Parallel Feed Forward Calculation of ANNs with CUDA

Philipp Reinig
reinig@ovgu.de
Otto-von-Guericke University
Magdeburg, Saxony-Anhalt, Germany

ABSTRACT

ANNs are a popular method in the field of artificial intelligence. For large networks, the feed forward calculation takes a long time because the underlying matrix multiplications have a runtime complexity of $O(n^3)$. Parallelization is one solution to address this issue. This paper presents an algorithm for parallel matrix multiplication, analyses it's performance and discusses when it makes sense to apply it. It was found that a parallel implementation of the matrix multiplication drastically reduces the execution time for large layer sizes (> 1000), but for smaller sizes, the serial implementation is quicker. The recommendation is therefore to use both a serial and parallel algorithm for matrix multiplcation during the feed forward calculation and choose dynamically based on the sizes of the current and next layer.

ACM Reference Format:
Philipp Reinig. 2023. Parallel Feed Forward Calculation of ANNs with CUDA. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Artificial neural networks (ANNs) have increased in popularity in the field of artifical intelligence in recent years. They are able to find connections in huge data sets and make predictions based on them. Large Language Models (LLMs) are special ANNs, which are able to generate human-like text. They are currently the most promising method in the field of natural language processing. LLMs like GPT-4 by OpenAI or Bard by Google consist of many billion parameters to achieve their impressive results. However, large ANNs increase the computational cost for training and feed forward calculations. The underlying mathematical operations are matrix multiplications with runtime complexity $O(n^3)$. Therefore parallelization is essential to run the feed forward calculations for large ANNs. This paper outlines an algorithm for parallel feed forward calculations with CUDA.

2 FEED FORWARD CALCULATION

The feed forward calculation describes the process of calculating the outputs of an ANN given a certain input vector. Each node's value is calculated by the dot product of the last layer's node values and the weights of this node.

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

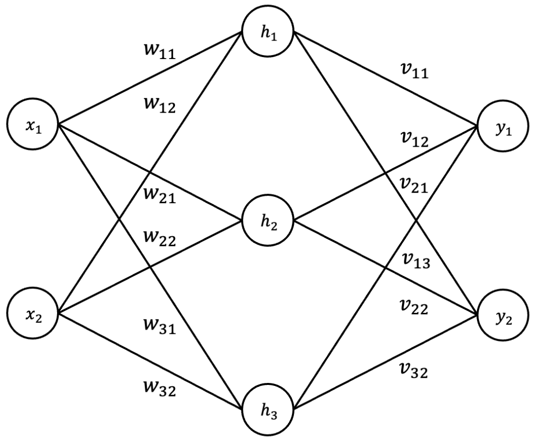

Figure 1: Example of an ANN with 1 hidden layer

Instead of calculating each node's value one by one, it's also possible to compute an entire layer by calculating a matrix multiplication. If one stores each layer's values and all weights in a matrix, the entire state of the ANN is represented by matrices and the feed forward calculation problem reduces to $n-1$ matrix multiplications for an ANN with n layers. For the example ANN of Figure 1, the matrix multiplications are:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}^T = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T * \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}^T = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}^T * \begin{bmatrix} v_{11} & v_{21} \\ v_{12} & v_{22} \\ v_{13} & v_{32} \end{bmatrix}$$

Therefore optimizing and parallelizing of the feed forward calculation reduces to optimizing and parallelizing matrix multiplication.

3 OPTIMIZATION AND PARALLELIZATION

3.1 Efficient Algorithms

First of all, one should decide on an algorithm which is efficient, but most importantly, it has to be well suited to be parallelized.

The standard algorithm for matrix multiplication consists of three nested for loops which is a runtime complexity of $O(n^3)$.

There is the recursive Strassen Algorithm which reduces the runtime complexity from $O(n^3)$ to $O(n^{log_2(7)}) \approx O(n^{2,8074})$. However, it only works with square matrices and in the case of matrix multiplications for the feed forward calculation the first operand matrix only consists of a single row. Therefore, the Strassen Algorithm is not applicable. This issue can be fixed by adding $m-1$

Algorithm 1 Trivial Matrix Multiplication

```
procedure MatrixMultiplication($A, B$)
    $n \leftarrow$ number of rows in $A$
    $m \leftarrow$ number of columns in $A$
    $o \leftarrow$ number of columns in $B$
    Let $C$ be a new matrix $n \times o$ matrix of zeros
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $m$ do
            for $k \leftarrow 1$ to $o$ do
                $C_{i,j} \leftarrow C_{i,j} + A_{i,k} * B_{k,j}$
            end for
        end for
    end for
    return $C$
end procedure
```

rows to the $1xm$ matrix containing just 0 values so the amount of rows and columns both equal m. This approach also increases the result matrix from 1 to m rows, of which $m-1$ rows contain just 0 values. These $m-1$ rows have to be sliced to obtain a matrix with the correct dimensions for the next iteration of the feed foward calculation. In summary, this means calculating $m - 1 * x$ unnecessary values to benefit from the slightly better runtime complexity, where x is the amount of columns of the first operand matrix. This comparison hasn't been analysed in detail, because it's highly hardware dependent, but regarded highly unlikely that the Strassen algorithm provides an advantage. The other issue with this algorithm for feed forward calculations is, that it only works with matrices of the same size. This can be fixed using the same idea just explained, but further increases the amount of unnecessary zero value calculations.

All other ideas available to optimize matrix multiplication are based on partitioning the matrices into blocks, like Fox's method and Cannon's method. However, they all face the same issues just described and are therefore all regarded unsuitable for the feed forward calculation.

Because of that, the only possibility for this case is parallelizing the trivial matrix multiplication algorithm.

3.2 Parallelization

It is possible to parallelize any of the three nested loops of the trivial matrix multiplication algorithm. Because a GPU is a highly parallelized processor with many thousands of threads running in parallel, it makes most sense to devide the computation into the smallest possible portions. This means parallelizing all three of the nested for loops.

3.3 Results

The serial and parallel implementations were compared for a neural network consisting of 5 hidden layers so 7 layers in total and different sizes of the layers.

Figure 2 shows a comparison between the runtime of the serial and the parallel implementation. For small layers (< 1000), the serial implementation is faster. However, the serial implementation scales linearly with the size of the matrices, whereas the parallel

Algorithm 2 Matrix Multiplication parallelized with CUDA (worker)

```
procedure MatrixMultiplication($A*, B*, C*$)
    $tid \leftarrow$ thread id
    $row \leftarrow$ tid / A.cols
    $col \leftarrow$ tid % A.cols
    for $k \leftarrow 1$ to $B.cols$ do
        $C_{row,col} \leftarrow C_{row,col} + A_{1,i} * B_{i,col}$
    end for
end procedure
```

implementation has constant runtime if the amount of necessary threads is less than the amount of streaming multiprocessors available on the GPU * 1024. If the matrix sizes exceed that point, the execution time scales sublinearly. The advantage of the parallel implementation is higher the bigger the size of the matrices become. It's therefore advised, to use both variants depending on the size of the matrix to obtain the lowest execution time. Unfortunately, it was not possible to test the parallel execution time for larger matrix sizes because the global memory of the GPU on the testing machine isn't enough.

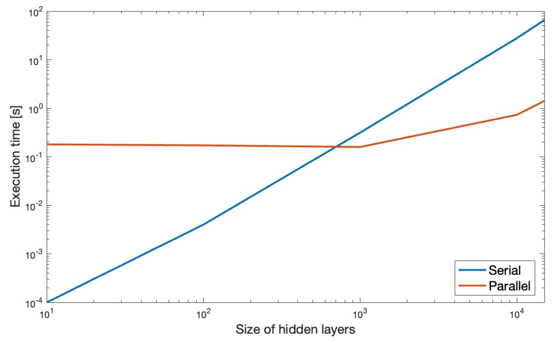

Figure 2: Comparison serial vs. parallel implementation

3.4 Possible improvements

- The parallel implementation could be improved by further increasing the amount of threads used per matrix multiplication by also parallelising the last for loop.
- Storing the value of the first operand matrix in shared memory. The values from the second operand matrix are only used once, so it would be a decrease in performance to copy them to shared memory
- Using the tensor cores available in modern Nvidia GPUs

4 SUMMARY

Parallelizing the feed forward calculation of ANNs is necessary to quickly calculate the output values for large layer sizes. Unfortunately more efficient matrix multiplication algorithms are not suited in the case of the feed forward calculation. Thus one has to parallelize the trivial matrix multiplication algorithm. It was shown, that a parallel implementation drastically improves the execution time for large layer sizes of the ANN. One should keep in mind that the execution time of the serial implementation is faster for small layer sizes. Therefore it's recommended to choose one of the two algorithms dynamically based on the sizes of the layers to compute the current feed forward iteration.