PyCUDA

Tutorial

Introduction

As we may have already seen, NVIDIA GPUs use CUDA cores to perform calculations. In this segment, we will be using Python to run CUDA programs.

The difference between running C and Python is that Python has a lot more QOL especially in regards to data visualization and scientific computing.

Aim

We will demonstrate one of the simplest applications of parallel programming. Adding of two matrices. Note that *each* element in the matrix will be added in parallel i.e. in a single step. We will forgo the classical for loop in this case. We will start with 2 2D matrices with an arbitrary size in numpy and then will proceed to add them.

Getting started

To start our virtual workspace, we will use Google Colaboratory, address at:

https://colab.research.google.com/

Google's cloud service will provide this to us for free (note that you will require a gmail account)

Setup

1. Top left, File-> new Python 3 notebook
2. Runtime -> Hardware accelerator -> GPU
3. Setup workspace with a new code cell

Since pycuda is not a native library in colab we need an additional line before importing the libraries.

!pip install pycuda.

Run the code segment first before proceeding (at the left, a play button)

Building

Wait for a bit while pycuda is being installed. After the build finishes, we are ready to proceed.

Firstly we need to import the basic libraries.

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

Declaring

We declare our matrix as follows

Of course, we are free to declare as we wish, but this will create a standard random number matrix, repeat for a “b” matrix. Note that the cloud probably does not support double precision accuracy so to avoid difficulties, we pre-convert to float32 or single precision.

```
import numpy

m =5
n = 5

a = numpy.random.randn(m,n)
a = a.astype(numpy.float32)
```

GPU details

As we may have seen earlier, but GPU's don't support data in them, that is to say, we can't say that
> accessing GPU
> GPU variable a
> a = 3

We will need to allocate memory and then create a copy from the device and access it via pointers and the run calculations on that.

We do this as follows:

```
a_gpu = cuda.mem_alloc(a.nbytes)
b_gpu = cuda.mem_alloc(b.nbytes)
```

As the memory allocation step
Then proceed with:

```
cuda.memcpy_htod(a_gpu, a)
cuda.memcpy_htod(b_gpu, b)
```

Which essentially copies a to a_gpu

htod stands for “host to device” that is, from your system to the GPU, we will be making the reverse of this command to extract data from our GPU.

Writing the function

Interestingly enough, the function body has to be written in C

```
module = SourceModule("""
  __global__ void add(float *a,float *b)
    {
      int idx = threadIdx.x + threadIdx.y*blockDim.x;
      a[idx] = a[idx] + b[idx];
    }
  """)
```

PS : don’t forget the “;”

The equivalence in this is such that all a[idx] are being executed simultaneously.

Linearization of the 2D matrix will give us the following relation between idx and (x,y) of an index.

a	a+1	a+2	a+3	a+4
a+5	a+(5+1)	a+(5+2)	a+(5+3)	a+(5+4)
a+(5*2)	a+(5*2+1)	.	.	.
.	.	.	.	.
.	.	.	.	a+(5*4+4) = a+24

Array indexing.

threadIdx.x refers to x
threadIdx.y refers to y

blockDim.x refers to dimension in x which in our case is 5
Hence, we can create an equivalent 1D array of size 25x1 with the following index notation.

Calling the function

Have to create a separate variable to get the function as follows

```
fx = module.get_function("add")
fx(a_gpu,b_gpu, block=(m,n,1))
```

This will extract the function from "SourceModule" and execute it on the GPU copies of a and b

Extracting the result

This step is pretty straightforward. For simplicity, we will create a different matrix and copy data from device to host using

dtoh

Which stands for “device to host”

```
Add = numpy.empty_like(a)
cuda.memcpy_dtoh(Add,a_gpu)
```

Result

We should get some output in the form of matrices

As an exercise, we could check the check the result without the GPU in action as a+b

And compare the results

```
[[-0.13510826  0.4002291  -0.5162719   0.19055353  1.6327566 ]
 [-1.0615492   0.54781806  2.3408303   0.3195578   3.4254677 ]
 [-0.10167599  2.0960226  -2.045706   -1.0421004   1.9976743 ]
 [-3.7153301  -0.04654479 -0.6567255  -0.7078032   0.9564961 ]
 [-0.0357745   2.1872225  -0.00945299 -0.572941   -1.7185318 ]]
[[-0.13510826  0.4002291  -0.5162719   0.19055353  1.6327566 ]
 [-1.0615492   0.54781806  2.3408303   0.3195578   3.4254677 ]
 [-0.10167599  2.0960226  -2.045706   -1.0421004   1.9976743 ]
 [-3.7153301  -0.04654479 -0.6567255  -0.7078032   0.9564961 ]
 [-0.0357745   2.1872225  -0.00945299 -0.572941   -1.7185318 ]]
```

Advantages of parallel programming

1. Faster execution of repeated computation
2. Efficient handling of data
3. Does not need high power machines in comparison to CPU's

Note: As we may already have seen, but parallel programming needs to be applied in very specific circumstances. More specifically, it is best suited for work where a computation is not affected by its next step.

Ex. repeated derivatives as opposed to complete derivative calculation.

Thank You