Using Performance Profiling and Code Optimization to Accelerate Data-Driven Fluid Simulations in SU2

Divyaprakash
Mentors: Evert Bunschoten & Joshua A. Kelly

SU2 conference 2025

September 29, 2025

SU2 foundation



Google Summer of Code

Background & Objectives

MLPCpp[1] in SU2 allows integration of dense, feed-forward neural networks for data-driven predictions during CFD simulations. It enables loading and using pre-trained MLP models to enhance physics-informed workflows in SU2.

The GSoC work focused on two main objectives:

1. Integrate a modern profiler (Tracy) into SU2 — **Success**
2. Profile and improve the MLPCpp submodule — **Partly successful**

[1] https://github.com/EvertBunschoten/MLPCpp

Objective - I

Profiler comparison: gprof vs Tracy

Aspect	gprof (SU2)	Tracy
Type	Offline instrumentation + statistical sampling; compile with `-Denable-gprof`	Hybrid: low-overhead instrumentation + sampling, with live GUI
Runtime	Offline: run program then post-process results	Real-time: GUI connects to running program and shows timelines
Resolution	Function-level aggregates, wall-clock percentages	Nanosecond-resolution timestamps
Overhead	Very low sampling overhead; requires program exit	Very low overhead per zone; suitable for long runs
Threading	Works, but accuracy can degrade with heavy threaded noise	Excellent multi-thread visibility (timelines, locks)

Tracy Integration: Step 1 - Meson Configuration[2]

1. Create Tracy Wrap File: Create `subprojects/tracy.wrap`

tracy.wrap

```
[wrap-git]
url = https://github.com/wolfpld/tracy.git
revision = master
depth = 1
```

2. Add Meson Option: Edit `meson_options.txt`

meson_options.txt

```
option('tracy_enable', type: 'boolean',
       value: false,
       description: 'Enable Tracy profiling support')
```

[2] https://su2code.github.io/docs_v7/Tracy-Integration/

Tracy Integration: Step 2 - Build Configuration

Modify main meson.build file:

Conditional Tracy Dependency

```
if get_option('tracy_enable')
    tracy_dep = dependency('tracy', static: true)
    su2_deps += tracy_dep
    su2_cpp_args += '-DTRACY_ENABLE'

    if get_option('buildtype') != 'debugoptimized'
        warning('For optimal Tracy profiling,
                 use --buildtype=debugoptimized')
    endif
endif
```

Build Command:

Configure and Build

```
./meson.py setup build -Dwith-mpi=disabled --buildtype=
    debugoptimized -Denable-tracy=true --prefix=<
    SU2_INSTALL_PATH>
ninja -C build_tracy install
```

Instrumenting SU2 Code

Functions are profiled using centralized wrapper macros, so profiling can be enabled or disabled at compile time without changes to the core logic.

1. Include the SU2 Tracy Wrapper Header:

Add an include directive for `tracy_structure.hpp` at the top of your C++ source file

2. Instrument the Function with SU2 Macros:

Use `SU2_ZONE_SCOPED_N("MyLabel")` to mark a function or scope with a custom name.

```
void MyFunction() {
  SU2_ZONE_SCOPED_N( MyFunction );
  // Function implementation
}
```

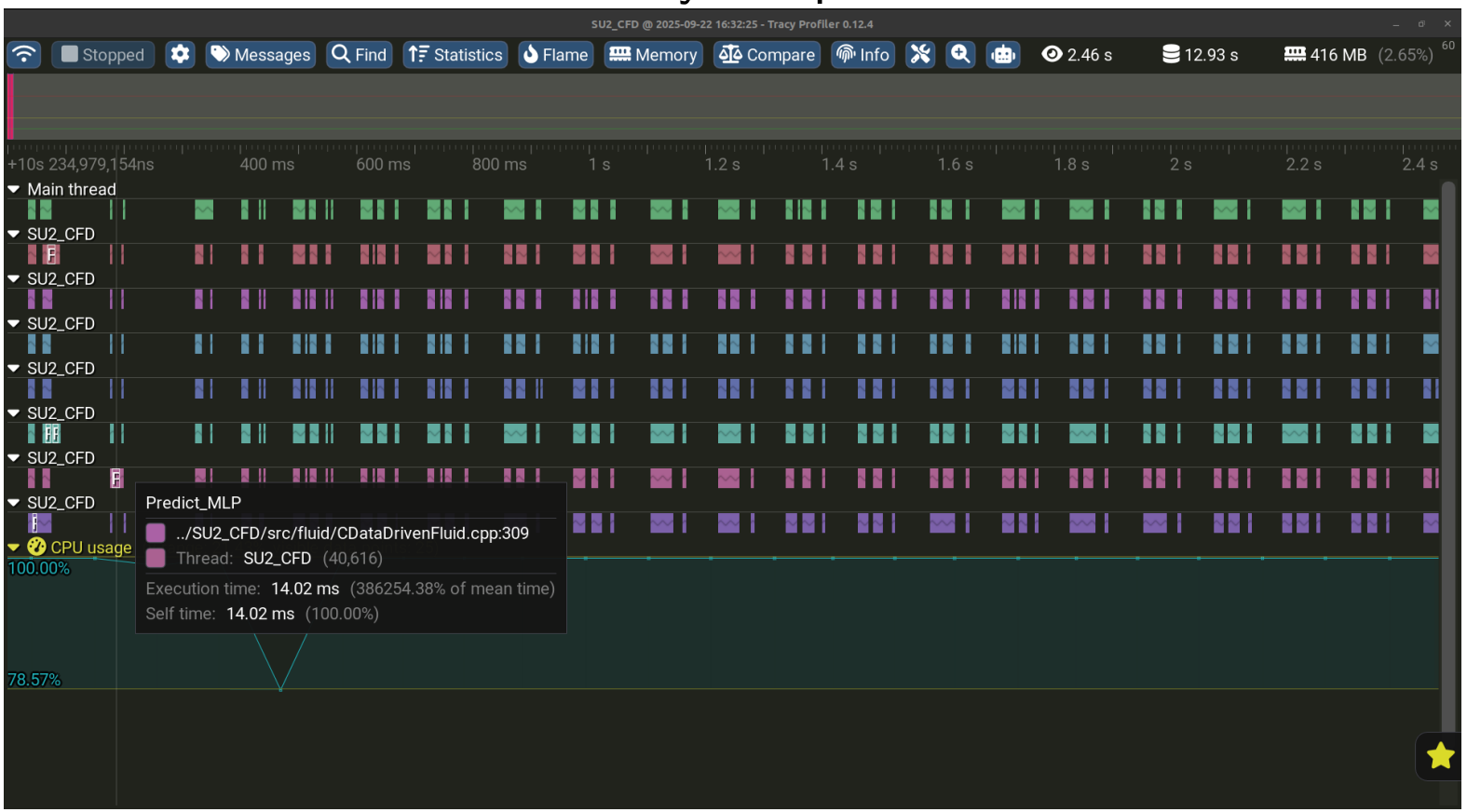
Output Comparison

gprof Output

```
Flat profile:

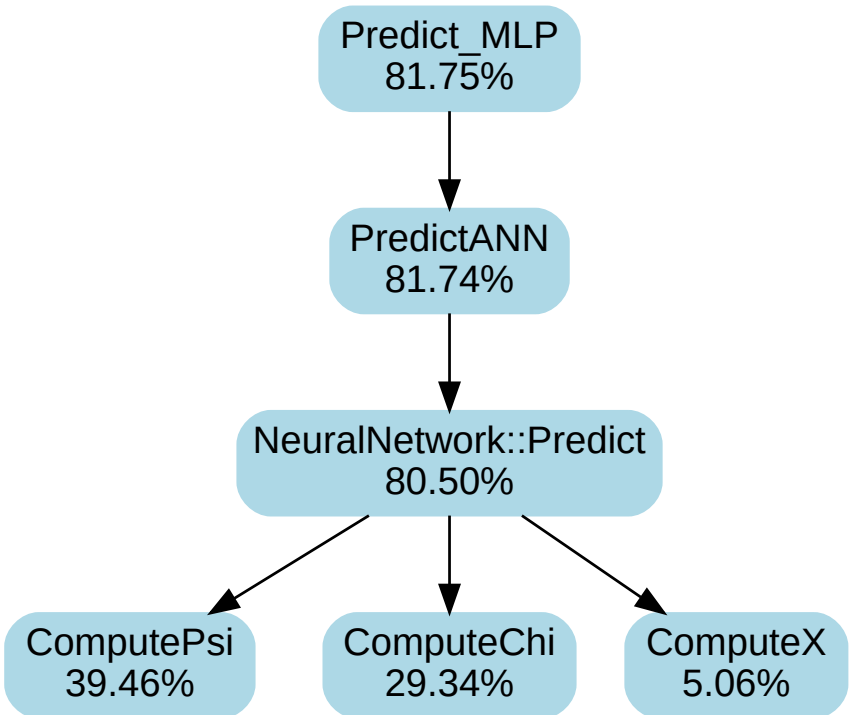
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 47.43      2.03     2.03                             MLPToolbox::CNeuralNetwork::Predict(std::vector<double, std::allocator<double> >&)
  8.18      2.38     0.35                             CUpwGeneralRoe_Flow::ComputeResidual(CConfig const*)
  7.24      2.69     0.31                             CSysMatrix<double>::ComputeLU_SGSPreconditioner(CSysVector<double> const&, CSysVector<double
>&, CGeometry*, CConfig const*) const
  6.31      2.96     0.27                             CSysMatrix<double>::Gauss_Elimination(double*, double*) const
  4.44      3.15     0.19                             CSysMatrix<double>::MatrixVectorProduct(CSysVector<double> const&, CSysVector<double>&, CGeo
metry*, CConfig const*) const
```

Tracy Output



Objective - II

Performance Bottleneck Analysis[3]



Key Findings

- **68.8%** of execution time spent in compute functions
- ComputePsi: 39.46% self-cost
- ComputeChi: 29.34% self-cost

Optimization Strategy

Focus on compute functions for maximum performance impact

[3] Profiled using Valgrind/KCacheGrind

Optimization: Reducing Memory Access Overhead

Problem Identified:

- Functions show extensive vector element access patterns
- Current 3D vector structure introduces overhead

```
mlpdouble ComputePsi(std::size_t iLayer, std::size_t iNeuron,
                     std::size_t jInput) const {
  mlpdouble psi = 0;
  for (auto jNeuron = 0u; jNeuron < total_layers[iLayer - 1]->GetNNeurons();
       jNeuron++) {
    psi += weights_mat[iLayer - 1][iNeuron][jNeuron] *
           total_layers[iLayer - 1]->GetdYdX(jNeuron, jInput);
  }
  return psi;
}
```

Proposed Solution: Replace:

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

with:

`mlpdouble*** weights_mat`

Benefits

Reduced indirection overhead, improved memory locality, better cache efficiency

3D Pointer Implementation

Refactoring Approach:

- Replaced 3D vector structure with triple pointer implementation
- Maintains identical indexing scheme: `weights_mat[layer][destination][source]`
- Public interface remains unchanged for existing code compatibility

Key Considerations:

- Manual memory management required (allocation and deallocation)
- Careful cleanup in reverse allocation order prevents memory leaks

Design Philosophy:

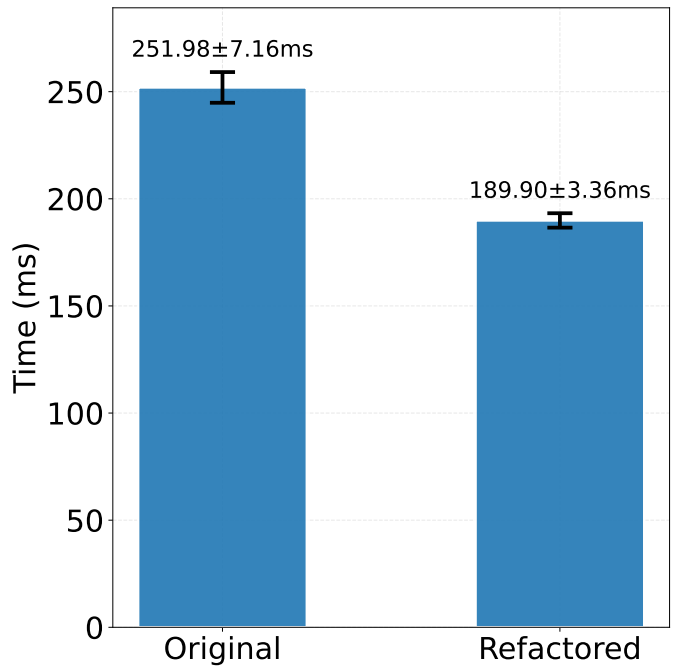
- KISS principle: Use simplest tool that meets requirements
- Weights sized once, no runtime resizing needed
- Eliminates unnecessary vector overhead for static structure

Performance Results

Test case: Evaluating a simple MLP lookup with derivative checks.

Standalone Performance

- **Original:** 251.98 $\pm$ 7.16 ms
- **Optimized:** 178.96 $\pm$ 4.56 ms
- **Speedup:** 1.41$\times$ (28.9% reduction)
- Better consistency across runs



Reality Check

Performance gains did **not** translate to full SU2 simulations

Optimizing Neural Network Inference Functions

Optimization Goals

Improve performance and numerical precision in neural network computation functions

Key Strategies

1. **Pre-extraction of frequently accessed data**
   - Cache weight vectors and neuron counts before loops

```
const auto& w=weights_mat[iLayer - 1][iNeuron];
```

   - Minimize repeated pointer dereferencing
2. **FMA instruction usage**
   - Replace separate multiply + add with `std::fma()`

```
x = std::fma(w[j], total_layers[iLayer - 1]->
    GetOutput(j), x);
```

   - Better numerical precision and potential performance gains

FindVariableIndices Function Optimization

Problem

Original implementation used nested loops to map variable names to ANN indices

Original Approach:

- Nested loop structure
- Outer loop: ANN variables
- Inner loop: Variable names
- Time complexity: $O(N \times M)$

New Approach:

- Hash map (`std::unordered_map`)
- $O(1)$ average lookup time
- Single iteration through ANN variables
- Time complexity: $O(N + M)$

GSoC 2025 Contributions Summary

Completed & Merged

- **Tracy Profiler Integration:** Low-overhead profiling tool with build system integration
- **Documentation:** Website documentation and usage instructions for Tracy with SU2

Completed & Under Review

- **Variable Mapping Optimization:** Hash-based lookup replacing nested loops
- **Network Evaluation Refactor:** Triple-pointer weight storage, cached neuron counts

Key Insights & Future Work

Lessons Learned

Standalone benchmarks may not reflect real application performance due to:

- Different computational contexts
- Memory access patterns
- System load variations

Next Steps

- Develop representative test cases matching SU2 environment
- Evaluate recursive computation strategies for forward pass
- Expand Tracy profiler integration and documentation

Thank you! Questions?

Divyaprakash

divyaprakash.poddar@gmail.com
divyaprakash-iitd
dpcfd.com

Feel free to connect for collaboration or questions.