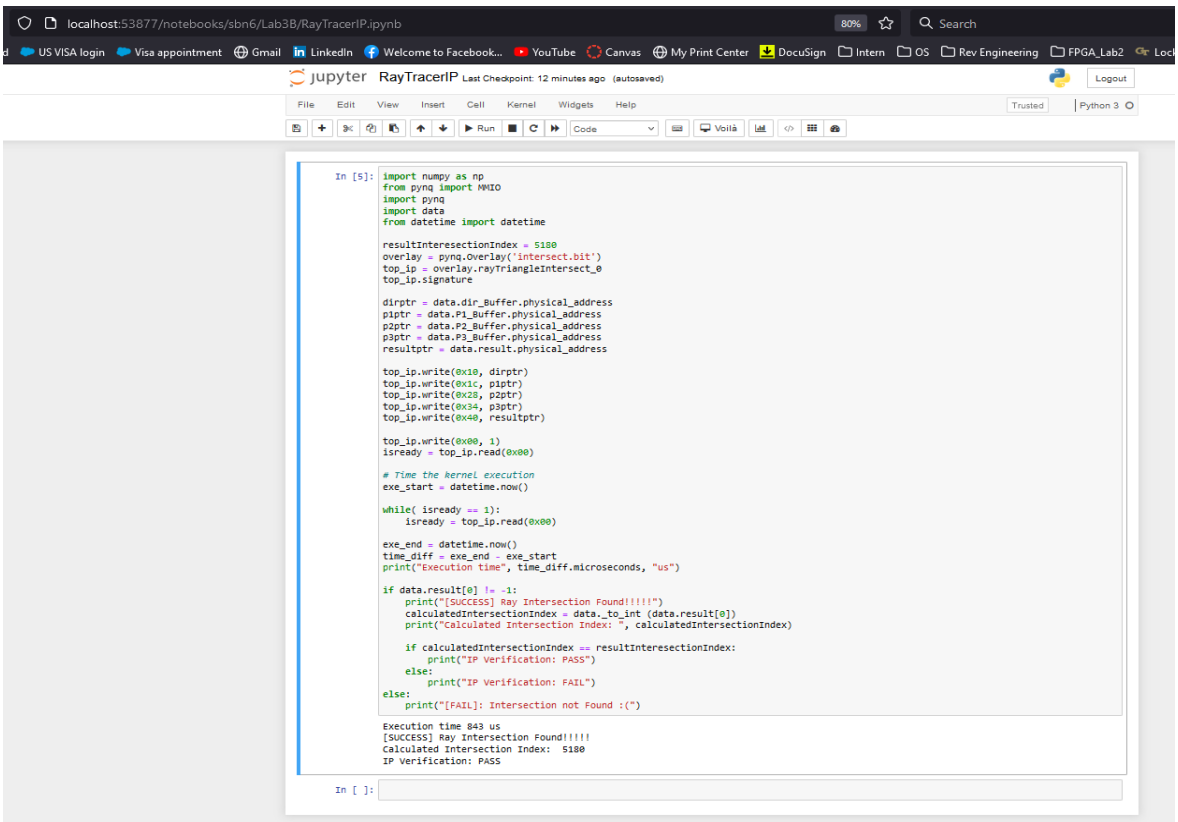
Lab3B-Implementing Ray Tracing Acceleration on PYNQ-Z2-FPGA

On board execution time vs HLS synthesis report:

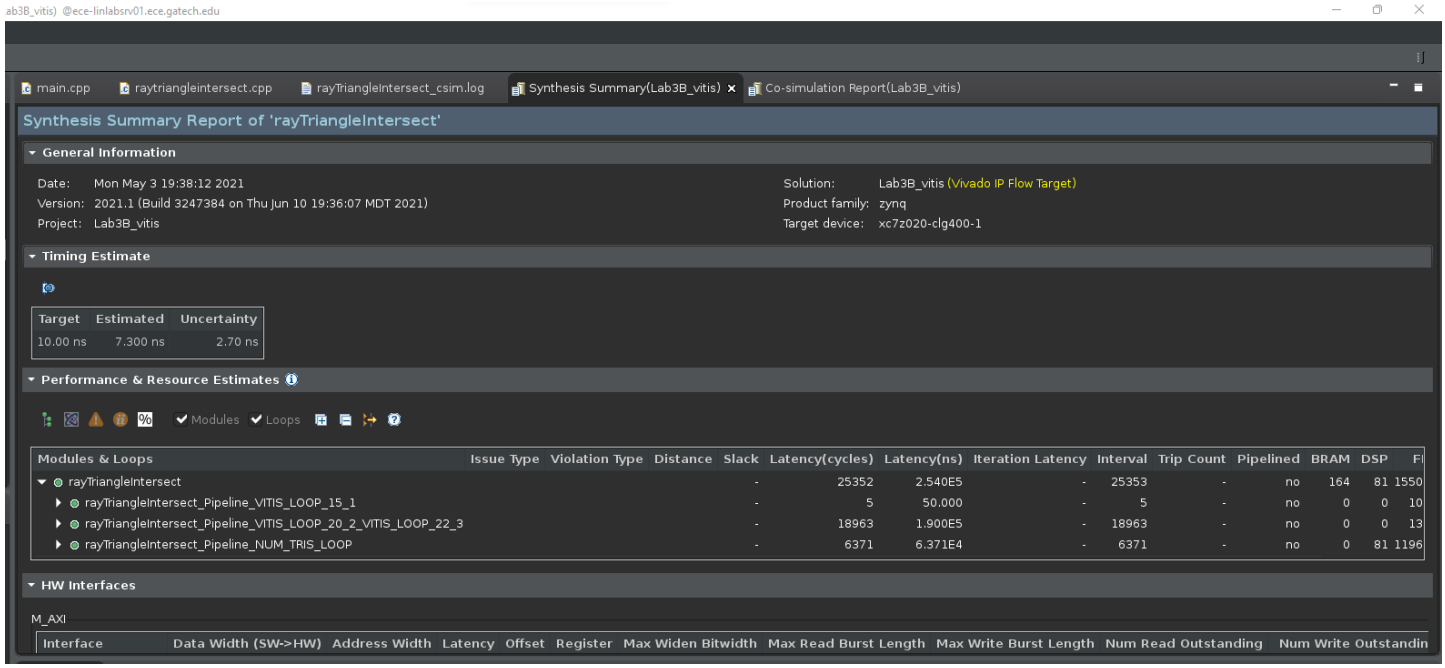
- The on-board execution time is **843us** and the HLS synthesis report's mentioned latency is **254us.**

Stepwise description of Lab3B:

- We use an auxiliary python file (data.py) that stores our primitive coordinate inputs in an array format. It also initializes the direction matrix and the result buffer that will be used by the *rayTriangleIntersect* IP. The data.py file also performs a Pynq allocate of the required size for all inputs required. We then copy the primitives (stored in python lists) to the Pynq allocated buffers.
- Importing the data.py to the Jupyter Notebook will set all the inputs (P1_Buffer, P2_Buffer, P3_Buffer, Direction matrix) required by the *rayTriangleIntersect* IP with appropriate values.
- Within the Jupyter Notebook, we import the data.py and other numpy, Pynq modules.
- We read the bit stream file 'intersect.bit' and instantiate a top_ip that points to the *rayTriangleIntersect_0* instance. We obtain the physical addresses of each input buffer and use it to create pointers.
- We write the pointers to the addresses found in the driver header file 'xraytriangleintersect_hw.h'.
- The next step is to start the HLS kernel by writing to the 0x00 location. We then wait for the completion of the kernel.
- Finally, we compare the result index with the expected result and decide whether the IP works as expected.
- As seen in the below figure, the Jupyter Notebook run shows that the result buffer is filled with triangle index that intersected with the ray and the index matches with the expected result and therefore the test is a **SUCCESS**.



HLS Synthesis Report – Latency Results:



Handling of fixed-point data type in Python:

- We developed a custom function to handle fixed-point data types in Python. We need to handle fixed-point in-order to pass our inputs to the *rayTriangleIntersect* IP.
- For a fixed-point data type of <32,16> we represent a 16-bit fractional part and 16-bit integer bitwidth. To perform this conversion, we take in the number, type cast it to a Python numpy longdouble and multiply it with 2^16, thereby, converting the input number to a fixed-point format.
- We also have a dedicated function to convert the fixed-point data to a python compatible int format that is used to verify the expected results.

```
def _to_fixed32(x) :
    i = int(x)
    f = x - i
    x = i + f
    fxp = int(np.longdouble(x) * (2.0 ** 16))
    return fxp


def _to_int(x) :
    i = int(x)
    fxp = int(np.longdouble(x) * (2.0 ** -16))
    return fxp
```

Vivado Block Diagram: