Optimizing automatic differentiation using activity analysis

IRIS-HEP Fellowship Program 2024

Andriichuk Maksym
Julius-Maximilians-Universität Würzburg

Mentors:
Vassil Vassilev, David Lange

Motivation

Clang is an efficient C, C++ and Objective-C compiler that can be a valuable tool for a math library project. Integrated with LLVM, compiles source code into highly optimized machine code, enhancing performance, which is crucial for a math library that requires efficient computation. Clad is a Clang plugin designed to provide automatic differentiation (AD) for C++ mathematical functions. It generates code for computing derivatives modifying abstract syntax tree using LLVM compiler features. AD breaks down the function into elementary operations and applies chain rule to compute derivatives of intermediate variables. Clad supports forward- and reverse-mode differentiation that are effectively used to integrate all kinds of functions.

Clad has undergone significant optimization in recent years. Still, it generates statements it never uses numerically computing derivatives. We seek to reduce the size of generated code by implementing different kinds of analysis.

Background

Clad is a source transformation based AD tool which can perform more advanced program optimization by implementing more sophisticated analyses because it has access to a rich program representation – the Clang AST. These optimizations investigate which parts of the computation graph are relevant for the AD rules. One such optimization is the To-Be-Recorded optimization which reduces the memory pressure to the clad tape data structure. TBR analysis is a part of an adjoint mode of AD. It finds variables whose present value is used in a derivative instruction and reduces the number of statements by not creating temporary variables for dependent variables that are being overwritten and not being used. Another optimization is the activity analysis optimization which discards all statements which are irrelevant for the generated code. That is, if the statements do not depend on the input/output variables of a routine in a differentiable way, they are ignored. The advantage is that this improves the performance of the generated code and reduces the phase space of features needed to be supported to enable differentiable STL, for example.

Implementation

During my internship I will focus on **activity analysis (AA)**. It is a combination of forward and backward analysis. We will operate on two sets of variables: “Varied” set contains all

variables that depend on some independent input and “Useful” contains all variables that influence some other dependent output.

In order to do this we will create a new class to keep information about the “Varied” and “Useful” variables sets using the AST visitor technology in clang. We will analyze statements for “Varied” variables the way it is implemented in the VarData class from TBRAnalyzer, since both TBR and “Varied” variables are determined after a forward-sweep. We will derive the “Useful” set of variables in a reverse-sweep, its implementation would also be relevant for the future AA extensions.

We will also use diff-dependency analysis to preliminary compute a set of all dependent variables before a certain statement. Using the transitivity of a “dependency in a differentiable way” we derive to certain relations between defined sets which at the end help to determine if certain statements are not important to add to the body of a generated derivative code.

Timeline

Week 1-2: Reading papers on AD and diff-dependency analysis, setting up a local development environment, getting familiar with the development cycle procedures in the team. **Deliverable**: Onboarding blog post for the compiler-research.org webpage.
Week 3: Investigating the relevant implementation and classes. **Deliverable:** A pull request adding a basic infrastructure of the AST visitor class responsible for the AA.
Week 4: Implementation of the initial algorithm. **Deliverable:** enhanced AA class with an initial set of tests.
Week 5: Buffer week if something takes longer than expected.
Week 6: Support of more advanced use cases such as arrays and simple STL primitives.
Week 7: Performance benchmark of the implementation. **Deliverable:** presentation about the performance and the overall status of the project at the weekly meetings project.
Week 8: Running tests and comparing efficiency with and without implemented parts. Fixing bugs and raising issues. Testing in different major workflows like ROOT’s RooFit package.
Week 9: Testing with more advanced features such as non-trivial STL constructs (HepEmShow). Adding minor changes to achieve compatibility. **Deliverable:** Report on the work done, clad uses dependency analysis by default.
Week 10: Investigate feasibility of enabling the feature to be on by default. Reading additional papers and exploring ways to implement diff-liveness analysis as a way to reduce the number statements in reverse-mode.
Week 11: Develop more documentation, tests and performance benchmarks.
Week 12: Working on optimizing TBR analysis using dependency analysis. Creating documentation for the newly added algorithms. **Deliverable:** Complete report on the project.
Week 13: Summarizing all the work, gathering data for a final report, showing the difference in performances. Elaborating on what parts of activity analysis could be implemented next. **Deliverable:** Present the final work; write a blog post.

References

[1] L.Hascoët, V.Pascual. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Transactions on Mathematical Software 39(3):20:1-20:43.*