Utilize second order derivatives from Clad in ROOT

Author

Baidyanath Kundu (sudo-panda)

Manipal Institute of Technology

Computer Science Undergraduate

kundubaidya99@gmail.com

Mentors

Vassil Vassilev
vvasilev@cern.ch

Ioana Ifrim
ioana.ifrim@cern.ch

Abstract

In mathematics and computer algebra, automatic differentiation (AD) is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. Automatic differentiation is an alternative technique to Symbolic differentiation and Numerical differentiation (the method of finite differences). Clad is based on Clang which provides the necessary facilities for code transformation.

ROOT is a framework for data processing, born at CERN, at the heart of the research on high-energy physics. Every day, thousands of physicists use ROOT applications to analyze their data or to perform simulations. ROOT has a clang-based C++ interpreter Cling and integrates with Clad to enable flexible automatic differentiation facility.

`TFormula` is a ROOT class which bridges compiled and interpreted code.

The project aims to add second order derivative support in `TFormula` using `clad::hessian`. The PR that added support for gradients in ROOT is taken as a reference and can be accessed here.

Optionally, if time permits, this project will also convert the gradient function into CUDA device kernels.

Basic Information

Personal Information

Name:	Baidyanath Kundu
Email:	kundubaidya99@gmail.com
Mobile:	+91 8292491638
Time Zone:	IST (GMT + 5:30)
GitHub	@sudo-panda
University:	Manipal Institute in Technology, Manipal
Major:	Computer Science
Current year:	3rd-year undergraduate
Degree:	Bachelor of Technology
Availability:	I am available for the entire coding period

Programming Experience

I'm currently working for IRIS-HEP on a project called "Reading CMS Run 1/2 miniAOD files with ServiceX and func adl," where I'm using python ast to create ROOT C++ code to process AOD and miniAOD files.

Working on a student project called Project Manas, where I worked on automating ground and aerial vehicles, gave me the majority of my C++ experience. I also learned about template meta-programming by contributing to Boost Geometry.

I believe I am a good choice for this project because I have some experience working with ROOT and am also interested in the development of Clad. This project will allow me to work on both Clad and ROOT at the same time, which is extremely exciting.

Development Environment

My primary IDE is CLion and it is very helpful while working with the Clang AST.

My primary OS is Pop!_OS 20.04 LTS.

I use zsh as my primary shell.

Breakdown of Tasks

Add `clad::hessian` support

On the ROOT side:

The PR needs only some minor refactoring and changes so as to work for `clad::hessian`.

1. Create hessian counterparts of gradient functions and variables:

    a. `fHessMethod` - pointer to the generated hessian method

    b. `fHessFuncPtr` - function pointer to the the generated hessian method

    c. `GetHessianFuncName()` - returns the hessian function name of the function being derived

    ```
    std::string GetHessianFuncName() const {
      assert(fClingName.Length() &&
          "TFormula is not initialized yet!");
      return std::string(fClingName) + "_hessian";
    }
    ```

    d. `HasHessianGenerationFailed()` - gives the state of hessian generation

    ```
    bool HasHessianGenerationFailed() const {
      return !fHessMethod && !fHessGenerationInput.empty();
    }
    ```

    e. `HasGeneratedHessian()` - checks if hessian has already been generated

    ```
    bool HasGeneratedHessian() const {
      return fHessMethod != nullptr;
    }
    ```

    f. `GenerateHessianPar()` - generates the hessian function

```
bool TFormula::GenerateHessianPar()
{
   // We already have generated the hessian.
   if (fHessMethod)
      return true;

   if (!HasHessianGenerationFailed()) {
      if (!TFormula::fIsCladRuntimeIncluded) {
         TFormula::fIsCladRuntimeIncluded = true;
         gInterpreter->Declare(
             "#include <Math/CladDerivator.h>\n"
             "#pragma clad OFF");
      }

      if (!functionExists(GetHessianFuncName())) {
         std::string HessReqFuncName = GetHessianFuncName()
             + "_req";
         fHessGenerationInput =
             std::string("#pragma cling optimize(2)\n") +
             "#pragma clad ON\n" +
             "void " + HessReqFuncName + "() {\n" +
             "clad::hessian(" +
             std::string(fClingName.Data()) + ");\n }\n" +
             "#pragma clad OFF";

         if (!gInterpreter->Declare(
             fHessGenerationInput.c_str())
             )
            return false;
      }

      Bool_t hasParameters = (fNpar > 0);
      Bool_t hasVariables = (fNdim > 0);
      std::string HessFuncName = GetHessianFuncName();
      fHessMethod = prepareMethod(hasParameters, hasVariables,
                                  HessFuncName.c_str(),
                                  fVectorized,
                                  /*IsHessian*/ true);
      fHessFuncPtr = prepareFuncPtr(fHessMethod.get());
      return true;
   }
   return false;
}
```

g. `HessianPar()` - if required generates the hessian function and calls it

```
void TFormula::HessianPar(const Double_t *x, TFormula::
HessianStorage& result)
{
   if (DoEval(x) == TMath::QuietNaN())
      return;

   if (!fClingInitialized) {
      Error("HessianPar", "Could not initialize the formula!");
      return;
   }

   if (!GenerateHessianPar()) {
      Error("HessianPar",
          "Could not generate a gradient for the formula %s!",
          fClingName.Data());
      return;
   }

   if ((int)result.size() < fNpar * fNpar) {
      Warning("HessianPar",
              "The size of gradient result is "
              "%zu but %d is required. Resizing.",
              result.size(), fNpar * fNpar);
      result.resize(fNpar * fNpar);
   }
   HessianPar(x, result.data());
}

void TFormula::HessianPar(const Double_t *x, Double_t *result)
{
   void* args[3];
   const double * vars = (x) ? x : fClingVariables.data();
   args[0] = &vars;
   if (fNpar <= 0) {
      args[1] = &result;
      (*fHessFuncPtr)(0, 2, args, /*ret*/nullptr);
   } else {
      const double *pars = fClingParameters.data();
      args[1] = &pars;
      args[2] = &result;
      (*fHessFuncPtr)(0, 3, args, /*ret*/nullptr);
   }
}
```

h. `GetHessianFormula()` - returns the hessian function definition

```
TString TFormula::GetHessianFormula() const {
   std::unique_ptr<TInterpreterValue> v =
       gInterpreter->MakeInterpreterValue();
   gInterpreter->Evaluate(GetHessianFuncName().c_str(), *v);
   return v->ToString();
}
```

2. Rename `GradientStorage` to `ResultStorage` as it can be used for both gradient and hessian results and then create aliases named `GradientStorage` and `HessianStorage` to it.

On the Clad side:

3. The `HessianModeVisitor::Derive()` function currently doesn't support pointer or array types due to its dependency on forward mode. So to add support for `TFormula` in hessian we first need to detect when `TFormula` is being differentiated.

   Following the example of reverse mode when an array type or pointer type is passed as parameters we need to differentiate it only if the parameter is named "p".

   `HessianModeVisitor` first calls forward mode on the function for every argument and then reverse mode on the derived function, so we also need to derive the function w.r.t. every element in `"p"`. Since we can't find the number of elements in `"p"` without going through the function, it needs to be traversed to find the number of parameters and then forward mode needs to be called on it to differentiate it w.r.t. the array element.

   The code will look similar to the one below in `HessianModeVisitor::Derive()` at L490:

```
if (independentArg->getType()->isArrayType() ||
    independentArg->getType()->isPointerType()) {
  if (independentArg->getName() == "p") {
    struct IndicesExtractor :
        RecursiveASTVisitor<IndicesExtractor> {
      clang::ASTContext& m_Context;
      std::set<int, std::greater<int> > m_Indices;

      IndicesExtractor(DerivativeBuilder &builder)
          : m_Context(builder.m_Context) {}

      bool VisitArraySubscriptExpr(ArraySubscriptExpr* ASE) {
        llvm::SmallVector<const clang::Expr*, 4> Indices{};
        const Expr* E = ASE->IgnoreParenImpCasts();
        while (auto S = dyn_cast<ArraySubscriptExpr>(E)) {
          Indices.push_back(S->getIdx());
          E = S->getBase()->IgnoreParenImpCasts();
        }
        std::reverse(std::begin(Indices), std::end(Indices));
        for (auto &Index : Indices) {
          llvm::APSInt IntValue;
          clad_compat::Expr_EvaluateAsInt(Index, IntValue,
                                          m_Context);
          m_Indices.insert(IntValue.getExtValue());
        }
        return true;
      }
    } indicesExtractor(m_Builder);
    indicesExtractor.TraverseStmt(FD->getBody());

    // Call forward mode and reverse mode on the function w.r.t
    // each of the elements in the array just the way hessian
    // differentiates it
    ...

  }
  continue;
}
```

4. The forward mode also needs to be modified to support array differentiation w.r.t. to individual variables of an array.

    a. The `VisitorBase::parseDiffArgs()` needs to be modified to check for the special arguments that are being sent from.

    In the function when checking if the `diffArgs` match the parameters of the function it will fail to do so because a `diffArg` points to an element of the array. So we put a check for the parameter at L178 and do not throw the error message if it matches the format of an array element.

```
if (it == std::end(param_names_map)) {
  it = std::find_if(std::begin(param_names_map),
                    std::end(param_names_map),
                    [&name] (const std::pair<llvm::StringRef,
                    VarDecl*> & p) {
                      return p.first == "p"; });
  if (it == std::end(param_names_map) ||
      !name.startswith("p")) {
    diag(DiagnosticsEngine::Error, diffArgs->getEndLoc(),
         "Requested parameter name '%0' was not found among "
         "function parameters",
         {name});
  }
  return {};
}
```

    b. We then need to handle the same in `ForwardModeVisitor::Derive()` where the args vector is checked if it is empty (L694) and if it is we need to check the args mentioned in the differentiation request. If it follows the same format we store it in `m_IndependentArrayElement` and also set an additional suffix for the derived function name.

```
if (args.empty()) {
  if (auto SL = dyn_cast<StringLiteral>(request.Args)) {
    llvm::StringRef requestedArgs = SL->getString().trim();
    if (requestedArgs.startswith("p") &&
        requestedArgs.find(",") == llvm::StringRef::npos) {
      m_IndependentArrayElement = requestedArgs;
      derivativeSuffix = "_" + m_IndependentArrayElement;
    } else {
      return {};
    }
  } else {
    return {};
  }
}
```

Another check needs to be done before retrieving the `m_IndependentVar` at L703:

```
if(m_IndependentArrayElement == "") {
  m_IndependentVar = args.back();
  if (!m_IndependentVar->getType()->isRealType()) {
    diag(DiagnosticsEngine::Error,
    m_IndependentVar->getEndLoc(),
         "attempted differentiation w.r.t. a parameter ('%0') "
         "which is not of a real type",
         {m_IndependentVar->getNameAsString()});
    return {};
  }
  m_ArgIndex = std::distance(
      FD->param_begin(),
      std::find(FD->param_begin(), FD->param_end(),
          m_IndependentVar));
} else {
  m_IndependentVar = FD->getParamDecl(1);
  m_ArgIndex = 1;
}
```

And finally the derived function name needs a bit of tweaking at L729:

```
IdentifierInfo* II = &m_Context.Idents.get(
    derivativeBaseName + "_d" + s + "arg"
    + std::to_string(m_ArgIndex) + derivativeSuffix);
```

c. The `ForwardModeVisitor::VisitArraySubscriptExpr()` needs to accommodate the derivative of `m_IndependentArrayElement` at L1290:

```
if(GetArraySubscriptExprAsString(ASE) ==
   m_IndependentArrayElement) {
  auto one = ConstantFolder::synthesizeLiteral(m_Context.IntTy,
                                               m_Context,
                                               1);
  return StmtDiff(cloned, one);
}
```

Convert `cuda::gradient` functions to CUDA kernels

On the ROOT side:

5. The `TFormula::GenerateGradientPar()` and `TFormula::GradientPar()` will be cloned into `TFormula::GenerateGPUGradientPar()` and `TFormula::GPUGradientPar()` and `clad::gradient` will be called with `gpu_mode` set to `true`. The `TFormula::GPUGradientPar()` will call the derived GPU gradient function.

On the Clad side:

6. The `clad::gradient` needs to be modified to add three more arguments `gpu_mode`, `gpu_f`, and `gpu_code` to accommodate the GPU gradient function.

```
template<typename ArgSpec = const char *, typename F>
CladFunction<ExtractDerivedFnTraits_t<F>>
__attribute__((annotate("G")))
gpu_gradient(F f, ArgSpec args = "", const char* code = "",
             F gpu_f = nullptr, ArgSpec gpu_args = "",
             const char* gpu_code = "") {
  assert(f && "Must pass in a non-0 argument");
  return CladFunction<ExtractDerivedFnTraits_t<F>>(
      reinterpret_cast<ExtractDerivedFnTraits_t<F>>(f),
      code,
      reinterpret_cast<ExtractDerivedFnTraits_t<F>>(gpu_f),
      gpu_code);
}
```

7. The CladFunction constructor needs an overload with four arguments `f`, `code`, `gpu_f`, `gpu_code`. These will be stored in the member variables and the derived GPU function will be called with `CladFunction::executeGPU()` function.

```
CladFunction(CladFunctionType f, const char* code,
             CladFunctionType gpu_f, const char* gpu_code) {
  assert(f && "Must pass a non-0 argument.");
  if (size_t length = strlen(code)) {
    m_Function = f;
    m_Code = (char*)malloc(length + 1);
    strcpy(m_Code, code);
    if (size_t gpu_length = strlen(gpu_code)) {
      m_GPUFunction = gpu_f;
      m_GPUCode = (char*)malloc(gpu_length + 1);
      strcpy(m_GPUCode, gpu_code);
    }
  } else {
    printf("clad failed to place the generated derivative "
           "in the object\n");
    printf("Make sure calls to clad are within a #pragma clad ON "
           "region\n");
    m_Function = nullptr;
    m_Code = nullptr;
  }
}
```

8. The `DiffCollector::VisitCallExpr()` will check the `gpu_mode` argument and set the `GPUMode` in the `request`.

9. The `ReverseModeVistor::Derive()` function needs to modified in the following ways when GPU mode is on:

   a. The name of the derived function needs `"_gpu"` appended

   b. Member functions aren't supported so it needs to be checked that GPU mode isn't on

   c. The `gradientFD` needs the `__device__` and `__host__` attributes.

   ```
   if(GPUMode) {
     CUDADeviceAttr *CDA = CUDADeviceAttr::CreateImplicit(
         m_Sema.getASTContext(), gradientFD->getSourceRange());
     CDA->setImplicit(false);
     CUDAHostAttr *CHA = CUDAHostAttr::CreateImplicit(
         m_Sema.getASTContext(), gradientFD->getSourceRange());
     CHA->setImplicit(false);
     if (m_GPUMode)
       gradientFD->setAttrs({CDA, CHA});
   }
   ```

10. The `ReverseModeVisitor::VisitCallExpr()` will also need `"_gpu"` appended to the derived call function name and if Clad is going to generate the derived function `GPUMode` needs to be set in the request.

11. The `DiffRequest::updateCall()` needs to generate the wrapper for the kernel and the GPU gradient function pointer and set it to the `gpu_f` function and the code tp `gpu_code` in the gradient function. The wrapper will include a kernel that calls the derived function and another function that calls the kernel after allocating CUDA memory for all the variables and returns the result.

N.B. *There is going to be a lot of code duplication and it will be reduced after the pipeline is implemented and works.*

Timeline

Phase 1: June 7 - July 12

1. **Week 1:** ***Complete tasks 1 and 2***

   Create hessian counterparts of the gradient functions, variables, and types present in `TFormula`.

2. **Week 2:** ***Complete tasks 3 and 4***

   Modify `HessianModeVisitor::Derive()` to support pointer types in order to differentiate parameters of `TFormula`. `VisitorBase::parseDiffArgs()`, `ForwardModeVisitor::Derive()` and `ForwardModeVisitor::VisitArraySubscriptExpr()`.

3. **Week 3:** ***Benchmark and optimize the code***

   Create benchmarks in ROOT Benchmarks for the new features. Analyze the code to find bottlenecks and replace them with more efficient code.

4. **Week 4:** ***Add tests and documentation***

   Improve the existing documentation and add to it for the added functionality. Also, add tests in roottest and also in Clad for the same.

5. **Week 5:** ***Buffer Period***

   Complete the remaining tasks and compile a progress report.

Phase 2: July 12 - August 16

1. **Week 6:** ***Complete tasks 5, 6, 7, and 8***

   Add the GPU gradient functions in TFormula and GPU mode support in `clad::gradient`, `CladFunction` and `DiffCollector::VisitCallExpr().`

2. **Week 7:** **Complete tasks 9, 10, and 11**

Add GPU mode support in `ReverseModeVisitor`. Also add code in `DiffRequest::updateCall()` to generate the wrapper code for GPU and update `gpu_f` and `gpu_code` in the `clad::gradient` call.

3. **Week 8:** ***Benchmark and optimize the code***

   Create benchmarks in ROOT Benchmarks for the new features. Analyze the code to find bottlenecks and replace them with more efficient code.

4. **Week 9:** ***Add tests and documentation***

   Improve and expand on the current documentation to accommodate the new features. Add tests to roottest and Clad as well.

5. **Week 10:** ***Buffer Period***

   Complete the remaining tasks and compile a progress report.