Performance Optimization and Profiling of MLPCpp in SU2 GSoC 2025 Final Report

Divyaprakash

August 29, 2025

1 Introduction

SU2 [1] is an open-source suite for computational fluid dynamics (CFD) widely used in both research and industrial design. Within SU2, the `MLPCpp` module provides a compact C++ library for enabling multilayer perceptron (MLP) inference as part of solver workflows. This capability supports a range of data-driven tasks, including surrogate modeling of fluid properties and inverse regression problems.

The original implementation of `MLPCpp` contained some performance bottlenecks and limitations in usability for advanced data-driven workflows. As part of this Google Summer of Code (GSoC) project, a series of profiling, optimization, and refactoring efforts were undertaken to improve runtime efficiency and extend the functionality of the library. Key directions included the introduction of more efficient data structures, the refactoring of computational kernels for improved memory locality and numerical precision, and enhancements to variable mapping mechanisms to support more complex workflows.

Profiling was initially conducted with `gprof`, followed by more detailed analysis with `valgrind` and `kcachegrind`, and later `Tracy`, to identify hotspots in the neural network inference pipeline. Guided by these insights, multiple optimizations were implemented, ranging from memory layout improvements in the weight matrices to function-level optimizations using fused multiply-add instructions. In addition, a low-overhead profiling capability was integrated into the SU2 build system along with documentation to enable future contributors to easily instrument and profile the code.

2 Code Profiling and Identifying Bottlenecks

Code profiling was initially carried out using the `gprof` profiler that is included with the `SU2` installation, in order to identify the functions contributing most to the overall runtime. However, to obtain a clearer picture of the child processes and to determine which operations were consuming time within those functions, `valgrind` and `kcachegrind` were subsequently employed. The latter provided an interactive visualization of the profiling data, including call graphs and annotated source code views, as shown in Figure 1. A detailed description of this process was also documented in a blog post.[1]

From the profiling results, it was observed that the function `Predict` itself did not consume a significant amount of self time. Instead, the majority of the runtime was spent in the functions `ComputePsi`, `ComputeChi`, and `ComputeX`. These functions were therefore identified as the main targets for optimization.

2.1 Suggested Improvements

The performance analysis indicated that the loop within `ComputeX` could be made more efficient. A representative code snippet is shown below:

```
mlpdouble ComputeX(std::size_t iLayer, std::size_t iNeuron) const {
  mlpdouble x;
  x = total_layers[iLayer]->GetBias(iNeuron);
```

[1] https://dpcfd.com/posts/2025/06/gsoc-week-2/

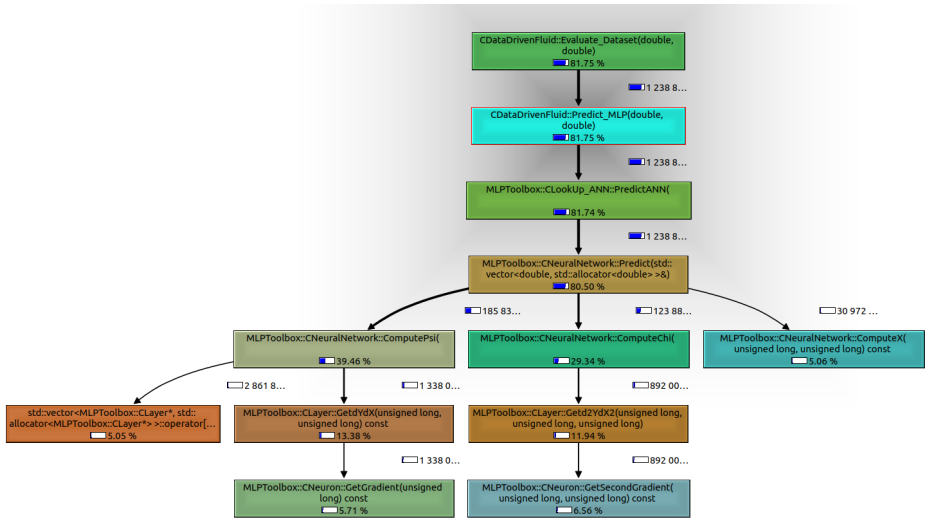


Figure 1: Call graph generated using `valgrind` and `kcachegrind`. The functions `ComputePsi`, `ComputeChi`, and `ComputeX` were identified as major contributors to the runtime.

```
  std::size_t nNeurons_previous = total_layers[iLayer - 1]->GetNNeurons();
  for (std::size_t jNeuron = 0; jNeuron < nNeurons_previous; jNeuron++) {
    x += weights_mat[iLayer - 1][iNeuron][jNeuron] *
         total_layers[iLayer - 1]->GetOutput(jNeuron);
  }
  return x;
}
```

The nested access pattern, arising from the use of a *vector of vectors of vectors*, was found to result in fragmented memory allocation:

```
std::vector<std::vector<std::vector<mlpdouble>>> weights_mat;
```

To improve performance, this structure was modified, with the changes and their effects described in the following sections.

2.2 Advanced Profiling with Tracy

In addition to `valgrind`, another profiling option was later explored, namely `Tracy`, which offers very low overhead and provides fine-grained insights into performance. This profiler was integrated with the `meson` build system of `SU2`, and changes were made to the `SU2` codebase to facilitate easy instrumentation. Both the integration and the accompanying documentation were contributed as two separate Pull Requests as part of this GSoC project.[2][3]

3 Optimizing Weights Matrix Storage

From the above discussions, it has been well established that one way to optimize the code is by improving how the weight matrices are stored. Currently, they are implemented as 3D nested vectors. To enhance performance, we consider two alternative approaches: replacing the 3D nested vectors with raw 3D pointers, or using flattened vectors. The implementation and results of both approaches are presented in the following subsections. Before that, however, we first review the current implementation of the weight matrix.

[2] https://github.com/su2code/SU2/pull/2536
[3] https://github.com/su2code/su2code.github.io/pull/180

3.1 Weights Matrix

Before modifying the implementation of the weights matrix, it is important to understand the existing design. Neural networks stored in `.mlp` files follow the TensorFlow convention, where weights are arranged as *source* $\times$ *destination*. In contrast, traditional mathematical notation typically uses the *destination* $\times$ *source* order. This difference is illustrated in Figure 2.

The `MLPCpp` code addresses this discrepancy in a clean manner:

1. `CReadNeuralNetwork::weights_mat` serves as a temporary container inside the reader class. It parses the `.mlp` file and stores the weights exactly as they appear.

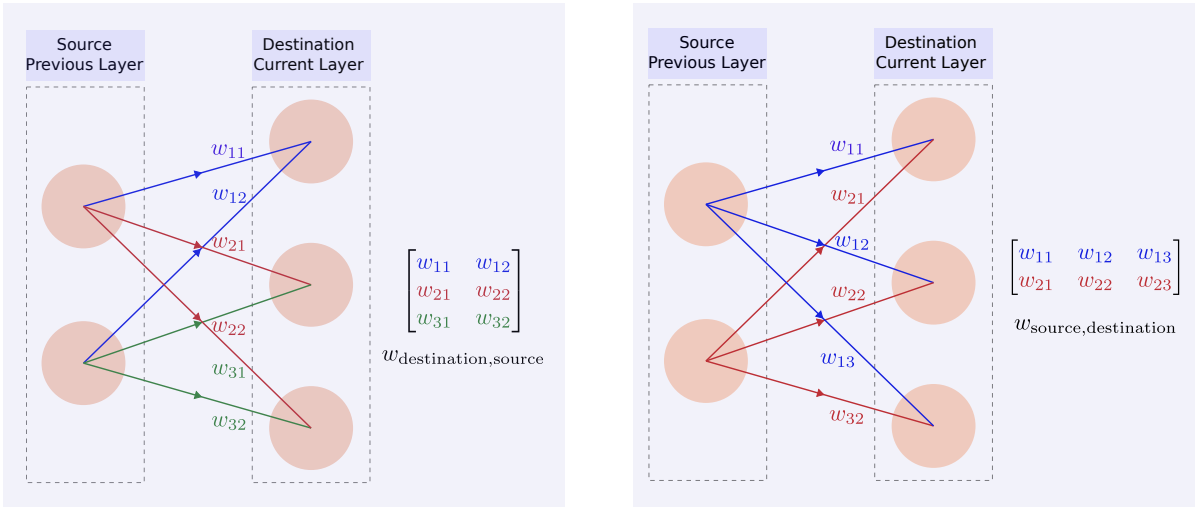
2. `CNeuralNetwork::weights_mat` belongs to the functional network class. Here, the matrix is transposed to match the conventional layout and is used in the `Predict` method for evaluation.

The key reason for this layout is computational efficiency. CPUs favor memory locality: they are significantly faster when reading data stored contiguously. Consider the dot product inside the `ComputeX` function:

```
// Core computation of the network
for (std::size_t jNeuron = 0; jNeuron < nNeurons_previous; jNeuron++) {
    x += weights_mat[iLayer - 1][iNeuron][jNeuron] *
         total_layers[iLayer - 1]->GetOutput(jNeuron);
}
```

Here, the input for a destination neuron (`iNeuron`) is computed by iterating through all source neurons (`jNeuron`). With the current `[layer][destination][source]` storage, all weights for a given destination neuron are contiguous in memory. During iteration, the CPU accesses them sequentially.

From the above discussions, it is concluded that any modification to the weight matrix storage for performance benefits must be applied to `CNeuralNetwork::weights_mat`.



(a) Conventional (b) Tensorflow

Figure 2: Comparison of weight matrix arrangements in neural networks: conventional mathematical notation (a) versus TensorFlow implementation (b). The diagrams show connections between a 2-neuron source layer and a 3-neuron destination layer, with corresponding weight matrix dimensions and indexing conventions.

3.2 3D Pointer

The neural network implementation previously employed a three-dimensional vector to represent the weight matrix.

`std::vector<std::vector<std::vector<mlpdouble>>> weights_mat`

While convenient, this approach introduced additional overhead and less predictable memory access patterns. To optimize performance and provide finer control over memory allocation, the structure has been refactored to use a triple pointer, declared as

`mlpdouble*** weights_mat`

3.2.1 Implementation

The refactored design maintains the logical indexing scheme

`weights_mat[layer][destination][source]`

while eliminating the overhead of nested `std::vector` containers. Allocation proceeds hierarchically:

1. An array of pointers for each layer.
2. For each layer, an array of neuron pointers.
3. For each neuron, an array of incoming weights.

It is implemented as follows in the `SizeWeights()` Function.

```
1   weights_mat = new mlpdouble**[n_hidden_layers + 1];
2   weights_mat[0] = new mlpdouble*[hiddenLayers[0]->GetNNeurons()];
3   for (auto iNeuron = 0u; iNeuron < hiddenLayers[0]->GetNNeurons(); iNeuron++)
4     weights_mat[0][iNeuron] = new mlpdouble[inputLayer->GetNNeurons()];
5
6   for (auto iLayer = 1u; iLayer < n_hidden_layers; iLayer++) {
7     weights_mat[iLayer] = new mlpdouble*[hiddenLayers[iLayer]->GetNNeurons()];
8     for (auto iNeuron = 0u; iNeuron < hiddenLayers[iLayer]->GetNNeurons();
9          iNeuron++) {
10      weights_mat[iLayer][iNeuron] = new mlpdouble[hiddenLayers[iLayer -
        ↪  1]->GetNNeurons()];
11    }
12  }
13  weights_mat[n_hidden_layers] = new mlpdouble*[outputLayer->GetNNeurons()];
14  for (auto iNeuron = 0u; iNeuron < outputLayer->GetNNeurons(); iNeuron++) {
15    weights_mat[n_hidden_layers][iNeuron] = new mlpdouble[hiddenLayers[n_hidden_layers
      ↪  - 1]->GetNNeurons()];
16  }
17
18  ANN_outputs = new mlpdouble[outputLayer->GetNNeurons()];
```

Because allocation is manual, explicit deallocation must follow the inverse order:

1. Deallocate each neuron's incoming weights.
2. Deallocate each layer's array of neuron pointers.
3. Finally, deallocate the outer layer array.

This approach ensures memory safety and prevents leaks. The updated destructor is given below:

```cpp
~CNeuralNetwork() {
    delete[] ANN_outputs;

    if (weights_mat != nullptr) {
        // Clean up first layer (input to first hidden)
        if (n_hidden_layers > 0) {
            for (auto iNeuron = 0u; iNeuron < hiddenLayers[0]->GetNNeurons(); iNeuron++)
            ↪ {
                delete[] weights_mat[0][iNeuron];
            }
            delete[] weights_mat[0];
        }

        // Clean up hidden layers
        for (auto iLayer = 1u; iLayer < n_hidden_layers; iLayer++) {
            for (auto iNeuron = 0u; iNeuron < hiddenLayers[iLayer]->GetNNeurons();
            ↪ iNeuron++) {
                delete[] weights_mat[iLayer][iNeuron];
            }
            delete[] weights_mat[iLayer];
        }

        // Clean up last layer (last hidden to output)
        if (outputLayer != nullptr) {
            for (auto iNeuron = 0u; iNeuron < outputLayer->GetNNeurons(); iNeuron++) {
                delete[] weights_mat[n_hidden_layers][iNeuron];
            }
            delete[] weights_mat[n_hidden_layers];
        }

        delete[] weights_mat;
    }

    delete inputLayer;
    delete outputLayer;
    for (std::size_t i = 1; i < total_layers.size() - 1; i++) {
        delete total_layers[i];
    }
};
```

The conversion from `std::vector` to a triple-pointer preserves the computational logic while improving control over allocation. Public interfaces remain unchanged, so expressions such as `weights_mat[i][j][k]` continue to function as before. The main trade-off is between safety and performance: the vector-based implementation offered automatic memory management and bounds checking, whereas the pointer-based design requires careful allocation and deallocation. However, it may provide performance benefits due to reduced overhead and improved cache locality.

Implementation Note: The complete source code for this refactoring approach is available in the `gsoc_dp` branch of the project repository[4].

[4] https://github.com/divyaprakash-iitd/MLPCpp/tree/gsoc_dp

3.3 Flattened Vector

Disclaimer

Some of the changes described in this section are still under debugging and may not yet work as intended.

In this section, an alternative approach is proposed to refactor the core weight storage from a 3D `std::vector` of vectors to a single, contiguous, flattened 1D `std::vector` for improved performance. The refactoring is to be performed in three safe, verifiable phases:

1. **Parallel Implementation:** Introduce the new flattened vector and its accessor functions *alongside* the existing implementation.
2. **Verification:** Run both the original `Predict` function and a new `Predict_flat` function to check if the new implementation is mathematically identical.
3. **Cleanup:** Once verification is successful, the old 3D vector and its associated functions are removed.

3.3.1 Implementation

This phase involves adding new code without modifying the existing, validated logic.

Step 1: Modify `CNeuralNetwork.hpp` - Data Structures & Accessors: New data structures and accessor functions are added to the `CNeuralNetwork` class. In the `private` section of the `CNeuralNetwork` class, new variables are added:

```
// Add this block in the private section of CNeuralNetwork
// --- New flattened vector implementation ---
std::vector<mlpdouble> flat_weights;
std::vector<int> layer_sizes;
std::vector<int> layer_offsets;
```

New private and public helper functions are added below. These functions utilize the `inline` keyword to ensure compiler optimization eliminates function call overhead when optimizations are enabled (e.g., with the `-O2` flag).

```
// Add this function to the private section of CNeuralNetwork
/**
 * @brief (NEW) Calculates the 1D index for the flattened weights vector.
 */
inline int get_flat_index(int i_layer, int i_dest_neuron, int i_src_neuron) const {
    return layer_offsets[i_layer] + i_dest_neuron * layer_sizes[i_layer] + i_src_neuron;
}

// Add these functions to the public section of CNeuralNetwork
/**
 * @brief (NEW) Sets a weight value in the flattened vector, handling the index swap.
 */
inline void SetFlatWeight(unsigned long i_layer, unsigned long i_src_neuron,
                          unsigned long i_dest_neuron, mlpdouble value) {
    int index = get_flat_index(i_layer, i_dest_neuron, i_src_neuron);
    flat_weights[index] = value;
}
```

```

/**
 * @brief (NEW) Gets a weight value from the flattened vector using the in-memory layout.
 */
inline mlpdouble GetFlatWeight(int i_layer, int i_dest_neuron, int i_src_neuron) const {
    return flat_weights[get_flat_index(i_layer, i_dest_neuron, i_src_neuron)];
}
```

Step 2: Modify `CNeuralNetwork.hpp` - Sizing the New Structures: In the `SizeWeights()` function, logic is added to calculate the total number of weights and correctly size the new helper vectors and the `flat_weights` vector.

```
// In CNeuralNetwork.hpp
void SizeWeights() {
    // ... existing code to set up total_layers ...

    // --- START NEW IMPLEMENTATION ---
    layer_sizes.clear();
    for(const auto& layer : total_layers) {
        layer_sizes.push_back(layer->GetNNeurons());
    }

    layer_offsets.clear();
    int total_weights = 0;
    layer_offsets.push_back(total_weights);

    for (size_t i = 0; i < GetNLayers() - 1; ++i) {
        int weights_in_layer = layer_sizes[i+1] * layer_sizes[i];
        total_weights += weights_in_layer;
        layer_offsets.push_back(total_weights);
    }
    flat_weights.resize(total_weights);
    // --- END NEW IMPLEMENTATION ---

    // --- OLD IMPLEMENTATION (unchanged for now) ---
    weights_mat.resize(n_hidden_layers + 1);
    // ... (rest of the old implementation)
}
```

Step 3: Modify `CLookUp_ANN.hpp` - Populating Both Data Structures: This represents the only change needed for the data loading part of the refactoring. In the `CLookUp_ANN` class, within the nested loop that sets the weights, a call to the new `SetFlatWeight` function is added. The `Reader` object has already parsed the `.ann` file into its own memory. This loop transfers that data into the `ANN` object used for computations. A line is simply added to populate the new `flat_weights` vector simultaneously with the original `weights_mat`.

```
// In CLookUp_ANN.hpp, inside the constructor or loading function, around line 108
for (auto i_layer = 0u; i_layer < ANN.GetNWeightLayers(); i_layer++) {
  ANN.SetActivationFunction(i_layer, Reader.GetActivationFunction(i_layer));
  for (auto i_neuron = 0u; i_neuron < ANN.GetNNeurons(i_layer); i_neuron++) {
    for (auto j_neuron = 0u; j_neuron < ANN.GetNNeurons(i_layer + 1); j_neuron++) {
```

```
      // Original call (unchanged)
      ANN.SetWeight(i_layer, i_neuron, j_neuron,
                    Reader.GetWeight(i_layer, i_neuron, j_neuron));

      // --- ADD THIS LINE ---
      // New call to populate the parallel flat vector structure
      ANN.SetFlatWeight(i_layer, i_neuron, j_neuron,
                        Reader.GetWeight(i_layer, i_neuron, j_neuron));
      // ----------------------
    }
  }
}
```

Step 4: Modify `CNeuralNetwork.hpp` - Create Parallel Compute Functions: For every function that uses `weights_mat`, a duplicate version with a `_flat` suffix is created that uses the new `GetFlatWeight` accessor.

```
// In CNeuralNetwork.hpp, add these new functions.

/**
 * @brief (NEW) Compute neuron activation function input using the flattened weights.
 */
mlpdouble ComputeX_flat(std::size_t iLayer, std::size_t iNeuron) const {
      mlpdouble x = total_layers[iLayer]->GetBias(iNeuron);
      std::size_t nNeurons_previous = total_layers[iLayer - 1]->GetNNeurons();
      for (std::size_t jNeuron = 0; jNeuron < nNeurons_previous; jNeuron++) {
          // Access is [layer][destination][source]
          x += GetFlatWeight(iLayer - 1, iNeuron, jNeuron) *
               total_layers[iLayer - 1]->GetOutput(jNeuron);
      }
      return x;
  }


/**
 * @brief (NEW) Compute Psi for gradients using the flattened weights.
 */
mlpdouble ComputePsi_flat(std::size_t iLayer, std::size_t iNeuron,
                                std::size_t jInput) const {
    // Implementation using GetFlatWeight...
}

// ... and so on for ComputeChi, etc. ...
```

Once verification with the flattened implementations is successful, the old implementation can be safely removed. After these steps, the project is fully migrated to the new, more efficient flattened weight structure, providing improved memory locality and cache performance for neural network computations.

Implementation Note: The complete source code for this refactoring approach is available in the `flatten`

branch of the project repository[5].

4 Optimizing Variable Matching

Efficient mapping of input and output variables is essential for integrating neural networks within `MLPCpp`. A hash map–based method, along with a cleaner replacement for the `FindVariableIndices` function, was proposed to improve clarity and maintainability.

4.1 Hash Maps

The changes were made to the `FindVariableIndices` function. This function is responsible for mapping variable names to their corresponding indices within the Artificial Neural Network (ANN).

The original implementation used a nested loop structure to find matching variable names. The outer loop iterated through the variables of the ANN, and the inner loop iterated through the list of variable names to be found. This approach has a time complexity of $O(N \times M)$, where N is the number of variables in the ANN and M is the number of variables to be mapped. For large models, this can lead to significant performance bottlenecks.

4.1.1 Implementation

The new implementation in `CLookUp_ANN.hpp` replaces the nested loops with a more efficient approach utilizing a `std::unordered_map` (hash map). The steps are as follows:

1. A hash map is created, mapping each variable name to its index in the input list.
2. The code then iterates a single time through the ANNś variables.
3. For each ANN variable, it performs a quick lookup in the hash map to see if there is a match.

This change reduces the time complexity to $O(N + M)$, which is a substantial improvement over the previous $O(N \times M)$.

```cpp
std::vector<std::pair<size_t, size_t>> FindVariableIndices(
  std::size_t i_ANN,
  std::vector<std::string> variable_names,
  bool input) const {
  std::vector<std::pair<size_t, size_t>> variable_indices;
  auto nVar = input ? NeuralNetworks[i_ANN].GetnInputs()
                    : NeuralNetworks[i_ANN].GetnOutputs();

  // Create a hash map from variable names to their indices
  std::unordered_map<std::string, size_t> name_to_index;
  for (size_t jVar = 0; jVar < variable_names.size(); ++jVar) {
      name_to_index[variable_names[jVar]] = jVar;
  }

  // Iterate over neural network variables
  for (size_t iVar = 0; iVar < nVar; ++iVar) {
      std::string ANN_varname = input ? NeuralNetworks[i_ANN].GetInputName(iVar)
                                      : NeuralNetworks[i_ANN].GetOutputName(iVar);
      // Look up the variable name in the hash map
      auto it = name_to_index.find(ANN_varname);
```

[5] https://github.com/divyaprakash-iitd/MLPCpp/tree/flatten

```
        if (it != name_to_index.end()) {
            variable_indices.push_back(std::make_pair(it->second, iVar));
        }
    }

    return variable_indices;
}
```

The new implementation offers several advantages:

- **Improved Performance:** The use of a hash map for lookups is significantly faster than the nested loop approach, especially as the number of variables increases.
- **Enhanced Readability:** The new code is more organized and easier to understand. The logic is more straightforward, which improves maintainability.
- **Better Scalability:** The improved performance ensures that the code will scale more effectively as the complexity of the neural network models grows.

Implementation Note: The complete source code for this refactoring approach is available in the `gsoc_dp` branch of the project repository[6].

4.2 Replacing `FindVariableIndices` Function

Disclaimer

Some of the changes described in this section are still under debugging and may not yet work as intended.

In this section, an alternative approach is proposed for refactoring of the variable matching logic within the `CLookUp_ANN` and `CNeuralNetwork` classes. The goal is to improve **efficiency, readability, and maintainability**. The key change is to replace the current `FindVariableIndices` function with a more powerful and efficient method, `GetVariableMapping`, inside the `CNeuralNetwork` class. This new method performs the complete variable matching and index retrieval in a single operation, eliminating redundant work and clarifying the code's intent.

The original implementation in `CLookUp_ANN::PairVariableswithMLPs` relied on calling `FindVariableIndices` twice (once for inputs, once for outputs) for every neural network in the collection.

```
// Original logic
std::vector<std::pair<size_t, size_t>> Input_Indices =
    FindVariableIndices(iMLP, inputVariables, true);
isInput = Input_Indices.size() > 0;

if (isInput) {
  std::vector<std::pair<size_t, size_t>> Output_Indices =
      FindVariableIndices(iMLP, outputVariables, false);
  isOutput = Output_Indices.size() > 0;

  if (isOutput) {
    // ... update map
  }
}
```

[6] https://github.com/divyaprakash-iitd/MLPCpp/tree/gsoc_dp

This approach has two main drawbacks:

1. **Inefficiency:** It involves multiple loops and lookups for a single network check.
2. **Separation of Concerns:** The logic for how a `CNeuralNetwork` matches against a set of variables was located in the `CLookUp_ANN` class, not within the `CNeuralNetwork` class itself.

While `operator==` was suggested, it is not ideal because it's expected to return a simple `bool`, whereas this operation requires the resulting index maps. A boolean check would still require a second operation to retrieve the indices, leading to redundant computation.

4.2.1 Implementation

The proposed solution centralizes the matching logic into a single, efficient method within the `CNeuralNetwork` class.

Step 1: Introduce `MatchResult` Struct and `GetVariableMapping` Method A new struct, `MatchResult`, is defined to hold the complete result of a matching operation. A new public method, `GetVariableMapping`, is added to `CNeuralNetwork`.

```
// CNeuralNetwork.hpp
// Add this struct definition before the CNeuralNetwork class
struct MatchResult {
    bool is_match = false;
    std::vector<std::pair<size_t, size_t>> input_indices;
    std::vector<std::pair<size_t, size_t>> output_indices;
};

// Add this public method to the CNeuralNetwork class
public:
  MatchResult GetVariableMapping(
      const std::vector<std::string>& lookup_inputs,
      const std::vector<std::string>& lookup_outputs
  ) const;
```

This method efficiently checks for matches and returns the `MatchResult` struct containing a boolean flag and the corresponding index vectors, all in one pass. The matching logic used if given below.

- **Inputs:** All of the network's defined inputs must be present in the `lookup_inputs`.
- **Outputs:** At least one of the network's defined outputs must be present in the `lookup_outputs`.

Step 2: Simplify `PairVariableswithMLPs` and Remove `FindVariableIndices` The `PairVariableswithMLPs` function in `CLookUp_ANN.hpp` is refactored to use the new method, making it significantly cleaner and more expressive. The `FindVariableIndices` function is no longer needed and should be deleted.

```
// CLookUp_ANN.hpp
// Replace the old PairVariableswithMLPs function with this:
void PairVariableswithMLPs(MLPToolbox::CIOMap &ioMap) {
    auto inputVariables = ioMap.GetInputVars();
    auto outputVariables = ioMap.GetOutputVars();

    for (size_t iMLP = 0; iMLP < NeuralNetworks.size(); iMLP++) {
      // Single, efficient call to the new method
```

```
      auto mapping_result =
          NeuralNetworks[iMLP].GetVariableMapping(inputVariables, outputVariables);

      if (mapping_result.is_match) {
        // Update the map directly with the results
        ioMap.PushMLPIndex(iMLP);
        ioMap.PushInputIndices(mapping_result.input_indices);
        ioMap.PushOutputIndices(mapping_result.output_indices);
      }
    }

    CheckUseOfInputs(ioMap);
    CheckUseOfOutputs(ioMap);
}

// The FindVariableIndices function should be deleted from this file.
```

Implementing these changes is expected to yield the following advantages:

- **Efficiency:** Eliminates redundant loops and string comparisons by performing the variable check and index retrieval in a single, optimized operation.
- **Readability:** The code in `PairVariableswithMLPs` now clearly expresses its intent: get the variable mapping from the network and use it if there is a match.
- **Encapsulation:** The logic for how a neural network matches variables is now correctly placed within the `CNeuralNetwork` class, improving code organization and maintainability.

Implementation Note: The complete source code for this refactoring approach is available in the `revarmap` branch of the project repository[7].

5 Neural Network Function Optimization

Several critical computational functions in the neural network implementation were optimized to improve both performance and numerical precision. The optimizations focused on reducing redundant array access operations and leveraging fused multiply-add (FMA) instructions for better floating-point accuracy.

5.1 Implementation

The key optimization strategies used are listed below.

1. **Pre-extraction of frequently accessed data**: Weight vectors and neuron counts are cached before loops to minimize repeated pointer dereferencing.
2. **FMA instruction usage**: Replaced separate multiplication and addition operations with `std::fma()` for improved numerical precision and potential performance gains.
3. **Consistent variable naming**: Standardized loop indices to improve code readability and maintainability.

Using the above strategies, the following compute functions were optimized.

[7] https://github.com/divyaprakash-iitd/MLPCpp/tree/revarmap

ComputeX Function: The `ComputeX` function computes the weighted sum input to a neuron's activation function:

```
mlpdouble ComputeX(std::size_t iLayer, std::size_t iNeuron) const noexcept {
  const auto& w = weights_mat[iLayer - 1][iNeuron];
  const auto  n = total_layers[iLayer - 1]->GetNNeurons();

  mlpdouble x = total_layers[iLayer]->GetBias(iNeuron); // start from bias
  for (std::size_t j = 0; j < n; ++j) {
    // If GetOutput(j) is cheap (non-virtual, inlined), this is fine.
    x = std::fma(w[j], total_layers[iLayer - 1]->GetOutput(j), x);
  }
  return x;
}
```

ComputePsi Function: The `ComputePsi` function computes the weighted sum of output derivatives from the previous layer:

```
mlpdouble ComputePsi(std::size_t iLayer, std::size_t iNeuron,
                     std::size_t jInput) const
{
    const auto* prev = total_layers[iLayer - 1];               // cache pointer
    const std::size_t n = prev->GetNNeurons();                 // cache bound
    const auto& wrow = weights_mat[iLayer - 1][iNeuron];       // one row of weights

    mlpdouble psi = 0;
    for (std::size_t j = 0; j < n; ++j) {
        psi = std::fma(wrow[j], prev->GetdYdX(j, jInput), psi);
    }
    return psi;
}
```

ComputeChi Function: The `ComputeChi` function computes the weighted sum of second-order derivatives from the previous layer:

```
mlpdouble ComputeChi(std::size_t iLayer, std::size_t iNeuron,
                     std::size_t jInput, std::size_t kInput) const {
  mlpdouble chi = 0;
  for (std::size_t jNeuron = 0;
       jNeuron < total_layers[iLayer - 1]->GetNNeurons();
       ++jNeuron) {
    chi = std::fma(
        weights_mat[iLayer - 1][iNeuron][jNeuron],
        total_layers[iLayer - 1]->Getd2YdX2(jNeuron, jInput, kInput),
        chi);
  }
  return chi;
}
```

ComputedOutputdInput Function: The `ComputedOutputdInput` function computes the weighted sum of output derivatives with respect to inputs:

```
mlpdouble ComputedOutputdInput(std::size_t iLayer, std::size_t iNeuron,
                               std::size_t iInput) const {
```

```
  const auto& w = weights_mat[iLayer - 1][iNeuron];
  const auto  n = total_layers[iLayer - 1]->GetNNeurons();

  mlpdouble doutput_dinput = 0;
  for (std::size_t j = 0; j < n; ++j) {
    doutput_dinput = std::fma(w[j], total_layers[iLayer - 1]->GetdYdX(j, iInput), doutput_dinput);
  }
  return doutput_dinput;
}
```

The use of `std::fma()` provides improved floating-point precision by performing the multiply-add operation as a single step, reducing intermediate rounding errors that would occur in separate multiplication and addition operations.

Memory Access Optimization: Pre-extracting weight vectors and neuron counts reduces the computational overhead of repeated array indexing and function calls within tight loops, particularly beneficial for networks with large layer sizes.

Cache Efficiency: The optimized access patterns improve cache locality by reducing the number of pointer dereferences and enabling more predictable memory access patterns during the forward and backward propagation phases.

Implementation Note: The complete source code for this refactoring approach is available in the `gsoc_dp` branch of the project repository[8].

6 Results

The optimizations described in Section 5 and Subsections 3.2 and 4.1 were combined in the code. The resulting implementation was submitted as a Pull Request[9] to the `MLPCpp` subproject of `SU2`.

To verify the correctness of the implementation, the code was compiled and the test case provided in the `MLPCpp` repository was executed. In this test, the derivatives obtained using finite differences were found to agree with those calculated analytically by the network, thereby confirming the validity of the implementation.

The prediction times of the original implementation were compared with those of the optimized GSoC version over multiple runs. The original implementation was observed to have an average runtime of **251.98 ± 7.16 ms**, whereas the optimized version reduced this to **178.96 ± 4.56 ms**. This corresponds to a **1.41× speedup**, or equivalently a **28.9% reduction in execution time**. Figure X presents the average runtimes with standard deviation error bars, which reflect the run-to-run fluctuations in the measurements. It may therefore be concluded that the optimized implementation not only achieves faster predictions but also exhibits greater consistency across runs.

7 Conclusion

This project has contributed to the performance improvement and usability of the `MLPCpp` module in SU2. Through systematic profiling and targeted optimizations, key runtime bottlenecks were identified and addressed. The refactoring of the weight matrix storage, introduction of hash-based variable lookup, and improvements to the neural network compute functions resulted in a measurable reduction in execution time. Specifically, the optimized implementation achieved a mean prediction runtime of 178.96 ± 4.56 ms compared to 251.98 ± 7.16 ms for the original version, corresponding to a 1.41× speedup (approximately 29% reduction in execution time).

[8] https://github.com/divyaprakash-iitd/MLPCpp/tree/gsoc_dp
[9] https://github.com/EvertBunschoten/MLPCpp/pull/2

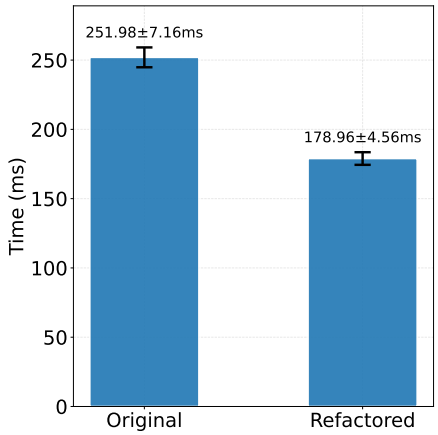


Figure 3: Average prediction time of the original and refactored implementations. Error bars represent the standard deviation across multiple runs. The refactored implementation was observed to achieve a mean runtime of 178.96 ± 4.56 ms compared to 251.98 ± 7.16 ms for the original, corresponding to a $1.41\times$ speedup (approximately 29% reduction in execution time).

Beyond performance improvements, this work also contributed to the long-term maintainability of SU2. The integration of the Tracy profiler into the Meson build system, along with the accompanying documentation, ensures that future development within SU2 can benefit from low-overhead runtime analysis and continuous performance monitoring.

8 Future Work

Although several important optimizations were completed during this project, some work remains in progress and would benefit from further development. In particular, the migration of the weight matrix storage from a nested vector structure to a flattened vector representation is still under debugging. Completing this transition would allow the full benefits of contiguous memory storage to be realized in terms of performance and cache efficiency.

Similarly, the proposed replacement of the `FindVariableIndices` function with a more direct and maintainable variable mapping approach requires additional testing and integration. Finalizing this change would improve both the clarity and the efficiency of variable lookups within `MLPCpp`.

It is hoped that these ongoing efforts can be carried forward, building on the foundation established in this work and contributing to the long-term performance and maintainability of SU2.

9 Acknowledgment

I would like to sincerely thank Google and the SU2 community for providing me with the opportunity to contribute to this project through GSoC 2025. I am especially grateful to my mentors, Evert Bunschoten and Joshua A. Kelly, for their constant guidance, encouragement, and support throughout the project. Their approachable and friendly mentorship created an excellent learning environment, and their insights were invaluable in shaping the progress and outcomes of this work. I am also thankful to the wider SU2 developer community for their feedback, code reviews, and discussions, which greatly enriched the overall experience.

References

[1] Thomas D Economon et al. “SU2: An open-source suite for multiphysics simulation and design”. In: *Aiaa Journal* 54.3 (2016), pp. 828–846.