TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR FOR COMPILER CONSTRUCTION

Faculty of Computer Science

Center for Advancing Electronics Dresden
Chair for Compiler Construction

Master Thesis

Compiler-Assisted Speculative Decoding for Accelerated LLM Inference on Heterogeneous Edge Devices

Alejandro Ruiz y Mesa

Matriculation number: 5126441

to achieve the academic degree

Master of Science (M.Sc.)

First referee

Prof. Dr.-Ing. Jerónimo Castrillón

Second referee

Dr.-Ing. Asif Ali Khan

Supervisors

Dr. João Paulo Cardoso de Lima

Dr. Guilherme Korol

M.Sc. Moritz Riesteter

Submitted on: 31st October 2025

TU Dresden Fakultät Informatik, Institut für Technische Informatik, Professur für Compilerbau

Task Description for Master Thesis

For:	**Alejandro Ruiz y Mesa**
Degree program:	Nanoelectronic Systems (Master)
Matriculation number:	5126441
E-mail:	alejandro.ruiz_y_mesa@mailbox.tu-dresden.de
Topic:	**Compiler-Assisted Speculative Decoding for Accelerated LLM Inference on Heterogeneous Edge Devices**

Large Language Models (LLMs) are the state-of-the-art of many Artificial Intelligence (AI) tasks. However, the size (billions of parameters) and complexity (hundreds of layers) of these models difficult their wider adoption at the Edge - where resources are restricted. On one hand, modern hardware heterogeneity can help deploying efficient LLM inference on edge devices. With multiple types of processing units, such as CPUs, GPUs, and NPUs, one can optimize the heavy-load LLMs for performance at controllable levels of energy consumption. Still, running LLMs at the edge challenges users given the lack of mature end-to-end AI compilation flows. Compilers like IREE, built upon the MLIR infrastructure, enable users with a flexible front-end and a modular compilation flow supporting heterogeneous hardware targets. However, even this MLIR-based mechanism currently relies on manual intervention and lacks a deeper understanding of the LLM models that can drive further performance improvements.

On the other hand, LLMs optimizations have been proposed to lower the LLMs cost. Particularly, Speculative decoding (SD) is a generation-refinement technique that addresses the sequential bottleneck of the autoregressive generative models. SD reduces latency by anticipating likely outcomes in the model's prediction sequence. Nonetheless, its efficiency critically depends on how the inherent algorithmic bottlenecks are managed and whether the speculative execution phase can effectively leverage the capabilities of heterogeneous hardware. Addressing these concerns is essential for enhancing overall inference performance, particularly in resource-constrained edge devices.

The project task focuses on addressing these challenges by integrating speculative decoding awareness into the MLIR-based IREE compiler. The first step is to systematically identify and analyze the bottlenecks within speculative decoding algorithm implementations. This profiling will enable us to assess where delays in the speculative phase occur and whether these may be mitigated through the use of the available heterogeneous processing units. To achieve this, the student will design and implement an annotation system within the IREE compilation flow to partition the model's graph between the SD components. With the partioning at hand, compile-time layer-wise decisions will be made to dispatch the bottleneck components to the most suitable processing unit.

For evaluating this proposal, the student will use a board containing heterogeneous ARM's Cortex cores and Mali GPU. The proposal will be assessed against CPU- and GPU-only executions of IREE-compiled SD models.

Start:	01.05.2025
End:	31.10.2025
1st referee:	Prof. Dr.-Ing. Jerónimo Castrillón
2nd referee:	Dr.-Ing. Asif Ali Khan
Supervisor:	Dr. João Paulo Cardoso de Lima
Co-supervisor:	Dr. Guilherme Korol

Jeronimo Castrillon Mazo
Digitally signed by Jeronimo Castrillon Mazo
Date: 2025.04.14 07:27:53 -05'00'

Prof. Dr.-Ing. Jerónimo Castrillón
(Professor in charge)

Statement of authorship

I hereby certify that I have authored this document entitled *Compiler-Assisted Speculative Decoding for Accelerated LLM Inference on Heterogeneous Edge Devices* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 31st October 2025

Alejandro Ruiz y Mesa

Abstract

The growing demand for on-edge computing has intensified the need to reduce inference latency on resource-constrained devices. These systems increasingly rely on heterogeneous hardware, such as CPUs, GPUs, and NPUs, to efficiently execute diverse workloads. Large Language Models (LLMs) are at the core of many edge applications, including health assistants, translation, and interactive dialogue systems. However, their autoregressive decoding process introduces a sequential bottleneck that limits parallelization. Speculative decoding has emerged as a promising approach to mitigate this limitation by generating multiple tokens in parallel while maintaining model performance.

This work addresses two interrelated problems. The first concerns performance and productivity: integrating an autoregressive generative pipeline with speculative decoding into a compilation flow, without sacrificing runtime efficiency, while preserving modularity, composability, and ease of use. The second problem raises from the compute diversity present in edge devices and focuses on heterogeneous partitioning: optimal assignment of subgraphs from an LLM with speculative decoding across different processing units (PUs) within a system-on-chip (SoC), particularly during the prefill phase and for short input sequences (which is characteristic of edge workloads). To solve these challenges, we propose an analytic cost model that explores multiple heterogeneous hardware configurations and supports coarse-grained partitioning strategies.

The analytic exploration reveals a maximum speedup of 1.68× for a translation task with a 90% acceptance rate when combining speculative decoding with heterogeneous execution. Moreover, for lower acceptance rates or heterogenous system settings, speculative decoding incurs excessive overhead and heterogeneous execution offers no benefit. The proposed analytic cost model was validated on a edge device confirms the model's predictive accuracy, showing the speedup of 1.68× is achieved at an acceptance rate four percentage points higher than expected. While focused on the prefill phase and short sequences, the proposed framework generalizes to other hardware configurations, inference phases, and speculative decoding techniques.

Contents

Symbols and Acronyms

SoC	System on Chip
NPU	Neural Processing Unit
PU	Processing Unit
SIMT	Single Instruction Multiple Threads
DSAs	Domain-Specific Architectures
DSL	Domain-Specific Language
ML	Machine Learning
DNN	Deep Neural Network
LLM	Large Language Model
CNN	Convolutional Neural Network
KV	Key-Value
TTFT	Time to First Token
FNN	Feedforward Neural Network
MAC	Multiply-Accumulate
MLIR	Multi-Level Intermediate Representation
IREE	Intermediate Representation Execution Environment
IR	Intermediate Representation
SPIR-V	Standard Portable Intermediate Representation V
CFG	Control Flow Graph
AOT	Ahead-of-Time
JIT	Just-in-Time

1 Introduction

The development of software for edge systems has traditionally been considered challenging. This perception is rooted in the constraints inherent to edge platforms, including the limited computational budget, diverse toolchains, and vendor-specific environments. Nevertheless, the recent acceleration in hardware innovation driven by the increasing demand of edge Machine Learning (ML) has exacerbated this challenge rather than alleviated it.

The slowdown of Moore's law has marked a turning point in computer architecture, generating a proliferation of specialized hardware and heterogeneous systems; the industry has turned into architectural diversity with the development of Domain-Specific Architectures (DSAs) as the new paradigm to populate modern edge System on Chip (SoC) designs. While these developments promise significant computational advantages, they also introduce additional layers of complexity into the software stack. Compiler infrastructures and programming models often struggle to keep pace with this hardware evolution, hindering the adoption and the effective utilization of these new architectures.

For edge ML researchers and engineers, this transformation presents both opportunity and difficulty. On one hand, the availability of heterogeneous and accelerated compute enables applications once confined to data centers to run directly on the edge. On the other hand, handling this diversity of hardware requires increasingly new programming tools, making edge system design and optimization more demanding. This growing gap between the changing hardware capabilities and the software abstractions threatens to diminish the practical benefits of such advancements.

Alongside this transition to the edge, optimization techniques originally developed to mitigate computational and memory demands in cloud environments are also being adapted for edge platforms. These techniques, such as quantization or pruning, to name a few, are not only applicable but become even more critical in edge settings due to their inherent resource constraints. Transformer-based Large Language Model (LLM) architectures are an example of this: they are propagating from cloud to edge. However, the sequential nature of LLM inference, generating one token at a time, challenges hardware designed for parallelism. To address this, new acceleration strategies, such as speculative and parallel decoding, have emerged [1], [2], [3].

Speculative decoding, in particular, has demonstrated remarkable improvements in the generative pipeline on high-end GPUs, achieving speedups up to 3× or even 6.5× compared to traditional autoregressive decoding, depending on the specific technique employed [4], [5]. These results make speculative decoding an appealing candidate for migration from cloud environments to edge platforms, where computational resources are far more constrained. Despite the limited processing power of edge devices, this algorithmic optimization offers a substantial performance boost with minimal hardware overhead, requiring only a slight increase in transistor count.

However, this transition remains challenging. Current machine learning frameworks supporting speculative decoding are largely optimized for high-end hardware, reducing productivity for edge ML researchers and engineers. The dramatic speedups observed in server-class environments often fail to materialize at the edge due to the complex interplay between speculative decoding and other optimizations such as quantization. Furthermore, the inherent heterogeneity of edge devices introduces significant programmability challenges, contrasting with the homogeneous GPU-centric architectures of the cloud, where CPUs primarily handle control and synchronization tasks. In this work, we achieve a performant integration between heterogeneity, quantization, speculative decoding, on edge devices through a layered ML compiler, that allows the edge ML practitioner to exploit the available resources through ahead-of-time optimizations, overcoming the challenging scarceness of bandwidth, compute, and memory that characterizes edge systems.

1.1 Goal

The objective of this thesis is to demonstrate that the complexity introduced by hardware heterogeneity during the application of speculative decoding to quantized LLMs running on edge devices can be effectively leveraged through an appropriate compiler design. By employing a layered and modular compiler architecture, it is possible to abstract low-level hardware details, thereby reducing programming complexity while maintaining or improving performance. Concretely, this work aims to illustrate how an open-source ML compiler and its frontend can be used to abstract and lift low-level hardware features to enable efficient acceleration of LLM inference using speculative decoding. This research is guided by two central questions:

1. **Is it beneficial to lift hardware abstractions and adopt a monolithic representation of the generative pipeline?** This question examines whether deploying generative models in edge environments can be simplified without sacrificing performance. Specifically, it explores two strategies: 1) exposing low-level hardware features, such as heterogeneity, to the frontend, and 2) representing the entire generative pipeline in a unified graph to minimize dependencies and to facilitate global optimizations.
2. **Under what conditions do heterogeneous execution and speculative decoding yield performance improvements on edge devices?** Here, we aim to examine the effect of applying heterogenous execution to complement speculative decoding. Given the wide variability in edge hardware configurations, the application of these two ML optimization techniques (speculative decoding and ML hardware acceleration) is non-trivial. To inform design decisions and ensure a close-to-optimal performance, we

propose the use of an analytical cost model that determines the conditions under which these techniques should be applied.

1.2 Structure of the Work

This thesis begins by explaining the fundamental concepts that form the basis for the work, including heterogeneous edge computing and speculative decoding in LLM inference. We also introduce ML compilers as a means to bridge the gap between fast-paced algorithmic innovation and evolving computer architectures; these foundational topics are covered in Chapter 2. In Chapter 3, we review relevant literature to contextualize our approach. Chapter 4 details the methodology used to address the core research questions, including our evaluation strategy. In Chapter 5, we present and analyze the results of our empirical study, discussing latency differences and their implications. Finally, Chapter 6 concludes our journey by summarizing our contributions, reflects on limitations, and outlines potential directions for future research.

2 Background

In this chapter we introduce the technical background that supports the work of this thesis. First, we present the key drivers of heterogenous edge processing. It briefly compares the cloud and edge computing emphasizing their complementarity and their trade-offs. Afterwards, we shift the focus to edge devices, bringing up a particular SoC as a paradigmatic representative of heterogenous edge devices; it comprises a CPU, GPU, among other units. The chapter then introduces the speculative decoding technique, which is a key algorithm-level optimization. Afterwards, we highlight the influence that the edge hardware has on allowing an effective speculative decoding and we present key metrics supporting this. Finally, the chapter presents ML compilers and runtimes as main building blocks that connect the inference of LLMs with the hardware heterogeneity that dominates the computation on edge-grade devices.

2.1 Edge Processing and Heterogenous Compute

The convergence of edge devices and LLMs has generated renewed interest in the trade-offs between cloud and on-device inference. Pushing some Deep Neural Network (DNN) workloads that have been traditionally run in the cloud to edge devices can reduce user latency and offload recurring network and cloud costs to end devices, but at the expense of dealing with constrained device resources and increased system complexity [6]. This has been possible thanks to advances in edge hardware and integration of heterogenous Processing Unit (PU)s on a single silicon die.

2.1.1 Edge Processing

The demand for edge compute is expanding rapidly and the market is reacting, mainly driven by the increasing need for real-time responsiveness, data privacy, and operational efficiency [7]. Tasks traditionally executed in centralized cloud infrastructures are progressively migrating toward the edge, where computation occurs closer to the data source. This shift is motivated by several factors: stringent privacy requirements, the need to reduce operational and transmission costs, the limitations of network bandwidth, and the latency constraints inherent to cloud-based processing. Among the computational workloads undergoing this transition, the deployment of transformer-based LLM architectures exemplifies this trend.

Beyond the migration of existing workloads, entirely new applications are emerging at the edge, enabled by advances in model compression, quantization, and hardware-software co-design. These include privacy-preserving inference, autonomous control, and real-time responsiveness. Figure 2.1 illustrates a selection of application domains for edge devices. For instance, personal agents such as AutoDroid [8] leverage edge-deployed LLMs to automate daily routines on the phone. In healthcare, model collections like BioMistral [9] facilitate real-time diagnostics and multimodal analysis directly on-device, enhancing confidentiality. Industrial applications, including AnomalyGPT [10] for fault detection, demonstrate the viability of deploying LLMs in latency-critical environments.

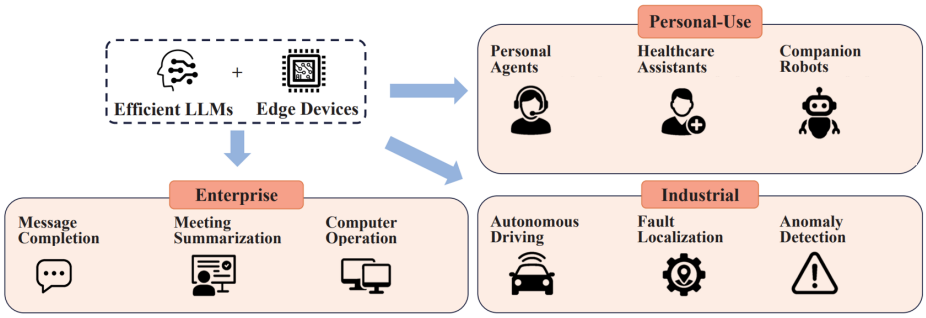

Figure 2.1: Some applications of on-device applications of LLMs. Adapted from [11].

2.1.2 SoCs as Heterogeneous Edge Devices

Edge devices are fundamentally resource-limited, an issue that exacerbates with more data and compute intensive workloads, such as LLMs. Performing LLM inference on edge evidences the limitations of this type of hardware, as it is largely memory-bound, which means that memory bandwidth and capacity often dominate the performance bottlenecks [12]. Additionally, continuous execution of LLMs remains elusive on edge devices, particularly on battery-powered ones [13], where thermal dissipation and battery draining issues become difficult to avoid.

The answer to this power challenge is the integration of DSAs in a single chip leading to heterogeneity in compute. This approach not only decreases the energy consumption by integrating specialized architectures for the efficient execution of a specific pattern (i.e., heavy computational operations in transformer architectures), it also increases programming complexity, since aspects such as data movement, scheduling, memory tiling, and synchronization across compute elements become essential [11].

The Google Tensor G Series [14], AMD Versal AI Edge Series [15], Rockchip RK3588 [16], NVIDIA Jetson Nano [17], STMicroelectronics STM32MP157 [18], and NXP i.MX95 [19], among others, exemplify the architectural heterogeneity that characterizes modern edge devices. These SoC platforms integrate diverse PUs including CPUs, GPUs, and increasingly Neural Processing Unit (NPU)s within a single package, designed to balance energy efficiency with computational performance. The CPUs provide general-purpose flexibility, however their sequential execution model limits their effectiveness for parallel workloads typical of transformer-based LLMs. To address this limitation, there is a growing trend toward augmenting CPUs with vector processing units and SIMD extensions

In contrast, GPUs offer high parallelism. However, compared to server GPUs, edge GPUs have more constrained caches, shared memory, and memory bandwidth. To achieve a high

performance, shader programs must be optimized and memory layouts (e.g., PHWC4) adopted to match GPU architecture, to improve cache locality and reduce bandwidth bottlenecks [20]. Moreover, optimizations such as operator fusion, reduced kernel invocations, tiling, efficient buffer reuse, and bandwidth-aware scheduling are essential since mobile GPU's compute units are usually underutilized by the inefficient software layering [21]

2.2 Large Language Model Inference

Transformer-based LLMs have revolutionized natural language processing by enabling high-quality generative capabilities across a wide range of tasks. Unlike traditional architectures used in computer vision, LLMs rely on attention mechanisms and autoregressive decoding, which introduce unique challenges for inference. In this section, we provide a brief overview of the inference process in transformer-based LLMs.

2.2.1 Transformer Architecture

Transformers are DNN architectures built around multi-head attention mechanism, which allows models to dynamically focus on different parts of an input sequence by computing attention scores followed by softmax normalization. Each transformer block also includes feed-forward networks, residual connections, and layer normalization [22]. This is illustrated in Figure 2.2, where transformer blocks are shown in green (a) and (c), and in blue (b) and (c).

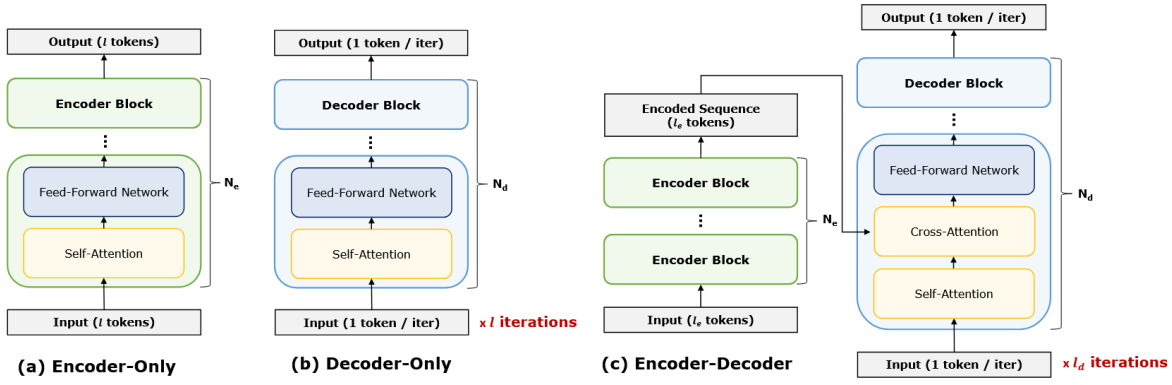

Figure 2.2: Three variants of the Transformer architecture: (a) encoder-only, (b) decoder-only, and (c) encoder-decoder. Taken from [23]

Transformers are categorized as shown in Figure 2.2. Encoder-only models (a) are used for parallel processing of input sequences in understanding task; decoder-only (b), models for sequential token generation (i.e., autoregressive generation). The encoder-decoder architecture (c) is a combination of the two previous classes. Encoder blocks scale linearly or quadratically with sequence length, while decoder blocks require iterative inference, resulting in lower arithmetic intensity and higher latency. These distinctions are discussed in more detail through this section.

2.2.2 Autoregressive Generation

Inference in LLMs is fundamentally different from discriminative tasks, particularly due to the sequential nature of token generation in decoder-only transformer architectures: they generate one token at a time, with each step conditioned on previously generated tokens. The inference workflow is depicted in Figure 2.3; it consists of two phases:

- **Prefill Phase**: The model processes the entire input prompt to initialize the Key-Value (KV) cache.
- **Decode Phase**: The model generates one token at a time, using the previously generated token (not the entire sequence) and the KV cache.

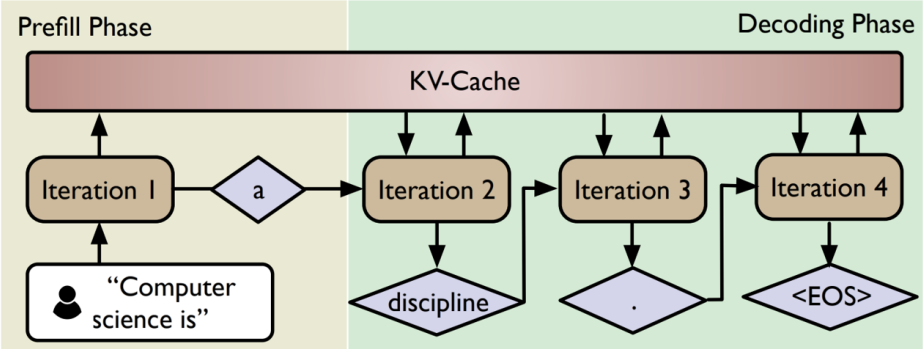

Figure 2.3: Prefill and decode phases during LLM inference. Taken from [24]

This sequential generation leads to a shape-changing workload: the input sequence length increases with each generated token. Consequently, the memory and compute requirements vary during inference, making LLM inference more complex than traditional discriminative inference [12].

2.2.3 Inference Regimes

To analyze and optimize LLM inference, it is useful to distinguish between different inference regimes based on two axes: the phase of inference (prefill vs. decode) and the input sequence length (S_L) (short vs long). These regimes exhibit distinct compute and memory characteristics [25]. Although this work focuses on the prefill phase under very short input sequence regimes, we briefly introduce the remaining execution phases below for the sake of completeness.

Compute-Boundness vs Memory-Boundness *Compute-bound operations* are constrained by the hardware's peak floating-point throughput and benefit primarily from algorithmic optimizations that reduce arithmetic intensity. In contrast, *memory-bound operations* are limited by available memory bandwidth and are better accelerated through techniques such as quantization and operator fusion. The *roofline model* provides a framework to classify operations into these categories by comparing their arithmetic intensity (measured in FLOPs per byte) against the target hardware's ridge point, thereby guiding optimization strategies [26].

Prefill vs Decode Phases During the prefill phase, computation is dominated by matrix-matrix multiplications, which exhibit high arithmetic intensity and benefit from batching and parallelism. In contrast, the decode phase primarily involves matrix-vector multiplications with lower arithmetic intensity, making it more susceptible to memory bottlenecks. This phase also requires frequent access to and updates of the KV cache, further emphasizing its memory-bound nature [25].

Short vs Long Sequence Lengths In transformer-based models, the dimensionality of the hidden states, denoted as d, plays a central role in determining computational characteristics. Sequence lengths (S_L) are typically categorized relative to this dimension: short sequences are those where S_L is shorter than the hidden dimension (i.e., $S_L \ll d$), while long sequences significantly exceed it ($S_L \gg d$). This distinction is crucial for understanding performance bottlenecks and it is summarized in Table 2.1 for the prefill phase.

Table 2.1: Impact of sequence length (S_L) relative to hidden dimension (d) on dominant operations and runtime scaling during the prefill phase [23].

Case	Sequence Length Relative to d	Runtime Behavior and Dominant Layers
1	$S_L \ll d$ (Short)	Linear layers dominate; scales $\mathcal{O}(S_L)$
2	$S_L \approx d$	Mixed dominance; scales $\mathcal{O}(S_L + S_L^2)$
3	$S_L \gg d$ (Long)	Attention dominates; scales $\mathcal{O}(S_L^2)$

2.3 Speculative Decoding as a Token Generation Optimization Technique

Considering that the computational and memory demands of LLMs often exceed the capabilities of edge hardware, a number of optimizations are required to make LLM inference feasible in such environments. Key techniques can be categorized into three levels [27]:

- **Algorithm-level optimizations**: Quantization, pruning, distillation, sparse or conditional computation reduce compute and memory footprint.
- **Hardware-level optimizations**: Some techniques include: designing or leveraging specialized accelerators (e.g., NPUs), optimizing memory bandwidth, cache hierarchies awareness, exploiting parallelism, mixed precision arithmetic, and using near-memory computing.
- **Software and system-level optimizations**: Efficient operator kernels, LLM-oriented scheduling, memory layout optimizations, batching strategies, pipelining, runtime adaptation of models (e.g., dynamically choosing kernels depending on the request).

Although all these techniques are suitable for DNNs in general, a persistent bottleneck remains in the token generation process in the autoregressive decoding of LLM inference. Each token depends on previous tokens, leading to largely serial execution which limits latency improvements. This is where speculative decoding emerges as a promising technique at the algorithm-level to mitigate the sequential dependency.

2.3.1 Speculative Decoding Overview

Broadly, speculative decoding works by having a faster, usually smaller, drafter or speculator mechanism which predicts one or more future tokens (drafts), under the assumption that many of these predictions will be obvious or correct. Then a slower, more precise target model, also called verifier, checks in parallel whether those drafted tokens are acceptable according to some criterion. The target model receives that name since it is the model that would be used in a non-speculative setting and is aimed to be accelerated [28].

Figure 2.4 exemplifies the speculative decoding pipeline. At the top, the Incremental Decoding Timeline depicts the standard sequential execution of the decode phase without speculative techniques. The middle and bottom timelines present two speculative decoding strategies: sequential drafting and tree-based drafting. In both cases, the orange segments represent the speculative generation performed by a lightweight model, while the green blocks denote the validation phase carried out by the target model, which confirms or rejects the drafted tokens.

Key design dimensions for selecting an appropriate drafting mechanism include:

- **Drafting method**: Different techniques can be used as a computational-light way to generate tokens and in that way to assist on the token generation. For example, smaller independent models can be used [4], or models that share features from the target model [5]. Additionally, look-up tables and N-grams can be used since they excel in some generative tasks as summarization [28].
- **Draft length (γ)**: How many tokens ahead to predict. Larger γ offers more potential acceleration gains but risks more mismatches (thus, wasting computation). This is represented by the small orange squares bellow each drafting phase in Figure 2.4, where the value of γ is two for the sequence-based speculative decoding.
- **Training and alignment of the drafting method**: Draft models may be trained (or adapted) to better align with target model behavior (i.e., to match the target's output distribution); some techniques use multi-layer feature fusion or training-time testing to improve draft accuracy.
- **Shape of the draft**: Some techniques generate trees of candidate sequences (beams), adjusting the draft width depending on how well the draft aligns with target's distribution [28]. Other methods generate a single sequence instead.

The evaluation phase verifies the draft's proposed tokens, ensures correctness, and handles mismatches. Its main aspects are:

- **Acceptance criteria and fallback**: Depending on the desired quality, acceptance of drafted tokens might require either an exact match or looser criteria (e.g., the draft's token falls within top-k of the target's prediction). If many mismatches occur, a fallback to regular autoregressive decoding can be configured. This is quantified using the acceptance rate (α), which is the average of the number of accepted drafted tokens divided into the draft length (γ). Usually, γ is balanced against (α) [4].
- **Overlap and mapping**: Some speculative decoding algorithms [29] overlap drafting and verification to reduce idle time of the validation and drafting phases. For example, CPUs may generate drafts while GPUs verify concurrently prior speculations [29]. In

contrast, the examples in the middle and in the bottom of Figure 2.4 represent sequential speculative decoding pipelines.

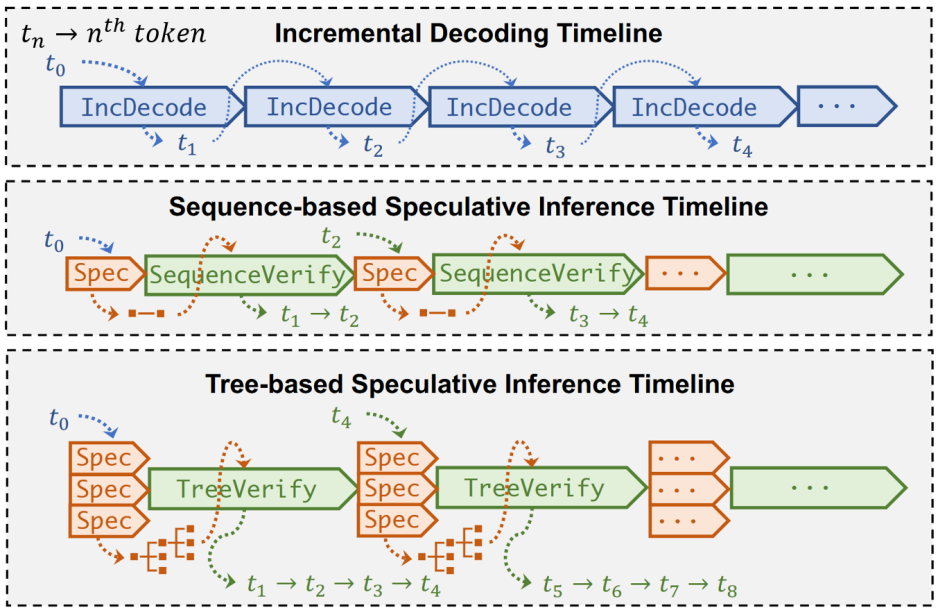

Figure 2.4: Three generative pipelines: standard sampling (top, in blue), and two variants of speculative decoding: sequential drafting (center) and tree-based drafting (bottom). In both speculative approaches, the drafting phase is highlighted in orange, while the target model responsible for validating the generated drafts is depicted in green. Adapted from [30].

With α and γ defined, we can compute the expected number of tokens produced in a single generative step. This step consists of a drafting phase, which generates a draft sequence of length γ, followed by a validation phase. We assume that the number of produced tokens follows a capped geometric distribution, with a cap of $\gamma + 1$ and a success probability of $1 - \alpha$ [4]. Under these assumptions, the expected number of generated tokens per generation step is given by Equation 2.1.

$$E(\#\text{ generated tokens}) = \frac{1 - \alpha^{\gamma+1}}{1 - \alpha} \tag{2.1}$$

2.3.2 Dependence on Hardware and Software Configurations

While in speculative decoding the drafting and evaluation phases are critical for achieving an acceleration, the actual speedup depends strongly on the workload, the task, and the hardware-software environment on which the speculative decoding is executed. For instance, factors like the available memory, the memory bandwidth, and cache sizes can be used statically during model design or compilation to select the sizes of the depth and the width of a tree-based draft [31]. In devices with limited memory, the verification phase may require storing intermediate states, activations, or drafts, which may not be feasible. To mitigate this, the drafting phase can be interrupted as soon as some condition is met [32].

A formalism modeling the hardware impact on the speculative decoding speedup is developed in [4]. It presents the speedup as a function of the acceptance rate's expected value (α), the draft length (γ), and the cost coefficient (c) which is a parameter that depends on the

hardware and software configuration. c is calculated as the ratio of the duration of one draft decoding step to that of the target model's duration. The speedup S is given by Equation 2.2.

$$S(\alpha, \gamma, c) = \frac{1 - \alpha^{\gamma+1}}{(1 - \alpha)(\gamma c + 1)} \tag{2.2}$$

Ideally, the cost of introducing speculative decoding should be negligible ($c \to 0$), leading to a speedup that is only function of the quality of the draft (i.e., how much is the average acceptance rate of the speculated tokens α) and the draft length (γ). However, in practice, c is non-negligible and can significantly affect the speedup. For instance, if the draft model is not sufficiently faster than the target model or if the acceptance rate α is low, the speedup may be minimal or below one.

We may also note that the condition $c < \alpha$ must hold to achieve any speedup at all. This guarantees that there is at least one value of γ that yields a speedup greater than 1. Furthermore, the optimal draft length γ^* that maximizes the speedup depends on both the hardware configuration and the quality of the drafting mechanism.

2.4 Machine Learning Compilers

The slowdown of Moore's Law and Dennard scaling pushed the development of DSAs and exposed the limitations of general-purpose processors in handling increasingly complex ML workloads. While frameworks and libraries such as TensorFlow [33], PyTorch [34], and ONNX [35] have facilitated model deployment, they often rely on vendor-specific implementations that tightly couple software to particular hardware platforms [36]. This fragmentation leads to significant engineering overhead when porting models across diverse devices. To mitigate this challenge, the emergence of ML compilers marks a paradigm shift toward more generalizable and scalable deployment solutions. These compilers accept models from various ML frameworks, abstract their representations, and enable systematic optimization and retargeting across heterogeneous hardware.

2.4.1 Domain-Specific Compilers

ML is a paradigmatic case of a domain-specific application that has encouraged the development of its own Domain-Specific Language (DSL), such as the one used in PyTorch and compiled through TorchDynamo [34]. DSLs capture the abstractions and computational structures required by a specific domain, enabling more efficient optimization and compilation strategies compared to general-purpose languages. The tight coupling between ML workloads and DSLs motivates the design of domain-specific compilers [37], since conventional compiler infrastructures often fail to capture the semantics and optimization opportunities inherent in ML programs. DSLs allow higher level abstractions, while domain-specific compilers bridge this expressiveness with the performance needs of the underlying hardware. Nevertheless, ML systems face challenges that are not trivial from a compilation perspective: the frameworks often rely on fragmented "glue code", opaque kernels, and highly specialized operators that hinder systematic optimization [38], [39], [40].

Several approaches seek to reduce these challenges. Outside the ML field, Julia's design emphasizes enabling users to define high-performance numerical operators that are not opaque: user-defined operations are first-class, and the compiler can introspect, optimize, and compose [41]. Similarly, the newest version of Torch moves toward transforming Python bytecode into graph Intermediate Representation (IR)s that can be optimized, fused, and lowered, reducing the burden of custom operator implementation [34].

2.4.2 Multi-Level Intermediate Representation Approach for Compiler Construction

Multi-Level Intermediate Representation (MLIR) provides an infrastructure for compiler construction by supporting different levels of abstraction in a unified and modular framework, thereby facilitating the design of domain-specific compilers [42]. This approach promotes explicit representation of operations (MLIR's fundamental units of computation that consume and produce typed values), which enhances opportunities for optimization techniques such as operator fusion. Furthermore, it enables the definition of new DSL dialects, which are modular groupings of operations, types, and attributes tailored to specific domains (e.g., linear algebra), representations (e.g., control flow), or hardware targets (e.g., custom hardware).

MLIR's extensible transformation framework allows compiler passes to manipulate operations generically using traits and interfaces, enabling optimizations while preserving correctness. Through a process called lowering, high-level abstractions can be progressively transformed into lower-level dialects, bridging the gap between expressive domain-specific semantics and efficient hardware execution. In the context of ML compilation, MLIR allows for a clear separation across the compilation stack ranging from high-level tensor semantics to mid-level structured loop representations (e.g., affine dialects), down to low-level hardware-specific streaming semantics which expresses asynchronicity, parallelism, and temporal behavior in a dataflow-oriented execution model. This stratified design significantly improves the portability and efficiency of model lowering across diverse hardware backends.

2.4.3 IREE as a Compiler for Machine Learning

IREE [43] is a compiler and runtime project that builds directly on the concepts of MLIR, providing a concrete instantiation of multi-level compilation for ML workloads. The IREE ecosystem is presented in Figure 2.5. It ingests a program from a library or a DSLs (e.g., PyTorch backed by TorchDynamo) (Fig. 2.5 a.) through the importers. For PyTorch, the integration is handled by IREE Turbine, which exports models into the MLIR-based `Torch` dialect. ONNX models are first converted to MLIR using the `iree-import-onnx` tool, which produces the same `Torch` dialect. This design ensures that both PyTorch and ONNX models follow a common compilation path and reduces maintenance efforts. The compilation process in IREE initiates with three phases (Fig. 2.5 b.): the input phase, which lowers frontend dialects like `Torch` into core MLIR dialects such as `linalg`; the ABI phase, which adapts the program's interface to the target execution environment (explained later) aided by IREE's `util` dialect; and the preprocessing phase, where tensor encodings are applied to enable data tiling and layout optimizations. This preprocessing enables hardware-specific transformations in later stages. Next, the global optimization phase performs high-level transformations such as

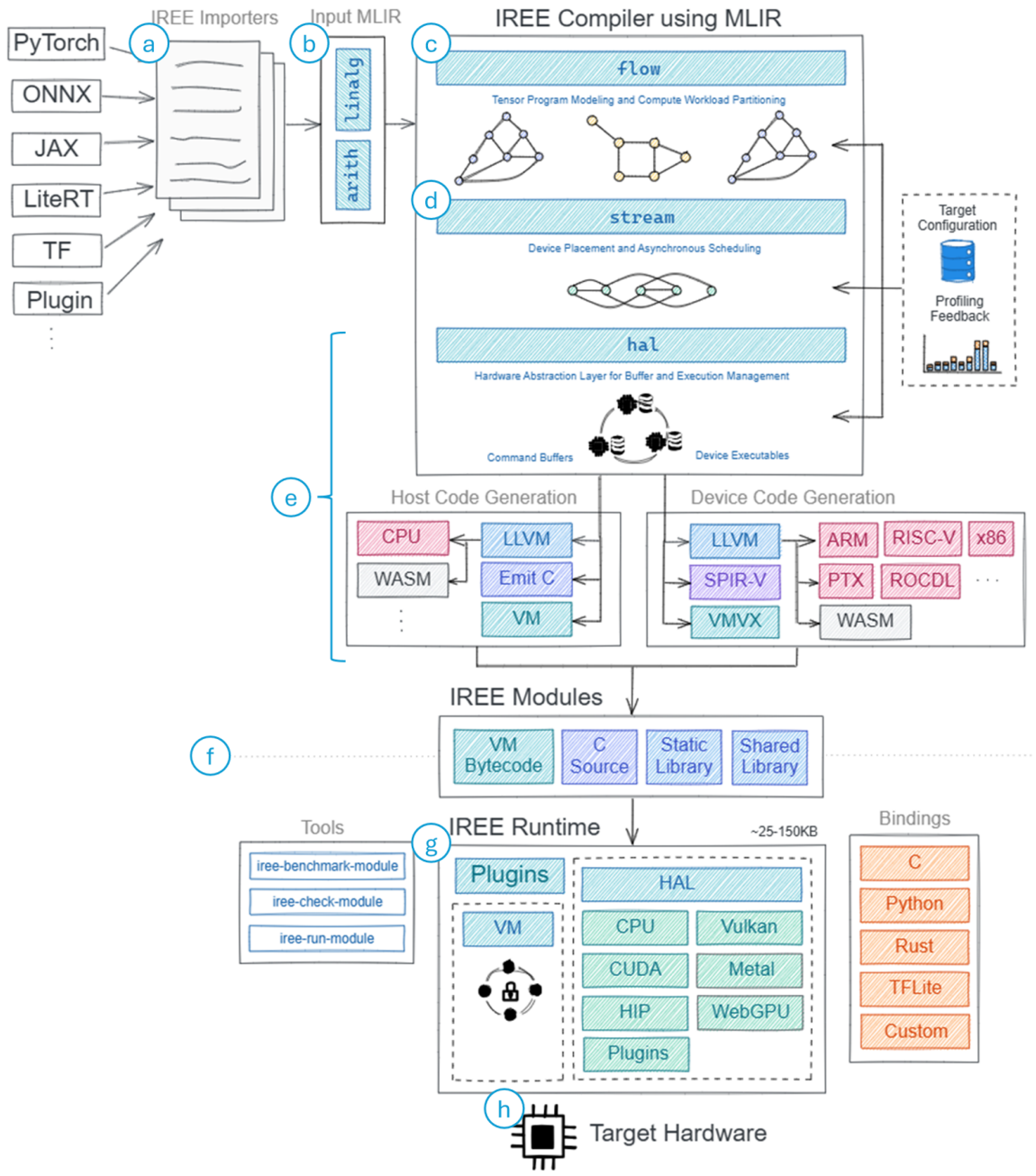

Figure 2.5: Ecosystem of Intermediate Representation Execution Environment (IREE) compiler and runtime highlighting its ML framework interoperability (top left) and its multitargeting capabilities (bottom). Modified from [43].

transpose propagation. Furthermore, it partitions the computation into dispatchable regions using integrating `flow` operations, preparing the module for efficient scheduling in further stages. The output at this point includes core MLIR dialects (`linalg`, `arith`, `scf`, `tensor`) and IREE's `util` dialect.

From now on, the program enters the middle-end phase and start diverging from tensor-semantics: first, the `flow` dialect (Fig. 2.5 c.) aims to the high-level dataflow and computation partitioning. It identifies dispatch regions and outlines them into executable units; this is a key for this thesis since it also allows the compiler to reason about which dispatch regions belong to which logical device, facilitating heterogeneity in downstream lowering. Next, the

`stream` (Fig. 2.5 d.) dialect intrudes a resource-centric, asynchronous execution model, and streaming semantics. Here, explicit lifetimes for resources are defined.

The `stream` and `HAL` dialects (Fig. 2.5 e.) adopt a unified abstraction for parallel execution that treats CPUs, GPUs, and other accelerators under a GPU-inspired scheduling model [43]. This design decision stems from the observation that ML workloads are inherently parallel and can benefit from the thread-workgroup-subgroup hierarchy familiar from GPU programming. The programming model adopted by IREE draws inspiration from Vulkan and OpenCL, offering mechanisms to partition workloads into parallel units, synchronize execution, and manage memory hierarchies. However, unlike Vulkan, where expert programmers must explicitly write low-level code, IREE automatically generates this code. This preserves high-level expressiveness for ML while still delivering fine-grained, device-specific performance. Between these two dialects (`stream` and `HAL`), IREE prepares for code generation through three phases: executable sources, executable configurations, and executable targets (Fig. 2.5 e.). In these stages, dispatches are translated and operations are materialized to target-specific dialects (e.g., LLVM (CPU), SPIR-V (Vulkan), PTX (CUDA)) demonstrating IREE's multitargeting capabilities; host-side code is lowered to `VM` dialect, which supports generic operations and enables portable execution. The final output is encapsulated in bytecode files, containing both host VM instructions and embedded device binaries (Fig. 2.5 f.).

2.4.4 Runtime for Heterogeneous Devices

While infrastructures such as IREE address many issues at the compilation stage, the runtime phase introduces additional challenges, particularly in the context of execution of ML workloads on heterogeneous and edge devices. The compilation phase alone cannot guarantee performance portability: runtime systems must also account for heterogeneity, communication costs, and dynamic workload behavior. This is especially relevant for edge and embedded devices, which are increasingly built on multi-core [44] and heterogeneous architectures, combining CPUs, GPUs, DSPs, NPUs and other specialized accelerators [39].

In this context, the role of the compiler is rethought in IREE. Here, the compiler is no longer seen as a purely offline tool. Instead, it interacts closely with the runtime to handle device-specific constraints and dynamic resource availability, without loosing modularity (Fig. 2.5 g.). Mechanisms such as IREE's Hardware Abstraction Layer (HAL) and the "Bring Your Own Codegen" principle in ML compilers [45] embody this approach by providing a modular interface where heterogeneous code generation and runtime execution can be reconciled, while abstracting away hardware-specific details (Fig. 2.5 h.).

Beyond heterogeneity, another key runtime challenge is the increasing dynamism in ML workloads. Many emerging ML models (e.g., those employing speculative decoding in LLMs) exhibit control flow that depends on data (e.g., early exits, recursion, conditional paths), varying input sizes, and dynamic architectures. This raises limitations for purely Ahead-of-Time (AOT) compilation, as well as for traditional Just-in-Time (JIT) techniques, particularly in resource-constrained embedded or edge environments. Using multiple precompiled kernels, and then using a scheduler to select them at runtime is an effort to address this and can balance the resource allocation accordingly [46].

3 Related Work

We can broadly define the related work under three main buckets: Firstly, other DNN optimization methods at the algorithmic level which tend to be orthogonal to our work. Secondly, the different methodologies that allow code generation and mapping of ML workloads onto many-core and heterogenous architectures to achieve high performance. The automatic techniques used in the deployment of such workloads on heterogeneous hardware constitute the third category. While each category addresses distinct challenges, they intersect in meaningful ways with the goals of this thesis.

3.1 Neural Network Optimization Methods

Deploying LLMs or even smaller neural networks on edge devices faces constraints in compute capacity, memory, energy, and thermal budget. Therefore, optimizing across multiple dimensions is essential to make inference feasible, efficient, and with acceptable latency and power consumption. Recent surveys categorize optimizations into three complementary classes: algorithmic methods, such as quantization, pruning, low-rank approximation, and knowledge distillation; hardware-specific optimizations, where accelerators like NPUs target both latency and energy efficiency [47], and mobile GPUs are exploited through tailored kernels [20]; and software techniques, such as ML compilers.

These approaches act orthogonally to the work of this thesis, enabling them to be combined, in principle; although, their effects are not always additive. For instance, while quantization reduces memory bandwidth demands and memory footprint, speculative decoding adds computational overhead (i.e., drafting and verification) which may offset, or reduce, the benefits coming from quantization. Therefore, a hierarchical framework is introduced in [48] to balance the trade-off between the drafting overhead and memory savings of quantization.

Additionally, some other works focus in the intersection of speculative decoding and hardware optimizations. Sequoia [31] adapts the generation of the draft tree to match hardware characteristics. DuoDecoding [49] explicitly schedules speculative decoding tasks across heterogeneous units to minimize the idle times. Similarly, Dovetail [50] splits speculative decoding work across CPU and GPU: some parts run on CPU, others on GPU. However, these

works do not consider heterogeneity on edge devices, focusing rather on high-end systems. In contrast, in this thesis we introduce heterogeneity on the level of a monolithic edge device.

3.2 Parallelism on Heterogenous Hardware

Mapping neural and non-neural workloads onto heterogeneous hardware has been a central theme in systems and accelerator research. The core challenge (how to partition computation and place work across devices with different instruction sets, memory hierarchies and peak throughput) has been addressed by a wide variety of methods, including polyhedral compilation, design-space exploration, combinatorial optimization, and runtime/adaptive scheduling. HeteroLLM [51] and Herald [52] extend these ideas to edge SoCs and to multi-DNN workloads, respectively. Tiramisu [53] and ZigZag [54] focus on space-exploration approaches for generating high performance parallel code using constrained optimization, analytical, or automatic approaches [55]. We build on these ideas, but focus on the unique challenges of speculative decoding, which introduces new dependencies and trade-offs not present in standard DNN inference.

Moreover, a challenge arises from the fact that heterogeneous platforms often integrate high-throughput accelerators (e.g., GPUs, NPUs) with general-purpose CPUs: severe load imbalance emerges when naive partitioning strategies are applied, since there is large disparity in the performance between these PUs. FlexInfer [56] explicitly characterizes this problem in the context of LLM inference and mitigates it by dynamically selecting the best mapping for each execution phase using both static hardware configuration and live runtime measurements. We translate these ideas into the context of edge devices, where AOT compilation is preferred over runtime scheduling due to the limited resources.

Additionally, multi-tenancy (i.e., placing multiple, concurrently active DNNs onto a shared heterogeneous platform) has been a study topic. MAGMA [57] formulates multi DNN mapping as an optimization problem and applies a genetic algorithm to identify near-optimal allocations of models to accelerator cores. Similarly, Adyna [46] addresses the scheduling of multiple dynamic DNN in spatial accelerators by formulating adaptive scheduling policies of different granularity. Both works highlight that multi-tenancy requires global reasoning about contention (compute and memory), model heterogeneity, and temporal interference, being pivotal to this thesis as well, where the the execution of the draft and the target model can be seen as a multi-tenant scenario. Nevertheless, MAGMA and Adyna evaluate against accelerator models or simulators rather than on silicon, as done in this work.

3.3 Machine Learning Compilers

Modern ML compilers exhibit different design decisions that influence their suitability for heterogeneous edge SoCs and speculative-decoding workloads. ONNX, primarily designed as an interchange format for ML models, prioritizes portability and operator coverage over compiler-oriented transformations. Its interpreter-style execution limits visibility of the full computation graph, hindering optimizations such as operator fusion, and its specification remains more aligned with ML representation than code generation needs. ExecuTorch [58] in

contrast, adopts a layered compilation architecture progressively lowering PyTorch models through structured abstractions while performing ahead-of-time memory planning.

Glow, one of the earliest end-to-end ML compilers, pioneered a two-phase intermediate representation (domain-level and low-level IRs) that enables aggressive graph lowering, static memory allocation, and backend-agnostic optimization [59]. However, its active development declined following the adoption of MLIR-based infrastructures that generalized its concepts. Finally, another early approach was TVM [60]. Influenced by Halide [61], TVM separates algorithm definition from scheduling, allowing autotuning and multi-target optimization across CPUs, GPUs, and accelerators.

Collectively, these systems illustrate the evolution from interpreted ML-specific runtimes toward progressively lowered, compiler-oriented stacks. While Executorch is a modern ML compiler system that adopts the modular multi-IR approach, it excludes high-level ML frameworks such as TensorFlow and ONNX. In contrast, our work build upon IREE, which supports interoperability across different high level ML frontends within a single IR [42]. This interoperability is crucial for enabling seamless integration of diverse models and workflows especially in heterogeneous environments where different tools rely on different ML ecosystems.

3.4 Thesis Contributions

This thesis distinguishes itself by being the first to power **speculative decoding on edge devices through heterogeneous partitioning**, a capability not addressed by prior works which either focus on high-end systems or lack multitarget deployment. Unlike existing approaches that rely on simulators, restrict themselves to homogeneous platforms or to commodity hardware, our method is evaluated on real silicon and leverages edge heterogeneity. This is allowed by the use a modern ML compiler and execution infrastructure: IREE, that uniquely supports multi-target code generation and interoperability across diverse ML frontends within a unified and layered intermediate representation, which is critical in edge scenarios where software stacks are fragmented and hardware diversity is high. Moreover, IREE's AOT compilation, backend-agnostic design, and extensibility make it particularly well-suited for existing and emerging edge devices, enabling the efficient deployment of speculative decoding on heterogenous platforms without vendor lock-in frameworks. To better contextualize this thesis' contributions, the Table 3.1 provides a comparative overview of representative works across these three dimensions, highlighting how our approach complements and extends existing efforts.

Table 3.1: Comparison of related works across key dimensions relevant to this thesis. reflects two important aspects: first, the deployment is not restricted to a single hardware vendor, ensuring broader applicability; and second, the evaluation is conducted on actual hardware platforms rather than relying solely on simulations or theoretical models. Spec. dec. stands for speculative decoding; quant., for quantization; het. for heterogenous; hier., for hierarchical.

Work	Spec. Dec. on Edge	Quant.-Aware Spec. Dec.	Het. Partitioning	Based on ML Compiler
Hier. Quant. [48]	✓	✓	✗	✗
Sequoia [31]	✗	✗	✗	✗
DuoDecoding [49]	✗	✗	✓	✗
Dovetail [50]	✗	✗	✓	✗
HeteroLLM [51]	✗	✗	✓	✓
Herald [52]	✗	✗	✓	✓
Tiramisu [53]	✗	✗	✗	✗
ZigZag [54]	✗	✗	✓	✓
FlexInfer [56]	✗	✗	✓	✓
MAGMA [57]	✗	✗	✓	✗
Adyna [46]	✗	✗	✓	✗
This Thesis	✓	✓	✓	✓

4 Methods and Implementation

This chapter outlines the methodology and implementation for integrating speculative decoding into ML compilation workflows. It begins by detailing how abstractions can be aligned with hardware-aware execution to preserve productivity while enabling a good performance. The second section presents a systematic approach to distribute LLMs across heterogeneous and unbalanced PUs, formulating the problem as a design space exploration and optimization task. Together, these sections provide a foundation for deploying efficient, generative pipelines on edge systems.

4.1 Expressing Speculative Decoding Pipelines Ahead of Time

The problem formulated in this section has two interrelated facets: 1) *performance*: the integration of an autoregressive generative pipeline with speculative decoding into a compilation flow without sacrificing performance with respect to a baseline implementation, and 2) *productivity*: introduce a generative pipeline with speculative decoding at the right level of abstraction, one that maximizes ease of interaction and of definition/expressiveness. The interplay of these two aspects reflects the longstanding trade-off between performance and productivity. In the context of heterogeneous partitioning, this balance becomes delicate: the more abstract and flexible the system, the greater the potential for performance overheads. The key question, therefore, is how much performance degradation can be tolerated in exchange for the benefits of modularity, composability, and productivity.

4.1.1 Matching Abstractions of Speculative Decoding and Heterogenous Execution

Selecting an appropriate abstraction level for declaring the spatial partitioning turns into an ambivalence. On one hand, speculative decoding techniques are more naturally and clearly articulated in the higher abstraction layers, such as within ML libraries or DSLs. On the other hand, heterogeneous partitioning operates inherently closer to hardware, where fine-grained control over the computational and communication resources is essential. To reconcile this tension, we propose to rise the heterogenous partitioning to the compiler's frontend,

integrating it directly with the speculative decoding construct. This decision is informed by observations in the literature (see Chapter 2) indicating that the ratio between the latencies of the speculative phase and the evaluation phase has a direct impact on the performance of speculative decoding techniques. Concretely, when the speculation phase execution time is not negligible, it becomes a bottleneck, making it a compelling optimization target.

By expressing the hardware-aware partitioning at the same level of abstraction of the speculative decoding logic, we preserve the flexibility and productivity gains of PyTorch level programming, while enabling targeted acceleration strategies. For instance, mapping the performance-critical speculative phase to an accelerated PU. This coarse-grained partitioning/scheduling approach is performed at the boundaries of the speculative phase and the evaluation phase, that is, spatially separating the large model from the small. Although a coarse-grained partition is also possible at the middle-end level of the ML compiler (e.g., `flow` and `stream` dialects in IREE), its benefits are diminished given increased complications: at this level, the interface between the speculative and evaluation phases are less obvious, necessitating a pattern matching mechanism to distinguish the two phases at the operation level. Such mechanisms are difficult to generalize to different model architectures or quantized variants, undermining scalability.

In contrast, the implementation of fine-granularity partitioning is more naturally implemented at deeper stages of the ML compilation pipeline [62], [63], [64]. This level of abstraction is better-suited for other type of optimizations such as the offloading of computationally intensive operations to specialized fixed-function accelerator units, including Multiply-Accumulate (MAC) arrays. Figure 4.1 shows the structure of a compiler framework that follows this principle: at the top level, the frontend performs graph-level transformations, while the backend focuses on finer-grained, tensor-level transformations.

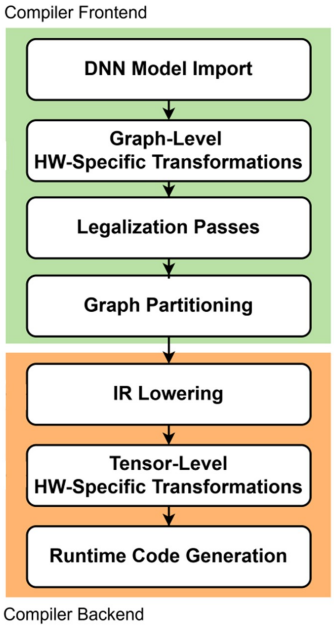

Figure 4.1: An example of a compiler framework demonstrating partitioning at different granularities: graph-level in the frontend and tensor-level in the backend. Adapted from [64].

The coarse-grained partitioning we propose in no way precludes the possibility of a fine-granularity partitioning. For example, one could assign the token generation phase (also referred to as speculative phase) to a general-purpose PU A, and the second phase assigned

to another general-purpose processor, device B. Furthermore, data-intensive operations could be offloaded from both devices, A and B, to an accelerator C. However, communication costs increase and the advantage of offloading heavy operations to an accelerated unit must be considered more carefully to avoid degraded performance.

Inversely, the performance impact due to communications using a coarse-grained is smaller. The number of data exchanges between PUs is lower and it also requires less bandwidth. For example, in a autoregressive pipeline, where the token generation phase is mapped to a different PU than the evaluation phase, only two communication events are necessary per generation step: from the evaluation of the previous generative step to the current sequence generation phase, and from this to the next evaluation. Whereas a finer granularity requires several information exchange and synchronization events between the PUs in a single forward pass, significantly increasing the communication complexity. This overhead can become prohibitive if factors such as the number of memory banks, memory hierarchy, and relative speeds between memories and PUs are not carefully accounted for.

4.1.2 Implementation of a Monolithic Graph for Edge Deployment

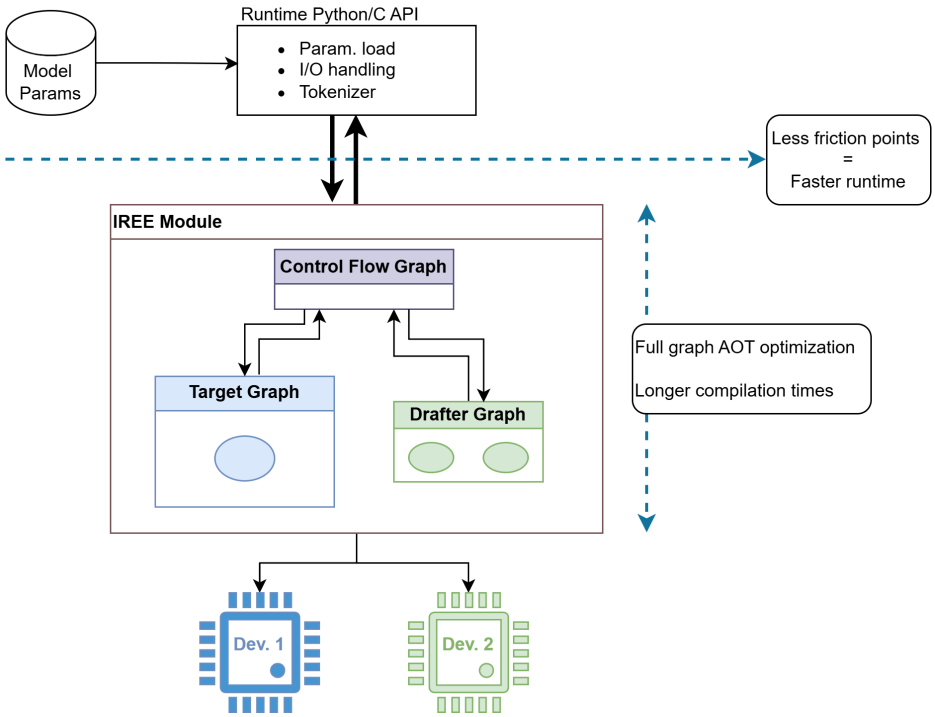

Figure 4.2: A single compiled graph (IREE module) contains the subgraph of the target model, of the drafter model, and the control flow graph that controls the generative pipeline interacting with the other two subgraphs. Within the same compiled IREE module, the device affiances are defined allowing an heterogenous execution.

IREE's capability (backed by MLIR) to integrate multiple dialect operations in a single program enables us to blend abstractions from different domains. This makes it possible to lift streaming semantics (like device affinities), together to high level tensor semantics. By lifting these semantics, an edge ML practitioner can manually (or even automatically) encode spatial partitioning without needing to manage device-specific details. This abstraction simplifies the design and improves portability between edge systems. In practice, IREE facilitates the injection of custom operators at the Torch graph level, making it possible to introduce low-level

constructs (e.g., device-level data-flow) into higher-level representations. This enables spatial partitioning to be expressed directly at the Torch module level. Operating at this level of abstraction, obviates the necessity of specifying the physical source and destination devices. Instead, it suffices to define monikers representing logical devices. These monikers are resolved to actual physical devices during compilation, allowing monikers to be mapped either to a single physical device or distributed across multiple devices, depending on the execution context and optimization strategy.

It is also worth noting that the procedural logic of speculative decoding (i.e., the algorithmic flow between drafting and verification phases, the acceptance logic, and the generative loop) is implemented within the same graph that contains the model arithmetic, as illustrated in Figure 4.2. The model's arithmetic components are depicted in blue and green, while the procedural logic is shown in grey; all encapsulated within the same IREE module. This monolithic representation of the entire generative pipeline has several advantages in the context of edge devices. For example, it is possible to perform end-to-end optimization across the entire pipeline. Similarly, this approach also minimizes the need of intermediary "glue code", which often introduces rigid boundaries between software modules and impedes global optimization. Furthermore, modularity is maintained, since each subcomponent of the pipeline can be encapsulated in a distinct subgraph, allowing independent replacement or refinement. For example, the speculation method could be changed in a modular manner without affecting the surrounding system. Moreover, deployment to hardware targets is streamlined, as the entire pipeline is represented as a single, self-contained artifact with minimal external dependencies, simplifying integration and reducing complexity. At the top of the same Figure, a minimalistic serving mechanism based on IREE runtime is depicted. It is responsible for loading the model parameters, handling user requests, generating responses, performing tokenization, and executing the IREE module

However, implementing the entire pipeline in an AOT and monolithic manner introduces notable challenges. Firstly, an increase in implementation complexity is expected, since partial rewrites of the code are required to ensure compatibility with graph capture mechanisms such as tracing. This issue is particularly critical for procedural logic involving control flow depending on dynamic data, where the number of possible execution paths can grow combinatorially. Such variability inflates the size of the computational graph and, in turn, increases the time required for global optimization and compilation.

This limitation can be mitigated by transitioning from the monolithic approach shown in Figure 4.2 to a more modular design, as illustrated in Figure 4.3. In this setup, the previously monolithic module is split into two separate components, each containing only the model arithmetic for the target and drafter models. This enhances modularity, as changes performed to, for example, the drafting mechanism would require recompiling only the corresponding module rather than the entire previously presented monolithic graph, resulting in faster recompilation times. Additionally, compilation time is further accelerated because the procedural logic is not compiled into the modules but is instead implemented within the serving mechanism, which operates on top of the IREE runtime, as shown at the top of the figure. However, as previously noted, this introduces a rigid separation between the model arithmetic and the procedural logic, evidenced by the increased number of thick black arrows connecting the serving mechanism (top) to the IREE modules (bottom). On the positive side,

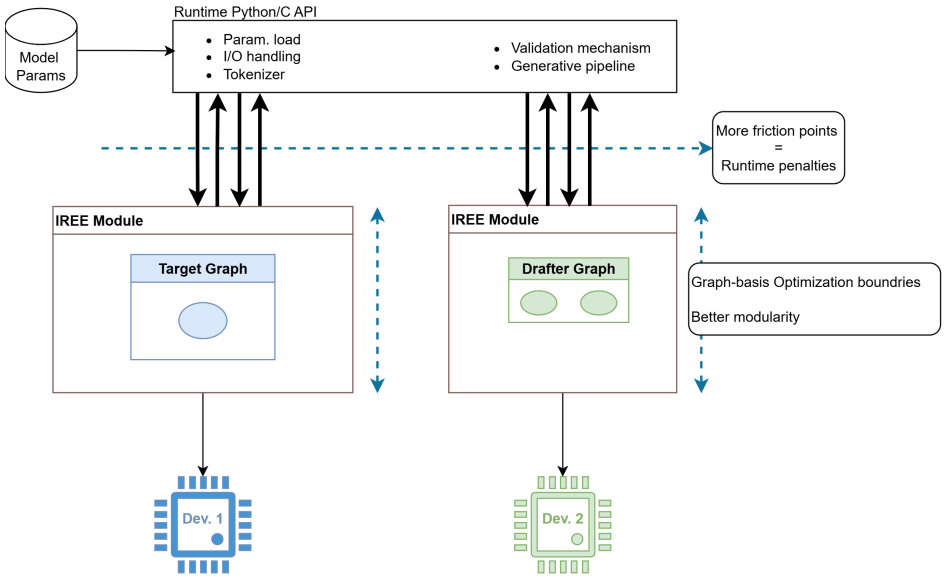

Figure 4.3: Each compiled graph (IREE module) contains only the model arithmetic and is assigned to a separate device. The control flow graph that orchestrates the generative pipeline is implemented within the serving platform, which is build on top of IREE runtime.

heterogeneous execution is simplified, as it is now distributed across two distinct compiled IREE modules.

4.2 Distributing Large Language Models across Unbalanced Processing Units

Another challenge we address is the mapping of a computational graph representing an LLM with speculative sampling and the assignment of its subgraphs to different PUs within a heterogenous system, specifically on an SoC. We focus on the prefill phase and short sequence lengths (S_L) of the generative pipeline. This is formulated as an optimization problem, for which we define the search space encoding, the search strategy, and the evaluation methodology.

Although the focus of the work is on the prefill phase, the method can be extended to other execution regimes such as the decode phase and to large sequence lengths. The prefill phase, which is the dominant contributor in Time to First Token (TTFT), is a critical determinant of responsiveness in real-time and edge deployments of LLMs. TTFT optimization directly impacts not only user-perceived latency but also total system performance [65]. This is especially important in applications like online translation, where low TTFT ensures fluid and near-instant feedback.

4.2.1 Evaluation Hardware

The NXP i.MX95 SoC is used as a hardware platform to evaluate the method presented in this section. This SoC is an example of an edge-targeted applications processor integrating multiple compute engines designed for general compute, graphics, and neural acceleration [19]. This heterogeneous device contains six ARM Cortex-A55 cores to handle generic applications, control logic and orchestration of tasks; a single-shader-core Mali-G310 GPU intended for

graphics processing but also usable for data-parallel compute and ML workloads. Therefore, our focus is on the CPU and GPU components of this SoC, which allow the execution of entire LLM architectures. These two devices are described in more detail below:

- **Hexacore Cortex-A55 CPU**: This is an energy-efficient general-purpose PU. The Cortex-A55 supports typical ARMv8 features (e.g., 64-bit addressing, NEON SIMD, and INT8 support) [66]. Given its limited memory bandwidth and lower compute density, using the CPU for full transformer attention layers or large matrix multiplies is often inefficient.
- **Mali-G310 GPU**: This is a single-shader low-power mobile GPU supporting efficient execution of vector and tensor operations and recent graphics APIs such as OpenCL and Vulkan. The GPU offers shader pipelines and parallelism for tasks amenable to Single Instruction Multiple Threads (SIMT) execution (e.g. matrix multiplications, elementwise activations) [67].

4.2.2 Assumptions

To constraint the search space, we assume that the parameter α (the expected value of the acceptance rate, see Equation 4.1) is known prior to compilation. This value is determined by the specific combination of drafting and evaluation methods employed and algorithm-level optimizations and can be assessed from an evaluation of the speculative decoding method on particular datasets and tasks. Regarding the hardware side, we assume that the system configuration (i.e., PUs and their availability) is also known a priori: for multiple PUs (cores on a CPU, shader units on a GPU, or processing elements in an NPU), we assume that the number of available cores for inference n_p of each PU P remains constant during runtime. Consequently, we do not consider scenarios involving dynamic resource contention from other processes, which would cause a dynamic variability in the available computational resources. In other words, we restrict our analysis to cases where information is available AOT to enable spatial partition of the workload on the highest-performing combination of PUs. Therefore, the input for our optimization problem requires: **the acceptance rate α (corresponding to the speculative decoding algorithm) and the system configuration (i.e., a list of available PUs with their possible configurations and the number of available cores)**.

Additionally, we restrict scheduling of the computational graph to spatial partitioning only, excluding any form of temporal scheduling. This simplification reduces the complexity of the search space, although it comes at the cost of flexibility. In practice, the computational load during LLM inference varies, for example, between the prefill and decoding phases, or as the KV cache grows from small to large. Dynamically reassigning subgraphs to different PUs during runtime could yield performance benefits, but such dynamic scheduling is beyond the scope of this work. Instead, we consider only the initial phase of inference, where the KV cache size is either empty or small, and the sequence length is much shorter than the model's hidden dimension. In this regime most of the computations are concentrated in the projections layers of the Feedforward Neural Network (FNN) section.

Finally, we assume that the speculative decoding strategy proceeds sequentially, with the lightweight drafting phase followed by the evaluation phase. Speculative decoding algorithms

that overlap these two phases, such as pipelined or interleaved approaches, are considered out of scope for this study.

4.2.3 Design Space Encoding

The search space we consider consists of the combinatorial set of valid assignments of subgraphs from the computational graph to PUs that support all operations within each subgraph. Each PU is characterized by a specific configuration, namely, a specific number of cores available for the inference task. This is a case of combinatorial distribution that includes m subgraphs that can be assigned to n processors, where $1 < n < N$, and N is the total number of available processors in the set $\{n_1, n_2, ..., n_N\}$. Each subgraph must be assigned to one processor, but there may be processors without any assigned subgraphs. The number of available combinations will be: N^m.

For the specific case of the i.MX95 SoC, different design alternatives are defined, determined by the combination of configurations of the Cortex-A CPU and the Mali G310 GPU. Different core configurations are available for the CPU cores (e.g., a single core or five cores can be designed for the speculative sampling application); however, the GPU has only one configuration due to its single shader core. Therefore, six different design alternatives are possible, as shown in Table 4.1. Notice that each one of these design has two PUs available, to which subgraphs can be assigned or not. For example, the design variant 1 consists of a CPU core and a GPU, giving us $N = 2$. Here, each of the m subgraphs can be mapped either only onto the CPU core, only onto the GPU, or distributed across both. In the former HW configuration, the GPU device remains free while the CPU core handles all computation.

As shown in the last column of Table 4.1, a design variant combined with a computational graph partitioned into only two subgraphs ($m = 2$) results in a relatively small possible hardware configurations. However, as the number of partitions increases, the number of possible hardware configurations grows accordingly. In the context of speculative decoding, we restrict our analysis to the case consisting of two physical PUs and two subgraph partitions ($m = 2$), where the first subgraph corresponds to the target model and the second one, to the drafter. Consequently, the size of each design alternative remains limited to $N^m = 2^2$. We always map the third subgraph (the procedural logic) onto the same device as the target model; therefore $m \neq 3$. This decision is based on the observation that the latency of the procedural logic subgraph is negligible.

Table 4.1: Different design alternatives arising from the different availability of computational cores

Design Alternative	Available Resources	No. of Possible HW Configurations
1	One CPU core and GPU	$2^2 = 4$
2	Two CPU cores and GPU	$2^2 = 4$
3	Three CPU cores and GPU	$2^2 = 4$
4	Four CPU cores and GPU	$2^2 = 4$
5	Five CPU cores and GPU	$2^2 = 4$
6	All CPU cores and GPU	$2^2 = 4$

4.2.4 Search Method

Given the relatively small size of the design variants, an initial exhaustive search may appear feasible. However, this is only the case for this particular edge platform. In larger edge systems, the number of heterogeneous PUs and available cores can be significantly higher, leading to a combinatorial explosion in the size of the design space. Consequently, performing exhaustive search with compilation, execution, and evaluation for every possible configuration would become prohibitively expensive in terms of time and computational resources.

To mitigate this, we propose strategies to reduce both the number and duration of compilations:

1. **Compile only homogeneous combinations**: we reduce the number of compilations to two targets: GPU and CPU. For different CPU configurations (i.e., varying numbers of cores) can be selected after compilation.
2. **Specialize the graph to a small number of generation steps**: We compile a graph tailored for ten decoding steps and use short initial sequence lengths (e.g., 4, 7, and 25 tokens). This approach not only shortens compilation time (by specializing the graph) but also aligns with our assumption that the sequence length remains much smaller than the hidden dimension, and that the impact of KV caching is negligible.
3. **Use an analytical performance model as a proxy**: This model serves to guide the search process, allowing us to estimate performance without executing every configuration, thereby accelerating the exploration of the design space.

4.2.5 Evaluation Method

Our optimization objective is to minimize the **prefill latency**, which is the dominant component of the TTFT [68]. Given our focus on the prefill stage, we adopt an experimental generative pipeline composed of multiple consecutive prefill forward passes. This design enables us to isolate and rigorously evaluate the performance of the prefill phase. The overall prefill latency is computed by aggregating the execution times of each individual prefill forward pass and taking the inverse of the total time.

We measure the prefill latency for homogeneous mappings by obtaining the execution time and the number of tokens generated under such configurations. However, for heterogeneous settings, an analytical cost model is used. A notable disadvantage of this kind of cost models is their simplicity: they often fail to model communication overhead and latencies across the heterogeneous compute and memory units. Additionally, these models tend to be tightly coupled to specific hardware configurations or narrowly defined workloads. In our case, the model is specifically tailored to optimize LLMs using speculative decoding. Conversely, our cost model can be generalized to different combinations of accelerators or PUs as exemplified in the Table 4.1. Finally, we validate the proposed model through measurements on silicon.

The analytical cost model is taken from [4], (see Equation 4.1) as it formalizes the speedup (or slowdown) of an LLM optimized with speculative decoding. Although the formula is derived for speculative sampling, it can be generalized to other types of speculative decoding in which the speculation and evaluation phases are sequential, since the calculation of the hardware

cost coefficient c assumes this condition. The cost model is presented again in Equation 4.1. For more details, review Chapter 2.

$$\frac{1-\alpha^{\gamma+1}}{(1-\alpha)(\gamma c+1)} \tag{4.1}$$

4.2.6 Methodology

For speculative sampling-type speculative decoding, we consider both quantized and unquantized versions of the model. According to our experiments and the existing literature ([48], [69]), these variants yield different expected acceptance rate values (α). However, one disadvantage of the quantization is that it changes the output distribution of the target model, such that the original unquantized output distribution $p(x)$ transforms into a quantized version $p_{quant}(x)$. The drafter model's output distribution $q(x)$ no longer aligns with the modified target distribution $p_{quant}(x)$ in the same way it did with $p(x)$. This leads to the two models having less well-fitting distributions, causing a drop in the acceptance rate. We consider the Llama 3.2 model with 3B and 1B parameters for the target model and the drafter model, respectively [70]. The unquantized version of `FP16` and a statically quantized version of `INT8` for both weights and activations are used.

We empirically determine the acceptance rate α using using a 16-core Intel Xeon W-3335 CPU. Although α is a model-dependent value, reflecting how closely the drafter model approximates the larger target model, it remains hardware-independent in most cases. However, slight deviations may occur when different devices handle rounding and precision differently, potentially leading to numerical discrepancies or precision divergence arising from quantization. Concretely, to estimate α, we employed the Spec-Bench, a benchmark designed to evaluate speculative decoding performance across multiple applications [71]. The dataset comprises 80 instances distributed among thirteen subtasks, covering the following variety of text generation scenarios:

1. Writing
2. Roleplay
3. Reasoning
4. Mathematics
5. Corresponding
6. Extraction
7. STEM
8. Humanities
9. Translation
10. Summarization
11. QA (question and answer)
12. RAG (Retrieval-Augmented Generation)
13. Math reasoning

In edge deployments, however, such variability is expected to diminish, since models are specialized for a single task and do not require the generality of foundational models. Techniques such as knowledge extraction or task-specific fine-tuning further reduce such high variability. Most importantly, the low median values originated by quantization and observed in Figure 5.2 are also anticipated to be absent in edge scenarios, where models are both fine-tuned and quantized using datasets that are directly relevant to the specific task at hand. Therefore, for the remainder of our analysis, we focus on the "translation" task. This task is commonly found on edge devices, where the prefill phase plays a critical role in ensuring responsive user experiences, particularly at the onset of live translation. Additionally, due to the nature of

translation, the length of generated tokens tends to closely match the input sequence length, which is typically short (a few tens of tokens representing a sentence).

Given that quantization plays a pivotal role in low-precision machine learning systems, especially relevant for edge devices with constrained memory, we evaluated the acceptance rate of multiple model configurations. These included (i) fully quantized target-drafter pairs, (ii) semiquantized combinations, where only the larger, target model was quantized, and (iii) unquantized FP16 counterparts as reference models. We do not consider a semiquantized configuration, where the target is quantized but the drafter remains unquantized, because the memory footprint during initialization (14.76 GB[1]) would place a significant burden on system memory of the edge device. Conversely, the configuration (ii) has a with a total memory footprint of 5.66 GB for the model weights; therefore, the memory required during initialization ascends to 11.32 GB[2]. All quantizations are static and are implemented in w8a8 precision using the Intel Neural Compressor [72].

It is important to highlight that the Mali GPU does not support the INT8 data type at the software layer. Therefore, it can only execute the FP16 drafter model, whereas the CPU fully supports both data types and can execute both the FP16-drafter and INT8-target models. Consequently, the homogenous execution scenario considered here consists of an only-CPU execution, while heterogeneity refers to offloading the entire drafting phase to the FP16-capable Mali GPU.

For each model configuration, α was computed over ten decoding steps, each comprising a draft length of five tokens ($\gamma = 5$) followed by one evaluation phase. To ensure deterministic comparability, greedy sampling was used consistently across all experiments. Nevertheless, a small divergence between the measured α and that which would be observed on a true embedded device remains possible, since the x86 architecture we use lacks native FP16 datatype support, again possibly introducing precision mismatches.

To compute the parameter c we use IREE 3.6.0 for compilation, with modifications primarily targeting the code generation pipelines for LLVM (CPU) and Standard Portable Intermediate Representation V (SPIR-V) (GPU), that are detailed in Appendix A. The target devices are Cortex-A55 with Neon SIMD and Mali G310 with Vulkan (robust mode disabled). The unquantized drafter model is compiled for both devices, while the quantized target model is compiled only for the CPU due to lack of INT8 support on Vulkan in this IREE version, which results in a total of three different compiled flat buffer artifacts. For benchmarking the end-to-end execution time, an unmodified version of IREE runtime 3.6.0 installed via pip package is used.

Each compiled artifact includes two entry points: one to initiate a generation of up to ten decodification steps with speculative decoding and the other with greedy sampling (i.e., without speculative decoding). This enables us to have a more direct comparison of the execution times of both decoding variants. Then, we extract the individual execution times of the target model and the draft model. With this information, it is possible to calculate the cost coefficient c for each heterogeneous mapping.

[1] In version 3.6.0 of the IREE compiler, the memory footprint during initialization is twice the size of the model parameters due to packing operations. While the resulting memory peak of 14.76 GB is manageable on an Intel Xeon CPU, it poses a significant challenge for the edge environment under consideration.

[2] While this is still relatively high for an edge environment, it remains manageable within our experimental setup, where no other large programs are competing for memory resources

With the expected acceptance rate values (α) obtained for task and the c parameters for different device configurations, we can apply the cost model of the Equation 4.1 to estimate the speedup that speculative decoding would bring for each heterogeneous combination of PUs as well to estimate the γ yielding such speedup. For each of the six design variants established in the Table 4.1, we calculate the performant mapping configurations. Finally, after identifying the best-performing mapping for each design variant, we validate the predictions of the configuration that yields the higher speedup measuring on silicon the actual acceleration compared with the baseline, which is the homogenous CPU execution without speculative decoding.

5 Evaluation and Analysis

This chapter presents the exhaustive evaluation of speculative sampling using the Spec-Bench benchmark and LLaMA 3.2 models, focusing on the prefill latency across homogeneous and heterogeneous hardware configurations. It begins by analyzing how quantization affects the acceptance rate (α), then explores the impact of hardware mapping on latency and speedup, utilizing a cost model to guide optimization. The model is validated through experiments that compare estimated and measured acceleration, confirming its utility in selecting optimal draft lengths (γ) and hardware configurations.

5.1 Homogeneous Mapping and Execution

We proceed to evaluate the acceptance rate values (α) using the Spec-Bench benchmark under a homogeneous hardware configuration.

5.1.1 Quantization Effects on α

The results of our empirical estimation of the α values performed against the Spec-Bench benchmark show that quantization negatively impacts the expected acceptance rate (α), as shown in Figure 5.1. Here, the distribution of α values (y-axis) across translation and summarization tasks from the Spec-Bench benchmark. Specifically, α values are shown for translation (left), summarization (center), and the entire benchmark dataset (right). The measurements were conducted over ten decoding steps using various model pairings. The target model is LLaMA 3B, while the drafter is a smaller 1B-parameter variant from the same model family. In each plot, three model pairings are presented: *Unquantized* refers to the original FP16 format, whereas *quantized* models use static INT8 precision. It clearly illustrates the impact of quantization on the acceptance rate α: in all three plots, when quantization is applied, the degradation in α becomes significant (notice the fall of the boxes in each plot while we increase the quantization). In particular, when both the draft and target models are quantized (left-most box in each plot), the acceptance rate drops completely.

Concretely, the highest α median value (0.46) was obtained using the unquantized models, whereas the lowest α value corresponds to the fully quantized models, where it drops to zero

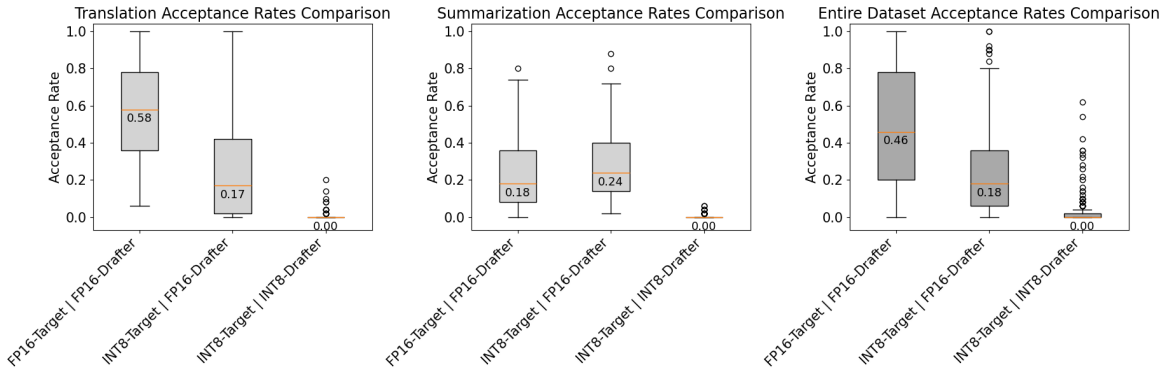

Figure 5.1: α values are shown for translation (left), summarization (middle), and all the dataset of the Spec-Bench benchmark (right).

(see left-most plot on Figure 5.1). This observation highlights an intrinsic trade-off associated of integrating quantization to speculative decoding: while quantization reduces memory usage, it introduces a trade-off between token generation speed and memory footprint (since quantization reduces the α value, but improves memory bandwidth and memory footprint).

This behavior was already explained and is theoretically expected. When both models are quantized, this distributional mismatch becomes even more pronounced, further reducing α. To mitigate this degradation, the semi-quantized configuration (quantized target model and unquantized drafter model) offers a practical balance: it provides manageable memory usage for the edge platform in question (i.MX95) while maintaining an intermediate acceptance rate that preserves decoding efficiency.

5.1.2 Considerations for the Acceptance Rate α on Edge

Figure 5.2 allows a closer inspection of the results revealing a strong task dependency of α (y-axis), among the thirteen different tasks of the dataset (x-axis). We also show the distribution of the entire set in the right-most box in the plot. Comparing the different tasks, we find a substantial intratask variability. For instance, the "extraction" task exhibited a median α of 0.20 and a relatively low spread variance (ranging from 0.02 to 0.53, excluding the outlier). In contrast, the "translation" task showed a slightly lower median α of 0.17, but with a wide spread having samples reaching down to 0.00 and up to 1.00. This variability reflects the differences in token predictability and model confidence across task types for both the LLaMA 3.2. model family and the Spec-Bench dataset. Consequently, the assumption of a known range of values for α, which is necessary to apply the cost model introduced in the second problem of Chapter 4, may no longer hold.

5.2 Heterogenous Mapping and Execution

In this section, we estimate the achievable acceleration utilizing the model combination that leads to the best trade-off between speedup and memory footprint: `INT8`-quantized target model and the unquantized `FP16` drafter model.

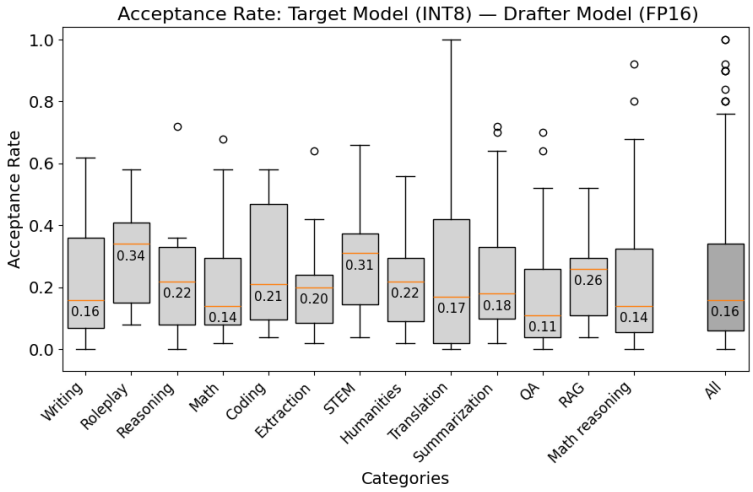

Figure 5.2: Similar to the model characteristics shown in Figure 5.1, this figure focuses on model combinations where only the target model is quantized. The remaining 11 tasks are displayed, highlighting both the strong task-specific dependence of α and the intra-task variability.

5.2.1 Data Collection for Modeling the Impact of Speculative Decoding and Heterogenous Execution

We individually measured the prefill latency for both the target and drafter models as a function of the input sequence length. These measurements are depicted in Figure 5.3, where in the x-axes (in logarithmic scale) represents the input length that is fed to the LLM that produces a concrete prefill latency (logarithmical y-axes). The left plot of Figure 5.3 illustrates the prefill latency of the unquantized drafter model across various homogeneous device configurations. CPU configurations are depicted using cold colors (refer to the legend above), while the GPU curve is shown in orange. The right panel presents analogous measurements for the INT8-quantized target model; however, the GPU curve is omitted due to the lack of support for the INT8 data type.

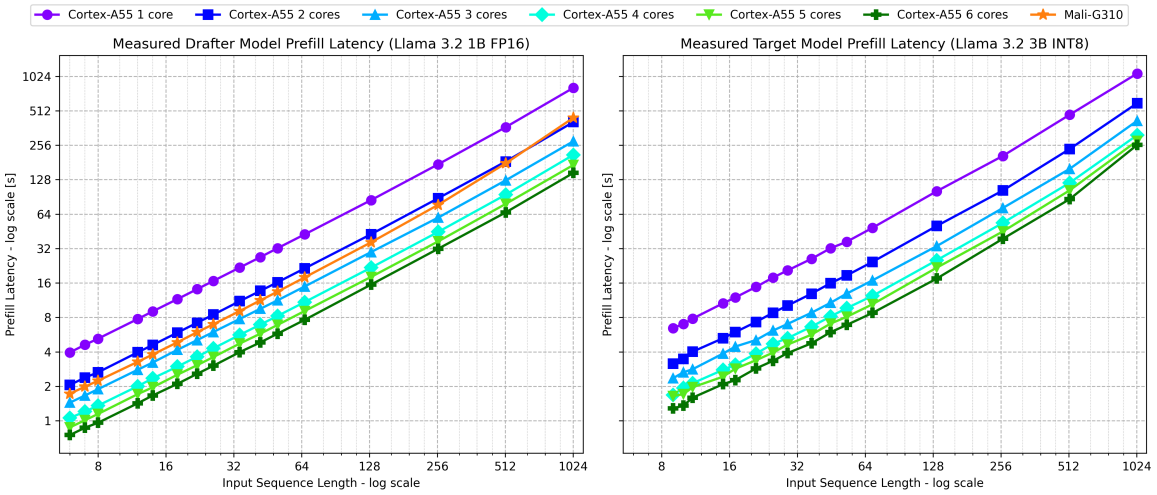

Figure 5.3: Comparison of the prefill latencies of the drafter model (left) and the target model (right), as a function of the input sequence length across various hardware configurations. To ensure hardware compatibility and maintain high drafter quality, the target model is quantized to INT8, and therefore not executed on the Mali-G310 GPU.

The results indicate that, for the drafter model (Figure 5.3, left), the Mali GPU consistently outperforms the single-core CPU configuration across all evaluated sequence lengths. Furthermore, the GPU maintains a performance advantage over the two CPU cores configuration, although this benefit is limited to input sequences shorter than 512 tokens. However, the reported GPU latencies were only achieved after applying a data layout reordering optimization to achieve a larger utilization of the GPU resources: during the compilation phase, a tiling strategy optimized for tall and skinny matrices (i.e., very low input sequence lengths) was applied, see Appendix B for details.

Continuing with the analysis of the GPU behavior (orange curve on left-side plot of Fig. 5.3), we observe that high input sequence lengths exhibits a more pronounced superlinear growth compared to its CPU counterparts, regardless of the number of cores (notice how the orange curve starts growing notoriously faster than the cold-colored curves). We argue that this shift in inference regime of the GPU is a significant contributor to this superlinear behavior. As the input sequence length approaches the hidden dimension of the drafter model (2048), the quadratic complexity of activation-to-activation matrix multiplications begins to dominate, overtaking the linear growth associated with the projection (i.e., FNN) of the transformer layers.

This regime transition further accounts for the latency patterns observed in the CPU, as depicted by the cold-colored curves in the same Fig. 5.3. For both right-side and left-side plots (i.e., the prefill latencies of drafter and target model, respectively) notice how for small input sequence lengths (less than 64 tokens) the latency increases linearly. As the sequence length grows, a superlinear trend emerges: for the drafter model (Fig. 5.3 left), this is more pronounced on the Mali GPU, as previously discussed (orange curve), while the cold-colored curves present a milder superlinear growth. While for the target model (Fig. 5.3 right), it becomes more evident in the six-core CPU configuration corresponding to bottom-most dark green curve, as is less strong in he one-core CPU configuration (top purple curve). The stronger superlinear behavior in the former case (the bottom-most dark green curve in the right-hand plot of Fig. 5.3) is attributed to its larger hidden size (3072 vs. 2048 in the drafter model), which amplifies the impact of quadratic operations during inference.

Using the measured prefill latency data, we then computed the cost coefficients c as described in Chapter 4. The Figure 5.4 plots the calculated cost coefficients based on the six evaluated design configurations as a function of the input sequence length. In the left, we present the cost coefficient for these six design variants for the homogenous mapping. In the right, we display the possible heterogenous combinations for such design variant.

In both cases, a red shadow highlights cost coefficients greater than 1, indicating unfeasible configurations where the drafter model's latency exceeds that of the target model. These scenarios occur only in heterogeneous mappings within design variants featuring four to six CPU cores. In contrast, the heterogeneous configuration with three CPU cores is only feasible for very short input sequence lengths. The most favorable cost coefficient corresponds to the lowest value, achieved by the heterogeneous mapping in the design variant with a single CPU core (represented by the purple line with circle markers on the right side of Figure 5.4). This configuration outperforms its homogeneous counterpart (purple curve on the left side), as the drafter execution is approximately twice as fast across all measured input sequence lengths when offloaded from the single CPU core to the Mali GPU. This performance gain is evident

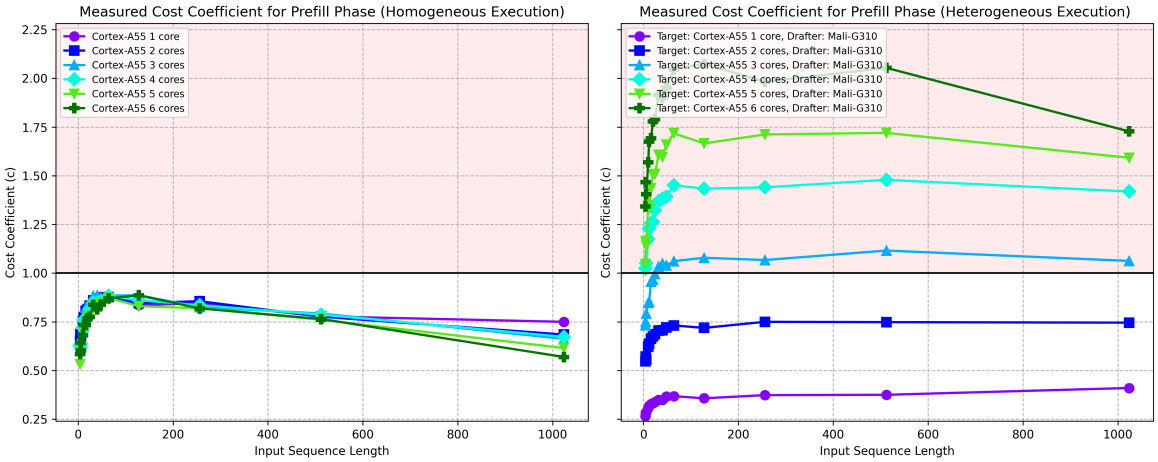

Figure 5.4: Calculated cost coefficients c based on measured latencies in function of the input sequence length for homogenous (left) and heterogeneous configurations (right). Red shadow is used for unfeasible cost models, that are larger than one.

when comparing the GPU latency (orange curve) with the single-core CPU latency (purple curve) in Fig. 5.3, left.

5.2.2 Estimating the Impact of Speculative Decoding and Heterogenous Execution

Obtaining these c coefficients enables a cost-based characterization that serves as the foundation for speedup estimation. This assessment is performed by interpolating the latency measurements previously presented in Figure 5.3, followed by applying Equation 4.1 to compute the corresponding speedup values. Tables 5.1 and 5.2 present the most performant mappings according to the cost model, and indicate for each design variant whether applying speculative decoding and/or a heterogenous mapping results in latency improvements compared to the baseline, defined as homogeneous CPU execution. We present the results of this analysis with a fixed input sequence length of 63 (as explained in the following paragraph) and two fixed α values of 0.90 and 0.17.

The selection of $\alpha = 0.90$ corresponds to the top 10% acceptance rates observed for the translation task in the dataset. This threshold also aligns with the assumption that, in edge scenarios, models are typically highly specialized for specific tasks, and therefore, high α values are expected. In contrast, $\alpha = 0.17$ represents the measured median acceptance rate for the same task (see Figure 5.2). The input sequence length of 63 is derived from the expression $S_L + \frac{G_L}{2}$, where S_L denotes the average input sequence length in the dataset for translation (42 tokens), and G_L represents the generated sequence length. As previously discussed, we assume $G_L = S_L$ due to the nature of translation tasks. Dividing G_L by 2 reflects the average speedup achieved through speculative decoding, which is justified by the observation that the performance curve is approximately linear within the considered range.

From these results, we observe in Table 5.1 that the effectiveness of speculative sampling and heterogenous mapping depends largely on the available hardware (i.e., the design variant). The design variant No. 1, the one with less available CPU cores, yielded the greater performance

gain (estimated at 1.68×) from employing speculative sampling and heterogenous execution for these specific heterogenous SoC and α. Moreover, the correct selection of the draft sequence length (γ) is crucial: each design variant reaches its better performance with a different γ value, ranging from 0 (no speculative sampling) to 5. In contrast, design variants with a higher number of available cores (i.e., from 3 cores upwards) experienced a performance decline when both speculative decoding and heterogeneous execution were applied. Therefore, these methods should be avoided in such configurations. For the design variant holding five CPU cores, there is a modest speedup, but heterogeneous mapping should not be applied. If the performance gain is too small, there is a risk that, in a real system, the improvement becomes negligible, especially when accounting for deployment overheads. As a result, we discourage the use of heterogeneous mapping in this scenario.

Table 5.1: Estimation of best achievable speedup across different design variants with an $\alpha = 0.90$ and a shorter input sequence length of 63.

Design Variant[3]	Speculative Sampling	Heterogenous Execution	Speedup [×]
1	Yes ($\gamma = 5$)	Yes	1.68
2	Yes ($\gamma = 2$)	Yes	1.10
3	No	NA	1
4	No	NA	1
5	Yes ($\gamma = 1$)	No	1.02
6	No	NA	1

Table 5.2: Applying speculative decoding and heterogenous mapping to model pairs with low acceptance rates (here $\alpha = 0.17$ and sequence length of 63) lead to no acceleration.

Design Space[3]	Speculative Sampling	Heterogeneous Exexution	Speedup [×]
1-6	No	NA	1

While selecting an appropriate draft length γ is important, ensuring high-quality speculative decoding, characterized by a high acceptance rate α relative to the cost coefficient c, is even more critical. When α is insufficiently high, the scenario illustrated in Table 5.2 emerges: the cost of the drafting method outweighs its benefits, making it preferable to avoid its use altogether. In such cases, the coarse-grained partitioning strategy is also ineffective and should be discarded.

Furthermore, we observe that the input sequence length significantly influences the effectiveness of the strategy. Table 5.3 presents a configuration similar to previous tables, assuming a high acceptance rate of $\alpha = 0.9$, but with a longer input sequence length of 800 tokens, more typical of summarization tasks. Despite its length, this still falls within the short-sequence regime, as it remains smaller than the model's hidden dimension. Notably, the behavior differs from that observed with shorter input sequences (see Table 5.1). In contrast to the shorter sequences,

[3]Each design variant features the Mali GPU and the number of CPU cores indicated in the numbers of the column. For example, the design variant No. 3 holds a configuration of three CPU cores and the GPU.

an input length of 800 yields theoretical speedups across all design variants; however, these gains become modest starting from the second design variant onward.

Table 5.3: Estimation of best achievable speedup across different design variants with an $\alpha = 0.90$ and a longer input sequence length of 800.

Design Space[3]	Speculative Sampling	Heterogeneous Execution	Speedup [×]
1	Yes ($\gamma = 4$)	Yes	1.64
2	Yes ($\gamma = 1$)	Yes	1.08
3	Yes ($\gamma = 1$)	No	1.06
4	Yes ($\gamma = 1$)	No	1.06
5	Yes ($\gamma = 1$)	No	1.07
6	Yes ($\gamma = 1$)	No	1.07

While these observations apply to the evaluated configuration, they may not generalize across different acceptance rate (α) values. This limitation is illustrated in Figure 5.5, where the x-axis represents α values from 0.35 to 0.90, and the y-axis shows the acceleration achieved by mapping the target model to a CPU core and the drafter model to the GPU. Each curve represents the speedup corresponding to a different draft sequence length γ. Values of α below 0.35 are excluded, as they do not yield any acceleration. As shown, higher α values consistently enable greater acceleration, especially with longer draft sequences. The choice of γ is therefore critical. For instance, selecting $\gamma = 5$ with $\alpha = 0.6$ results in a performance slowdown, as indicated by the orange curve falling below an acceleration of 1. Moreover, when α drops below 0.37, even the shortest draft sequence ($\gamma = 1$, shown in purple) leads to negative performance impact, making speculative decoding counterproductive. These findings highlight the importance of selecting a drafting strategy that aligns with the task characteristics. A higher α not only improves performance but also facilitates effective use of heterogeneous hardware mappings.

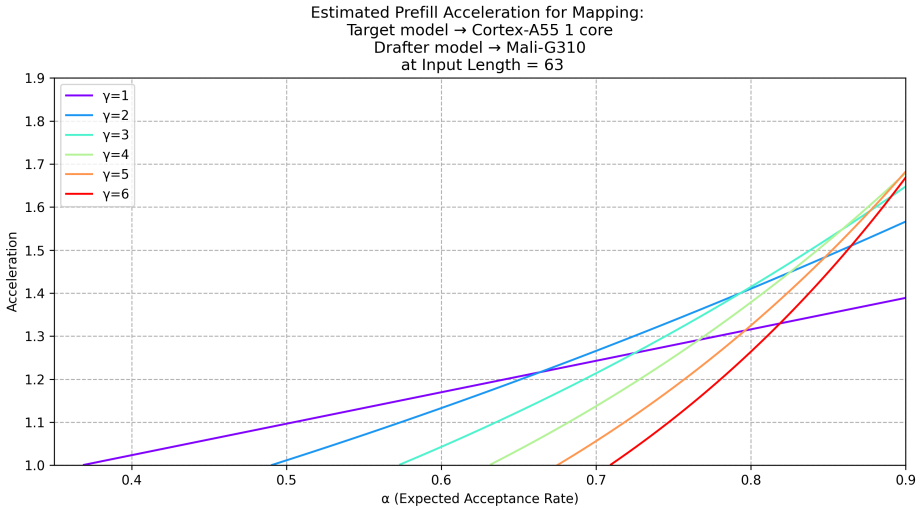

Figure 5.5: Estimated acceleration introduced by speculative sampling with different drafted token lengths γ executed heterogeneously on one CPU core and on GPU. The input sequence length is set to 63.

Although these findings are specific to the selected model-hardware combination, the proposed cost model generalizes to other configurations by collecting a small amount of runtime data for each heterogeneous mapping. While our analysis focuses on the prefill phase and relatively short input sequences (where runtime characteristics differ from those in the decode phase and long-sequence scenarios) the same analytical framework can be systematically extended to those execution regimes (i.e., prefill with long sequence lengths, decode with short and long sequence lengths), as well as to other hardware platforms and tasks.

5.3 Model Validation

To validate the proposed cost model, we implemented, compiled, and executed both the baseline configuration and the hardware setup predicted to yield the highest acceleration according to the analytical estimation. This configuration corresponds to design variant 1, (i.e., a single CPU core combined with the GPU), where speculative decoding is enabled and heterogeneous execution is employed. In this mapping, the FP16 drafter model is executed on the GPU, while the quantized target model remains on the CPU. In this section, we first describe the changes of methodology that lead to the results we present subsequently.

5.3.1 Methodology Changes

The original design aimed at implementing a single monolithic graph representing the entire generative pipeline proved impractical in the context of heterogeneous execution. Although such a unified graph could have reduced runtime dependencies and potentially improved performance, runtime constraints in IREE 3.6.0 prevented its deployment in the current framework. Consequently, we compiled two separate computational graphs, one for each device, and orchestrated their execution using IREE's Python runtime API. While this introduces a friction interface that could reduce acceleration, it provided the flexibility necessary for cross-device coordination and heterogeneity execution. Nonetheless, the drafter phase remained monolithic, with separate compiled versions for each γ value. Thus, in the speculative decoding pipeline, each generation cycle involved only two Python-level calls: one to the appropriate drafter model and one to the target model. In the baseline case, only a single call to the target model was required.

Following the rationale of the previous sections, we selected the translation task as a representative edge application. This choice is motivated by its prevalence in on-device inference scenarios and by the highly variable acceptance rate distribution observed in this category, ranging from very low (0%) to very high (100%) with greedy sampling. Such variability allows for an extensive comparison between measured and estimated results across the full range of γ values considered.

We used an input sequence length of 63 tokens, for the same reasons explained above for the translation task. During the experiment, random input sequences values of the specified length were generated, and a dedicated validation mechanism progressively reduced the acceptance rate from 1 to 0 in discrete steps. This setup, though artificially controlled, was designed to

[3]Each design variant features the Mali GPU and the number of CPU cores indicated in the numbers of the column. For example, the design variant No. 3 holds a configuration of three CPU cores and the GPU.

isolate the influence of acceptance rate on performance, thereby validating the cost model independently of the drafter model's semantic accuracy.

In this experimental setup, the drafter model executes γ times per generation phase. Its generated sequence is then consumed by the target model, which produces one additional token. Depending on the current acceptance rate, drafter outputs accepted or rejected in a controlled manner. This cycle repeats ten times, always with an input sequence length of 63. In the end, we take the average of these ten runs and compare it with the baseline. For the baseline configuration, we employed the same sequence length and token generation setup, but without the drafter phase, meaning that all computations were performed on one CPU core.

5.3.2 Results

The results of this experiment are presented alongside the previous estimations in Figure 5.6, plotted against varying α values on the x-axis for direct comparison. Each curve corresponds to a different draft sequence length, γ. The plot on the left, illustrating the estimated acceleration, is adapted from Figure 5.5, with the range of α extended from 0.90 to 0.95 to ensure consistency with the measured results shown on the right. The right-hand plot includes empirical data for $\gamma \in [1, 6]$, enabling a direct comparison between.

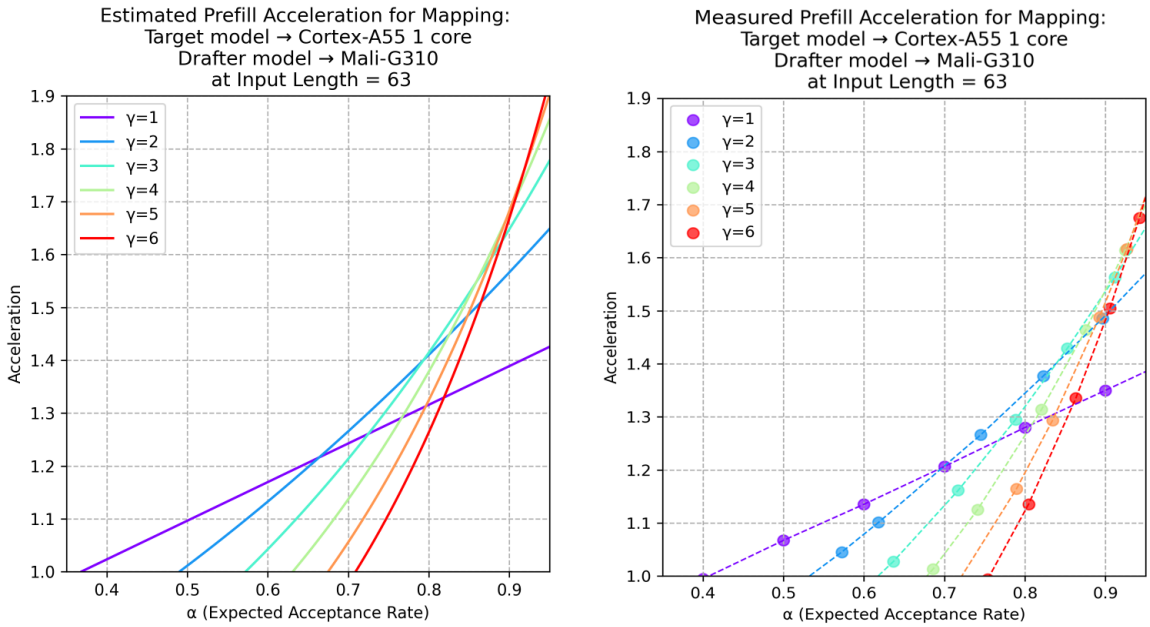

Figure 5.6: Measured acceleration introduced by speculative sampling with different drafted token lengths γ executed heterogeneously on one CPU core and on GPU. The input sequence length is set 63 and we report the average after ten runs.

Overall, the measured results (Fig. 5.6 right) closely align with the predictions of the cost model (compare to the left): lower γ values yield the highest speedups at lower α values, while greater accelerations are observed with higher γ and α combinations. However, a more detailed analysis reveals subtle discrepancies, particularly in the intersection points between α values and the γ curves. All six measured γ curves cross the negligible acceleration threshold

(i.e., 1× speedup) at higher α values than predicted. For example, the estimated curve for $\gamma = 1$ (purple curve in the left) reaches negligible acceleration at approximately $\alpha = 0.37$, whereas the measured data shows this transition occurring at $\alpha = 0.40$ (purple curve in the right). This suggests that the actual cost coefficient c is slightly higher than anticipated, likely around $c = 0.40$ instead of the expected $c = 0.37$. A more detailed comparison is provided in Table 5.4.

Table 5.4: Comparison between estimated and measured prefill latency values and cost coefficient c for an input sequence length of 63. The prefill latency values are normalized relative to the target's estimated prefill latency.

Component	**Estimated Value**	**Measured Value**
Drafter latency on GPU	0.37	0.39 – 0.40
Target latency on one CPU core	1	0.96
Cost Coefficient c	0.37	0.41–0.42

Notably, the measured prefill latency for the target model was approximately 4% lower than anticipated, which we attribute to a change in the runtime driver. In contrast, the drafter execution time increased between 5.3% to 8.2%, depending on the design alternative. This variation arises from higher γ values producing longer input sequences, thereby incrementally increasing inference time at each step. The combined effect (slower drafting and faster target model inference) results in a higher observed cost coefficient c. This explains the shift to the right of roughly four percentage points in the acceleration curves, from predicted to measured values (see Fig. 5.6, left vs. right plot). Similarly, the expected acceleration of 1.68× at an acceptance rate of $\alpha = 0.90$ and a drafter length of $\gamma = 5$ (orange curve, left plot), as reported in Table 5.1, was also slightly shifted.

Surprisingly, the introduction of the Python orchestration layer did not result in any measurable performance degradation in the execution time of the target model on the CPU core. However, we observed a slight increase in the cost of the drafter on the GPU, which we attribute to the overhead introduced by Python. We hypothesize that this modest impact is due to the coarse temporal granularity of the operations, generation times occur on the order of seconds (as illustrated in Figure 5.3). For tasks involving more sensitive latency requirements, the overhead introduced by Python may become more pronounced and potentially detrimental to performance.

In summary, this validation demonstrates that even a simple analytical cost model, built from a limited set of empirical measurements, can effectively guide design decisions. These include whether to adopt speculative decoding, how to select the optimal draft sequence length (γ), and whether to deploy on a heterogeneous or homogeneous system architecture, depending on the underlying system characteristics. Moreover, the results underscore the substantial impact of hardware configuration, task characteristics, and speculative decoding technique on system performance. As illustrated in Table 5.1, the most performant setup varies significantly across scenarios, highlighting the need for an analytic heuristic that can statically inform optimization decisions. This approach could be readily integrated into compilation frameworks such as IREE, enabling automatic hardware-software co-optimization for edge-oriented generative workloads.

6 Conclusion and Future Work

This work addressed two research questions concerning the integration of speculative decoding and heterogeneous execution for edge-oriented generative workloads. Regarding the first question, the feasibility and benefit of expressing the entire pipeline as a single static graph, we found that, although low-level partitioning could be lifted to the high level during compilation, practical deployment remains constrained by runtime limitations. Specifically, executing a monolithic graph with multiple devices affinities proved impractical under the current runtime framework. Consequently, we adopted a Python-based orchestration layer to coordinate heterogeneous execution. Interestingly, this additional interface did not introduce measurable performance degradation, likely because its overhead was negligible compared to the overall model execution time. However, finer-grained orchestration or lower model latency could amplify this overhead, making runtime design choices critical for future implementations.

The second research question explored the benefits of heterogeneous computing combined with speculative decoding. We demonstrated that an analytical cost model can effectively guide decisions on enabling these techniques using only a small set of measurements collected under homogeneous settings. The model was validated on a representative edge application (machine translation) across six design variants. For high acceptance rates, speculative decoding combined with heterogeneous execution yielded up to 1.68× speedup for short sequences when the drafting phase was offloaded to the Mali GPU; this speedup was estimated to be achieved by an $\alpha = 0.9$ and was measured with $\alpha = 0.94$. Conversely, for lower acceptance rates (e.g., $\alpha = 0.17$), speculative decoding introduced no acceleration, underscoring the importance of task-specific tuning to achieve high drafter alignment. Furthermore, the effectiveness of speculative decoding depends on multiple factors: the cost coefficient c, the acceptance rate (α, which reflects the alignment between drafter and target distributions), and the sequence length. Hardware characteristics also play a decisive role; for instance, configurations with more CPU cores often showed negligible or negative gains when applying heterogeneous mapping in the studied edge platform. Additionally, quantization emerged as a critical constraint: while it reduces memory footprint, it significantly degrades acceptance rates, making speculative decoding ineffective when both models are quantized.

Despite these results, several limitations remain. The technique was evaluated on a single SoC type, and while varying the number of CPU cores provided insights into generality, broader validation across diverse heterogeneous systems is needed. Moreover, the proposed cost model

is tightly coupled to the speculative decoding strategy employed. Alternative drafting methods, such as tree-based approaches or pipelined validation, would require adapting the model, particularly the definition of parameters like γ. Finally, the analysis focused on the prefill phase and short-sequence regimes; extending the framework to other inference phases and longer sequences is essential for comprehensive applicability.

Future work should pursue directions. First, extending the analysis to other execution regimes, such as the decode phase, would provide a more complete understanding of speculative decoding dynamics on edge environments. Second, integrating the proposed cost model into compiler infrastructures like IREE could enable automated hardware-software co-optimization, dynamically selecting performant mappings as design variants change. Third, the coarse-grained partitioning explored here opens opportunities for multi-tenant ML applications, where workloads share resources across CPU cores and accelerators under a unified runtime. Finally, incorporating finer-grained partitioning strategies would allow the system to account NPU offloading and optimize offloading decisions more effectively, further improving performance in heterogeneous edge systems.

Bibliography

[1] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020, ISSN: 1558-2256. DOI: 10.1109/JPROC.2020.2976475 Accessed: Sep. 12, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9043731/

[2] L. Chen et al., "HeteroLLM: Accelerating Large Language Model Inference on Mobile SoCs platform with Heterogeneous AI Accelerators," arXiv:2501.14794 [cs], arXiv, Jan. 2025. DOI: 10.48550/arXiv.2501.14794 Accessed: Sep. 17, 2025. [Online]. Available: http://arxiv.org/abs/2501.14794

[3] J. Xu et al., *On-Device Language Models: A Comprehensive Review*, arXiv:2409.00088 [cs], Sep. 2024. DOI: 10.48550/arXiv.2409.00088 Accessed: Mar. 9, 2025. [Online]. Available: http://arxiv.org/abs/2409.00088

[4] Y. Leviathan, M. Kalman, and Y. Matias, "Fast Inference from Transformers via Speculative Decoding," en, in *Proceedings of the 40th International Conference on Machine Learning*, ISSN: 2640-3498, PMLR, Jul. 2023, pp. 19 274–19 286. Accessed: Mar. 13, 2025. [Online]. Available: https://proceedings.mlr.press/v202/leviathan23a.html

[5] Y. Li, F. Wei, C. Zhang, and H. Zhang, *EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test*, arXiv:2503.01840 [cs] version: 1, Mar. 2025. DOI: 10.48550/arXiv.2503.01840 Accessed: Mar. 13, 2025. [Online]. Available: http://arxiv.org/abs/2503.01840

[6] S. Laskaridis, S. I. Venieris, A. Kouris, R. Li, and N. D. Lane, "The Future of Consumer Edge-AI Computing," *IEEE Pervasive Computing*, vol. 23, no. 3, pp. 21–30, Jul. 2024, ISSN: 1558-2590. DOI: 10.1109/MPRV.2024.3417314 Accessed: Apr. 30, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10591422/authors

[7] H. Hua, Y. Li, T. Wang, N. Dong, W. Li, and J. Cao, "Edge Computing with Artificial Intelligence: A Machine Learning Perspective," en, *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, Sep. 2023, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3555802 Accessed: Oct. 29, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3555802

[8] H. Wen et al., "AutoDroid: LLM-powered Task Automation in Android," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, ser. ACM MobiCom '24, New York, NY, USA: Association for Computing Machinery, May 2024, pp. 543–557, ISBN: 979-8-4007-0489-5. DOI: 10.1145/3636534.3649379 Accessed: Oct. 28, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3636534.3649379

[9] Y. Labrak, A. Bazoge, E. Morin, P.-A. Gourraud, M. Rouvier, and R. Dufour, *BioMistral: A Collection of Open-Source Pretrained Large Language Models for Medical Domains*, arXiv:2402.10373 [cs], Jul. 2024. DOI: 10.48550/arXiv.2402.10373 Accessed: Oct. 29, 2025. [Online]. Available: http://arxiv.org/abs/2402.10373

[10] Z. Gu, B. Zhu, G. Zhu, Y. Chen, M. Tang, and J. Wang, "AnomalyGPT: Detecting industrial anomalies using large vision-language models," in *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'24/IAAI'24/EAAI'24, vol. 38, AAAI Press, Feb. 2024, pp. 1932–1940, ISBN: 978-1-57735-887-9. DOI: 10.1609/aaai.v38i3.27963 Accessed: Oct. 29, 2025. [Online]. Available: https://doi.org/10.1609/aaai.v38i3.27963

[11] Y. Zheng, Y. Chen, B. Qian, X. Shi, Y. Shu, and J. Chen, "A Review on Edge Large Language Models: Design, Execution, and Applications," en, *ACM Computing Surveys*, p. 3 719 664, Feb. 2025, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3719664 Accessed: Mar. 9, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3719664

[12] Z. Yuan et al., *LLM Inference Unveiled: Survey and Roofline Model Insights*, arXiv:2402.16363 [cs] version: 4, Mar. 2024. DOI: 10.48550/arXiv.2402.16363 Accessed: Mar. 21, 2025. [Online]. Available: http://arxiv.org/abs/2402.16363

[13] S. Laskaridis, K. Katevas, L. Minto, and H. Haddadi, "MELTing point: Mobile Evaluation of Language Transformers," arXiv:2403.12844 [cs], Jul. 2024. Accessed: Mar. 9, 2025. [Online]. Available: http://arxiv.org/abs/2403.12844

[14] G. Team et al., *Gemini: A Family of Highly Capable Multimodal Models*, arXiv:2312.11805 [cs], May 2025. DOI: 10.48550/arXiv.2312.11805 Accessed: Oct. 29, 2025. [Online]. Available: http://arxiv.org/abs/2312.11805

[15] T. Knopp, J. Chu, and S. Ahmad, "AMD Versal AI Edge Series Gen 2," *IEEE Micro*, vol. 45, no. 3, pp. 22–30, May 2025, ISSN: 1937-4143. DOI: 10.1109/MM.2025.3551319 Accessed: Oct. 29, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10926865

[16] Rockchip, *Rockchip*. Accessed: Oct. 29, 2025. [Online]. Available: https://www.rockchips.com/a/en/products/RK35_Series/2022/0926/1660.html

[17] T. P. Swaminathan, C. Silver, and T. Akilan, *Benchmarking Deep Learning Models on NVIDIA Jetson Nano for Real-Time Systems: An Empirical Investigation*, arXiv:2406.17749 [cs], Jun. 2024. DOI: 10.48550/arXiv.2406.17749 Accessed: Oct. 29, 2025. [Online]. Available: http://arxiv.org/abs/2406.17749

[18] STMicroelectronics, *STM32MP157 MPU Dual Arm Cortex-A7 cores - STMicroelectronics*, en. Accessed: Oct. 29, 2025. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32mp157.html

[19] NXP, *I.MX 95*, 2025. Accessed: Sep. 11, 2025. [Online]. Available: https://www.nxp.com/products/i.MX95

[20] J. Lee et al., *On-Device Neural Net Inference with Mobile GPUs*, arXiv:1907.01989 [cs], Jul. 2019. DOI: 10.48550/arXiv.1907.01989 Accessed: Aug. 14, 2025. [Online]. Available: http://arxiv.org/abs/1907.01989

[21] S. Jiang, L. Ran, T. Cao, Y. Xu, and Y. Liu, "Profiling and optimizing deep learning inference on mobile GPUs," en, in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, Tsukuba Japan: ACM, Aug. 2020, pp. 75–81, ISBN: 978-1-4503-8069-0. DOI: 10.1145/3409963.3410493 Accessed: Aug. 14, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3409963.3410493

[22] A. Vaswani et al., "Attention is All you Need," in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017. Accessed: Oct. 29, 2025. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[23] S. Kim et al., "Full Stack Optimization of Transformer Inference: A Survey," en, in *Architecture and System Support for Transformer Models*, Issue: arXiv:2302.14017 arXiv:2302.14017 [cs], Feb. 2023. Accessed: Apr. 3, 2024. [Online]. Available: http://arxiv.org/abs/2302.14017

[24] B. Li, Y. Jiang, V. Gadepally, and D. Tiwari, *LLM Inference Serving: Survey of Recent Advances and Opportunities*, arXiv:2407.12391 [cs], Jul. 2024. DOI: 10.48550/arXiv.2407.12391 Accessed: Oct. 29, 2025. [Online]. Available: http://arxiv.org/abs/2407.12391

[25] R. Tiwari et al., "QuantSpec: Self-Speculative Decoding with Hierarchical Quantized KV Cache," Feb. 2025. Accessed: Mar. 9, 2025.

[26] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," en, *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, Number: 4, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1498765.1498785 Accessed: Apr. 26, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/1498765.1498785

[27] A. N. Mazumder et al., "A Survey on the Optimization of Neural Network Accelerators for Micro-AI On-Device Inference," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 11, no. 4, pp. 532–547, Dec. 2021, Number: 4 Conference Name: IEEE Journal on Emerging and Selected Topics in Circuits and Systems, ISSN: 2156-3365. DOI: 10.1109/JETCAS.2021.3129415 Accessed: Apr. 16, 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9627710/authors#authors

[28] Y. Hu, Z. Liu, Z. Dong, T. Peng, B. McDanel, and S. Q. Zhang, *Speculative Decoding and Beyond: An In-Depth Survey of Techniques*, arXiv:2502.19732 [cs], Mar. 2025. DOI: 10.48550/arXiv.2502.19732 Accessed: Mar. 9, 2025. [Online]. Available: http://arxiv.org/abs/2502.19732

[29] B. Butler, S. Yu, A. Mazaheri, and A. Jannesari, "PipeInfer: Accelerating LLM Inference using Asynchronous Pipelined Speculation," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA: IEEE, Nov. 2024, pp. 1–19, ISBN: 979-8-3503-5291-7. DOI: 10.1109/SC41406.2024.00046 Accessed: Mar. 15, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10793190/

[30] X. Miao et al., "SpecInfer: Accelerating Generative Large Language Model Serving with Tree-based Speculative Inference and Verification," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, arXiv:2305.09781 [cs], Apr. 2024, pp. 932–949. DOI: 10.1145/3620666.3651335 Accessed: Mar. 15, 2025. [Online]. Available: http://arxiv.org/abs/2305.09781

[31] Z. Chen et al., "Sequoia: Scalable, Robust, and Hardware-aware Speculative Decoding," en, Feb. 2024. Accessed: Mar. 15, 2025.

[32] D. Xu et al., "EdgeLLM: Fast On-Device LLM Inference With Speculative Decoding," *IEEE Transactions on Mobile Computing*, vol. 24, no. 4, pp. 3256–3273, Apr. 2025, Conference Name: IEEE Transactions on Mobile Computing, ISSN: 1558-0660. DOI: 10.1109/TMC.2024.3513457 Accessed: Mar. 15, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10812936/?arnumber=10812936

[33] *TensorFlow*, en. Accessed: Oct. 29, 2025. [Online]. Available: https://www.tensorflow.org/

[34] J. Ansel et al., "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," en, in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, La Jolla CA USA: ACM, Apr. 2024, pp. 929–947, ISBN: 979-8-4007-0385-0. DOI: 10.1145/3620665.3640366 Accessed: Mar. 18, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3620665.3640366

[35] *ONNX | Home*. Accessed: Oct. 29, 2025. [Online]. Available: https://onnx.ai/

[36] H. Zhang, M. Xing, Y. Wu, and C. Zhao, "Compiler Technologies in Deep Learning Co-Design: A Survey," en, *Intelligent Computing*, vol. 2, p. 0040, Jan. 2023, ISSN: 2771-5892. DOI: 10.34133/icomputing.0040 Accessed: Apr. 3, 2024. [Online]. Available: https://spj.science.org/doi/10.34133/icomputing.0040

[37] M. Li et al., "The Deep Learning Compiler: A Comprehensive Survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, Mar. 2021, Number: 3 Conference Name: IEEE Transactions on Parallel and Distributed Systems, ISSN: 1558-2183. DOI: 10.1109/TPDS.2020.3030548 Accessed: May 9, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/9222299

[38] P. Barham and M. Isard, "Machine Learning Systems are Stuck in a Rut," en, in *Proceedings of the Workshop on Hot Topics in Operating Systems*, Bertinoro Italy: ACM, May 2019, pp. 177–183, ISBN: 978-1-4503-6727-1. DOI: 10.1145/3317550.3321441 Accessed: Aug. 10, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3317550.3321441

[39] K. De Bosschere et al., "High-Performance Embedded Architecture and Compilation Roadmap," en, in *Transactions on High-Performance Embedded Architectures and Compilers I*, P. Stenström, Ed., Berlin, Heidelberg: Springer, 2007, pp. 5–29, ISBN: 978-3-540-71528-3. DOI: 10.1007/978-3-540-71528-3_2

[40] M. O'Boyle, *Rethinking the Role of the Compiler in a Heterogeneous World*, en-US, Jul. 2020. Accessed: Sep. 10, 2025. [Online]. Available: https://www.sigarch.org/rethinking-the-role-of-the-compiler-in-a-heterogeneous-world/

[41] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017, _eprint: https://doi.org/10.1137/141000671. DOI: 10.1137/141000671 [Online]. Available: https://doi.org/10.1137/141000671

[42] C. Lattner et al., "MLIR: A Compiler Infrastructure for the End of Moore's Law," arXiv:2002.11054 [cs], arXiv, 2021. DOI: 10.48550/arXiv.2002.11054 Accessed: Sep. 19, 2025. [Online]. Available: http://arxiv.org/abs/2002.11054

[43] L. Foundation, *IREE*, 2025. Accessed: Sep. 11, 2025. [Online]. Available: https://iree.dev/

[44] A. K. Singh, M. Shafique, and A. Kumar, "Mapping on multi/many-core systems: Survey of current and emerging trends," en, 2013.

[45] Z. Chen et al., *Bring Your Own Codegen to Deep Learning Compiler*, arXiv:2105.03215, May 2021. Accessed: Oct. 13, 2024. [Online]. Available: http://arxiv.org/abs/2105.03215

[46] Z. Li, B. Yang, J. Li, T. Chen, X. Li, and M. Gao, "Adyna: Accelerating Dynamic Neural Networks with Adaptive Scheduling," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, ISSN: 2378-203X, Mar. 2025, pp. 549–562. DOI: 10.1109/HPCA61900.2025.00049 Accessed: Sep. 11, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10946301

[47] D. Xu et al., *Empowering 1000 tokens/second on-device LLM prefilling with mllm-NPU*, arXiv:2407.05858 [cs] version: 1, Jul. 2024. DOI: 10.48550/arXiv.2407.05858 Accessed: Sep. 12, 2025. [Online]. Available: http://arxiv.org/abs/2407.05858

[48] Y. Zhang et al., *Speculative Decoding Meets Quantization: Compatibility Evaluation and Hierarchical Framework Design*, en, arXiv:2505.22179 [cs], May 2025. DOI: 10.48550/arXiv.2505.22179 Accessed: Jun. 20, 2025. [Online]. Available: http://arxiv.org/abs/2505.22179

[49] K. Lv, H. Guo, Q. Guo, and X. Qiu, *DuoDecoding: Hardware-aware Heterogeneous Speculative Decoding with Dynamic Multi-Sequence Drafting*, arXiv:2503.00784 [cs], Mar. 2025. DOI: 10.48550/arXiv.2503.00784 Accessed: Aug. 5, 2025. [Online]. Available: http://arxiv.org/abs/2503.00784

[50] L. Zhang, Z. Zhang, B. Xu, S. Mei, and D. Li, "Dovetail: A CPU/GPU Heterogeneous Speculative Decoding for LLM inference," arXiv:2412.18934 [cs], arXiv, Dec. 2024. DOI: 10.48550/arXiv.2412.18934 Accessed: Mar. 15, 2025. [Online]. Available: http://arxiv.org/abs/2412.18934

[51] C. Li, F. Dahu, F. Erhu, Z. Rong, and W. Yingrui, "HeteroLLM: Accelerating Large Language Model Inference on Mobile SoCs with Heterogeneous AI Accelerators," 2024. Accessed: Aug. 11, 2025.

[52] H. Kwon, L. Lai, M. Pellauer, T. Krishna, Y.-H. Chen, and V. Chandra, "Heterogeneous Dataflow Accelerators for Multi-DNN Workloads," arXiv:1909.07437 [cs], arXiv, Dec. 2020. Accessed: Sep. 15, 2025.

[53] R. Baghdadi et al., *Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code*, arXiv:1804.10694 [cs], Dec. 2018. DOI: 10.48550/arXiv.1804.10694 Accessed: Sep. 12, 2025. [Online]. Available: http://arxiv.org/abs/1804.10694

[54] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, "ZigZag: A Memory-Centric Rapid DNN Accelerator Design Space Exploration Framework," en, *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1160–1174, Aug. 2020, Issue: arXiv:2007.11360 arXiv:2007.11360 [cs]. Accessed: Apr. 3, 2024. [Online]. Available: http://arxiv.org/abs/2007.11360

[55] R. Baghdadi et al., "A Deep Learning Based Cost Model for Automatic Code Optimization," arXiv:2104.04955 [cs], 2021. DOI: 10.48550/arXiv.2104.04955 Accessed: Jun. 18, 2024. [Online]. Available: http://arxiv.org/abs/2104.04955

[56] S. Na et al., "FlexInfer: Flexible LLM Inference with CPU Computations," en, 2025.

[57] S.-C. Kao and T. Krishna, "MAGMA: An Optimization Framework for Mapping Multiple DNNs on Multiple Accelerator Cores," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, ISSN: 2378-203X, Apr. 2022, pp. 814–830. DOI: 10.1109/HPCA53966.2022.00065 Accessed: Apr. 16, 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9773188?casa_token=zpijoZWAVgMAAAAA:mMwv3MuPFPV-smRcntT0chaqEA9VyHodCFXa-j7gSYphFMTelnVT-Z8CnzZxcefuOYaeLJ1xvg

[58] *Welcome to the ExecuTorch Documentation — ExecuTorch 0.7 documentation*, 2025. Accessed: Sep. 21, 2025. [Online]. Available: https://docs.pytorch.org/executorch/stable/index.html

[59] S. Abdulrasool et al., *Glow: Graph Lowering Compiler Techniques for Neural Networks*, en, 2018.

[60] T. Chen et al., *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*, arXiv:1802.04799 [cs], Oct. 2018. DOI: 10.48550/arXiv.1802.04799 Accessed: Jun. 18, 2024. [Online]. Available: http://arxiv.org/abs/1802.04799

[61] J. Ragan-Kelley, C. Barnes, and A. Adams, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," en, 2013.

[62] S. Alabed et al., "PartIR: Composing SPMD Partitioning Strategies for Machine Learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, arXiv:2401.11202 [cs], Mar. 2025, pp. 794–810. DOI: 10.1145/3669940.3707284 Accessed: Oct. 31, 2025. [Online]. Available: http://arxiv.org/abs/2401.11202

[63] X. Peng, X. Shi, H. Zhang, Y. Zhao, and X. Qian, *DawnPiper: A Memory-scablable Pipeline Parallel Training Framework*, arXiv:2505.05856 [cs], May 2025. DOI: 10.48550/arXiv.2505.05856 Accessed: Oct. 31, 2025. [Online]. Available: http://arxiv.org/abs/2505.05856

[64] S. Ahmadifarsani, D. Mueller-Gritschneder, and U. Schlichtmann, *A High-Level Compiler Integration Approach for Deep Learning Accelerators Supporting Abstraction and Optimization*, arXiv:2507.04828 [cs], Jul. 2025. DOI: 10.48550/arXiv.2507.04828 Accessed: Oct. 31, 2025. [Online]. Available: http://arxiv.org/abs/2507.04828

[65] Duke University, USA and S. Sagi, "Optimizing LLM Inference: Metrics that Matter for Real Time Applications," en, *Journal of Artificial Intelligence & Cloud Computing*, pp. 1–4, Feb. 2025, ISSN: 27546659. DOI: 10.47363/JAICC/2025(4)446 Accessed: Oct. 30, 2025. [Online]. Available: https://www.onlinescientificresearch.com/articles/optimizing-llm-inference-metrics-that-matter-for-real-time-applications.pdf

[66] *Cortex-A55*, en, 2025. Accessed: Sep. 21, 2025. [Online]. Available: https://developer.arm.com/Processors/Cortex-A55

[67] ARM, *Mali-G310*, en, 2025. Accessed: Sep. 11, 2025. [Online]. Available: https://developer.arm.com/Processors/Mali-G310

[68] J. Liu, B. Chen, and C. Zhang, *Speculative Prefill: Turbocharging TTFT with Lightweight and Training-Free Token Importance Estimation*, arXiv:2502.02789 [cs], May 2025. DOI: 10.48550/arXiv.2502.02789 Accessed: Oct. 30, 2025. [Online]. Available: http://arxiv.org/abs/2502.02789

[69] Y. Zhao et al., *From Quarter to All: Accelerating Speculative LLM Decoding via Floating-Point Exponent Remapping and Parameter Sharing*, arXiv:2510.18525 [cs], Oct. 2025. DOI: 10.48550/arXiv.2510.18525 Accessed: Oct. 27, 2025. [Online]. Available: http://arxiv.org/abs/2510.18525

[70] A. Grattafiori et al., *The Llama 3 Herd of Models*, arXiv:2407.21783 [cs], Nov. 2024. DOI: 10.48550/arXiv.2407.21783 Accessed: Oct. 30, 2025. [Online]. Available: http://arxiv.org/abs/2407.21783

[71] H. Xia et al., "Unlocking Efficiency in Large Language Model Inference: A Comprehensive Survey of Speculative Decoding," in *Findings of the Association for Computational Linguistics 2024*, arXiv:2401.07851 [cs], Jun. 2024. DOI: 10.48550/arXiv.2401.07851 Accessed: Oct. 13, 2025.

[72] *Intel/neural-compressor*, original-date: 2020-07-21T23:49:56Z, Oct. 2025. Accessed: Oct. 13, 2025. [Online]. Available: https://github.com/intel/neural-compressor

List of Figures

List of Tables

A Appendix - Modifications to IREE Compiler (v3.6.0.)

To enable full compilation of LLaMA 3.2 models for the Cortex-A55 CPU and Mali-G310 GPU targets, several modifications were introduced in IREE compiler version 3.6.0. For CPU code generation, changes were applied both to IREE and to the pinned LLVM/MLIR version; these are detailed in Tables A.1 and A.2. GPU compilation required additional minor adjustments, summarized in Table A.3. These modifications were not upstreamed; however, issues were opened to document potential failure points and propose directions for future fixes.

Table A.1: LLVMCPU code generation - IREE changes

Pipeline Affected	Change	Observation
CPUDataTiling Pipeline	Line 594 added: `funcPassManager.addPass (createCanonicalizerPass());`	Adding the canonicalization at this stage ensures that the pattern `DecomposeOuterUnit DimsPackOpPattern` in the Linalg transformation functions correctly.

```
// We check that the above condition is met, if not, we derivate another logic
if (!llvm::all_of(packOp.getInnerDimsPos(),
                  [&srcRank, &numTiles](int64_t dimPos) {
                    return dimPos >= (srcRank - numTiles);
                  })){

    auto it = llvm::find_if(packOp.getInnerDimsPos(),
                            [&srcRank, &numTiles](int64_t dimPos) {
                                return dimPos < (srcRank - numTiles);
                            });
    if (it != packOp.getInnerDimsPos().end()) {
    int64_t index = std::distance(packOp.getInnerDimsPos().begin(), it);
    // Here we assume that all the tiles sizes in tileSizes are 1 expect for one value
        .
    int countNotOne = llvm::count_if(tileSizes, [](OpFoldResult v) {
        auto cst = getConstantIntValue(v);
        return !(cst.has_value() && cst.value() == 1);
        });
    if (countNotOne != 1) {
```

```
        return rewriter.notifyMatchFailure(
            packOp, "more than one tile size is not 1");
    }

    transShapeForEmptyOp.assign(srcRank,
                            oneIdxAttr);
    auto nonOneTileSizeIt = llvm::find_if(tileSizes, [](OpFoldResult v) {
        auto cst = getConstantIntValue(v);
        return !(cst.has_value() && cst.value() == 1);
    });
    if (nonOneTileSizeIt != tileSizes.end()) {
        transShapeForEmptyOp[index] = *nonOneTileSizeIt;
    }
    }
} else {
    transShapeForEmptyOp.assign(srcRank - numTiles,
```

Listing A.1: Updated logic for DecomposeOuterUnitDimsPackOpPattern

```
// Use type converter to convert the type of the operation, for example vector<1xT>)
    is not available in SPIR-V, so the SPIRVTypeConverter will extract the element
    type T.
auto *converter = this->template getTypeConverter<SPIRVTypeConverter>();
Type dstType = converter->convertType(roundOp.getType());
if (!dstType)
    return rewriter.notifyMatchFailure(
        roundOp, "failed to convert type for math.round operation");
```

Listing A.2: Updated logic for DecomposeOuterUnitDimsPackOpPattern

```
Value operand = adaptor.getOperand();
Type ty = dstType;
```

Listing A.3: Updated logic for DecomposeOuterUnitDimsPackOpPattern

Table A.2: LLVMCPU code generation - MLIR changes

Pattern Affected	Change	Observation
DecomposeOuterUnit DimsPackOpPattern	Lines 1195-1197 and 1219 removed; lines 1225-1257 added (see Listing A.1)	The original pattern assumed all inner dimensions of `linalg::PackOp` were at the end of the source tensor (`dimPos >= (srcRank - numTiles)`), simplifying shape computation by setting outer dims to 1 and appending all tile sizes. The update introduces logic to handle cases where inner dimensions are not positioned at the end.

Table A.3: SPIR-V backend code generation - MLIR changes

Pattern Affected	**Change**	**Observation**
RoundOp Pattern (MathToSPIRV)	Lines 450-451 removed; lines 446-454 and 459–460 added (see Listings A.2 and A.3, respectively).	Fixes failures in quantized models during `ConvertToSPIRV`. Correctly converts `math.round` and `math.roundeven` using the SPIR-V type converter.

B Appendix - Tall and Skinny Matrix Multiplication

A substantial portion of inference latency in transformer-based models, (at least 88% for LLaMA 3.2 1B being executed at full precision on Mali-G310 GPU) is attributed to using general matrix multiplication operations. One relevant case corresponds to those involving skinny matrix multiplications, where the input matrix has significantly more columns than rows. During decode phase, this workload is prevalent particularly for single-batch, where the number of rows is 1. Nevertheless, our focus lies in the prefill phase, where such configurations arise primarily when the number of processed tokens is small. These cases pose challenges to conventional GPU tiling strategies, which rely on subdividing the output matrix into numerous tiles to maximize compute unit utilization.

The primary challenge posed by skinny GEMMs lies in their underutilization of the GPU. To address this, we apply a strategy that increases GPU usage by leveraging the associativity of matrix multiplication. Specifically, we decompose the operation along the shared dimension, commonly referred to as the K axis, into multiple smaller matrix multiplications that can be executed concurrently, as shown in Equation B.1, where S is the number of segments in which the K axis is split

$$c_{ij} = \sum_{k=1}^{K} a_{ik} b_{kj} = \left(\sum_{k=1}^{K/S} a_{ik} b_{kj} \right) + \left(\sum_{k=1+K/S}^{K} a_{ik} b_{kj} \right) \tag{B.1}$$

Each new smaller matrix multiplication independently utilizes the same number of compute units as the original operation, thereby multiplying overall utilization N. Although this approach introduces an additional summation step to aggregate partial results, the computational workload per compute unit is proportionally reduced. If the reduction in the workload per compute unit outweighs the cost of summing the intermediate outputs, the overall latency of the operation can be significantly improved.

Table B.1 summarizes the performance gains obtained by applying K-axis splitting to the reduction dimension of five matrix multiplications that collectively accounted for approximately 99.8% of the execution time prior to optimization. For each operation, we identified the optimal split factor N, which ranged between 2 and 8, yielding the highest acceleration. After splitting,

the shape of each matrix multiplication changes from $B \times N \times M \times K$ to $S \times N \times M \times \frac{K}{S}$, with the batch dimension reduced due to single-batch workloads. This approach resulted in speedups ranging from 17× to 67×,with an average improvement of approximately 50×. The split was implemented as a pass, shown in Listing B.1. We recommend to integrate in the future this optimization in the tensor encodings infrastructure, since it is not fully supported in IREE 3.6.0.

Table B.1: Execution time breakdown and speedup after applying k-split optimization across different batched matrix multiplication dispatches in the unquantized LLaMA 3.2 1B model.

Dispatch [B×N×M×K]	Segments	Counts	Exe. before Tiling [%]	Exe. after Tiling [%]	Speedup [×]
1xDx128256x2048	2	1	29.42	19.42	67
1xDx8192x2048	8	32	36.40	37.33	43
1xDx2048x8192	8	16	27.70	19.30	63
1xDx2048x2048	8	32	5.35	9.33	25
1xDx512x2048	8	32	0.91	2.31	17
Accumulated	-	-	99.77	87.68	50

```
struct LinalgBatchKSplit8Conversion :
    public OpRewritePattern<linalg::BatchMatmulOp> {
        using OpRewritePattern::OpRewritePattern;
        LogicalResult matchAndRewrite(linalg::BatchMatmulOp op,
                                      PatternRewriter &rewriter) const override {
          Location loc = op.getLoc();
          MLIRContext *ctx = getContext();

          // Get operand shapes
          Value lhs = op.getOperands()[0];
          Value rhs = op.getOperands()[1];
          Value output = op.getOutputs()[0];
          auto lhsType = llvm::dyn_cast<RankedTensorType>(lhs.getType());
          auto rhsType = llvm::dyn_cast<RankedTensorType>(rhs.getType());
          auto outputType = llvm::dyn_cast<RankedTensorType>(output.getType());

          if (!lhsType || !rhsType || !outputType)
            return rewriter.notifyMatchFailure(op, "invalid tensor types");

          // Verify rank of operators
          if (lhsType.getRank() != 3 || rhsType.getRank() != 3 || outputType.getRank()
              != 3)
            return rewriter.notifyMatchFailure(op, "invalid tensor rank, must be 3");

          auto lhsShape = lhsType.getShape();
          auto rhsShape = rhsType.getShape();

          // Define k-split
          const int kSplit = 8;
          // Get dimensions
          int64_t batchSize = lhsShape[0];
          int64_t N = lhsShape[1];
          int64_t K = lhsShape[2];
          int64_t M = rhsShape[2];

          if ( (M == 2048 && K == 8192) ) {
            return rewriter.notifyMatchFailure(op, "K dimension not supported");
          }

```

```
39          // Verify that sequence length (N) is dynamic and other shapes are static.
                We apply this pattern only to matmuls with static batch sizes and
                dynamic sequence lengths, since this was found to be beneficial for
                performance on MaliG310.
40          if (N != ShapedType::kDynamic || lhsShape[0] == ShapedType::kDynamic ||
41              lhsShape[2] == ShapedType::kDynamic || rhsShape[0] == ShapedType::
                    kDynamic ||
42              rhsShape[1] == ShapedType::kDynamic || rhsShape[2] == ShapedType::
                    kDynamic)
43            return rewriter.notifyMatchFailure(op, "invalid tensor dimensions");
44
45          // Verify that K is divisible by k-split
46          if (K % kSplit != 0)
47            return rewriter.notifyMatchFailure(op, "K dimension not divisible by 8");
48
49          int64_t tileSize = K / kSplit;
50          Type elementType = lhsType.getElementType();
51
52          // Get dynamic dimension value for empty create constant for filled ops
53          Value dimN = rewriter.create<tensor::DimOp>(loc, lhs, 1);
54          Value fillConstant = rewriter.create<arith::ConstantOp>(
55              loc, rewriter.getZeroAttr(elementType));
56
57          // Create empty and filled ops for pack ops
58          SmallVector<int64_t> packedLhsShape = {batchSize, kSplit, ShapedType::
                kDynamic, tileSize};
59          SmallVector<int64_t> packedRhsShape = {batchSize, kSplit, M, tileSize};
60
61
62          Value emptyPackedLhs = rewriter.create<tensor::EmptyOp>(
63              loc, packedLhsShape, elementType, ValueRange{dimN});
64          Value emptyPackedRhs = rewriter.create<tensor::EmptyOp>(
65              loc, packedRhsShape, elementType);
66
67          Value filledPackedLhs = rewriter.create<linalg::FillOp>(
68              loc, fillConstant, emptyPackedLhs).getResult(0);
69          Value filledPackedRhs = rewriter.create<linalg::FillOp>(
70              loc, fillConstant, emptyPackedRhs).getResult(0);
71
72          // Create empty and filled ops for tiled matmul
73          SmallVector<int64_t> outputMatmulShape = {batchSize, kSplit, ShapedType::
                kDynamic, M};
74          auto outputMatmulType = RankedTensorType::get(outputMatmulShape, elementType
                );
75
76          Value outputInit = rewriter.create<tensor::EmptyOp>(
77              loc, outputMatmulShape, elementType, ValueRange{dimN});
78          Value outputFilled = rewriter.create<linalg::FillOp>(
79              loc, fillConstant, outputInit).getResult(0);
80
81          // Create empty and filled ops for reduction
82          SmallVector<int64_t> outputReducedShape = {batchSize, ShapedType::kDynamic,
                M};
83          auto outputReducedType = RankedTensorType::get(outputReducedShape,
                elementType);
84
85          Value outputInitReduced = rewriter.create<tensor::EmptyOp>(
86              loc, outputReducedShape, elementType, ValueRange{dimN});
87          Value outputFilledReduced = rewriter.create<linalg::FillOp>(
88              loc, fillConstant, outputInitReduced).getResult(0);
89
90          // Create pack ops
91          SmallVector<int64_t> outerDimPermLhs = {0, 2, 1};
92          SmallVector<int64_t> innerDimPosLhs = {2};
93          SmallVector<int64_t> innerDimPosRhs = {1};
94          SmallVector<OpFoldResult> innerTiles;
95          innerTiles.push_back(rewriter.getIndexAttr(tileSize));
```

```
96
97          Value packLhs = rewriter.create<linalg::PackOp>(
98              loc, lhs, filledPackedLhs, innerDimPosLhs, innerTiles,
99              std::nullopt, outerDimPermLhs);
100
101         Value packRhs = rewriter.create<linalg::PackOp>(
102             loc, rhs, filledPackedRhs, innerDimPosRhs, innerTiles);
103
104         // Create affine maps
105         // #mapA = affine_map<(batch, d0, d1, d2, d3) -> (batch, d0, d1, d3)>
106         // #mapB = affine_map<(batch, d0, d1, d2, d3) -> (batch, d0, d2, d3)>
107         // #mapC = affine_map<(batch, d0, d1, d2, d3) -> (batch, d0, d1, d2)>
108         // #mapD = affine_map<(batch, d0, d1, d2) -> (batch, d0, d1, d2)>
109         // #mapE = affine_map<(batch, d0, d1, d2) -> (batch, d1, d2)>
110         auto mapA = AffineMap::get(5, 0, {
111             rewriter.getAffineDimExpr(0), rewriter.getAffineDimExpr(1),
112             rewriter.getAffineDimExpr(2), rewriter.getAffineDimExpr(4)}, ctx);
113         auto mapB = AffineMap::get(5, 0, {
114             rewriter.getAffineDimExpr(0), rewriter.getAffineDimExpr(1),
115             rewriter.getAffineDimExpr(3), rewriter.getAffineDimExpr(4)}, ctx);
116         auto mapC = AffineMap::get(5, 0, {
117             rewriter.getAffineDimExpr(0), rewriter.getAffineDimExpr(1),
118             rewriter.getAffineDimExpr(2), rewriter.getAffineDimExpr(3)}, ctx);
119         auto mapD = AffineMap::get(4, 0, {
120             rewriter.getAffineDimExpr(0), rewriter.getAffineDimExpr(1),
121             rewriter.getAffineDimExpr(2), rewriter.getAffineDimExpr(3)}, ctx);
122         auto mapE = AffineMap::get(4, 0, {
123             rewriter.getAffineDimExpr(0), rewriter.getAffineDimExpr(2),
124             rewriter.getAffineDimExpr(3)}, ctx);
125         // Create tiled matmul
126         SmallVector<utils::IteratorType> iteratorTypes = {
127             utils::IteratorType::parallel, utils::IteratorType::parallel,
128             utils::IteratorType::parallel, utils::IteratorType::parallel,
129             utils::IteratorType::reduction};
130         Value tiledMatmul = rewriter.create<linalg::GenericOp>(
131             loc, outputMatmulType, ValueRange{packLhs, packRhs},
132             ValueRange{outputFilled}, ArrayRef<AffineMap>{mapA, mapB, mapC},
133             iteratorTypes,
134             [&](OpBuilder &b, Location loc, ValueRange args) {
135               Value mul = b.create<arith::MulFOp>(loc, args[0], args[1]);
136               Value add = b.create<arith::AddFOp>(loc, args[2], mul);
137               b.create<linalg::YieldOp>(loc, add);
138             }).getResult(0);
139         SmallVector<utils::IteratorType> reductionIteratorTypes = {
140             utils::IteratorType::parallel, utils::IteratorType::reduction,
141             utils::IteratorType::parallel, utils::IteratorType::parallel};
142         // Create reduction
143         Value result = rewriter.create<linalg::GenericOp>(
144             loc, outputReducedType, ValueRange{tiledMatmul},
145             ValueRange{outputFilledReduced}, ArrayRef<AffineMap>{mapD, mapE},
146             reductionIteratorTypes,
147             [&](OpBuilder &b, Location loc, ValueRange args) {
148               Value add = b.create<arith::AddFOp>(loc, args[1], args[0]);
149               b.create<linalg::YieldOp>(loc, add);
150             }).getResult(0);
151
152         rewriter.replaceOp(op, result);
153         return success();
154       }
155   };
```

Listing B.1: Implementation of the k-split tiling for Mali-G310 as a pass in the preprocessing phase of the compilation