⚡Instant Rice 🍚

A book of quick and easy Rice runtimes for weeknight compiling,

By Gavin Zhao & Milo Kron

Summary

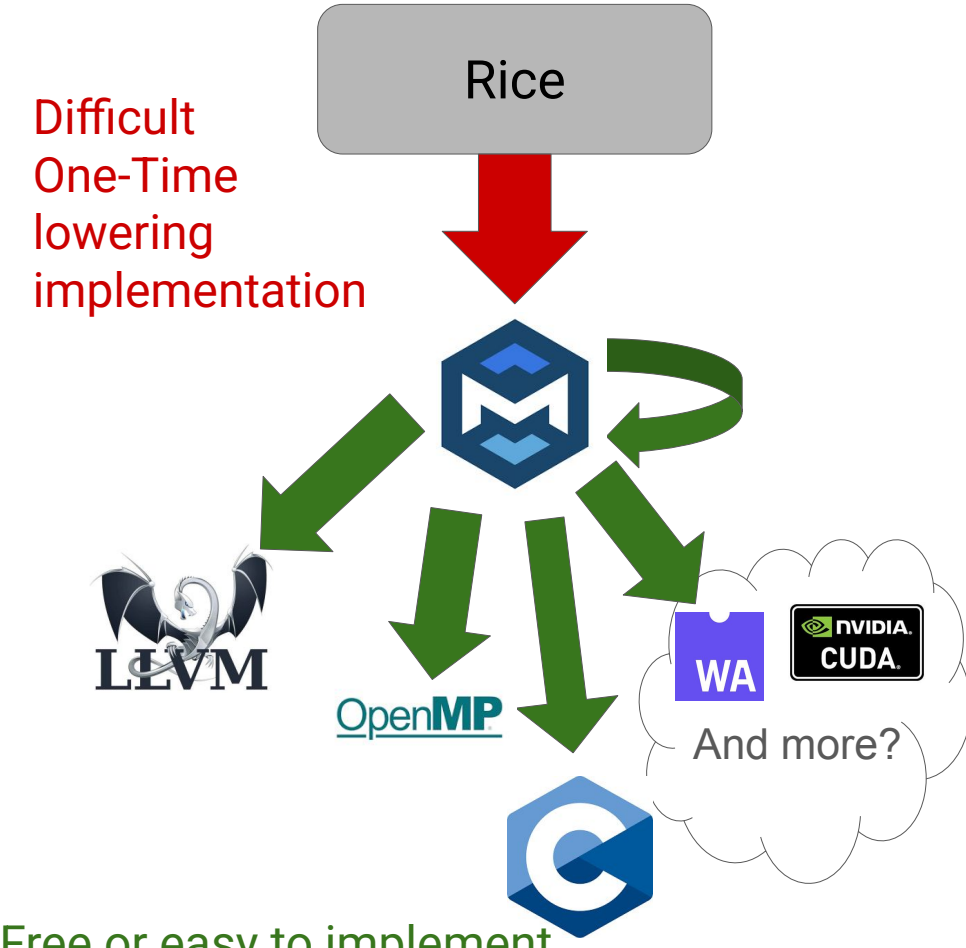
This project presents several implementations of **Rice**–an academic language used in Will Crichton's CSCI1951Q, conventionally compiled to Web Assembly–using **MLIR** (Multi-Level Intermediate Representation), a general purpose intermediate representation and compiler infrastructure.

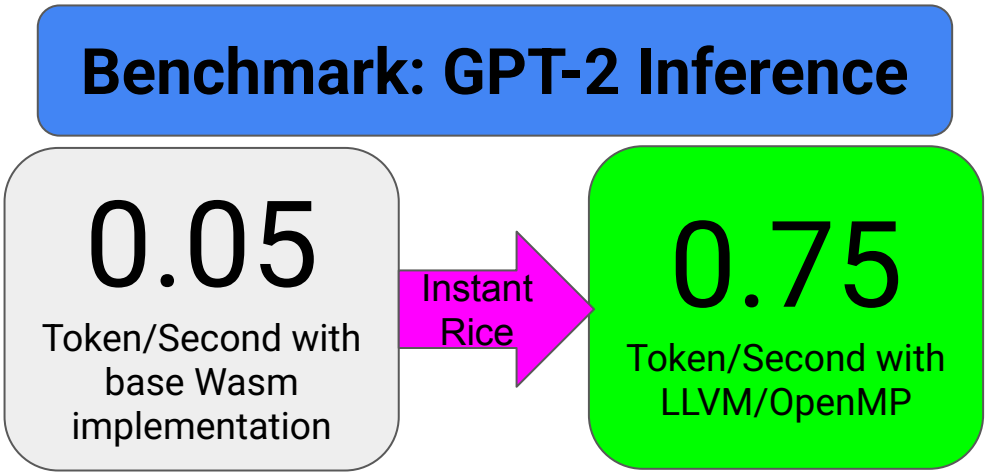
Motivation

The exercise is meant to demonstrate MLIR's support for relatively easy conversion into several lower level runtimes. In the course of this project, we implemented a single compilation step from (a subset of) Rice to MLIR, followed by a number of MLIR passes which allowed us to compile Rice source code t0 OpenMP, C, and LLVM.

Results

We ran GPT-2 inference in Rice's base Wasm runtime, and in an optimized, parallelized compilation using LLVM and OpenMP. This resulted in **15x** throughput improvement.





GitHub source: https://github.com/Second-Last/csci1951q-rice-mlir. Reproduction instructions in the README.

Introduction

MLIR (Multi-Level Intermediate Representation), released in 2019, is an intermediate representation and compiler infrastructure framework. It can be thought of as “an IR to rule them all”. It achieves this by allowing many different dialects to coexist in one MLIR program as long as the types of the arguments to an operator are accepted. A dialect defines custom types and operators, and these types and operators can operate on types and operators from other dialects. There are *low-level dialects* that are almost if not exactly word-by-word translatable to low-level code, such as the `llvm` dialect for LLVM and `emitc` dialect for C. A MLIR program written purely in the `llvm` or `emitc` dialect are really just LLVM IR or C programs written in the MLIR syntax. Then, there are *high-level dialects* like:

- `builtin` for basic datatypes like `i32`, `i64`, `f32`, etc.
- `memref` for describing regions of memory.
- `scf` for describing control flow blocks such as `if` and `for` loops.
- `arith` dialect for describing arithmetic operations.
- `tensor` dialect for describing abstract operations on tensors.

There are generally two types of passes in MLIR: *optimization passes* that don't transform its inputs to a new dialect, and *translation passes* that transform its input to a (usually) lower-level dialect. The beauty of MLIR is that passes that work on one (or a few) dialects can still run even if the program contains other dialects. MLIR’s power comes from this composability. For example, an optimization pass written for the `memref` dialect automatically benefits all languages that represent their array operations using `memref`.

In this way, MLIR hopes to, quoting from the MLIR website, “significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together.” MLIR has become popular especially in the AI community for optimizing AI models or transforming code to run on custom hardware accelerators. However, no mainstream programming language has a MLIR-based compiler that is used in production.

Even worse, most existing tutorials of MLIR simply stop at being able to compile their toy language into the `llvm` dialect. Very few tutorials explore the modularity aspect of MLIR. That is, the language implementer only needs to implement a dialect and appropriate translation passes *just* for the special operators and types of their language that cannot be expressed with any existing dialect; then, with little to no code, they can use the optimization and transformation passes available from MLIR’s wide ecosystem to compile to multiple backends, such as `llvm` dialect for compiling to LLVM IR, `emitc` dialect for compiling to C, `openmp` for CPU parallelization, and `spirv` for compiling to GPU, etc. To our knowledge, there exist no

up-to-date examples that show an end-to-end compiler compiling a toy language to *multiple* backends/runtimes using MLIR.

Is this just because language implementers are still catching up to this new technology, or does MLIR itself poses usability issues despite the advantages that it claims? To answer this question, we will implement a MLIR backend for the Rice compiler and benchmark it on a Rice implementation of LLM text generation on the GPT2 model. Rice is the instructional language used for CSCI1951Q that compiles to WebAssembly and uses the Wasmtime runtime. Rice is an imperative language and its features important to this project are: ints, floats, strings, multi-dimensional arrays, if statements, and while loops. These features, plus some standard library functions for performing file I/O, are all we need to implement GPT2 inference.

This compiler will emit MLIR using high-level dialects mixed with our own `rice` dialect for custom Rice operators. Then, we will write the appropriate translation passes that only need to translate the `rice` dialect into lower-level dialects. We'll then use existing MLIR passes to compile a Rice program down to native machine code and *parallelized* native machine code (through OpenMP runtime). This project serves both as a learning experience for MLIR and an opportunity to evaluate how well the current status of MLIR holds up to its claims.

Implementation

Since many high-level MLIR dialects are building blocks for high-level code, we decided to generate MLIR from TIR.

Rice Dialect Design

To maximize the modularity of MLIR, we want to avoid using custom `rice` dialect operators as much as possible and try to represent as much as we can of the language using high-level MLIR dialects:

- `fnt` -> `i32`
- `float` -> `f32`
- `string` -> `memref<?xi8>`

(Any object-oriented code is not supported.)

Note on `memref`, since it'll be referenced in more detail later:

- `memref<nxT>` is a pointer to a block of `n` contiguous elements of type `T` in memory.
- `memref<?xT>` is the same, except that the number of elements is unknown at compile-time.
- `memref<nxmxT>` is a point to a block of `n * m` contiguous elements of type `T` in memory. By default it uses a row-major layout with `n` rows and `m` columns, but one can specify different layouts by adding more parameters like `memref<nxmxT, layout>`.

- `memref<?x?xT>` is similar except that both the row and column dimensions are unknown.

Simple binary operators like math operations are directly compiled to their corresponding operators in the `arith` dialect, so we won't discuss them in detail.

Rice Arrays

We'd like to particularly highlight the design of Rice arrays. The original, canonical Rice arrays essentially follow the array-of-pointers design, where a multi-dimensional array like `[[int]]` doesn't store all the `int`s in a contiguous block of memory, but instead stores an array of pointers to `[int]`s. This makes the implementation pleasing, but is not friendly to performance optimizations, especially parallelization.

Our design is to have two array representations. Say we have a `[[int]]`.

- The original array-of-pointers design, compiles to `memref<?xmemref<?xi32>>`.
- Contiguous array, compiles to `memref<?x?xi32>`.

The beauty of MLIR is that in the high-level Rice MLIR dialect, they're represented using a unified custom type, `rice.array<T>`. Then, there are two passes that implement the two different compilation strategies. Then, the contiguous array pass is configured to only run on functions annotated with the `#[kernel]` attribute, where the array-of-pointers pass is run on all other code. Array operators are also encoded in the `rice` dialect such as `rice.array_load` and `rice.array_copy`, that gets compiled to their corresponding `memref` implementations.

This aligns with the design philosophy of MLIR: you have one high-level dialect that is more generalized and consistent across different representations/backends, and then you have different passes to lower them to different representations/backends depending on your needs.

(Admittedly this changes the semantics of the Rice programming language a bit, but the original behavior was not used in the GPT2 implementation so in the end this had no effect. If we were lowering to MLIR after the BC stage, we could've run some dataflow analysis to inform us what arrays never pass out a portion/slice of their contents. It is one of our "what-if"s that if we had tried to compile to MLIR after the BC stage, we might have harder initial MLIR generation but we could've generate optimized code due to the dataflow analysis passes we would've been able to run)

Parallelization

This is where the modularity of MLIR comes into play. Normally, if you want to implement parallelization support for a programming language, you need to either go really low-level and implement all those threading using libc, or generate obscure, almost DSL-like code such as

OpenMP. However, here in our Rice MLIR compiler, even though we also compile to OpenMP, we only need to write a single simple pass to make loops parallelized.

The pipeline to compile to OpenMP is:

1) Rice has a `scf` dialect for writing control flow such as `scf.if` and `scf.while`, which is what we compile our Rice `if` loops and `while` loops into.
2) Rice doesn't have `for` loops, but MLIR has a pass `scf-uplift-while-to-for` that transforms `scf.while` loops with certain patterns into equivalent `scf.for` loops. Most of our `while` patterns for array iterations are simple enough to be recognized.
3) We write a pass `scf-for-to-parallel` that transforms all `scf.for` in functions annotated with `#[parallel_for]` into `scf.parallel`, which has the exact same API as `scf.for` except that each iteration of the loop gets run in parallel. This pass is less than 100LOC.
4) MLIR has a pass `convert-scf-to-openmp` that translates `scf.parallel` for loops into OpenMP parallel constructs.

The only part we had to write code for is to 3) write the pass `scf-for-to-parallel`. Then, by specifying the correct order to apply those passes, all `for` loops in functions marked as `#[parallel_for]` gets parallelized through OpenMP. In less than 100 LOC, we achieve a 2x speed up on a 8-core machine compared to the non-parallelized MLIR version.

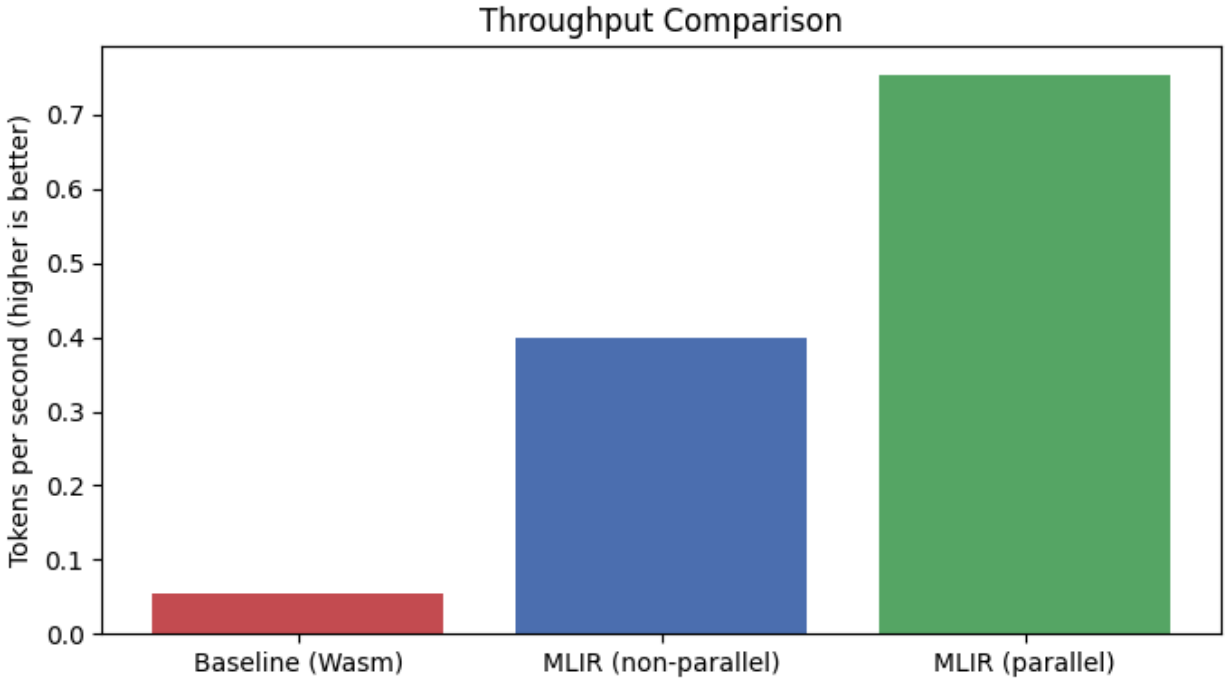
Evaluation

We translated PicoGPT, a reference implementation for generating text using the GPT2 large language model, into Rice. Due to the 4.1GiB memory limit of the Wasm runtime, our GPT2 implementation only runs the core algorithm which is the 12 rounds of transformer blocks. The pre-processing of converting text into tokens and weights and post-processing of converting the matrix into string text is done in a Python wrapper script. Our running time measurement only contains the time it takes for the Rice program to run and doesn't include the Python wrapper's running time.

We compare the throughput (number of tokens generated per second) of this code generating text using the GPT2 124M model with the following three implementations:

1. Baseline (Wasm): The baseline performance uses the Rice compiler compiled with `--release` and `#[jit]` annotated for every function. Garbage Collection was disabled (using `Collector::Null` as the garbage collector) to give a fair comparison since we never free our arrays in our MLIR implementation.
2. MLIR (non-parallel): translates the Rice compiler's high-level MLIR output into low-level `llvm` dialect which MLIR then translated to LLVM IR and we manually compiled to machine code with `clang -O2`.

3. MLIR (parallel): the same as above, but between translating the high-level MLIR to low-level `llvm`, we insert passes that convert Rice `while` loops into parallelized `for` loops expressed with the `openmp` dialect, which itself also has a pass to get translated into the `llvm` dialect. The final LLVM IR is compiled with `clang -O2 -fopenmp`.



We can see that in the best case, the parallel MLIR is around 15x faster than the baseline performance. Even the non-parallel MLIR is around 7.5x faster than the baseline, but it's possible that this is mostly due to running in Wasmtime vs running native code, so this improvement is less significant.

Perhaps the most significant result is this: the pass converting non-parallel to parallel MLIR was 100 lines, took, generously, half an hour to write, and yielded a 2x improvement in performance. We think this alone is enough to demonstrate the modularity of MLIR, and give a sense of how much time and effort it can save for language implementers. We see this even in less than optimal implementation conditions–we did not have a full dataflow analysis pass. If we were to redesign Rice's compilation pipeline, one could imagine having access to more fine-grained analysis before MLIR code generation, which would give us an opportunity to parallelize more parts of Rice. For example, `scf.parallel` is a very aggressive operation and doesn't do any checks to make sure the iterations are independent, which means MLIR cannot be too aggressive itself when translating it down to `openmp`. However, if we had a dataflow analysis pass that provides proof of independent accesses between loop iterations, we could supply this proof to the `affine.parallel` operation from the `affine` dialect which provides more optimizations like loop fusion and reordering out of the box.

We have also experimented with compiling Rice to C by lowering the `rice` dialect to the `emitc` dialect. This was our initial implementation, but unfortunately the `emitc` dialect only allows `memref` with static dimensions (i.e. no `memref<?xi32>`), so `rice.array<T>` must be lowered directly to `emitc` which proved to be too difficult. We decided to switch to lowering to the `llvm` dialect in the middle of implementation.The ease of this transition was in itself an endorsement of MLIR's flexibility.