An MLIR-based Compiler for Interoperability between Machine Learning and Science Frameworks

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

The adoption of artificial intelligence and machine learning (AI/ML) in science domains is increasing at a rapid pace. This includes integration of AI/ML in several science simulations for biology, chemistry, material science and cosomology. However, modern ML and traditional scientific high-performance computing (HPC) tend to use completely different software ecosystems. In this work, we show that a compiler-based approach can bridge the gap between ML frameworks and scientific software with less developer effort and better efficiency. We use the Multi-level Intermediate Representation (MLIR) ecosystem to compile a pre-trained convolutional neural network (CNN) in PyTorch to freestanding C++ source code in the performance portable Kokkos programming model. Our compiler-generated Kokkos and C++ source code can be directly integrated into any Kokkos-based science application with no dependencies on Python or cross-language interfaces. In addition, this provides an easy path to support accelerators from AMD, Intel, and NVIDIA as Kokkos provides backends to them.

1 Introduction

Physics-informed AI/ML is becoming another tool in the tool box in the hard sciences like classical mechanics, magnetohydrodynamics, density functional theory, molecular dynamics, chemistry and electronic circuits. Several of these applications focus on numerical simulation of some physical phenomena. Such simulations are computationally demanding especially for high fidelity needs.

Machine learning (ML) techniques like deep neural networks (DNNs) can help with some of these challenges. For example, DNNs were used as a surrogate for the computational bottleneck of density functional theory (DFT), allowing the electronic structure of a molecule to be determined efficiently from just its atomic configuration [3]. In a multiscale simulation, the first-principle calculation is replaced by the ML surrogate within a larger scale simulations for phenomena at the mesoscale or macroscale.

Unfortunately, most scientific applications are written in C++ while the state-of-the-art ML frameworks like PyTorch and TensorFlow provide their first-class interfaces in Python. Interoperating between C++ and Python usually requires significant boilerplate and developer effort. In this work, we show that a compiler can completely bypass the gap between C++ and Python, while yielding additional benefits in portability, efficiency and safety. Instead of calling low level functions in an ML framework from a C++ application (e.g., the mesoscale or macroscale simulation), we allow the developer to write ML-related functions in native Python, and then automatically compile that Python to Kokkos-based C++ source code, which can be integrated directly into scientific applications as if it had been written by hand. Kokkos [7] is a shared-memory parallel programming model for C++ that is designed for high performance across a variety of CPU and GPU architectures. To implement the

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

compilation pipeline, we use the MLIR (Multi-Level Intermediate Representation) library within the LLVM project [4].

Python's popular ML frameworks like PyTorch [5] and TensorFlow [1] are two early users of MLIR. An existing extension for PyTorch called torch-mlir provides the functionality to generate an MLIR function from a model. We used a pre-trained image classification CNN model from PyTorch, ResNet18, as the target for testing our compiler. We were successful in creating a fully automated pipeline that generates working Kokkos C++ source code from the PyTorch model. This code can successfully perform ResNet18 inference on real images when integrated into a standalone C++ program. This transformation also allows transforming Python sources to multiple hardware targets such as CPUs and GPUs. Kokkos uses template meta-programming to support serial, OpenMP threads, CUDA, HIP and SYCL backends. We demonstrate transformations to serial, OpenMP and CUDA backends of Kokkos.

2 Methods

The goal of this work was to investigate whether a compiler-based approach could resolve the issues with Python-C++ interoperability especially for use cases such as in multiscale simulations. This allows us to focus primarily on inference. For example, a microscale simulation surrogate can be trained offline. However, the trained model has to be deployed within a mesoscale or microscale simulation framework.

To prove that such a compiler-based approach is actually feasible, we set out to construct a working compilation pipeline that can automatically expose some non-trivial machine learning functionality to a C++ application. A goal from the beginning was to leverage the MLIR library from the LLVM project [4].

The core of MLIR is an abstract language specification that is similar to the LLVM IR, but builds upon it with higher-level constructs organized into various dialects. For example, a matrix-matrix multiplication can be expressed with a single "instruction" from the linear algebra dialect, `linalg.matmul`. Compared to a lower-level scalar IR like LLVM, MLIR retains high-level information about code behavior, enabling powerful compiler optimizations (e.g. fusion of linear algebra operations) that would otherwise be very difficult to perform safely on scalar LLVM code. In addition to the dialect specifications, MLIR includes built-in compiler transformations to replace instructions to equivalent but lower-level code ("lowering"). For example, a `linalg.matmul` instruction might be replaced with a triply-nested for loop which computes the result matrix one element at a time with scalar arithmetic.

MLIR is just an intermediate representation with associated transformations and tooling. Creating an MLIR module from Python code (or code in any other programming language) is the responsibility of an external frontend. The two most popular machine learning frameworks, PyTorch and TensorFlow, currently provide such frontends. For the purposes of this project, PyTorch's frontend (torch-mlir) was used as the first step in the compiler pipeline. This was convenient because torch-mlir provides a full example where the pre-trained ResNet18 image classification model is converted from PyTorch to freestanding MLIR, progressively lowered through a pipeline of built-in MLIR transformations, and finally just-in-time (JIT) compiled to native serial code using LLVM. Here, "freestanding" means that the MLIR code has no external dependencies - it no longer involves Python and it includes all constant data needed by the model such as the CNN weight matrices. Our objective was now to replicate this example but with parallel Kokkos C++ code as the final target instead of LLVM. We would also like to target at least two different types of parallelism in the generated code (OpenMP and CUDA).

To get from MLIR to Kokkos, we used MLIR's existing C++ emitter as a starting point. This emitter accepts a low-level subset of dialects and produces serial C++ source code. One of the dialects is `EmitC`, whose instructions directly map to C/C++ constructs like variable declarations and `#include`. We found that a higher-level set of dialects was a closer fit to the Kokkos model, however. The fundamental constructs in Kokkos are parallel kernels belonging to one of three patterns (for, reduce and scan), and multi-dimensional arrays (`Kokkos::View`) [7]. These constructs and the `View` are templated on how they have to be run (serial, OpenMP, CUDA), and where memory is (host, device, shared memory). MLIR has dialects and instructions that are equivalent to these constructs. The `scf.parallel` instruction executes some body of code for each element in a multi-dimensional iter-

Table 1: Examples of common MLIR instructions and their equivalents in Kokkos.

MLIR	Kokkos
`memref.store %100 %A[%1]`	`A(i) = 100;`
`%0 = memref.alloc() : memref<200xf32>`	`View<float[200]> v("v", 200);`
`scf.parallel(%arg1) = %1 to %2...`	`parallel_for(RangePolicy<>...`
`%a = math.sqrt %b`	`float a = Kokkos::sqrt(b);`
`%a = arith.subf %b, %c`	`float a = b - c;`

Table 2: A list of the most significant built-in MLIR and torch-mlir lowering transformations we apply between the PyTorch frontend and the Kokkos emitter.

Transformation Name	Effect
`tm-tensor-bufferize`	Implement abstract PyTorch tensors as multidimensional arrays in memory.
`linalg-bufferize`	Implement 2D matrices and 1D vectors as arrays in memory.
`tm-tensor-to-loops`	Replace tensor arithmetic with loops and scalar arithmetic.
`convert-linalg-to-parallel-loops`	Replace linear algebra operations with parallel loops and scalar arithmetic.
`lower-affine`	Replace affine expressions (e.g. padded and strided address calculations) with integer arithmetic.

ation space, just like a `Kokkos::parallel_for`. `scf.parallel` can optionally perform reduction to an output variable as well, becoming equivalent to `Kokkos::parallel_reduce`. The `memref` dialect provides several operations for operating on strongly typed multidimensional memory buffers, just like `Kokkos::Views`. Table 1 shows the equivalence between some MLIR instructions and Kokkos that we use in the Kokkos emitter. As in LLVM, names preceded by % in MLIR denote SSA (static single assignment) values. In the MLIR, type annotations are omitted here but are always present in real code. Table 2 describes the effects of some of the built-in transformations we use to generate MLIR in the desired dialects.

Like the built-in C++ emitter, our Kokkos emitter performs an in-order walk of the MLIR syntax tree and emits the C++ code for one instruction at a time. The SSA form of MLIR makes this straightforward – we simply store the result of each instruction in a new C++ variable, and later rely on the C++ compiler's optimization to keep variables alive for only the duration they are needed. The one exception is that for scalar constants (`arith::constant`), each reference to the SSA variable is replaced by its value as a literal. This improves performance when using the CUDA backend because the compiler does not propagate host constants into device code (see Table 3 for the speedup of this optimization). Assigning one `Kokkos::View` to another does an inexpensive shallow copy operation with reference counting, so memory management and safety do not cause any issues. Globally scoped Views (such as the constants for the model) are allocated and filled during a module initialization function and deallocated during a finalization function. These Views cannot simply alias constant arrays within the library since they may end up in GPU memory. Views in GPU memory cannot be allocated until after Kokkos has been initialized at runtime.

The main pipeline is complete once the Kokkos C++ code has been written to a file. For the purposes of testing, a Kokkos backend class was created in Python. This backend works as a drop-in replacement for the JIT RefBackend of torch-mlir. Given an MLIR module from the PyTorch frontend and a location where Kokkos is installed, our backend automatically goes through the steps needed to actually run the generated code:

- Execute the transformation pipeline described above
- Emit Kokkos C++ to a file in a known location
- Compile the Kokkos C++ into a shared library with the addition of a CTypes-friendly wrapper (tensors are in the form of raw host pointers and sizes, and function names are not mangled)

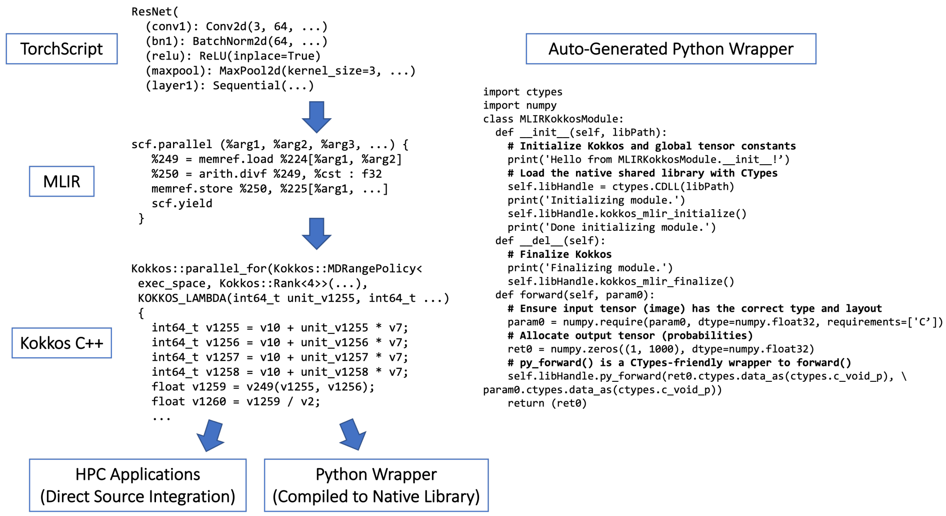

Figure 1: Full compiler pipeline from PyTorch to natively compiled Kokkos C++, and the automatically generated Python wrapper module.

- Generate a Python wrapper module using the CTypes API, which accepts tensors as NumPy arrays
- Import the module (this loads the shared library, which in turn initializes Kokkos)

This extended pipeline is useful for testing because the resulting wrapper module provides an identical interface to the example PyTorch to LLVM JIT backend. In the case of the ResNet18 example, using our code is as simple as:

```
import kokkosModule
probabilities = kokkosModule.forward(image)
```

where “image” is a preprocessed NumPy array representing the pixel values of the input, and “probabilities” is the resulting vector of probabilities that the image belongs to each class. This vector can be directly compared with the one returned by the interpreted PyTorch model, or the LLVM JIT version of the model. Figure 1 outlines the overall pipeline, and also shows the full annotated source code of the Python wrapper module for ResNet18. We show the example transformed to serial C++ backend and OpenMP and CUDA backends of Kokkos.

3 Results and Future Work

The primary result of this work is the confirmation that a compiler-based approach can be used to bridge the language gap between ML frameworks in Python and scientific applications in C++. To the best of our knowledge, this project is the first example of a compiler that maps high-level operations like convolutions to a portable programming model like Kokkos. Although our Kokkos emitter is only a prototype, it is general enough to work with any MLIR program that has been transformed into the subset of dialects our emitter expects (`scf`, `arith`, `memref`, etc.). To be production-ready, the compiler should be packaged into a single executable, rather than as multiple components split across different parts of the MLIR repository. In the future, other frontends beyond PyTorch and torch-mlir can be supported. TensorFlow [1] and JAX [2] can both be compiled to high-performance native code through the IREE compiler [6], which is also based on MLIR. As in this project, we could compile Python programs to Kokkos C++ source code by using the right sequence of lowering transformations and our Kokkos emitter. Alternatively, the full capabilities of IREE could be used to produce native binaries and a separate MLIR pass could generate interfaces for seamless integration with scientific applications.

Another area needing improvement is the efficiency of the generated code. Although linear algebra and tensor operations can be expressed as multidimensional parallel for loops, such naïve implementations

Table 3: The time per inference on four different implementations of ResNet18.

ResNet18 Implementation	Inference Time (s)
Interpreted PyTorch	0.357
LLVM JIT (RefBackend)	14.9
Kokkos (OpenMP, 8 Threads)	14.2
Kokkos (Cuda)	0.918
Kokkos (Cuda) w/ Constant Prop.	0.722

do not make effective use of caches or GPU scratchpad memory. Hand-optimized tensor kernels that use explicit tiling could be a major improvement (e.g. the cuDNN library). Rather than lower every tensor and linear algebra operation to a parallel loop, an extra transformation could be inserted to detect the operations for which optimized kernels are available and replace them with calls to those kernels instead. The ResNet18 example would benefit especially from 2D convolutions being optimized in this way.

Table 3 shows the times to execute 3 different implementations of ResNet18 inference: interpreted PyTorch, the LLVM JIT backend, and our Kokkos backend compiled for CPU and GPU. The PyTorch and LLVM JIT versions were run on a single Intel Skylake CPU core (Xeon W-2155), while the Kokkos versions enabled either the OpenMP backend (8 threads on the same CPU) or the Cuda backend (Quadro P2000 GPU). Although the Kokkos version is slower than the interpreted PyTorch version, our module compiled for Cuda (especially with scalar constant propagation) is faster than the serial LLVM JIT code. ResNet18 provides a benchmark to develop optimizations in the future beyond our initial proof-of-concept compiler.

4 Conclusion

Physics-informed AI/ML is becoming an important tool for computational science across many domains, from molecular dynamics to electronic circuit design. ML is being used to learn the behavior of physical systems so accurately that some first-principle numerical computations can be replaced by model inference at a much lower computational cost [3]. However, scientific computing work favors a C++ software ecosystem, while modern machine learning is best done in Python. There are several ways to interface between the two languages but all of them require the tedious process of explicitly defining the interface of each function to be called from the other language.

We demonstrate that a compiler using MLIR can automate this process. Domain scientists can exchange data seamlessly between simulation code in C++ and machine learning models from Python. Our compiler can take a general PyTorch model and generate parallel, portable C++ source code that uses the Kokkos programming model, making it trivial to integrate into existing Kokkos-based applications that can run on different CPUs and GPUs.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow, Large-scale machine learning on heterogeneous systems, 11 2015.

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

[3] J. A. Ellis, L. Fiedler, G. A. Popoola, N. A. Modine, J. A. Stephens, A. P. Thompson, A. Cangi, and S. Rajamanickam. Accelerating finite-temperature kohn-sham density functional theory with

deep neural networks. *Phys. Rev. B*, 104:035120, Jul 2021. doi: 10.1103/PhysRevB.104.035120. URL https://link.aps.org/doi/10.1103/PhysRevB.104.035120.

[4] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308.

[5] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[6] The IREE Authors. IREE. https://github.com/iree-org/iree, 9 2019.

[7] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4): 805–817, 2022. doi: 10.1109/TPDS.2021.3097283.