TEACHING UNDERGRADUATES PYTHON AND CUDA FOR FREE USING GOOGLE COLAB AND NUMBA

Scott Cook
Tarleton State University
Box T-0470
Stephenville, TX 76402
scook@tarleton.edu

Abstract

CUDA[10] and Python[11] have risen dramatically in popularity among scientists, mathematicians, and data scientists over the past decade. GPU's and parallel algorithms are ubiquitous because they allow computations that once required days on a supercomputer days to be done in minutes a personal computer[4]. Colleges and universities must equip students for a job market that places a high value on such skills. However, teaching CUDA in the classroom can be difficult due to hardware costs and coding complexity. We present solutions to these obstacles developed for a junior/senior level scientific computing course from Spring 2019.

Background

The CPU (central processing unit) is the most common type of processor in most everyday computers. A typical CPU contains 1 to 8 cores. In contrast, a GPU (graphics processing unit) contains hundreds of cores that work simultaneously. While each GPU core is less capable than a CPU core, GPU's can greatly accelerate a large task if it can be split into simpler, independent pieces which can be assigned to the cores for parallel processing.

GPU's sales have sharply risen in recent years[1], fueled in part by their roles in machine learning and data science. As GPU's become increasingly widespread, employers are looking for candidates with skills to take advantage of them. Naturally, college and universities should respond by incorporating parallel computing into the curriculum. However, there are some specific obstacles that we address below.

Overcoming the Hardware Obstacle

One major difficulty is hardware; each student needs a GPU-equipped machine and must

be able to install specialized software to use it. Relatively few computer labs on college campuses meet this requirement.

A solution is Google Colab[5]. This free cloud-based Python computing platform allows users to attach a GPU (or TPU) and freely install packages. It requires no setup, maintenance, or installation (other than GPU specific packages) and is accessible from any internet connection. Code can be stored in the user's Google Drive account and accessed again later from any computer.

This service effectively solves the hardware obstacle with a few caveats. The performance of Colab is roughly comparable to a standard laptop. While not a good choice for large tasks, it is perfect for coursework and learning. Also, files uploaded or produced during a session must be copied into permanent storage because the compute instance "recycles" after the session.

Overcoming the Coding Obstacle

GPU's were originally developed to render graphics. However, in the late 1990's, scientists started harnessing their many independent processors to run large calculations[13]. This hack required substantial computer science knowledge and coding background.

In 2006, the NVIDIA Corporation released the CUDA platform[2] which greatly reduced the complexity of code required for parallel processing on GPU's. It was designed for use with C/C++.

Many scientists and students prefer Python to C/C++ because it is quicker to learn and has an extensive collection of packages, called the "Scientific Stack"[12], that handles many common scientific tasks with ease. However, Python is much slower than C/C++.

Recently packages like PyCUDA[7] and Numba[6], have been developed that provide the best of both. They allow CUDA kernels to be embedded and run within a larger Python program. This combines the ease and convenience of Python with the high performance of CUDA at key bottlenecks.

For classroom use, we prefer Numba to PyCUDA. In PyCUDA, GPU kernels use C/C++ syntax within a string literal. In Numba, GPU kernels use Python syntax, thus avoiding the need for students to learn *both* C/C++ and Python. We will focus on Numba here.

Thus, Google Colab and Python/Numba provide excellent solutions to the hardware and coding obstacles.

Example - Julia Sets

We now present a basic program illustrating these tools. See the live Colab notebook at bit.ly/2Y4YAaS for more advanced features, such as Colab widgets for interactive graphics

and matplotlib's FuncAnimation for .mp4 movies. The terms "device" and "host" refer respectively to the GPU and the computer it is in.

The goal is to reproduce images of Julia sets from en.wikipedia.org/wiki/Julia_set[3] using parallel processing via Colab and Python/Numba. Recall that a Julia set is obtained by repeated application of $f(z) = z^2 + c$ for a chosen value of c in the complex plane. A point is in the Julia set if it never escapes a given region. More precisely, $J(f) = \{z \in \mathbb{C} : |f^n(z)| \leq 2,\ \forall n \in \mathbb{N}\}$

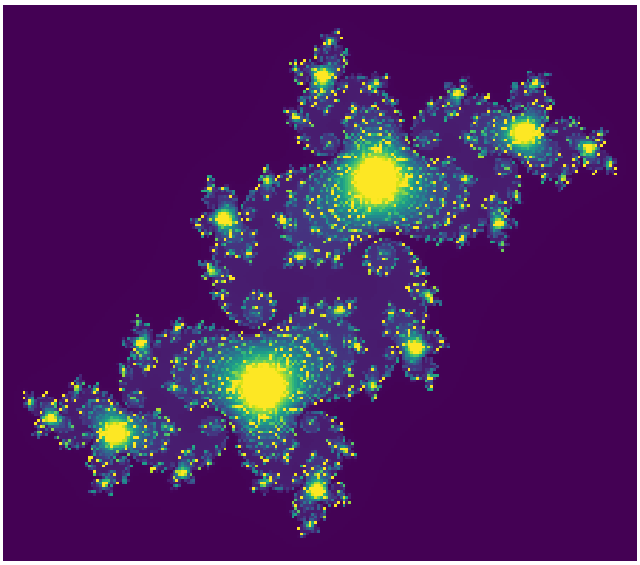

Figure 1: Julia set from example code

Configure

```
import os
os.system('pip install --upgrade numba')
os.system('apt install -y --no-install-recommends -q nvidia-cuda-toolkit')
os.environ['NUMBAPRO_LIBDEVICE'] = "/usr/lib/nvidia-cuda-toolkit/libdevice"
os.environ['NUMBAPRO_NVVM'] = "/usr/lib/x86_64-linux-gnu/libnvvm.so"

import numpy as np
import numba as nb
import numba.cuda as cuda
import matplotlib.pyplot as plt
import matplotlib.cm as cm
```

From "Runtime" choose "Change runtime type" and set Python 3 and hardware accelerator to GPU. Install and configure CUDA specific packages and import necessary packages.

Define Functions

```
1 @cuda.jit(device=True)
2 def f(x, y, c_re, c_im):
3     return x**2 - y**2 + c_re, 2*x*y + c_im
```

Lines 2-3 implement the function f as a standard Python function. Line 1 is a decorator that compiles it into a CUDA kernel; "device=True" indicates that this CUDA kernel will be called from another CUDA kernel.

Note the absence of C/C++ syntax. Numba allows the coder to use standard Python syntax and use the @cuda.jit decorator to trigger automatic conversion into CUDA.

```
1  @cuda.jit
2  def ker(x_pts, y_pts, c_re, c_im, escape_radius, escape_step, escape_step_max):
3      tx, ty = cuda.grid(2)          # What thread am I?
4      x, y = x_pts[tx], y_pts[ty]  # What x & y coords do I start at
5
6      for step in range(escape_step_max):
7          x, y = f(x, y, c_re, c_im)  # apply f
8          if x**2 + y**2 > escape_radius**2:   # Did I escape?
9              break
10     step += 1
11     escape_step[tx, ty] = step   # Record my escape_step in the correct place
```

These are the instructions each thread will follow independently. It learns its index relative to other threads. This determines the real (x) and imaginary (y) parts of the starting point that this thread will check. The loop applies f and checks for escape. The final line records the resulting escape step in the correct location in the array escape_step for output.

Note again the @cuda.jit decorator in line 1, but without the device argument. This kernel will be called from the host Python program.

Execute

```
1  def julia(c_re=-0.1, c_im=0.65, escape_radius=2, escape_step_max=1000,
2            escape_step_min=5, x_range=[-2, 2], y_range=[-2, 2],
3            resolution=5, plot=True, cmap=cm.viridis, trim=True):
4
5      # GPU's have a large number of "threads" which are flexibly arranged into blocks,
       which are then arranged into a grid.
6
7      # Create square blocks
8      max_threads_per_block = cuda.get_current_device().MAX_THREADS_PER_BLOCK
9      block_len = int(np.sqrt(max_threads_per_block))
10     block_shape = (block_len, block_len)
11
12     # Create a square grid of 2^grid_len blocks
13     grid_shape = (2**resolution, 2**resolution)
14
15     # Create starting coordinates
16     shape = (grid_shape[0]*block_shape[0], grid_shape[1]*block_shape[1])
17     x_pts = np.linspace(*x_range, shape[0])
18     y_pts = np.linspace(*y_range, shape[1])
19
20     # Copy from CPU to GPU - equivalent of memcopy in cuda
21     x_pts_gpu = cuda.to_device(x_pts)
22     y_pts_gpu = cuda.to_device(y_pts)
23     escape_step_gpu = cuda.device_array(shape, dtype=np.uint16)
24
25     ####### Key line here - Call the GPU kernel from the host python code
```

```
26      ker[grid_shape, block_shape](x_pts_gpu, y_pts_gpu, c_re, c_im, escape_radius,
        escape_step_gpu, escape_step_max)
27
28      # Copy output from GPU to CPU
29      escape_step = escape_step_gpu.copy_to_host()
30
31      ####### The remaining code plots the image nicely
32      # Trim uninteresting parts from image
33      aspect = 1.0
34      if trim:
35          trim_rows = (np.max(escape_step, axis=1) < escape_step_min)
36          trim_cols = (np.max(escape_step, axis=0) < escape_step_min)
37          escape_step = escape_step[~trim_rows, :]
38          escape_step = escape_step[:, ~trim_cols]
39
40          aspect = escape_step.shape[1] / escape_step.shape[0]
41          if aspect < 1:  # transpose so width > height
42              escape_step = escape_step.T
43              aspect /= 1
44      escape_time = escape_step / escape_step.max()
45
46      if plot:
47          width = 4
48          fig, ax = plt.subplots(figsize=(width, width*aspect))
49          plt.axis('off')
50          ax.imshow(escape_time, interpolation='nearest', cmap=cmap)
51          plt.show()
52
53      return shape, escape_time, escape_step
54
55  out = julia()
```

All but the last line are within the function "julia" defines in lines 1-3. It has several arguments, all with default values.

- Lines 8-10 - In CUDA, threads are arranged into blocks, which are then arranged into a grid. We must specify the block and grid shapes. These lines get the hardware limit for threads per block and sets block_shape to the largest possible square. Grid_shape is specified by the function argument "resolution". Higher resolution uses more threads so that starting points are more closely spaced, giving a higher quality image which can take longer to run.
- Lines 16-18 - Create the grid of starting points.
- Lines 21-23 - Create arrays on the GPU. The arrays x_pts and y_pts live on the host; they must be explicitly copied to the GPU. Line 23 creates a new array on the GPU where the results will be recorded.
- Line 26 - Executes the GPU program. The normal Python function values are passed inside $(\ldots)$. The block and grid shapes are passed inside $[\ldots]$; this is the only piece that differs from standard Python syntax.
- Lines 33-43 - Optional code to trim the image. There can be entire rows or columns near the edges that escape very quickly and look like background which can be trimmed for a better image.
- Line 44 - escape_time is an array of floats in $[0, 1]$ to determine pixel color below.

- Lines 46-51 - Plot the Julia set. Lines 47-49 set image attributes and 51 makes it appear. The key line is 50. It uses matplotlib's[9] imshow to create a "heat map" where color is determined by escape_time. Interpolation smooths the image while cmap specifies the color scheme[8].

Conclusion

Teaching CUDA and Python in college classrooms can be challenging due to hardware obstacles and coding complexity. We showed how Google's Colab service effectively solves the hardware issues and the Numba package for Python substantially reduces the coding challenges. We hope the simple example program presents is a valuable starting points for other instructors aiming to corporate parallel processing into their curriculum. See bit.ly/2Y4YAaS for a live working copy with more advanced features.

Thanks to Bryant Wyatt (Tarleton State University) and André Caldas de Souza (University of Brasilia) for their input and for creating opportunities to hone these ideas.

Bibliography

[1] Ayushi Bajpai and Komal Sharma, *Graphic processing unit (gpu) market to reach* $157.1 billion, globally by 2022$, 2016. https://www.alliedmarketresearch.com/press-release/graphic-processing-unit-market.html.

[2] Wikimedia Foundation, *Cuda*. https://en.wikipedia.org/wiki/CUDA.

[3] _____, *Julia set*. en.wikipedia.org/wiki/Julia_set.

[4] Frederic Friedel, *Cray-1 - the eight million dollar super-computer*, 2016. https://medium.com/@frederic_38110/cray-1-the-eight-million-dollar-super-computer-f020a2ac9257l.

[5] Google, *Welcome to colaboratory*. https://colab.research.google.com/notebooks/welcome.ipynb.

[6] Anaconda Inc., *Numba*. http://numba.pydata.org/.

[7] Andreas Kloeckner, *Pycuda*. https://documen.tician.de/pycuda/.

[8] Matplotlib.org, *Colormaps*. matplotlib.org/3.1.0/tutorials/colors/colormaps.html.

[9] _____, *Matplotlib*. https://matplotlib.org/.

[10] Timothy Prickett Morgan, *Talking up the expanding markets for gpu compute*, 2018. https://www.nextplatform.com/2018/04/10/the-more-you-buy-the-more-you-save/.

[11] David Ramel, *Popularity index: Python is 2018 'language of the year'*, 2018. https://adtmag.com/articles/2019/01/08/tiobe-jan-2019.aspx.

[12] SciPy.org, *Python scientific stack*. https://www.scipy.org/index.html.

[13] Lauren Wolf, *The gpu revolution*, 2010. https://pubsapp.acs.org/cen/science/88/8844sci1.html.