Question:

How much of a performance benefit is accelerated code vs non-accelerated code on a GPU? Specifically for the below cases:

1. Matrix Multiplication
2. Sum Reduction
3. Mandelbrot

Experiment Setup:

This experiment was performed on google collab using the NVIDIA T4 GPU [1]:
Specs:

- 16GB of GDDR6 memory
- CUDA 12.2
- 2560 CUDA cores

To run the program we just need a google account and log into colab and open the .ipynb file. CUDA is preinstalled on google colab so no additional steps were required. There is a limit on the usage of the GPUs unless it's a paid version. We did need to import a few libraries like numba, time, random, numpy, math, and pylab. These libraries are required for specific parts of the experiment. This experiment was performed solo.

Tests Procedure:

The tests were carried out on Colab using the .ipynb file as the testing environment. To ensure the reliability of our results and to mitigate the impact of any outliers, each case was run three times for both the non-accelerated and accelerated code. This approach aimed to provide a comprehensive understanding of the performance characteristics across multiple runs. In each test iteration, we generated a random array of size N for the different cases under examination. This standardised the input data across all tests, enabling a fair comparison of the performance between the non-accelerated and accelerated implementations. By applying identical logic to both versions, we could accurately evaluate which approach demonstrated superior performance under various conditions. Providing a detailed examination of the methodologies employed and the corresponding performance outcomes. This comprehensive exploration allows us to gain valuable insights into the effectiveness of optimization techniques and their impact on real-world applications.

Test Results / Accuracy:

Matrix Multiplication:

For this experiment, we utilised two matrices, each sized 1000, and performed matrix multiplication to obtain the resultant matrix. Initially, we generated two random matrices and subsequently applied a multiplication function, which accepts these matrices as input and produces the output matrix. The multiplication function incorporates three nested loops to iterate over the matrices.

Similarly, in the accelerated code, we adhere to the same procedure. We use a random_2d function responsible for generating the random matrices utilising the xoroshiro128p_uniform_float32 function. Subsequently, we utilise the multiply function to carry out the matrix multiplication, storing the result in the output matrix C. When utilising CUDA, direct matrix updating in C becomes feasible. A distinct approach is adopted here: employing a 2D CUDA grid where each element utilises threads to compute within its block. This strategy maximises parallel execution, optimising the computational process on the GPU for enhanced efficiency.

Below are the test results from the three runs done.

Matrix Multiplication	Run-1	Run-2	Run-3	Avg
non-accelerated	864.4105	864.4753	863.8885	864.2581
accelerated	0.1349	0.1366	0.1375	0.1366

Sum Reduction:

For this experiment, we use a 1D array. We aim to obtain the summation of the array as a result. For the non-accelerated code, we simply create an array of size 100000000 with random numbers. The reduction function is straightforward; it involves a loop where values are added to a single variable.

The accelerated code follows a similar pattern. We generate a random array of 100000000 numbers using the xoroshiro128p_uniform_float32 function. Then, we implement the reduction function. This function begins with each thread adding together pairs of elements in the shared memory array, doubling the stride at each iteration until all elements are combined into a single element.

Below are the results from the three runs:

Sum Reduction	Run-1	Run-2	Run-3	Avg
non-accelerated	14.4695	9.6587	9.6749	11.2677
accelerated	0.1587	0.1494	0.2467	0.1849

Mandelbrot:

For this experiment, we generate a Mandelbrot image with dimensions 1024 x 1536. This is achieved by creating an empty 2D array and then accessing each element as a pixel to calculate the Mandelbrot colour for that pixel. In the non-accelerated code, we utilise two nested "for" loops to iterate over the entire image.

In the accelerated code, we perform the same process as described above, but each pixel is calculated using parallel threads, leveraging the GPU.

Below are the results from the three runs:

Mandelbrot	Run-1	Run-2	Run-3	Avg
non-accelerated	6.2197	5.2599	6.1270	5.8688
accelerated	0.1981	0.3151	0.2000	0.2377

Note:

There was overhead when calculating the sum reduction using accelerated code, taking approximately 22 to 30 seconds for the entire block to run. The create_xoroshiro128p_states function consumed the most time in generating the random numbers. This resulted in an overall longer execution time compared to the non-accelerated code. Therefore, it might be more beneficial to use the non-accelerated code in this case.

Conclusion:

In conclusion, based on the above test comparing accelerated code to non-accelerated code, we can clearly see that the accelerated code performs at least 10 times better than the non-accelerated code in terms of performance, especially when dealing with computationally intensive tasks. The utilisation of parallel processing and hardware acceleration enhances the overall performance and optimises resource usage. Overall, the accelerated code appears to be more reliable for larger datasets and complex computations, making it more practical for real-world applications where volume and complexity may vary dynamically. However, this may not always hold true, as sometimes the overhead of the accelerated code could be higher. Hence, it is important to carefully study and profile the data before choosing the approach.

Learning:

The above experiment has helped me understand optimization techniques that can be applied to make code faster and make better use of various optimization strategies. It also assists in applying these techniques for practical applications through this experiment. Furthermore, it helps to understand that not every code requires optimization, but it is critical to understand the dataset and overhead before applying these techniques. Overall, addressing these questions provides valuable insights into performance optimization strategies and their implications for real-world applications, contributing to a deeper understanding of software development practices and principles.

References:

1. https://drlee.io/utilizing-gpu-and-tpu-for-free-on-google-colab-a-comprehensive-guide-fe2841592851#:~:text=Step%209%3A%20GPU%20Options%20in%20Colab&text=Tesla%20T4%3A%20With%2016GB%20of,memory%20and%203%2C584%20CUDA%20cores

2. https://numba.readthedocs.io/en/stable/cuda/examples.html

3. https://www.kaggle.com/code/landlord/numba-cuda-mandelbrot

4. https://csuchico-my.sharepoint.com/personal/bcdixon_csuchico_edu/_layouts/15/onedrive.aspx?ga=1&sw=auth&id=%2Fpersonal%2Fbcdixon%5Fcsuchico%5Fedu%2FDocuments%2FCourses%2FCSCI640%5FSpring2024%2FShared%2Fmatrix%2Epy&parent=%2Fpersonal%2Fbcdixon%5Fcsuchico%5Fedu%2FDocuments%2FCourses%2FCSCI640%5FSpring2024%2FShared