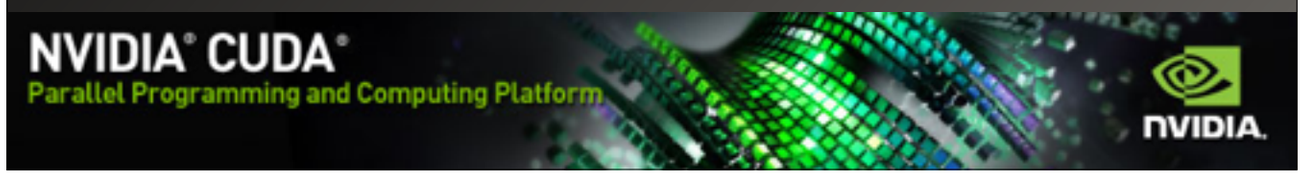
Jacob Knoles and Dr. Eric Ayars
California State University Chico

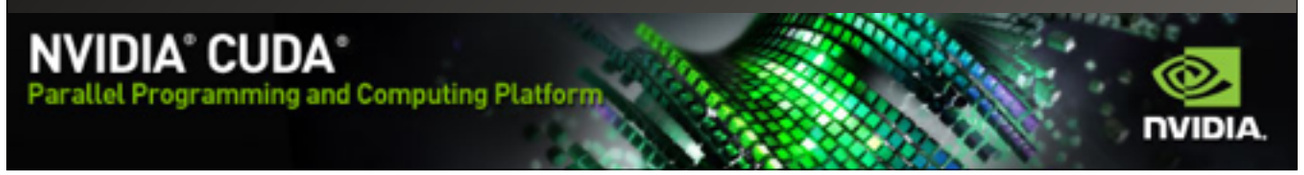
NVIDIA CUDA

At Home Supercomputing



Welcome, today we will be taking a brief look into the world of parallel programming and using the GPU to accelerate your programs.

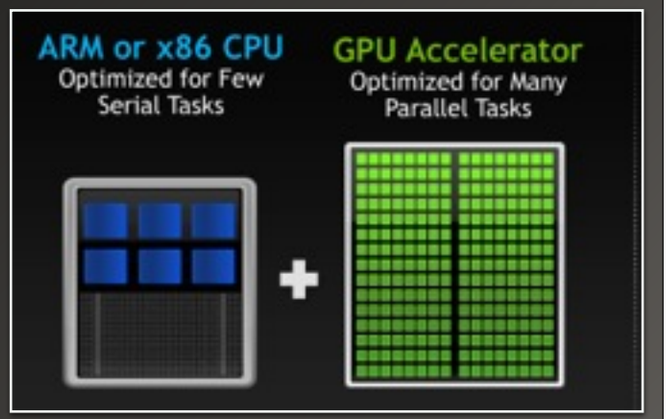
What is CUDA?

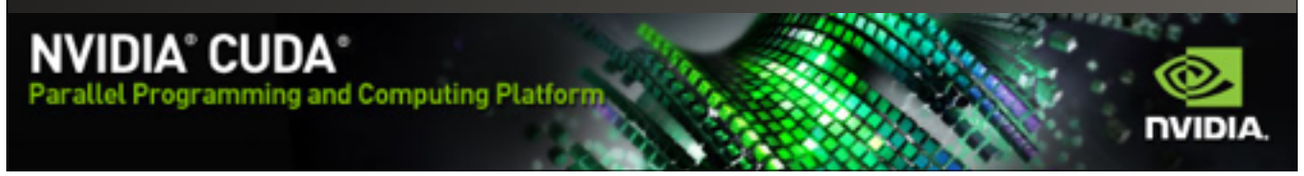


The graphics company NVIDIA, revolutionized the computational world with the development of a familiar, easy to use method for programming a GPU known as CUDA. Using an Objective-C style of code, along with a readily available API and backward compatibility with hundreds of devices it is relatively easy to learn and implement.

What Does CUDA Give Us?

- Massively parallel processing capabilities
  - Modern GPUs contain upwards of 3,000 CUDA cores
  - A modern CPU averages 4 to 8 cores
- Ability to simulate and render simultaneously
- Supercomputing at a fraction of the cost

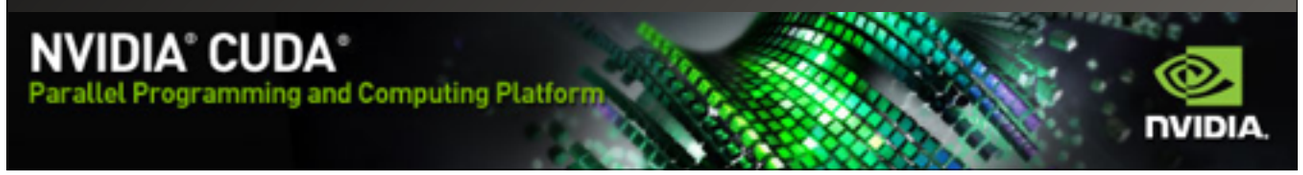




So why do we want to learn CUDA? While it's true that todays processors are incredibly powerful, we are in fact hitting a concrete celling in terms of hardware capabilities. The CPU clock speed has been hovering right around 3GHz and we don't really see more than eight processing cores. GPUs however are not required to perform the same complex tasks as the CPU and therefore opt for simpler processors, but a lot more of them. Also, thanks to the local real estate you can find GPUs with up to 5,760 cores just waiting to be used. And there is so much more headroom for GPU architecture that the innovation simply keeps on rolling.

CUDA and Python

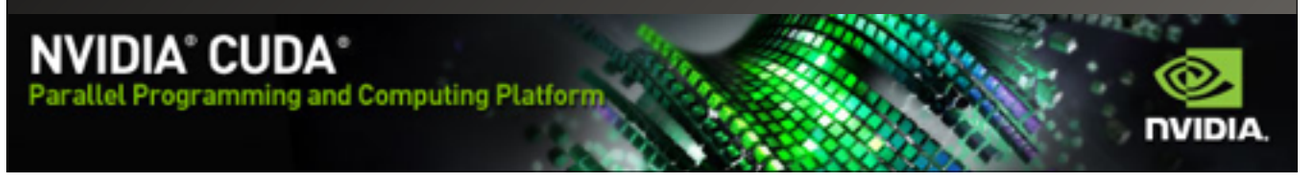
- CUDA alone is an objective C language
- Several wrappers and compilers have been written to add CUDA acceleration to other languages
- PyCUDA and Anaconda Accelerate (NumbaPro) are two for Python
  - PyCUDA is free and open source
  - Accelerate is proprietary but free for academics
    - Provided by Continuum Analytics



Alright, now to the good stuff; using this power, and now even easier thanks to Python. There are two known Python APIs that make CUDA available, the open source PyCUDA, and the proprietary Anaconda Accelerate. Accelerate is a part of NumbaPro and provided by Continuum Analytics free of charge for valid .edu emails. Overall Accelerate is a little easier to work with, well supported and is the API that I will be using.

So how can I use it?

Example: Relaxation

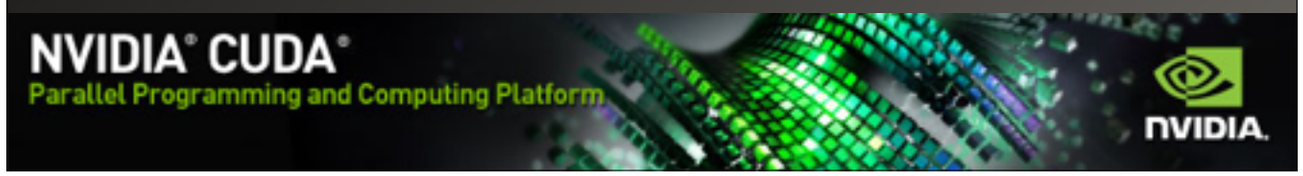


Accelerate was designed to quickly and efficiently accelerate Python code with little to no effort. Let's explore this by looking at a brief example of a simple relaxation algorithm.
(Walk through code)
Now we can parallelize this with the addition of two lines of code (CLICK). By placing the decorator in front of a function that can be parallelized the compiler will do its best to parallelize the code. Be mindful though that autojit by default still uses the CPU, albeit in parallel, you can add an option to target compatible GPUs using autojit(target='gpu') (CLICK)

So how can I use it?

Example: Relaxation

```
def relaxation(old,new):
  while delta > 1e-6:
    delta = 0.0
    steps += 1

    for i in range(1, N-1):
      for j in range(1, N-1):

        if changing[i,j]:
          delta=(old[i+1,j] + old[i-1,j] + old[i,j+1] + old[i,j-1])*0.25 - old[i,j]

        else:
            delta = 0.0

        new[i,j] = old[i,j] + delta
    old = new[:,:]
```

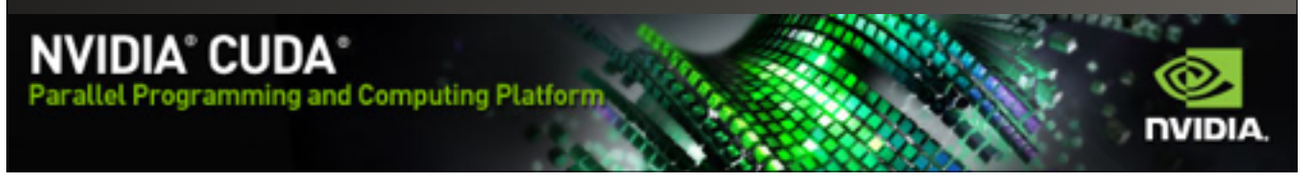


Accelerate was designed to quickly and efficiently accelerate Python code with little to no effort. Let's explore this by looking at a brief example of a simple relaxation algorithm.
(Walk through code)
Now we can parallelize this with the addition of two lines of code (CLICK). By placing the decorator in front of a function that can be parallelized the compiler will do its best to parallelize the code. Be mindful though that autojit by default still uses the CPU, albeit in parallel, you can add an option to target compatible GPUs using autojit(target='gpu') (CLICK)

So how can I use it?

Example: Relaxation

```
from numbapro import cuda
. . .
@cuda.autojit()
def relaxation(old,new):
  while delta > 1e-6:
    delta = 0.0
    steps += 1

    for i in range(1, N-1):
      for j in range(1, N-1):

        if changing[i,j]:
          delta=(old[i+1,j] + old[i-1,j] + old[i,j+1] + old[i,j-1])*0.25 - old[i,j]

        else:
            delta = 0.0

        new[i,j] = old[i,j] + delta
    old = new[:,:]
```

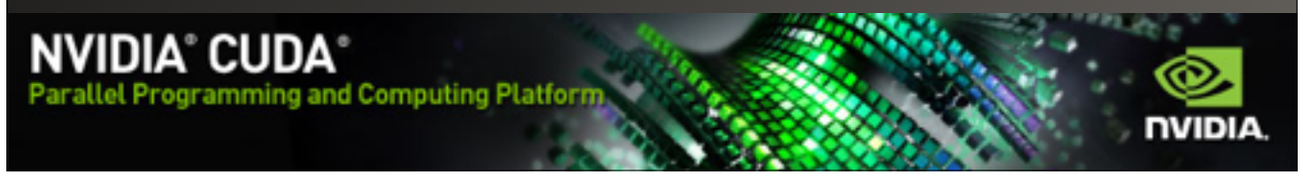


Accelerate was designed to quickly and efficiently accelerate Python code with little to no effort. Let's explore this by looking at a brief example of a simple relaxation algorithm.

(Walk through code)

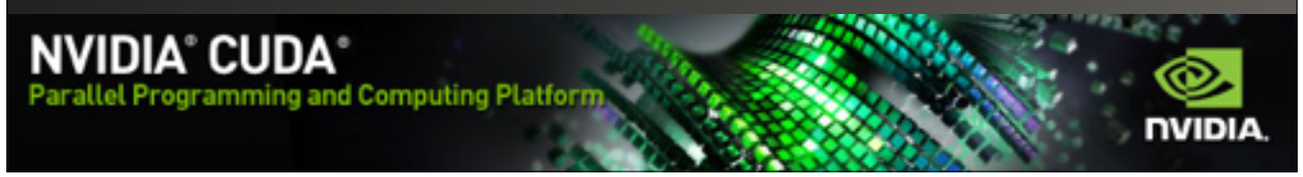
Now we can parallelize this with the addition of two lines of code (CLICK). By placing the decorator in front of a function that can be parallelized the compiler will do its best to parallelize the code. Be mindful though that autojit by default still uses the CPU, albeit in parallel, you can add an option to target compatible GPUs using autojit(target='gpu') (CLICK)

Now lets CUDA it



Now autojit works well for simple algorithms, If you are making something more complex or just don’t trust the compiler then you can parallelize the code yourself using the cuda.jit() decorator. When using this method the code gets a lot more detailed but you have a much finer control. Take a look at this relaxation core.
(Walk through code)

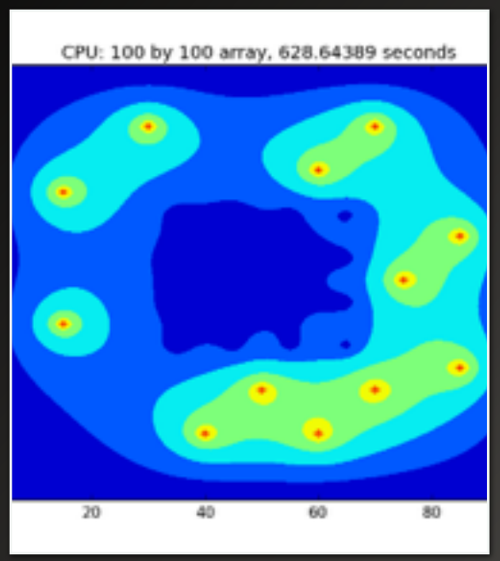
```
import numbapro
...
@cuda.jit(f8[:,:],f8[:,:])
def cuda_relaxation(d_old, d_new):

  tx = cuda.threadIdx.x
  ty = cuda.threadIdx.y
  bx = cuda.blockIdx.x
  by = cuda.blockIdx.y

  i,j = cuda.grid(2)

  while change > 1.0e-6:
    if i >= 1 and i <= N-1 and j >= 1 and j <= N-1:
     if changing[i,j]:
      delta = (d_old[i-1,j] + d_old[j,j-1] + d_old[i,j+1] + d_old[i+1,j])*0.25 - d_old[i,j]
     else:
       delta = 0.0
     d_new[i,j] = d_old[i,j] + delta
     change = change + abs(delta)
    d_old = d_new[:,:]
...
####### Main Code #######
blockdim = (32,32)                              # Establish the dimension of each block
griddim = (N/blockdim[0], N/blockdim[1])        # Establish the grid dimension

cuda_relaxation[griddim, blockdim](d_old, d_new)        # Kernel launch
```

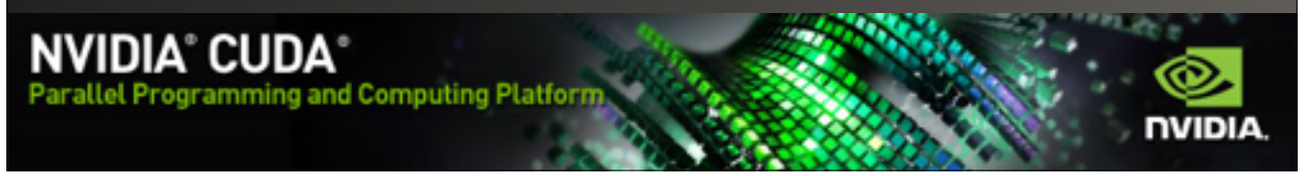


Now autojit works well for simple algorithms, If you are making something more complex or just don’t trust the compiler then you can parallelize the code yourself using the cuda.jit() decorator. When using this method the code gets a lot more detailed but you have a much finer control. Take a look at this relaxation core.

(Walk through code)

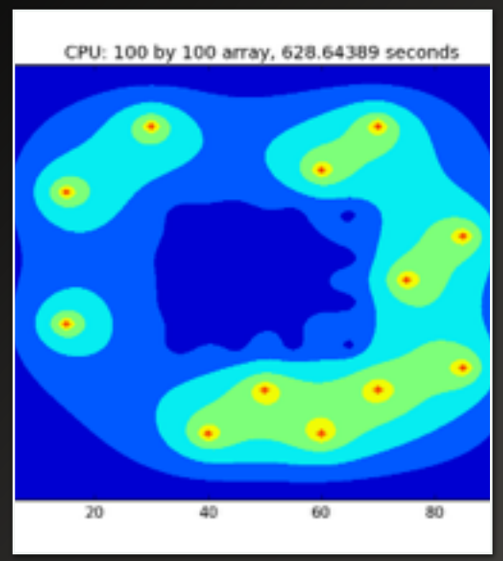
Example Results

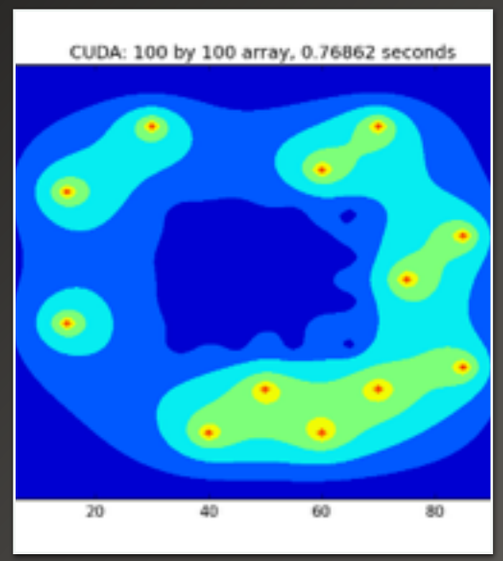


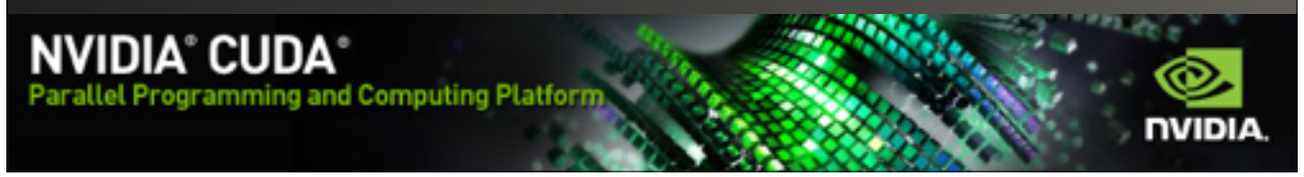


Here are some example results of a 100x100 relaxation. As you can see the CPU gets the job done in about 630 seconds or 10.5 minutes. When we run the algorithm through CUDA (CLICK)we get the same results, to the same accuracy, in 0.76 seconds. A speed-up factor of over 800.

818 Times Faster





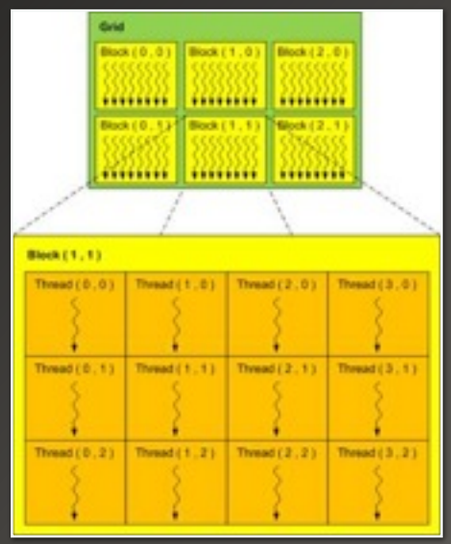


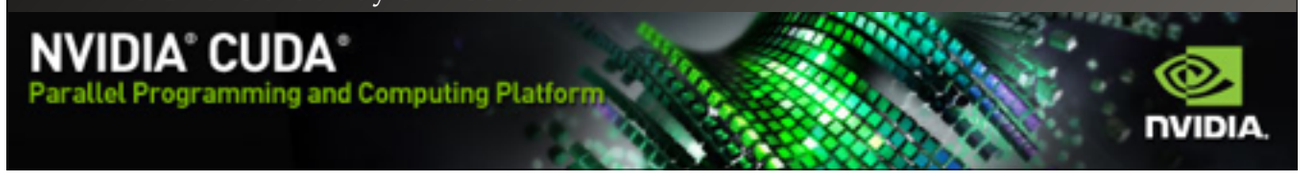
Here are some example results of a 100x100 relaxation. As you can see the CPU gets the job done in about 630 seconds or 10.5 minutes. When we run the algorithm through CUDA (CLICK)we get the same results, to the same accuracy, in 0.76 seconds. A speed-up factor of over 800.

Threads, Blocks and Grids

For more in depth control we have to cover the basics.

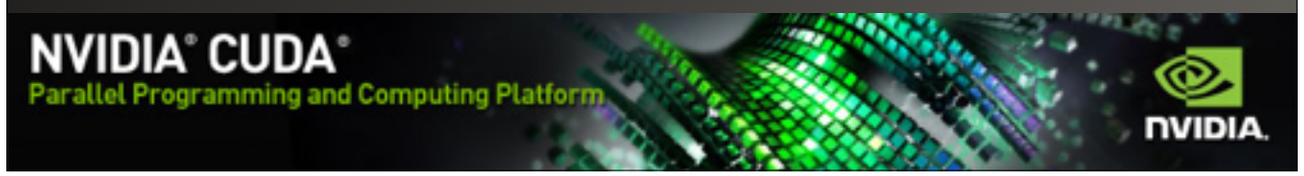
- Programs run on Threads
  - CPUs can handle up to 16
  - GPUs have thousands
- 32 Threads make up a Block
- The number of Blocks depends on your device hardware
- The Grid is the array of Blocks





Now we need to address the elephant from two slides ago, the chunk of code involving threadIdx, this turns out to be very important for GPU programming as the GPU is launching thousands of threads all at once. These commands allow us to access and track each individual thread throughout the program, but how? Think of the GPU as an array, or a grid, each space of this grid is called a block, and in each of these blocks there is another array of 32 threads. So to address the individual thread you need the grid address, the block address, and the thread address each time, or simply call threadIdx which does the hard work for you. It is very important to remember this structure when designing CUDA code.

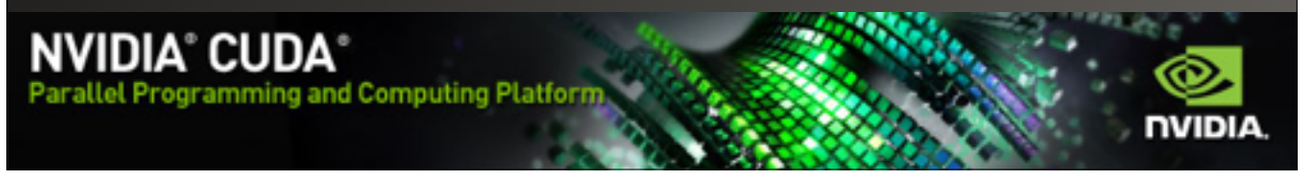
Advanced Commands: NumbaPro

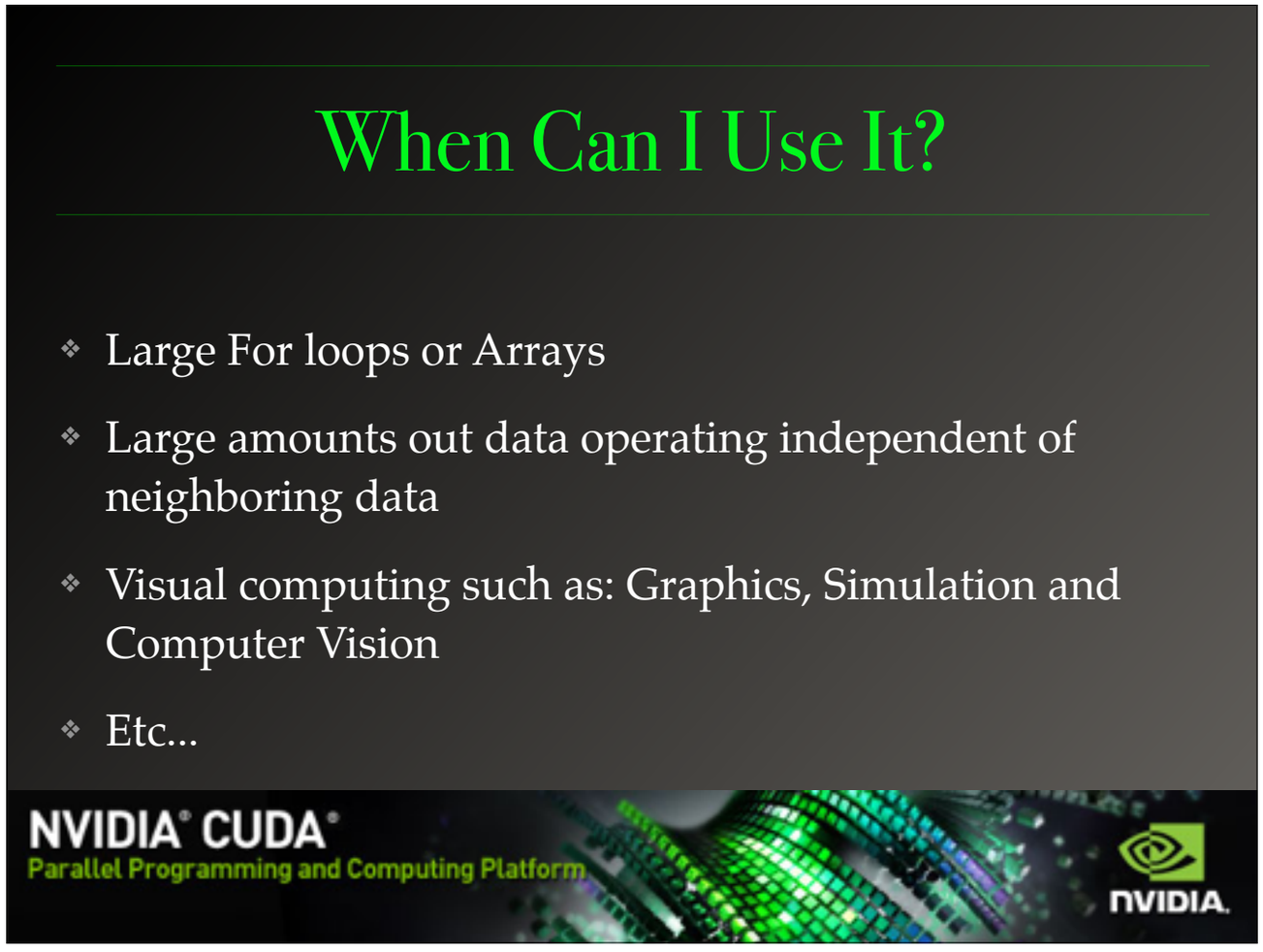


Advanced Commands: NumbaPro

- cuda.jit(*args,**kws)
- cuda.threadIdx._
- cuda.grid(#)
- cuda.to_device(*args,**kws)
- cuda.copy_to_host(*args,**kws)
- cuda.stream(*args,**kws)
- cuda.syncthreads()

- Decorator identifying cuda kernal
- Retrieves the ID of any thread (x,y,z)
- Shortcut to compute thread IDs
- Copy data to the GPUs memory
- Copy data back to the CPU
- Generate a list of commands to execute
- Set a “rally point” for threads

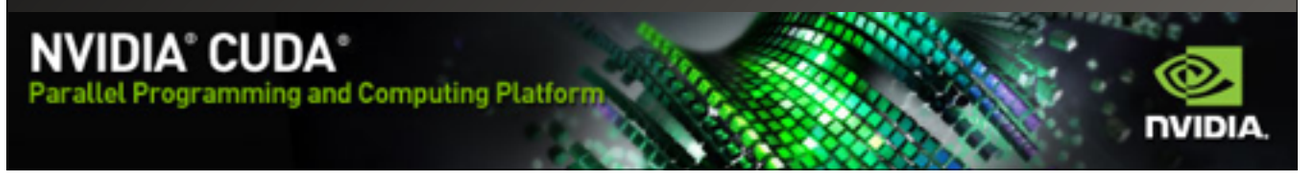




There are a couple of key indicators that let you know when it might be appropriate to use CUDA, For loops and arrays for instance. Anytime you have large amounts of data that doesn't strictly depend on itself or its neighbors, and of course visual computing.

Where to Get More

- https://www.nvidia.com/cuda
- https://store.continuum.io/cshop/accelerate/
- http://docs.continuum.io/accelerate/index.html
- http://physics.csuchico.edu/~eayars/code
  - cpuRelax.py and cudaRelax.py



More information can be found at NVIDIAs CUDA page, as well as Continuum Analytics site for information on NumbaPro and Accelerate. Udacity offers a free online class on parallel programming using the standard C language and you can find this slideshow and my relaxation code on the CSU Chico Physics site.