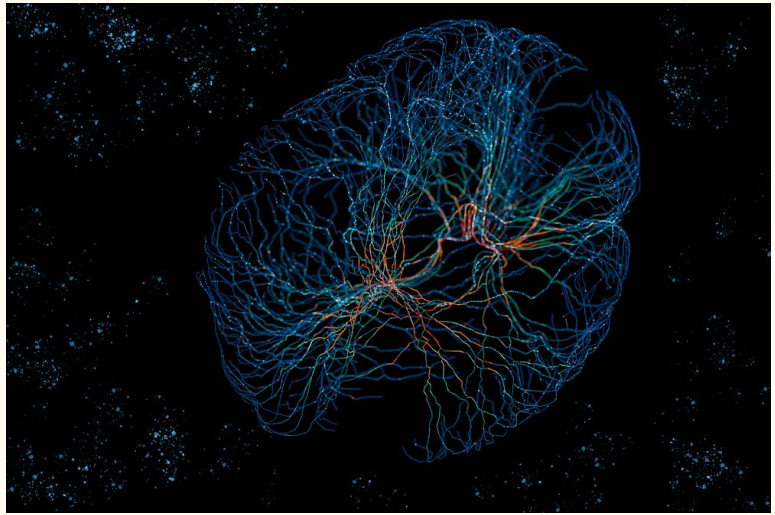
Parallel Deep Learning Neural Network: Final Presentation

—

Thomas Keller, Zack Humphries, Anuradha Agarwal

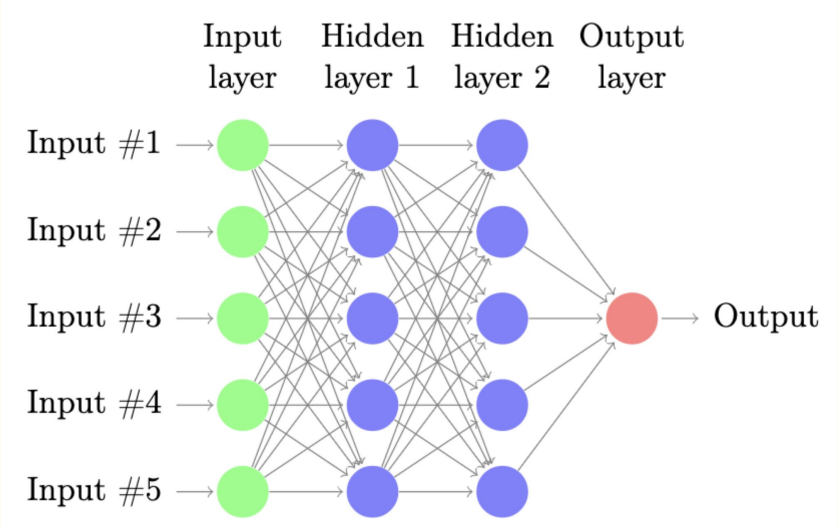
Outline

- ➔ Introduction
- ➔ Literature Review
- ➔ OpenMP
- ➔ CUDA
- ➔ Conclusion

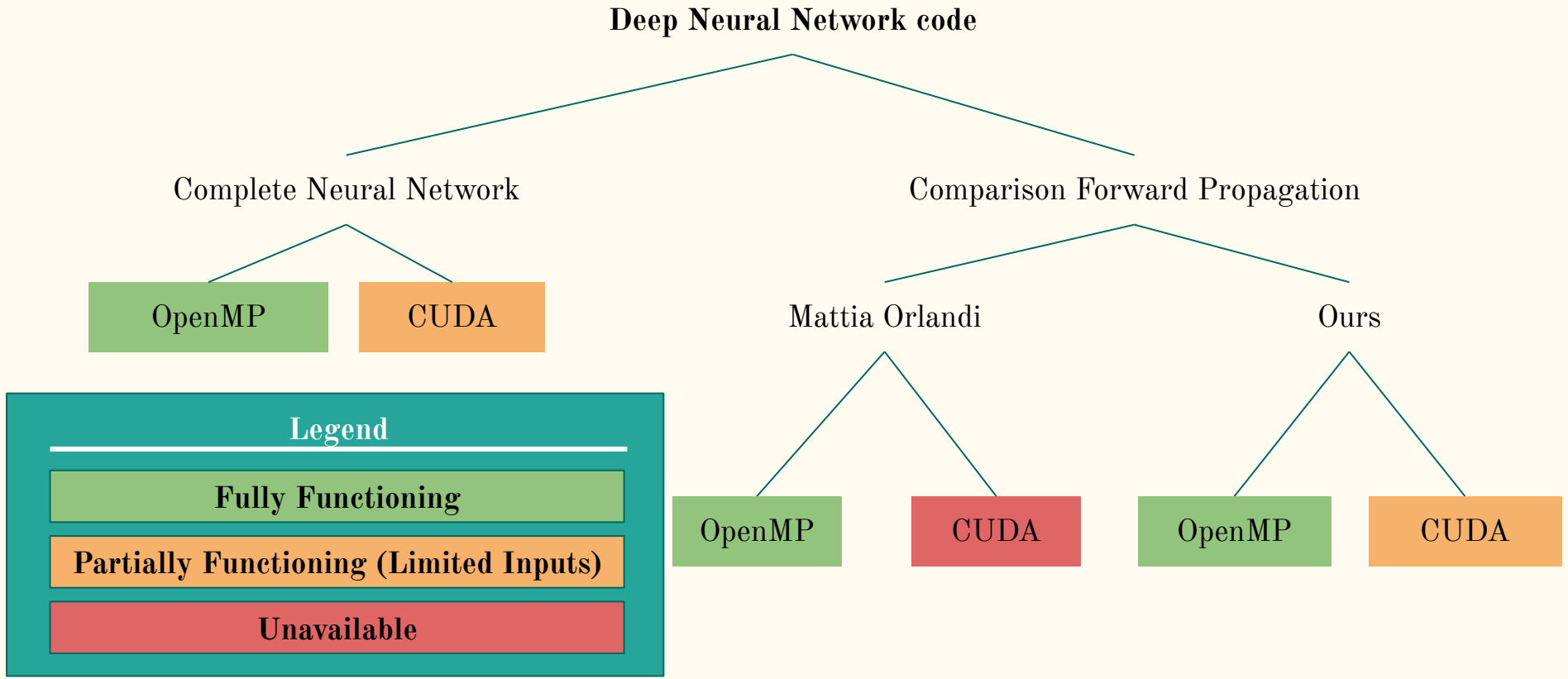


Introduction

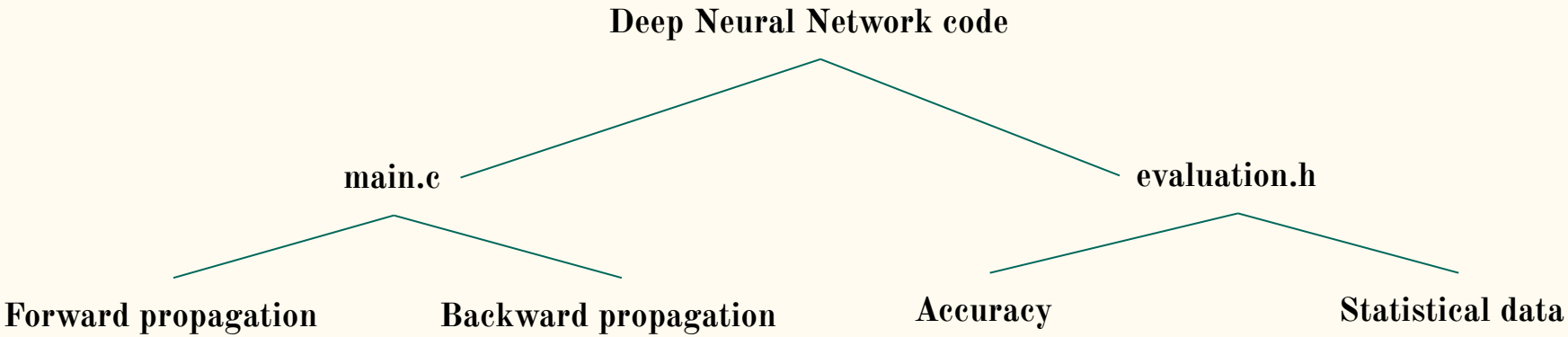
- Breast Cancer Data Set
- Deep Neural Network - serial code
  - Binary classification
  - 2 Hidden Layers
  - Output Layer with 1 node
  - 30/4800 inputs
- Parallelizing:
  - OpenMP
  - CUDA



Literature Review: Exploring Comparison Code



Serial Code and Accuracy Evaluation



Accuracy and Confusion matrix

- All metrics are determined by the test data set with 114 test cases.
- 30 inputs, 2 hidden layers
- Epochs: 1500
- Learning rate: 0.001
- 98.25% accuracy

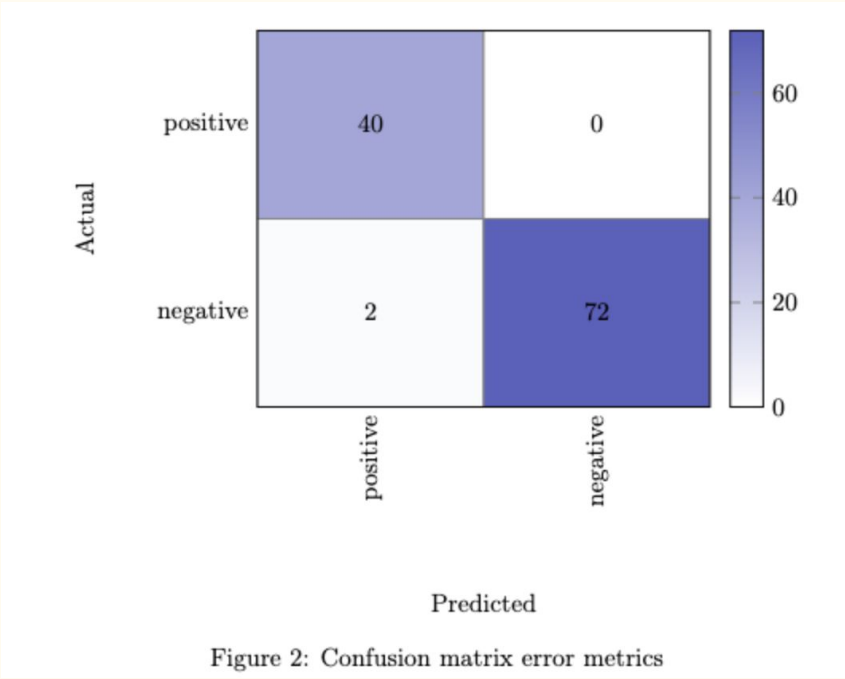


Figure 2: Confusion matrix error metrics

OpenMP Code: Forward

```
    // forward pass
    // compute hidden layer activation

    // hidden layer 1
    #pragma omp parallel for num_threads(thread_count)
    for(int j =0; j < numHiddenNodes; j++){
        double activation = hiddenLayerBias[j];

        for(int k = 0; k < numInputs; k++){
            activation += trainingInputs[i][k] * hiddenWeights[k][j];
        }

        hiddenLayer[j] = relu(activation);
    }

    // hidden layer 2
    double activation = 0;
    #pragma omp parallel for reduction(+:activation) num_threads(thread_count)
    for(int j =0; j < numHiddenNodes2; j++){
        activation = hiddenLayerBias2[j];

        for(int k = 0; k < numHiddenNodes; k++){
            activation += hiddenLayer[k] * hiddenWeights2[k][j];
        }

        hiddenLayer2[j] = relu(activation);
    }

    // compute output layer activation
    #pragma omp parallel for reduction(+:activation) num_threads(thread_count)
    for(int j =0; j < numOutputs; j++){
        activation = outputLayerBias[j];

        for(int k = 0; k < numHiddenNodes2; k++){
            activation += hiddenLayer2[k] * outputWeights[k][j];
        }

        outputLayer[j] = sigmoid(activation);
    }
```

Listing 3: Forward path openMP of main.c

- The first loop computes the activation of the neurons in the first hidden layer.
- Each thread executes the loop body for a subset of the iterations.
- The second loop computes the activation of the neurons in the second hidden layer.
- The third loop computes the activation of the output layer neurons
- In addition, the reduction clause is used to sum up the values of the activation variable across threads.

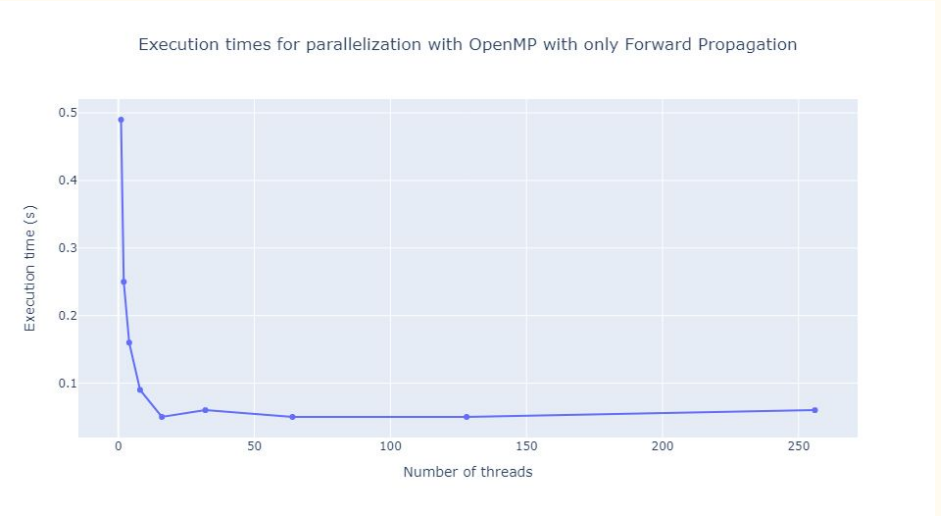
OpenMP Code: Backward

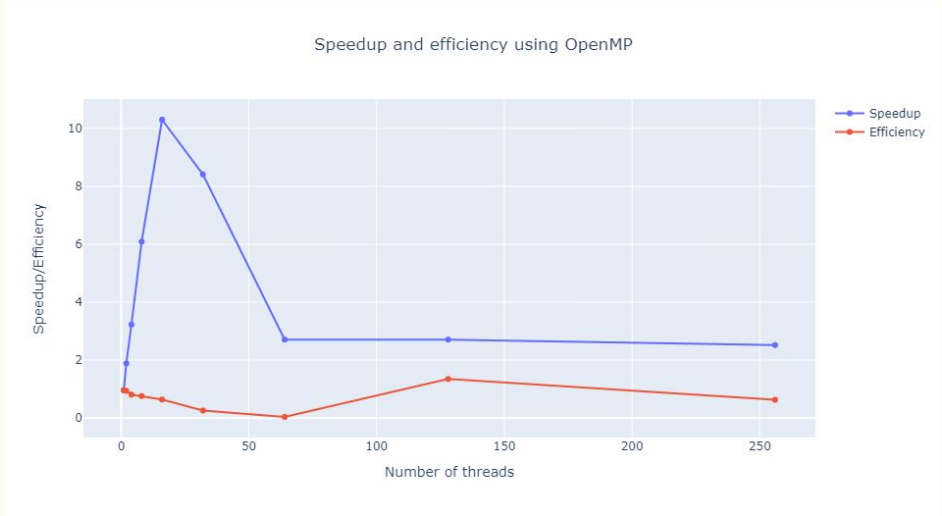
```
// Backpropagation
// Compute change in output weights
double deltaOutput[numOutputs];
#pragma omp parallel for num_threads(thread_count)
for(int j = 0; j < numOutputs; j++){
    double error = (trainingOutputs[i][j] - outputLayer[j]); // L1
    deltaOutput[j] = error * dSigmoid(outputLayer[j]) ;
}

// Compute change in hidden weights (second layer)
double deltaHidden2[numHiddenNodes2];
#pragma omp parallel for num_threads(thread_count)
for(int j = 0; j < numHiddenNodes2; j++){
    double error = 0.0f;
    for(int k = 0; k < numOutputs; k++){
        error += deltaOutput[k] * outputWeights[j][k];
    }
    deltaHidden2[j] = error * dRelu(hiddenLayer[j]);
}

// Compute change in hidden weights (first layer)
double deltaHidden[numHiddenNodes];
#pragma omp parallel for num_threads(thread_count)
for(int j = 0; j < numHiddenNodes; j++){
    double error = 0.0f;
    for(int k = 0; k < numHiddenNodes2; k++){
        error += deltaHidden2[k] * hiddenWeights2[j][k];
    }
    deltaHidden[j] = error * dRelu(hiddenLayer2[j]);
}

// Apply change in output weights
#pragma omp parallel for num_threads(thread_count)
for(int j = 0; j < numOutputs; j++){
    outputLayerBias[j] += deltaOutput[j] * lr;
    for(int k = 0; k < numHiddenNodes2; k++){
        outputWeights[k][j] += hiddenLayer2[k] * deltaOutput[j] * lr;
    }
}

// Apply change in second hidden layer weights
#pragma omp parallel for num_threads(thread_count)
for(int j = 0; j < numHiddenNodes2; j++){
    hiddenLayerBias[j] += deltaHidden[j] * lr;
    for(int k = 0; k < numHiddenNodes; k++){
        hiddenWeights2[k][j] += hiddenLayer[k] * deltaHidden2[j] * lr;
    }
}

// Apply change in first hidden layer weights
#pragma omp parallel for num_threads(thread_count)
for(int j = 0; j < numHiddenNodes; j++){
    hiddenLayerBias[j] += deltaHidden[j] * lr;
    for(int k = 0; k < numInputs; k++){
        hiddenWeights[k][j] += trainingInputs[i][k] * deltaHidden[j] * lr;
    }
}
```

Listing 4: Backward path openMP of main.c

- The code starts by computing the change in output weights using the training inputs and outputs.
- Next, the code computes the change in hidden weights for the second layer and then the first layer.
- After computing the changes in the weights, the code applies them to the network by updating the weights and biases.

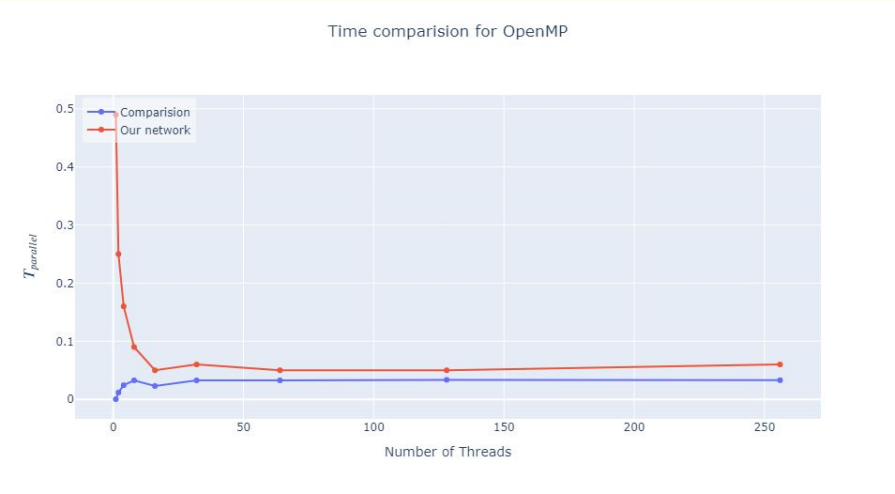
Time and Speedup (OpenMP)

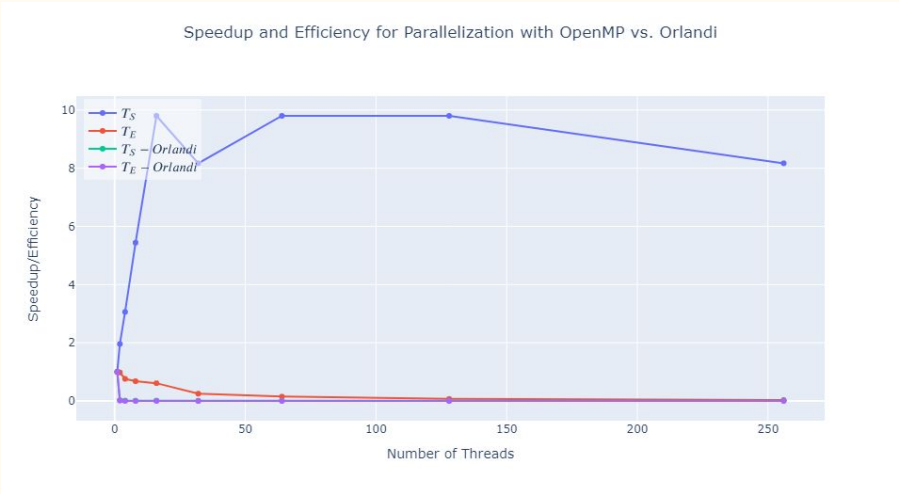




- The parallel execution times decrease as the number of threads increases, indicating that the parallelization is effective. However, the execution time does not continue to decrease at the same rate as the number of threads increases. The speedup increases as the number of processors increases, but with diminishing returns.

Forward Propagation Comparison (Orlandi)





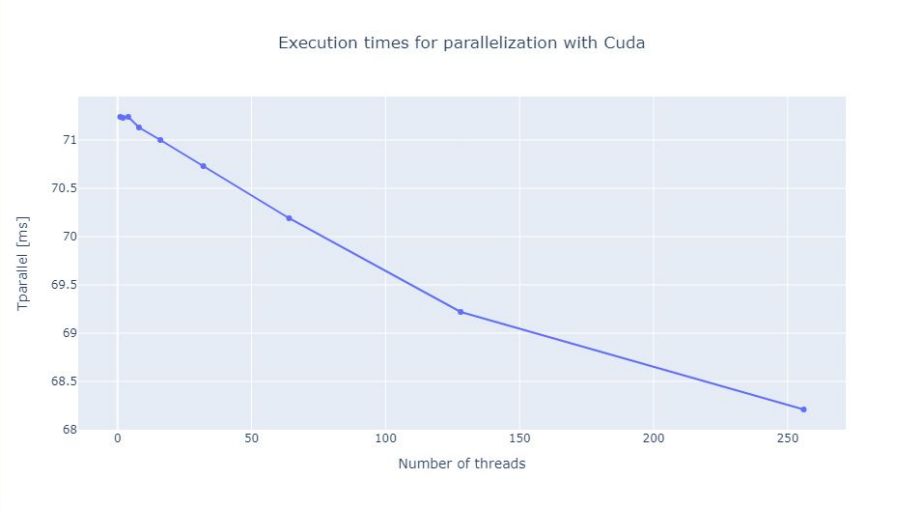
- Our DNN takes longer to train compared to Orlandi's; nevertheless, we are able to parallelize the process using OpenMP efficiently. We are able to reach higher speedups, and our code is more efficient comparatively.

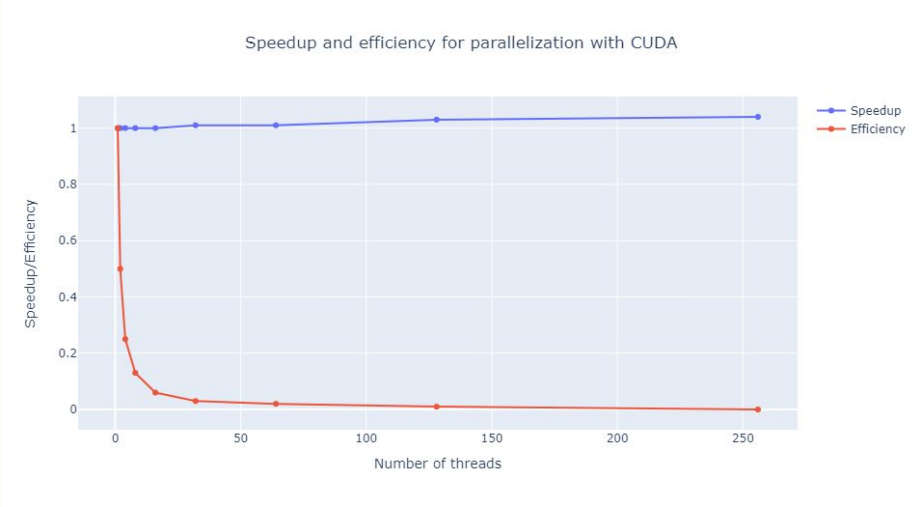
CUDA Code Forward and Backward

```cuda
// Forward propagation kernel
__global__ void forward_kernel(double* X, double* W1, double* W2, double* W3, double* b1,
    double* b2, double* b3, double* hidden1, double* hidden2, double* output) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < HIDDEN_SIZE) {
        double sum = 0.0;
        for (int j = 0; j < INPUT_SIZE; j++) {
            sum += X[j] * W1[j * HIDDEN_SIZE + tid];
        }
        hidden1[tid] = relu(sum + b1[tid]);
    }

    __syncthreads();

    if (tid < HIDDEN_SIZE) {
        double sum = 0.0;
        for (int i = 0; i < HIDDEN_SIZE; i++) {
            sum += hidden1[i] * W2[i * HIDDEN_SIZE + tid];
        }
        hidden2[tid] = relu(sum + b2[tid]);
    }

    __syncthreads();

    if (tid == 0) {
        double sum = 0.0;
        for (int i = 0; i < HIDDEN_SIZE; i++) {
            sum += hidden2[i] * W3[i];
        }
        *output = sigmoid(sum + b3[0]);
    }
}
```

```cuda
__global__ void backward_kernel(double* X, double* W1, double* W2, double* W3, double* b1,
    double* b2, double* b3, double* hidden1, double* hidden2, double* output, double target
    ) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid == 0) {
        // calculate output weight
        double d_output = (*output - target) * dSigmoid(*output);

        // calculate hidden 2 weight
        double d_hidden2[HIDDEN_SIZE];
        for (int i = 0; i < HIDDEN_SIZE; i++) {
            d_hidden2[i] = dRelu(hidden2[i]) * W3[i] * d_output;
        }

        // calculate hidden 1 weight
        double d_hidden1[HIDDEN_SIZE];
        for (int i = 0; i < HIDDEN_SIZE; i++) {
            double sum = 0.0;
            for (int j = 0; j < HIDDEN_SIZE; j++) {
                sum += W3[j * HIDDEN_SIZE + i] * d_hidden2[j];
            }
            d_hidden1[i] = dRelu(hidden1[i]) * sum;
        }

        // update hidden weights 1 & bias
        for (int i = 0; i < HIDDEN_SIZE; i++) {
            for (int j = 0; j < INPUT_SIZE; j++) {
                W1[j * HIDDEN_SIZE + i] -= LEARNING_RATE * X[j] * d_hidden1[i];
            }
            b1[i] -= LEARNING_RATE * d_hidden1[i];
        }

        // update hidden weights 2 & bias
        for (int i = 0; i < HIDDEN_SIZE; i++) {
            for (int j = 0; j < HIDDEN_SIZE; j++) {
                W2[j * HIDDEN_SIZE + i] -= LEARNING_RATE * hidden1[i] * d_hidden2[j];
            }
            b2[i] -= LEARNING_RATE * d_hidden2[i];
        }

        // update output weights & bias
        for (int i = 0; i < HIDDEN_SIZE; i++) {
            W3[i] -= LEARNING_RATE * hidden2[i] * d_output;
        }
        b3[0] -= LEARNING_RATE * d_output;
    }
}
```

Time and Speedup (CUDA)





- The execution time remains almost constant, around 71 seconds, with only a slight improvement as the number of processors increases. This indicates that the parallelization is not effective in reducing the overall execution time.
- Note that there our CUDA code could not handle up to 4800 inputs (so limited to 570). Plus, we were unable to compare it to Orlandi's code so these times should be taken with "a grain of salt"

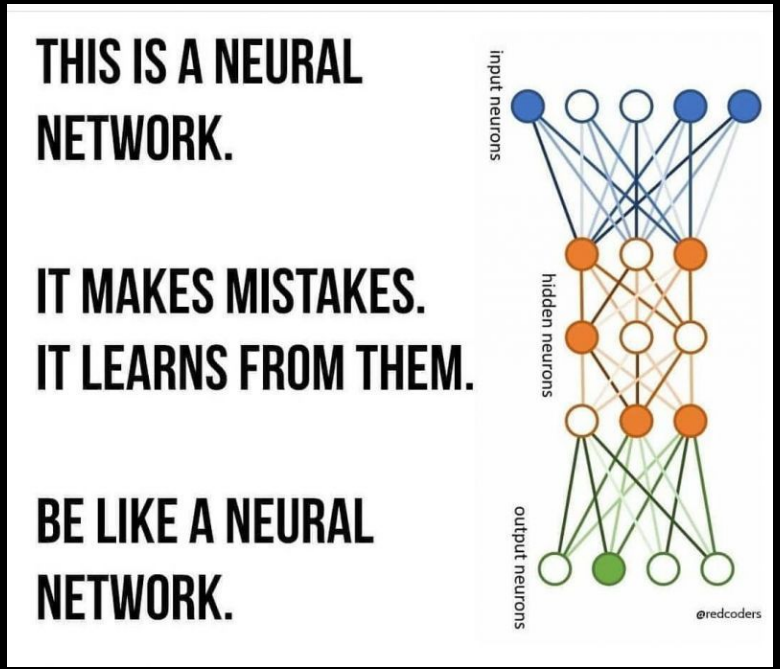
Conclusion

- Forward Propagation
- Backward Propagation
- OpenMP - speedup
  - 8 and 16 Threads
- CUDA - no speedup (570 inputs)
  - Matrix size
  - Lack of knowledge
  - Time

References

- W. Nick Street Dr. William H. Wolberg and Olvi L. Mangasarian, UCI machine learning repository: Breast cancer wisconsin (diagnostic) data set, University of California, Irvine, School of Information and Computer Sciences, 1995.
- Orlandi, Neural network implementation with opemmp and cuda (2020).
- https://github.com/nihil21/parallel_nn/
- https://github.com/thomkell/DeepNeuralNetwork

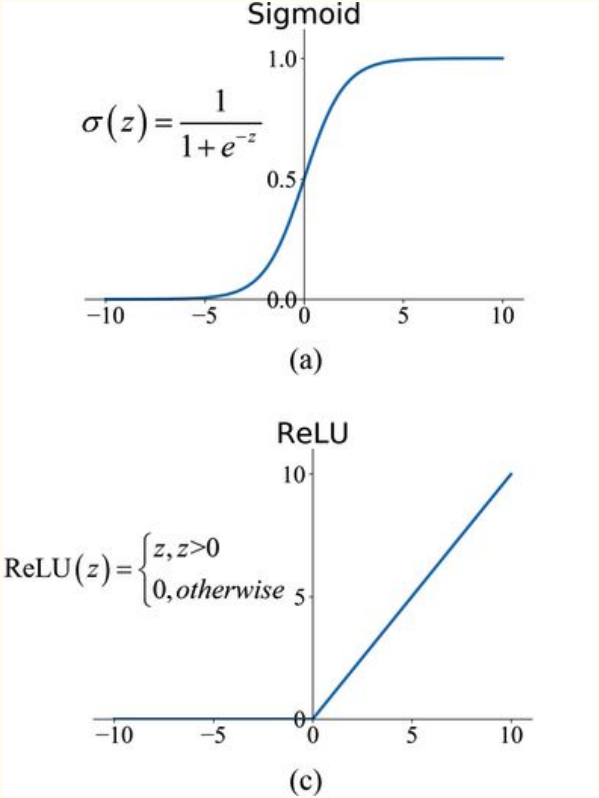
Questions?


THIS IS A NEURAL NETWORK.
IT MAKES MISTAKES.
IT LEARNS FROM THEM.
BE LIKE A NEURAL NETWORK.
input neurons
hidden neurons
output neurons
@redcoders

Problem Statement and Dataset

- Both the forward and backward propagation processes involve matrix multiplication (dot product)
  - Although the results of the previous layer are the inputs for the next layer, the multiplication itself can be parallelized
- We plan to use a publicly available dataset with 569 instances of breast cancer data to predict whether certain attributes of a mass are benign or malignant
  - In order to see the effectiveness of our parallelization, we replicated the data from 30 inputs to 4800 inputs

Backup Slide: Activation Functions



Literature Review: Exploring Comparison Code

- Mattia Orlandi's "Neural Network Implementation with OpenMP and CUDA"
  - Explores the parallelization of only the forward propagation
    - Inner-Loop: Parallelization of the matrix multiplication
      - What we focused on
    - Outer-Loop: Parallelization of each epoch (individual forward propagation)
      - Unrealistic as there would be data discrepancies when including a back propagation
      - All of the weights and biases need to be updated through gradient calculations (gradient descent)
- We created a fully-functioning parallelized neural network with OpenMP, a parallelized forward propagation with OpenMP to compare to Orlandi's version
- We were unable to run Orlandi's CUDA code on Tuckoo as his code does not recognize Tuckoo's internal architecture
  - Able to create a partially functioning parallelized neural network with CUDA and a partially-functioning parallelized forward propagation with CUDA

Serial Code

Forward Propagation

```
    // forward pass
    // compute hidden layer activation

    // hidden layer 1
    for(int j =0; j < numHiddenNodes; j++){
        double activation = hiddenLayerBias[j];

        for(int k = 0; k < numInputs; k++){
            activation += trainingInputs[i][k] * hiddenWeights[k][j];
        }

        hiddenLayer[j] = relu(activation);
    }

    // hidden layer 2
    for(int j =0; j < numHiddenNodes2; j++){
        double activation = hiddenLayerBias2[j];

        for(int k = 0; k < numHiddenNodes; k++){
            activation += hiddenLayer[k] * hiddenWeights2[k][j];
        }

        hiddenLayer2[j] = relu(activation);
    }

    // compute output layer activation
    for(int j =0; j < numOutputs; j++){
        double activation = outputLayerBias[j];

        for(int k = 0; k < numHiddenNodes2; k++){
            activation += hiddenLayer2[k] * outputWeights[k][j];
        }

        outputLayer[j] = sigmoid(activation);
    }
```

Backward Propagation

```
    // Backpropagation
    // Compute change in output weights
    double deltaOutput[numOutputs];
    for(int j = 0; j < numOutputs; j++){
        double error = (trainingOutputs[i][j] - outputLayer[j]); // L1
        deltaOutput[j] = error * dSigmoid(outputLayer[j]) ;
    }

    // Compute change in hidden weights (second layer)
    double deltaHidden2[numHiddenNodes2];
    for(int j = 0; j < numHiddenNodes2; j++){
        double error = 0.0f;
        for(int k = 0; k < numOutputs; k++){
            error += deltaOutput[k] * outputWeights[j][k];
        }
        deltaHidden2[j] = error * dRelu(hiddenLayer[j]);
    }

    // Compute change in hidden weights (first layer)
    double deltaHidden[numHiddenNodes];
    for(int j = 0; j < numHiddenNodes; j++){
        double error = 0.0f;
        for(int k = 0; k < numHiddenNodes2; k++){
            error += deltaHidden2[k] * hiddenWeights2[j][k];
        }
        deltaHidden[j] = error * dRelu(hiddenLayer2[j]);
    }

    // Apply change in output weights
    for(int j = 0; j < numOutputs; j++){
        outputLayerBias[j] += deltaOutput[j] * lr;
        for(int k = 0; k < numHiddenNodes2; k++){
            outputWeights[k][j] += hiddenLayer2[k] * deltaOutput[j] * lr;
        }
    }

    // Apply change in second hidden layer weights
    for(int j = 0; j < numHiddenNodes2; j++){
        hiddenLayerBias[j] += deltaHidden[j] * lr;
        for(int k = 0; k < numHiddenNodes; k++){
            hiddenWeights2[k][j] += hiddenLayer[k] * deltaHidden2[j] * lr;
        }
    }

    // Apply change in first hidden layer weights
    for(int j = 0; j < numHiddenNodes; j++){
        hiddenLayerBias[j] += deltaHidden[j] * lr;
        for(int k = 0; k < numInputs; k++){
            hiddenWeights[k][j] += trainingInputs[i][k] * deltaHidden[j] * lr;
        }
    }
```

Discussion/Challenges

- As a showcase of the parallelization of neural networks, we have expanded upon Orlandi's forward propagation parallelization and have implemented a fully functioning parallelized neural network.
- Finally, as with any software project, future iterations could benefit from additional testing and optimization to ensure optimal performance and accuracy.
- Overall, we are proud of the performance of our neural network model and have a deeper understanding of how neural networks and parallelization work.