Undergraduate GPU-enabled Research Through Python

Garrett Allen, Lindsay Blake, Matt Bellis (mbellis@siena.edu)
Department of Physics and Astronomy, *Siena College*, *Loudonville, NY*

SIENA*college* Physics and Astronomy

Exposing undergraduates to GPU concepts

This poster details our experiences with exposing undergraduate students to parallel processing concepts, specifically on the GPU, using a Python interface, bypassing the usual learning path that makes use of the CUDA C-libraries.

Parallel processing general and GPU programming in particular has the potential to transform insurmountable challenges into tractable problems.

It is therefore imperative that we introduce our students to these concepts early in their career so that they can become comfortable with (and eventually proficient in) these frameworks.

At Siena College, a small liberal-arts college in upstate-NY, most Computer Science and other Science majors will not take a C/C++ course, but they *are* exposed to Python. In the summer of 2014, we explored the idea of introducing CUDA programming through a Python wrapper and we settled upon the Continuum Analytics ***numba*** library.

Continuum Analytics numba.cuda library

Continuum Analytics (CA) distributes a consistent installation of Python and many useful libraries. In addition to this bundling, they also produce their own contributions to the HPC-Python ecosystem such as numba, a potential replacement for numpy. http://continuum.io/

CONTINUUM ANALYTICS

CA also provides a Python interface to CUDA through their ***numba*** and ***numbapro*** libraries, both distributed through their Anaconda packaging tool. We used the ***numba.cuda*** library as it allowed us to most closely mimic what one would do in C.

For example, a simple C-CUDA kernel and the call might resemble the following code.

```
__global__ void kernel (int *a)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = idx;
}

int main()
{
    .
    .
    kernel<<<grid,block>>>(d_a);
    .
    .
}
```

While the same kernel would look like the following in ***numba.cuda***.

```
from numba import cuda

@numba.cuda.jit("void(float32[:])")
def kernel(arr_a):
    idx = cuda.blockIdx.x*cuda.blockDim.x + cuda.threadIdx.x
    arr_a[idx] = idx
.
.
kernel[block_ct,thread_ct](a)
```

Exploring nearest-neighbor fitting bias

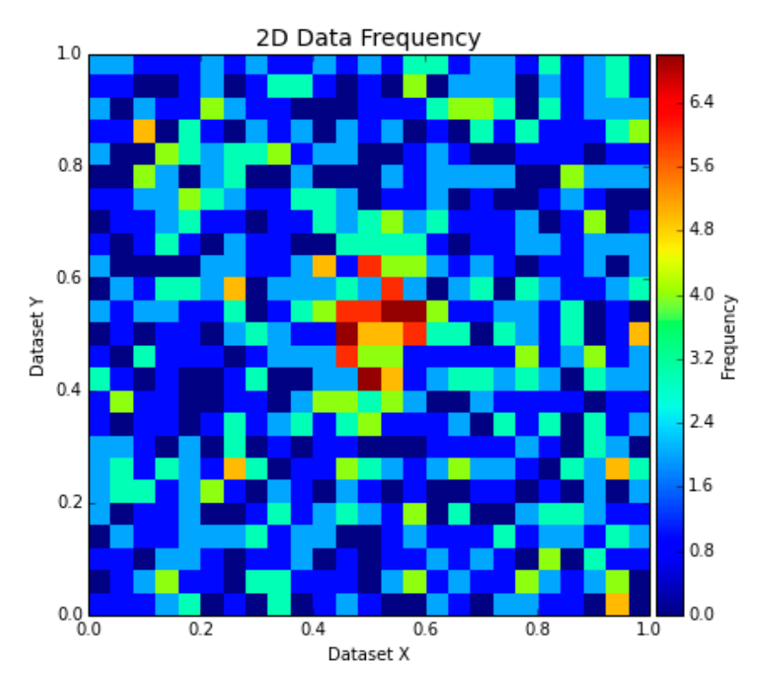
In particle physics (the primary research focus of our group), one is often looking for a ``bump" or ``peak" on top of some background, which is how new particles can be discovered! The traditional approach uses the Maximum Likelihood Method (MLM) in which the analyst tests different hypotheses of the shapes, or Probability Distribution Functions (PDFs), for signal and background, and the relative fraction of each. If the data are multidimensional, correlations between the PDFs must be taken into account, though this may not be known a priori.

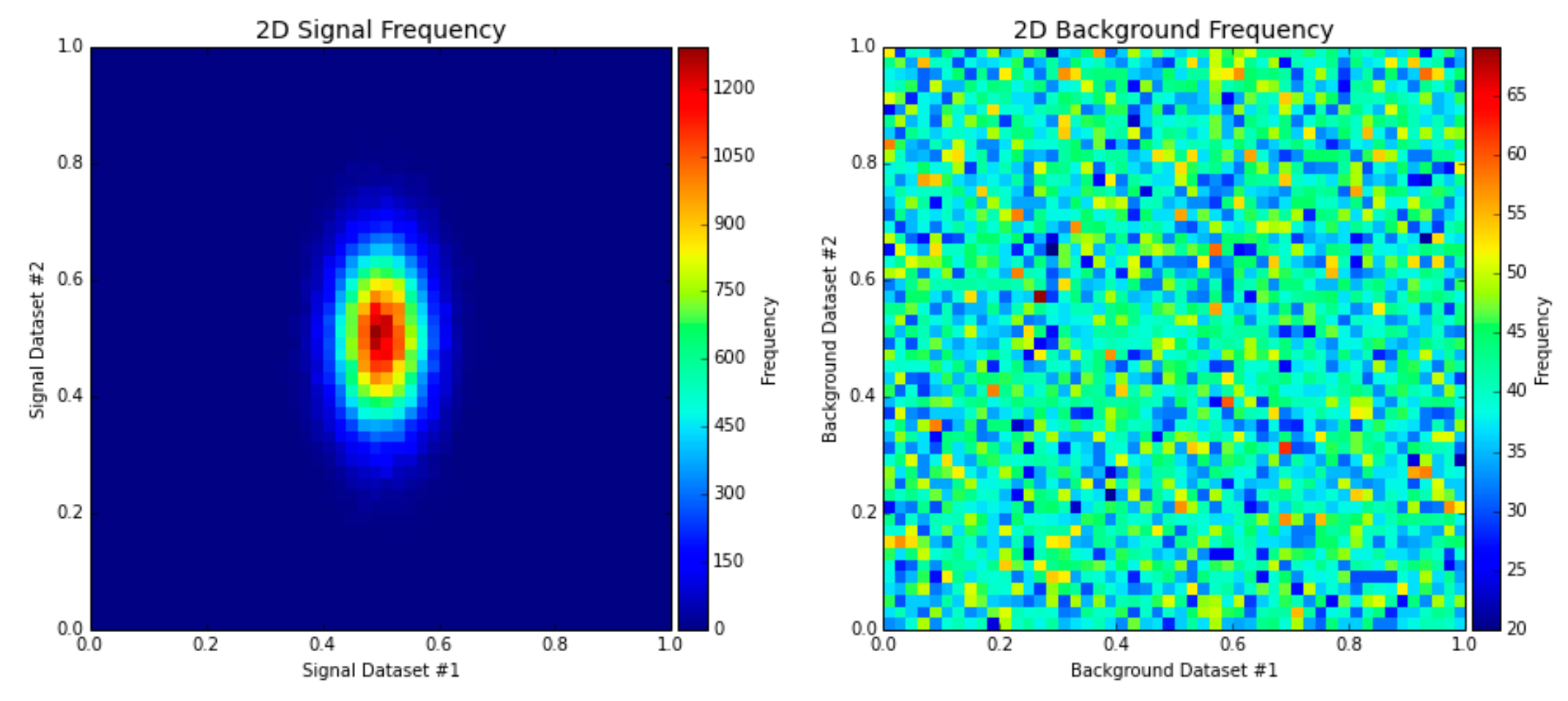
In many particle physics analysis, there may not be an analytic function for the PDF and other approximation methods must be used. These generally rely on templates generated through Monte Carlo techniques and simulations of the detector response.

Lindsay Blake, a rising-junior Physics major explored using the density of ``nearest-neighbors" as a stand-in for a PDF, and using the MLM to determine the relative fraction of signal and background. The templates are generated from some known distribution for this simple test case, though this may be different in a real experiment. *The challenge for this type of study is calculating the density of nearest neighbors, as this is computationally intensive.*

An outline of the procedure follows.

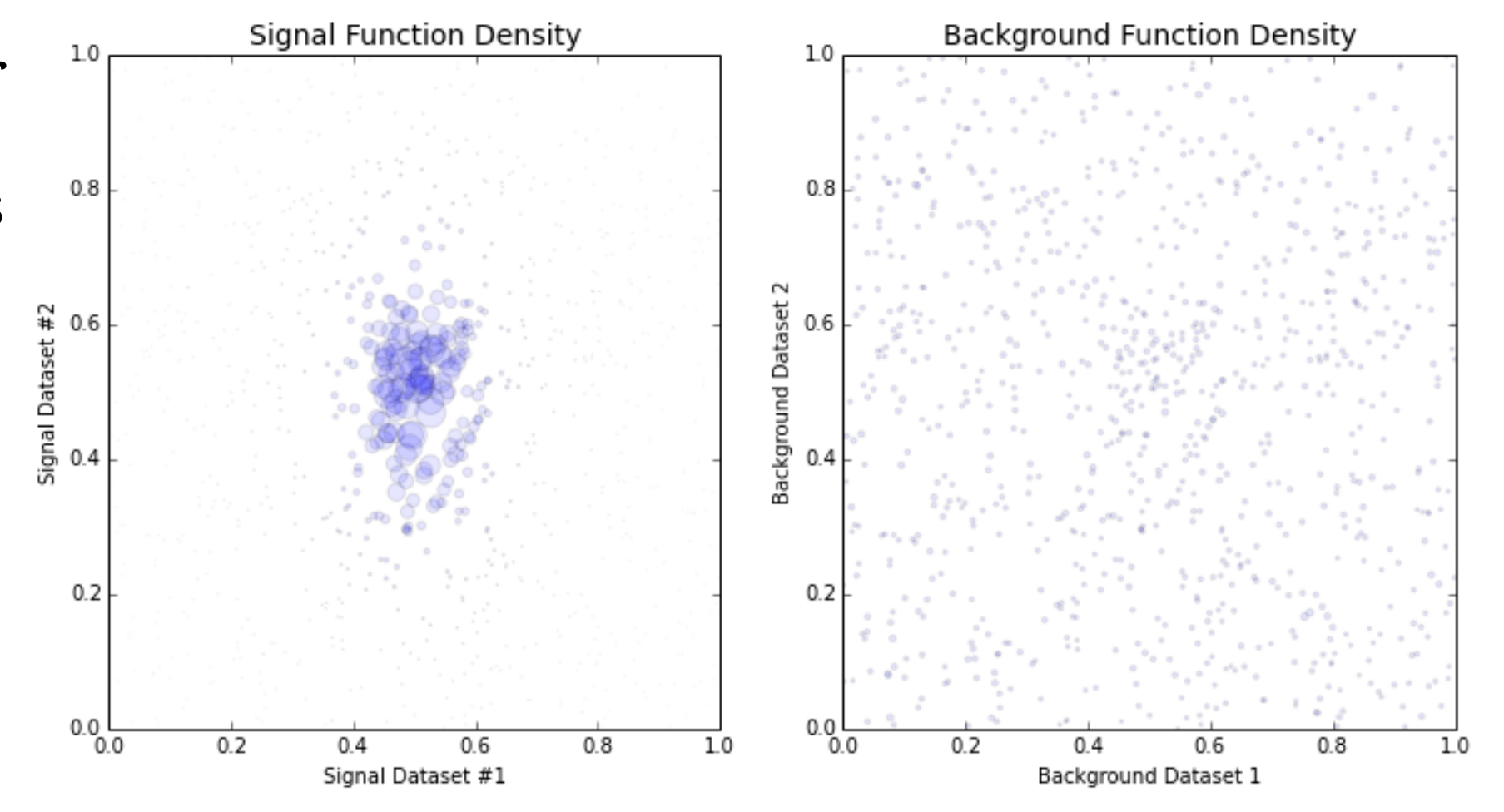
We start out with a mock dataset representing the data that come out of some experiment, and these data are parametrized by two variables, *X* and *Y*. The density in the center represents our peak signal on top of a flat background. For our preminary tests, we worked with ~100 signal events and ~900 background events (1000 total data points).

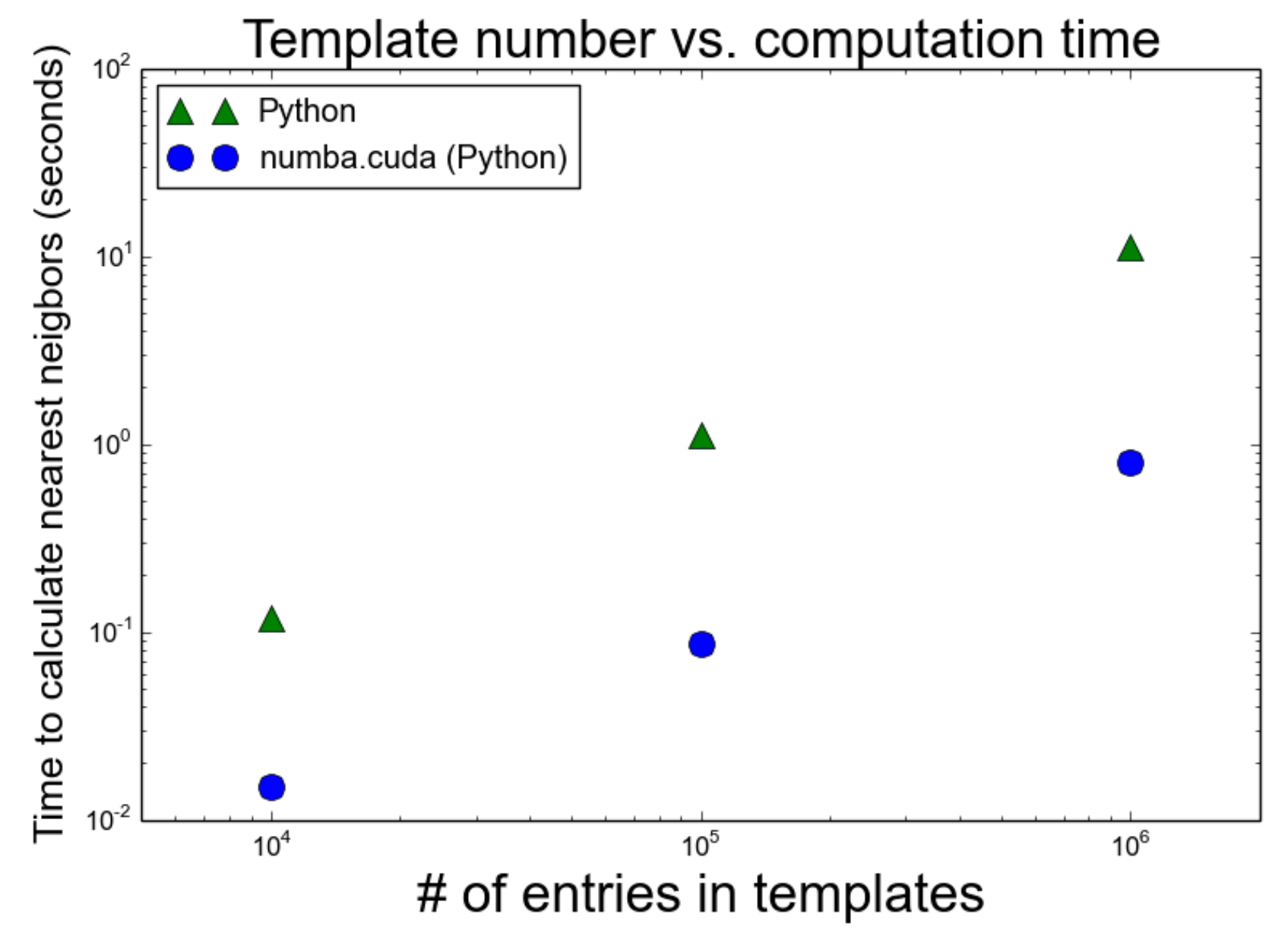




Our templates for the signal (left) and background (right). We ran tests with 10k, 100k, and 1M events in these templates. The background is uniformly distributed and the signal is modeled as a double Gaussian.

We then choose a radius (0.01 for our tests) and count how many of the signal or background template points are within that radius of each data point. ***This nearest-neighbors counting is very computationally intensive and goes as the number of templates points.*** The size of the circles on to the right represent the density of template points around each data point.





Lindsay ran the fits on different samples ~1000 times and found that there is a slight bias (~3%) in the amount of signal extracted by the fit. She calculated the nearest-neighbors using the ***scipy.spatial.dist.cdist*** routine. To test the bias with more template points we needed a faster routine, and so we implemented a ***numba.cuda*** nearest-neighbor routine, *giving us a speed-up of over 10x!* However the bias still persists, and will be the subject of further study.

Number theory

Garrett Allen, a rising-junior Computer Science major at Siena, had become intrigued by a conjecture by Siena Mathematics professor, Mohammad Javaheri, called the **Modified 3x + 1 Conjecture**, which we state below.

``Conjecture states that the iteration of the map

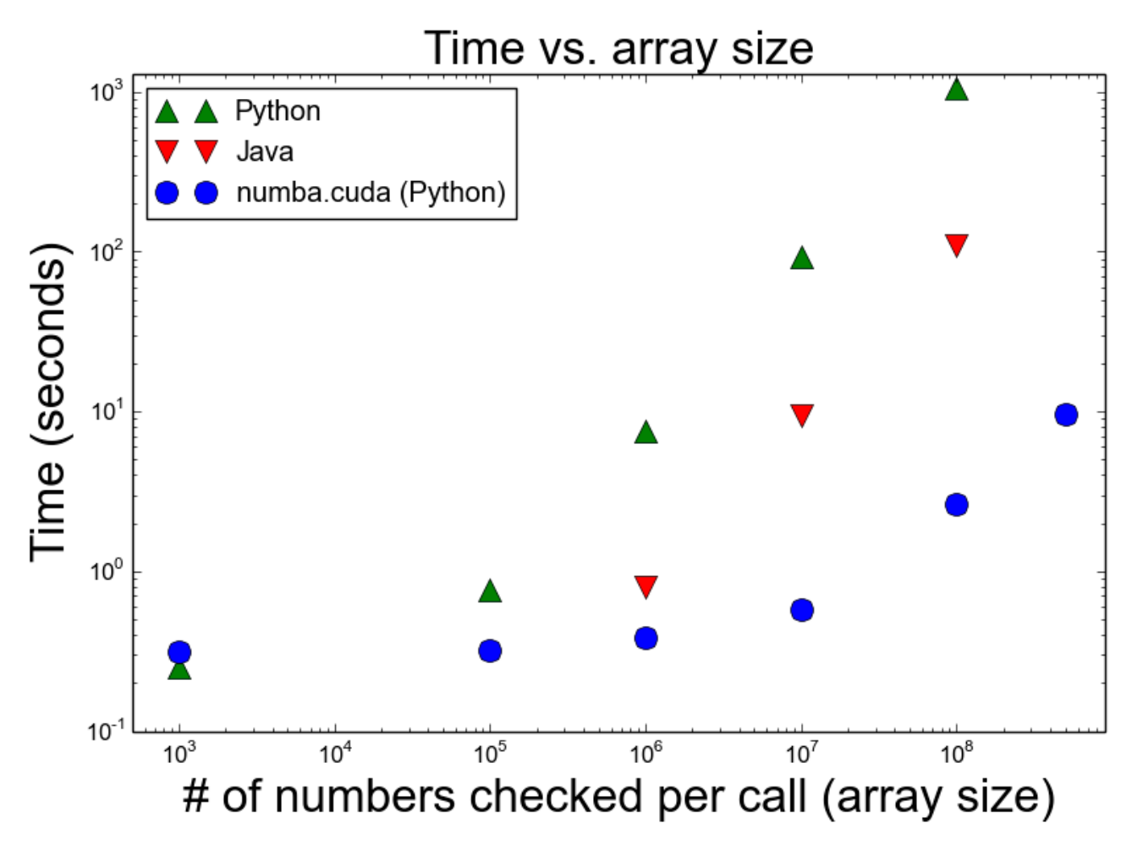
$$T(x) = \begin{cases} x/2 & \text{if } x \text{ is even;} \\ 3x+1 & \text{if } x \text{ is odd,} \end{cases}$$

on every positive integer ends in the cycle **{1, 2, 4}**. By viewing the branches of the map ***T(x)*** comprising a semigroup action on positive integers, we have the following modification: Let $\boldsymbol{T_1(x) = x/2}$ and $\boldsymbol{T_2(x) = 3x + 1}$. Let ***m*** and ***n*** be two positive integers that are not divisible by 3. Then there exists a sequence of iterations of $\boldsymbol{T_1}$ and $\boldsymbol{T_2}$ that maps ***m*** to ***n***. By examining the problem backwards, we verify this modified conjecture for all ***m***, $\boldsymbol{n \le 1.676 \times 10^{13}}$."

Garrett had taken a class in Java and written a program to check for counter-examples to this conjecture (referred to above). After just a few weeks of playing with some example code, he decided to implement the same algorithm in ***numba.cuda***. The results are shown below, comparing Java, ***numba.cuda***, and native Python for comparison. The *x*-axis shows the number of numbers (array size) that were checked in any one function/kernel call.

When a small amount (<1M) of numbers are checked for counter-examples, we see little difference. For larger quantities (>10M), the ***numba. cuda*** is about 10x faster!

While a C-CUDA implementation would be even faster, that was not the point of this exercise. Garrett did not have experience with C, and yet was able to indulge his number theory interests using GPU programming techniques!



After about 2 weeks of continuous running the ***numba.cuda*** code, Garrett was unable to find any counter examples in the first *50 trillion* numbers! The conjecture is still unproven, however.

Hardware used

All CPU tests were done on a Intel Xeon CPU E5-1620 v2 @ 3.70GHz (8-core machine, though only one was used for any given test) with 32 GB of RAM. The build was by Microway.

All GPU tests were performed on an NVIDIA Tesla K40, donated by NVIDIA.

Conclusions and Acknowledgements

Prior to this work, Garrett had never programmed in any sort of parallel framework and Lindsay was not aware of the concept. Garrett made use of the first lesson of Udacity's ``Introduction to Parallel Programming" course and slides provided by Continuum Analytics to introduce ***numba.cuda***, and in a few weeks was able to pursue his own ideas on number theory and contribute to Lindsay's project, which she was already programming in Python.

We found the Pythonic numba.cuda to be an excellent tool to expose undergraduate students to the world of GPU programming. It has a fairly low threshold for entry and we plan to continue to explore its efficacy in undergraduate education.

It should be noted that Continuum Analytics did not contribute to this poster, nor were they aware of this work, beyond bug reports filed on Github.

Support was provided by a grant from the Center for Undergraduate Research and Creative Activity (CURCA), Siena College and by NSF grant PHY-1307562. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research.

NSF