Enable CUDA Compilation on Cppy-Numba generated IR

Google Summer Of Code 2024

Mentors: Aaron Jomy, Vassil Vassilev, Wim Lavrijsen, Jonas Rembser

Mentee: Riya Bisht
INTRODUCTION

- A third-year Computer Science & Engineering undergrad from Graphic Era University, India
- Interested in low-level systems, compilers, runtimes.
- Curious about mysteries of the universe, science and technoculture stuff
- Loves to research and explore new technologies
- Previously, contributed to different open source projects like Unikraft, KDE, Fedora

Contacts:

Website: https://riyabisht.com/

Github: https://github.com/chococandy63

Twitter: https://twitter.com/chococandy63

Discord username: ri_root
PROBLEM STATEMENT

Cppyy is a tool that automatically generates Python-C++ bindings at runtime, allowing Python to call C++ code and vice versa. It has recently added support for Numba, a high-performance Python compiler that compiles looped code containing C++ objects, methods, and functions defined via Cppyy (example in the coming slides) into efficient machine code.

GOAL

The goal is to demonstrate that Cppyy can define CUDA kernels in C++ and launch them from Python code compiled with Numba, enabling Python users to easily utilize the power of CUDA-accelerated computing. GPUs applications spans beyond graphics in the scientific community. Examples: image processing, genetic encoding (simulation of genetic codes).
GPGPU Ecosystem & Scientific Computing

- High compute intensive workloads/code on GPU
- Normal compute intensive workloads on CPU

PAST → PRESENT/FUTURE
Gaming, 3D Graphics → GPGPUs (Scientific computation and simulation)
"Mojo and Triton are based on MLIR dialects, allowing compiler frontend optimizations to be reused"

Sources:
- https://mlir.llvm.org/
- https://openai.com/index/triton/
- https://www.modular.com/max/mojo
IMPORTANCE OF THIS PROJECT

- Allows scientists to leverage powerful C++ libraries from Python, combining the performance of C++ with the simplicity and rich ecosystem of Python.
- Avoids cross language overhead.
- Interoperability between static and dynamic language (Differences between C++ and Python).
- Enabling CUDA compilation of the Numba-generated code would allow Python users to easily utilize GPU acceleration when working with C++ libraries, without sacrificing performance.
- Accelerate Research and Development in Scientific Computing like Data analysis (ROOT), Machine Learning, computational sciences like simulating genetic code, protein structures, etc that rely on both languages.
Understanding Cppyy

1. Bindings/Wrappers

[Interaction between C++/Cuda and Python]
[Proxies exposes C++ objects and classes to Python side]
[Reflections enables advanced features like runtime template instantiation, function callbacks, cross-language inheritance, etc]

2. GPU compilation pipeline

[ Enabled via Pre-Compiled CUDA headers]

[After enabling CUDA with `CLING_ENABLE_CUDA=1`, CUDA code can be used and kernels can be launched from JITed code by in `cppyy.cppdef()`]

Source: https://morepypy.blogspot.com/2012/06/architecture-of-cppyy.html
Cppyy-CUDA support

Cppyy (Python front-end) → Cppyy-backend (Cling Interactive C++ interpreter) → Pre-Compiled CUDA headers → CUDA

Pre-Compiled C++ headers → Pre-Compiled C++ headers → C++

Numba ??

(Task: Need to figure out how Numba fits the picture)
Why Numba?

- High performance python JIT compiler
- Numba’s IR- llvm, based on LLVM IR

Proof of concept
TASKS ACCOMPLISHED

- Project setup done (Faced issues #223 and #232 on my system - Fixed #234)
- Started a docs enhancements PR - #233
- Traced test_numba.py tests using PyCharm debugger (Learnt about proxies, reflections, etc)
- Tested support of Python 3.12 on Cppyy (Reported errors to the mentors)
- Tried running vector add CUDA kernel and reported my findings to the mentors

*ToDo*: Add a blogpost and presentation to compiler-research-website

*Coding period starts !!!!*

- Currently: Working on implementing `cppyy.cudadef` function (similar to cppdef)
**Proposed Plan**

**Midterm Evaluations**
*Deliverables:* To add some basic level of implementation. For example, use `std::vector` that contains an image. Have a piece of python that implements gaussian blur and run both on a gpu.

**Final Evaluation**
*Deliverables:* To replace the uniform data from the image that we showed from the previous deliverable with an irregular memory. For example, `std::vector` of some object.

**Future Scope**
Add support for Testing and Documentation(Also, Debugging support), Research on adding support for other graphics APIs.
THANK YOU

for listening!