GSoC 2025 Proposal

Implement activity analysis for reverse-mode differentiation of (CUDA) GPU kernels

Mentors: Vassil Vassilev, David Lange

Contact details:
Name: Maksym Andriichuk
Email: maksym.andriichuk@icloud.com
Github: ovdiiuv
Location: Germany

Candidate's Prior Experience

IRIS-HEP Fellow 2024:

- Optimizing automatic differentiation using activity analysis, mentors: Vassil Vassilev, Petro Zarytskyi, David Lange

Motivation

GPUs and CUDA have completely changed modern computing, going far from just rendering graphics. Now they're a go-to thing in a variety of fields because of their ability to handle lots of tasks at once. GPUs are great at parallel processing meaning they can handle thousands of operations simultaneously. This makes them perfect for things like AI, machine learning and data analysis, where they can significantly speed up the calculations. Subsequently, modern science requires *efficient* tools that are compatible with the GPU architecture.

Project Overview

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. AD breaks down the function into elementary operations and applies chain rule to compute derivatives of intermediate variables. It is an alternative technique to symbolic differentiation or numerical differentiation (the method of finite differences).

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(AST)* using LLVM compiler features. It performs advanced program optimization by implementing more sophisticated analyses because it has access to a rich program representation – the Clang AST. Clad supports reverse-mode differentiation of the CUDA kernels, however it is not always optimal because the generated code might contain the data-race conditions, significantly slowing up the execution. *Thread Safety Analysis(TSA)* is a static analysis that detects possible data-race conditions that would enable reducing atomic operations in the Clad-produced code.

Expected Outcome

As mentioned above, we aim to remove some of the atomic operations, where we can determine that no data-race condition occurs. In the code below:

```
#include "clad/Differentiator/Differentiator.h"

__global__ void assign_kernel(int *out, int *in) {
 int index = threadIdx.x + blockIdx.x * blockDim.x;
 out[index] += in[index];
}
```

```
// Gradient of assign_kernel
void assign_grad(int *out, int *in, int *_d_out, int *_d_in) {
   unsigned int _t1 = blockIdx.x;
   unsigned int _t0 = blockDim.x;
   int _d_index = 0;
   int index0 = threadIdx.x + _t1 * _t0;
   int _t2 = out[index0];
   out[index0] += in[index0];
   {
       out[index0] = _t2;
       int _r_d0 = _d_out[index0];
       atomicAdd(&_d_in[index0], _r_d0); <===>_d_in[index0] += _r_d0
   }
}
```

we use atomicAdd, since multiple threads may access `_d_in[index0]` at same time. However `index` is injective, meaning there are no two threads that share it and hence -- no data race. TSA would be able to detect cases like that and replace atomic operation with corresponding non-atomic one.

Implementation details

Task 1: Implement the analyzer.

- Implement a structure to store the intermediate result of the analysis.
- Implement a *ThreadSafetyAnalyzer* inherited from clang::RecursiveASTVisitor.
- Implement nested CallExpr handling.

Task 2: Integrate analysis to Clad codebase.

- Utilize the result of the analysis in shouldUseCudaAtomicOps.
- Investigate ways to benefit from the available analyses(Activity Analysis, To-Be-Recorded Analysis, Dependency Analysis)

Task 3: Tests and Benchmarks

- Implement extensive testing to cover all Clad capabilities.
- Test Clad’s performance against other CUDA-supported tools with analysis on/off

Task 4: Provide documentation

- Update the existing user documentation.

Timeline

Community Bonding Period		
Engage with the new members in a community. Set up CUDA cloud computing service. Review the documentation and improve if needed.		
Coding period begins		
Week 1-2: 27.05.2025-09.06.2025	Summarize work on sparsity patterns in clad, write documentation, open issues if needed. Prepare a presentation for the weekly meeting	*Deliverable:* Present work on a weekly meeting, summarizing progress on sparsity patterns.
Week 3-4: 10.06.2025-23.06.2025	Get acquainted with Clad's CUDA-related codebase. Look into Clang's parallel computing infrastructure and read related papers on the topic.	*Deliverable:* Start a blog post to introduce the project, prepare a presentation of the project for the weekly meeting.
Week 5-6: 24.06.2025-07.07.2025	Implement basic ThreadSafetyAnalyzer infrastructure. Add tests.	*Deliverable:* Update a blogpost on the progress.
Week 7: 08.07.2025-14.07.2025	Buffer week.	*Deliverable:* Prepare a presentation for the midterm evaluation.
Midterm Evaluations		
Week 8: 20.07.2025-26.07.2025	Enable CUDA+ThreadSafetyAnalyzer benchmarking.	*Deliverable:* Prepare plots for future presentations.
Week 9-10: 27.07.2025-10.08.2025	Adjust CUDA infrastructure in Clad to use Activity and TBR analyses. Open potential issues.	
Week 11: 11.08.2025-17.08.2025	Debug Add more tests, recompile benchmarks with available analyses.	
Week 12: 18.08.2025-24.08.2025	Buffer week.	*Deliverable:* Update blog post summarizing all progress.

Week 13-14: 25.08.2025-09.09.2025	Gather all data for a final presentation showing differences in performance. Extend the documentation.	*Deliverable:* Final presentation on the weekly meeting.

Candidate's other commitments

The only commitment other than GSoC is my studies at the Julius Maximilians Universität Würzburg. The courses selected for the summer semester are not challenging and do not oblige me to attend every lecture there is. Additionally the courses have non-differentiated grading, so I would not be overloaded at the end of the term either.