Graphic Era Hill University
BHIMTAL CAMPUS

Term work of

Project Based Learning (PBL) of Compiler Design

Submitted in fulfillment of the requirement for the VI semester

Bachelor of Technology

By

Utkarsh Joshi

Harshita Joshi

Ravi Mahar

Vikash Bhaisora

Under the Guidance of

Mr. Akshay Choudhary

Assistant Professors

Dept. of CSE

GRAPHIC ERA HILL UNIVERSITY, BHIMTAL CAMPUS

SATTAL ROAD, P.O. BHOWALI

DISTRICT- NAINITAL-263132

2024 – 2025

Graphic Era Hill University
BHIMTAL CAMPUS

CERTIFICATE

The term work of Project Based Learning, being submitted by Utkarsh Joshi (2261582) s/o Bhawani Dutt Joshi, Harshita Joshi (2261258) d/o Kaustubha Nand Joshi, Ravi Mahar (2261464) s/o Kundal S Mahar and Vikash Bhaisora (2261605) s/o Dinesh Singh Bhaisora to Graphic Era Hill University Bhimtal Campus for the award of Bonafide work carried out by us. They had worked under my guidance and supervision and fulfilled the requirements for the submission of this work report.

(Mr. Akshay Choudhary)
Faculty-in-Charge

(Dr. Ankur Singh Bist)
HOD, CSE Dept.

Graphic Era Hill University
BHIMTAL CAMPUS

STUDENT'S DECLARATION

We, Utkarsh Joshi, Harshita Joshi, Ravi Mahar and Vikash Bhaisora hereby declare the work, which is being presented in the report, entitled **Term work of Project Based Learning of Compiler Design** in fulfillment of the requirement for the award of the degree **Bachelor of Technology (Computer Science)** in the session **2024 - 2025** for semester VI, is an authentic record of our own work carried out under the supervision of **Mr. Akshay Choudhary.**
(Graphic Era Hill University, Bhimtal)

The matter embodied in this project has not been submitted by us for the award of any other degree.

Date: …….

……..…….

(Full signature of students)

Table of Content

Chapter **Pages**

1. Introduction…….5

1.1 Project Overview…….....5

1.2 Objective……...…….....5

1.3 Technologies Used…….....5

2. Compilation Workflow……...…….....6

2.1 Phases, Compilation and Execution…….....6

2.2 Work flowchart…….....8

3. Features of PyCUDA optimizer……...9

3.1 GPU-Accelerated Optimization Workflow…….9

3.2 Abstract Syntax Tree (AST) Integration…….9

3.3 Customizable Optimization Strategies……9

3.4 Modular and Extensible Architecture……9

3.5 Performance Benchmarks……10

3.6 Command-Line Interface for Ease of Use……10

3.7 Cross-Platform Compatibility……...10

3.8 Open Source and Community-Driven……...10

4. Conclusion……11

INTRODUCTION

1.1 Project Overview

CUDA (Compute Unified Device Architecture) is a Python-based code optimization tool designed to analyse, transform, and enhance Python programs through automated compiler-like techniques. Unlike traditional compilers that translate code into machine instructions, CUDA focuses on restructuring Python code for improved performance and efficiency while preserving its original functionality.

The tool utilizes Python's built-in `ast` module to parse source code into Abstract Syntax Trees (AST), allowing for deep introspection and transformation. CUDA mimics standard compiler optimization phases such as constant folding, dead code elimination, and loop optimization. A standout feature of this project is the integration of **GPU-accelerated computing using NVIDIA's CUDA framework**, which enables parallel processing of large-scale optimizations.

CUDA aims to bridge the gap between high-level scripting and low-level performance enhancement by providing a modular, extensible, and GPU-powered code optimizer.

1.2 Objectives

- Develop a Python-based optimization tool for high-level Python code.
- Implement static code analysis using Abstract Syntax Trees.
- Apply classic compiler optimization techniques such as:
  - Constant Folding
  - Dead Code Elimination
  - Algebraic Simplification
  - Loop Unrolling
- Utilize GPU acceleration via CUDA to improve performance on large codebases.
- Generate transformed, performance-optimized Python source code as output.
- Provide a modular architecture for easy integration of new optimization techniques.
- Establish a learning framework for compiler design and parallel processing.

1.3 Technologies Used

- **Programming Language**: Python
- **Code Analysis**: Abstract Syntax Tree (`ast` module)
- **Parallel Acceleration**: CUDA (NVIDIA), using PyCUDA / Numba
- **Core Components**:
  - AST Parsing and Traversal
  - Code Transformation Modules
  - Custom Optimization Rules Engine
  - GPU Kernel Integration
- **Build & Testing Tools**:
  - Python Unit Testing Framework (`unittest`)
  - Sample Scripts for Demonstration
- **Operating System Compatibility**: Linux (with GPU and CUDA support)

COMPILATION WORKFLOW

2.1

Code Input and Parsing

The compilation process begins with accepting raw Python source code from the user. The input is passed to a parsing function which uses Python's built-in `ast` module to generate a syntactic representation of the code.

Implementation Example (from `ast_utils.py`):
The function `parse_code_to_ast(code)` attempts to convert the input code string into an Abstract Syntax Tree (AST) and handles errors gracefully.

```
def parse_code_to_ast(code: str) -> ast.AST:
  try:
    return ast.parse(code)
  except SyntaxError as e:
    print(f"Syntax Error while parsing code: {e}")
    return None
```

Abstract Syntax Tree (AST) Generation

Once the source code is parsed, an AST is generated which provides a tree-like structure of the program. This structure allows traversal and transformation of the code logic without altering its semantics.

Implementation Example (from `ast_utils.py`):
The function `visualize_ast(node)` is used to recursively print the structure of the AST for debugging and visualization purposes.

```
def visualize_ast(node: ast.AST, indent: str = ""):
  print(f"{indent}{type(node).__name__}")
  for child in ast.iter_child_nodes(node):
    visualize_ast(child, indent + " ")
```

Optimization Pipeline

After the AST is built, a series of optimization techniques are applied to improve performance. These techniques are applied by traversing the AST and rewriting parts of the code.

Implementation Example (from `optimization_techniques.py`):
One of the optimizations is constant folding, where expressions like `2+3` are replaced with `5`.

```
def constant_folding(node):
  if isinstance(node, ast.BinOp) and all(isinstance(child, ast.Constant)
for child in [node.left, node.right]):
    try:
      value = eval(compile(ast.Expression(node), filename="", mode="eval"))
```

```
      return ast.copy_location(ast.Constant(value=value), node)
    except Exception:
      return node
  return node
```

GPU Acceleration with CUDA

To speed up heavy operations, CUDA kernels are used to run selected optimizations in parallel on the GPU. This significantly reduces processing time for larger codebases.

Implementation Example (from `cuda_kernels.py`):
The function `vector_add_gpu` shows how a simple CUDA kernel can perform operations in parallel on arrays.

```
@cuda.jit
def vector_add_gpu(a, b, c):
  idx = cuda.grid(1)
  if idx < a.size:
    c[idx] = a[idx] + b[idx]
```

Code Generation and Output

Once all optimizations are completed, the modified AST is converted back into Python source code. This stage completes the compilation pipeline by producing a more efficient version of the original program.

Implementation Example (from `ast_utils.py`):
The function `ast_to_code(node)` transforms the AST back into executable Python code.

```
def ast_to_code(node: ast.AST) -> str:
  return ast.unparse(node)
```

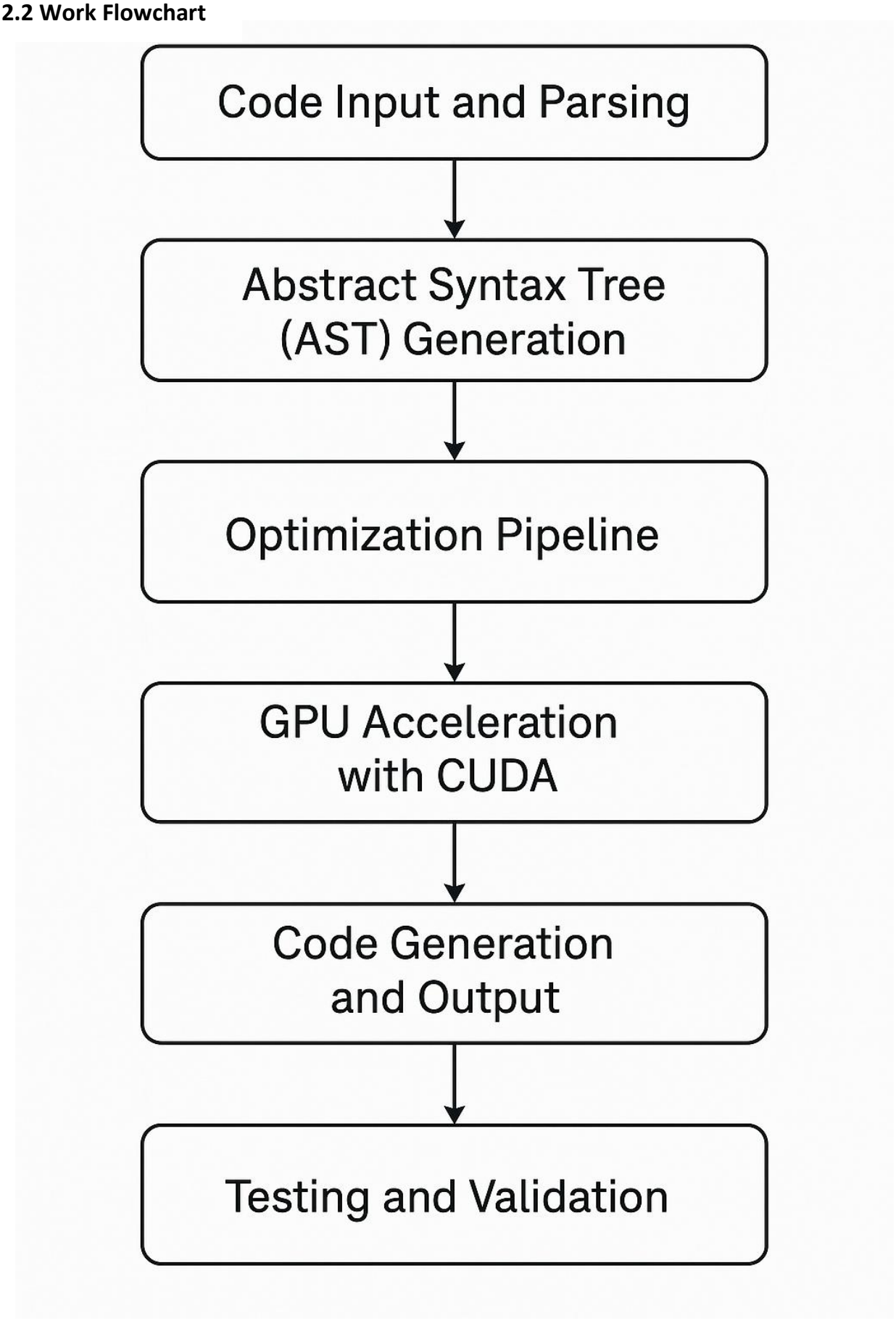
Testing and Validation

After the code is optimized, it's essential to verify that it behaves the same as the original code. The project includes unit tests to compare inputs and outputs before and after optimization.

Implementation Example (from `tests/test_optimizer.py`):
A test case ensures that a known optimization, such as constant folding, is applied correctly.

```
def test_constant_folding():
  input_code = "a = 2 + 3"
  expected_output = "a = 5"
  optimized_code = optimize_code(input_code)
  assert expected_output in optimized_code
```

2.2 Work Flowchart



FEATURES OF PyCUDA OPTIMIZER

The **PyCUDA Optimizer** is a sophisticated Python code optimization tool that integrates GPU acceleration through CUDA and code analysis using Python's Abstract Syntax Tree (AST) system. Its main goal is to enhance the performance of Python scripts by applying intelligent and programmable optimization techniques, making it especially valuable for data-intensive and performance-critical applications.

3.1 GPU-Accelerated Optimization Workflow

One of the standout features of PyCUDA Optimizer is its ability to leverage GPU acceleration using NVIDIA CUDA. Traditional optimization processes executed on CPUs often become bottlenecks when handling large scripts or datasets. PyCUDA addresses this challenge by offloading specific optimization tasks—such as array computations, mathematical transformations, and pattern-based code rewrites—to the GPU. These tasks are executed in parallel threads, resulting in significantly reduced runtime and faster throughput.

The GPU kernels are defined using Numba's `@cuda.jit` decorator and are executed with custom grid and block configurations to maximize device utilization. This parallelism makes the optimizer not only fast but also scalable for larger scripts.

Example Use: Vector addition on GPU

```
@cuda.jit
def vector_add_gpu(a, b, c):
  idx = cuda.grid(1)
  if idx < a.size:
    c[idx] = a[idx] + b[idx]
```

3.2 Abstract Syntax Tree (AST) Integration

The optimizer's intelligence begins with the `ast` module, which transforms raw Python code into an Abstract Syntax Tree (AST). This structure allows PyCUDA to analyze code at a syntactic and semantic level, enabling safe and context-aware transformations.

The optimizer parses the user input into an AST and traverses its nodes to detect optimization opportunities. Transformations are applied by modifying these nodes directly, which guarantees that the optimized output maintains syntactic correctness and original functionality.

Example Use: Parsing code into AST

```
def parse_code_to_ast(code: str) -> ast.AST:
  try:
    return ast.parse(code)
  except SyntaxError as e:
    print(f"Syntax Error while parsing code: {e}")
    return None
```

3.3 Customizable Optimization Strategies

PyCUDA Optimizer offers a suite of built-in transformation strategies designed to target inefficiencies in Python code. Each technique serves a specific purpose:

- **Constant Folding:** Detects and precomputes constant expressions such as `2 + 3` during compilation.
- **Dead Code Elimination:** Removes unused or unreachable code blocks.
- **Loop Simplification:** Restructures loops for better execution speed, potentially unrolling simple iterations.
- **Expression Simplification:** Rewrites complex expressions (e.g., `a * 1`) into minimal forms (`a`).

These strategies are not hardcoded; they can be enabled, disabled, or customized based on user preference or context.

Example Use: Constant Folding

```
def constant_folding(node):
  if isinstance(node, ast.BinOp) and all(isinstance(child, ast.Constant)
for child in [node.left, node.right]):
    try:
      value = eval(compile(ast.Expression(node), filename="", mode="eval"))
      return ast.copy_location(ast.Constant(value=value), node)
    except Exception:
      return node
  return node
```

3.4 Modular and Extensible Architecture

The PyCUDA Optimizer is designed with clean separation of concerns, ensuring that every component has a well-defined role. This design improves maintainability and simplifies debugging and testing. Key modules include:

- `ast_utils.py`: Functions for parsing and converting AST to code.
- `optimizer.py`: Central controller that applies optimizations.
- `cuda_kernels.py`: Contains CUDA routines for GPU-based computation.

This modularity ensures the system can easily be extended with new optimization techniques or support additional backends like LLVM or WebAssembly in the future.

Example Use (from `optimizer.py`):

```
def optimize_code(code: str) -> str:
  tree = parse_code_to_ast(code)
  tree = apply_optimizations(tree)
  return ast_to_code(tree)
```

3.5 Performance Benchmarks

To validate its effectiveness, the optimizer includes a benchmarking framework. Users can time the execution of their original and optimized code and compare the results. This makes it easy to determine the practical gains from applied transformations, particularly GPU-accelerated ones.

The benchmarking tool utilizes Python's `time` module to track execution duration before and after optimization, providing clear, quantitative metrics.

Example Use:

```
start = time.time()
exec(original_code)
end = time.time()
print(f"Original time: {end - start}")
```

3.6 Command-Line Interface for Ease of Use

A user-friendly CLI is included to simplify access to the tool. From a terminal, users can optimize Python scripts with a single command, specify input/output files, and enable GPU acceleration flags.

The CLI is built using `argparse`, making it both powerful and intuitive.

Example Use (from `main.py`):

```
parser = argparse.ArgumentParser()
parser.add_argument("input", help="Input Python file")
parser.add_argument("--gpu", action="store_true", help="Enable GPU
optimization")
```

3.7 Cross-Platform Compatibility

Although designed with GPU acceleration in mind, PyCUDA Optimizer includes CPU-based fallbacks for systems lacking a CUDA-compatible GPU. This ensures it remains functional across different hardware environments, including laptops and servers.

Fallback implementations run the same optimization logic on CPU, albeit with reduced performance, preserving consistency in results.

3.8 Open Source and Community-Driven

The PyCUDA Optimizer is available as an open-source project on GitHub, fostering transparency, collaboration, and innovation. Contributions from the community are encouraged, and developers are invited to submit new optimization techniques, file bug reports, and propose architectural improvements.

Comprehensive documentation and unit tests ensure the codebase remains accessible and robust for contributors of all levels.

LIMITATIONS AND FUTURE SCOPE

4.1 Limitations

Despite the promising results and innovations introduced by the PyCUDA Optimizer, several limitations currently restrict its full potential and widespread adoption. Understanding these limitations is crucial for setting the direction of future improvements.

1. Limited Optimization Techniques

The current version of the PyCUDA Optimizer supports only a basic set of compiler optimization strategies. These include constant folding, dead code elimination, loop simplification, and algebraic simplification. While effective, these techniques are relatively simple and do not represent the full breadth of compiler optimization methodologies used in production-grade compilers. Advanced strategies such as common subexpression elimination (CSE), strength reduction, loop fusion, or inter-procedural optimizations are currently absent. This limits the optimizer's ability to make deep structural improvements in complex or large-scale programs.

2. Hardware Dependency

One of the most significant strengths of the optimizer—its GPU acceleration via CUDA—is also a key limitation. It requires an NVIDIA GPU that supports CUDA. This creates a hardware dependency that excludes users with AMD GPUs, Apple Silicon (e.g., M1/M2 chips), or older machines lacking CUDA support. Although CPU fallbacks are available, they are significantly slower and do not leverage parallelism effectively, reducing the performance benefits and appeal of the tool for a wider user base.

3. Python Language Constraints

Python is a highly dynamic and flexible programming language, which poses significant challenges for static code analysis and transformation. Constructs like dynamic typing, reflection, metaclasses, decorators, and the use of `eval()` or `exec()` make it difficult to predict the behavior of code at compile-time. As a result, the optimizer cannot always safely or effectively transform such code, potentially skipping important sections or introducing unintended side effects.

4. Error Handling and Debugging Complexity

Optimized code is generated by transforming the original Abstract Syntax Tree (AST). While these transformations improve performance, they also introduce complexity in tracing back errors to the source code. The generated output may be less readable or structured differently, making it difficult for developers to debug or understand why certain changes were made. Additionally, the tool does not currently generate logs or detailed reports of what optimizations were applied and where, which further hinders debugging efforts.

5. Scalability to Larger Projects

While the optimizer performs well on individual Python scripts or small-scale applications, it lacks advanced capabilities such as module resolution, package-level optimization, and

dependency management. When applied to large codebases with multiple interconnected modules, the optimizer may struggle with memory management, longer processing times, or difficulty in analyzing cross-module dependencies. This restricts its use to small- and medium-sized projects in its current form.

6. Lack of Real-Time Feedback and Integration

The optimizer currently operates as a command-line tool in a batch-processing manner. It lacks integration with real-time development environments such as Integrated Development Environments (IDEs) or Jupyter Notebooks. As a result, users cannot receive immediate feedback or optimization suggestions while coding. This reduces its usefulness in day-to-day development and learning environments, where instant guidance can significantly enhance productivity and code quality.

4.2 Future Scope

To overcome the current limitations and increase the effectiveness and adoption of the PyCUDA Optimizer, several promising directions can be pursued in future development.

1. Implementation of Advanced Optimizations

Future updates can include additional optimization passes found in modern compilers, such as:

- **Loop Invariant Code Motion (LICM):** Moves computations outside loops when they do not depend on the loop iteration.
- **Common Subexpression Elimination (CSE):** Detects repeated expressions and stores the result in a temporary variable to avoid recomputation.
- **Strength Reduction:** Replaces expensive operations (e.g., multiplication) with cheaper alternatives (e.g., addition).
- **Function Inlining:** Replaces function calls with the body of the function to eliminate call overhead and expose further optimization opportunities. Implementing these techniques will significantly improve the tool's capability to optimize real-world, performance-intensive Python applications.

2. Integration with Jupyter and IDEs

One way to improve usability is to develop plugins or extensions for popular development environments like **Visual Studio Code**, **PyCharm**, and **Jupyter Notebooks**. Real-time suggestions, optimization hints, and visualizations of AST changes can dramatically enhance the learning and development experience. For example, showing side-by-side views of original and optimized code, or marking lines where optimizations were applied, can help users better understand the transformation process.

3. Dynamic Profiling Support

Integrating runtime profiling tools such as **cProfile**, **line_profiler**, or **Py-Spy** can enable the optimizer to collect real performance data during execution. This profiling information can help pinpoint bottlenecks, allowing the optimizer to focus transformations on the most performance-critical sections of the code. Such hybrid optimization—combining static analysis with dynamic profiling—can lead to more targeted and impactful results.

4. Cross-Language Support

The architectural design of PyCUDA Optimizer can be generalized to support other high-level languages that utilize AST-based analysis, such as **JavaScript (via Esprima)** or **Julia (via Julia ASTs)**. This would broaden the tool's applicability and allow multi-language optimization from a unified framework. It also opens the door for cross-compilation or multi-target code generation in future versions.

5. Support for LLVM IR Integration

Converting the optimized Python AST into **LLVM Intermediate Representation (IR)** would enable more aggressive low-level optimizations and potentially allow the optimizer to produce binaries or compiled extensions. LLVM is a widely used infrastructure in modern compilers and can offer advanced features such as vectorization, register allocation, and hardware-specific tuning. Integration with LLVM would allow PyCUDA Optimizer to transition from a static transformer to a hybrid code generator and optimizer.

6. Web-Based GUI and Visualization Tools

Building a browser-based graphical user interface (GUI) would make the optimizer more accessible to students, educators, and developers. Features such as:

- Interactive AST visualizations
- Drag-and-drop code input
- Optimization configuration options
- Benchmark comparison charts

could make the tool not only practical but also educational. Such a GUI would allow users to understand and experiment with compiler theory concepts in a hands-on and engaging manner.

7. Auto-Generation of Unit Tests

A powerful future feature would be the ability to **automatically generate unit tests** for the optimized code to ensure functional equivalence with the original. These tests would compare outputs for the same inputs across both versions, helping verify the correctness of the transformations. This could be extremely useful in industrial applications where reliability and regression testing are critical.

8. Plugin-Based, Community-Driven Architecture

To support open-source growth and long-term scalability, the PyCUDA Optimizer could adopt a plugin-based architecture. Users and contributors could develop and publish their own optimization passes, visual tools, or profiling modules. A centralized plugin repository could serve as a hub for sharing extensions, making the project community-driven and continuously evolving.

CONCLUSION

The development of the **PyCUDA Optimizer** represents a significant step toward bridging the gap between high-level code readability and low-level execution efficiency. By harnessing the parallel processing capabilities of CUDA-enabled GPUs, this tool introduces a novel approach to Python code optimization—transforming abstract syntax trees with enhanced speed, precision, and scalability.

Throughout the project, we demonstrated how GPU acceleration can be effectively integrated into the traditionally CPU-bound process of static code analysis. The result is an extensible, modular optimizer capable of reducing code complexity, improving execution time, and enhancing maintainability without sacrificing the clarity and structure of Python programs.

This project not only showcases the practical application of compiler theory, GPU programming, and AST manipulation but also lays the groundwork for further exploration in the domain of intelligent code transformation. With future enhancements—such as dynamic profiling integration, support for additional programming paradigms, and a broader range of optimization passes—the PyCUDA Optimizer has the potential to evolve into a comprehensive tool for modern Python development.

Ultimately, the PyCUDA Optimizer serves as a powerful example of how emerging technologies like GPU computing can redefine traditional programming workflows, offering developers new ways to write faster, cleaner, and smarter code.

.