Parallel Multi-layer Feed-Forward Neural Network using Python

School on Parallel Programming and Parallel Architecture for High Performance Computing

An ICTP 60th Anniversary Satellite Event

Roman M. Richard

Computer Engineering, Technological Institute of the Philippines - Quezon City

May 2, 2024

What is a Feed-Forward Neural Network

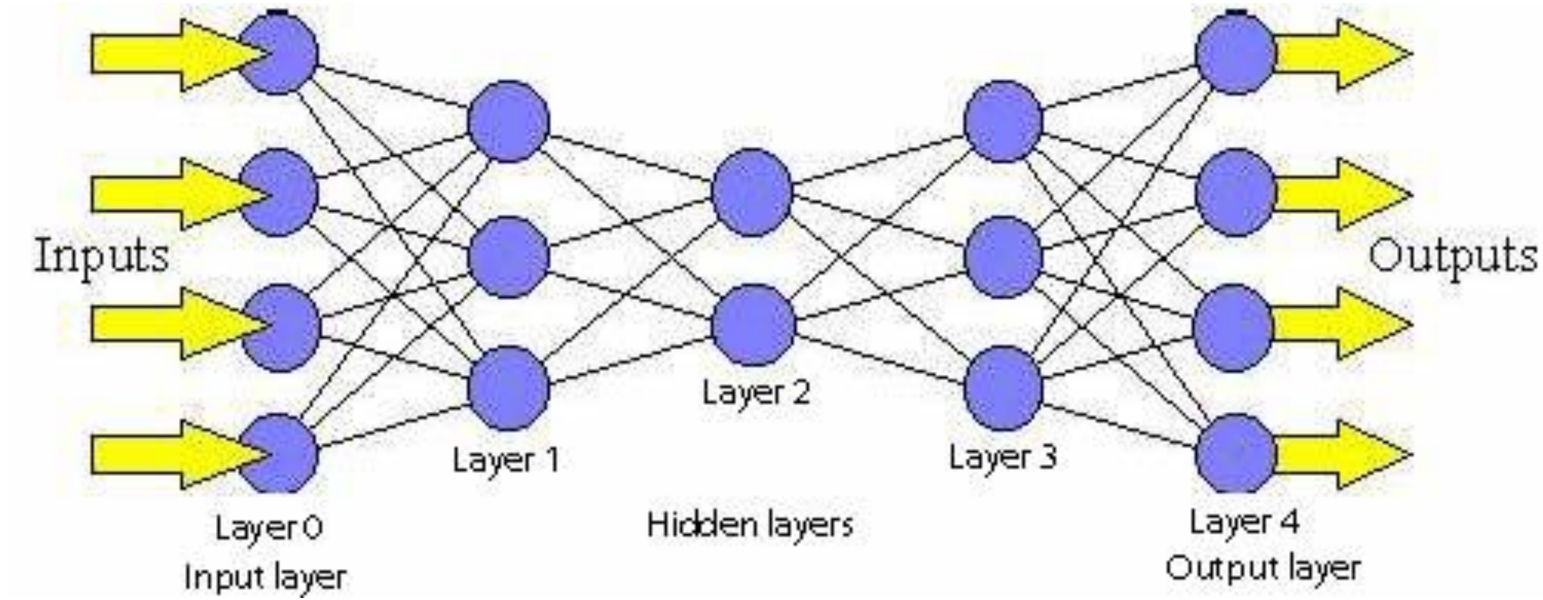
Feed-forward networks have the following characteristics:

1. Perceptrons are arranged in layers.
2. Each perceptron in one layer is connected to every perceptron on the next layer. Hence information is constantly "fed forward" from one layer to the next.
3. There is no connection among perceptrons in the same layer.

Backpropagation is not part of this project yet.

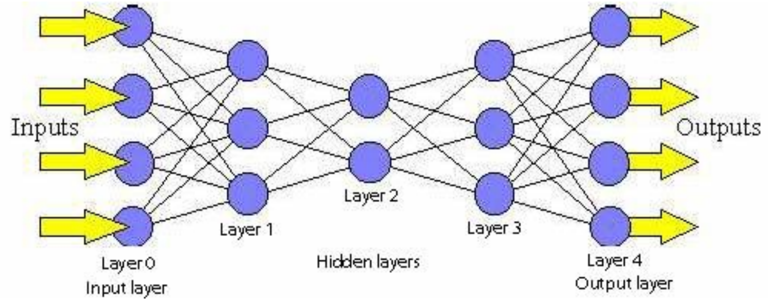
Source: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Architecture/feedforward.html

Multi-layer Feed-forward Neural Network

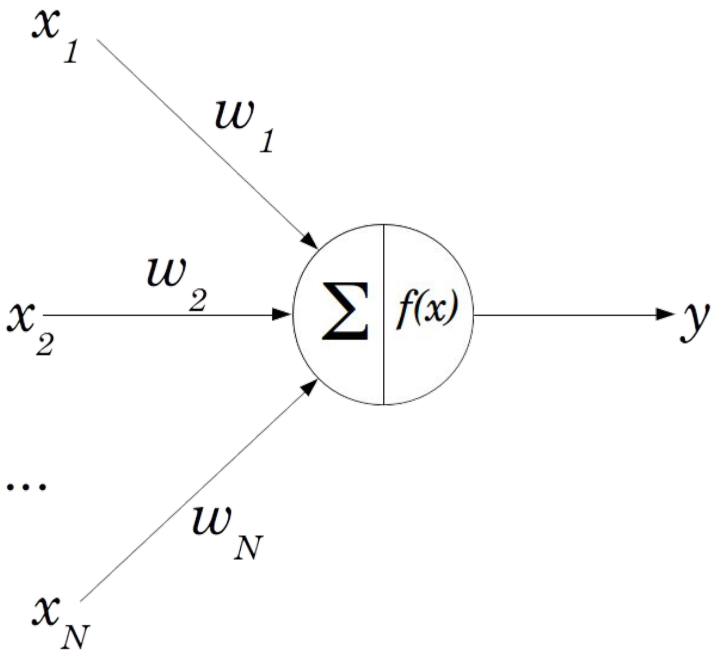


https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Architecture/feedforward.html

Multi-layer Feed-forward Neural Network

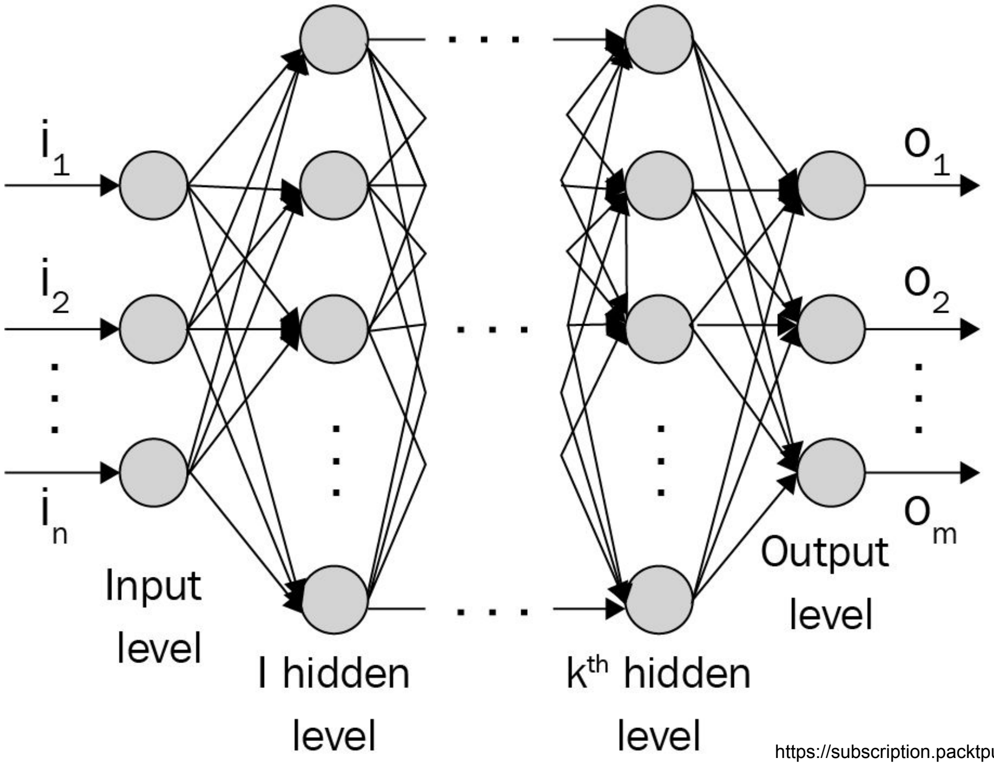


https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Architecture/feedforward.html



https://lucidar.me/en/neural-networks/perceptron/

Parallelizable? Yes!



https://subscription.packtpub.com/book/data/9781789346640/1/ch01lvl1sec03/architecture-of-a-simple-neural-network

Multi-layer Feed-forward Neural Network

Steps taken:

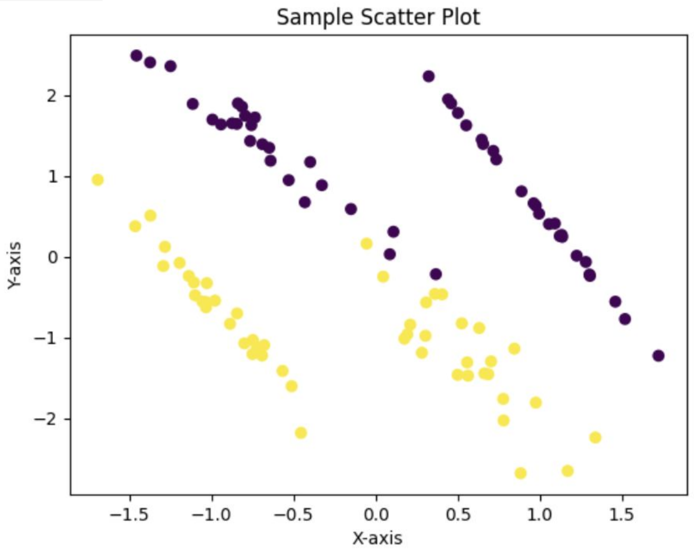
1. Prepare the data
2. Create CPU code
3. Identify what can be parallelized
4. Use GPU tools/techniques taught
5. Compare / Interpret Results

Data Preparation

Data Preparation

```
# Test on dummy data
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification

# Generate data with 10 features, 1000 samples, and 2 classes with informative features
X, y = make_classification(n_samples=no_samples, n_features=no_features, n_classes=no_classes, random_state=42)

# X is the data (features), y are the labels (class assignments)
print(X.shape)  # Output: (100, 4) - 100 samples with 4 features each
print(y.shape)  # Output: (100,) - 100 labels (0 or 1)

# Create the scatter plot
plt.scatter(X[:, 0], X[:, 1], c=y)  # Using first two columns of X for x and y axes

# Add labels and title
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Sample Scatter Plot")

# Display the plot
plt.show()
```

```
(100, 5)
(100,)
```



Various sample sizes were used to test the performance.

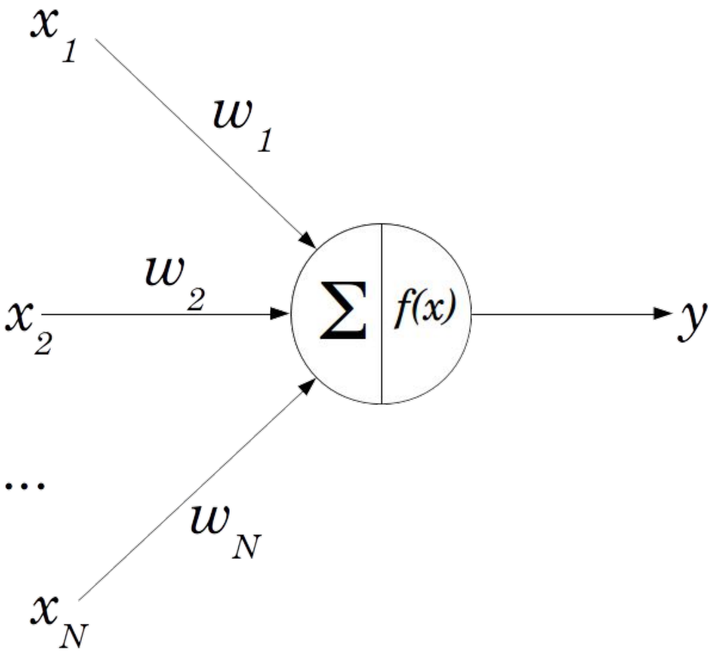
CPU Code

Both Object-Oriented vs Procedural approaches were considered. Ended up with procedural code instead.

```python
import numpy as np
import pandas as pd
import math

# The input layer
# Load and ready the data to be passed to the hidden layer
def cpu_inputLayer(inputVec):
  vecToReturn = np.array(inputVec)
  return vecToReturn
```

```python
def cpu_activationFunc(inputVector, activation = ""):
  activation = activation.lower()

  for i in inputVector:
    if activation == "sigmoid":
      i = (1 / 1 + np.exp(-1 * i))
    elif activation == "relu":
      i = np.max([0.01 * i, i])
    elif activation == "linear":
      i = i # No changes

  return inputVector

def cpu_outputLayer(prevLayer):
  outputLayer = cpu_hiddenLayer(no_classes, prevLayer)
  outputLayer = cpu_activationFunc(outputLayer, "sigmoid")

  return max(outputLayer), np.argmax(outputLayer)
```

Remember the Neuron



Implementation on CuPy

Using:

1. cupy.array()
2. Reduction Kernel
3. cupy.fuse()

… and other cupy functions.

```
nodeSum_kernel = cp.ReductionKernel(
    'T x, U y',  # input params
    'T z',  # output params
    'x * y',  # map
    'a + b',  # reduce
    'z = a',  # post-reduction map
    '0',  # identity value
    'nodeSum_kernel'  # kernel name
)
```

Implementation on CuPy

Using:

1. cupy.array()
2. Reduction Kernel
3. cupy.fuse()

```
nodeSum_kernel = cp.ReductionKernel(
    'T x, U y',  # input params
    'T z',  # output params
    'x * y',  # map
    'a + b',  # reduce
    'z = a',  # post-reduction map
    '0',  # identity value
    'nodeSum_kernel'  # kernel name
)
```

```
# Part of this is parallelizable
def cupy_hiddenLayer(numNodes, inputLayer):
  # Accept the number of nodes for this hidden layer
  # Accepts the previous layer
  layerToReturn = cp.zeros(no_features)
  bias = cp.random.random()

  # Per node, compute the product of the inputLayer and the randomized inputWeights
  for i in range(no_features):
    weights = cp.random.random(no_features)
    #layerToReturn[i] = cp.dot(inputLayer, weights) + bias
    layerToReturn[i] = nodeSum_kernel(inputLayer, weights) + bias
    # Add the bias term to the node's sum

  # This layer outputs a vector containing the individual values per node
  # That must be passed to the individual node's activation function
  # Note: This function must be called multiple times if more layers are generated
  return layerToReturn
```

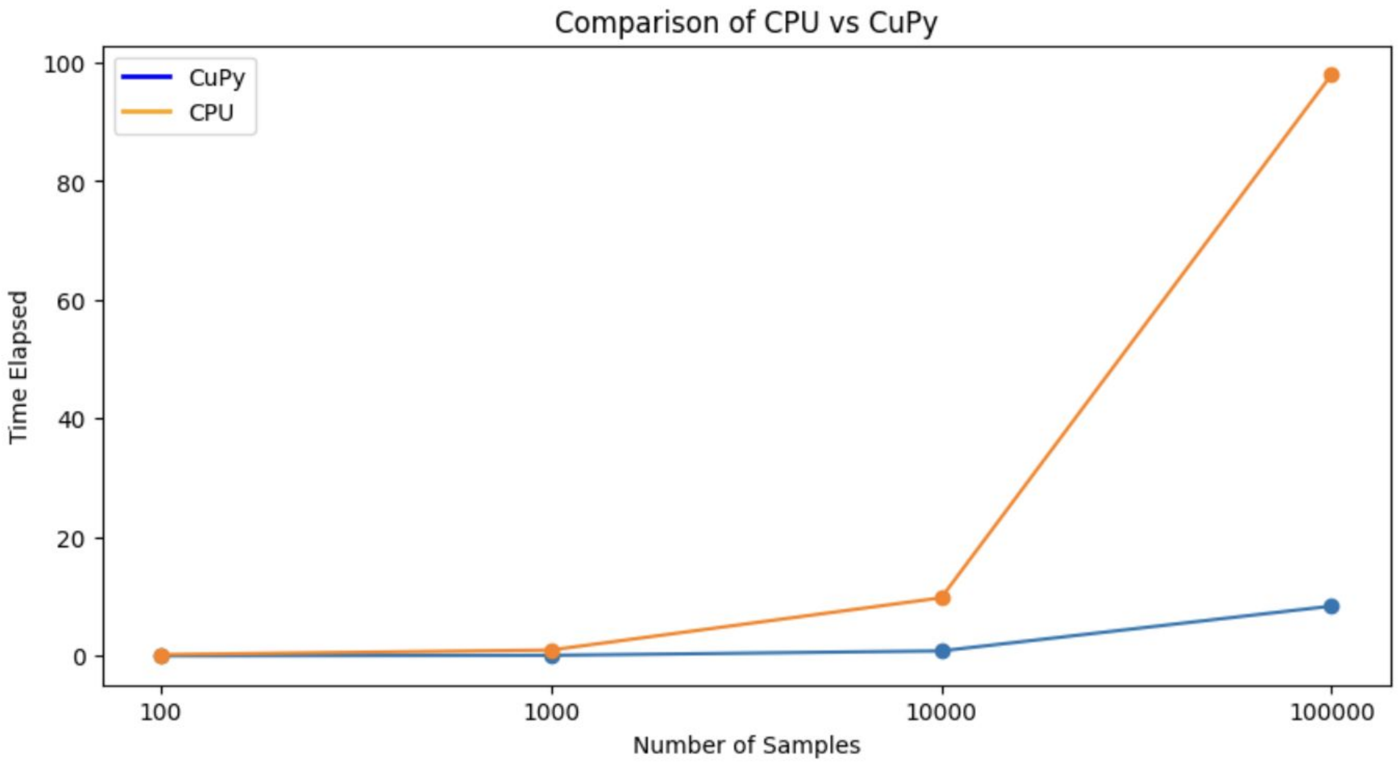
Implementation on CuPy

Using:

1. cupy.array()
2. Reduction Kernel
3. cupy.fuse()

```python
# Parallelized to be an element-wise copy
@cp.fuse(kernel_name='cupy_activationFunc')
def cupy_activationFunc(inputVector, outputVector, activation):

  def apply_activation(x):
    if activation == 1:
      return cp.reciprocal(1 + cp.exp(-1 * x))
    elif activation == 2:
      return cp.maximum(0.01 * x, x)
    elif activation == 3:
      return x
    else:
      raise ValueError("Invalid activation function")

  outputVector = apply_activation(inputVector)
```

Performance Comparison



Numba Hidden Layer Implementation

```
# Part of this is parallelizable
def numba_hiddenLayer(numNodes, inputLayer):
  # Accept the number of nodes for this hidden layer
  # Accepts the previous layer
  layerToReturn = cp.zeros(numNodes)
  bias = cp.random.random()

  # Per node, compute the product of the inputLayer and the randomized inputWeights
  """for i in range(no_features):
    weights = cp.random.random(no_features)
    #layerToReturn[i] = cp.dot(inputLayer, weights) + bias
    layerToReturn[i] = nodeSum_kernel(inputLayer, weights) + bias
    # Add the bias term to the node's sum"""

  TPB = 512

  @cuda.jit
  def numba_nodeSum(inputVec, weightVec, nodeToSum):
      # Define an array in the shared memory
      # The size and type of the arrays must be known at compile time
      sInputVec = cuda.shared.array(shape=(TPB), dtype=float32)
      sWeightVec = cuda.shared.array(shape=(TPB), dtype=float32)
```

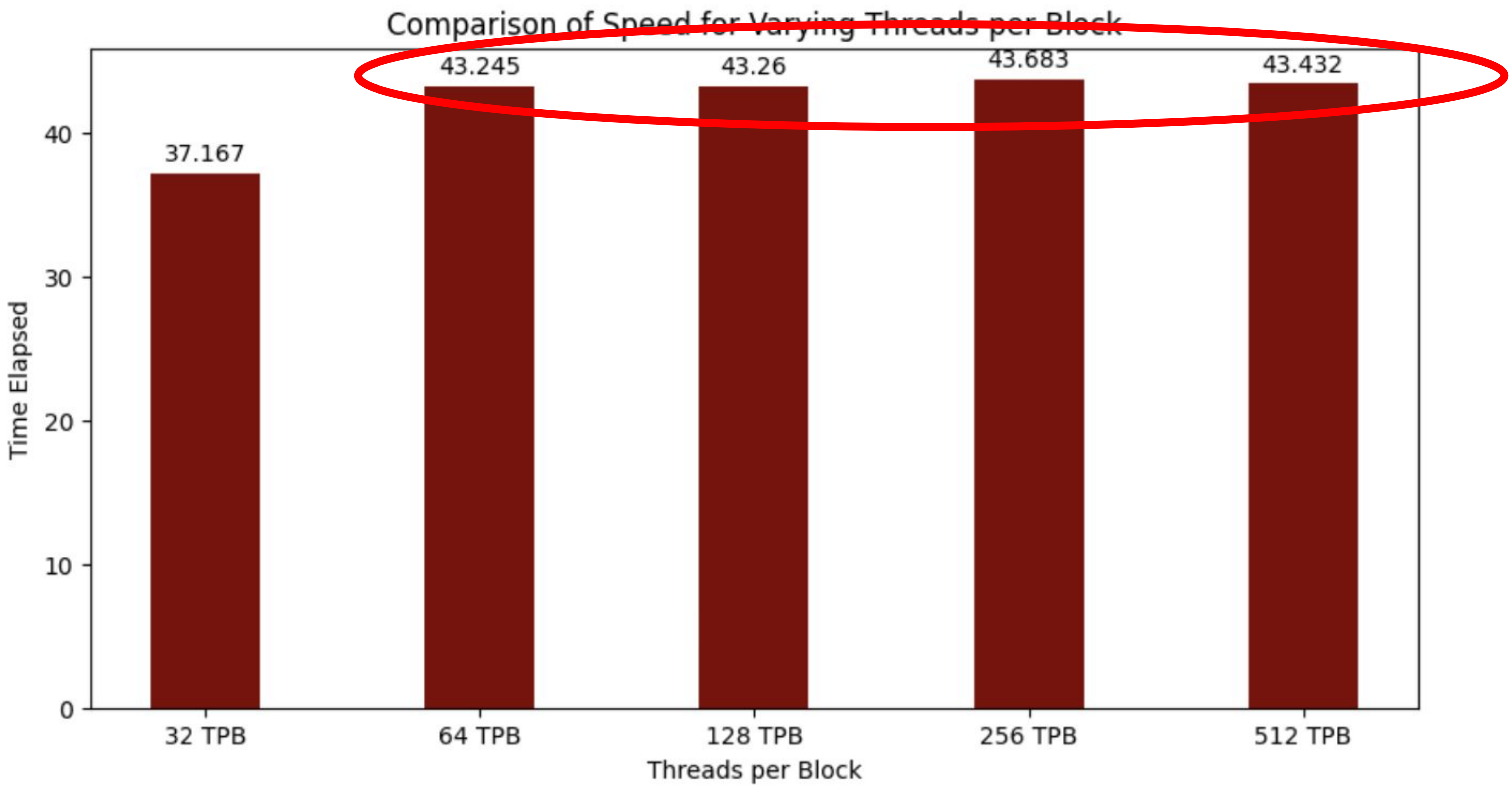
```
tx = cuda.threadIdx.x

if x >= nodeToSum.shape[0]:
    # Quit if (x, y) is outside of valid C boundary
    return

# Each thread computes one element in the result matrix.
# The dot product is chunked into dot products of TPB-long vectors.
tmp = 0.
for i in range(int(inputVec.shape[0] / TPB)):
    # Preload data into shared memory
    sInputVec[tx] = inputVec[tx + i * TPB]
    sWeightVec[tx] = weightVec[tx + i * TPB]

    # Wait until all threads finish preloading
    cuda.syncthreads()

    # Computes partial product on the shared memory
    for j in range(TPB):
        tmp += sInputVec[j] * sWeightVec[j]

  # Wait until all threads finish computing
    cuda.syncthreads()

nodeToSum[x] = tmp
# Function Ends here
```

```python
# The Data Arrays
inputVec = inputLayer
weightVec = np.random.random(len(inputVec))

# Shared Memory for Both Arrays
A_global_mem = cuda.to_device(inputVec)
B_global_mem = cuda.to_device(weightVec)
C_global_mem = cuda.device_array(len(inputVec))

# Configure the blocks
threadsperblock = (TPB)
blockspergrid_x = int(math.ceil(inputLayer.shape[0] / threadsperblock))
blockspergrid_y = int(math.ceil(weightVec.shape[0] / threadsperblock))
blockspergrid = (blockspergrid_x, blockspergrid_y)

for i in prange(numNodes):
    numba_nodeSum[blockspergrid, threadsperblock](A_global_mem, B_global_mem, C_global_mem)
    res = C_global_mem.copy_to_host()
    layerToReturn[i] = np.sum(res) + bias

# This layer outputs a vector containing the individual values per node
# That must be passed to the individual node's activation function
# Note: This function must be called multiple times if more layers are generated
return layerToReturn.get()
```

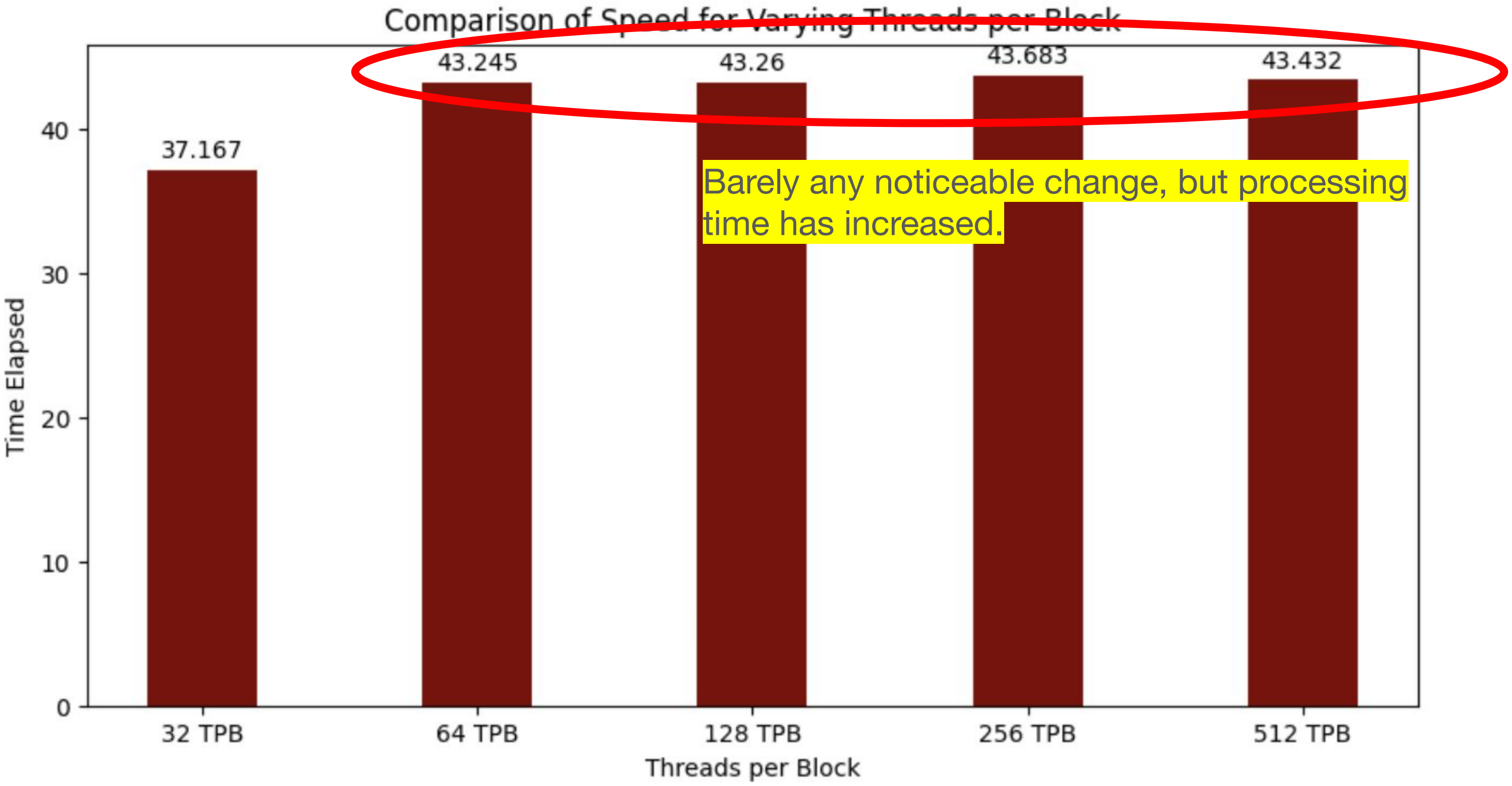
```
def numba_activationFun(inputVector, outputVector, activation):
    #tid = numba.cuda.threadIdx.x
    for i in range(len(inputVector)):
        if activation == 1:
          outputVector[i] = np.reciprocal(1 + np.exp(-1 * inputVector[i]))
        elif activation == 2:
          outputVector[i] = np.maximum(0.01 * inputVector[i], inputVector[i])
        elif activation == 3:
          outputVector[i] = inputVector[i]
        else:
          raise ValueError("Invalid activation function")

def numba_outputLayer(prevLayer):
  outputLayer_y = np.zeros(no_classes)

  outputLayer = numba_hiddenLayer(no_classes, prevLayer)

  numba_activationFun(outputLayer, outputLayer_y, 1)

  return np.max(outputLayer), np.argmax(outputLayer)
```

Numba

Neural Network implemented using Numba had 1,000 samples throughout with different TPB values.

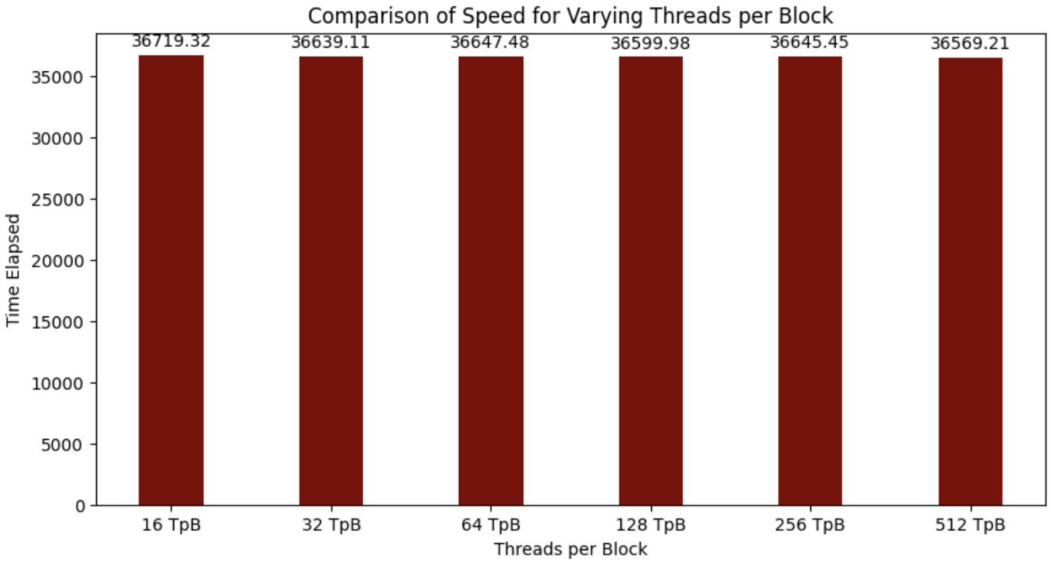


Neural Network implemented using Numba had 1,000 samples throughout with different TPB values.



Areas for Improvement

There are already some improvements! **Initial parallelization attempts using CuPy that gave very different results with similar implementation.**



Remember the bottleneck when moving data to and from with something like Python.

Thank you!