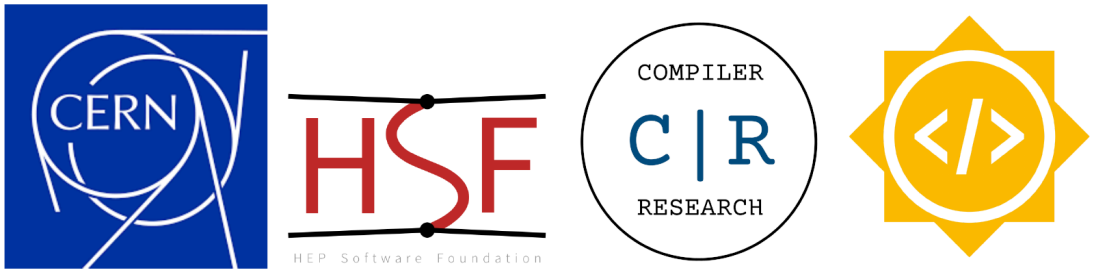

Clad as a First-Class Gradient Engine in LibTorch

Draft Proposal for Google Summer of Code 2026

HEP Software Foundation

Applicant: Kacent Huang 黄嘉诚
Email: fogsong@gmail.com
University: Nanjing University
Timezone: Asia/Shanghai (UTC+8)
GitHub: https://github.com/fogsong233

Synopsis

This proposal targets the HSF 2026 idea *Clad as a first class gradient engine in libtorch*, a 350-hour project intended to make compiler-generated gradients available from LibTorch and, eventually, easier to reuse from the ROOT ecosystem.[1]

My goal is to build a practical and well-scoped integration path that allows a LibTorch C++ training loop to call Clad-generated derivative code for a small but meaningful subset of workloads. The primary implementation path is to wrap Clad-generated backward logic inside `torch::autograd::Function`, because the PyTorch C++ frontend explicitly supports custom autograd functions and presents them as the standard way to integrate optimized forward and backward code in extensions.[2] If time permits and the API proves cleaner, I will also evaluate a custom-operator path based on the PyTorch C++ extension mechanisms.[3]

This proposal is intentionally scoped as a proof of concept, not as a general replacement for LibTorch autograd. A realistic first milestone is to differentiate selected training kernels or small model components written in C++, expose them to LibTorch, and measure where compiler-generated derivatives are correct, maintainable, and competitive. This produces a concrete result for mentors and also establishes a clear baseline for future work on broader operator coverage, deeper ROOT integration, or GPU support.

---

[1] Source: HSF GSoC 2026 project page: Clad-Libtorch
[2] Source: PyTorch C++ autograd tutorial
[3] Source: PyTorch custom C++ and CUDA operators tutorial

Benefits to the Community

This project can benefit the HSF and ROOT communities in three concrete ways.

- It turns the 2026 project idea into a reproducible C++ prototype that ROOT and HSF contributors can run, inspect, and extend.
- It tests whether Clad can become a useful differentiation backend for C++ users who already prefer LibTorch or ROOT-based workflows, rather than only a research experiment.
- It produces benchmarks and engineering notes that clarify where compiler-generated derivatives are a good fit and where PyTorch's native autograd should remain the default.

The proposal is also aligned with recent Clad work. Clad is a source-transformation automatic differentiation tool for C++, already supports user-defined custom derivatives, and has recently been extended toward CUDA kernels and machine-learning-oriented workloads.[4][5].

Related Work and Scope Assumptions

The project page suggests `torch::autograd::Function` and custom ATen operators as plausible integration points. I propose to begin with `torch::autograd::Function`, because it is the smallest path to an end-to-end demo in pure C++ and because it naturally exposes a `forward`/`backward` contract that matches Clad-generated derivatives.[6]

I also assume that the first successful version will target a limited workload class, for example dense or element-wise kernels used inside a small multilayer perceptron, rather than arbitrary LibTorch graphs. This assumption is important: Clad differentiates C++ code at compile time, while LibTorch programs often rely on dynamic tensor operations, dispatch, and a very broad operator surface. A kernel-level or component-level integration is therefore a safer and more credible first milestone than a fully general autograd replacement.

The initial implementation will focus on CPU execution. PyTorch's C++ frontend is explicitly designed for both CPU and GPU use, but broad GPU support would add substantial complexity to a first integration project and should be treated as a stretch goal unless the core CPU path is already solid.[7]

Scope and Non-Goals

To keep the project feasible within one GSoC term, I want to state the scope very explicitly.

- In scope: a CPU-first prototype that demonstrates how Clad-generated derivatives can be used from a LibTorch C++ training loop on a limited, well-documented workload.
- In scope: one or two reference kernels or model components, such as a dense layer, a compact loss path, or a toy MLP composed from small differentiable C++ building blocks.
- In scope: correctness checks against native LibTorch autograd and a benchmark discussion of engineering tradeoffs.
- Not a goal: replacing LibTorch's general autograd engine or supporting arbitrary `torch::Tensor` graphs.
- Not a goal: differentiating ATen or LibTorch internals directly.
- Not a goal: full GPU support in the first version unless the CPU prototype is already stable ahead of schedule.

---

[4]Source: Clad documentation: core concepts and custom derivatives

[5]Source: GSoC 2025 Clad project blog on source-transformed gradients for CUDA kernels and LLMs

[6]Source: PyTorch custom autograd in C++

[7]Source: PyTorch C++ frontend documentation

This scoping follows from the different roles of the two systems. Clad is a compile-time source-transformation AD tool, while LibTorch exposes a dynamic tensor and operator ecosystem. The most credible integration point for a first project is therefore at the level of user-defined C++ kernels or components, not at the level of the full LibTorch operator universe.

Technical Approach

The project will be organized as an incremental path from a minimal proof of concept to a documented benchmarked prototype.

1. Define a narrow baseline.

I will begin by selecting one or two differentiable kernels or model components that are small enough to reason about and test rigorously. Candidate examples include a dense layer, an activation-plus-loss combination, or another compact training kernel that can be expressed as ordinary C++ loops over contiguous memory or tensor accessors. In parallel, I will prepare a LibTorch baseline that uses native autograd for the same workload so that correctness and performance can be compared fairly. I will prefer a kernel or component whose implementation is fully visible at compile time, because that is a much better fit for Clad than trying to differentiate opaque library internals.

2. Generate and validate gradients with Clad.

For the chosen kernels, I will use Clad to emit derivative code and validate the generated gradients first outside LibTorch, then against finite differences and the corresponding LibTorch autograd results. This stage is important because it separates "is the differentiated kernel correct?" from "is the framework integration correct?".

3. Build a LibTorch integration layer.

The primary integration path is a custom `torch::autograd::Function` subclass whose `forward` method calls the reference C++ kernel and whose `backward` method calls the Clad-generated derivative function. Saved tensors and metadata will be kept minimal so that the interface stays understandable and measurable. If this approach works cleanly, the result should already demonstrate the main idea of the project: compiler-generated gradients being used inside a LibTorch training loop written in C++. This keeps the proof of concept at the user-extension boundary that LibTorch officially supports, instead of requiring changes to the framework's internal autograd machinery.

4. Evaluate an optional operator-registration path.

If the `torch::autograd::Function` route is stable early enough, I will evaluate whether registering a custom operator yields a cleaner user-facing API or lower integration overhead. PyTorch's official custom-operator workflow already supports C++ operators and gradient registration, so this is a realistic stretch goal rather than speculative future work.[8]

5. Benchmark, document, and package the result.

The final phase will compare the Clad-backed path with standard LibTorch autograd on the same workload and hardware. I will report correctness first, and then runtime and engineering tradeoffs such as code complexity, API clarity, and limits of the supported workload subset. I will also prepare a small tutorial and build instructions so that mentors and future contributors can reproduce the results.

[8] Source: PyTorch custom operator workflow

Deliverables

Required Deliverables

- A minimal but complete LibTorch C++ example that uses Clad-generated derivatives through `torch::autograd::Function`.
- A reference CPU workload chosen to match Clad's strengths, with source-visible kernels or model components that can be differentiated reliably at compile time.
- A correctness suite comparing gradients against LibTorch autograd and, where appropriate, finite-difference checks.
- A documented reference workload, such as a toy MLP or a compact training kernel set, with reproducible build and run instructions.
- A benchmark report describing where the Clad-backed path helps, where it does not, and what limitations remain.
- Developer-facing documentation that explains the design choices, supported scope, and extension points for future contributors.

Optional Stretch Deliverables

- An alternative integration route based on PyTorch custom operators if it improves ergonomics or performance.
- A ROOT-facing example, note, or small demo showing how the LibTorch prototype could fit into existing ROOT or HSF workflows.
- An exploratory note on what would be required to extend the prototype toward broader operator coverage or GPU execution.

Proposed Timeline

The Google Summer of Code contributor guide asks applicants to discuss project goals, deliverables, and a realistic schedule with mentors, and the 2026 program timeline starts community bonding on May 1, 2026, with coding beginning on May 25, 2026.[9][10]

Because my final exams end on July 5, 2026, and my next semester begins on August 23, 2026, I should not pretend that June is a normal coding month for me. The contributor guide explicitly says applicants should inventory outside commitments and be upfront about finals, while the time-management guide notes that medium and large projects can use extended schedules when that is agreed early with mentors and the organization.[11][12] I therefore propose an extended timeline from the outset: keep the workload as low as possible before July 5, use July 6 to August 22 as the main implementation window, and reserve the official extension period for documentation, review feedback, cleanup, and carefully scoped stretch goals. In 2026, the extended timeline runs through November 2.[13]

To make progress easy to evaluate, I am presenting the plan as a per-week timeline with explicit deliverables.

Week 1: May 1 - May 7

- Focus: start community bonding and align with mentors on the first workload to target.

---

[9] Source: GSoC contributor guide: writing a proposal
[10] Source: Google Summer of Code 2026 timeline
[11] Source: GSoC contributor guide: be upfront about finals and possible extensions
[12] Source: GSoC contributor guide: time management and flexible project lengths
[13] Source: Google Summer of Code 2026 timeline: extended timelines continue through November 2

- Deliverables: a shortlist of candidate differentiable kernels or model components and an agreed evaluation direction.

Week 2: May 8 - May 14
- Focus: read the Clad, PyTorch, and ROOT-side background in depth and turn it into an implementation plan.
- Deliverables: a concrete implementation checklist, benchmark plan, and initial test strategy.

Week 3: May 15 - May 21
- Focus: prepare the local development environment for Clad and LibTorch.
- Deliverables: reproducible setup notes for the reference build and the required toolchain versions.

Week 4: May 22 - May 28
- Focus: finish community bonding and begin the exam-constrained ramp-up with only low-risk setup work.
- Deliverables: a confirmed first source-visible kernel or model component plus a harness outline for later validation.

Week 5: May 29 - June 4
- Focus: keep communication regular during exams while making the baseline build dependable.
- Deliverables: a verified baseline build and smoke-test path for the Clad plus LibTorch environment.

Week 6: June 5 - June 11
- Focus: define the interface between Clad-generated code and `torch::autograd::Function`.
- Deliverables: a wrapper design note describing the forward/backward contract and data flow.

Week 7: June 12 - June 18
- Focus: use the remaining light-availability time for small exploratory validation work.
- Deliverables: either a tiny exploratory experiment or a refined derivative validation plan if experiments must wait.

Week 8: June 19 - June 25
- Focus: lock the first reference workload and the correctness criteria for the main coding window.
- Deliverables: mentor-aligned workload scope, acceptance criteria, and a prioritized July task list.

Week 9: June 26 - July 2
- Focus: close out the preparation backlog and queue the first implementation steps.
- Deliverables: a dependency-ordered implementation checklist for the full-time period beginning after July 5.

Week 10: July 3 - July 9
- Focus: move into full-time work after exams and implement the first differentiable kernel or model component.
- Deliverables: the first Clad-generated derivative path compiling successfully for the chosen component.

Week 11: July 10 - July 16
- Focus: validate the generated derivatives outside LibTorch before framework integration.
- Deliverables: standalone correctness tests comparing the generated derivatives against expected results.

Week 12: July 17 - July 23
- Focus: build the first minimal end-to-end path through `torch::autograd::Function`.
- Deliverables: an isolated prototype where LibTorch invokes Clad-generated backward logic successfully.

Week 13: July 24 - July 30
- Focus: integrate the differentiated component into a complete LibTorch C++ training loop.
- Deliverables: a training loop that runs the reference workload through the custom autograd path.

Week 14: July 31 - August 6
- Focus: add correctness checks against native autograd and finite-difference baselines.
- Deliverables: a correctness report or test suite comparing Clad-backed gradients with trusted baselines.

Week 15: August 7 - August 13
- Focus: expand from the minimal prototype to the main reference workload and run the first benchmarks.
- Deliverables: the main workload wired into the prototype plus an initial benchmark report draft.

Week 16: August 14 - August 20
- Focus: use the last full-time summer week to stabilize the main workload and identify the dominant overheads.
- Deliverables: a hotspot list, cleaner tests, and at least one targeted improvement to the reference path.

Week 17: August 21 - August 27
- Focus: transition into the semester and decide whether the custom-operator stretch goal is still worthwhile.
- Deliverables: a mentor-reviewed scope decision plus a stabilized primary prototype.

Week 18: August 28 - September 3
- Focus: start the extended timeline with documentation and cleanup around the working prototype.
- Deliverables: draft supported-scope documentation explaining the current workload assumptions and limitations.

Week 19: September 4 - September 10
- Focus: incorporate feedback and strengthen regression coverage around the main path.
- Deliverables: additional automated tests for the primary workload and known edge cases.

Week 20: September 11 - September 17
- Focus: improve reproducibility so mentors and contributors can rerun the prototype easily.
- Deliverables: a reproducible run script or setup guide for building, testing, and benchmarking the prototype.

Week 21: September 18 - September 24
- Focus: refine the benchmark methodology and compare against native LibTorch autograd.
- Deliverables: updated result tables with methodology notes and a clearer performance narrative.

Week 22: September 25 - October 1
- Focus: polish the integration layer and resolve review feedback.
- Deliverables: a cleaner wrapper API, revised documentation, and resolved code review comments.

Week 23: October 2 - October 8
- Focus: finalize developer-facing documentation and evaluate a ROOT-facing note if the core path is already stable.
- Deliverables: a setup guide plus either a ROOT-facing note/demo or a more polished core documentation set.

Week 24: October 9 - October 15
- Focus: attempt stretch work only if the main prototype is already dependable.
- Deliverables: either a custom-operator spike, an extra workload experiment, or a future-work note explaining why stretch work was deferred.

Week 25: October 16 - October 22
- Focus: stabilize stretch work if it exists, otherwise finish the final documentation pass.
- Deliverables: final developer and user documentation, with stretch-goal results only if they do not risk the core prototype.

Week 26: October 23 - October 29
- Focus: prepare the final handoff package and integrate the last round of mentor feedback.
- Deliverables: a polished benchmark report, reproducibility appendix, and concise follow-up backlog.

Week 27: October 30 - November 2
- Focus: use the last few days as submission buffer and final review time.
- Deliverables: the final submission and a short handoff note summarizing the working prototype and its current limits.

Risks and Mitigations
The main technical risk is scope. LibTorch's operator surface is broad, and not every interesting model component will map cleanly to compile-time differentiation. I mitigate this by starting from a narrow workload class and by treating generality as something to earn after a correct end-to-end prototype exists.

Another risk is the boundary between source-transformed derivatives and framework-managed tensors. I mitigate this by designing the first differentiated kernels around source-visible code and simple data access patterns, validating them outside LibTorch first, and only then wrapping them inside the framework integration layer.

The final risk is schedule pressure caused by my exam period. I mitigate this by requesting an extended timeline from the start and front-loading design, setup, and small validation tasks before July 5, then doing the heaviest coding work between July 6 and August 22 while I am fully available.

Availability
I want to be explicit about availability so that the timeline is honest and easy for mentors to evaluate.

- May 1 - May 24, 2026: around 3 to 4 hours per week for community bonding, reading, planning, and environment setup.
- May 25 - July 5, 2026: around 0 to 2 hours per week during the exam period. I can stay in touch and finish lightweight preparation tasks, but I should not promise substantial implementation work in June.

- July 6 - August 22, 2026: around 35 to 40 hours per week. This will be the main implementation window for the project.
- August 23 - November 2, 2026: around 8 to 10 hours per week after the semester starts, focused on stabilization, documentation, review feedback, and stretch goals if the core prototype is already complete.

This schedule is intentionally back-loaded but still fits a large project once an extended timeline is agreed early with mentors and the organization.[14]

I am based in China Standard Time (UTC+8). I can overlap with European working hours in my afternoon and evening, and I am comfortable providing a short weekly written status update plus regular async communication between meetings.

About Me

I am a second-year undergraduate student in Computer Science at Nanjing University. My preparation for this project comes from the overlap between systems programming, C++, and machine learning.

Why I Am Interested in This Project

This project is attractive to me because it sits at a technically interesting intersection: compiler techniques, machine-learning systems, and scientific C++ software. I like the fact that the project is not only about "making something faster", but also about clarifying the engineering boundary between a source-transformation AD tool and a large ML framework. Even a carefully scoped prototype will teach us something useful about that boundary.

I also think the project is a good fit for the HSF community because the result can be evaluated in a very concrete way. Either the prototype integrates cleanly with LibTorch and produces correct gradients on a defined workload, or it does not. That makes the milestones measurable and keeps the proposal grounded.

References

- HSF GSoC 2026 project page: Clad-Libtorch
- ROOT Users Workshop 2025 contribution on using Clad as a first-class gradient engine in LibTorch
- GSoC 2025 Clad project blog on source-transformed gradients for CUDA kernels and LLMs
- Clad documentation: core concepts and custom derivatives
- PyTorch tutorial: C++ autograd
- PyTorch tutorial: custom C++ and CUDA operators
- PyTorch C++ frontend documentation
- GSoC contributor guide: writing a proposal
- GSoC contributor guide: time management and flexible project lengths
- Google Summer of Code 2026 timeline

[14] Source: GSoC admin guidance: medium and large projects can be extended and Org Admins set the project schedule