Sdlewis 12134058

- **What is your question?**

  For this assignment, I tried to compare the native Python code to Python GPU accelerated code using the Numba package.

  So, I chose the following three questions.

  1. Matrix Multiplication
     In matrix multiplication I am trying to multiply two matrices together to get a new one. Usually, doing this in Python is slow, especially if the matrices are big. But if we use Numba with GPU acceleration, it speeds things up a lot.

  2. Vector Addition
     In vector addition I am just adding the numbers in two vectors together. In Python, if the vectors are large, then traversing it takes time. But with Numba and GPU acceleration, it gets way faster.

  3. Sum Reduction
     In sum reduction I am adding all the numbers in a list, the usual method in Python i.e. to traverse the entire list and accumulating the sum is slow. But with Numba and GPU acceleration, it speeds things.

- **Your experimental setup (machines, etc)**
  - **Are there any side effects or artifacts related to your experimental setup that might impact your results?**
  - **Who you worked with, if anyone, to help setup this environment**

  Google Colab provides an option to switch to a T4 GPU, which can significantly accelerate computations. However, I encountered memory limit issues with different input sizes for my experiments. Specifically:

  - Matrix Multiplication: The program took too long to run when the matrix sizes were set to 1000 due to memory constraints.
  - Vector Addition: I encountered a Memory Limit Exceeded (MLE) error when attempting to perform vector addition with an input size of 10,000,00000.
  - Sum Reduction: Similarly, I faced an MLE error when executing Python GPU-accelerated code for sum reduction with an input size of 1,000,000,000.

  Collaboration: I conducted the experiments independently without collaborating with anyone.

- **How you ran your tests to answer your question**

  - Matrix Multiplication:
    1. I tried multiplying matrices of sizes 100, 250, and 500.
    2. First, I timed how long it took using Python with GPU acceleration, then without.
    3. I compared the times to see if using the GPU made it faster.
  - Vector Addition:
    1. I added together different-sized lists of numbers: 1000000, 10000000, and 100000000 numbers.
    2. I timed how long it took using Python with GPU acceleration and without for each list size.
    3. Then, I looked at the times to see if using the GPU made it faster.
  - Sum Reduction:
    1. I added up numbers in lists of sizes 1000000, 10000000, and 100000000.
    2. First, I timed how long it took using Python with GPU acceleration and without for each list size.
    3. Then, I compared the times to see if using the GPU made it faster.

- **Test results:**
  - **How accurate are they?**
    - **Make sure you run enough tests that your results are accurate enough.**
  - **Do you have variation in your testing?**

1. Accuracy:

To ensure accuracy, I conducted multiple tests for each question and input size. I ran each sample size three times just to ensure that the time I obtained in the second and third runs was close to the value obtained in the first run.

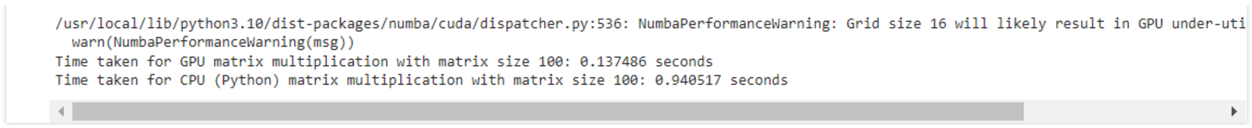
2. Variation:

To ensure variation, I conducted multiple tests for each question and input size. For matrix multiplication, I ran tests with matrices of sizes 100, 250, and 500. For vector addition, I tested with list sizes of 1000000, 10000000, and 100000000 elements. Lastly, for sum reduction, I tested list sizes of 1000000, 10000000, and 100000000 elements. By running multiple tests, I aimed to obtain consistent results and minimize the impact of random variations or anomalies.

- **Your conclusion/answer to the question**

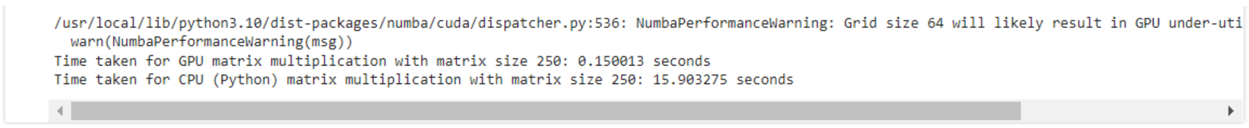
Matrix Multiplication

For sample size 100



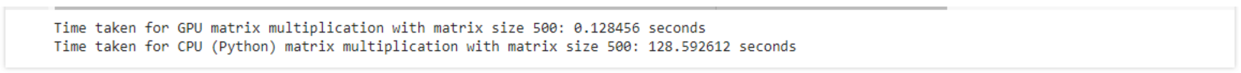
```
/usr/local/lib/python3.10/dist-packages/numba/cuda/dispatcher.py:536: NumbaPerformanceWarning: Grid size 16 will likely result in GPU under-uti
  warn(NumbaPerformanceWarning(msg))
Time taken for GPU matrix multiplication with matrix size 100: 0.137486 seconds
Time taken for CPU (Python) matrix multiplication with matrix size 100: 0.940517 seconds
```

For sample size 250



```
/usr/local/lib/python3.10/dist-packages/numba/cuda/dispatcher.py:536: NumbaPerformanceWarning: Grid size 64 will likely result in GPU under-uti
  warn(NumbaPerformanceWarning(msg))
Time taken for GPU matrix multiplication with matrix size 250: 0.150013 seconds
Time taken for CPU (Python) matrix multiplication with matrix size 250: 15.903275 seconds
```

For sample size 500



```
Time taken for GPU matrix multiplication with matrix size 500: 0.128456 seconds
Time taken for CPU (Python) matrix multiplication with matrix size 500: 128.592612 seconds
```

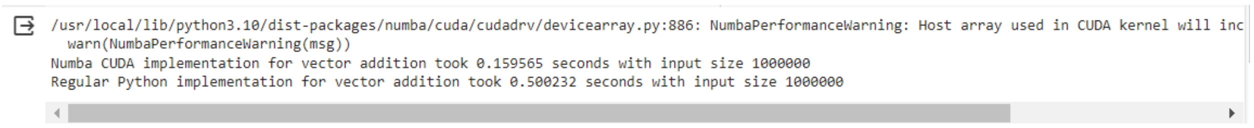
Type/Size	100	250	500
GPU code time	0.137	0.150	0.128
Native Python time	0.940	15.903	128.59

In summary, the table compares execution times (in seconds) between GPU-accelerated code and native Python code across input sizes of 100, 250, and 500. The GPU code consistently demonstrates faster execution times than native Python code, with values ranging from 0.137 to 0.128 seconds for the GPU code and 0.940 to 128.59 seconds for native Python code.

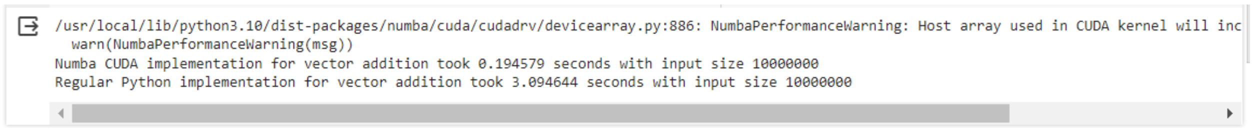
Sdlewis 12134058

Vector Addition

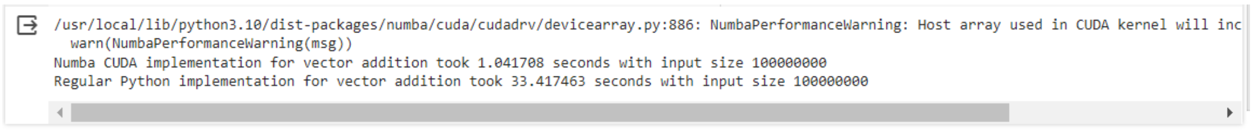
For sample size 1000000


```
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:886: NumbaPerformanceWarning: Host array used in CUDA kernel will inc
  warn(NumbaPerformanceWarning(msg))
Numba CUDA implementation for vector addition took 0.159565 seconds with input size 1000000
Regular Python implementation for vector addition took 0.500232 seconds with input size 1000000
```

For sample size 10000000


```
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:886: NumbaPerformanceWarning: Host array used in CUDA kernel will inc
  warn(NumbaPerformanceWarning(msg))
Numba CUDA implementation for vector addition took 0.194579 seconds with input size 10000000
Regular Python implementation for vector addition took 3.094644 seconds with input size 10000000
```

For sample size 100000000


```
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:886: NumbaPerformanceWarning: Host array used in CUDA kernel will inc
  warn(NumbaPerformanceWarning(msg))
Numba CUDA implementation for vector addition took 1.041708 seconds with input size 100000000
Regular Python implementation for vector addition took 33.417463 seconds with input size 100000000
```

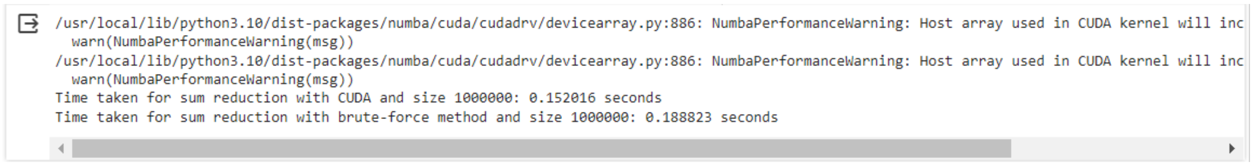
Type/Size	1000000	10000000	100000000
GPU code time	0.159	0.194	1.041
Native Python time	0.500	3.09	33.417

In summary, the table compares execution times (in seconds) between GPU-accelerated code and native Python code across input sizes of 1000000, 10000000, and 100000000. The GPU code consistently demonstrates faster execution times than native Python code, with values ranging from 0.159 to 1.041 seconds for the GPU code and 0.500 to 33.417 seconds for native Python code.

Sdlewis 12134058

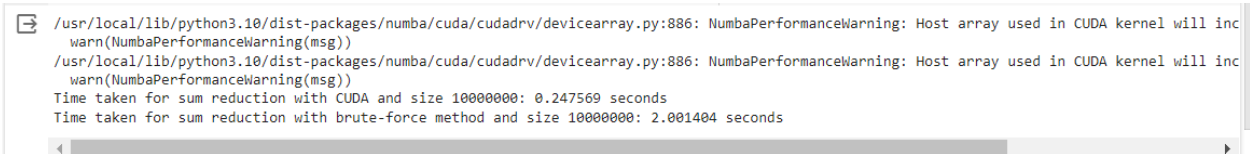
Sum Reduction

For sample size 1000000



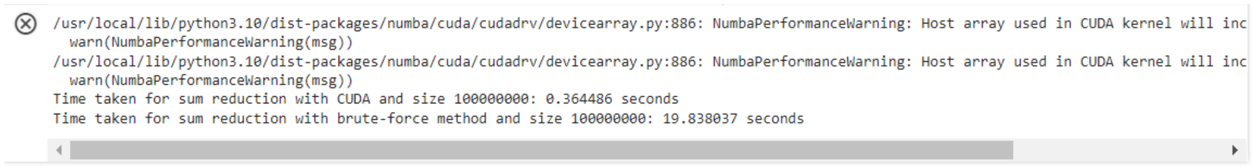
```
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:886: NumbaPerformanceWarning: Host array used in CUDA kernel will inc
  warn(NumbaPerformanceWarning(msg))
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:886: NumbaPerformanceWarning: Host array used in CUDA kernel will inc
  warn(NumbaPerformanceWarning(msg))
Time taken for sum reduction with CUDA and size 1000000: 0.152016 seconds
Time taken for sum reduction with brute-force method and size 1000000: 0.188823 seconds
```

For sample size 10000000



```
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:886: NumbaPerformanceWarning: Host array used in CUDA kernel will inc
  warn(NumbaPerformanceWarning(msg))
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:886: NumbaPerformanceWarning: Host array used in CUDA kernel will inc
  warn(NumbaPerformanceWarning(msg))
Time taken for sum reduction with CUDA and size 10000000: 0.247569 seconds
Time taken for sum reduction with brute-force method and size 10000000: 2.001404 seconds
```

For sample size 100000000



```
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:886: NumbaPerformanceWarning: Host array used in CUDA kernel will inc
  warn(NumbaPerformanceWarning(msg))
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:886: NumbaPerformanceWarning: Host array used in CUDA kernel will inc
  warn(NumbaPerformanceWarning(msg))
Time taken for sum reduction with CUDA and size 100000000: 0.364486 seconds
Time taken for sum reduction with brute-force method and size 100000000: 19.838037 seconds
```

Type/Size	1000000	10000000	100000000
GPU code time	0.159	0.194	1.041
Native Python time	0.500	3.09	33.417

In summary, the table compares execution times (in seconds) between GPU-accelerated code and native Python code across input sizes of 1000000, 10000000, and 100000000. The GPU code consistently demonstrates faster execution times than native Python code, with values ranging from 0.159 to 1.041 seconds for the GPU code and 0.500 to 33.417 seconds for native Python code.

As the input size increases, the performance advantage of GPU-accelerated code becomes more evident, highlighting its efficiency for larger computations.

- **What you learned from answering the question**

  Matrix Multiplication:

  Through my exploration of matrix multiplication, I've seen how using GPU-accelerated code makes a big difference. It's much faster than using regular Python code, especially when dealing with bigger matrices. This showed me that using GPUs can really speed up certain types of calculations.

  Vector Addition:

  When I looked into adding up vectors, I found that using GPU-accelerated code is a lot quicker than regular Python. This is especially noticeable when working with large sets of numbers. It became clear that using GPUs can make arithmetic tasks much faster.

  Sum Reduction:

  In my investigation of sum reduction, I noticed a big difference in speed between GPU-accelerated code and regular Python. This difference becomes even more apparent with larger sets of numbers. It was pretty clear that using GPUs can really speed up the process of adding up lots of numbers.

  I also found that threading on the GPU allows tasks to be done at the same time across its cores, making things faster. By using threading with GPU-accelerated code, I could take advantage of this feature. It helped me because it meant tasks like matrix multiplication and adding up numbers could be done much quicker. In simple terms, threading on the GPU made my experiments run faster by making the most out of its cores, which work together to get things done more efficiently.

- References to your papers and other references

Sdlewis 12134058

1. https://oneflow2020.medium.com/how-to-choose-the-grid-size-and-block-size-for-a-cuda-kernel-d1ff1f0a7f92
2. https://www.youtube.com/watch?v=uUEHuF5i_qI
3. https://github.com/olcf-tutorials/vector_addition_cuda
4. https://shreeraman-ak.medium.com/parallel-reduction-with-cuda-d0ae10c1ae2c
5. https://www.youtube.com/watch?v=bpbit8SPMxU