µCompiler: A Domain-Specific Compiler for LLM Inference on Embedded Accelerators

Michael Nath [1]

Abstract

Domain specific accelerators - specialized chips that are designed to efficiently execute a specific worklod often require unique instruction set architectures (ISAs). µThinker, an accelerator developed for large language model (LLM) inference on embedded micro-robotic agents, features an ISA optimized for the transformer forward pass. The complexity of µThinker's ISA and the precise, model-dependent order of operations mean that distinct Transformer models require significantly different instruction programs. This necessitates µCompiler, a domain-specific compiler designed to translate Transformer model weights and configurations into programs compatible with µThinker's ISA and hardware design.

µCompiler houses several featuers key to the programmability of µThinker. It provides native support for µThinker's tiled computation and data movement. µCompiler investigates the design of a novel Intermediate Representation (IR) for an embedded C interpreter. This IR employs compile-time templating with runtime placeholder mechanisms to manage dynamic inputs (such as current sequence length) and complex operational flow on µThinker. Finally, µCompiler defines a binarization format for this IR, which enables the embedded C interpreter to parse, execute, and orchestrate heterogeneous operations between the CPU and the µThinker accelerator.

1. Introduction

Domain specific accelerators are specialized processors that are designed to efficiently execute a particular workload. By virtue of their specificity, they often involve a unique instruction set architecture (ISA). At the Tambe Lab, we are developing such an accelerator where the domain is large language model (LLM) inference embedded in a micro-robotic agent. These robots weigh only a few pounds, are no bigger than a closed fist, and are often deployed to intelligently perform time-sensitive operations such as search-and-rescue during natural disasters. Thus, it is crucial for the LLM inference to have as low a latency as possible in between receiving an input and generating a course of action for the agent taken. Our accelerator — dubbed µThinker — therefore possesses an ISA optimized for just one operation: a transformer forward pass. Transformers are a popular class of models that are used for LLM inference, where the forward pass is distinguished by running the Attention operation popularized by (Vaswani et al., 2023).

The memory consumption of the Attention operation scales quadratically with the length of the input text. Programs that perform LLM inference on micro-robotic chips, which are physically constrained to hold little memory, must therefore be efficient with data movement during the inference. µThinker alleviates much of the memory bottlenecks by enforcing a rigid ISA that expects the programmer to submit a precise order of instructions representing the Attention operation. This precise order allows µThinker to internally optimize data movement and computation. Categories of instructions include: configuring various components of the processor, running matrix multiplications on the compute units, and grabbing data from main memory. While the order (e.g. configuration and data transfers occur done before matrix multiplication) of these instructions remain constant across all Transformer models, their frequencies relative to one another can vary significantly depending on the model size and parameter characteristics. For instance, a model that contains tenfold more parameters than another model will roughly require tenfold more data transfer operations while still sharing the same sequence of configuration instructions. Accelerating just one Transformer model manually involves painstaking calculations to ensure that the right data is going to the right location on µThinker for processing. For large enough models, it may even become intractable to perform instruction generation by hand.

These issues motivate the design of a compiler, µCompiler, which is a software that can configure a given Transformer model to a program of instructions compatible with µThinker's ISA and hardware design.

2. Compiler Overview

µCompiler takes as input the weights of some Transformer model (e.g. its `model.state_dict()`) as well as its configuration (e.g., number of attention heads, dimension of each head, etc.). The micro-robotic agent will leverage models such as MobileLLM, which is derived from the Llama family of Transformer models (Liu et al., 2024; Touvron et al., 2023). Therefore, both µCompiler and µThinker assume the structure of the input to be of Llama-like models. Figure 1 provides a high-level overview of this compilation process.

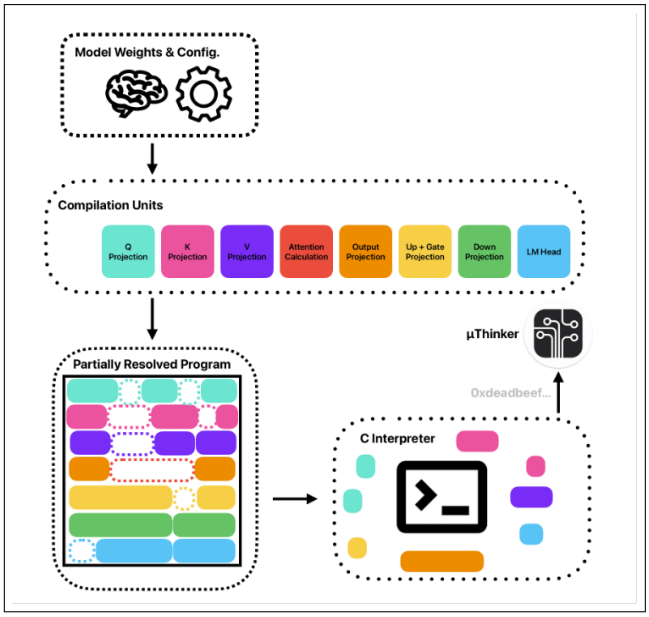


Figure 1. High-level overview of the µCompiler workflow. Model weights and configuration are streamed throughout compilation units that serve as templates for the order of operations on µThinker. These units output a partially resolved program, which is then fed to an on-device C interpreter that interacts with the µThinker accelerator.

Table 1 enumerates the high-level operations of the Transformer forward pass in the order that µThinker executes them. For each operation, µCompiler emits the corresponding sequence of instructions while referencing the input whenever required. Once all operations have been successfully implemented, µCompiler encodes the program of instructions to a format that can be fed to the CPU accompanying µThinker. The CPU will then place the program in the instruction memory of the accelerator, and now the accelerator is able to execute the compiled instructions whenever the CPU asks the accelerator to perform inference.

Table 1. Transformer Fwd. Order of Operations

Stage	Input buffer	Weight buffer	Destination
Q proj.	I_BUF0	W_BUF0	I_BUF1
K proj.	I_BUF0	W_BUF1	W_BUF1
QK matmul	I_BUF1	W_BUF0	GB
V proj.	I_BUF0	W_BUF1	W_BUF0
PV matmul	I_BUF1	W_BUF0	GB
O projection	I_BUF0	W_BUF1	GB
UPGATE proj.	I_BUF1	W_BUF0	I_BUF0
DN proj.	I_BUF0	W_BUF1	GB
LM head	I_BUF1	W_BUF0/1	DRAM

2.1. Compiler Requirements

2.1.1. Tiling

µThinker physically supports buffers for the various kinds of matrices involved in a Transformer: embedding vectors, weight projection matrices, attention matrix, etc. The processing elements, which house the multiply-and-accumulate computation units of µThinker, are intended to perform matrix multiplications. Since µThinker will be embedded in a micro-robot, factors such as power consumption, hardware utilization, and rate of data movement become crucial to efficiently executing gigabyte-scale Transformer models. To satisfy these factors, we have designed µThinker to work in terms of tiles: parameters such as model weights must be streamed to the buffers in tile-sized chunks; due to the memory consumption of large intermediates in the forward pass, processing elements must perform tiled matrix multiplication. Therefore, we needed to make sure that µCompiler principally supports tiled data movement and computation on µThinker.

2.1.2. Quantization

Jang & Tambe (Jang & Tambe, 2025) have developed a novel block-quantization scheme for efficient LLM inference. µThinker natively supports this quantization and expects all tiled data to be quantized accordingly. We want µThinker to focus on just running the forward pass of a Transfomer model, so we perform the model quantization before we deploy the accelerator to run a real workload. This offline quantization motivated us to design µCompiler to include quantization as part of the compilation process. Here, µCompiler performs quantization on the model weights and subsequently tiles the quantized data. The compiler then emits data transfer instructions so the accelerator could retreive the correct data. Thus, code generation is deeply coupled with quantization and tiling.

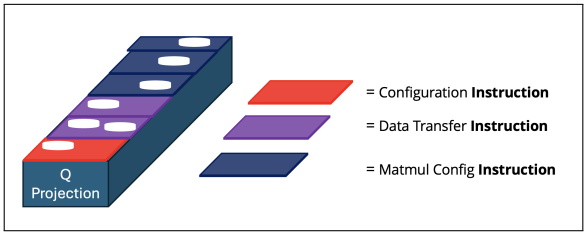
To accommodate the tiled nature of data movement and computation, the idiom of µThinker is to repeat operations for as many tiles as there are available. As an example, the attention score calculation in µThinker is chunked iterative:

1) the input sequence of N tokens is broken up into N / N_T chunks, where N_T denotes the number of tokens per tile. 2) For each tile T, we then iterate through every other tile T'. 3) We compute query states Q_T for tile T and compute key states K'_T for tile T'. 4) We produce the attention scores for just Q_T and K'_T, storing the tiled result on the accelerator. We repeat steps 2-4, but this time with the next candidate for T'. Once that is complete, we move to the next candidate for T.

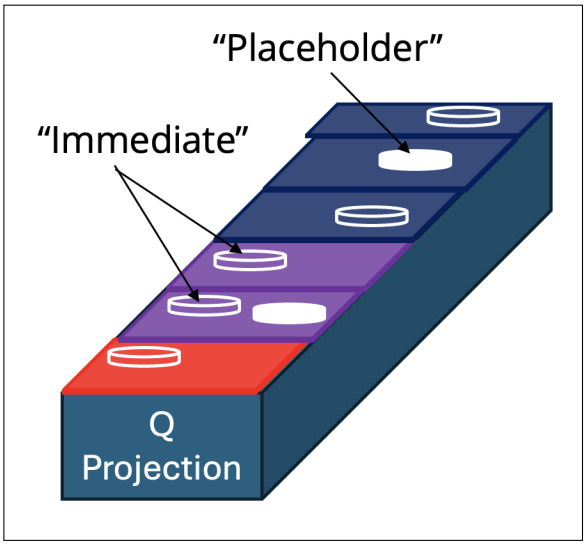
2.1.3. Making the most of Compile-time

Writing µCompiler to express this idiomatic looping involves handling two kinds of values: compile-time parameters and runtime variables. The above attention score operation proves this: N_T is a parameter that can be passed as an argument to the compiler program; on the other hand, N is a variable that depends entirely on the host program. Compilers typically address dynamic loops by leveraging both the notion of a processor register and the notion of a jump instruction. µThinker has neither. Since the atom of µThinker is a tile, and a significant amount of tiling in µThinker is derived from runtime values, this also presumably means a significant amount of loops cannot be described by µCompiler during compilation. The principle of µThinker's parent System-on-Chip (SoC) is to do as little configuration during runtime as possible; dynamic loops would therefore render µCompiler practically useless. To address this requirement of making the most of compile-time, µCompiler leverages the precise order of operations on µThinker to produce an IR that templates the entire forward pass. This template, which can be thought of as a partially resolved compilation, gets fully resolved during runtime once the dynamic inputs are available. Figure 2 illustrates this concept.

Because µThinker is part of a SoC, the accelerator must coordinate with the CPU when executing the forward pass. Using the attention score operation is the example, this is most apparent when sending over the projection weights from the SoC DRAM to the various buffers on the accelerator. The mechanism of transfer is the on-chip DMA engine available, which can be programmed only by the CPU. Thus, it is not enough for µCompiler to ultimately produce a sequence of accelerator-specific instructions. Rather, the compiler has to be able to "talk" to the CPU and instruct it to perform CPU-side operations such as setting up a DMA transfer with the appropriate source DRAM address and destination accelerator address. Therefore, in a sense, µCompiler is required to take on the role of a heterogeneous compiler. The CPU - in the form of an embedded C program - is then in charge of processing the instructions emitted by µThinker. The program executes the CPU-side instructions and otherwise delegates execution to the accelerator.



(a) The stack of instructions for a single operation. For the Q Projection stage, µCompiler emits configuration, data transfer, and matrix multiplication setup instructions.



(b) The same instruction stack, highlighting the different types of values. Some instruction fields are compile-time constants (Immediates), while others are resolved at runtime (Placeholders).

Figure 2. A conceptual model of the templated IR for a projection step in the Transformer forward pass. (a) shows the different types of instructions involved. (b) illustrates how instruction fields can be either compile-time constants pulled from the input model configuration or left as unknowns only resolved during runtime.

3. Implementation

3.1. Tiling and Quantization

The processing elements of µThinker parallelly operate on tiles that take the form of submatrices that are 64 rows long and 8 columns wide. Throughout the forward pass, these input tiles originate from either the SoC DRAM or one of the accelerator buffers. If the input originates from DRAM, µCompiler emits a DMA transfer operation. While based on the current stage of the computation it may be clear as to where to send the tiles, it is not as obvious as to where the tiles reside in DRAM or how many bytes to transfer from DRAM. This is because of the quantization scheme. Block-based quantization formats operate by partitioning a contiguous array of elements into blocks of a fixed size. The quantization scheme then independently compresses each

block; some blocks may compress more easily, yielding fewer bits per element than those from another block. Thus, quantization schemes such as the one used for µThinker yield variable-sized elements, and in particular, the scheme developed by Jang & Tambe also yield variable-sized blocks. Therefore, when emitting the CPU-side DMA instruction, µCompiler must maintain a mapping of tiles to the blocks they span.

µCompiler accommodates the variable-sized quantization by first specifying a protocol for the quantization scheme to follow before and after quantization. First, each element of the original array is tagged with what tile it would belong to. Before any quantization, the scheme proceeds to tag all blocks with what tile they would correspond to based on the tagged tile of the constituent elements. Once the blocks become quantized, they are then laid out in DRAM in ascending order of their tile identifier. As each block is laid in memory, metadata describing the block size in bytes is prepended. At the beginning of the block sequence, further metadata is inserted, which describes the location of all block metadata — the offset from the base address of the sequence — and keyed by their tile identifier.

If this protocol is executed offline, then a custom file format storing these blocks and their metadata can be provided. During compilation, the bytes from the file are loaded into process memory and, thus, whenever µCompiler wishes to emit a DMA instruction, it can pull the required parameters from the formatted block sequence. If the quantization is done concurrently with the compilation, the block sequence can be directly stored into process memory.

3.2. Dealing with Runtime Variables

Dynamic inputs appear plenty during the compilation process. While the current sequence length is one that is observable to the end-user and must be known for appropriate tiling, µThinker also needs to know whether a tile is the last of its kind. Referencing the attention score operation, this may be informing µThinker that it is at the last tile T, or perhaps when it is at the last tile T'. The accelerator is not able to deduce this internally precisely because of inputs such as the dynamic sequence length. To accommodate dynamic values while doing as much as possible with what can be determined during compile-time, a custom IR is created for µCompiler.

Figure 3 summarizes the constructs available in this IR. The unit of execution in this IR is an `Instruction` object, which can describe any of the CPU-side or accelerator-side instructions. As there are fields in our instructions, an `Instruction` is composed of fields that are of type `Variable`. In our IR, there are two kinds of `Variables`: an `Immediate` or a `Placeholder`. Compile-time parameters, such as the fixed weights of a Transformer model and its configuration, are represented in this IR as `Immediates`. Runtime inputs, such as the current sequence length and any corollary values for µThinker, appear as `Placeholder` nodes. Once we have a few `Instructions` defined, then we can encapsulate them in a `Code` object. To repeat the execution of a `Code` object in a loop, where each invocation is parameterized by a loop variable, then there also exists a `For` object.

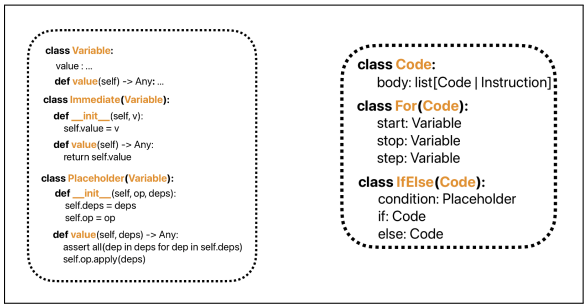


Figure 3. Summary of the constructs available in the custom Intermediate Representation (IR) for µCompiler, including `Instruction`, `Variable` (`Immediate`, `Placeholder`), `Code`, and `For` objects.

With `Instruction`, `Immediate`, `Placeholder`, `Code`, and `For`, µCompiler can produce a IR that deals with the runtime and compile-time properties of the Transformer workload. Whenever µCompiler wishes to emit an `Instruction`, it simply has to annotate the fields as being either a `Placeholder` or an `Immediate` depending on the available context. Placing down an `Immediate` implies that the field has a value calculated from other values known during compile-time. Placing down a `Placeholder` implies a more involved logic: rather than affixing a value, a set of dependencies along with a resolve function is provided. The dependencies are other `Variables`, and the resolve function describes the operations that must be performed on the dependencies so as to ultimately resolve a value for the field.

Eventually, after appropriately annotating all `Instructions`, a `Code` object is produced. For a given Transformer model, this `Code` object represents a template of the sequence of instructions that must be executed. This `Code` object remains a template so as long as there are unresolved `Placeholder` fields in any of the contained `Instructions`.

3.3. Interpreting Heterogenous Instructions

The IR that is produced in compile-time is a standalone description of the forward pass of some Transformer model. `Placeholders` prevent the need for recompilation, allowing the IR to be portable and unique to a model. Yet,

by virtue of being an intermediate representation, this IR is not executable. While compilation can occur on a PC offline, the IR must ultimately be lowered to the sequence of formatted hexadecimal bytes interpretable by the CPU or the accelerator. This lowering can only happen once the runtime — in this case an embedded C program executing on the CPU — provides the appropriate inputs to the requesting `Placeholder` objects. Therefore, the last component of µCompiler is an embedded C interpreter that parses the IR and recursively resolves all `Placeholders`.

Before parsing the IR, the interpreter assumes that the IR has been properly flattened and serialized. Each `Instruction` becomes serialized to a hexadecimal number that is a function of the fields determined by either the ISA of µThinker or the requirements of the CPU. For fields that are `Placeholders`, a label is included in the serialization that references the identifier of the `Placeholder`. The IR becomes flattened in the sense that all `Placeholders` come before the `Instructions` that depend on them. This ensures that the interpreter can resolve those values before resolving the overall `Instruction`.

Once an instruction has been resolved, the interpreter can either dispatch the execution to the CPU or to the accelerator. The decision depends on the opcode accompanying the serialized `Instruction`. In this way, the interpreter goes through the sequence of bytes and terminates once all bytes have been processed.

4. Status & Next Steps

µCompiler is currently able to take an input Transformer model, produce the templated IR, and serialize it to a format interpretable by the embedded C interpreter. Now I am focused on optimizing the interpreter to integrate with the micro-robotic applications that the lab has in mind. Meanwhile, I am setting up µCompiler to be compatible with the testing framework the lab has. This will allow the compiler to participate in the many integration tests that are set up in the µAgent project.

Acknowledgements

µCompiler is part of the ongoing µAgent project at the Tambe Lab. I am grateful to Professor Thierry Tambe and the µAgent team for giving me the ownership to work on µCompiler. I thank Professor Fred Kjolstad for taking the role of CS191W advisor. Lastly, I appreciate the CS191W program for sponsoring an opportunity to conduct work on µCompiler.

References

Jang, W. and Tambe, T. Blockdialect: Block-wise fine-grained mixed format quantization for energy-efficient LLM inference, 2025. URL https://arxiv.org/abs/2501.01144.

Liu, Z., Zhao, C., Iandola, F., Lai, C., Tian, Y., Fedorov, I., Xiong, Y., Chang, E., Shi, Y., Krishnamoorthi, R., Lai, L., and Chandra, V. Mobilellm: Optimizing sub-billion parameter language models for on-device use cases, 2024. URL https://arxiv.org/abs/2402.14905.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models, 2023. URL https://arxiv.org/abs/2302.13971.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.