Towards General Compilation for Heterogeneous Backends: The Unified Compiler

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Neil Phan

Spring, 2024

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Briana Morrison, Department of Computer Science

Towards General Compilation for Heterogeneous Backends: The Unified Compiler

CS4991 Capstone Report, 2024

Neil Phan
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
nnp3axx@virginia.edu

ABSTRACT

Working with new hardware devices is difficult, leaving only a small subset of programmers able to program on devices like GPUs and FPGAs. I propose a unified compiler capable of compiling to multiple backends can improve accessibility and increase the benefit of these new devices. Through the use of MLIR, a library that can create dialects for use by a compiler, multiple dialects can be designed to take advantage of each hardware's backend. Each new dialect can then be connected to a higher-level dialect that will serve as the domain-specific language to be standardized. Through this one domain-specific language proposal, many programmers would be able to flex the abilities of different hardware backends without needing extensive knowledge of them. Next steps in the proposal involve learning more about the different heterogeneous devices so that the initial dialects for each backend can be developed and built upon in the domain-specific language.

1. INTRODUCTION

With the decline of Moore's Law, hardware accelerators have become more popular in the past couple of decades. Most common hardware accelerators include the graphics processing unit (GPU) and the field-programmable gate array (FPGA) each of which have their own niche uses. There also exist many other backends that are currently in development, such as processing-in-memory (PIM) for DRAM and SRAM. Each new backend creates a need to know the general architecture behind it in order to take full advantage of the hardware accelerators' potential. This leaves many who are not proficient in hardware unable to fully utilize the potential of hardware accelerators.

2. RELATED WORKS

Compilers were developed to address the difficulties in using hardware. The first major ones to come out included CUDA, a programming API built on top of C++ and C that allows programmers to write C++ functions that run on the GPU (NVIDIA, n.d.). This was a major step in improving the readability of hardware accelerator code and providing more access to the public. However, compilers such as CUDA from NVIDIA have generally been developed by companies and were close-sourced. While the new programming interface provided extensive programmability for the hardware, it would still require many users to learn the tricks of the language in order to master it and take full advantage of the hardware.

This led into the era of MLIR, an infrastructure that supports the development of multiple dialects that can connect with each other (MLIR, n.d.). An open-source project that expanded from LLVM, this was an innovative change in how one could create

compilers for backends. Dialects could "connect" with each other and create a clearly-defined pipeline, like a general GPU dialect that could lower down into more specific languages like nvvm for NVIDIA and rocdl for AMD. This would spark others to build new domain-specific languages, such as HeteroCL, which takes advantage of different heterogeneous backends and applies different optimizations in a high-level domain-specific language (MLIR, n.d.). The project supports generating code for CPUs and FPGAs. HeteroCL takes advantage of the kernel concept, where one can specify a generic algorithm like matrix multiply and then apply a series of optimizations to it through imperative programming. The development of projects like MLIR and HeteroCL have continued to push further for the idea of general programmability in heterogeneous backends.

3. PROPOSED COMPILER DESIGN

For the unified compiler proposal, it will follow similar structures to other compilers today: the design of a front-end, middle-end, and back-end. By using MLIR, we can define the dialects necessary for each hardware device. For now, the proposal will focus on compiling code down to the CPU and the GPU.

Figure 1 illustrates the compiling pipeline proposed for the unified compiler. Users of the compiler will create code in the front-end, which will then be converted to middle-end code via MLIR. Once the code is processed in the middle-end, we can reduce it to its binary form for the respective hardware accelerator that is being targeted via the back-end. We define each component of the compiler in the future subsections below.

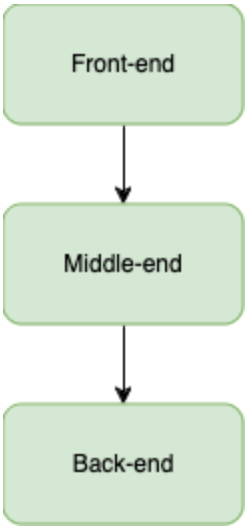

Figure 1. Pipeline Diagram for the Unified Compiler

3.1 FRONT-END DESIGN

The front-end focuses primarily on the user experience and ensuring that all the functionality the user desires exists in a clean and concise manner. Many modern languages today such as Rust and C++ have their front-end language defined so that users can create human-readable code, rather than having to write in lower-level languages like Assembly for computers to read.

For the unified compiler proposal, there will also exist a front-end language that users will program in and then compile down to via the unified compiler. To simplify the need of having to learn a new language, it will be a domain-specific language, since the goal of the proposal is to provide an interface that is easy for the users to program in and can be lowered down into different accelerators. The DSL will be based in Python, given that it is a simple-to-use language that many users already know and can pick up easily otherwise.

The DSL is available to use via a python library import. From there, users can specify a function that they want to write and lower it

down with DSL functions provided by the compiler. Figure 2 demonstrates how one could write a vector add and reduce it to a GPU program. The code maintains a Pythonic style while allowing users to easily define the code with minimal additions to the function.

```
def VA(A, B): -> vector
    N = len(A)
    C = vector(N)
    for i in range(N):
        C[i] = A[i] + B[i]
    return C

com = unified_compiler.init()
com.target("GPU")
com.compile(VA)
com.generate(VA)
```

Figure 2. Vector Add Program Example

The benefit of having the front-end defined like this is for user simplicity. Rather than having the burden on the users to maintain the program, it is all abstracted away via the middle-end and back-end code which will work on defining the operations necessary to reduce the Python programs down into GPU binaries and much more.

3.2 MIDDLE-END DESIGN

The middle-end tackles the issue of how to take the Python DSL program defined by the user and lower it into a more verbose language to eventually create a binary for the specific hardware accelerator targeted. While a middle-end isn't always necessary for a compiler, it allows the compiler programmer to lower down to more fine-grained optimizations that need more information that isn't provided in just the DSL code. Figure 4 shows how the conversion from the front-end to middle-end of a compiler can allow the compiler programmer to have more freedom in how to express the code and optimize it to their desires.

The unified compiler proposal will tackle starting the foundation for how to lower the Python DSL front-end into different hardware accelerator binaries. For this proposal, it will only focus on GPUs but highlight how it can expand to other accelerators such as FPGAs. Figure 6 shows the pipeline diagram for how a user in the front-end can specify which target accelerator they want to compile down to, and Figure 3 shows the entire flow diagram to reach the back-end. This can all be done by creating different compiler passes, which are programs that will take in input the python DSL program and generate the middle-end code depending on the target accelerator.

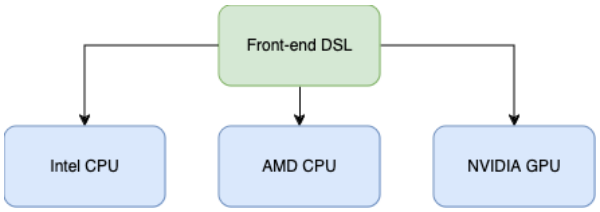

Figure 3. Middle-end Pipeline Diagram

The specific design of how to do it for a GPU requires extensive GPU knowledge and will not be explained in this paper, but the GPU compiler pass for the unified compiler proposal will be able to encapsulate the optimizations necessary to squeeze performance out of the DSL program, such as tiling the data in the GPU as best as possible.

3.3 BACK-END DESIGN

The back-end's goal is to take the middle-end language from MLIR and reduce it down to the respective binaries for the targeted hardware accelerators. Now that the user has the middle-end code defined and reduced down to the respective hardware accelerator, it can be compiled down to the binary necessary to run it on the device.

The unified compiler proposal will take a similar approach to how the middle-end

reduced the Python DSL code down to MLIR, only this time it will instead compile down to the respective assembly code that the device needs to run the program. For example, NVIDIA GPUs will need to be reduced to their PTX binary in order to run it on their devices, so a back-end compiler pass will be developed to compile the MLIR GPU code in the unified compiler down to the PTX binary. This step can be reiterated for different GPU vendors, different hardware accelerators, and so on.

The user is now left with the binary for the hardware accelerator they selected in the front-end. Since they are only left with the assembly file, they will need to take the extra steps to run it on the device.

4. ANTICIPATED RESULTS

The unified compiler proposal does not have explicit results since it is only a proposal, but will cover the methodologies of verifying the compiler and its performance.

The first part of the proposed results will involve verifying the integrity of the unified compiler. To ensure that the results are accurate, a benchmark suite composed of different kernels will be run on via general Python code and the respective Python DSL. The results will compare whether or not the results of the code are equivalent. Some kernels that can be used include, GEMM, K-Means, Gaussian Elimination, and more.

The second part of the proposed results will involve analyzing the performance of the unified compiler. Since the unified compiler will focus mainly on CPUs and GPUs, the unified compiler will be tested on an Intel CPU, AMD CPU, and NVIDIA GPU. The unified compiler will be compared to the original compilers of the hardware. For example, NVIDIA GPUs will be compiled via CUDA as a baseline. The main components that will be tested are run-time of the program and the memory-usage on the device.

5. CONCLUSION

The unified compiler proposal aims to reduce the barriers of users looking to take advantage of hardware accelerators by introducing a domain-specific language that has minimal learning curve. Through designing the different components of the compiler, it will be able target different hardware accelerator back-ends while maintaining one language, allowing users to focus on creating code and not learning specific intrinsics to the hardware.

6. FUTURE WORK

Next steps involve gathering information on the specifics of the hardware, such as learning more about the optimizations done for NVIDIA GPUs, CPUs, and more. A more in-depth design on how the compiler will function will also be necessary to avoid deprecation and duplication of features. Creating a community and gathering funding is also necessary to bring the domain-specific language to life.

REFERENCES

About CUDA. (n.d.). NVIDIA Developer. Retrieved February 20, 2024, from https://developer.nvidia.com/about-cuda

Lai, Y.-H., Chi, Y., Hu, Y., Wang, J., Yu, C. H., Zhou, Y., Cong, J., & Zhang, Z. (2019). HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 242–251. https://doi.org/10.1145/3289602.3293910

MLIR. (n.d.). Retrieved February 20, 2024, from https://mlir.llvm.org/