ECE 498ICC

Lab 2: Enable Edge AI in Raspberry Pi (Node-RED Implementation and CUDA Programming)

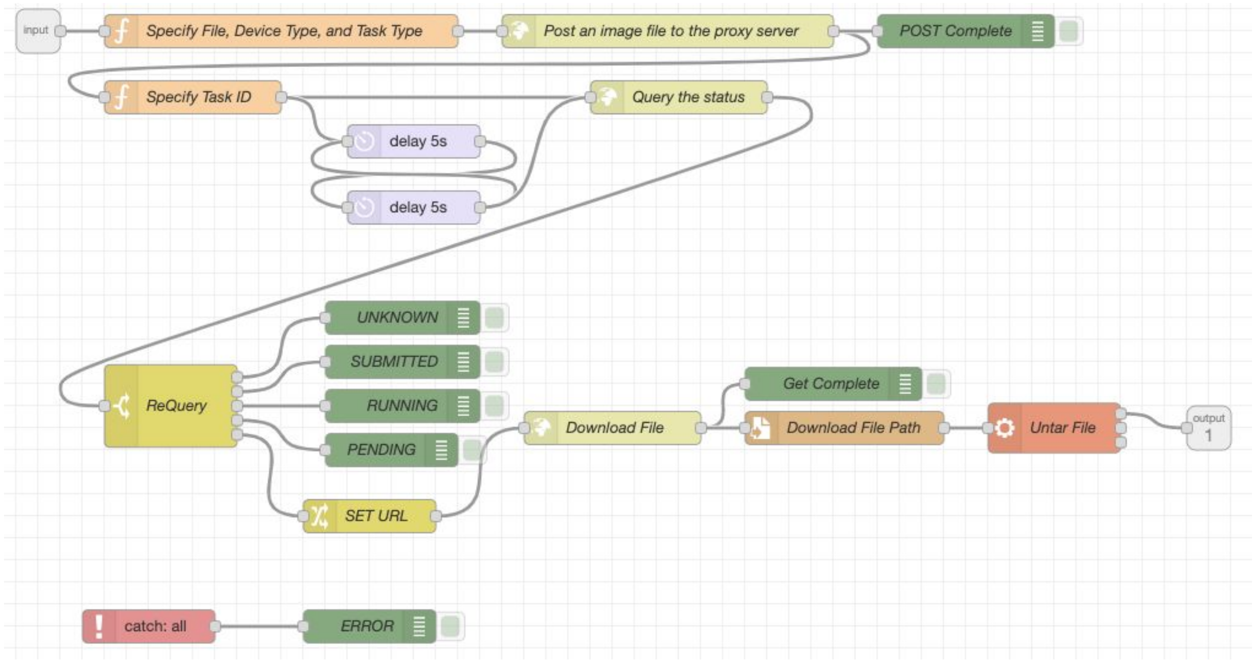
Naveen Nathan
Dyshant Patel

NetID: nnathan2, dpate216
Lab Section: IL4

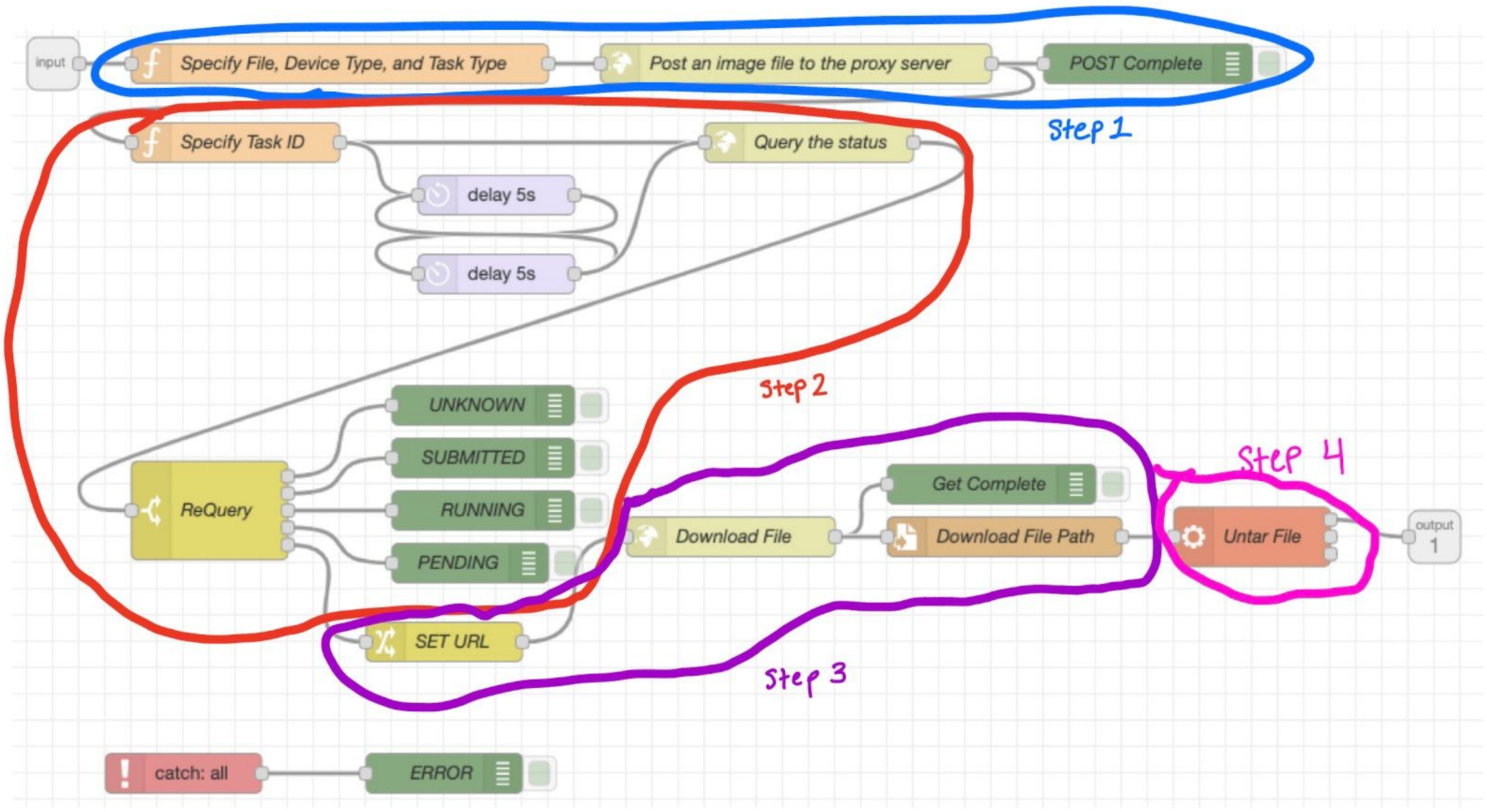
March 25th, 2020

4.2.1 Part 1 [5 pts]

[1 pts] Properly label all nodes within the Remote GPU service(by changing the name fields in their properties panel) and include a screenshot of the Remote GPU service subflow.



[2 pts] According to steps listed in the implementation checklist, divide nodes within the Remote GPU service into groups for achieving different goals and comment on each of these groups to justify that your design meets our requirements.



Step 1: Uses one function node, one http request node, and a debug node. The function node is used to specify the multipart/form-data data type and the required fields for a post request. The payload was specified to use Jetson TX2 as the device type and skynet as the task type as told in the lab documentation. The POST request was made with the http request node. This is where we placed the encoded token for nnathan2 (for bearer authentication) and URL. The debug node was used to make sure the post was successful and a taskID is returned. 1

Step 2: This step uses one function node, one http request node, two delays, a switch node, and four debug nodes. The “Specify Task ID” node places the taskID from the previous POST request into a payload to be processed in the GET request to determine the current status of the job. Once the GET has a response it compares it to the “ReQuery” node that checks what the response is and then is connected to its debug node. This allows us to see the response in the debug tab. The delays are used to continuously query the status every 10 seconds. The reason we use two delays is to preserve the taskID between queries.

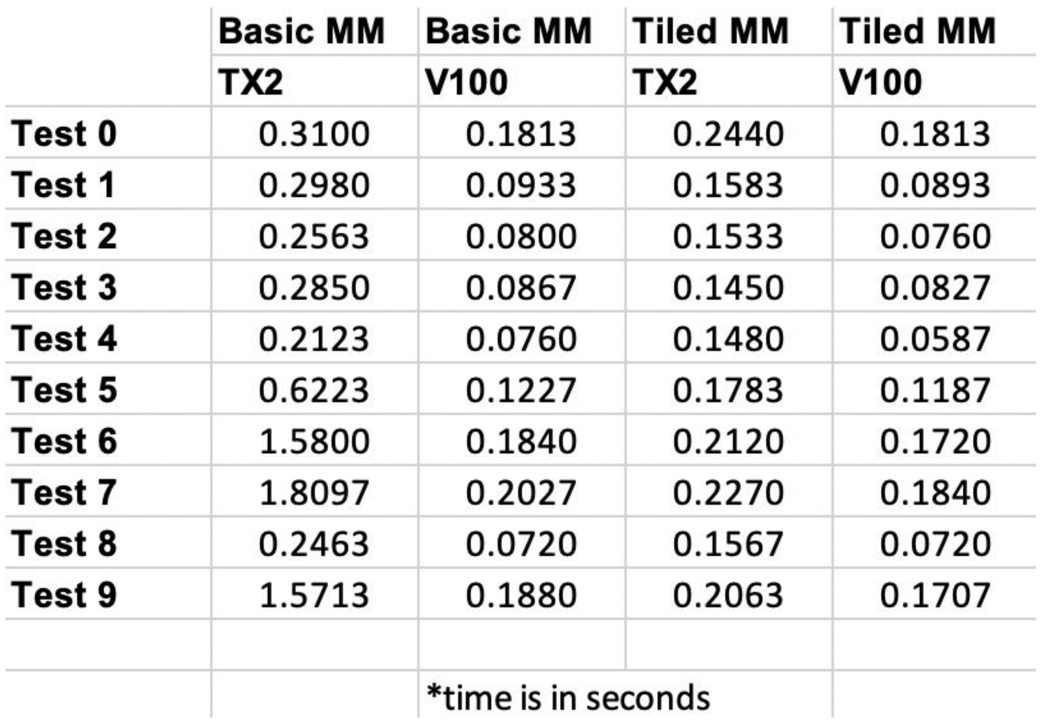
Step 3: This step uses one http request node, one file node, and one change node. The change node takes the URL that we receive from step 2 and puts it into the payload for the GET request made to get the zipped file. The GET node does not have any fields since the URL is contained in the payload and returns a binary buffer. Using the file node we set the absolute path we want the .tar.gz file to be downloaded.

Step 4: The exec node is used to unzip the .tar file we downloaded from step 3. Using the command “tar -xvf (file path) -C (file path)” we unzip the file specified and move it to a given directory.

[2 pts] Describe at least three challenges you encountered during the development of the Node-RED flow in part1 and how you resolve them.

1. No file found error - we were not providing the absolute path to the file.
2. Continuously getting UNKNOWN after requerying - we realized that we were not preserving the taskID so the payload fed into the GET request was always incorrect.
3. The second GET request was not receiving a URL - we realized that we needed a change node to set the msg.url to the msg.payload and we couldn’t just append msg.payload to the query string parameters.

4.2.2 Part 2 [5 pts]
[1 pts] Collect and report the execution time of all datasets with a table, for basic and tiled MM on both V100 and TX2.



	Basic MM	Basic MM	Tiled MM	Tiled MM
	TX2	V100	TX2	V100
Test 0	0.3100	0.1813	0.2440	0.1813
Test 1	0.2980	0.0933	0.1583	0.0893
Test 2	0.2563	0.0800	0.1533	0.0760
Test 3	0.2850	0.0867	0.1450	0.0827
Test 4	0.2123	0.0760	0.1480	0.0587
Test 5	0.6223	0.1227	0.1783	0.1187
Test 6	1.5800	0.1840	0.2120	0.1720
Test 7	1.8097	0.2027	0.2270	0.1840
Test 8	0.2463	0.0720	0.1567	0.0720
Test 9	1.5713	0.1880	0.2063	0.1707

*time is in seconds

[2 pts] Compare the execution time and explain how tiled MM can reduce global memory access. Calculate and report the maximum theoretical reduction of global memory access with your implementation.

In comparing the execution times, we find that the Tiled MM is generally faster than the Basic MM and the V100 is generally faster than the TX2. Tiled MM reduces global memory access by taking advantage of shared memory. For each tile, Tiled MM copies the tile from the global memory to the shared memory and then performs its calculations. The reason shared memory usage is advantageous is because it only takes around 5 cycles to access shared memory while it takes around 500 cycles to access global memory. Although copying data to and from the shared memory will take a certain amount of cycles, the number of cycles will still be less than if the global memory was accessed every time for every calculation.

No Tiled:
(Total Cells in Result Matrix)*(global memory accesses)* =
500 x (Total Cells in Result Matrix) cycles
Tiled:
(Total Cells in Result Matrix / Tile Width)*(global memory accesses)* =
500 x (Total Cells in Result Matrix / (32 x 32)) cycles

Maximum Theoretical Reduction: $\frac{500\ x\ Total\ Cells}{\frac{500\ x\ Total\ Cells}{32\ x\ 32}}$ = 32 x 32 = **1024 → Tile Size**

[2 pts] Compare the hardware specifications of V100 and TX2 (e.g. Number of SMs, etc.) and briefly explain the factors that cause the performance difference.

	V100	**TX2**
Memory/GPU	16GB	8GB
CUDA Cores	5120	256
Memory Bandwidth	900 GB/sec	58.3 GB/sec
Number of SMs	80	2

As shown in the table above, the V100 has a larger memory, more CUDA cores, a larger memory bandwidth, and more SMs, than the TX2. With more cores and SMs, the V100 is able to perform more tasks concurrently. Additionally, the higher memory and memory bandwidth allow the V100 to process larger amounts of instructions/data. Essentially, the V100 has a significantly higher throughput than the TX2, which is why the V100 performs better as seen in the matrix multiplication trial data above. However, the V100 doesn’t always perform faster than the TX2 in some cases. This can be because the memory transfer takes longer with the V100 than with the TX2 due to design specifications. Another reason could be that the full resources of the V100 are not being taken advantage of.