Parallelism on Neural Networks

TEAM MEMBERS

- Deepak T
- Karthick N G
- Aadithya C R

Neural Network Structure

- ***Tensor*** *(Tensor.cpp)* **:** Abstract class describing a tensor, it is a wrapper for arrays of floats, providing constructor, assignment operators and destructor that automate memory allocation and de-allocation.
- ***Vector*** *(Vector.cpp)* **:** Class derived from ***Tensor*** and describing a vector (i.e. a 1D tensor), useful to represent input data.
- ***Matrix*** *(Matrix.cpp)* **:** Class derived from ***Tensor*** and describing a matrix (i.e. a 2D tensor), useful to represent layer weights.
- ***SparseLinearLayer*** *(SparseLinearLayer.cpp)* **:** Class describing a single layer of the network, characterized by a matrix of weights, by a bias, and by a non-linearity (in this case, a ***sigmoid*** function).
- ***NeuralNetwork*** *(NeuralNetwork.cpp)* **:** Class describing the whole neural network, consisting in a certain number of *stacked* SparseLinearLayers.

Sequential Implementation

- The ***NeuralNetwork*** class provides a forward method which, given an input Vector, calls sequentially the forward method of every ***SparseLinearLayer.***

```
Vector SparseLinearLayer::forward(const Vector &in_vector) const {
    /* ... */
    for (int i = 0; i < _out_features; i++) {
        float val = 0;
        for (int r = 0; r < cols; r++)
            val += in_vector[i + r] * _weights[i * cols + r];
        out_vector[i] = _non_linearity(val + _bias);
    }
    return out_vector;
}
```

Calculating Problem Size P

- The sequential implementation comprises two nested for loops:
  1. The external for loop iterates over the (N − t(R − 1)) output nodes $\boldsymbol{y_i}$, where ***t*** is the current layer number.
  2. The inner for loop iterates over the R relevant input nodes $\boldsymbol{x_{i+r}}$, multiply them to the corresponding weights $\boldsymbol{W_{i,r}}$, with r = 0…(R − 1), and accumulate the partial results.

- Therefore the Problem Size "***P***" is:

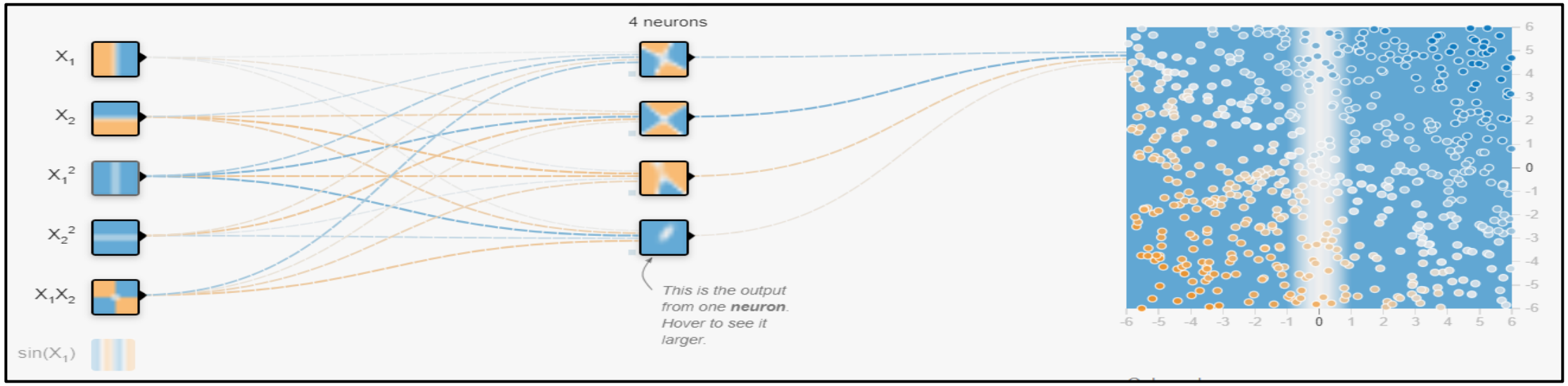
$$\mathcal{P} = \sum_{t=1}^{K-1} (N - t(R-1)) \cdot R = \left((K-1)N - \frac{K(K-1)}{2} \cdot (R-1) \right) \cdot R$$

- This can be used to compute ***strong*** and ***weak scaling efficiency***.

OpenMP Approach / Implementation

The forward method of the ***NeuralNetwork*** class consists of *three* for loops:

1. One iterating over the network's layer : ***Not parallelizable*** due to data dependency between layers.
2. One iterating over the output nodes of a certain layer : ***Embarassingly parallel*** 1D .
3. One iterating over the relevant input nodes of a certain output node in a layer and accumulating the partial result : ***Parallel reduction***



Parallel Outer for

- In this technique, parallelization is achieved by distributing the workload of iterations in the outer loop among **multiple threads**. To implement this approach effectively, we utilize *pragma directives preceding* the relevant for loop.
- Particularly, we set the scheduling strategy to **'*static*'** as the workload for each iteration remains predictable and consistent. In this context, each iteration of the loop involves a fixed number of operations: R multiplications, R + 1 sums, 1 sigmoid computation.
- By employing a static scheduling strategy, the compiler can efficiently allocate iterations to threads during compile time. This preemptive allocation *enhances* runtime performance by *minimizing runtime overhead* .
- This method *maximizes* computational efficiency across available threads.

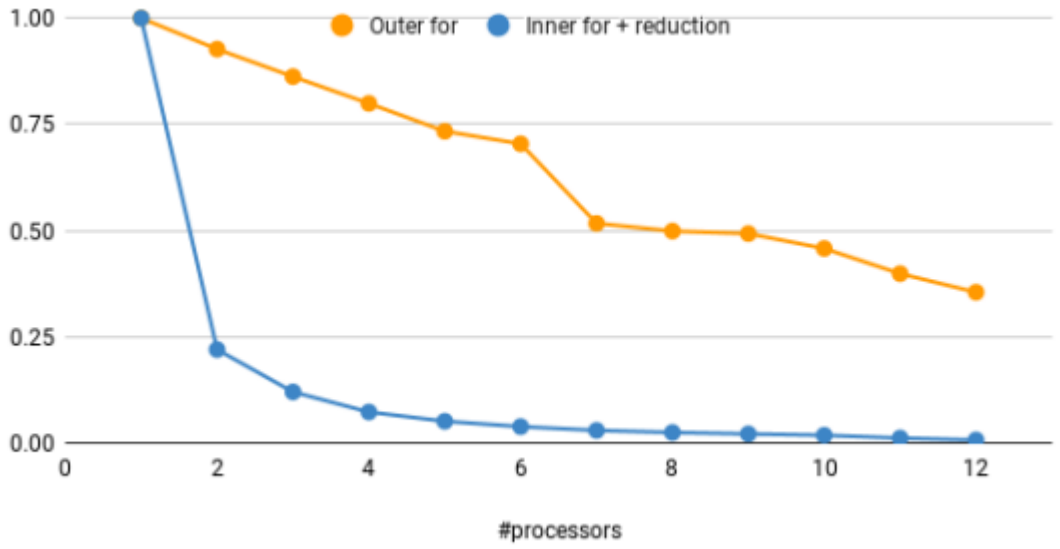
Parallel Inner for with red

- In the case of *Parallel inner for loops with reduction*, the objective is to distribute the workload of iterations in the inner loop among ***multiple threads***. Each thread executes a subset of these iterations and accumulates a *partial* result. Subsequently, these partial results are combined or "*reduced*" into a single value.
- To implement this technique, *pragma* directives are placed before the inner for loop, specifying the OpenMP options. These options are largely similar to those used in the previous technique. However, the key distinction lies in the application of a reduction on a specific variable, typically denoted as ***'val'***.
- Under this approach, each thread maintains a *private copy* of ***'val'*** and updates it during its assigned iterations. Following this, a reduction operator (typically '+') is applied to these private copies, as well as to an initial value (usually 0), effectively aggregating them into a final result.
- Due to the assumption of a small value for 'R' as per the problem's specification, this technique may be ***less effective compared to the previous one***.

Performance Evaluation

- ***Strong Scaling:*** The problem size remains constant whereas the number of processors is increased.

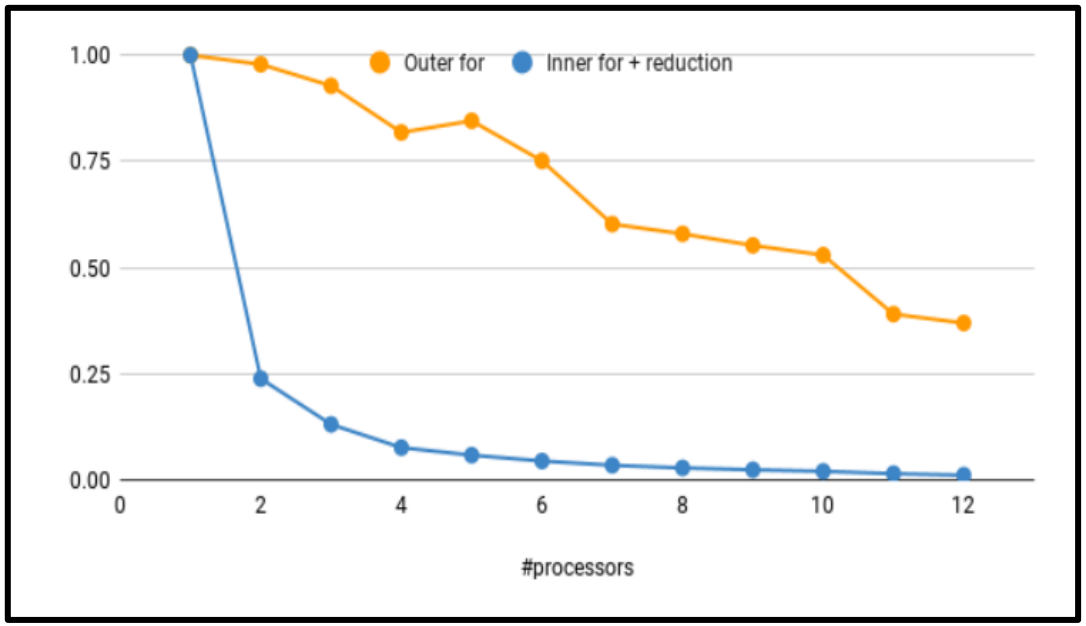
- As expected, the parallelization of ***outer for*** lead to a much higher efficiency than the reduction strategy.
- Also, we can notice a drop in efficiency in the transition between Processor #6 and Processor #7, this is because the CPU has only 6 physical cores and then a small overhead when using the other 6 Logical cores.



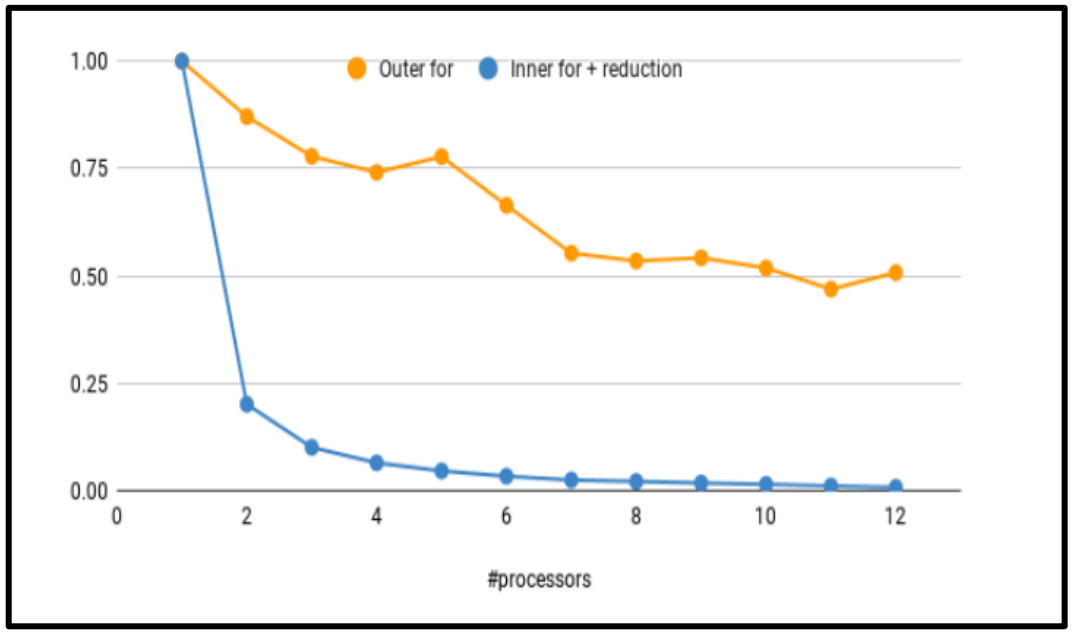
STRONG SCALING EFFICIENCY

Performance Evaluation

- ***Weak Scaling:*** The processor work remains constant whereas the number of processors is increased.
- As expected, the parallelization of ***outer for*** lead to a much higher efficiency than the reduction strategy.



WEAK SCALING EFFICIENCY
N=1000, K=25



WEAK SCALING EFFICIENCY
N=1000, K=250

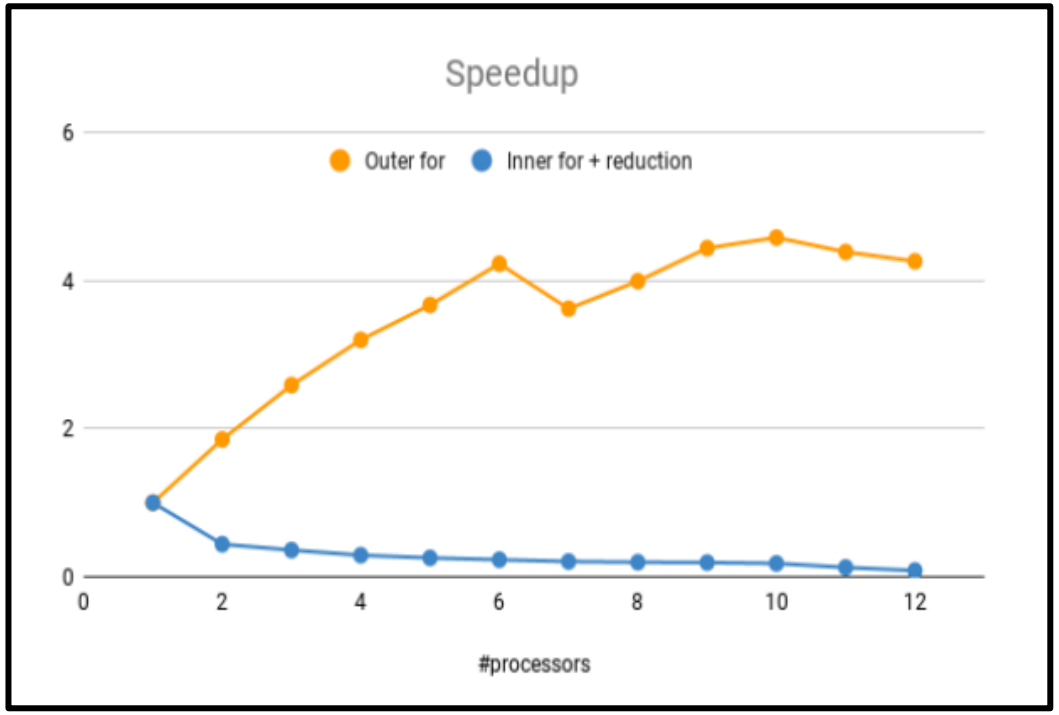
Performance Evaluation

SPEED UP:

- $$\frac{\text{Running time of program in 1 processing element}}{\text{Running time of program in p processing element}}$$

$$\text{Overall Speedup} = \frac{\text{Old execution time}}{\text{New execution time}}$$

$$= \frac{1}{\left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)}$$



SPEED UP

N=1000 K=250