PyCUDA
Important links:

- Documentation: [https://documen.tician.de/pycuda/](https://documen.tician.de/pycuda/)
- Main website: [https://mathema.tician.de/software/pycuda/](https://mathema.tician.de/software/pycuda/)
- Tutorial: [https://documen.tician.de/pycuda/tutorial.html](https://documen.tician.de/pycuda/tutorial.html)
- Github: [https://github.com/inducer/pycuda](https://github.com/inducer/pycuda)
- Installation: [https://wiki.tiker.net/PyCuda/Installation](https://wiki.tiker.net/PyCuda/Installation)
- FAQ: [https://wiki.tiker.net/PyCuda/FrequentlyAskedQuestions](https://wiki.tiker.net/PyCuda/FrequentlyAskedQuestions)
Usual Steps for Most PyCUDA Programs

- Import pycuda.driver and pycuda.autoinit. autoinit gets CUDA prepared for kernel submission and driver acts as the interface to the GPU.
- Create initial variables on the CPU if you need to.
- Allocate memory on the GPU (cuda.memalloc()).
- Copy any arrays/variables that you might have from the system to the device (cuda.memcpy_htod()).
- Create a Module from the CUDA source code (compiler.SourceModule).
- Get a reference to the kernel function using the Module (get_function("function_name")).
- Execute the function using the reference obtained in the previous step along with the required parameters.
- Copy data back from the device to the system (cuda.memcpy_dtoh()).
- Do something with the copied data if necessary.
Points to note

- **pycuda.gluarray.GPUArray** can be used to associate numpy arrays with arrays on the GPU. It even use syntax similar to numpy when dealing with the arrays.
- **GPUArray** also allows application of element wise functions as well as reductions. Both of which can be very useful to write clear code.
- **Cuda-gdb** can be used to debug PyCUDA (Ex: `$ cuda-gdb --args python -m pycuda.debug program_name.py`)
- Reachable objects are never garbage collected.
- If unreachable, objects can get released at an unspecified future time.
- Memory can (and ideally should) be freed explicitly.
Example - 1 (Matrix doubling)

import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy

a = numpy.random.randn(4,4) # Common practice to use numpy arrays
a = a.astype(numpy.float32) # Nvidia devices support only single precision
a_gpu = cuda.mem_alloc(a.size * a.dtype.itemsize) # Allocate memory on the GPU
cuda.memcpy_htod(a_gpu, a) # Transfer to the GPU
# Kernel function

mod = SourceModule(""

__global__ void doublify(float *a)
{
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
}
"

")

func = mod.get_function("doublify")  # Get a reference to the function

func(a_gpu, block=(4,4,1)) # Call the function with the previous value along with a block size of 4x4

a_doubled = numpy.empty_like(a) # Create a new empty array

cuda.memcpy_dtoh(a_doubled, a_gpu) # Transfer back from GPU

print("original array: ", a) # Display the original values

print("doubled with kernel: ", a_doubled) # Display the newly doubled values
import numpy as np
from pycuda import driver, compiler, gpuarray, tools
import pycuda.autoinit

kernel_code_template = ""
__global__ void MatrixMulKernel(float *a, float *b, float *c) {
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float Pvalue = 0;
    for (int k = 0; k < %(MATRIX_SIZE)s; ++k) {
        float Aelement = a[ty * %(MATRIX_SIZE)s + k];
float Belement = b[k * %(_MATRIX_SIZE)s + tx];

    Pvalue += Aelement * Belement;

}

c[ty * %(_MATRIX_SIZE)s + tx] = Pvalue;

""

MATRIX_SIZE = 2  # Define matrix dimensions

a_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)  # Allocate random first matrix

b_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)  # Allocate random second matrix

c_cpu = np.dot(a_cpu, b_cpu)  # Compute product using the CPU

a_gpu = gpuarray.to_gpu(a_cpu)  # Transfer to the GPU

b_gpu = gpuarray.to_gpu(b_cpu)  # Transfer to the GPU

c_gpu = gpuarray.empty((MATRIX_SIZE, MATRIX_SIZE), np.float32)  # Allocate empty GPU matrix to hold product
kernel_code = kernel_code_template % {
    'MATRIX_SIZE': MATRIX_SIZE # Pass in the matrix size as a parameter to the kernel template
}

mod = compiler.SourceModule(kernel_code)

matrixmul = mod.get_function("MatrixMulKernel") # Get reference to function

# Execute the function using the matrices and the specified block size

matrixmul(
    a_gpu, b_gpu,
    c_gpu,
    block = (MATRIX_SIZE, MATRIX_SIZE, 1),
)
print("Matrix A (GPU):")
print(a_gpu.get())

print("Matrix B (GPU):")
print(b_gpu.get())

print("Matrix C (GPU):")
print(c_gpu.get())

print("CPU-GPU difference:")
print(c_cpu - c_gpu.get()) # Confirm answer

np.allclose(c_cpu, c_gpu.get())
More examples available at: https://wiki.tiker.net/PyCuda/Examples