Google Summer of Code

Project Proposal, 2024

Enable reverse-mode automatic differentiation of (CUDA) GPU kernels using Clad

Organization:

Cern-HSF

Mentors: Parth Arora, Vassil Vassilev

Contact details:

Christina Koutsou

GitHub: @kchristin22

E-mail: christinakoutsou22@gmail.com

Phone number: +30 6985033663

1. Project

1.1. Abstract

Nowadays, the rise of AI has shed light into the power of GPUs. The notion of General Purpose GPU Programming is becoming more and more popular and it seems that the scientific community is increasingly favoring it over CPU Programming. Consequently, implementation of mathematics and operations needed for such projects are getting adjusted to GPU's architecture.
Automatic differentiation is a notable concept in this context, finding applications across diverse domains from ML to Finance to Physics. Clad is a clang plugin for automatic differentiation that performs source-to-source transformation and produces a function capable of computing the derivatives of a given function at compile time. This project aims to widen Clad's use range and audience by enabling the reverse-mode automatic differentiation of CUDA kernels.

1.2. Current State

Currently, `Clad` supports differentiation of *__device__* functions. These functions can be accessed by the host (CPU) only through invoking kernels. This example from `test/CUDA/GradientCuda.cu` showcases the typical interaction of CPU and GPU:

```
__device__ __host__ double gauss(double* x, double* p, double
sigma, int dim) {
   double t = 0;
   for (int i = 0; i< dim; i++)
      t += (x[i] - p[i]) * (x[i] - p[i]);
   t = -t / (2*sigma*sigma);
   return std::pow(2*M_PI, -dim/2.0) * std::pow(sigma, -0.5) *
                                                  std::exp(t);
}

__global__ void compute(double* d_x, double* d_p, int n, double*
d_result) {
  auto gauss_g = clad::gradient(gauss, "p");
  gauss_g.execute(d_x, d_p, 2.0, n, d_result);
}

int main(void) {
  (...) // memory allocations and initializations
```

```
  compute<<<1, 1>>>(d_x, d_p, N, d_result);
  cudaDeviceSynchronize();
  // copy back result to CPU
  cudaMemcpy(result.data(), d_result, N * sizeof(double),
                                    cudaMemcpyDeviceToHost);
  return 0;
}
```

It is evident, however, that instead of creating a helper kernel to derive `gauss`, we could declare it as a kernel instead and, since the derived function keeps the same attributes as the original one, we could launch the derived kernel directly. Thus, the expected result would instead look like this- omissions have been made here, see [Implementation](#) for the suggested additions:

```
__global__ void gauss(double* x, double* p, double sigma, int dim,
double result) {
   double t = 0;
   for (int i = 0; i< dim; i++)
      t += (x[i] - p[i]) * (x[i] - p[i]);
   t = -t / (2*sigma*sigma);
   std::pow(2*M_PI, -dim/2.0) * std::pow(sigma, -0.5) *
                                                std::exp(t);
}

int main(void) {
  (...) // memory allocations and initializations
  auto gauss_g = clad::gradient(gauss, "p");
  gauss_g.execute(d_x, d_p, 2.0, n, result, d_result);
  cudaDeviceSynchronize();
  // copy back result to CPU
  cudaMemcpy(result.data(), d_result, N * sizeof(double),
                                    cudaMemcpyDeviceToHost);
  return 0;
}
```

In other words, the total goal of the project is to support the differentiation of CUDA kernels that may include typical CUDA built-in objects (e.g. threadIdx, blockDim etc.), that are employed to prevent race conditions, using `Clad`.

1.3. Implementation

This project can be broken down into two main categories:

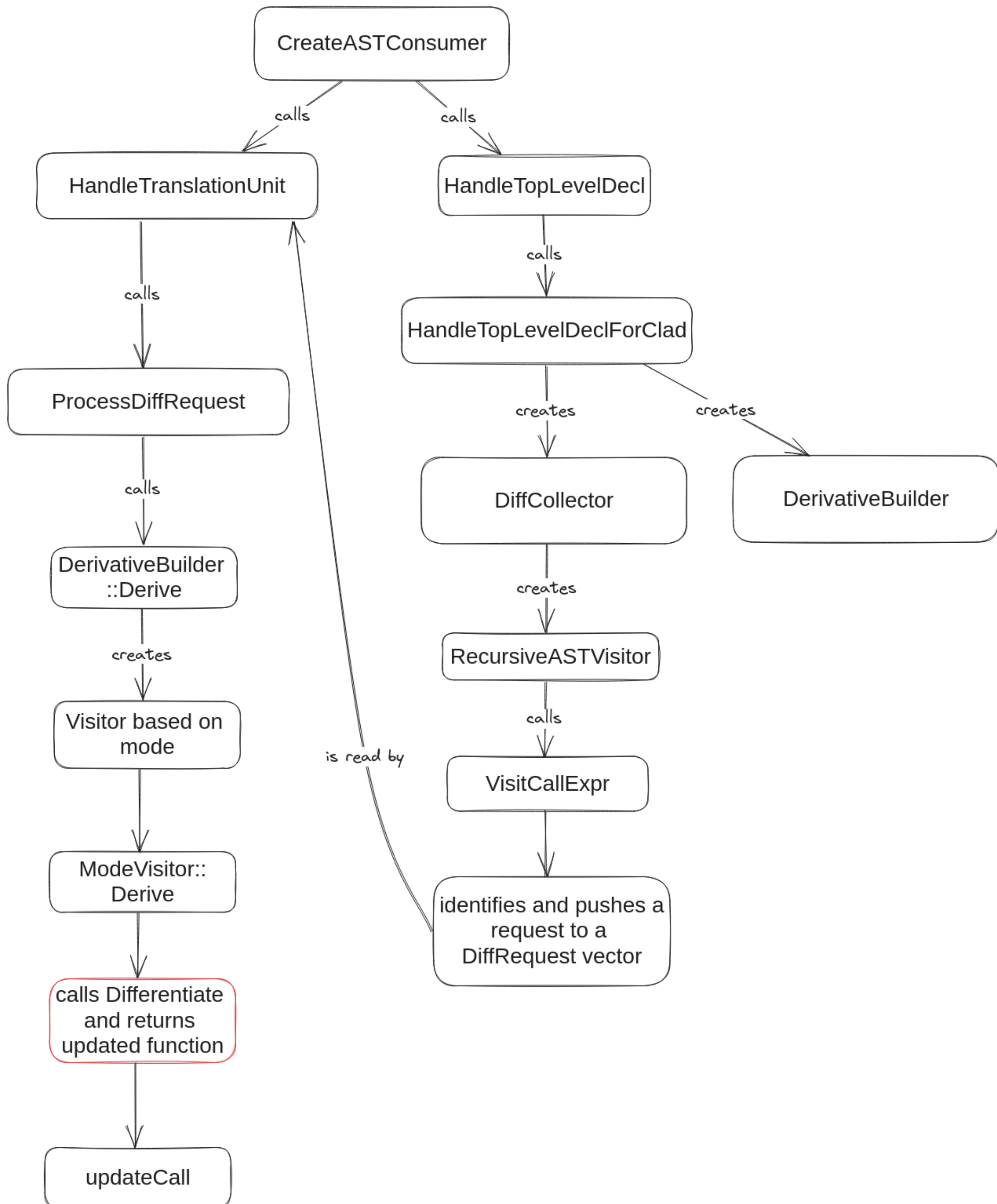
- General support: The ability to produce and execute a kernel's derivative function without it necessarily being correct
  - Produce the derived kernel during compilation
  - Execute the aforementioned function as a kernel
  - Produce the derived kernel when it's being called inside a function to derive
  - Execute the call to the derived kernel when it's being called inside a function to derive
- Support of kernel's nature: Derive the kernel with respect to its characteristics
  - Kernels are void functions and, as a result, we cannot derive based on a return statement
  - Kernels are executed by multiple threads and, thus, we need to account for write race conditions when computing the final result
  - Kernels usually use CUDA built-in objects and features

General support: Kernel's derivative compilation

What differentiates CUDA kernels from device or host functions is that they cannot be called without a provided grid configuration to be used. Hence, any call to them should be accompanied by such an expression. These invocations can occur in two different cases:

- During creation of derivative kernel
- Kernel execution

Since the original function is not called during the computation of its derivative, any calls to it are the user's responsibility, which falls outside the scope of `Clad`.
That leaves us with the derived function. In order to grasp where the differentiation of kernels fails, it is beneficial to study `Clad`'s workflow.



The arrows indicate the order of execution. It's apparent that in reverse mode, the derived function is generated within `ReverseModeVisitor::Derive()`, and at some point, this kernel is invoked upon its return, or, in other words, a `clang::CallExpr*` is created. Upon closer examination, it becomes evident that the error that is currently preventing the kernel derivation stems from `clang::Sema::ActOnCallExpr()`. Internally, this function triggers `BuildCallExpr()`, resulting in the creation of a CallExpr. To fix the error, the following changes are proposed:

- In `VisitorBase::BuildCallExprToFunction()`:
  - Detect if the function that was derived is a kernel function:

    ```
    If (FD->hasAttr<CUDAGlobalAttr>())
    ```

  - If true:
    - Create a configuration expression of (grid size, block size, shared memory size, stream ID) = (1,1,0,0) (in order to be called once) using `ActOnCUDAExecConfigExpr()`
    - Assign it to `configExpr`
  - Else:

    ```
    configExpr = nullptr;
    ```

  - Pass the configuration expression as an argument to `ActOnCallExpr()`:

```
call = m_Sema
           .ActOnCallExpr(
           getCurrentScope(),
           /*Fn=*/exprFunc,
           /*LParenLoc=*/noLoc,
           /*ArgExprs=*/
            llvm::MutableArrayRef<Expr*>(argExprs),
           /*RParenLoc=*/m_Function->getLocation(),
            /*ExecConfig=*/configExpr )
           .get();
```

The configuration settings applied here should not influence the actual execution of the kernel later on.

General support: Kernel's derivative execution

For the latter category of kernel invocation, after its creation, the API can be extended to include an overload function of `execute()` to which the user can pass the configuration parameters as arguments. These arguments will in turn pass through an overload of `execute_helper()` and subsequently `execute_with_default_args()`. Since CUDA supports the use of variables (both static and dynamic) in its configuration, the latter function will be modified like so:

```
return
f<<<grid_size,block_size,shared_mem_size,stream>>>(static_cast
                  <Args>(args)..., static_cast<Rest>(nullptr)...);
```

To prevent misuse of this capability, `assert()` statements will be added to ensure that these functions are being used only on kernels.

General support: Derivation of a kernel call

To further enhance the support for kernel differentiation, we should ensure that deriving a function that includes a kernel invocation can successfully call the derived kernel. Thus, when the `CUDAKernelCallExpr` is visited during the top level of derivation, we should store its configuration to use for the pullback kernel call. Specifically, a `Visit` function for kernels calls will be written, that mimics the one used for typical `CallExpr` nodes. However, it will additionally include storing the specified configuration using `getConfig()` on the kernel node and then passing it to the creation of the pullback kernel call expression through `clang::Sema::ActOnCallExpr()`.

Support of kernel's characteristics: Specify output argument to derive

As far as support of kernel's internal nature goes, it's important to consider that kernels are void functions and, consequently, their output is provided as an argument.

```
__global__ void kernelExample(int *in, int *out);
```

Hence, `Clad` needs to be able to compute and export the derivative of a parameter with respect to another one. For this purpose, the API must once again be updated to include an overload of `gradient()` that allows the user to specify the output variable of a void function. Similarly to the previously mentioned extension, an `assert()` statement will ensure that the function is used to derive void ones, not necessarily kernels this time.

```
gradient(F f, ArgSpec args = "", ArgSpec retArg = "",
          DerivedFnType derivedFn =
                        static_cast<DerivedFnType>(nullptr),
          const char* code = "") {
     assert(f && "Must pass in a non-0 argument");
     if(!std::is_same<void, return_type_t<F>>::value &&
        retArg != "")
          assert(0 && "Ret arg is only supported for void
                                                     functions");
     return CladFunction<DerivedFnType, ExtractFunctorTraits_t<F>,
        true>(derivedFn /* will be replaced by gradient*/, code);
  }
```

`DiffRequest` will now have an extra class member for the output argument and be assigned the second argument of `gradient()`:

```
 request.ReturnArg = E->getArg(2); // nullptr or specified arg
```

Currently, the argument to be used for the function differentiation along with the one that represents the function derivative value with regard to it (e.g. a and _d_a) are stored in a map named `m_Variables` in `ReverseModeVisitor::Derive()`. It's important to clarify that this map isn't utilized for creating the parameters of the derived function. Instead, its purpose is to identify which expressions are worth differentiating. As a result, we can freely modify it and append the output parameter and its derivative, upon creating it.
In more detail, if this argument is specified, then when the request is processed in `ReverseModeVisitor::Derive()`, the original function's parameters are scanned to find the one with the same name. If none is found, meaning the user gave an invalid parameter name, then an error is returned and the compilation fails. Else, the `m_Variables` map is updated.

```
if (request.ReturnArg) {
      auto E = request.ReturnArg->IgnoreParenImpCasts();
      // Store param name that represents the result of the void
     // function call
      if (auto SL = dyn_cast<StringLiteral>(E)) {
        llvm::StringRef retParamName = SL->getString().trim();
        auto fArgs = m_Function->parameters();
        auto it = std::find_if(std::begin(fArgs), std::end(fArgs),
                            [&retParamName](const ParmVarDecl* PVD)
                            {
                                  return PVD->getName() ==
                                        retParamName;
                            });
        if (it == std::end(fArgs))
            diag(
            clang::DiagnosticsEngine::Error, noLoc,
            "Return argument not found in the list of independent
             variables");
        else{
            // create output's derivative equivalent variable using
            // getParamType of iterator `it` and
            // CreateUniqueIdentifier,
            // append to m_Variables
            m_RetParamName = retParamName; // store output var name
        }
     }
}
```

Afterwards, within `DifferentiateWithClad()`, this variable is tracked and initialized to 1 (df/df = 1).

```
if (m_Variables.count(param)) {
      if (m_RetParamName == param->getName()) {
            auto size_type = m_Context.getSizeType();
            unsigned size_type_bits =
                                    m_Context.getIntWidth(size_type);
            auto* one = IntegerLiteral::Create(m_Context,
                                    llvm::APInt(size_type_bits, 1),
                                              m_Context.IntTy, noLoc);
            auto* left = param->getType()->isPointerType()
                              ? BuildOp(UO_Deref, m_Variables[param])
                              : m_Variables[param];
            addToBlock(BuildOp(BinaryOperatorKind::BO_Assign, left,
                                                    one),  m_Globals);
      }
      continue;
}
```

This way, when the AST Nodes of the original kernel are visited, expressions that include the output argument on the left-hand side will be correctly derived. The final result, though, will be stored in the derivative variable that shares the same name as the parameter with respect to whom the function was derived.

An example of the above is depicted below:

```
void foo(int *in, int *out){
  *out = 2 * *in;
}

// reverse pass:
// da = dout
// din = 2 * da ( or else din = 2 * dout)

void foo_grad(int *in, int *out, clad::array_ref<int> _d_in) {
     * _d_out = 1;
      int _t0;
      _t0 = *out;
      *out = 2 * *in;
      {
         *out = _t0;
```

```
        int _r_d0 = * _d_out;
        * _d_out -= _r_d0;
        * _d_in += 2 * _r_d0;
    }
}
```

Support of kernel's characteristics: Account for write race conditions in computation of the derivative value

To also account for the multithreaded environment of a kernel, the final result should be assigned its value and not ordered to add it to itself in `ReverseModeVisitor::VisitDeclRefExpr()`. As a result, if more than one thread executes an operation to the memory address of where the result will be stored, they will all execute the same assignment. To make it more clear, the above example would be modified like so:
`* _d_in += 2 * _r_d0; -> * _d_in = 2 * _r_d0;`
To further check the validity of this change for other cases as well, let's consider the example:

```
__global__ void compute(double *in, double *out, double val)
{
  int index = threadIdx.x;
  out[index] = in[index] + val;
}
```

In case, `_d_out[index]` is not initialized evenly among its indexes and we derive with regard to `val`, `_d_val` will suffer from a write condition. Hence, a better and more complete solution would also include a track of the array's subscript that corresponds to the output variable (return argument of the void function), if it's indeed denoted as an array, and build an array subscript expression for the `_d_val` with the same index to use in the assign operation.

Overall, the new version would look like:

```
void compute_grad(double *in, double *out, double val,
clad::array_ref<double> _d_in, clad::array_ref<double> _d_out,
clad::array_ref<double> _d_val) __attribute__((global)) {
      int _d_index = 0;
      double _t0;
      int index0 = threadIdx.x;
      _t0 = out[index0];
      out[index0] = in[index0] + val;
      {
```

```
        out[index0] = _t0;
        double _r_d0 = _d_out[index0];
        _d_out[index0] -= _r_d0;
        _d_in[index0] += _r_d0;
        * _d_val[index0] = _r_d0;
        }
}
```

On the same note, if the original kernel includes a for loop, then `Clad` uses its own implementation of a FILO queue, called a tape, to store useful values during derivation. Since these tapes are locally defined, the multithreaded environment will not affect their proper function. Hence, no change should be needed regarding them. However, tests will be conducted to ensure this is the case in every situation.

Support of kernel's characteristics: Handling of CUDA built-in objects

Another common characteristic of CUDA kernels is the use of built-in objects, e.g. `threadIdx`, as depicted above, to ensure that each thread computes its own share of the final result. These nodes should only be cloned when visited in the global initializations and be treated as integers when differentiated, or in other words their derivatives should be 0. In addition, support for the shared memory macro must also be included in the project's scope, by not discarding it when visiting a variable or array declaration.

1.4. Testing and documentation

For every new feature added in `Clad`, the `test` folder must be updated to ensure that each one has been tested thoroughly in a range of cases. `Clad`'s CI chain also includes code coverage tests that help meet the above requirement.
For documentation purposes, demo examples of how to use `Clad` with CUDA will be written with typical use cases of automatic differentiation as kernels, e.g. cost functions in ML, the Navier-Stokes equations etc.

2. Timeline

Below is a timetable for the project implementation, including deliverables and milestones:

Note: All the tasks are described in detail in the Implementation section and are only briefly referenced here

Duration	Tasks
Community Bonding Period	
May 2 - May 5	Meet mentors, learn about the team's meetings and way of work, fix local debugger, go through the documentation and submit improvements if necessary **Deliverable**: Write a blog post to introduce myself to the community, prepare a short presentation of the project and the plan of action for its implementation, fix reported bugs
May 6 - May 12	Discuss the project with mentors, add configuration expression in derivative kernel's creation **Deliverable**: Kernels are differentiated without a compilation error
May 13 - May 19	Add configuration in overloaded execute function of CladFunction object **Deliverable**: Kernels can be called for execution with a configurable grid
May 20 - May 26	Add simple test based on the example in GradientCuda.cu **Deliverable**: Verified general support of kernel's derivative compilation with Clad
Coding Period 1	
Week 1. May 27 - June 2	Add ability to specify the output argument of a void function in the API and store its name in the ReverseModeVisitor **Deliverable**: Return argument can be specified by the user only in void functions and be read by the lower level of Clad
Week 2. June 3 - June 9	Add this parameter to m_Variables map to derive any expression that includes it and initialize it to 1 **Deliverable**: Expressions that include the output argument are derived
Week 3. June 10 - June 16	Change add-assign operation of the final result to Clad's return argument that stores the value of the function's derivative (not only in void functions) **Deliverable**: Multithreaded functions don't face write race condition when the initialization of the function's derivative output is even

Duration	Tasks
Week 4. June 17 - June 23	Track the indexes of the output argument's array used and appoint the final value to the same index of Clad's return argument **Deliverable**: Write race conditions are taken into account for any function and the result is written to an appropriate position of the return argument
Week 5. June 24 - July 7	Write tests for the above, including ones where the function is a kernel, and identify any loopholes in the implementation **Deliverable**: Increasing coverage of function cases that may use a parallel computing API
Week 6. July 1 - July 7	Update tests already included in Clad and update README on user guidelines to access the derivative value **Deliverable**: Verified support for any function, void or not, that may be called by multiple threads
Midterm Evaluation	
Week 6. July 8 - July 12	Add support for CUDA built-in objects: threadIdx, blockIdx, blockDim and gridDim
Coding Period 2	
Week 6. July 13 - July 14	Add support for CUDA built-in objects: threadIdx, blockIdx, blockDim and gridDim when assigned to a variable **Deliverable**: Kernels that include CUDA built-ins regarding its grid, are correctly derived
Week 7. July 15 - July 21	Write tests for the above **Deliverable**: Verified support of CUDA built-in objects that refer to the grid configuration when they're assigned to a variable
Week 8. July 22 - July 28	Ensure that CUDA built-ins that are used directly as an array subscript are correctly cloned and tracked **Deliverable**: All cases of CUDA built-ins' usage are thought of and implemented
Week 9. July 29 - August 4	Write tests for the above **Deliverable**: Verified support of CUDA built-in objects that refer to the grid configuration however they're used

Duration	Tasks
Week 10. August 5 - August 11	Add support of __shared__ macro **Deliverable**: The derived kernels of original kernels that employ the shared memory of a block also use the shared memory in the same manner
Week 11. August 12 - August 18	Write tests for the above **Deliverable**: Verified support of differentiation of CUDA kernels that use the shared memory
Week 12. August 19 - August 25	Add support for kernel call derivation when they're inside a host function to be derived **Deliverable**: Support of kernel pullback function creation (kernels can be derived directly and indirectly)
Week 13. August 26 - September 1	Write tests for the above **Deliverable**: Verified support for kernel call derivation and generation
Week 14. September 2 - September 8	Discuss with mentors and Research use cases to implement in CUDA and Clad, that point to additional built-ins of CUDA worth supporting **Deliverable**: Prepare a short technical report to present findings and suggestions
Week 15 - 22. September 2 - October 31	Period dedicated to writing demos and additional support. Project Finalization. **Deliverable**: Verified correct derivation of multiple CUDA kernels using Clad

3. Personal details

3.1. About me

I'm a 5-th year student pursuing a degree in ECE with an integrated master's in Aristotle University of Thessaloniki, Greece. During my studies, I have been involved in multiple projects which helped me narrow down my interests as I progressed. One of the courses that impacted me the most was the one on Parallel Programming, where I implemented three projects of scientific applications using a different tool for each. In the first two projects I utilized APIs used for parallel and distributed programming in CPUs (pthreads, OpenMP, OpenCilk and MPI), while the third one was in CUDA and centered around

simulating the Ising model on a cellular automaton (all the projects are available to skim through on my github page). In each project, I didn't settle for the bare minimum but created multiple versions and algorithms to determine the most efficient one. This is more prominent in the last project, where I experimented with different CUDA features and got very familiar with CUDA's API. In addition, these projects really sharpened my skills in C++ and debugging.
I have come to really appreciate the open-source philosophy by having been a member of an open-source team, SpaceDot, for two years. SpaceDot is a non-profit, volunteering and interdisciplinary student team that participates in "Fly Your Satellite! 3", a European Space Agency (ESA) Education Office's program, and that is working on building a satellite. Even though it's a team consisting of merely students, the strong work ethic and the values of this team is its greatest appeal. During my stay there, I was appointed coordinator of the environmental campaign of a PCB I co-designed and also wrote code for. This task came with a lot of responsibility and taught me how to communicate efficiently, collaborate, manage my time and adapt to different situations. It also highlighted the power of initiative and showed me how to become a better member of a group myself. Not to mention that the freedom that comes with this project as long as its complexity have helped me develop my creative thinking and problem solving skills.
Joining the `Clad` team presents an exciting opportunity for me to further develop my skills and gain valuable experience contributing to a professional open-source project. The prospect of being mentored by Dr. Vassil Vassilev and Parth Arora is truly a privilege, and I am eager to absorb their guidance and insights. I am fully committed to making the most of this opportunity and actively contributing to the success of the Clad project.

3.2. Interest in CERN-HSF

CERN has always been an organization that I admired, due to my long passion for physics. When I first learned about `Clad`, its premise immediately captivated my interest and as I delved deeper into understanding its inner workings, I found myself getting more and more hooked and reminded of the intrigue that lower-level programming offers. My very positive experience with CUDA was what first drew me to this project, along with my newfound love for scientific computing. This project perfectly combines all the above, making it the only project I devoted myself to, and I truly believe in its importance. Moreover, `Clad` has really won me over and I would love to continue contributing to it even after this program.

3.3. Experience with Clad

During my effort to familiarize myself with `Clad` and better grasp the premise of this project, I identified and fixed issues. Notably, I took the initiative to open the following issues:

- #812 Examine clang-15 failure on CUDA Gradient script
- #813 Expand the CI to include CUDA set up
- #822 Fix derivative initialization in void functions in reverse mode
- #832 Consider a redesign of the for loop's body in reverse pass

The PRs I completed or I'm currently working on are:

- #806 Fix CUDA gradient script - Merged
- #823 Fix derivative initialization of void functions in reverse mode
- #833 Fix Incorrect derivative when loops contains continue
- #834 Add inst folder to gitignore - Merged
- #835 Redesign of loop's body in reverse pass

Despite the ongoing debate surrounding the necessity of #823 and #835, I found great value in experimenting with these ideas to enhance Clad and carrying them through. These initiatives made me more familiar with `Clad`'s lowest levels and I can now navigate through `Clad`'s source code with ease. Hence, I'm confident that I can begin contributing meaningfully to `Clad` immediately.

Moreover, as I researched for void functions' support, I tested and verified a variation of some of the suggested additions proposed in the Implementation section, specifically the snippets 1, 2,3 and 4 (working branch).

3.4. Other commitments during summer

Due to the challenging nature of the project and my desire to ensure that it is carried out successfully, I believe extending the timeline to the fullest will provide enough time to counter any unexpected bugs found along the way. The timeline mentioned earlier takes that into consideration and also matches my university schedule, to guarantee that all the project requirements are met during this time period without compromising on quality or deadlines.