tensorscout

Release 2.1

Andrew Garcia, Ph.D.

Apr 08, 2023

CONTENTS

CHAPTER

ONE

WHAT IF FOR SOME REASON, YOU COULD UNLOCK 100% OF YOUR PROCESSING POWER?

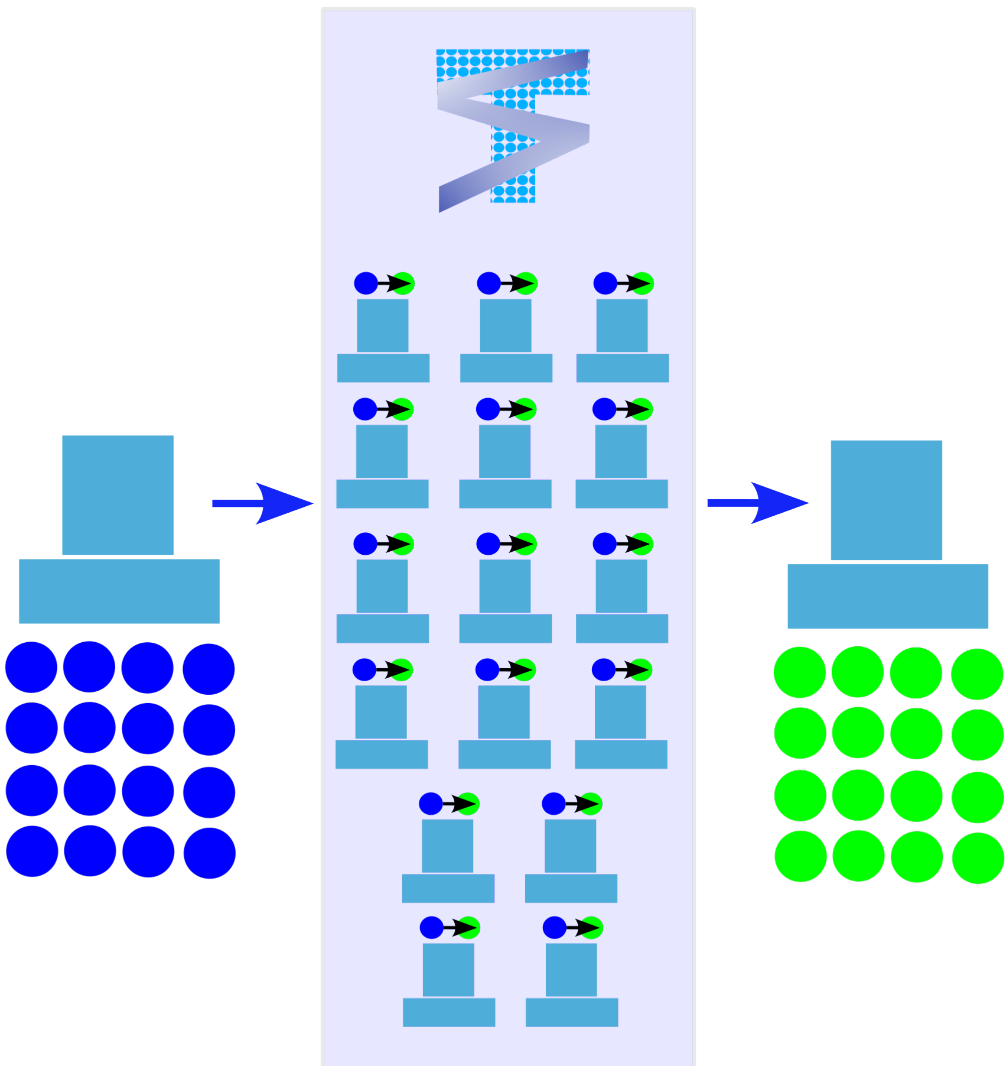


Fig. 1: A single computer leveraging the multiprocessing capabilities of tensorscout to distribute tasks to 16 computers and aggregate the results back to the original machine.

This Python package is simply, a collection of decorators that streamline the use of parallel processing with Python. These decorators are powered by pathos and allow users to distribute operations over multiple CPU cores or vCPUs (with cloud computing), significantly reducing the time required for computation.

Specifically, these decorators allow users to partition arrays into sectors and allocate operations for each sector over the defined available cores. The package currently does not include support for GPUs for faster processing, thought it may be a desired feature for the future.

Overall, this package is ideal for users working with large-scale tensor operations and seeking to optimize performance through parallel processing.

Check out the *Usage* section for further information, including how to *Installation* the project.

Note: This project is under active development.

CHAPTER

TWO

CONTENTS

2.1 Usage

2.1.1 Installation

It is recommended you use voxelmap through a virtual environment. You may follow the below simple protocol to create the virtual environment, run it, and install the package there:

```
$ virtualenv venv
$ source venv/bin/activate
(.venv) $ pip install tensorscout
```

To exit the virtual environment, simply type `deactivate`. To access it at any other time again, enter with the above `source` command.

2.1.2 Simulations and/or sampling with multiple processors

Simple Return Value

When performing Monte Carlo sampling at a high number, it can significantly impact computing power. To address this, we have developed the `@multicarlo` decorator, which allocates a specific number of iterations to a defined number of available processors or cores. In our case, since we have a computer with 4 cores, we have set the num_cores to 4. However, you can set it to as many cores as your computer or server may have available.

In this example, we compare the runtime performance of this multiprocessing decorator with the bare approach, which uses a single core. We begin by importing all the required modules and defining a function that is used in both approaches to avoid redundancy.

```python
import tensorscout as scout
import numpy as np
import matplotlib.pyplot as plt
from timethis import timethis

def make_histograms(results, title):
   plt.figure()
   plt.title(title+'(N = 100,000)')
   plt.hist(data,alpha=0.5,label='data')
   plt.hist(results,alpha=0.5,label=title)
   plt.legend()
```

The operations we run on both methodologies are random sampling operations which take random numbers from a distribution. For both methods, we set the number of samples to 100,000, which is a considerable amount. In the following code block, we apply the `@multicarlo` decorator to our random sampling function `monte_carlo_function` and distribute the sampling iterations across four cores.

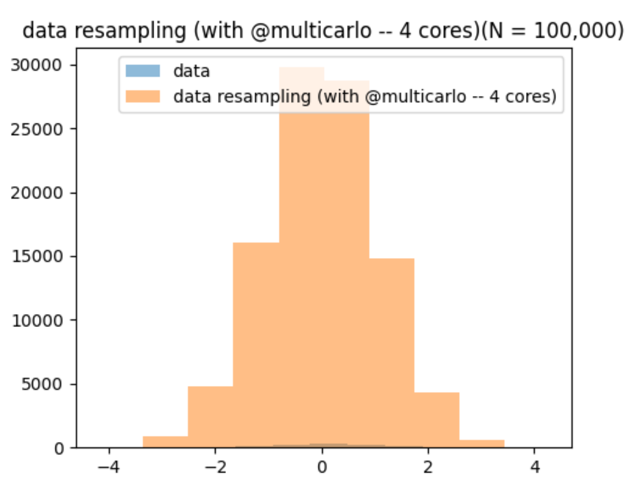
The `timethis()` function is used to record the run times of both methods and print them as a terminal output.

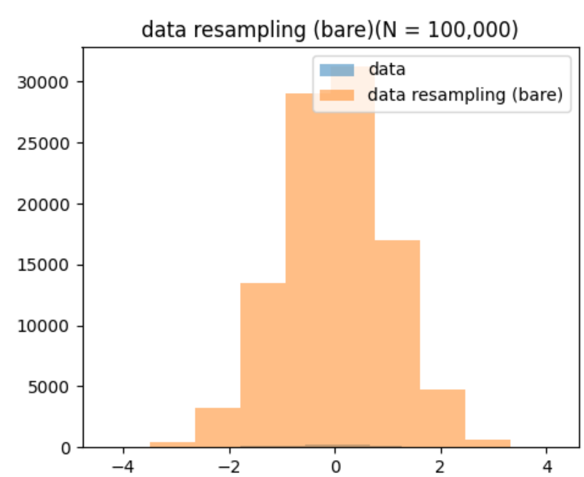
```
title = 'data resampling (with @multicarlo -- 4 cores)'
with timethis(title):

   @scout.multicarlo(num_iters=100000, num_cores=4)
   def monte_carlo_function(data, *args, **kwargs):
      simulated_data = np.random.normal(np.mean(data), np.std(data))
      return simulated_data

   data = np.random.normal(0, 1, 1000)
   results = monte_carlo_function(data)
   make_histograms(results,title)
```

The following code block executes the same tasks as the previous block, but using a bare approach, meaning that it uses a single core to perform all 100,000 random samples.

```
title='data resampling (bare)'
with timethis(title):

   def monte_carlo_function_bare(data, *args, **kwargs):
      simulated_data = np.random.normal(np.mean(data), np.std(data))
      return simulated_data

   data = np.random.normal(0, 1, 1000)
   results = [monte_carlo_function_bare(data) for i in range(100000)]
   make_histograms(results,title)

plt.show()
```

The output for the previous three code blocks is displayed below.





```
>>> [OUT]
monte carlo resampling (with @multicarlo -- 4 cores): 2.812 seconds
monte carlo resampling (bare): 4.328 seconds
```

Both approaches produce a similar random sampling distribution outcome, but the `@multicarlo` decorated function that uses multiprocessing on 4 cores shows around 50% better runtime performance.

Campfire: Generating a Multiprocessing-Powered Dictionary

Fig. 1: Much like a campfire which brings people together and allow for sharing stories and experiences, this decorator brings together the results of simulations across `num_cores` multiple processors and regroups them in a dictionary by key.

If the algorithm is refined further, we may consider `campfire` a more powerful method decorator than the former because dictionaries can return several outputs and may be accessed by their keys. The below example is from the Python tests section and shows how to return values from a "simulation" stored in `x y z` keys.

```
with timethis("campfire dictionary"):

    @scout.campfire(num_iters=400, num_cores=4)
    def simulate_data(data, num_iters):
        for i in range(1000):
           'stress test == 1000 iters'
            x = [np.random.normal(0, 1) for i in range(num_iters)]
            y = [np.random.normal(0, 1) for i in range(num_iters)]
            z = [np.random.normal(0, 1) for i in range(num_iters)]
        return {'x': x, 'y': y, 'z': z}

    data = {'data': None}

    results = simulate_data(data, num_iters=1)
    results = results["data"]
```

(continues on next page)

(continued from previous page)

```
    map = {key: [] for key in ['x','y','z']}
    for i in results:
        for key in i.keys():
            map[key].append(i[key][0])

    print(map.keys())
    print(len(map['x']))

print('-------------------------------------')

with timethis("bare dictionary"):

    def simulate_data_bare(data, num_iters):
        for i in range(1000):
            x = [np.random.normal(0, 1) for i in range(num_iters)]
            y = [np.random.normal(0, 1) for i in range(num_iters)]
            z = [np.random.normal(0, 1) for i in range(num_iters)]
        return {'x': x, 'y': y, 'z': z}

    data = 'hot-dog'
    results = simulate_data_bare(data, num_iters=400)

    print(results.keys())
    print(len(results['x']))
```

```
>>> [OUT]
dict_keys(['x', 'y', 'z'])
400
campfire dictionary: 0.851 seconds
-------------------------------------
dict_keys(['x', 'y', 'z'])
400
bare dictionary: 1.269 seconds
```

The `campfire` is still under development.

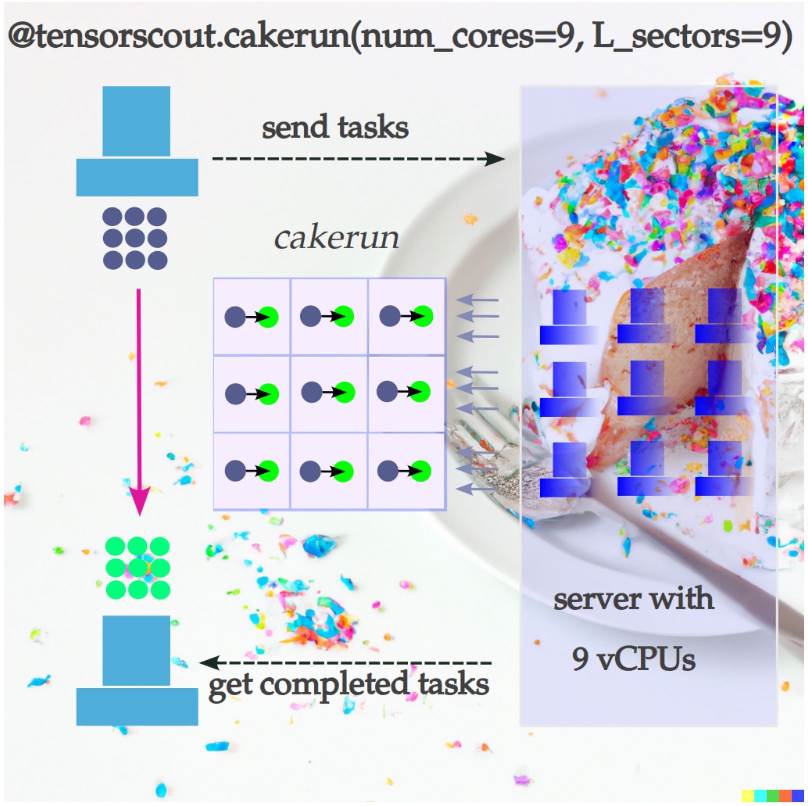
2.1.3 Parallel Computation on Sectorized Matrices using Multiprocessing

The question of whether it's faster to eat a cake alone or have 100 people cut a slice and eat their portions until it's gone highlights the main concept behind the cakerun decorator. Essentially, the decorator partitions an array into a specified number of equally-sized sectors and performs the same task on all sectors in parallel.

In this example, we set the number of cores to 4 and compare the performance of using multiprocessing versus using a single core. Before proceeding, we import all necessary modules and define the draw function which is used in both approaches to avoid redundancy. Additionally, we define the initial matrix, which is a 252 x 252 matrix of 1s, that will be operated on by both methodologies.

```
import tensorscout as scout
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)



(continued from previous page)

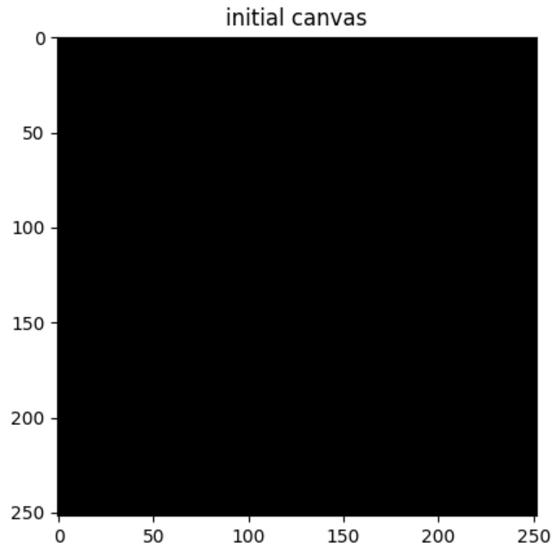
```
from timethis import timethis

num_iters = 40000

def draw(result):
    plt.figure()
    plt.title('{} -- $N_{{perforated}}$ = {}'.format(title, np.multiply(*result.shape) -␣
↪np.count_nonzero(result)))
    plt.imshow(result,cmap='bone')


matrix = np.ones((252,252))

plt.imshow(matrix,cmap='bone')
plt.title('initial canvas')
```

In this example, the initial matrix is composed entirely of 1s and will appear as a single color when drawn. The purpose of this code is to apply an operation called "perforation" to the matrix. At each iteration, a random x-y coordinate is selected and the value at that location is set to 0.

The first case demonstrates the use of the `@cakerun` decorator to split the matrix into sectors and apply the perforate function to each sector. The former code block specifies 40,000 perforating iterations, which for the case of this aprroach has them evenly distributed across the 4 sectors, resulting in 10,000 iterations per sector, ocurring simultaneously.

```
title = 'cakerun MP (4 cores)'
with timethis("{}".format(title)):
```

(continues on next page)



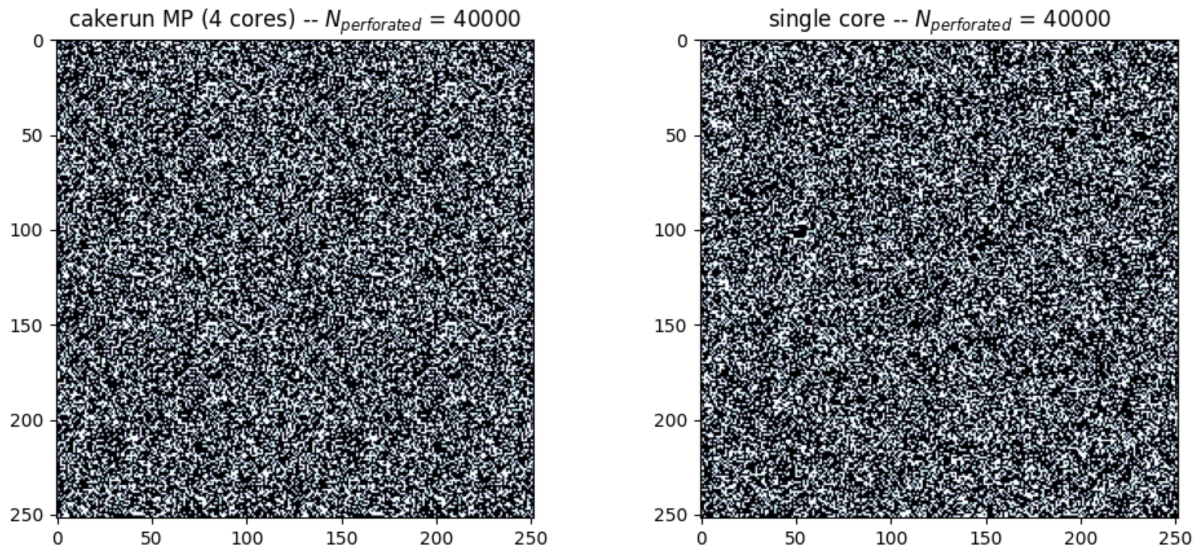
(continued from previous page)

```
    cores = 4
    @scout.cakerun(num_cores=cores, L_sectors=2)
    def perforate(sector):

        for i in range(num_iters // cores):
                cds = np.argwhere(sector!=0)
                sector[tuple(cds[np.random.randint(cds.shape[0])])] = 0
        return sector

    result = perforate(matrix)
    draw(result)
```

In the next code block, the perforating operation is applied for 40,000 iterations using a bare approach with a single processor. Hence, there is no task split involved.

```
title = 'single core'
with timethis("{}".format(title)):

    def perforate_bare(sector):
        for i in range(num_iters):
                cds = np.argwhere(sector!=0)
                sector[tuple(cds[np.random.randint(cds.shape[0])])] = 0
        return sector


    result = perforate_bare(matrix)
    draw(result)


plt.show()
```

The following are graphical and runtime comparisons of both methods:




```
>>> [OUT]
cakerun MP (4 cores): 2.968 seconds
single core: 25.868 seconds
```

It is apparent that both approaches yield a similar outcome and have the same number of perforations. However, the `@cakerun` decorated function, which uses four cores simultaneously, has a runtime that is 8-9 times faster than the bare approach.

2.2 API Reference

2.2.1 Global Methods

At the time, tensorscout is a lean module composed of only 2 decorators.

class tensorscout.**cakerun**(*num_cores*, *L_sectors*)

This decorator partitions an array into sectors and applies a given function to each sector in parallel. The result of each computation is merged into a final output array.

Parameters

num_cores: int
Number of processors to use

L_sectors
[int] The length scale for the number of sectors [per column]. For non-square arrays, the number of sectors per row gets adjusted as a function of this value

`class tensorscout.campfire(num_iters, num_cores)`

Much like a campfire which brings people together and allow for sharing stories and experiences, this decorator brings together the results of simulations across `num_cores` multiple processors and regroups them in a dictionary by key.

Parameters

num_cores: int
Number of processors to use

num_iters
[int] The number of iterations to perform for a specific model / Monte Carlo simulation.

`class tensorscout.multicarlo(num_iters, num_cores)`

This decorator performs a non-dynamic operation or task for a specified number of iterations num_iters and distributes the tasks across a requested number of available processors num_cores.

Parameters

num_cores: int
Number of processors to use

num_iters
[int] The number of iterations to perform for a specific model / Monte Carlo simulation.

INDEX

C

M