

Advanced Methods for Scientific Computing

Analysis and Implementation of L-BFGS methods in Neural Networks

Giorgio Barocco, Francesco Bazzano, Tommaso Tron

Politecnico di Milano

Academic Year 2025-2026

Doxxygen link

Agenda

- 1 The Optimization Problem
- 2 Stochastic L-BFGS
- 3 Implementation and Architecture
- 4 Experimental Results
- 5 Beyond Classification: PINNs

Challenges in Neural Network Training

Training is a high-dimensional, non convex optimization problem.

- **State of the Art (SGD):**

- Computationally efficient per iteration ($O(d)$).
- **Drawback:** Relies only on first-order information (gradient). Slow convergence on ill-conditioned problems.

- **Second-Order Methods (Newton/BFGS):**

- Incorporate curvature (Hessian). Quadratic or super-linear convergence.
- **Drawback:** Prohibitive computational cost on large datasets.

Goal: Approximate Newton's direction using limited curvature information.

Two-Loop Recursion:

- ① Backward Pass
- ② Scaling
- ③ Forward pass

Line-Search: Strong Wolfe Line Search to ensure sufficient decrease and stable convergence

Goal: Combine the computational efficiency of stochastic methods with the convergence speed of Quasi-Newton methods.

Our implementation relies on two pillars for stability and efficiency:

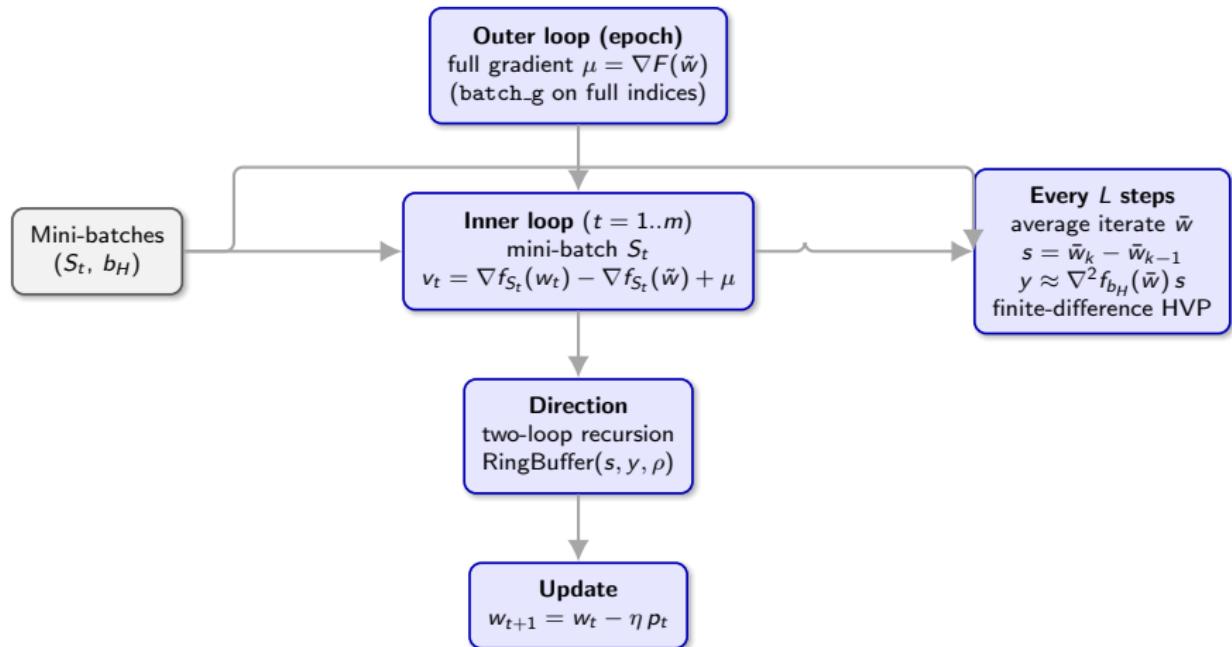
① Variance Reduction (SVRG):

- Reduces the "noise" of the stochastic gradient as the solution is approached.
- Allows for more stable steps compared to standard GD.

② Stable Curvature Updates:

- Curvature pairs (s, y) are computed from averaged iterates.
- Hessian-vector products are computed on mini-batches.

S-LBFGS: Control Flow (CPU Implementation)



Implemented in `src/minimizer/s_lbfgs.hpp`: variance reduction + stable curvature updates are CPU-only.

Dual Backend Architecture (High-Level)

We developed a modular C++ framework with **two concrete backends** sharing the same high-level API.

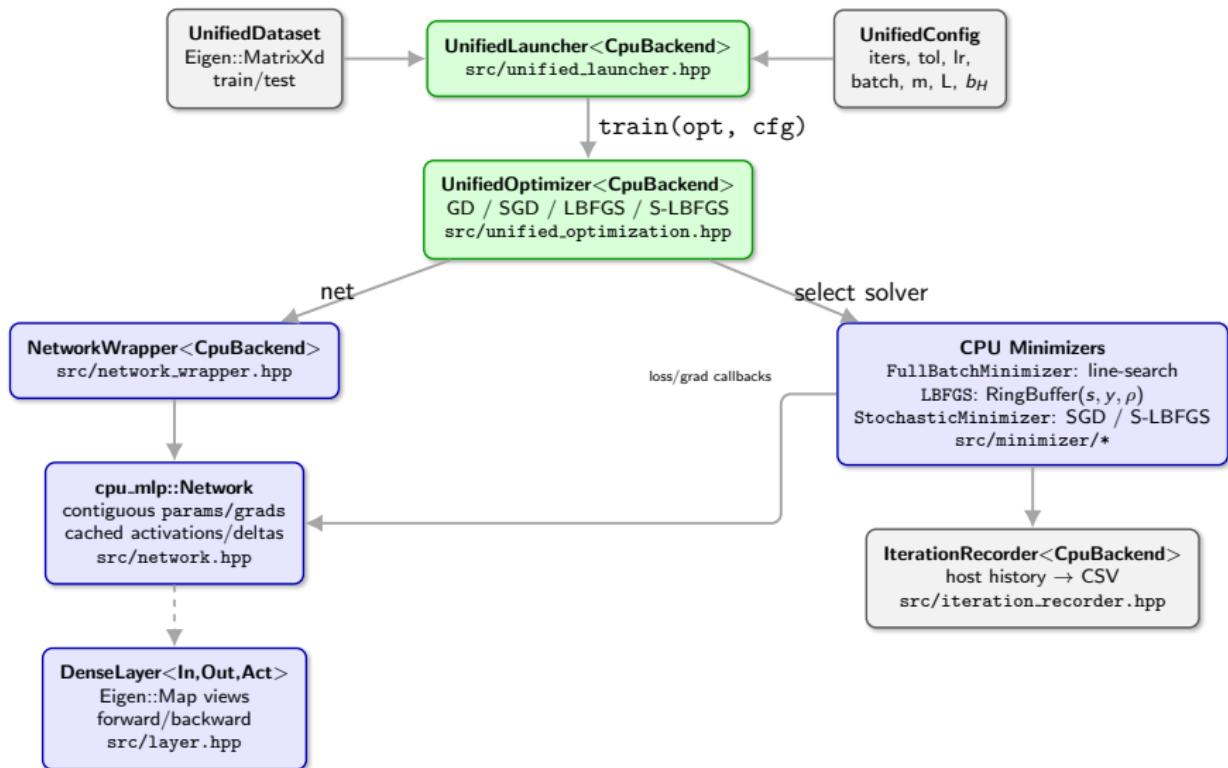
CPU Backend

- **Eigen** for batched linear algebra.
- Optional **OpenMP** to enable Eigen threading.

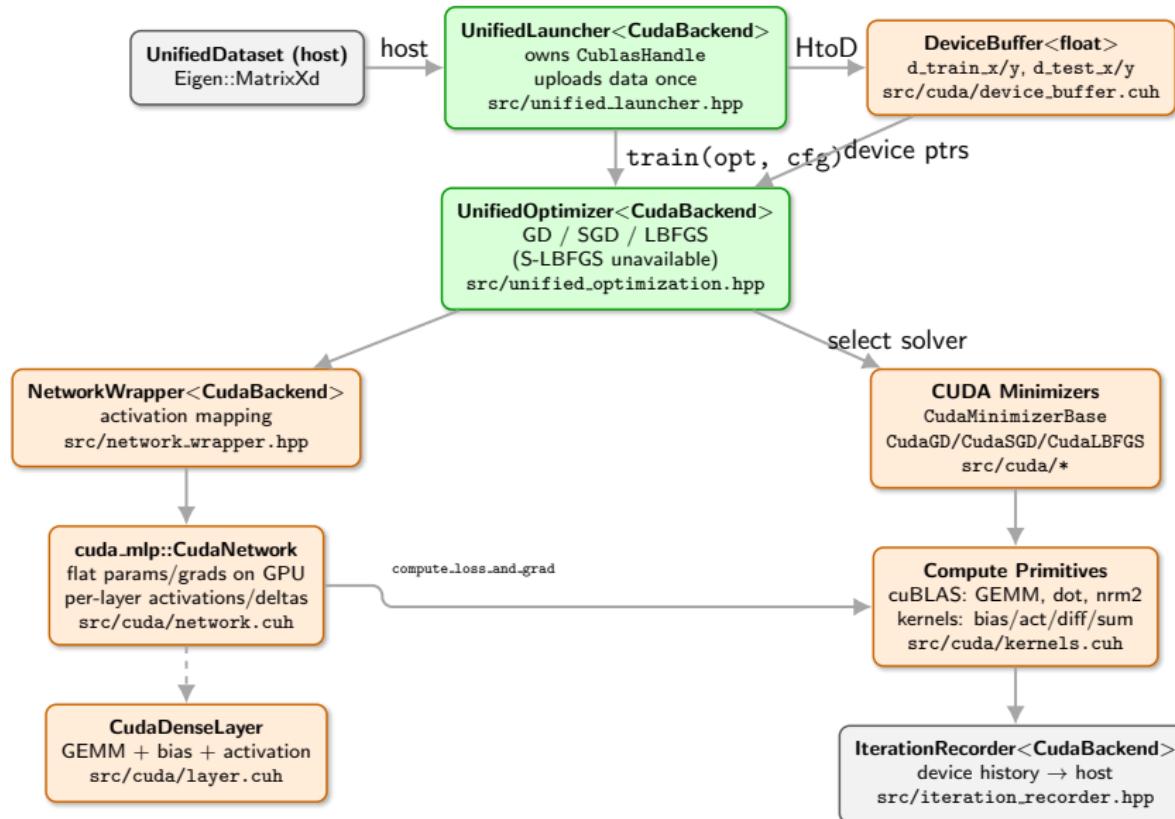
CUDA Backend

- **cuBLAS** for GEMMs + custom CUDA kernels.
- RAII GPU resources:
`DeviceBuffer`, `CublasHandle`.

CPU Backend: Modules and Data Flow



CUDA Backend: Modules and Data Flow



Architectural Patterns & Optimizations

Memory Layout Strategy

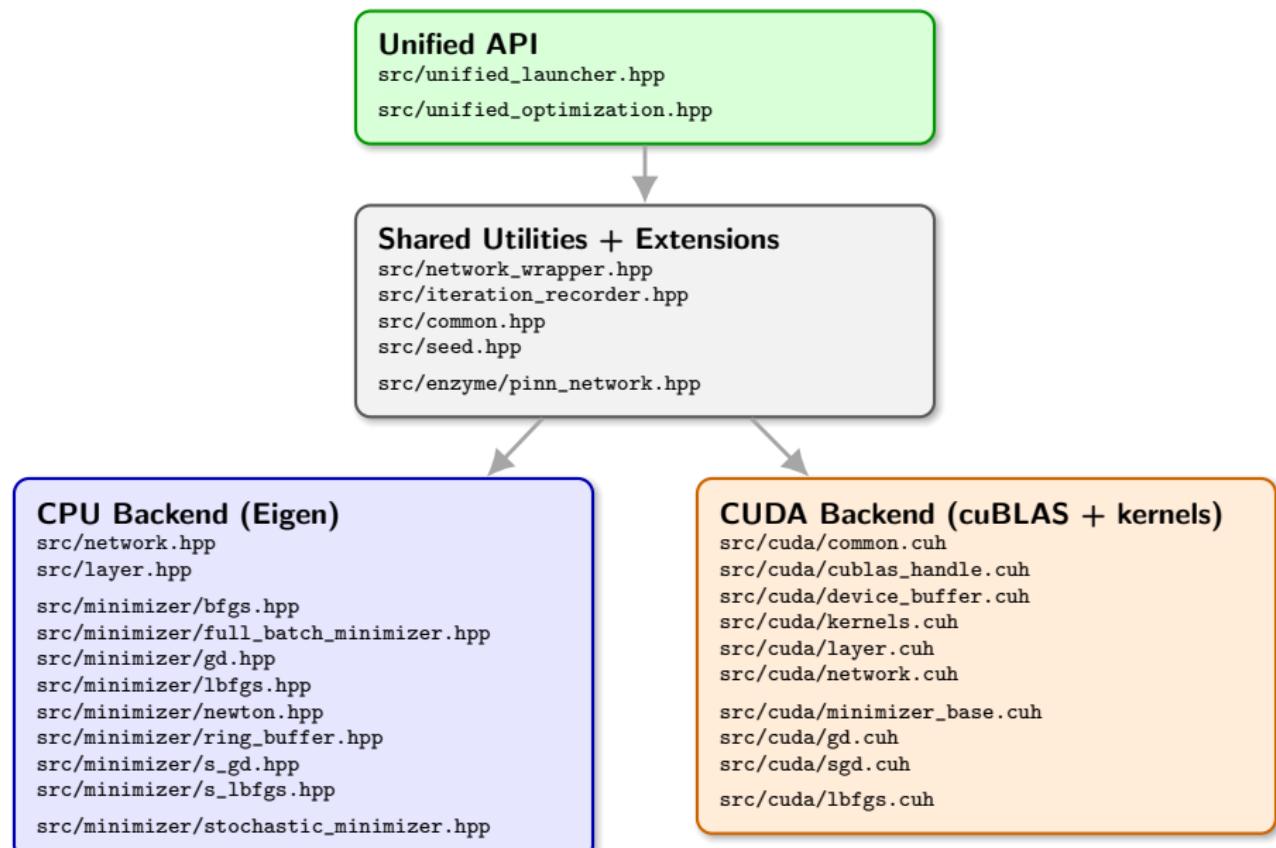
- **Problem:** Optimizers (L-BFGS) need flat vectors; Networks are structured layers.
- **Solution:** Allocate one contiguous block (Host or Device). Layers own **views** (pointers/maps) into this block.
- **Benefit:** $O(1)$ alignment for minimizers, structured access for layers.

Template Variance

- **Problem:** CPU needs double precision (stability); GPU needs float (throughput).
- **Solution:** UnifiedLauncher<Backend> selects the numeric type:
 - CpuBackend \rightarrow double
 - CudaBackend \rightarrow float

Hybrid Control Flow: CPU backend handles complex logic (branching/S-LBFGS control) efficiently; CUDA backend focuses on massive SIMD throughput for GEMMs.

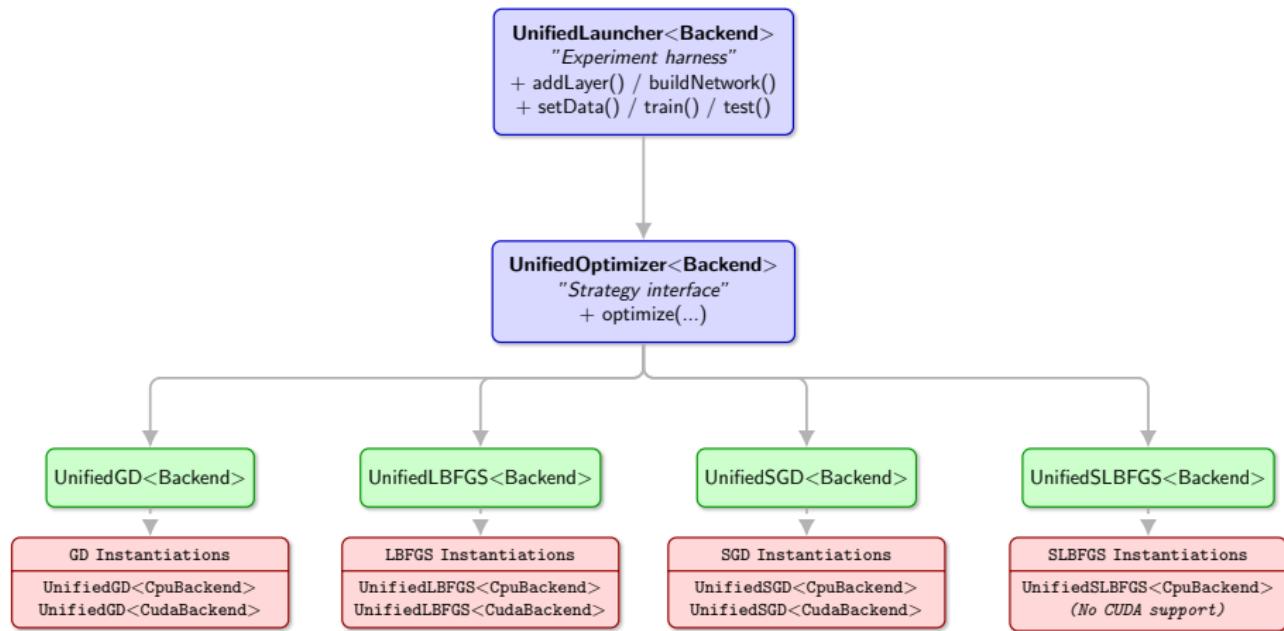
Project File Map (Source Tree)



Test Pattern

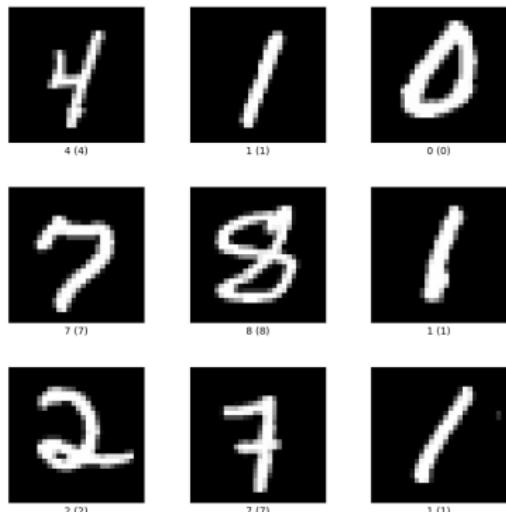
```
1 // 1) Setup
2 using Backend = CpuBackend;
3 UnifiedLauncher<Backend> launcher;
4
5 // 2) Build network
6 launcher.addLayer<in, hidden, Activation>();
7 launcher.addLayer<hidden, out, Linear>();
8 ...
9 launcher.buildNetwork();
10
11 // 3) Load dataset + bind
12 UnifiedDataset data;
13 data.train_x = Loader::loadImages();
14 ...
15 launcher.setData(data);
16
17 // 4) Train and Test
18 for each experiment e:
19     cfg <- makeConfig(e)
20     opt <- LBFGS<Backend>
21     launcher.train(opt, cfg)
22     launcher.test()
```

Unified Architecture for Testing

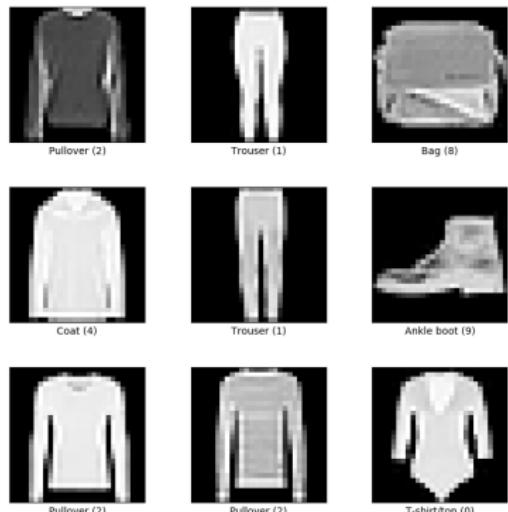


Training set

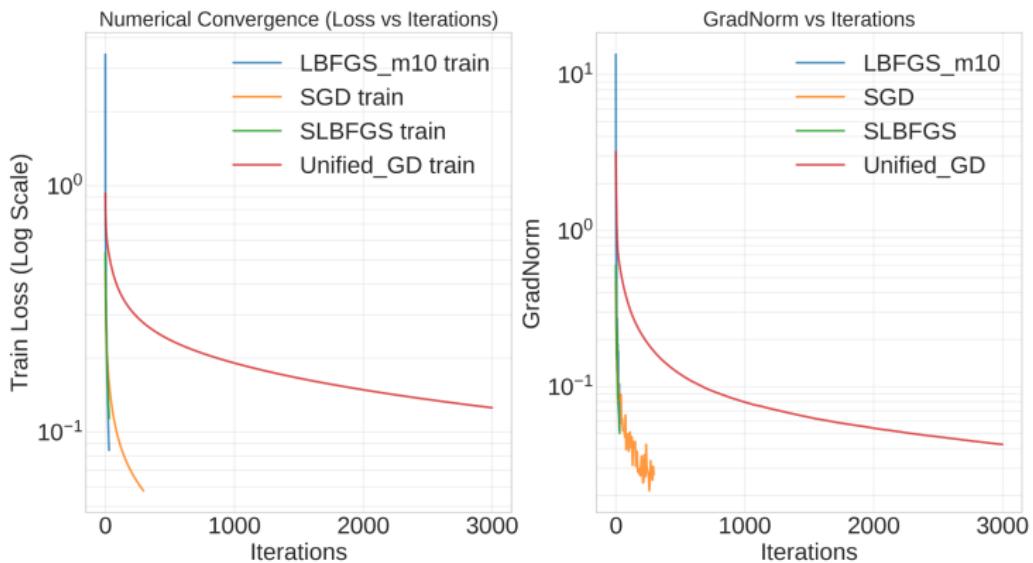
MNIST



Fashion - MNIST



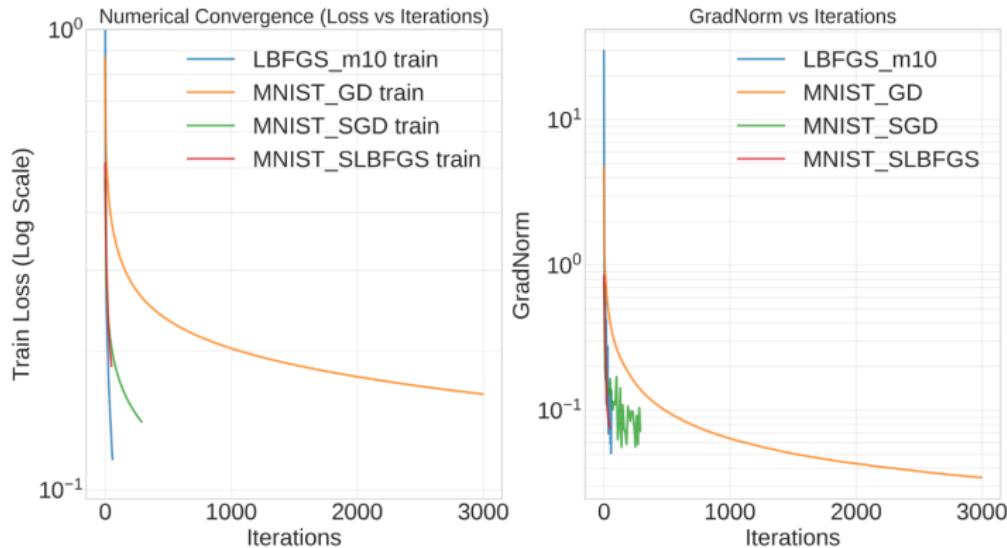
Results: MNIST (CPU)



Setup: $784 \rightarrow 128 \rightarrow 10$, $N = 5,000$ (CPU)

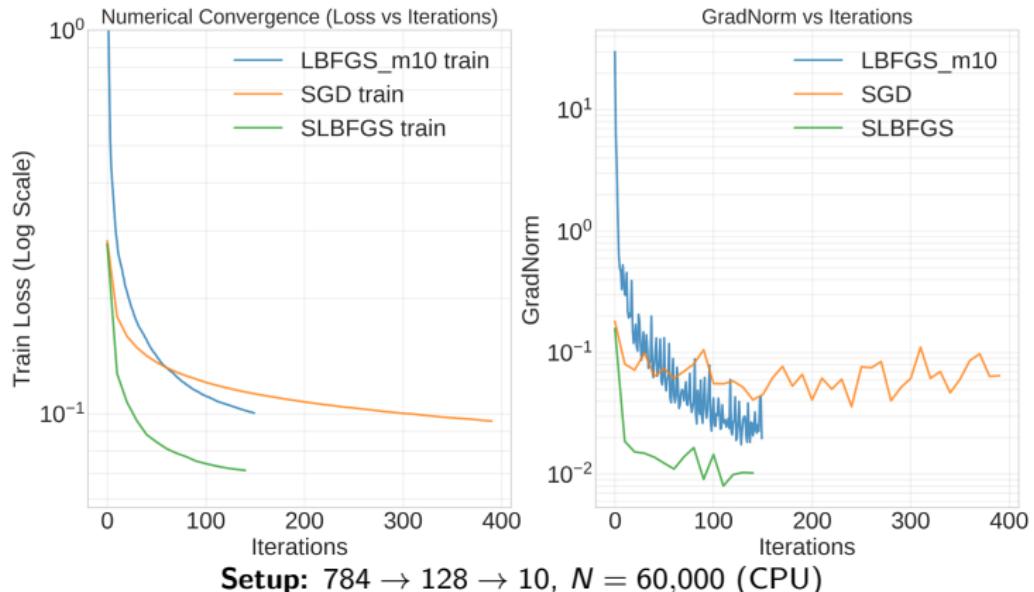
Solver	Train Acc. (%)	Test Acc. (%)
GD	91.98	89.50
SGD	97.64	93.15
S-LBFGS	93.42	90.94
L-BFGS ($m = 10$)	94.30	92.11

Results: Fashion-MNIST (CPU)



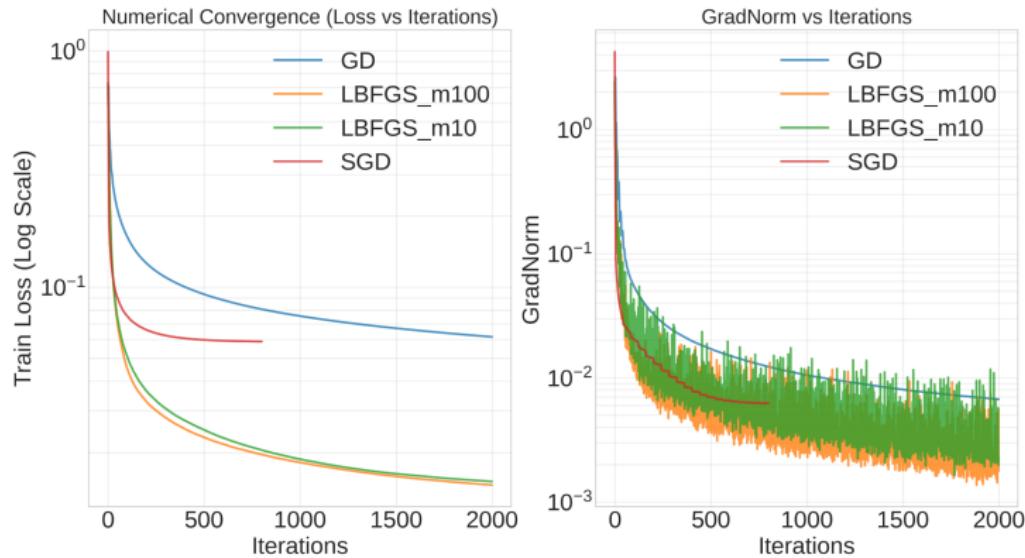
Solver	Train Acc. (%)	Test Acc. (%)
GD	84.40	79.58
SGD	86.40	81.42
S-LBFGS	83.84	79.72
L-BFGS ($m = 10$)	88.10	82.38

Results: Fashion-MNIST (CPU, Full Dataset)



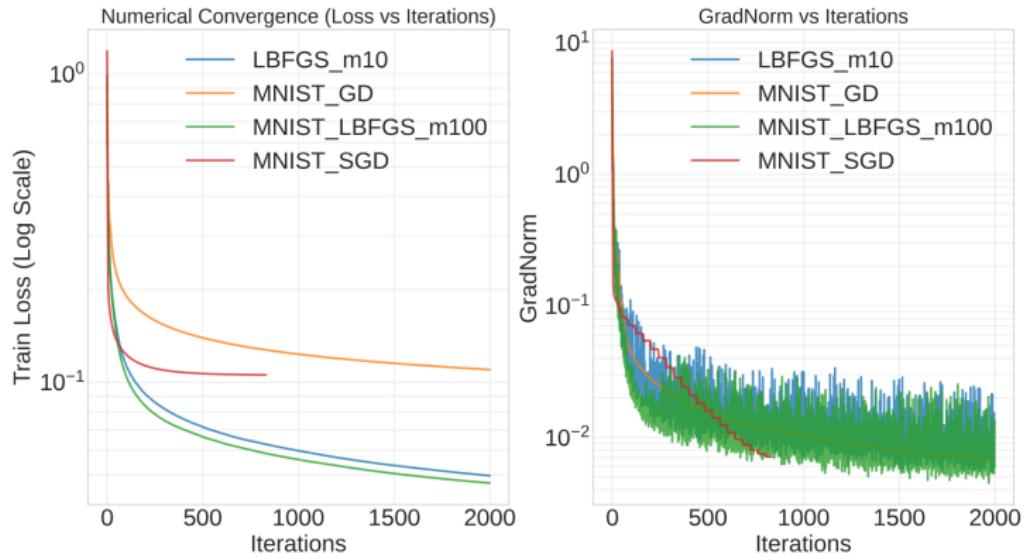
Solver	Train Acc. (%)	Test Acc. (%)
SGD	89.43	86.98
L-BFGS ($m = 10$)	88.59	86.62
S-LBFGS	93.46	87.98

Results: MNIST (GPU)



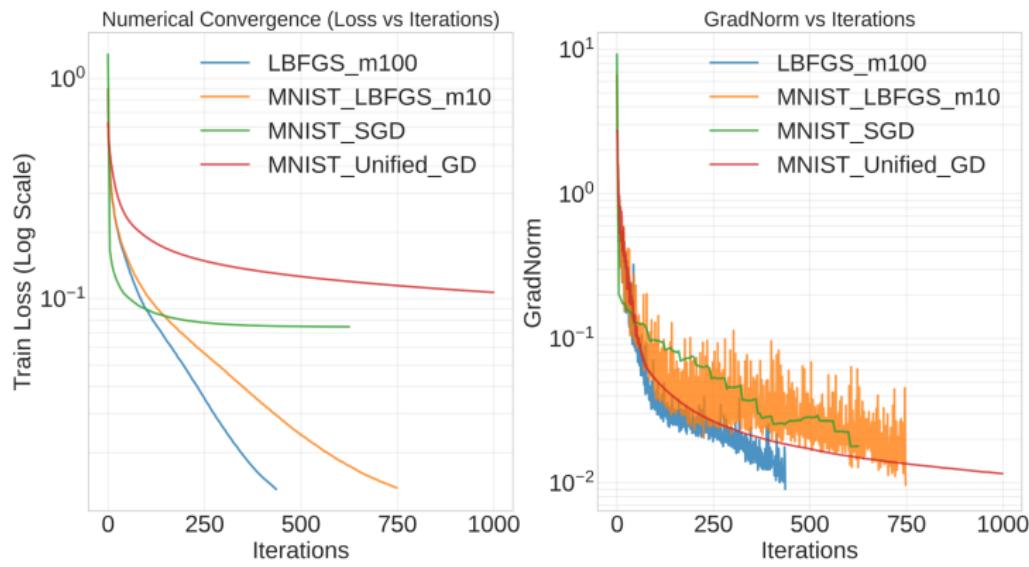
Setup: $784 \rightarrow 128 \rightarrow 10$, $N = 60,000$ (GPU)

Results: Fashion-MNIST (GPU)



Setup: $784 \rightarrow 128 \rightarrow 10$, $N = 60,000$ (GPU)

Results: Fashion-MNIST (GPU, Deep Network)



Setup: $784 \rightarrow 256 \rightarrow 128 \rightarrow 64 \rightarrow 10$

Solver	Train Acc. (%)	Test Acc. (%)
GD	87.55	85.37
SGD	91.37	88.16
L-BFGS ($m = 10$)	98.32	88.75
L-BFGS ($m = 100$)	98.40	88.30

Scalability on Multi-Core CPU

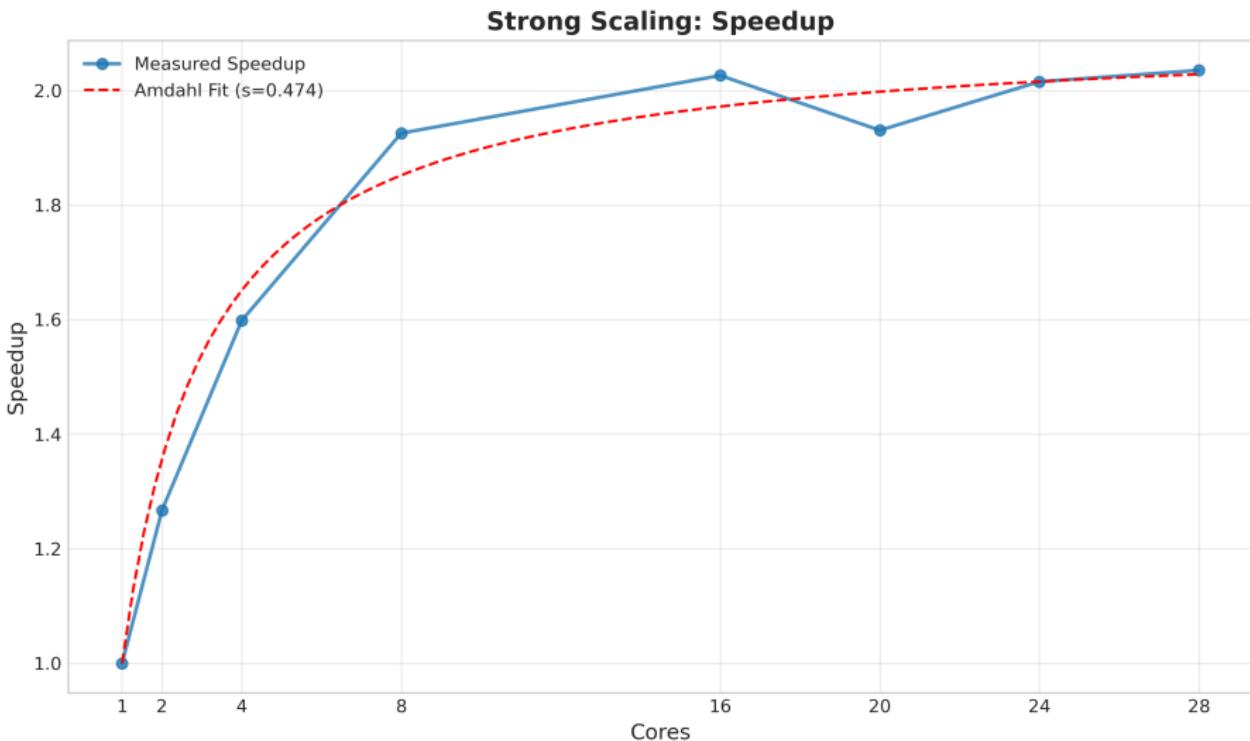
Speedup analysis of S-LBFGS on a 28-core cluster node.

- **Memory-Bound Behavior:**
- Speedup saturates at approximately **2.1x** (Amdahl's Law).
- About 47% of time is dominated by serial components or memory access.

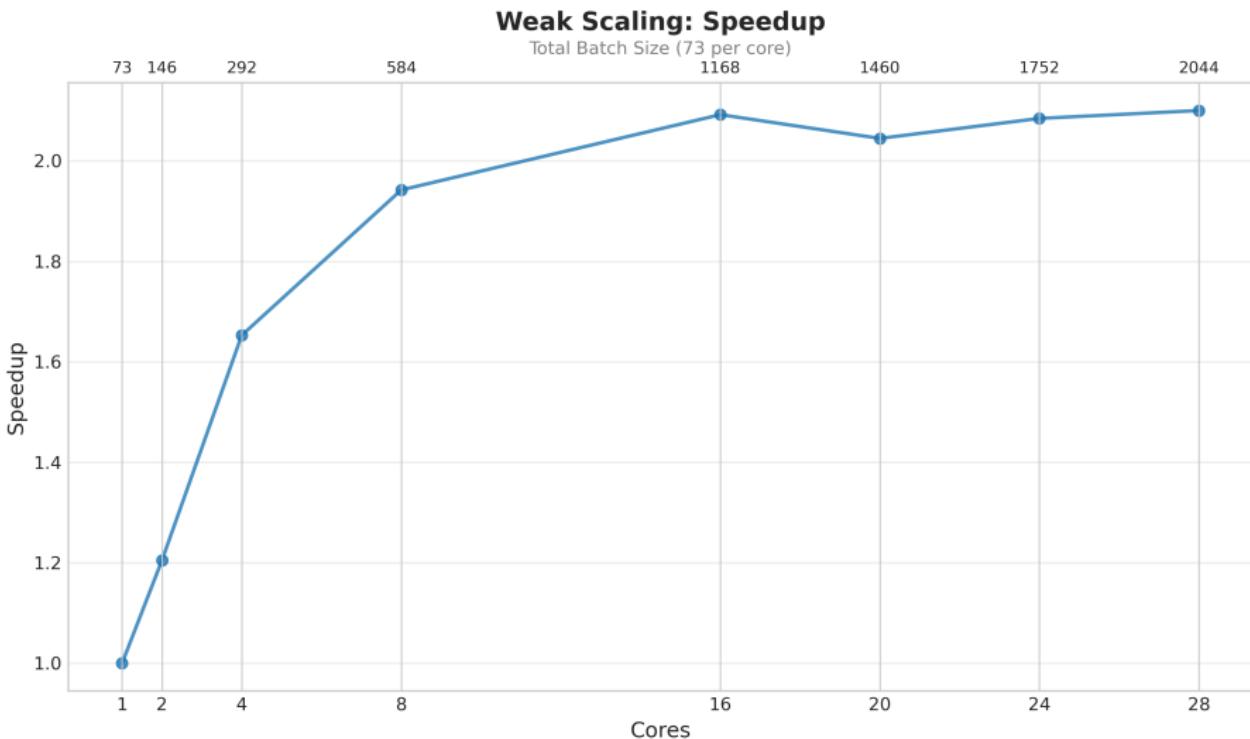
Hardware Conclusion:

For small/medium networks the CPU is competitive, but memory bandwidth quickly becomes the bottleneck for parallel stochastic algorithms.

Scalability CPU: Strong Scaling of S-LBFGS



Scalability CPU: Weak Scaling of S-LBFGS



Beyond Classification: PINNs

Application to Burgers' Equation:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \quad x \in (-1, 1), t \in [0, 1]$$

$$u_0(x) = \sin(\pi x) \quad \forall x \in (-1, 1)$$

$$u(-1, t) = u(1, t) = 0 \quad \forall t \in [0, 1]$$

$$F_{\text{PDE}} = \frac{1}{N_{\text{PDE}}} \sum_{i=1}^{N_{\text{PDE}}} \left(\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} - \nu \frac{\partial^2 u}{\partial x^2} \right)^2$$

Loss Function

$$F(\omega) = w_{\text{IC}} F_{\text{IC}} + w_{\text{BC}} F_{\text{BC}} + w_{\text{PDE}} F_{\text{PDE}}$$

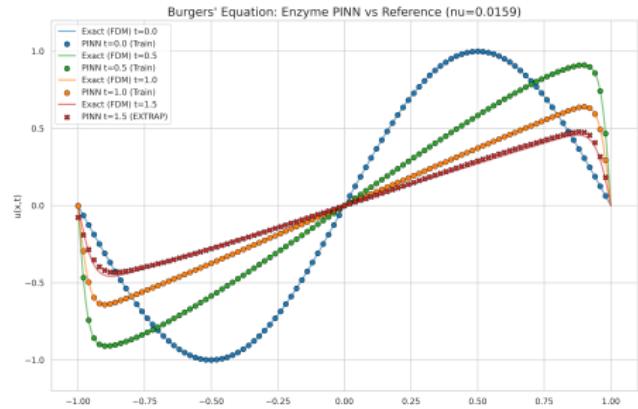
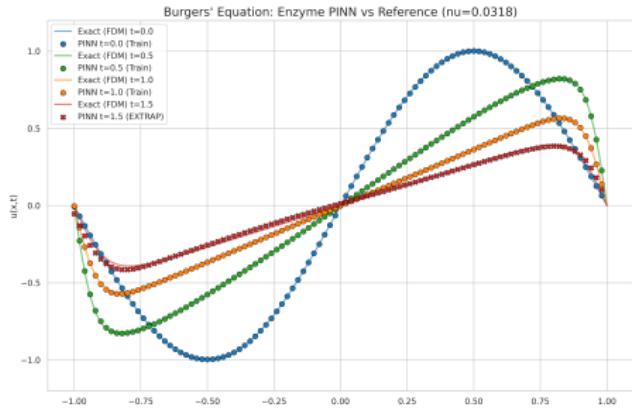
$$F_{\text{PDE}} = \frac{1}{N_{\text{PDE}}} \sum_{i=1}^{N_{\text{PDE}}} \left(\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} - \nu \frac{\partial^2 u}{\partial x^2} \right)^2$$

$$F_{\text{BC}} = \frac{1}{N_{\text{BC}}} \sum_{i=1}^{N_{\text{BC}}} (u(x_i, t_i) - g(x_i, t_i))^2$$

$$F_{\text{IC}} = \frac{1}{N_{\text{IC}}} \sum_{i=1}^{N_{\text{IC}}} (u(x_i, 0) - u_0(x_i))^2$$

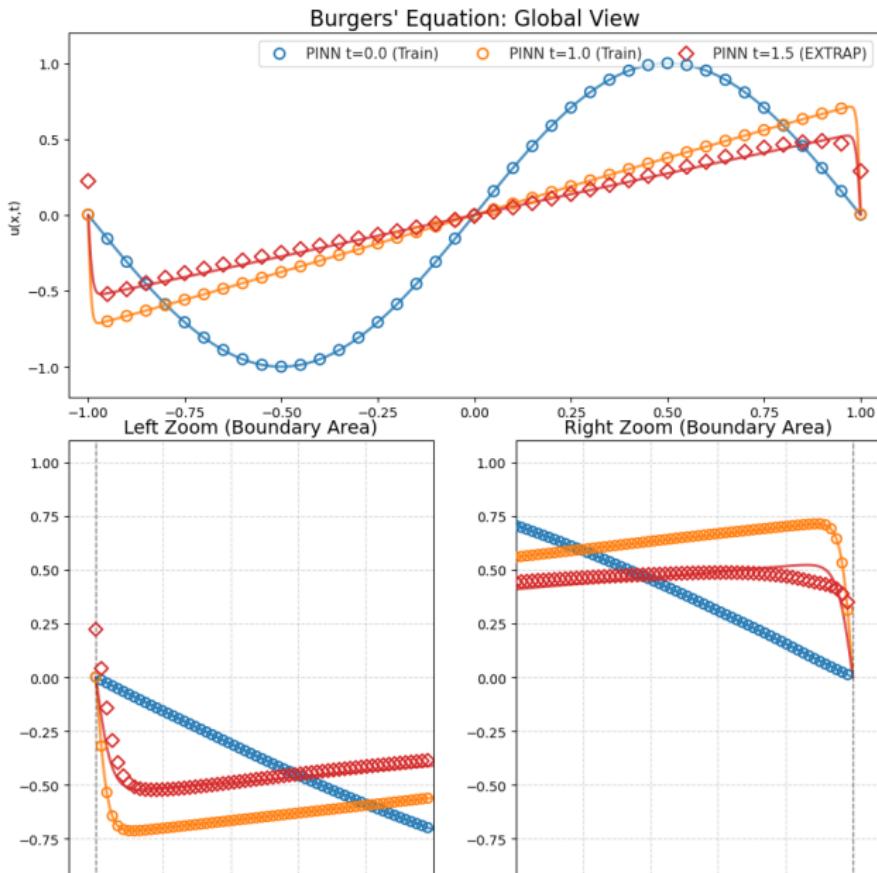
PINN Results: Burgers' Equation

- **Optimizer Strategy:** L-BFGS
 - Good convergence on physics based loss compared to standard GD.
- **Prediction** from the PINN at $t = 1.5\text{s}$
- **Outcome:** The model accurately captures the solution dynamics and shock formation across different viscosity regimes (ν).



- Solid Line: Exact analytical solution.
- Dots: Neural Network estimation (PINN output).

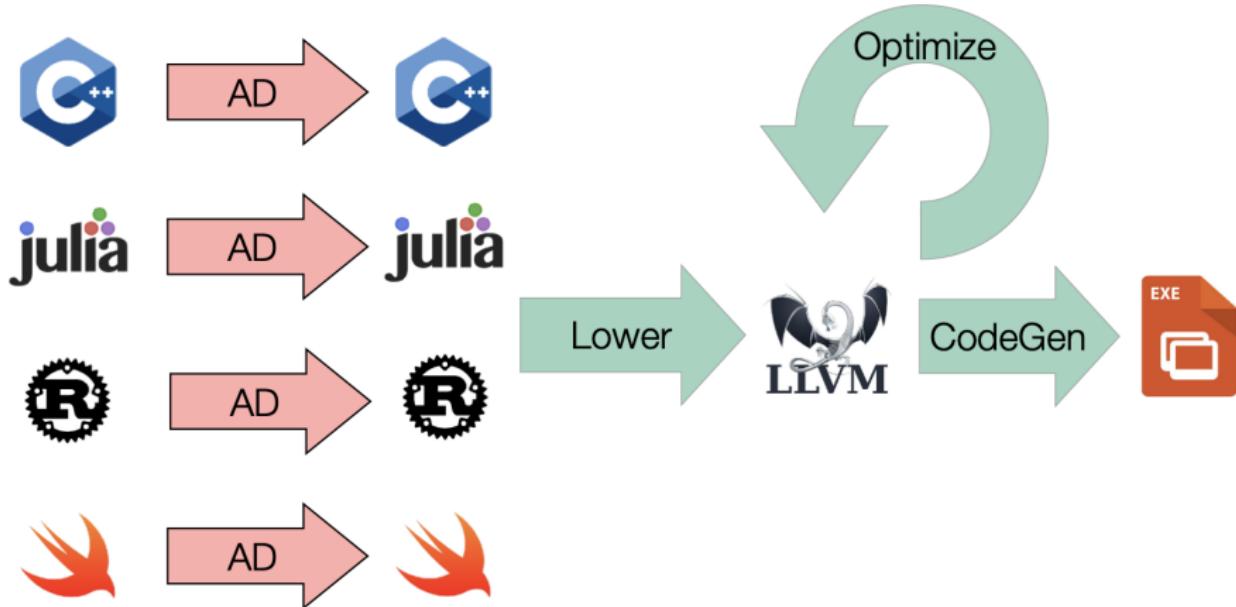
PINN results



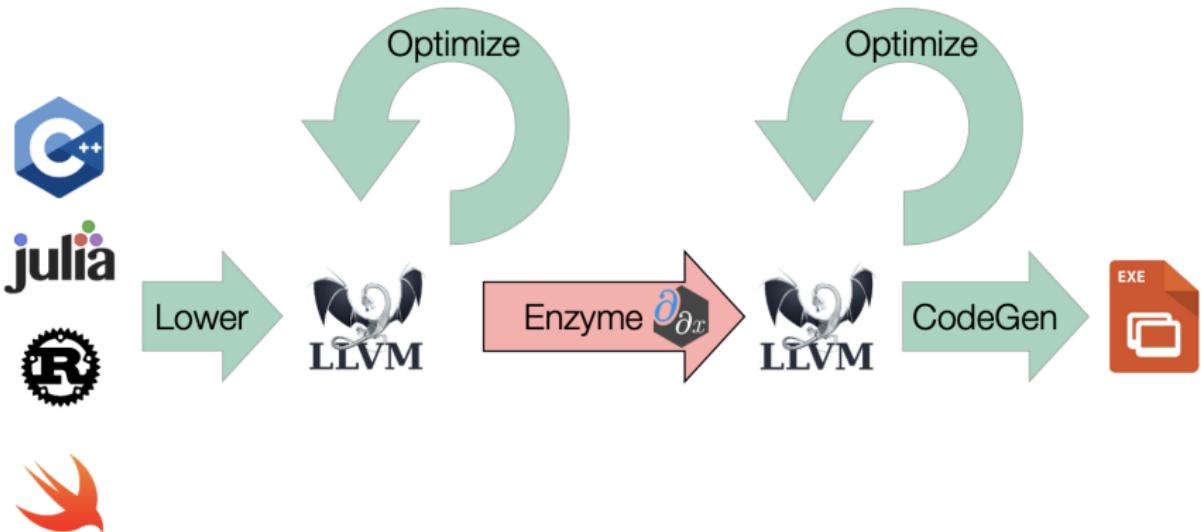
The Differentiation Challenge:

- Standard frameworks (AutoDiff) fail with nested high-order derivatives in the loss function.
- **Solution:** Integration of **EnzymeAD** (LLVM IR level).
- Enables differentiation through the derivative code itself without breaking the computation graph.
- An implementation that would also work on GPU with CUDA

Existing AD Flow



EnzymeAD Flow



EnzymeAD

An example from EnzymeAD docs:

 $O(n^2)$

```
for i=0..n {  
    out[i] /= mag(in)  
}
```

Optimize

 $O(n)$

```
res = mag(in)  
for i=0..n {  
    out[i] /= res  
}
```

AD

 $O(n)$

```
d_res = 0.0  
for i=n..0 {  
    d_res += d_out[i]...  
}  
Vmag(d_in, d_res)
```

 $O(n^2)$ $O(n^2)$

```
for i=0..n {  
    out[i] /= mag(in)  
}
```

AD

```
for i=n..0 {  
    d_res = d_out[i]...  
}  
Vmag(d_in, d_res)
```

EnzymeAD

While working on the understanding and implementation of PINNs on a CUDA backend using EnzymeAD:

```
/home/gio/amsc_temp/tests/mnist/../../src/cuda/network.cuh:18:68: required from here
/usr/include/c++/13/bits/stl_construct.h:197:1: internal compiler error: Segmentation fault
 197 |   }
    | ^
0xd32248 crash_signal
  ../../src/gcc/toplev.cc:314
0x70cca744251f ???
  ./signal/../../sysdeps/unix/sysv/linux/x86_64/libc_sigaction.c:0
0x1697e1b lookup_page_table_entry
  ../../src/gcc/ggc-page.cc:630
0x1697e1b ggc_set_mark(void const*)
  ../../src/gcc/ggc-page.cc:1550
0x1690eed gt_ggc_mx_lang_tree_node(void*)
  ./gt-cp-tree.h:107
0x16922d5 gt_ggc_mx_lang_tree_node(void*)
  ./gt-cp-tree.h:439
0x16910e1 gt_ggc_mx_lang_tree_node(void*)
  ./gt-cp-tree.h:418
0x16912bd gt_ggc_mx_lang_tree_node(void*)
  ./gt-cp-tree.h:569
0x169a3f0 gt_ggc_mx(tree_node*&)
  /build/gcc-13-IVzKaI/gcc-13-13.1.0/build/gcc/gtype-desc.cc:1806
0x169a3f0 void at_gac_mx<tree_node*>(vec<tree_node*, va_gc, vl_embed*>)
```

Thank you!