
%timeit -n 1 -r 1 convolve2d_cpu(deltas, gauss)
```

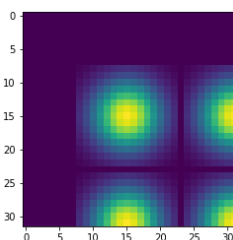
PYTHON < >

Obviously, the time to perform this convolution will depend very much on the power of your CPU, but I expect you to find that it takes a couple of seconds.

```
2.4 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

OUTPUT < >

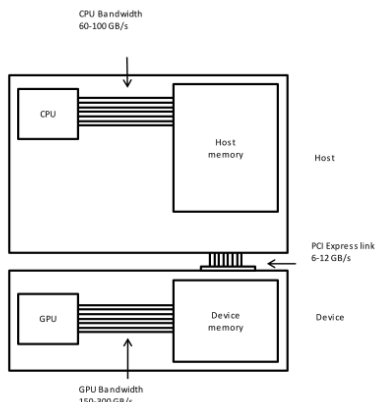
When you display the corner of the image, you can see that the "ones" surrounded by zeros have actually been blurred by a Gaussian, so we end up with a regular grid of Gaussians.



Regular grid of Gaussians

## Convolution on the GPU Using CuPy

This is part of a lesson on GPU programming, so let us use the GPU. Although there is a physical connection - i.e. a cable - between the CPU and the GPU, they do not share the same memory space. This image depicts the different components of CPU and GPU and how they are connected:



CPU and GPU are two separate entities, each with its own memory

This means that an array created from e.g. an iPython shell using NumPy is physically located into the main memory of the host, and therefore available for the CPU but not the GPU. It is not yet present in GPU memory, which means that we need to copy our data, the input image and the convolving function to the GPU, before we can execute any code on it. In practice, we have the arrays `deltas` and `gauss` in the host's RAM, and we need to copy them to GPU memory using CuPy.

```
import cupy as cp

deltas_gpu = cp.asarray(deltas)
gauss_gpu = cp.asarray(gauss)
```

PYTHON < >

Now it is time to do the convolution on the GPU. SciPy does not offer functions that can use the GPU, so we need to import the convolution function from another library, called `cupyx`; `cupyx.scipy` contains a subset of all SciPy routines. You will see that the GPU convolution function from the `cupyx` library looks very much like the convolution function from SciPy we used previously. In general, NumPy and CuPy look very similar, as well as the SciPy and `cupyx` libraries, and this is on purpose to facilitate the use of the GPU by programmers that are already familiar with NumPy and SciPy. Let us again record the time to execute the convolution, so that we can compare it with the time it took on the host.

PYTHON &lt; &gt;

```
from cupyx.scipy.signal import convolve2d as convolve2d_gpu

convolved_image_using_GPU = convolve2d_gpu(deltas_gpu, gauss_gpu)
%timeit -n 7 -r 1 convolved_image_using_GPU = convolve2d_gpu(deltas_gpu, gauss_gpu)
```

Similar to what we had previously on the host, the execution time of the GPU convolution will depend very much on the GPU used. These are the results using a NVIDIA Tesla T4 on Google Colab.

OUTPUT &lt; &gt;

```
98.2 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 7 loops each)
```

This is a lot faster than on the host, a performance improvement, or speedup, of 24439 times. Impressive, but is it true?

## Measuring performance

So far we used `timeit` to measure the performance of our Python code, no matter if it was running on the CPU or was GPU accelerated. However, execution on the GPU is asynchronous: the control is given back to the Python interpreter immediately, while the GPU is still executing the task. Because of this, we cannot use `timeit` anymore: the timing would not be correct.

CuPy provides a function, `benchmark` that we can use to measure the time it takes the GPU to execute our kernels.

PYTHON &lt; &gt;

```
from cupyx.profiler import benchmark

execution_gpu = benchmark(convolve2d_gpu, (deltas_gpu, gauss_gpu), n_repeat=10)
```

The previous code executes `convolve2d_gpu` ten times, and stores the execution time of each run, in seconds, inside the `gpu_times` attribute of the variable `execution_gpu`. We can then compute the average execution time and print it, as shown.

PYTHON &lt; &gt;

```
gpu_avg_time = np.average(execution_gpu.gpu_times)
print(f"{gpu_avg_time:.6f} s")
```

Another advantage of the `benchmark` method is that it excludes the compile time, and warms up the GPU, so that measurements are more stable and representative.

With the new measuring code in place, we can measure performance again.

OUTPUT &lt; &gt;

```
0.020642 s
```

We now have a more reasonable, but still impressive, speedup of 116 times over the host code.

### CHALLENGE: CONVOLUTION ON THE GPU WITHOUT CUPY

Try to convolve the NumPy array `deltas` with the NumPy array `gauss` directly on the GPU, without using CuPy arrays. If this works, it should save us the time and effort of transferring `deltas` and `gauss` to the GPU.

#### Solution

We can directly try to use the GPU convolution function `convolve2d_gpu` with `deltas` and `gauss` as inputs.

PYTHON &lt; &gt;

```
convolve2d_gpu(deltas, gauss)
```

However, this gives a long error message ending with:

OUTPUT &lt; &gt;

```
TypeError: Unsupported type <class 'numpy.ndarray'>
```

It is unfortunately not possible to access NumPy arrays directly from the GPU because they exist in the Random Access Memory (RAM) of the host and not in GPU memory.

## Validation

To check that we actually computed the same output on the host and the device we can compare the two output arrays `convolved_image_using_GPU` and `convolved_image_using_CPU`.

```
np.allclose(convolved_image_using_GPU, convolved_image_using_CPU)
```

[PYTHON < >](#)

As you may expect, the result of the comparison is positive, and in fact we computed the same results on the host and the device.

```
array(True)
```

[OUTPUT < >](#)

### CHALLENGE: FAIRER COMPARISON OF CPU VS. GPU

Compute again the speedup achieved using the GPU, but try to take also into account the time spent transferring the data to the GPU and back.

Hint: to copy a CuPy array back to the host (CPU), use the `cp.asnumpy()` function.

### Solution

A convenient solution is to group both the transfers, to and from the GPU, and the convolution into a single Python function, and then time its execution, like in the following example.

```
def transfer_compute_transferback():
    deltas_gpu = cp.asarray(deltas)
    gauss_gpu = cp.asarray(gauss)
    convolved_image_using_GPU = convolve2d_gpu(deltas_gpu, gauss_gpu)
    convolved_image_using_GPU_copied_to_host = cp.asnumpy(convolved_image_using_GPU)

    execution_gpu = benchmark(transfer_compute_transferback, (), n_repeat=10)
    gpu_avg_time = np.average(execution_gpu.gpu_times)
    print(f"{gpu_avg_time:.6f} s")
```

[PYTHON < >](#)

```
0.035400 s
```

[OUTPUT < >](#)

The speedup taking into account the data transfers decreased from 116 to 67. Taking into account the necessary data transfers when computing the speedup is a better, and more fair, way to compare performance. As a note, because data transfers force the GPU to sync with the host, this could also be measured with `timeit` and still provide correct measurements.

## A shortcut: performing NumPy routines on the GPU

We saw earlier that we cannot execute routines from the `cupyx` library directly on NumPy arrays. In fact we need to first transfer the data from host to device memory. Vice versa, if we try to execute a regular SciPy routine (i.e. designed to run on the CPU) on a CuPy array, we will also encounter an error. Try the following:

```
convolve2d_cpu(deltas_gpu, gauss_gpu)
```

[PYTHON < >](#)

This results in

```
.....
.....
.....
TypeError: Implicit conversion to a NumPy array is not allowed. Please use `.get()` to construct a NumPy array explicitly.
```

[OUTPUT < >](#)

So SciPy routines cannot have CuPy arrays as input. We can, however, execute a simpler command that does not require SciPy. Instead of 2D convolution, we can do 1D convolution. For that we can use a NumPy routine instead of a SciPy routine. The `convolve` routine from NumPy performs linear (1D) convolution. To generate some input for a linear convolution, we can flatten our image from 2D to 1D (using `ravel()`), but we also need a 1D kernel. For the latter we will take the diagonal elements of our 2D Gaussian kernel. Try the following three instructions for linear convolution on the CPU:

```
deltas_1d = deltas.ravel()
gauss_1d = gauss.diagonal()
%timeit -n 1 -r 1 np.convolve(deltas_1d, gauss_1d)
```

You could arrive at something similar to this timing result:

```
270 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

OUTPUT < >

We have performed a regular linear convolution using our CPU. Now let us try something bold. We will transfer the 1D arrays to the GPU and use the NumPy routine to do the convolution.

```
deltas_1d_gpu = cp.asarray(deltas_1d)
gauss_1d_gpu = cp.asarray(gauss_1d)

execution_gpu = benchmark(np.convolve, (deltas_1d_gpu, gauss_1d_gpu), n_repeat=10)
gpu_avg_time = np.average(execution_gpu.gpu_times)
print(f"{gpu_avg_time:.6f} s")
```

PYTHON < >

You may be surprised that we can issue these commands without error. Contrary to SciPy routines, NumPy accepts CuPy arrays, i.e. arrays that exist in GPU memory, as input. In the [CuPy documentation](#) you can find some background on why NumPy routines can handle CuPy arrays.

Also, remember when we used `np.allclose` with a NumPy and a CuPy array as input? That worked for the same reason.

The linear convolution is actually performed on the GPU, which also results in a lower execution time.

```
0.014529 s
```

OUTPUT < >

Without much effort, we obtained a 18 times speedup.

## A real world example: image processing for radio astronomy

In this section, we will perform the four major steps in image processing for radio astronomy: determination of background characteristics, segmentation, connected component labelling and source measurements.

### Import the FITS image

Start by importing a 2048<sup>2</sup> pixels image of the Galactic Center, an image made from observations by the Indian Giant Metrewave Radio Telescope (GMRT) at 150 MHz. The image is stored in [this repository](#) as a FITS file, and to read it we need the `astroropy` Python package.

```
from astroropy.io import fits

with fits.open("GMRT_image_of_Galactic_Center.fits") as hdul:
    data = hdul[0].data.byteswap().newbyteorder()
```

PYTHON < >

The latter two methods are needed to convert byte ordering from big endian to little endian.

### Inspect the image

Let us have a look at part of this image.

```
from matplotlib.colors import LogNorm

maxim = data.max()

fig = plt.figure(figsize=(50, 12.5))
ax = fig.add_subplot(1, 1, 1)
im_plot = ax.imshow(np.fliplr(data), cmap=plt.cm.gray_r, norm=LogNorm(vmin = maxim/10, vmax=maxim/100))
plt.colorbar(im_plot, ax=ax)
```

PYTHON < >



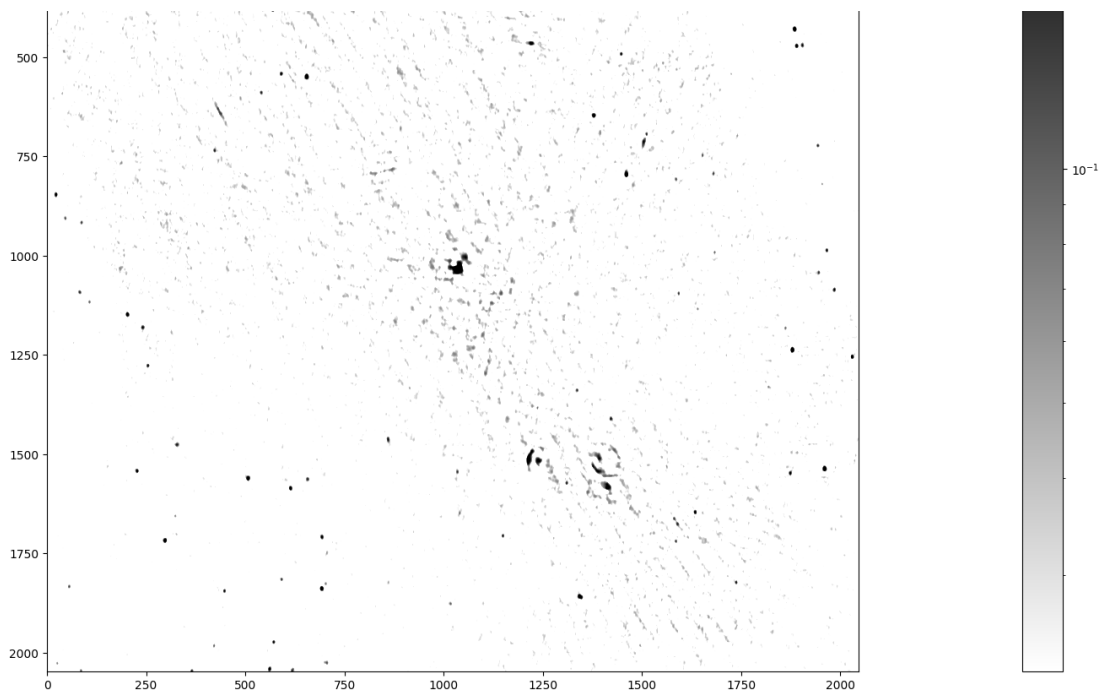


Image of the Galactic Center

The data has been switched left to right because Right Ascension increases to the left, so now it adheres to this astronomical convention. We can see a few dozen sources, especially in the lower left and upper right, which both have relatively good image quality. The band across the image from upper left to lower right has the worst image quality because there are many extended sources - especially the Galactic Center itself, in the middle - which are hard to deconvolve with the limited spacings from this observation. You can, however, see a couple of ring-like structures. These are actually supernova shells, i.e. the remnants from massive stars that exploded at the end of their lives.

## Determine the background characteristics of the image

We want to identify all the sources - meaning e.g. stars, supernova remnants and distant galaxies - in this image and measure their positions and fluxes. How do we separate source pixels from background pixels? When do we know if a pixel with a high value belongs to a source or is simply a noise peak? We assume the background noise, which is a reflection of the limited sensitivity of the radio telescope, has a normal distribution. The chance of having a background pixel with a value above 5 times the standard deviation is  $2.9e-7$ . We have  $2^{22} = 4.2e6$  pixels in our image, so the chance of catching at least one random noise peak by setting a threshold of 5 times the standard deviation is less than 50%. We refer to the standard deviation as  $\sigma$ .

How do we measure the standard deviation of the background pixels? First we need to separate them from the source pixels, based on their values, in the sense that high pixel values more likely come from sources. The technique we use is called  $\kappa, \sigma$  clipping. First we take all pixels and compute the standard deviation ( $\sigma$ ). Then we compute the median and clip all pixels larger than  $median + 3 * \sigma$  and smaller than  $median - 3 * \sigma$ . From the clipped set, we compute the median and standard deviation again and clip again. Continue until no more pixels are clipped. The standard deviation from this final set of pixel values is the basis for the next step.

Before clipping, let us investigate some properties of our unclipped data.

```

mean_ = data.mean()
median_ = np.median(data)
stddev_ = np.std(data)
max_ = np.amax(data)
print(f"mean = {mean_:.3e}, median = {median_:.3e}, stddev = {stddev_:.3e}, \
maximum = {max_:.3e}")

```

PYTHON &lt; &gt;

The maximum flux density is 2506 mJy/beam, coming from the Galactic Center itself, so from the center of the image, while the overall standard deviation is 19.9 mJy/beam:

```

mean = 3.898e-04, median = 1.571e-05, stddev = 1.993e-02, maximum = 2.506e+00

```

OUTPUT &lt; &gt;

You might observe that  $\kappa, \sigma$  clipping is a compute intense task, that is why we want to do it on a GPU. But let's first issue the algorithm on a CPU.

This is the NumPy code to do this:

PYTHON &lt; &gt;



```

# Flattening our 2D data first makes subsequent steps easier.
data_flat = data.ravel()
# Here is a kappa, sigma clipper for the CPU
def kappa_sigma_clipper(data_flat):
    while True:
        med = np.median(data_flat)
        std = np.std(data_flat)
        clipped_lower = data_flat.compress(data_flat > med - 3 * std)
        clipped_both = clipped_lower.compress(clipped_lower < med + 3 * std)
        if len(clipped_both) == len(data_flat):
            break
        data_flat = clipped_both
    return data_flat

data_clipped = kappa_sigma_clipper(data_flat)
timing_ks_clipping_cpu = %timeit -o kappa_sigma_clipper(data_flat)
fastest_ks_clipping_cpu = timing_ks_clipping_cpu.best
print(f"Fastest CPU ks clipping time = \
      {1000 * fastest_ks_clipping_cpu:.3e} ms.")

```

OUTPUT &lt; &gt;

```

793 ms ± 17.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Fastest CPU ks clipping time = 7.777e+02 ms.

```

So that is close to 1 second to perform these computations. Hopefully, we can speed this up using the GPU. How has  $\kappa, \sigma$  clipping influenced our statistics?

PYTHON &lt; &gt;

```

clipped_mean_ = data_clipped.mean()
clipped_median_ = np.median(data_clipped)
clipped_stddev_ = np.std(data_clipped)
clipped_max_ = np.amax(data_clipped)
print(f"mean of clipped = {clipped_mean_:.3e}, median of clipped = \
      {clipped_median_:.3e} \n standard deviation of clipped = \
      {clipped_stddev_:.3e}, maximum of clipped = {clipped_max_:.3e}")

```

All output statistics have become smaller which is reassuring; it seems `data_clipped` contains mostly background pixels:

OUTPUT &lt; &gt;

```

mean of clipped = -1.945e-06, median of clipped = -9.796e-06
standard deviation of clipped = 1.334e-02, maximum of clipped = 4.000e-02

```

#### CHALLENGE: $\kappa, \sigma$ CLIPPING ON THE GPU

Now that you understand how the  $\kappa, \sigma$  clipping algorithm works, perform it on the GPU using CuPy and compute the speedup. Include the data transfer to and from the GPU in your calculation.

Solution

```

PYTHON < >
def ks_clipper_gpu(data_flat):
    data_flat_gpu = cp.asarray(data_flat)
    data_gpu_clipped = kappa_sigma_clipper(data_flat_gpu)
    return cp.asnumpy(data_gpu_clipped)

data_clipped_on_GPU = ks_clipper_gpu(data_flat)
timing_ks_clipping_gpu = benchmark(ks_clipper_gpu, \
    (data_flat, ), n_repeat=10)
fastest_ks_clipping_gpu = np.amin(timing_ks_clipping_gpu.gpu_times)
print(f"1000 * fastest_ks_clipping_gpu:.3e} ms")

```

```

OUTPUT < >
6.329e+01 ms

```

```

PYTHON < >
speedup_factor = fastest_ks_clipping_cpu/fastest_ks_clipping_gpu
print(f"The speedup factor for ks clipping is: {speedup_factor:.3e}")

```

```

OUTPUT < >
The speedup factor for ks clipping is: 1.232e+01

```

## Segment the image

We have seen that clipping at the  $5\sigma$  level of an image this size (2048<sup>2</sup> pixels) will yield a chance of less than 50% that from all the sources we detect at least one will be a noise peak. So let us set the threshold at  $5\sigma$  and segment it. First check that we find the same standard deviation from our clipper on the GPU:

```

PYTHON < >
stddev_gpu_ = np.std(data_clipped_on_GPU)
print(f"standard deviation of background_noise = {stddev_gpu_:.4f} Jy/beam")

```

```

OUTPUT < >
standard deviation of background_noise = 0.0133 Jy/beam

```

With the standard deviation computed we apply the  $5\sigma$  threshold to the image.

```

PYTHON < >
threshold = 5 * stddev_gpu_
segmented_image = np.where(data > threshold, 1, 0)
timing_segmentation_CPU = %timeit -o np.where(data > threshold, 1, 0)
fastest_segmentation_CPU = timing_segmentation_CPU.best
print(f"Fastest CPU segmentation time = {1000 * fastest_segmentation_CPU:.3e} \
ms.")

```

```

OUTPUT < >
6.41 ms ± 55.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
Fastest CPU segmentation time = 6.294e+00 ms.

```

## Labelling of the segmented data

This is called connected component labelling (CCL). It will replace pixel values in the segmented image - just consisting of zeros and ones - of the first connected group of pixels with the value 1 - so nothing changed, but just for that first group - the pixel values in the second group of connected pixels will all be 2, the third connected group of pixels will all have the value 3 etc.

This is a CPU code for connected component labelling.

PYTHON < >

```

from scipy.ndimage import label as label_cpu
labelled_image = np.empty(data.shape)
number_of_sources_in_image = label_cpu(segmented_image, output = labelled_image)
sigma_unicode = "\u03C3"
print(f"The number of sources in the image at the 5{sigma_unicode} level is \
{number_of_sources_in_image}.")

timing_CCL_CPU = %timeit -o label_cpu(segmented_image, output = labelled_image)
fastest_CCL_CPU = timing_CCL_CPU.best
print(f"Fastest CPU CCL time = {1000 * fastest_CCL_CPU:.3e} ms.")

```

This gives, on my machine:

```

The number of sources in the image at the 5σ level is 185.
26.3 ms ± 965 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)
Fastest CPU CCL time = 2.546e+01 ms.

```

Let us not just accept the answer, but also do a sanity check. What are the values in the labelled image?

```

print(f"These are all the pixel values we can find in the labelled image: \
{np.unique(labelled_image)}")

```

This should show the following output:

```

These are all the pixel values we can find in the labelled image: [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11.
14. 15. 16. 17. 18. 19. 20. 21. 22. 23. 24. 25. 26. 27.
28. 29. 30. 31. 32. 33. 34. 35. 36. 37. 38. 39. 40. 41.
42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53. 54. 55.
56. 57. 58. 59. 60. 61. 62. 63. 64. 65. 66. 67. 68. 69.
70. 71. 72. 73. 74. 75. 76. 77. 78. 79. 80. 81. 82. 83.
84. 85. 86. 87. 88. 89. 90. 91. 92. 93. 94. 95. 96. 97.
98. 99. 100. 101. 102. 103. 104. 105. 106. 107. 108. 109. 110. 111.
112. 113. 114. 115. 116. 117. 118. 119. 120. 121. 122. 123. 124. 125.
126. 127. 128. 129. 130. 131. 132. 133. 134. 135. 136. 137. 138. 139.
140. 141. 142. 143. 144. 145. 146. 147. 148. 149. 150. 151. 152. 153.
154. 155. 156. 157. 158. 159. 160. 161. 162. 163. 164. 165. 166. 167.
168. 169. 170. 171. 172. 173. 174. 175. 176. 177. 178. 179. 180. 181.
182. 183. 184. 185.]

```

## Source measurements

We are ready for the final step. We have been given observing time to make this beautiful image of the Galactic Center, we have determined its background statistics, we have separated actual cosmic sources from noise and now we want to measure these cosmic sources. What are their positions and what are their flux densities?

Again, the algorithms from `scipy.ndimage` help us to determine these quantities. This is the CPU code for measuring our sources.

```

from scipy.ndimage import center_of_mass as com_cpu
from scipy.ndimage import sum_labels as sl_cpu
all_positions = com_cpu(data, labelled_image, \
                        range(1, number_of_sources_in_image+1))
all_integrated_fluxes = sl_cpu(data, labelled_image, \
                              range(1, number_of_sources_in_image+1))

print (f'These are the ten highest integrated fluxes of the sources in my \n\
image: {np.sort(all_integrated_fluxes)[-10:]}')

```

which gives the Galactic Center as the most luminous source, which makes sense when we look at our image.

```

These are the ten highest integrated fluxes of the sources in my image: [ 38.90615184 41.91485894 43.02203498 47.30590784 5:
58.07289425 68.85673917 70.31223921 95.16443585 363.58937774]

```

Now we can try to measure the execution times for both algorithms, like this:

## GPU Programming: Using your GPU with CuPy

```
%timeit -o
all_positions = com_cpu(data, labelled_image, \
                        range(1, number_of_sources_in_image+1))
all_integrated_fluxes = sl_cpu(data, labelled_image, \
                               range(1, number_of_sources_in_image+1))
```

Which yields, on my machine:

```
797 ms ± 9.32 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
<TimeitResult : 797 ms ± 9.32 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)>
```

[OUTPUT < >](#)

To collect the result from that timing in our next cell block, we need a trick that uses the `_` variable.

```
timing_source_measurements_CPU = _
fastest_source_measurements_CPU = timing_source_measurements_CPU.best
print(f"Fastest CPU set of source measurements = \
{1000 * fastest_source_measurements_CPU:.3e} ms.")
```

[PYTHON < >](#)

Which yields

```
Fastest CPU set of source measurements = 7.838e+02 ms.
```

[OUTPUT < >](#)

### CHALLENGE: PUTTING IT ALL TOGETHER

Combine the first two steps of image processing for astronomy, i.e. determining background characteristics e.g. through  $\kappa$ ,  $\sigma$  clipping and segmentation into a single function, that works for both CPU and GPU. Next, write a function for connected component labelling and source measurements on the GPU and calculate the overall speedup factor for the combined four steps of image processing in astronomy on the GPU relative to the CPU. Finally, verify your output by comparing with the previous output, using the CPU.

Solution

PYTHON &lt; &gt;

```

def first_two_steps_for_both_CPU_and_GPU(data):
    data_flat = data.ravel()
    data_clipped = kappa_sigma_clipper(data_flat)
    stddev_ = np.std(data_clipped)
    threshold = 5 * stddev_
    segmented_image = np.where(data > threshold, 1, 0)
    return segmented_image

def ccl_and_source_measurements_on_CPU(data_CPU, segmented_image_CPU):
    labelled_image_CPU = np.empty(data_CPU.shape)
    number_of_sources_in_image = label_cpu(segmented_image_CPU,
                                           output=labelled_image_CPU)
    all_positions = com_cpu(data_CPU, labelled_image_CPU,
                           np.arange(1, number_of_sources_in_image+1))
    all_fluxes = sl_cpu(data_CPU, labelled_image_CPU,
                       np.arange(1, number_of_sources_in_image+1))
    return np.array(all_positions), np.array(all_fluxes)

CPU_output = ccl_and_source_measurements_on_CPU(data, \
        first_two_steps_for_both_CPU_and_GPU(data))

timing_complete_processing_CPU = \
    benchmark(ccl_and_source_measurements_on_CPU, (data, \
        first_two_steps_for_both_CPU_and_GPU(data)), \
        n_repeat=10)

fastest_complete_processing_CPU = \
    np.amin(timing_complete_processing_CPU.cpu_times)

print(f"The four steps of image processing for astronomy take \
{1000 * fastest_complete_processing_CPU:.3e} ms\n on our CPU.")

from cupyx.scipy.ndimage import label as label_gpu
from cupyx.scipy.ndimage import center_of_mass as com_gpu
from cupyx.scipy.ndimage import sum_labels as sl_gpu

def ccl_and_source_measurements_on_GPU(data_GPU, segmented_image_GPU):
    labelled_image_GPU = cp.empty(data_GPU.shape)
    number_of_sources_in_image = label_gpu(segmented_image_GPU,
                                           output=labelled_image_GPU)
    all_positions = com_gpu(data_GPU, labelled_image_GPU,
                           cp.arange(1, number_of_sources_in_image+1))
    all_fluxes = sl_gpu(data_GPU, labelled_image_GPU,
                       cp.arange(1, number_of_sources_in_image+1))
    # This seems redundant, but we want to return ndarrays (Numpy)
    # and what we have are lists. These first have to be converted to
    # Cupy arrays before they can be converted to Numpy arrays.
    return cp.asnumpy(cp.asarray(all_positions)), \
           cp.asnumpy(cp.asarray(all_fluxes))

GPU_output = ccl_and_source_measurements_on_GPU(cp.asarray(data), \
        first_two_steps_for_both_CPU_and_GPU(cp.asarray(data)))

timing_complete_processing_GPU = \
    benchmark(ccl_and_source_measurements_on_GPU, (cp.asarray(data), \
        first_two_steps_for_both_CPU_and_GPU(cp.asarray(data))), \
        n_repeat=10)

fastest_complete_processing_GPU = \
    np.amin(timing_complete_processing_GPU.gpu_times)

print(f"The four steps of image processing for astronomy take \
{1000 * fastest_complete_processing_GPU:.3e} ms\n on our GPU.")

overall_speedup_factor = fastest_complete_processing_CPU / \
    fastest_complete_processing_GPU
print(f"This means that the overall speedup factor GPU vs CPU equals: \
{overall_speedup_factor:.3e}\n")

all_positions_agree = np.allclose(CPU_output[0], GPU_output[0])
print(f"The CPU and GPU positions agree: {all_positions_agree}\n")

all_fluxes_agree = np.allclose(CPU_output[1], GPU_output[1])
print(f"The CPU and GPU fluxes agree: {all_positions_agree}\n")

```

OUTPUT &lt; &gt;

```
The four steps of image processing for astronomy take 1.060e+03 ms
on our CPU.
The four steps of image processing for astronomy take 5.770e+01 ms
on our GPU.
This means that the overall speedup factor GPU vs CPU equals: 1.838e+01

The CPU and GPU positions agree: True

The CPU and GPU fluxes agree: True
```

### KEYPOINTS

- "CuPy provides GPU accelerated version of many NumPy and Scipy functions."
- "Always have CPU and GPU versions of your code so that you can compare performance, as well as validate your code."