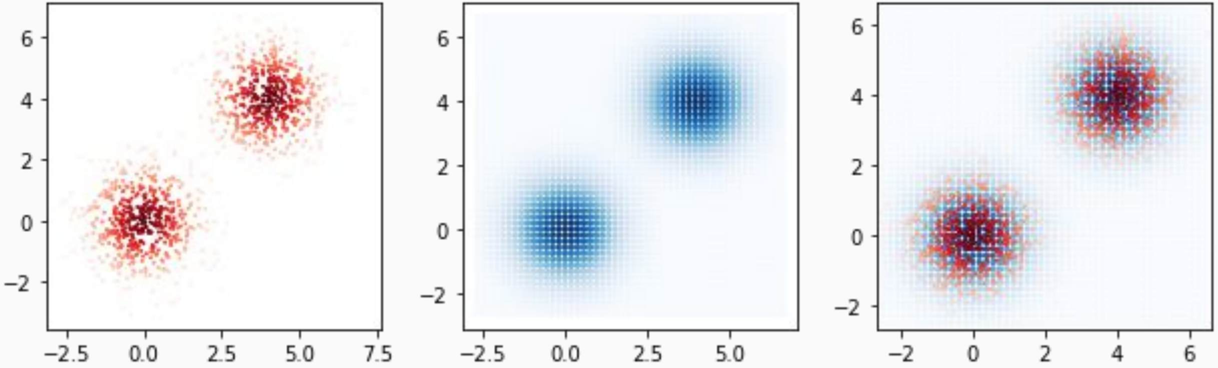
Progress Report

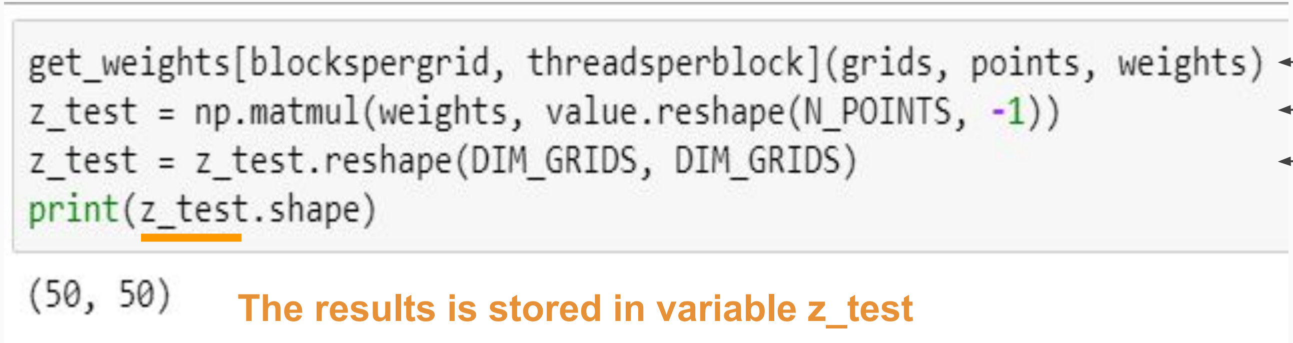
Kuei-Yueh (Clint) Ko

Try skcuda on small example

```
N_POINTS = 1000
DIM_GRIDS = 50
mean = ([0.0, 0.0], [4.0, 4.0])
prop = (0.5, 0.5)
```



Try skcuda on small example



```
get_weights[blockspergrid, threadsperblock](grids, points, weights)
z_test = np.matmul(weights, value.reshape(N_POINTS, -1))
z_test = z_test.reshape(DIM_GRIDS, DIM_GRIDS)
print(z_test.shape)
```

```
(50, 50)
```

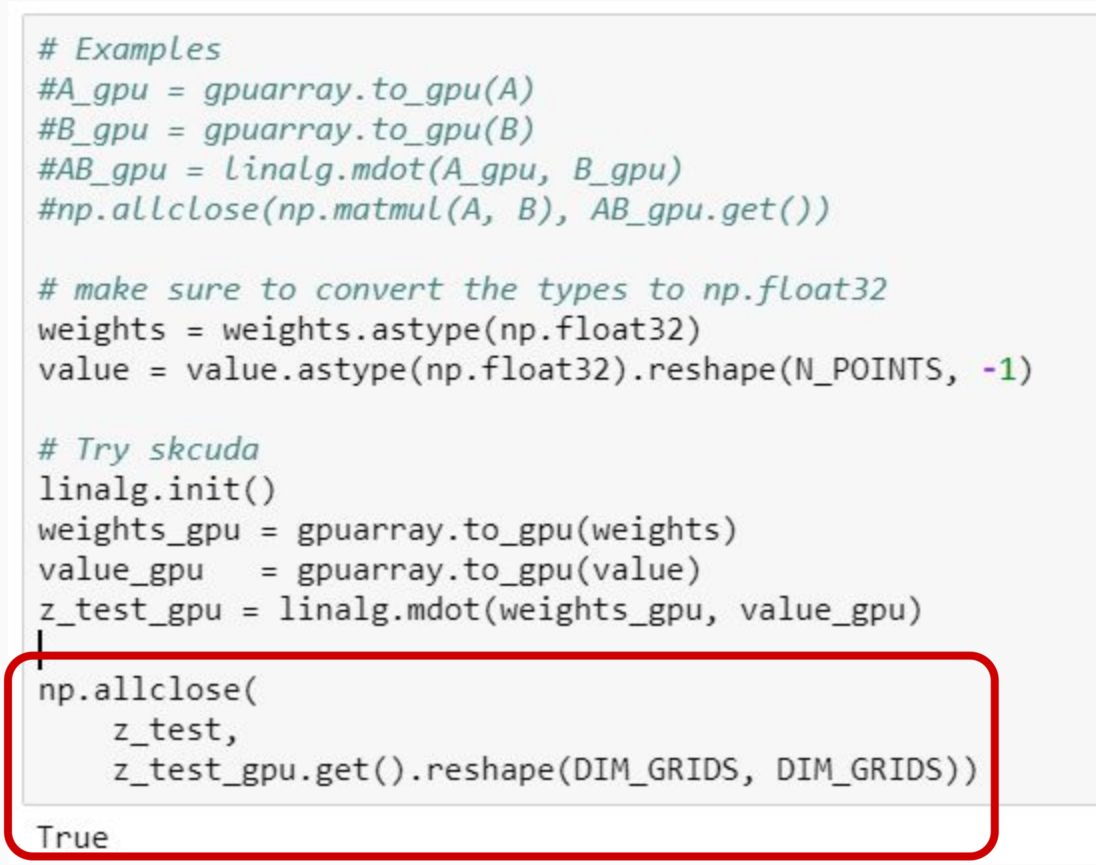
The results is stored in variable z_test

Calculate weights

Matrix multiplication

Change shape

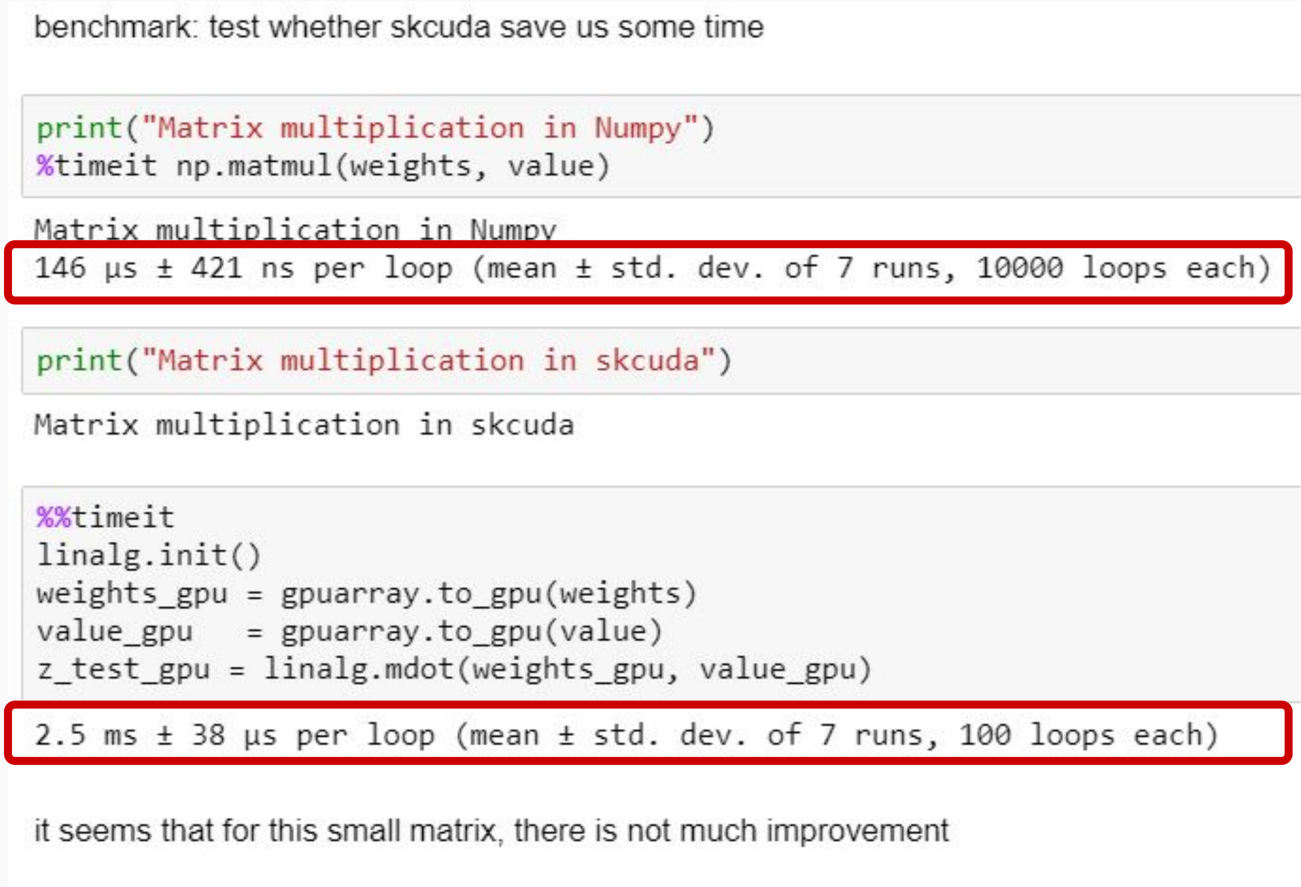
Try skcuda on small example



```
# Examples
#A_gpu = gpuarray.to_gpu(A)
#B_gpu = gpuarray.to_gpu(B)
#AB_gpu = linalg.mdot(A_gpu, B_gpu)
#np.allclose(np.matmul(A, B), AB_gpu.get())

# make sure to convert the types to np.float32
weights = weights.astype(np.float32)
value = value.astype(np.float32).reshape(N_POINTS, -1)

# Try skcuda
linalg.init()
weights_gpu = gpuarray.to_gpu(weights)
value_gpu   = gpuarray.to_gpu(value)
z_test_gpu = linalg.mdot(weights_gpu, value_gpu)

np.allclose(
    z_test,
    z_test_gpu.get().reshape(DIM_GRIDS, DIM_GRIDS))
```

```
True
```

Did not save much time in this small example



benchmark: test whether skcuda save us some time

```
print("Matrix multiplication in Numpy")
%timeit np.matmul(weights, value)
```

```
Matrix multiplication in Numpy
146 µs ± 421 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
print("Matrix multiplication in skcuda")
```

```
Matrix multiplication in skcuda
```

```
%%timeit
linalg.init()
weights_gpu = gpuarray.to_gpu(weights)
value_gpu   = gpuarray.to_gpu(value)
z_test_gpu = linalg.mdot(weights_gpu, value_gpu)
```

```
2.5 ms ± 38 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

it seems that for this small matrix, there is not much improvement

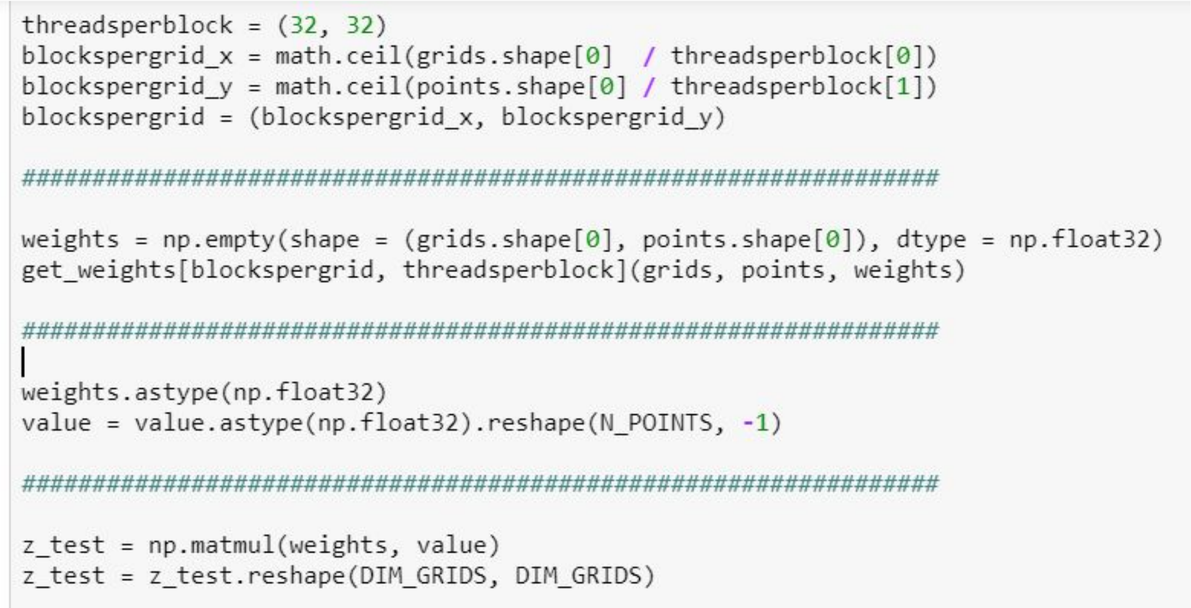
Try skcuda on larger example

N_POINTS = 10000

DIM_GRIDS = 128

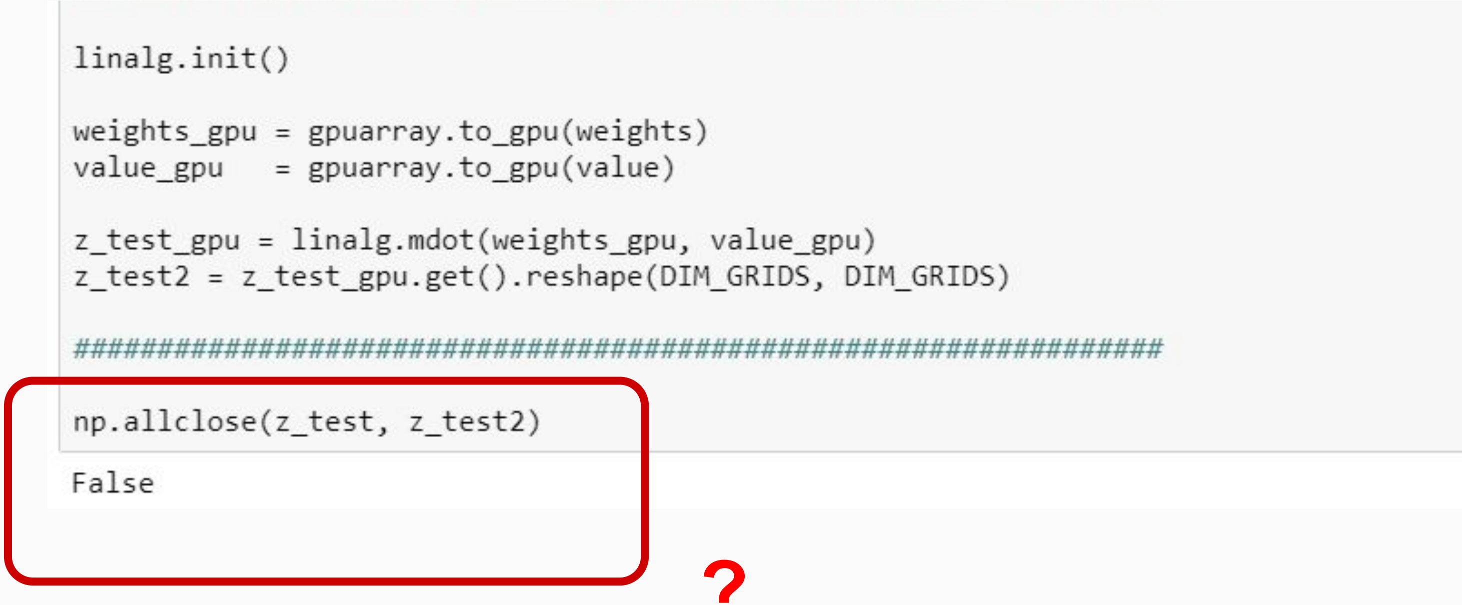
mean = ([0.0, 0.0], [4.0, 4.0])

prop = (0.5, 0.5)



```
threadsperblock = (32, 32)
blockspergrid_x = math.ceil(grids.shape[0]  / threadsperblock[0])
blockspergrid_y = math.ceil(points.shape[0] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)

#################################################################

weights = np.empty(shape = (grids.shape[0], points.shape[0]), dtype = np.float32)
get_weights[blockspergrid, threadsperblock](grids, points, weights)

#################################################################

weights.astype(np.float32)
value = value.astype(np.float32).reshape(N_POINTS, -1)

#################################################################

z_test = np.matmul(weights, value)
z_test = z_test.reshape(DIM_GRIDS, DIM_GRIDS)
```

Try skcuda on larger example

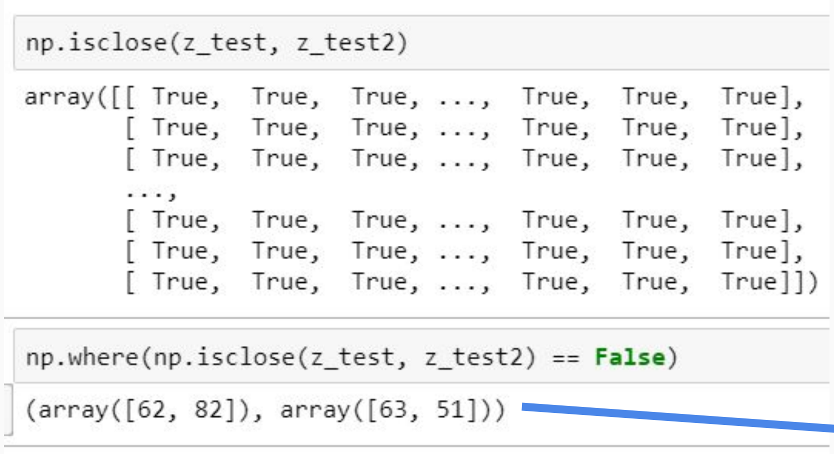


```
linalg.init()

weights_gpu = gpuarray.to_gpu(weights)
value_gpu   = gpuarray.to_gpu(value)

z_test_gpu = linalg.mdot(weights_gpu, value_gpu)
z_test2 = z_test_gpu.get().reshape(DIM_GRIDS, DIM_GRIDS)

#################################################################

np.allclose(z_test, z_test2)
```

```
False
```

?

Try to find the reason of False in the previous slides

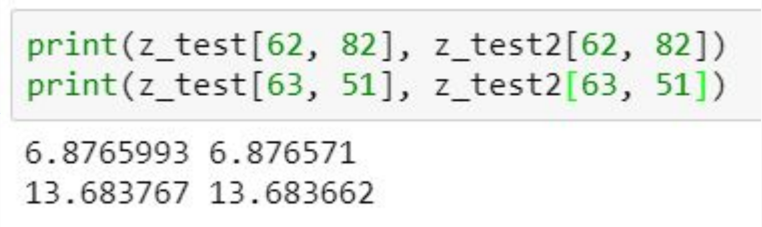


```
np.isclose(z_test, z_test2)
```

```
array([[ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       ...,
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True]])
```

```
np.where(np.isclose(z_test, z_test2) == False)
```

```
(array([62, 82]), array([63, 51]))
```

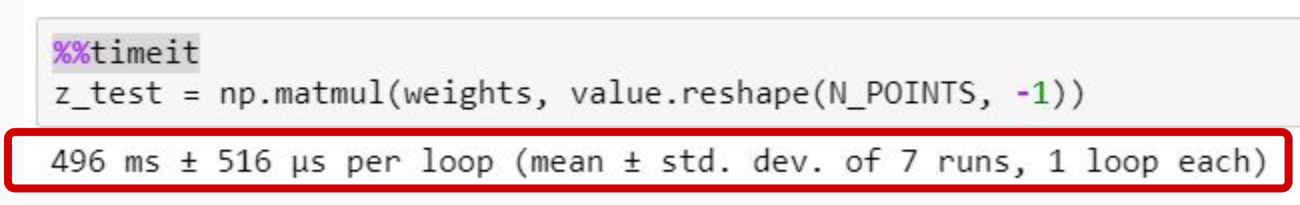


```
print(z_test[62, 82], z_test2[62, 82])
print(z_test[63, 51], z_test2[63, 51])
```

```
6.8765993 6.876571
13.683767 13.683662
```

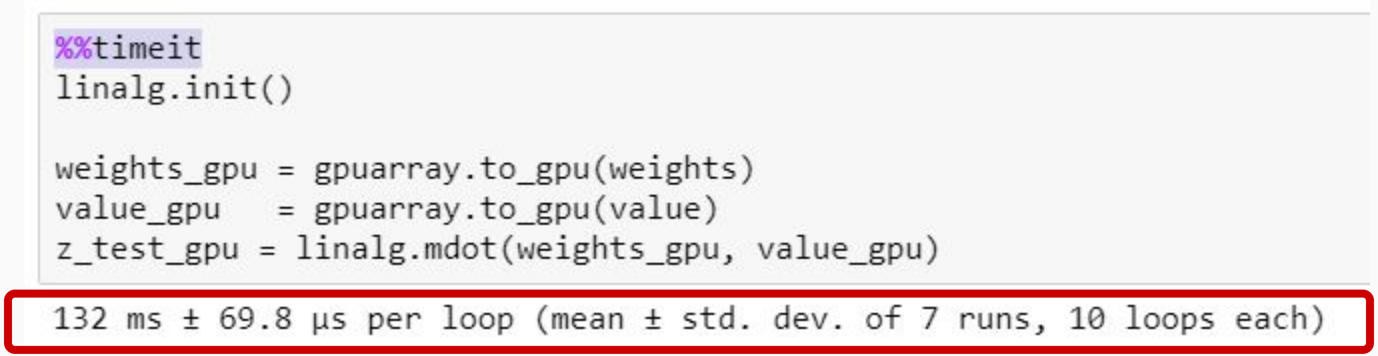
The differences between these two results are subtle.

Benchmarking performance on larger example



```
%%timeit
z_test = np.matmul(weights, value.reshape(N_POINTS, -1))
```

```
496 ms ± 516 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

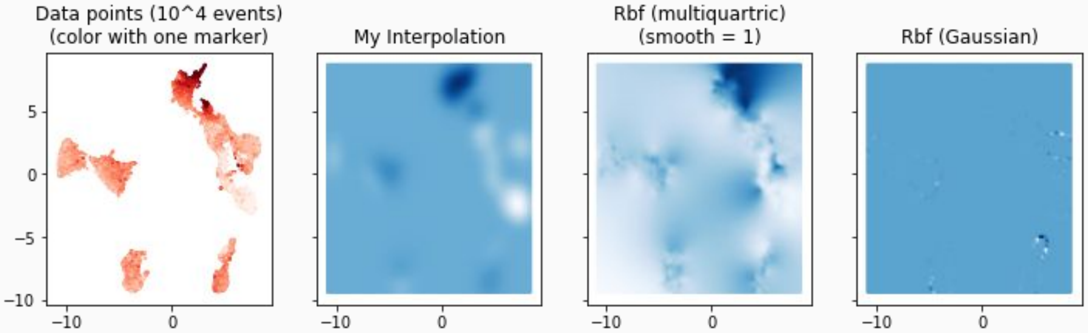


```
%%timeit
linalg.init()

weights_gpu = gpuarray.to_gpu(weights)
value_gpu   = gpuarray.to_gpu(value)
z_test_gpu = linalg.mdot(weights_gpu, value_gpu)
```

```
132 ms ± 69.8 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

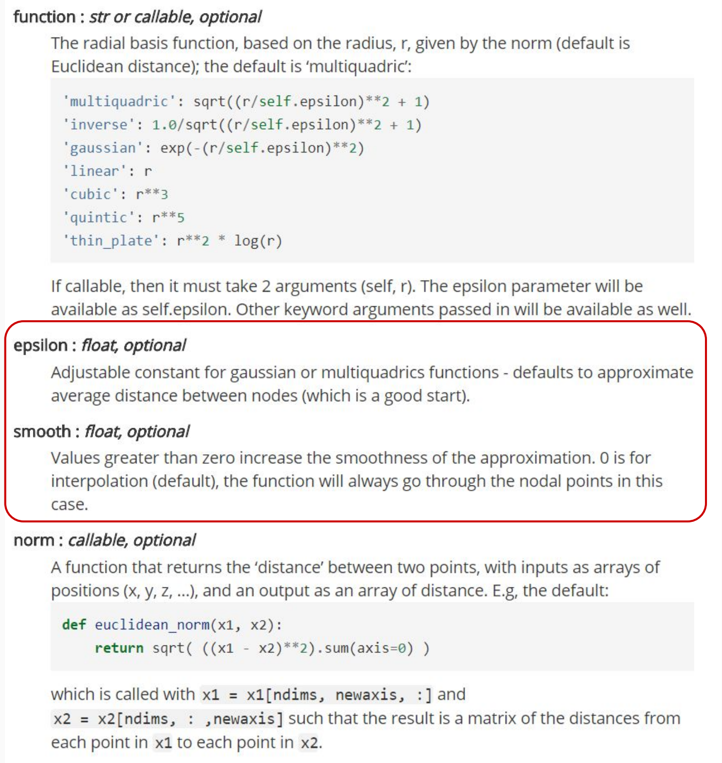
The improvement is much obvious when it comes to larger dataset.

Results of interpolation of EQAPOL data



Discussion:
Change the standard deviation of the kernel.

scipy.interpolate.Rbf



function : *str or callable, optional*

The radial basis function, based on the radius, r, given by the norm (default is Euclidean distance); the default is 'multiquadric':

```
'multiquadric': sqrt((r/self.epsilon)**2 + 1)
'inverse': 1.0/sqrt((r/self.epsilon)**2 + 1)
'gaussian': exp(-(r/self.epsilon)**2)
'linear': r
'cubic': r**3
'quintic': r**5
'thin_plate': r**2 * log(r)
```

If callable, then it must take 2 arguments (self, r). The epsilon parameter will be available as self.epsilon. Other keyword arguments passed in will be available as well.

epsilon : *float, optional*

Adjustable constant for gaussian or multiquadrics functions - defaults to approximate average distance between nodes (which is a good start).

smooth : *float, optional*

Values greater than zero increase the smoothness of the approximation. 0 is for interpolation (default), the function will always go through the nodal points in this case.

norm : *callable, optional*

A function that returns the 'distance' between two points, with inputs as arrays of positions (x, y, z, ...), and an output as an array of distance. E.g, the default:

```
def euclidean_norm(x1, x2):
    return sqrt( ((x1 - x2)**2).sum(axis=0) )
```

which is called with `x1 = x1[ndims, newaxis, :]` and `x2 = x2[ndims, : ,newaxis]` such that the result is a matrix of the distances from each point in `x1` to each point in `x2`.