**Lab 3: Writing parallel programs using GPU and CUDA**

In this project, you will write a parallel application using GPUs. Your grade for the project will be based on the code, your documentation, and your testing results. You should turn in a report about your code, how it works, and the testing results with screenshots. Your code should be included with the report, and it should be well structured and documented code. You should include tests you used with your code. Please include all you are turning in for the project as a single tar or zip file that you upload to Canvas. (For each group, only one member needs to upload.)

This lab is based on the tutorial series available at <https://towardsdatascience.com/cuda-by-numba-examples-1-4-e0d06651612f> with a total of 4 parts. The assignment section extends the tutorial to ask you develop solutions for specific problems.

The goal of this project is to get familiar with CUDA programming with Python. CUDA was originally designed to support C. Later versions extended it to C++ and Fortran. In the Python ecosystem, one of the ways of using CUDA is through Numba, a Just-In-Time (JIT) compiler for Python that can target GPUs. With Numba, one can write kernels directly with (a subset of) Python, and Numba will compile the code on-the-fly and run it. While it does not implement the complete CUDA API, its supported features are often enough to obtain impressive speedups compared to CPUs (for all missing features, see the Numba documentation at https://numba.readthedocs.io/en/stable/cuda/overview.html#missing-cuda-features).

Numba is not the only library supporting CUDA. Another Python library, CuPy, offers both high level functions and low-level CUDA support for integrating kernels written in C. In this lab, we focus on using Numba, but you may explore other options as needed in the future.

For target programming environment, it is recommended that you use the Colab notebooks, which are available from Google (<https://colab.research.google.com/>), to perform the experiments. This does not require you to install anything on your computer as Colab allows you to connect to a GPU instance directly. To do so, go to Runtime and set the notebook settings as shown below. You do not need to access the premium GPUs for this experiment.

Graphical user interface, text, application

Description automatically generated

First we ensure the GPU is supported by using the code below:

Graphical user interface, text, application

Description automatically generated

You should see that your Colab notebook is connected to a backend CUDA-enabled device.

The main approach of Numba CUDA to distribute code on GPU cores is the cuda.jit decorator. It is used to define functions which will run in the GPU. We’ll start by defining a simple function, which takes two numbers and stores them on the first element of the third argument. Our first lesson is that kernels (GPU functions that launch threads) cannot return values. We get around that by passing inputs and outputs. This is a common pattern in C, but not very common in Python.

# Example 1.1: Add scalars

@cuda.jit

def add\_scalars(a, b, c):

    c[0] = a + b

dev\_c = cuda.device\_array((1,), np.float32)

add\_scalars[1, 1](2.0, 7.0, dev\_c)

c = dev\_c.copy\_to\_host()

print(f"2.0 + 7.0 = {c[0]}")

#  2.0 + 7.0 = 9.0

Note that in the kernel code, add\_scalars[1, 1](2.0, 7.0, dev\_c), the square brackets refer to the number of blocks in a grid, and the number of threads in a block, respectively.

If you can get the following output, it means your CUDA environment is properly set up:



The project assignment is asking you to implement the Monte Carlo method on GPUs for the specific problem of calculating the value of π. The Monte Carlo method is a powerful tool for simulating complex physical and mathematical systems. It is a statistical method that relies on generating random samples to approximate the solution to a problem. The method is often used in fields such as physics, finance, and engineering to model real-world systems and predict outcomes.

The Monte Carlo method can be time-consuming, as it requires many random samples to achieve a good approximation of the solution. In such cases, a parallel implementation of the method can greatly reduce the time required to find a solution. By dividing the problem into smaller parts and running them simultaneously on different processors, parallel computing can significantly reduce the time required to find a solution.

Diagram

Description automatically generated with medium confidence

Figure Estimation of PI (https://www.101computing.net/estimating-pi-using-the-monte-carlo-method/)

In this project, the goal is to implement a parallel version of the Monte Carlo method for calculating π using Numba and CUDA. The idea is that you will simulate taking a random point out of a square shape as shown in Figure 1. If you divide the number of points within the circle to the total number of points, then you will get an estimate on the area of the circle, which can be used to infer the value of π.

The report for your project should include your choices of multiple groups of parameters for the CUDA kernel, by changing the number of blocks and threads. Your result should also show the latency for the calculation, and demonstrate how the accuracy improves with more samples. You should include in your report a table showing the correlation of accuracy with the number of samples, as well as the computational time. The table is illustrated as below. You should include at least ten rows in the report.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter settings | Total number of samples | Computational time | Accuracy of result |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |