Cross-language inlining on GPU through LLVM

Allen MacFarland, Jed Brown, Jeremy Thompson

August 14, 2025

Abstract

I introduce a method of inlining functions from an external programming language by individually manipulating the steps of LLVM compilation. Despite slightly slower compilations, it yields almost identical runtime performance to single-language alternative compilation schemes. Intended for developers who wish to integrate cross-language inlining into their own systems.

1 Introduction

I have been working on a C library called `libCEED`, which provides an efficient framework for matrix-free discretizations on CPU and GPU. This library requires users to write a small C function to define their mathematical operators; I changed this to allow users to write this function in Rust, and this paper will explain how this compilation scheme works in enough detail for anyone wishing to do the same between any other LLVM-based languages. While this frames why I needed to do this work, it is not the focus of this paper, which should be applicable to anyone interested in GPU compilation. For a reference implementation of the method described in this paper, please see the `libCEED` source code.

Inlining is a type of compile-time optimization where two functions can be combined into one (Figure 1). This reduces function calls, which is especially important when compiling to GPU targets, where function calls are incredibly expensive.

Typically, automatic inlining optimizations stay within language barriers, and it was generally accepted that cross-language calls should not be done in very performance-critical sections of code. However, thanks to intermediate representations (IR) of languages, it is now possible to inline across language barriers which have previously effectively blocked cross-language GPU calls.

```
// Two functions before inlining...
fn calculate_position(t: i32) -> i32 {
    return square(t) + t - 5;
}
fn square(x: i32) -> i32 {
    return x * x;
}

// ...turn into one function after inlining
fn calculate_position(t: i32) -> i32 {
    return (t * t) + t - 5;
}
```

Figure 1: A rust-style pseudocode example of inlining

IR was designed to be an intermediary step between languages and targets, such that not every language needed to write a compiler to every target; Languages could simply compile to LLVM, the largest IR, where they would get access to every target. However, this compiler design comes with a hidden benefit: it is possible to combine languages through LLVM IR and optimize with knowledge of the entire codebase, including multiple languages.

`clang` and `clang++` are LLVM-based C and C++ compilers that are closely tied with development of the LLVM IR. Rust is a modern programming language with an LLVM-based compiler `rustc` and package manager `cargo`, where packages are organized as “crates”.

2 The Pipeline

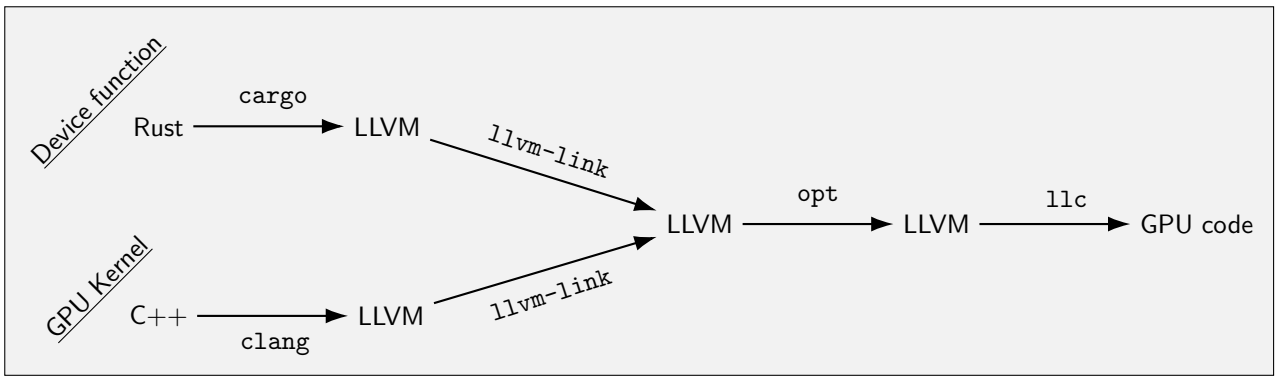

Figure 2: A diagram of the compilation scheme.

Our approach to compilation is to split the compilation into each of the traditional LLVM compilation steps and add the Rust LLVM as though it were part of a regular single-language LTO compile; see Figure 2.

In other words, the Rust and C++ are individually compiled first to LLVM with their respective compilers, then both of these LLVM files are linked with `llvm-link`, optimized (including inlining) with `opt`, and finally compiled to gpu code with `llc`. This produces a `.ptx` file which can be fed directly into CUDA.

3 Potential Pitfalls and Limitations

When working on manual LLVM compilation systems like these, there are a number of problems that special care must be taken to avoid. Three of them are described here

3.1 Generating Valid LLVM Output

Many programming languages that allow LLVM output only intended their LLVM outputs to be used for debugging, and never considered that it could instead be routed into a compilation pipeline. This may lead to unforeseen problems, depending on the programming language.

For example, in Rust, the well-documented `--emit=llvm-ir` API is not capable of emitting dependencies, including `core` or `std`, at least one of which is required for compilation of almost anything. Instead, developers must use the `linker-plugin-lto` rustflag and `build-std` nightly feature to generate a `staticlib`. Among other things, this contains the LLVM bitcode, so it must be passed to `llvm-link` with the `ignore-non-bitcode` flag.

3.2 LLVM Version Mismatches

New versions of LLVM are frequently released, and they are not backwards-compatible. This means that the entire pipeline must run the same version of LLVM, including compilers for other languages. For example, Rust nightly frequently updates their LLVM toolchain and maintains their own slightly modified branch, so the entire `libCEED` pipeline depends on the Rust-provided LLVM tools.

Those who wish to implement this system with another programming language should take care to ensure that the LLVM versions of all relevant tools match. Version mismatches **do not trigger version mismatch errors**, and may appear to be an entirely unrelated error. Additionally, it is possible to get "lucky" with LLVM versions and have LLVM generated in one version work in another. This is never recommended because small configuration changes could break code in hard-to-trace ways.

While it would be convenient for LLVM to be a stable platform to target for cross-compilation, this is not one of the goals of their project and is unlikely to change soon. Developers using LLVM in this way should be aware that this is not the intended use of the LLVM tools.

3.3 Distro Support for new LLVM Versions

Many distros ship only an outdated version of LLVM, which can cause frustrations for users with incompatible distros.

For example, Rust is typically installed with a standalone script that gets the latest version, regardless of distro, and our build method relies on the nightly release channel of Rust; at the time of writing, Ubuntu LTS only ships LLVM version 19, even though Rust uses LLVM version 20. Because we require the nightly release channel for the `build-std` feature (and this is never expected to land in stable), and it's not reasonable to use an older version of Rust nightly, our solution is effectively limited to bleeding-edge distros.

4 Performance

In `libCEED`, GPU compilation is done Just-in-Time (JiT), so the cost of compilation is included in the cost of runtime execution. It can be compared to a reference implementation with the proprietary `nvrtc` compiler and to a single-language variant of the `clang` compile process

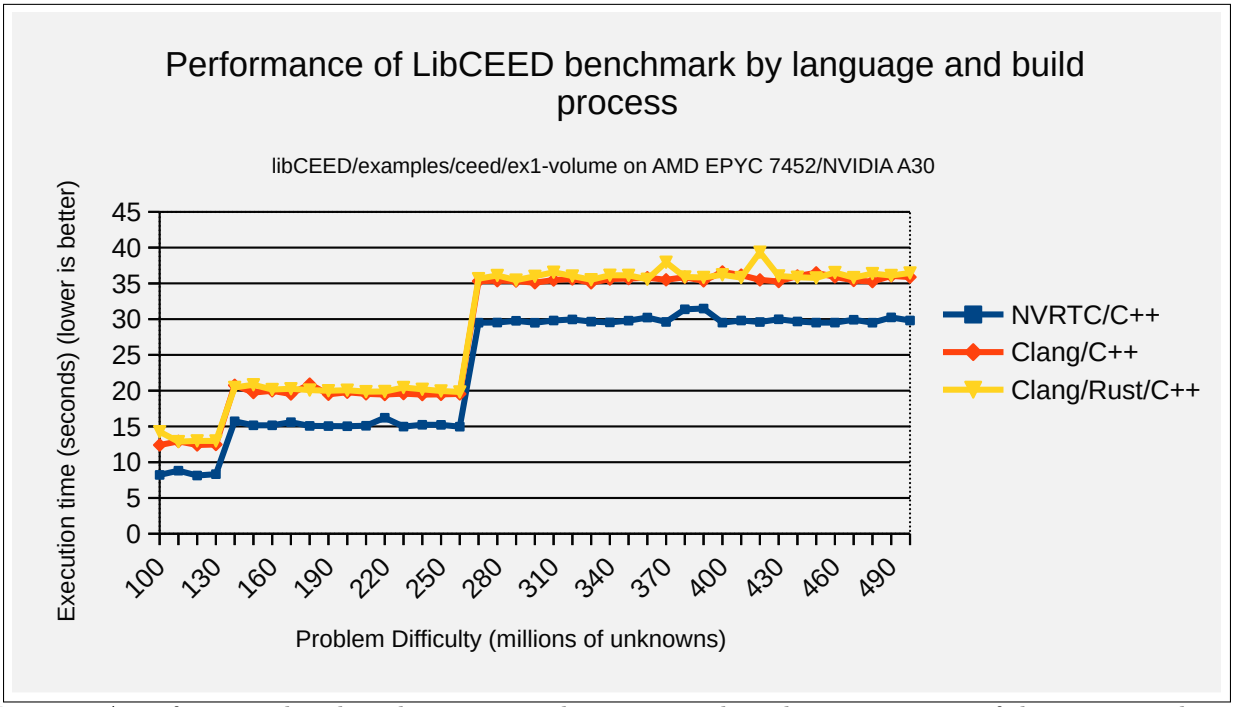

Figure 3: A performance benchmark comparing the new compile and execution time of the new compilation scheme relative to 2 possible controls: a single-language compilation scheme with the same compiler, and the proprietary `nvrtc`

As shown in Figure 3, `clang` takes longer to compile, but this is only an $O(1)$ cost, so as the problem size increases, the relative gap between all implementations decreases.

5 Conclusion

Combining languages with LLVM is a promising new compilation technique, especially on GPU targets, where inlining is essential.

The process described here for inlining Rust device functions into C++ kernels should be roughly applicable to inlining between any two LLVM-based languages. Further work could be done on implementing such integrations.

Further work could also be done on improving the pain points described in section 3 on the LLVM side. Improving LLVM error messages or committing to a more stable IR could significantly simplify development of many integrations.

6 Acknowledgments

This work was funded by the United States Department of Energy

- PSAAP - Predictive Science Academic Alliance Program
- SciDAC - Scientific Discovery through Advanced Computing
- Exascale computing project

This work was completed by an undergraduate researcher funded by the SPUR program of the University of Colorado Boulder, funded by the Engineering Excellence Fund

References

[03] *LLVM*. https://llvm.org/. 2003. URL: https://llvm.org/.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: San Jose, CA, USA, Mar. 2004, pp. 75–88.

[07] *Clang*. https://clang.llvm.org/. 2007. URL: https://clang.llvm.org/.

[15] *The Rust Programming Language*. https://www.rust-lang.org/. 2015. URL: https://www.rust-lang.org/.

[Bro+21] Jed Brown et al. "libCEED: Fast algebra for high-order element-based discretizations". In: *Journal of Open Source Software* 6.63 (2021), p. 2945. DOI: 10.21105/joss.02945.

[21] *libCEED development site*. https://github.com/ceed/libceed. 2021. URL: https://github.com/ceed/libceed.