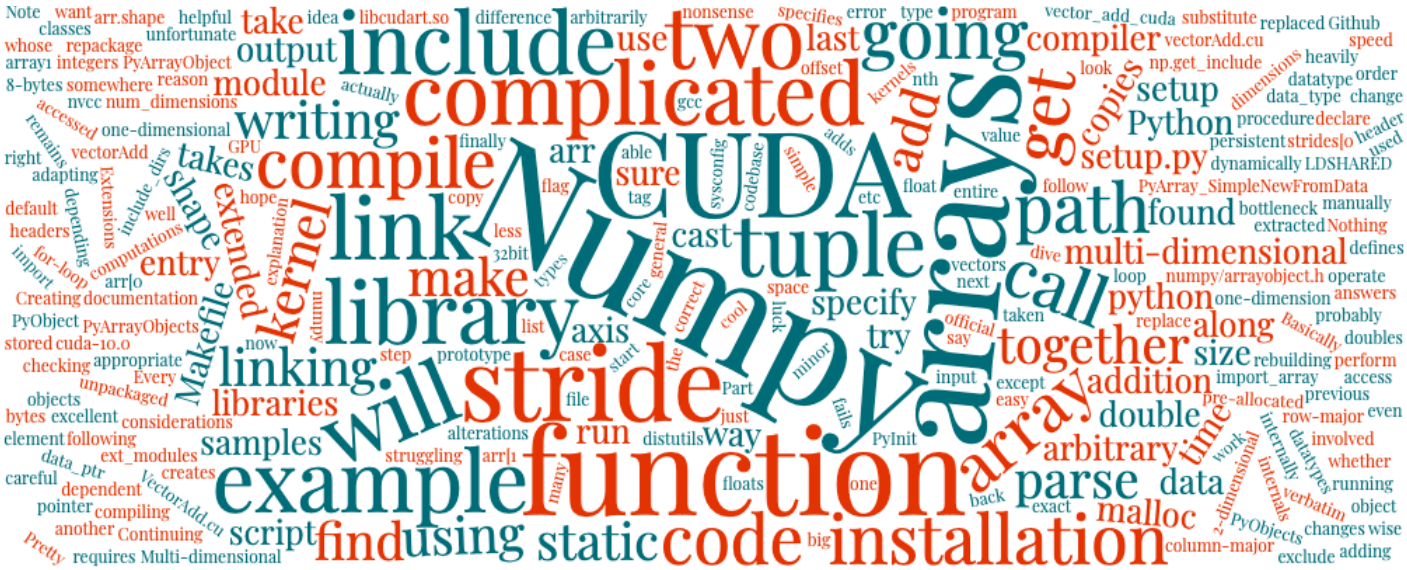
Jacob Austin
My projects and thoughts

Creating C Extensions for Python with Numpy and CUDA (Part 2)

July 4, 2019



This is Part 2 of a series on the Python C API and CUDA/Numpy integration. Part 1 can be found here. A full Github repository containing all this code can be found here.

We're going to dive right away into how to parse Numpy arrays in C and use CUDA to speed up our computations. Every Numpy object is internally a PyArrayObject, with a data pointer, a shape tuple (think arr.shape in Numpy), and a stride tuple that defines how to access the nth entry along each axis. While we don't need to get into the weeds, the stride is a tuple which specifies how many bytes you need to step along each axis to get to the next element, say from arr[0] to arr[1] or arr[:, 0] to arr[:, 1]. For a one-dimensional array, the stride will just be a one entry tuple whose size is the size of the datatype (8 in the case of doubles, 4 for 32bit floats, 4 for integers, etc). For 2-dimensional arrays, this can be more complicated, depending on whether the array is stored in row-major (the default) or column-major order. An excellent explanation can be found here!

Now what does that mean for us? Let's try writing a function to parse two Numpy float arrays and add them together (same as the addition function in Numpy):

```c
#define PY_SSIZE_T_CLEAN
#include <stdio.h>
#include <Python.h>
#include <numpy/arrayobject.h>

static PyObject* vector_add(PyObject* self, PyObject* args) {
    PyArrayObject* array1, * array2;

    if (!PyArg_ParseTuple(args, "O!O!", &PyArray_Type, &array1
        return NULL;

    if (array1 -> nd != 1 || array2 -> nd != 1 || array1->desc
        PyErr_SetString(PyExc_ValueError, "arrays must be one-
        return NULL;
    }

    int n1 = array1->dimensions[0];
    int n2 = array2->dimensions[0];

    printf("running vector_add on dim1: %d, stride1: %d, dim2:

    if (n1 != n2) {
        PyErr_SetString(PyExc_ValueError, "arrays must have th
        return NULL;
    }

    double * output = (double *) malloc(sizeof(double) * n1);

    for (int i = 0; i < n1; i++)
        output[i] = *((double *) array1 -> data + i) + *((doub

    return PyArray_SimpleNewFromData(1, PyArray_DIMS(array1),
}

static PyMethodDef methods[] = {
```

```
    {"helloworld", helloworld, METH_NOARGS, "Prints Hello Worl
    {"fib", fib, METH_VARARGS, "Computes the nth Fibonacci numb
    {"vector_add", vector_add, METH_VARARGS, "add two numpy fl
    {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC PyInit_myModule(void) {
    import_array();
    return PyModule_Create(&myModule);
}
```

To actually compile this, we're also going to have to make a few changes to our setup code. Numpy requires us to include the numpy/arrayobject.h header file, and we need to makes sure our compiler can find that path. We can manually specify include paths in setup.py, by adding an include_dirs kwarg to the ext_modules list. The include path should point to your Numpy installation, for example

```
import numpy as np
from distutils.core import setup, Extension

setup(name = 'myModule', version = '1.0',  \
   ext_modules = [
      Extension('myModule', ['myModule.c'],
      include_dirs=[np.get_include()])
```

Note that we find the Numpy include path using the `np.get_include()` function. The core code is not too complicated. We parse two `PyObjects` as `PyArrayObjects` with the *"O!"* flag, and check to make sure they are one-dimensional and have the right type. Then we malloc the output array, add the two arrays together, and repackage them. We cast them to `(double *)` because this has the correct stride of 8-bytes. We could also have cast them to `(char *)` and then used *array1 -> strides[0] * i* as our offset. The `PyArray_SimpleNewFromData` function takes *(num_dimensions, shape tuple, data_type, data_ptr)* and creates an appropriate PyObject. Now try rebuilding this and running it! *Pretty cool*. This can be extended to arbitrary dimensions and arbitrary datatypes as well. Now you can run this code from Python with:

```
import myModule
```

```
import numpy as np
myModule.vector_add(np.random.randn(1000000), np.random.randn(
```

A few words to the wise: when using the Numpy array classes, you have to call import_array() in your PyInit function. You also need to be careful about stride. It's easy to get nonsense answers if your arrays are accessed the wrong way. Multi-dimensional arrays are complicated when stride is involved.

CUDA

Now we've finally come to CUDA. CUDA can operate on the unpackaged Numpy arrays in the same way that we did with our for loop in the last example. The only difference is writing the vectorAdd kernel and linking the libraries. We are going to more or less copy the VectorAdd.cu example from the official CUDA samples, adapting it to take the vectors as input. We're also going to compile the CUDA function as a static library and link it into our module at compile time. Let's start with the CUDA kernel:

```
#include <stdio.h>
#include <cuda_runtime.h>


__global__ void vectorAdd(const double *A, const double *B, do
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements) {
        C[i] = A[i] + B[i];
    }
}

double * myVectorAdd(double * h_A, double * h_B, int numElemen
    size_t size = numElements * sizeof(double);
    printf("[Vector addition of %d elements]\n", numElements);

    // Allocate the host output vector C
    double *h_C = (double *)malloc(size);
```

```
    // Allocate the device input vectors
    double *d_A = NULL;
    cudaMalloc((void **)&d_A, size);

    double *d_B = NULL;
    cudaMalloc((void **)&d_B, size);
    double *d_C = NULL;
    cudaMalloc((void **)&d_C, size);

    printf("Copy input data from the host memory to the CUDA d
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Launch the Vector Add CUDA Kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =(numElements + threadsPerBlock - 1) / t
    printf("CUDA kernel launch with %d blocks of %d threads\n"
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_

    // Copy the device result vector in device memory to the h
    // in host memory.
    printf("Copy output data from the CUDA device to the host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device global memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return h_C;
}
```

I will exclude error checking for space considerations, but the entire codebase can be found on my Github. This is a simple program that takes two pre-allocated double arrays (extracted from our Numpy arrays), copies them to the GPU, and then adds them using a kernel and copies them back. Nothing too complicated. This is taken from the CUDA samples, so you can find more

documentation there. We copy the provided host arrays into CUDA memory using cudaMemcpy after allocating the memory with cudaMalloc. We compute the correct number of threadsPerBlock and blocksPerGrid using the `(numElements + threadsPerBlock – 1) / threadsPerBlock` formula and launch the kernel.

To incorporate this into our module, we will take the previous Numpy example verbatim, except we'll substitute our for-loop addition with a call to this function. We also have to declare the prototype somewhere. We replace the malloc call with a call to this function.

```
extern double * myVectorAdd(double * h_A, double * h_B, int nu


static PyObject* vector_add_cuda(PyObject* self, PyObject* arg
    ...
    output = myVectorAdd((double *) array1 -> data, (double *)
    ...
}

static PyMethodDef methods[] = {
    ...,
    {"vector_add_cuda", vector_add_cuda, METH_VARARGS, "add tw
    ...,
    {NULL, NULL, 0, NULL}
};
```

You probably get the idea by now. Now all that remains is to compile the vectorAdd.cu script to a static library and link it. This will be heavily dependent on your exact installation, but the following Makefile ought to work with minor alterations:

```
.PHONY: all build test clean

all: build
    CC=g++ LDSHARED='$(shell python scripts/configure.py)' pyt
    python setup.py install
    python tests/test.py

build:
```

```
    nvcc -rdc=true --compiler-options '-fPIC' -c -o temp.o vec
    nvcc -dlink --compiler-options '-fPIC' -o vectorAdd.o temp
    rm -f libvectoradd.a
    ar cru libvectoradd.a vectorAdd.o temp.o
    ranlib libvectoradd.a

test: build
    g++ tests/test.c -L. -lvectoradd -o main -L${CUDA_PATH}/li

clean:
    rm -f libvectoradd.a *.o main temp.py
    rm -rf build
```

Note that the main script calls a python script scripts/configure.py. This replaces the LDSHARED value in the Makefile under the python tag using the script

```
from distutils import sysconfig
print(sysconfig.get_config_var('LDSHARED').replace("gcc", "g++
```

Make sure this is in the appropriate location. For some unfortunate reason, gcc fails to link the library with nvcc, so we must use g++ or another C++ compiler to perform linking. What we're doing is compiling the .cu code as a static library and then linking it with the module we're writing. In your setup.py script, you'll also have to specify all the libraries you want to link. You can do that with your new setup.py file:

```
import os
from distutils.core import setup, Extension
import numpy as np

os.environ["CC"] = "g++"
os.environ["CXX"] = "g++"

if 'CUDA_PATH' in os.environ:
   CUDA_PATH = os.environ['CUDA_PATH']
else:
   print("Could not find CUDA_PATH in environment variables. D
```

```
    CUDA_PATH = "/usr/local/cuda"

if not os.path.isdir(CUDA_PATH):
    print("CUDA_PATH {} not found. Please update the CUDA_PATh
    exit(0)

setup(name = 'myModule', version = '1.0',  \
    ext_modules = [
        Extension('myModule', ['myModule.c'],
        include_dirs=[np.get_include(), os.path.join(CUDA_PATH,
        libraries=["vectoradd", "cudart"],
        library_dirs = [".", os.path.join(CUDA_PATH, "lib64")]
)])
```

Basically, we need to include the CUDA headers and link dynamically with the libcudart.so library. Here we track down all the dependencies we need and configure the script. Now you should be able to run `make`, and then follow the above example with vector_add_cuda to add two Numpy arrays together. This too can be arbitrarily extended to multi-dimensional Numpy arrays and other data types.

Now we can do all sorts of cool things:

```
>>> import myModule
>>> myModule.helloworld()
Hello World
>>> import numpy as np
>>> myModule.trace(np.random.randn(4, 4))
0.9221652081491398
>>> myModule.vector_add(np.random.randn(500), np.random.randn(
[0.01, ..., 0.54]
>>> myModule.vector_add_cpu(np.random.randn(500), np.random.ra
[0.07, ..., 0.24]
```

I hope this was helpful. There's so much more you can do, but struggling through the installation and setup procedure is the big bottleneck. Now you can look into Numpy internals, stride, multi-dimensional arrays, more complicated CUDA kernels, persistent CUDA objects, and more! Good luck!

All this code is available on a Github page here with an installation script

---
