Title

Full Stack Approach for Efficient Deep Learning Inference

Permalink

https://escholarship.org/uc/item/4wf834q8

Author

Kim, Sehoon

Publication Date

2024

Full Stack Approach for Efficient Deep Learning Inference

By

Sehoon Kim

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kurt Keutzer, Chair
Professor Jiantao Jiao
Professor Gopala Anumanchipalli
Professor Song Han

Fall 2024

Full Stack Approach for Efficient Deep Learning Inference

Abstract

Full Stack Approach for Efficient Deep Learning Inference

by

Sehoon Kim

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Kurt Keutzer, Chair

Recent advancements in AI technologies have led to unprecedented growth in model sizes, particularly with the advent of large language models (LLMs). While these models have shown great capabilities in various domains, their exponential scaling has introduced significant inference-time overheads, such as increased memory requirements, latency, and computational costs, thereby making efficient deployment and serving challenging. This thesis addresses these challenges through a *full-stack* approach that enhances efficiency across four key components of the AI inference stack: model optimization, inference methods, model architectures, and applications.

For model optimization, we introduce quantization techniques to optimize inference-time compute and memory requirements. I-BERT optimizes compute by leveraging integer-only quantization, which achieves up to a 3.5× latency speedup and enables deployment of the Transformer architectures on integer-only hardware. SqueezeLLM, which employs extremely low-bit weight quantization, effectively reduces memory requirements without sacrificing accuracy during LLM inference. For enhanced inference methods, we present the Big Little Decoder, a speculative decoding framework that accelerates autoregressive LLM inference by up to 2× through a collaboration between small and large models. Regarding model architectures, we propose an efficient design for speech recognition using a Temporal U-Net structure, which improves inference efficiency by shortening input sequence lengths. Finally, at the application level, we introduce `LLMCompiler`, a framework for efficiently orchestrating multiple function calls in LLM-based applications, which reduces execution latency and costs while enhancing robustness by decomposing complex user inputs into smaller, easier tasks. Collectively, these contributions provide a full-stack strategy for optimizing AI model inference from low-level systems to high-level applications to enable the efficient deployment and serving of state-of-the-art AI solutions.

To my family, whose unwavering love, support, and belief in me have been the foundation of this journey.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude to my advisor, Professor Kurt Keutzer, for his invaluable advice and guidance throughout my Ph.D. journey. His support extended far beyond academic advice — his insights on entrepreneurial mindset, ability to interpret industrial trends, and effective communication and presentation skills have profoundly influenced not only my academic career but also how I navigate life. After 4.5 years of working under his advice, I learned how to think critically, strategize effectively, and present my work in the most impactful way. I feel fortunate to have been his student and am deeply appreciative of his unwavering support to become an individual researcher.

I would also like to extend my appreciation to Amir Gholami, who was my closest collaborator throughout my research journey. Amir played a pivotal role in helping me establish myself in the research world, offering invaluable technical guidance and engaging in countless in-depth discussions. Beyond technical insights, he taught me invaluable lessons on team management and collaboration, which will serve as a solid foundation as I continue to grow as a researcher and leader in the future.

Next, I would like to acknowledge Professor Jiantao Jiao, Professor Gopala Anumanchipalli, and Professor Song Han for serving on both my qualifying exam and dissertation committee to provide constructive feedback and valuable advice on shaping my research directions. Additionally, I would like to acknowledge Professor Michael Mahoney and Professor Sophia Shao for their detailed guidance and insights on various projects throughout my Ph.D. journey.

I was very fortunate to have had the opportunity to work with many talented collaborators at Berkeley, from whom I have learned a lot. I would like to thank Coleman Hooper, Suhong Moon, and Nicholas Lee for our close collaborations on various projects. Working with them significantly expanded the scope of my research, allowing me to build deeper insights into hardware, algorithms, and many other domains. I would also like to acknowledge Zhewei Yao, Zhen Dong, Sheng Shen, and Xiuyu Li for our exciting collaborative works on efficient ML research. My special thanks go to Woosuk Kwon for his consistently insightful feedback on ML systems research. I would also like to acknowledge Karttikeya Mangalam for his remarkable insights on model training. Without his input, my first model training project, Squeezeformer, would not have reached its completion. I would like to appreciate Josh Minwoo Kang, Jenny Hwang, and Hasan Genc for our collaboration on hardware-software co-design. Finally, I enjoyed the process of brainstorming and building research ideas together with Rishabh Tiwari and Haocheng Xi, even during our brief overlap. I would also like to acknowledge the privilege of mentoring and collaborating with a group of extraordinary students: Lutfi Eren Erdogan, Ryan Tabrizi, Sid Jha, Monishwaran Maheswaran, Hiva Mohammadzadeh, Thanakul Wattanawong, Sean Lin, Aditya Tomar, Kerem Dilmen, Sebastian Zhao, and many others. I greatly enjoyed co-developing ideas and exploring directions together.

Moreover, I am deeply thankful to my mentors and collaborators from the industry who have greatly enriched my Ph.D. journey. I would like to acknowledge the Narada team, with whom I explored the dynamics of entrepreneurship: David Park, Amir Gholami, Zizheng Tai, Thanakul Wattanawong, Nicholas Lee, Lutfi Eren Erdogan, and Amogh Tantradi. I am also grateful to Mohammad Shoeybi and Vijay Anand Korthikanti, who mentored me during my internship at NVIDIA, offering invaluable guidance and support. Additionally, I was honored to have worked closely with Amir Yazdanbakhsh and Suvinay Subramanian from Google, Maxwell Horton and Mahyar Najibi from Apple, June Paik from Furiosa AI, and Woosang Lim from POSCO Holdings. Lastly, I sincerely acknowledge the generous support of the Korea Foundation for Advanced Studies (KFAS) for the unwavering encouragement and assistance throughout my journey.

Last but not least, I would like to express my gratitude to my family. Their unwavering love, support, and belief in me have been the foundation of this journey. Their constant encouragement and support were the greatest driving force that enabled me to persevere through challenges and complete this journey.

Chapter 1

Introduction

AI technologies have made unprecedented advances across a wide range of domains including natural language processing, computer vision, and speech recognition. However, the prevalent strategy of scaling mode sizes has introduced substantial inference-time overheads, leading to challenges in deploying and serving state-of-the-art models efficiently. For example, as shown in Figure 1.1, since the introduction of the Transformer architecture [266] in 2017 with 65M parameters, model sizes have grown exponentially – by 410× every two years – opening up the era of large language models (LLMs), highlighted by GPT-3 with 175B parameters and other billion-scale models. This growth has far outpaced GPU memory scaling, which has only doubled every two years. Consequently, the expansion in model size has not only resulted in significant memory requirements, often exceeding the capacity of a single GPU, but also introduced challenges in latency, power efficiency, and the computational costs of running these large models.

To address this issue and reduce the runtime overhead of AI solutions, *full-stack* optimization across the AI inference stack is essential. As shown in Figure 1.2, this thesis will cover improving efficiency across four key components at different hierarchies of the inference stack: model optimization, inference methods, model architecture, and application. These span from the hardware-facing layer at the bottom to the user-facing layer at the top, addressing efficiency holistically from low-level systems to high-level applications.

Model Optimization. Model optimization is a key approach for deploying models efficiently by reducing their size and enabling more efficient use of underlying hardware resources such as compute and memory. Common techniques include quantization, which compresses model weights and activations by using lower-bit precision (e.g., 8-bit) instead of standard 32-bit or 16-bit floating-point (i.e., FP32 or FP16), and pruning, which removes less important weights from the model. These methods, often applied after model architecture design and training are completed, enable models to maintain comparable accuracy while significantly reducing computational and memory requirements, making them more suitable on resource-constrained environments.

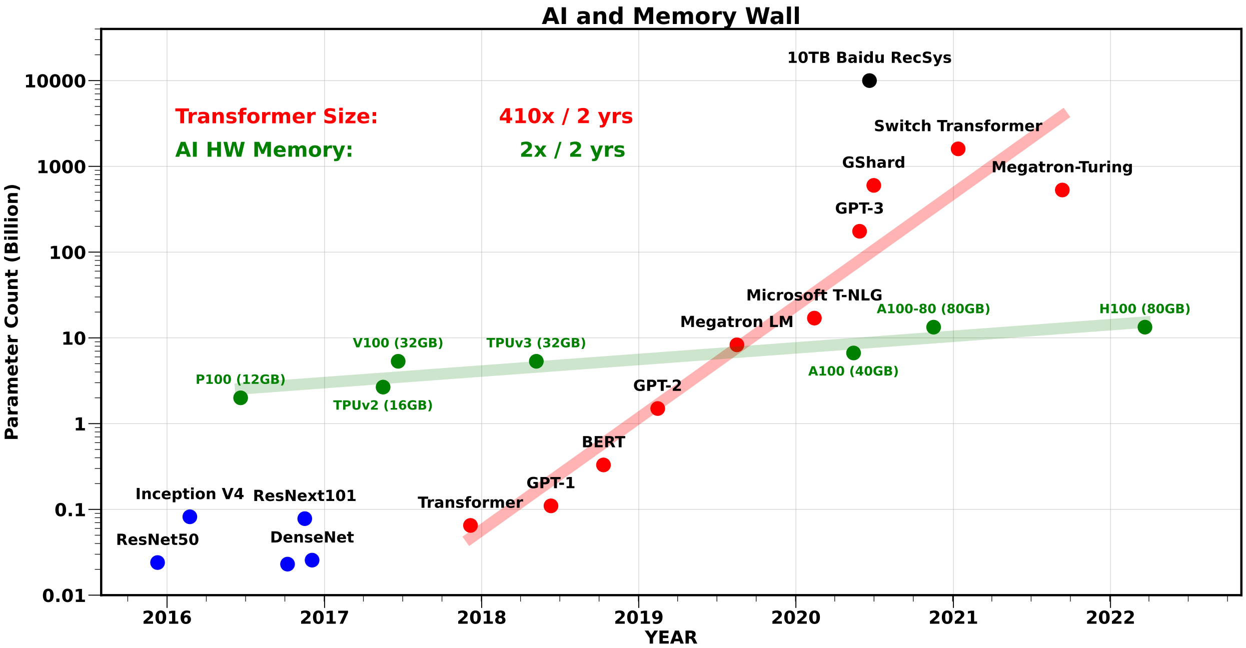


Figure 1.1: The evolution of the number of parameters of state-of-the-art models over the years, along with the AI accelerator memory capacity. The number of parameters in Transformer models has been exponentially increasing by the rate of 410× every two years, while the single GPU memory has only been scaled at a rate of 2× every two years.

This thesis introduces quantization techniques aimed at improving both compute and memory efficiency during Transformer inference. In Chapter 2, we present I-BERT, a method that enhances *compute efficiency* by leveraging integer-only quantization. By conducting the entire inference process with integer arithmetic, I-BERT not only achieves up to a 3.5× speedup in latency, but also enables the deployment of Transformer models on integer-only hardware. Chapter 3 introduces SqueezeLLM, a quantization technique that optimizes *memory efficiency* during LLM inference via extremely low-bit weight quantization. Given that memory operations often become the major bottleneck in autoregressive generation tasks of LLMs, SqueezeLLM offers a precise quantization strategy that preserves the underlying weight distributions with reduced bitwidths (e.g. 3 or 4 bits), significantly lowering memory requirements without sacrificing model accuracy.

Inference Methods. To efficiently serve large-scale models, it is also essential to understand their inference dynamics in order to minimize redundant operations and maximize resource utilization. In Chapter 4, we introduce Big Little Decoder (BiLD), a speculative decoding framework designed to address the inefficiencies of memory operations during autoregressive inference of LLMs. Autoregressive generation is often memory-bound, as each token generation requires a costly memory operation to load a large weight matrix. Therefore, it is essential to reduce run-time memory traffic in order to improve inference efficiency.

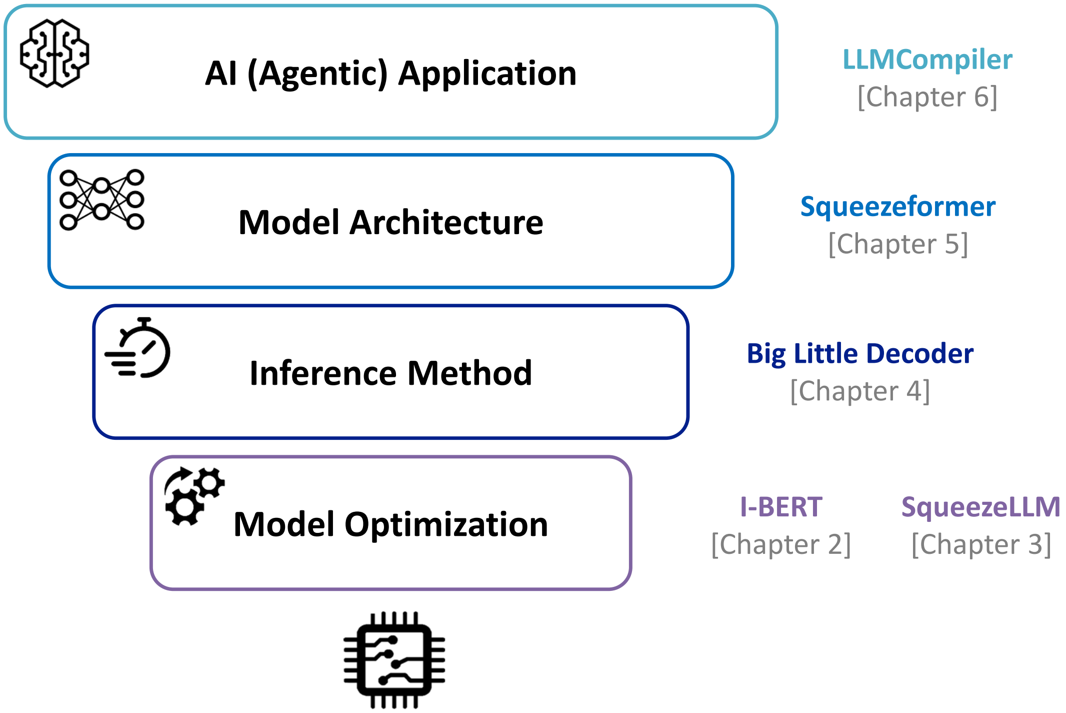


Figure 1.2: A full-stack perspective on improving the efficiency of AI solutions, spanning from low-level systems to high-level applications. From the bottom to the top in this figure, this thesis covers efficiency improvement across four key components at different layers of the inference stack: model optimization (Chapter 2 and 3), inference methods (Chapter 4), model architectures (Chapter 5), and applications (Chapter 5).

BiLD tackles this challenge by leveraging collaboration between small and large models – the smaller model quickly generates multiple tokens, while the larger model intermittently reviews and refines the small model's predictions. This approach allows the large model to perform non-autoregressive execution to process multiple tokens within a single iteration, resulting in a 2× inference speedup without compromising the quality of the generated output.

Model Architectures. Post-training methods for enhancing efficiency, such as model optimization and better inference methods, have gained popularity due to their flexibility to be applied after model design and training; however, further efficiency gains often require developing new model architectures tailored to specific domains. A key factor in this process is the use of *inductive bias*, which plays a critical role in guiding model design. Inductive bias [185] refers to the assumptions made by a learning algorithm that allow it to generalize from finite training data to a general model of the domain. For instance, convolutional neural networks (CNNs) use locality as an inductive bias for image-based tasks in computer vision, demonstrating how domain-specific inductive biases can inform better architecture design.

Transformers have demonstrated impressive performance when provided with a large amount of data even with their minimal inductive bias. However, this approach may be less effective for smaller models or in relatively data-scarce domains. In these scenarios, designing architectures with strong, domain-specific inductive biases can lead to more efficient and effective model performance, especially when data or computational resources are limited. To this end, in Chapter 5, we introduce a more compact architecture for speech recognition. By focusing on the redundancies along the temporal axis of continuous speech signals, we propose a Temporal U-Net structure that significantly enhances efficiency by effectively shortening the input sequence length. This design leads to a more accurate speech recognition model within a fixed resource budget, enhancing both performance and efficiency.

AI Applications. Recent advancements in the reasoning capabilities of LLMs have expanded their potential beyond content generation, enabling them to solve more complex problems. A key factor behind this expanded problem-solving ability is their function (or tool) calling capability, which allows LLMs to invoke external functions and integrate their outputs to assist in task completion. This ability to integrate function calls has enabled a paradigm shift in how LLM-based applications are developed, leading to the rise of *agentic applications*. In these applications, LLMs actively interact with their environments by taking actions and gathering information through external functions, enabling them to autonomously complete user tasks.

As a consequence, to improve the efficiency of these LLM-based applications, it is not enough to focus solely on optimizing the efficiency of a single model – whether through model optimization, better inference methods, or more efficient model architectures. It is equally important to enhance the efficiency of the dynamic interactions between LLMs and their external functions for building more efficient, scalable, and responsive agentic applications. In Chapter 6, we introduce `LLMCompiler` that efficiently orchestrates multiple function calls by decomposing user inputs into executable tasks and their interdependencies. `LLMCompiler` significantly reduces execution latency and costs by running independent tasks in parallel, while also enhancing the robustness of complex tasks by breaking down user inputs into smaller, manageable tasks. This approach takes a step toward building more efficient and scalable agentic applications that are capable of handling increasingly sophisticated workflows.

Chapter 2

Compute Optimization: Integer-only Transformer Quantization

2.1 Introduction

The recent Transformer based Neural Network (NN) models [266], pre-trained from large unlabeled data (e.g., BERT [43], RoBERTa [171], and the GPT family [213, 214, 12]), have achieved a significant accuracy improvement when fine-tuned on a wide range of Natural Language Processing (NLP) tasks such as sentence classification [269] and question answering [218]. Despite the state-of-the-art results in various NLP tasks, pre-trained Transformer models are generally orders of magnitude larger than prior models. For example, the BERT-Large model [43] contains 340M parameters. Much larger Transformer models have been introduced in the past few years, with even more parameters [214, 12, 240, 223, 305, 153, 215]. Efficient deployment of these models has become a major challenge, even in data centers, due to limited resources (energy, memory footprint, and compute) and the need for real-time inference. Obviously, these challenges are greater for edge devices, where the compute and energy resources are more constrained.

One promising method to tackle this challenge is quantization [142, 48, 292, 320, 293, 119], a procedure that compresses NN models into smaller sizes by representing parameters and/or activations with low bit precision, e.g., 8-bit integer (INT8) instead of 32-bit floating point (FP32). Quantization reduces memory footprint by storing parameters/activations in low precision. With the recent integer-only quantization methods, one can also benefit from faster inference speed by using low-precision integer multiplication and accumulation, instead of floating-point arithmetic. However, previous quantization schemes for Transformer based models use simulated quantization (aka fake quantization), where all or part of operations in the inference (e.g., GELU [95], Softmax, and Layer Normalization [3]) are carried out with floating point arithmetic [236, 318, 8] (Figure 2.1 Left and Middle). This approach has multiple drawbacks for deployment in real edge application scenarios. Most importantly, the resulting NN models cannot be deployed on neural accelerators or popular edge processors

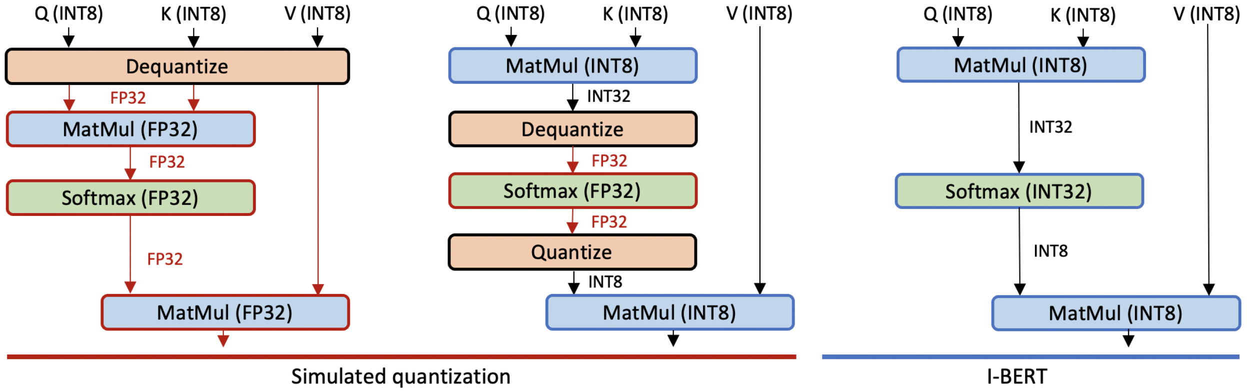


Figure 2.1: Comparison of different quantization schemes applied to the self-attention layer in the Transformer architecture. (Left) Simulated quantization, where all operations are performed with floating point arithmetic. Parameters are quantized and stored as integer, but they are dequantized into floating-point for inference. (Middle) Simulated quantization, where only a part of operations are performed with integer arithmetic. Because the Softmax in this figure is performed with floating point arithmetic, the input to the Softmax should be dequantized; and the output from the Softmax should be quantized back into integer to perform the subsequent integer MatMul. (Right) The integer-only quantization that we propose. There is neither floating point arithmetic nor dequantization during the entire inference.

that do not support floating-point arithmetic. For instance, the recent server class of Turing Tensor Cores has added high throughput integer logic that is faster than single/half-precision. Similarly, some of the edge processor cores in ARM Cortex-M [1] family for embedded systems only contain integer arithmetic units, and they can only support NN deployment with the integer-only kernels [148]. Moreover, one has to consider that compared to the integer-only inference, the approaches that use floating point arithmetic are inferior in latency and power efficiency. For chip designers wishing to support BERT-like models, adding floating point arithmetic logic occupies larger die area on a chip, as compared to integer arithmetic logic. Thus, the complete removal of floating point arithmetic for inference could have a major impact on designing applications, software, and hardware for efficient inference at the edge [1] (Figure 2.1 Right).

While prior work has shown the feasibility of integer-only inference [119, 310], these approaches have only focused on models in computer vision with simple CNN layers, Batch Normalization (BatchNorm) [117], and ReLU activations. These are all linear or piece-wise linear operators. Due to the non-linear operations used in Transformer architecture, e.g., GELU, Softmax, and Layer Normalization (LayerNorm), these methods cannot be applied to Transformer based models. Unlike ReLU, computing GELU and Softmax with integer-only arithmetic is not straightforward, due to their non-linearity. Furthermore, unlike BatchNorm whose parameters/statistics can be fused into the previous convolutional layer in inference, LayerNorm requires the dynamic computation of the square root of the variance for each input. This cannot be naïvely computed with integer-only arithmetic. Another challenge is that processing GELU, Softmax, and LayerNorm with low precision can result in significant accuracy degradation [318, 8]. For these reasons, other quantization methods such as [318, 236, 8] keep these operations in FP32 precision.

In this work, we propose I-BERT to address these challenges. I-BERT incorporates a series of novel integer-only quantization schemes for Transformer based models. Specifically, our contributions are:

- We propose new kernels for the efficient and accurate integer-only computation of GELU and Softmax. In particular, we approximate GELU and Softmax with lightweight second-order polynomials, which can be evaluated with integer-only arithmetic. We utilize different techniques to improve the approximation error, and achieve a maximum error of 1.8×10^{-2} for GELU, and 1.9×10^{-3} for Softmax. See Section 2.2 and 2.2 for details.
- For LayerNorm, we perform integer-only computation by leveraging a known algorithm for integer calculation of square root [34]. See Section 2.3 for details.
- We use these approximations of GELU, Softmax, and LayerNorm to design integer-only quantization for Transformer based models. Specifically, we process Embedding and matrix multiplication (MatMul) with INT8 multiplication and INT32 accumulation. The following non-linear operations (GELU, Softmax, and LayerNorm) are then calculated on the INT32 accumulated result and then requantized back to INT8. We represent all parameters and activations in the entire computational graph with integers, and we never cast them into floating point. See Figure 2.1 (Right) for a schematic description.
- We apply I-BERT to RoBERTa-Base/Large, and we evaluate their accuracy on the GLUE [269] downstream tasks. I-BERT achieves similar results as compared to a full-precision baseline. Specifically, I-BERT outperforms the baseline by 0.3 and 0.5 on the GLUE downstream tasks for RoBERTa-Base and RoBERTa-Large, respectively. See Table 2.2 in Section 2.3 for details.
- We deploy INT8 BERT models with the integer-only kernels for non-linear operations on a T4 GPU using TensorRT [193]. We show that INT8 inference achieves up to 4× speedup as compared to FP32 inference. See Table 2.3 in Section 2.3 for details.

2.2 Methodology

Basic Quantization Method

Under *uniform symmetric quantization* scheme, a real number x is uniformly mapped to an integer value $q \in [-2^{b-1}, 2^{b-1} - 1]$, where b specifies the quantization bit precision. The formal definition is:

$$q = \mathrm{Q}(x, b, S) = \mathrm{Int}\left(\frac{\mathrm{clip}(x, -\alpha, \alpha)}{S}\right), \tag{2.1}$$

where Q is the quantization operator, Int is the integer map (e.g., round to the nearest integer), clip is the truncation function, α is the clipping parameter used to control the outliers, and S is the scaling factor defined as $\alpha/(2^{b-1}-1)$. The reverse mapping from the quantized values q to the real values (aka dequantization) is:

$$\tilde{x} = \mathrm{DQ}(q, S) = Sq \approx x, \tag{2.2}$$

where DQ denotes the dequantization operator. This approach is referred to as uniform symmetric quantization. It is *uniform* because the spacing between quantized values and their corresponding mapping to real values is constant. However, several different non-uniform quantization methods have also been proposed [293, 320, 28, 203]. While non-uniform quantization approaches may better capture the distribution of parameters/activations than uniform quantization, they are in general difficult to deploy on hardware (as they often require a look-up table which results in overhead). Thus, we focus only on uniform quantization in this work. In addition, this approach is *symmetric* because we clip the values symmetrically within a range $[-\alpha, \alpha]$; while in asymmetric quantization, the left and right sides of this range could be asymmetric/different. Finally, we use *static quantization* where all the scaling factors S are fixed during inference to avoid the runtime overhead of computing them. See Section A.1 for more details on quantization methods.

Non-linear Functions with Integer-only Arithmetic

The key to integer-only quantization is to perform all operations with integer arithmetic without using any floating point calculation. Unlike linear (e.g., MatMul) or piece-wise linear operations (e.g., ReLU), this is not straightforward for non-linear operations (e.g., GELU, Softmax, and LayerNorm). This is because the integer-only quantization algorithms in previous works [310, 119] rely on the linear property of the operator. For example, MatMul(Sq) is equivalent to $S \cdot$ MatMul(q) for the linear MatMul operation. This property allows us to apply integer MatMul to the quantized input q and then multiply the scaling factor S to obtain the same result as applying floating point MatMul to the dequantized input Sq. Importantly, this property *does not* hold for non-linear operations, e.g., $\mathrm{GELU}(Sq) \neq S \cdot \mathrm{GELU}(q)$. One naïve solution is to compute the results of these operations and store them in a look up table [148]. However, such an approach can have overhead when deployed on chips with limited on-chip memory, and will create a bottleneck proportional to how fast the look up table could be performed. Another solution is to dequantize the activations and convert them to floating point, and then compute these non-linear operations with single precision logic [318, 8]. However, this approach is not integer-only and cannot be used on specialized efficient hardware that does not support floating point arithmetic, e.g., ARM Cortex-M [1].

To address this challenge, we approximate non-linear activation functions, GELU and Softmax, with polynomials that can be computed with integer-only arithmetic. Computing polynomials consists of only addition and multiplication, which can be performed with integer arithmetic. As such, if we can find good polynomial approximations to these operations, then

Algorithm 1 Integer-only Computation of Second-order Polynomial $a(x+b)^2+c$

```
Input: $q, S$: quantized input and scaling factor
Output: $q_{out}, S_{out}$: quantized output and scaling factor
function I-POLY($q, S$)                                  ▷ $qS = x$
    $q_b \leftarrow \lfloor b/S \rfloor$
    $q_c \leftarrow \lfloor c/aS^2 \rfloor$
    $S_{out} \leftarrow \lfloor aS^2 \rfloor$
    $q_{out} \leftarrow (q+q_b)^2 + q_c$
    return $q_{out}, S_{out}$                            ▷ $q_{out}S_{out} \approx a(x+b)^2+c$
end function
```

we can perform the entire inference with integer-only arithmetic. For instance, a second-order polynomial represented as $a(x+b)^2+c$ can be efficiently calculated with integer-only arithmetic as shown in Algorithm 1.[1]

Polynomial Approximation of Non-linear Functions

There is a large body of work on approximating a function with a polynomial [251]. We use a class of *interpolating polynomials*, where we are given the function value for a set of $n+1$ different data points $\{(x_0, f_0), \ldots, (x_n, f_n)\}$, and we seek to find a polynomial of degree at most n that exactly matches the function value at these points. It is known that there exists a unique polynomial of degree at most n that passes through all the data points [281]. We denote this polynomial by L, defined as:

$$L(x) = \sum_{i=0}^{n} f_i l_i(x) \text{ where } l_i(x) = \prod_{\substack{0 \le j \le n \\ j \ne i}} \frac{x - x_j}{x_i - x_j}. \tag{2.3}$$

Interestingly for our problem, we have two knobs to change to find the best polynomial approximation. Since we know the actual target function and can query its exact value for any input, we can choose the interpolating point (x_i, f_i) to be any point on the function. The second knob is to choose the degree of the polynomial. While choosing a high-order polynomial results in smaller error (see Appendix A.2), there are two problems with this. First, high-order polynomials have higher computational and memory overhead. Second, it is challenging to evaluate them with low-precision integer-only arithmetic, as overflow can happen when multiplying integer values. For every multiplication, we need to use double bit-precision to avoid overflow. As such, the challenge is to find a good low-order polynomial that can closely approximate the non-linear functions used in Transformers. This is what we discuss next, for GELU and Softmax, in Section 2.2 and 2.2, respectively, where we show that one can get a close approximation by using only a second-order polynomial.

[1] In Algorithm 1, $\lfloor \cdot \rfloor$ means the floor function. Note that, q_b, q_c, and S_{out} can be pre-computed under static quantization. That is to say, there is no floating point calculation, e.g., of S/b, in inference.

Integer-only GELU

GELU [95] is a non-linear activation function used in Transformer models, defined as:

$$\begin{aligned} \mathrm{GELU}(x) &:= x \cdot \frac{1}{2}\left[1 + \mathrm{erf}(\frac{x}{\sqrt{2}})\right], \\ \text{where } \mathrm{erf}(x) &:= \frac{2}{\sqrt{\pi}} \int_0^x \exp{(-t^2)} dt. \end{aligned} \tag{2.4}$$

Here, erf is the error function. Figure 2.2 shows the behaviour of the GELU function (shown in red). GELU has a similar behaviour as ReLU (shown in green) in the limit of large positive/negative values, but it behaves differently near zero. Direct evaluation of the integration term in erf is not computationally efficient. For this reason, several different approximations have been proposed for evaluating GELU. For example, [95] suggests using Sigmoid to approximate erf:

$$\mathrm{GELU}(x) \approx x\sigma(1.702x), \tag{2.5}$$

where $\sigma(\cdot)$ is the Sigmoid function. This approximation, however, is not a viable solution for integer-only quantization, as the Sigmoid itself is another non-linear function which requires floating point arithmetic. One way to address this is to approximate Sigmoid with the so-called hard Sigmoid (h-Sigmoid) proposed by [106] (designed in the context of efficient computer vision models) to obtain an integer-only approximation for GELU:

$$\text{h-GELU}(x) := x\frac{\mathrm{ReLU6}(1.702x + 3)}{6} \approx \mathrm{GELU}(x). \tag{2.6}$$

We refer to this approximation as h-GELU. Although h-GELU can be computed with integer arithmetic, we observed that replacing GELU with h-GELU in Transformers results in a significant accuracy drop. This is due to the large gap between h-GELU and GELU as depicted in Table 2.1.[2] Figure 2.2 (left) also shows the noticeable gap between those two functions.

A simple way to address the above problem is to use polynomials to approximate GELU, by solving the following optimization problem:

$$\begin{aligned} &\min_{a,b,c} \frac{1}{2}\left\|\mathrm{GELU}(x) - x \cdot \frac{1}{2}\left[1 + \mathrm{L}(\frac{x}{\sqrt{2}})\right]\right\|_2^2, \\ &\text{s.t.} \quad L(x) = a(x+b)^2 + c, \end{aligned} \tag{2.7}$$

where $\mathrm{L}(x)$ is a second-order polynomial used to approximate the erf function. Directly optimizing Equation 2.7 results in a poor approximation since the definition domain of erf contains the entire real numbers. To address this, we only optimize $L(x)$ in a limited range since erf approaches to 1 (-1) for large values of x. We also take advantage of the fact that

[2] Later in our ablation study, we show this can lead to accuracy degradation of up to 2.2 percentages, as reported in Table 2.4.

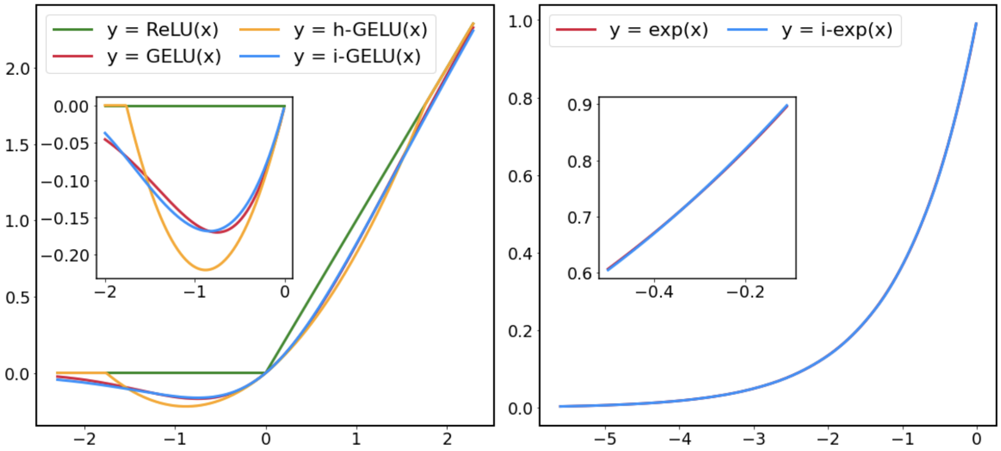


Figure 2.2: (Left) Comparison between RELU, GELU, h-GELU and i-GELU. (Right) Comparison between exponential (exp) and our integer-only exponential (i-exp).

Table 2.1: Comparison of different approximation methods for GELU. The second column (Int-only) indicates whether each approximation method can be computed with integer-only arithmetic. As metrics for approximation error, we report L^2 and L^∞ distance from GELU across the range of [-4, 4].

	Int-only	L^2 dist	L^∞ dist
$x\sigma(1.702x)$	✗	0.012	0.020
h-GELU	✓	0.031	0.068
i-GELU (Ours)	✓	**0.0082**	**0.018**

erf is an odd function (i.e., $\text{erf}(-x) = -\text{erf}(x)$), and thus only consider approximating it in the positive domain. After finding the best interpolating points, i.e., (x_i, f_i) in Equation 2.3, and applying these adjustments we arrive at the following polynomial:

$$\mathrm{L}(x) = \mathrm{sgn}(x)\left[a(\mathrm{clip}(|x|, \max = -b) + b)^2 + 1\right], \tag{2.8}$$

where $a = -0.2888$ and $b = -1.769$, and sgn denotes the sign function.[3] Using this polynomial we arrive at i-GELU, the integer-only approximation for GELU, defined as:

$$\text{i-GELU}(x) := x \cdot \frac{1}{2}\left[1 + \mathrm{L}(\frac{x}{\sqrt{2}})\right]. \tag{2.9}$$

Algorithm 2 summarizes the integer-only computation of GELU using i-GELU. We illustrate the behaviour of i-GELU in Figure 2.2 (left). As one can see, i-GELU closely

[3] Note that $L(x)$ is approximating GELU in the range of $[0, -b]$.

Algorithm 2 Integer-only GELU

```
Input: $q, S$: quantized input and scaling factor
Output: $q_{out}, S_{out}$: quantized output and scaling factor
function I-Erf($q, S$)                                          ▷ $qS = x$
    $a, b, c \leftarrow -0.2888, -1.769, 1$
    $q_{\text{sgn}}, q \leftarrow \text{sgn}(q), \text{clip}(|q|, \text{max} = -b/S)$
    $q_L, S_L \leftarrow$ I-Poly($q, S$) with $a, b, c$              ▷ Equation 2.8
    $q_{out}, S_{out} \leftarrow q_{\text{sgn}} q_L, S_L$
    return $q_{out}, S_{out}$                                   ▷ $q_{out} S_{out} \approx \text{erf}(x)$
end function
function I-Gelu($q, S$)                                         ▷ $qS = x$
    $q_{\text{erf}}, S_{\text{erf}} \leftarrow$ I-Erf($q, S/\sqrt{2}$)
    $q_1 \leftarrow \lfloor 1/S_{\text{erf}} \rfloor$
    $q_{out}, S_{out} \leftarrow q(q_{\text{erf}} + q_1), S S_{\text{erf}}/2$
    return $q_{out}, S_{out}$                                   ▷ $q_{out} S_{out} \approx \text{GELU}(x)$
end function
```

approximates GELU, particularly around the origin. We also report the approximation error of i-GELU along with h-GELU in Table 2.1, where i-GELU has an average error of 8.2×10^{-3} and a maximum error of 1.8×10^{-2}. This is $\sim 3\times$ more accurate than h-GELU whose average and maximum errors are 3.1×10^{-2} and 6.8×10^{-2}, respectively. Also, i-GELU even slightly outperforms the Sigmoid based approximation of Equation 2.5, but without using any floating point arithmetic. Note that computing the Sigmoid requires floating point. Later in the results section, we show that this improved approximation, actually results in better accuracy of i-GELU as compared to h-GELU (see Table 2.4).

Integer-only Softmax

Softmax normalizes an input vector and maps it to a probability distribution:

$$\text{Softmax}(\mathbf{x})_i := \frac{\exp x_i}{\sum_{j=1}^{k} \exp x_j}, \quad \text{where } \mathbf{x} = [x_1, \ldots, x_k]. \tag{2.10}$$

Approximating the Softmax layer with integer arithmetic is quite challenging, as the exponential function used in Softmax is unbounded and changes rapidly. As such, prior Transformer quantization techniques [8, 318] treat this layer using floating point arithmetic. Some prior work have proposed look up tables with interpolation [231], but as before we avoid look up tables and strive for a pure arithmetic based approximation. In addition, although [92] proposes polynomial approximation methods for the exponential function, it uses significantly high-degree polynomials, and is only applicable on a limited finite domain.

Similar to GELU, we cannot use a high-order polynomial, but even using such polynomial is ineffective in approximating the exponential function in Softmax. However, it is possible

Algorithm 3 Integer-only Exponential and Softmax

```
Input: $q, S$: quantized input and scaling factor
Output: $q_{out}, S_{out}$: quantized output and scaling factor
function I-Exp($q, S$)                                   ▷ $qS = x$
    $a, b, c \leftarrow 0.3585, 1.353, 0.344$
    $q_{\ln 2} \leftarrow \lfloor \ln 2 / S \rfloor$
    $z \leftarrow \lfloor -q / q_{\ln 2} \rfloor$
    $q_p \leftarrow q + z q_{\ln 2}$                             ▷ $q_p S = p$
    $q_L, S_L \leftarrow$ I-Poly($q_p, S$) with $a, b, c$      ▷ Equation 2.13
    $q_{out}, S_{out} \leftarrow q_L$>>$z, S_L$
    return $q_{out}, S_{out}$                    ▷ $q_{out} S_{out} \approx \exp(x)$
end function
function I-Softmax($q, S$)                               ▷ $qS = x$
    $\tilde{q} \leftarrow q - \max(q)$
    $q_{\exp}, S_{\exp} \leftarrow$ I-Exp($\tilde{q}, S$)
    $q_{out}, S_{out} \leftarrow q_{\exp} / \mathrm{sum}(q_{\exp}), S_{\exp}$
    return $q_{out}, S_{out}$                ▷ $q_{out} S_{out} \approx \mathrm{Softmax}(x)$
end function
```

to address the problem by limiting the approximation range of Softmax. First, we subtract the maximum value from the input to the exponential for numerical stability:

$$\mathrm{Softmax}(\mathbf{x})_i = \frac{\exp\left(x_i - x_{\max}\right)}{\sum_{j=1}^{k} \exp\left(x_j - x_{\max}\right)}, \tag{2.11}$$

where $x_{\max} = \max_i(x_i)$. Note that now all the inputs to the exponential function, i.e., $\tilde{x}_i = x_i - x_{\max}$, become non-positive. We can decompose any non-positive real number $\tilde{x}$ as $\tilde{x} = (-\ln 2)z + p$, where the quotient z is a non-negative integer and the remainder p is a real number in $(-\ln 2, 0]$. Then, the exponential of $\tilde{x}$ can be written as:

$$\exp(\tilde{x}) = 2^{-z} \exp(p) = \exp(p) \texttt{>>} z, \tag{2.12}$$

where >> is the bit shifting operation. As a result, we only need to approximate the exponential function in the compact interval of $p \in (-\ln 2, 0]$. This is a much smaller range as compared to the domain of all real numbers. Interestingly, a variant of this method was used in the Itanium 2 machine from HP [259, 39], but with a look up table for evaluating $\exp(p)$.

We use a second-order polynomial to approximate the exponential function in this range. To find the coefficients of the polynomial, we minimize the L^2 distance from exponential function in the interval of $(-\ln 2, 0]$. This results in the following approximation:

$$L(p) = 0.3585(p + 1.353)^2 + 0.344 \approx \exp(p). \tag{2.13}$$

Algorithm 4 Integer-only Square Root

```
Input: $n$: input integer
Output: integer square root of $n$, i.e., $\lfloor\sqrt{n}\rfloor$
function I-SQRT($n$)
    if $n = 0$ then return 0
    Intialize $x_0$ to $2^{\lceil Bits(n)/2 \rceil}$ and $i$ to 0
    repeat
        $x_{i+1} \leftarrow \lfloor (x_i + \lfloor n/x_i \rfloor)/2 \rfloor$
        if $x_{i+1} \geq x_i$ then return $x_i$
        else $i \leftarrow i + 1$
end function
```

Substituting the exponential term in Equation 2.12 with this polynomial results in i-exp:

$$\text{i-exp}(\tilde{x}) := L(p) \texttt{>>} z \tag{2.14}$$

where $z = \lfloor -\tilde{x}/\ln 2 \rfloor$ and $p = \tilde{x} + z \ln 2$. This can be calculated with integer arithmetic. Algorithm 3 describes the integer-only computation of the Softmax fucntion using i-exp. Figure 2.2 (right) plots the result of i-exp, which is nearly identical to the exponential function. We find that the largest gap between these two functions is only 1.9×10^{-3}. Considering that 8-bit quantization of a unit interval introduces a quantization error of $1/256 = 3.9 \times 10^{-3}$, our approximation error is relatively negligible and can be subsumed into the quantization error.

2.3 Results

Integer-only LayerNorm

LayerNorm is commonly used in Transformers and involves several non-linear operations, such as division, square, and square root. This operation is used for normalizing the input activation across the channel dimension. The normalization process is described as:

$$\tilde{x} = \frac{x - \mu}{\sigma} \text{ where } \mu = \frac{1}{C}\sum_{i=1}^{C} x_i \text{ and } \sigma = \sqrt{\frac{1}{C}\sum_{i=1}^{C}(x_i - \mu)^2}. \tag{2.15}$$

Here, μ and σ are the mean and standard deviation of the input across the channel dimension. One subtle challenge here is that the input statistics (i.e., μ and σ) change rapidly for NLP tasks, and these values need to be calculated dynamically during runtime. While computing μ is straightforward, evaluating σ requires the square-root function.

The square-root function can be efficiently evaluated with integer-only arithmetic through an iterative algorithm proposed in [34], as described in Algorithm 4. Given any non-negative

Table 2.2: Integer-only quantization result for RoBERTa-Base and RoBERTa-Large on the development set of the GLUE benchmark. Baseline is trained by the authors from the pre-trained models, and I-BERT is quantized and fine-tuned from the baseline. We also report the difference (Diff) between the baseline accuracy and the I-BERT accuracy.

(a) RoBERTa-Base

	Precision	Int-only	MNLI-m	MNLI-mm	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	Avg.
Baseline	FP32	✗	**87.8**	**87.4**	**90.4**	**92.8**	94.6	61.2	**91.1**	90.9	78.0	86.0
I-BERT	INT8	✓	87.5	**87.4**	90.2	**92.8**	**95.2**	**62.5**	90.8	**91.1**	**79.4**	**86.3**
Diff			-0.3	0.0	-0.2	0.0	+0.6	+1.3	-0.3	+0.2	+1.4	+0.3

(b) RoBERTa-Large

	Precision	Int-only	MNLI-m	MNLI-mm	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	Avg.
Baseline	FP32	✗	90.0	89.9	92.8	94.1	96.3	68.0	**92.2**	91.8	86.3	89.0
I-BERT	INT8	✓	**90.4**	**90.3**	**93.0**	**94.5**	**96.4**	**69.0**	**92.2**	**93.0**	**87.0**	**89.5**
Diff			+0.4	+0.4	+0.2	+0.4	+0.1	+1.0	0.0	+1.2	+0.7	+0.5

integer input n, this algorithm iteratively searches for the exact value of $\lfloor \sqrt{n} \rfloor$ based on Newton's Method and only requires integer arithmetic. This algorithm is computationally lightweight, as it converges within at most four iterations for any INT32 inputs and each iteration consists only of one integer division, one integer addition, and one bit-shifting operation. The rest of the the non-linear operations in LayerNorm such as division and square are straightforwardly computed with integer arithmetic.

In this section, we first measure the accuracy of I-BERT using the General Language Understanding Evaluation [269] (GLUE) benchmark (Section 2.3). Then, we discuss the latency speedup of I-BERT using direct hardware deployment and compare it with pure FP32 model (Section 2.3). Finally, we conduct ablation studies to showcase the effectiveness of our integer-only approximation methods (Section 2.3).

Accuracy Evaluation on GLUE

We implement I-BERT on the RoBERTa [171] model using [197]. For the integer-only implementation, we replace all the floating point operations in the original model with the corresponding integer-only operations that were discussed in Section 2.2. In particular, we perform MatMul and Embedding with INT8 precision, and the non-linear operations with INT32 precision, as using INT32 for computing these operations has little overhead. See Section A.3 for implementation details. For each of the GLUE downstream tasks, we train both FP32 baseline and integer-only I-BERT models, and evaluate the accuracy on the development set. See Appendix A.3 and A.3 for training and evaluation details. While we

Table 2.3: Inference latency speedup of INT8 inference with respect to FP32 inference for BERT-Base and BERT-Large. Latency is measured for different sentence lengths and batch sizes.

Sequence Length	128				256				Avg.
Batch Size	1	2	4	8	1	2	4	8	
BERT-Base	2.42	3.36	3.39	3.31	3.11	2.96	2.94	3.15	3.08
BERT-Large	3.20	4.00	3.98	3.81	3.19	3.51	3.37	3.40	3.56

only test RoBERTa-Base/Large, our method is not restricted to RoBERTa. The integer-only approximations can be performed for any NN models including Transformers that uses similar non-linear operations.

The integer-only quantization results for RoBERTa-Base/Large are presented in Table 2.2. As one can see, I-BERT consistently achieves comparable or slightly higher accuracy than baseline. For RoBERTa-Base, I-BERT achieves higher accuracy for all cases (up to 1.4 for RTE), except for MNLI-m, QQP, and STS-B tasks, where we observe a small accuracy degradation up to 0.3. We observe a similar behaviour on the RoBERTa-Large model, where I-BERT matches or outperforms the baseline accuracy for all the downstream tasks. On average, I-BERT outperforms the baseline by 0.3/0.5 for RoBERTa-Base/Large, respectively.

Latency Evaluation

We evaluate the latency speedup of INT8 inference of I-BERT, by direct deployment on a Tesla T4 GPU with Turing Tensor Cores that supports accelerated INT8 execution. Although T4 GPU is not a pure integer-only hardware, we select it as our target device due to its extensive software support [193, 25], and in particular Nvidia's TensorRT library [193]. Furthermore, as we do not exploit any T4-specific exclusive features or requirements, our work can be extensively deployed on other hardware as well. See Section A.3 for the detailed environment setup. For evaluation, we implement two variants of BERT-Base/Large: (1) pure FP32 models using naïve FP32 kernels for non-linear operations; and (2) quantized INT8 models using customized kernels for the non-linear operations. The customized kernels compute GELU, Softmax, and LayerNorm based on the integer-only methods described in Section 2.2. We measure the inference latency for different sequence lengths (128 and 256) and batch sizes (1, 2, 4, and 8).

Table 2.3 shows the inference latency speedup of INT8 models with respect to FP32 models. As one can see, the INT8 inference of I-BERT is on average 3.08× and 3.56× faster than pure FP32 inference for BERT-Base and BERT-Large, respectively, achieving up to 4.00× speedup. The result implies that, when deployed on specialized hardware that supports efficient integer computations, I-BERT can achieve significant speedup as com-

Table 2.4: Accuracy of models that use GELU, h-GELU and i-GELU for GELU computation. Note that the former is full-precision, floating point computation while the latter two are integer-only approximations.

	Int-only	QNLI	SST-2	MRPC	RTE	Avg.
GELU	✗	94.4	96.3	92.6	85.9	92.3
h-GELU	✓	94.3	96.0	92.8	84.8	92.0
i-GELU	✓	**94.5**	**96.4**	**93.0**	**87.0**	**92.7**

pared to FP32 models. Further speedups are possible with NVIDIA's custom Transformer plugins [187] which fuse the multi-head attention and Softmax layers (see Section A.3).

While the greatest value of our work will become evident when our approach enables quantization on lower-end microprocessors without floating-point hardware, this demonstration must wait for improved software support for implementing quantized NN models on those processors. In the meantime, we believe the promise of our approach is illustrated by these latency reductions shown above.

Ablation Studies

Here, we perform an ablation study to show the benefit of i-GELU as compared to other approximation methods for GELU, and in particular h-GELU in Equation 2.6. For comparison, we implement two variants of I-BERT by replacing i-GELU with GELU and h-GELU, respectively. The former is the exact computation of GELU with floating point arithmetic, and the later is another integer-only approximation method for GELU (see Section 2.2). We use RoBERTa-Large model as baseline along with the QNLI, SST-2, MPRC, and RTE tasks. All models are trained and fine-tuned according to the procedure described in Section 2.3, and the final accuracies are reported in Table 2.4.

As one can see, replacing GELU with h-GELU approximation results in accuracy degradation for all downstream tasks except for MRPC. Accuracy drops by 0.5 on average and up to 1.1 for RTE task. Although accuracy slightly improves for MRPC, the amount of increase is smaller than replacing GELU with i-GELU. This empirically demonstrates that h-GELU is not sufficiently tight enough to approximate GELU well. Approximating GELU with i-GELU results in strictly better accuracy for all four downstream tasks than h-GELU. In particular, i-GELU outperforms h-GELU by 0.7 on average, and it achieves comparable or slightly better result to the non-approximated full-precision GELU. i-GELU also performs better than GELU, which is quite interesting, but at this time, we do not have an explanation for this behaviour.

2.4 Related Work

Efficient Neural Network. There are several different approaches to reduce the memory footprint, latency, and power of modern NN architectures. These techniques can be broadly categorized into: (1) pruning [89, 158, 179, 151, 186, 303, 183, 55, 75, 217, 180, 227]; (2) knowledge distillation [100, 184, 209, 221, 228, 253, 122, 258, 264, 255, 276, 298]; (3) efficient neural architecture design [116, 226, 257, 106, 149, 38]; (4) hardware-aware NN co-design [87, 74, 145]; and (5) quantization. Here, we only focus on quantization and briefly discuss the related work.

Quantization. For quantization, the parameters and/or activations are represented with low bit precision [28, 32, 48, 119, 219, 320, 337, 157, 293, 33, 273]. While this line of research mostly focuses on CNN models, there have been recent attempts to introduce quantization techniques into Transformer based models as well. For example, [8] and [318] propose an 8-bit quantization scheme for Transformer based models and compress the model size up to 25% of the original size. Another work [236] applies uniform and mixed-precision to quantize BERT model, where a second-order sensitivity method is used for the mixed-precision setting. [56] quantizes a different subset of weights in each training iteration to make models more robust to quantization. Recently, there have been attempts to quantize BERT with even lower precision. [317] presents a 3/4-bit centroid-based quantization method that does not require fine-tuning. [326, 6] leverage knowledge distillation [100] to ternarize/binarize weights. [123] combines knowledge distillation and learned step size quantization [53] method to achieve up to 2-bit quantization of BERT.

However, to the best of our knowledge, all of the prior quantization work on Transformer based models use *simulated quantization* (aka fake quantization), where all or part of operations are performed with floating point arithmetic. This requires the quantized parameters and/or activations to be dequantized back to FP32 for the floating point operations. For example, [236, 317] perform the entire inference using floating point arithmetic, as schematically shown in Figure 2.1 (left). While [8, 318, 326, 6] attempt to process Embedding and MatMul efficiently with integer arithmetic, they keep the remaining operations (i.e., GELU, Softmax, and LayerNorm) in FP32, as illustrated in Figure 2.1 (middle). However, our method I-BERT uses integer-only quantization for the entire inference process—i.e., without any floating point arithmetic and without any dequantization during the entire inference. This is illustrated in Figure 2.1 (right). This allows more efficient hardware deployment on specialized accelerators or integer-only processors [1] as well as faster and less energy-consuming inference. While we focus on uniform quantization, our method is complementary to other mixed and/or low-precision methods, and can be deployed for those settings as well.

To briefly discuss, there are also several quantization works for computer vision. [119] introduces an integer-only quantization scheme for popular CNN models, by replacing all floating point operations (e.g., convolution, MatMul, and ReLU) with integer operations. Similarly, the recent work of [310] extends this approach to low precision and mixed precision dyadic quantization, which is an extension of integer-only quantization where no integer division is used. However, both of these works are limited to CNN models that only contain linear and piece-wise linear operators, and they cannot be applied to Transformer based models with non-linear operators, e.g., GELU, Softmax, and LayerNorm. Our work aims to address this limitation by extending the integer-only scheme to the Transformer based models without accuracy drop.

2.5 Conclusions

We have proposed I-BERT, a novel integer-only quantization scheme for Transformers, where the entire inference is performed with pure integer arithmetic. Key elements of I-BERT are approximation methods for nonlinear operations such as GELU, Softmax, and LayerNorm, which enable their approximation with integer computation. We empirically evaluated I-BERT on RoBERTa-Base/Large models, where our quantization method improves the average GLUE score by 0.3/0.5 points as comapred to baseline. Furthermore, we directly deployed the quantized models and measured the end-to-end inference latency, showing that I-BERT can achieve up to 4.00× speedup on a Tesla T4 GPU as compared to floating point baseline. As part of future work, one could consider using our approximation to improve the training speed as well. For instance, one could consider replacing GELU with i-GELU during training. Also, further studies are needed to evaluate the performance benefit of i-GELU as compared to GELU.

Chapter 3

Memory Optimization: Dense-and-Sparse Quantization for Large Language Models

3.1 Introduction

Recent advances in Large Language Models (LLMs) trained on massive text corpora, with up to hundreds of billions of parameters, have showcased their remarkable problem-solving capabilities across various domains [12, 216, 229, 50, 101, 30, 244, 325, 260, 262]. However, deploying these models for inference has been a significant challenge due to their demanding resource requirements. For instance, the LLaMA-65B model requires at least 130GB of RAM to deploy in FP16, which exceeds current GPU capacity. Even storing such large-sized models has become costly and complex.

As will be discussed in Section 3.2, the main performance bottleneck in LLM inference for generative tasks is memory bandwidth rather than compute. This means that the speed at which we can load and store parameters becomes the primary latency bottleneck for memory-bound problems, rather than arithmetic computations. However, recent advancements in memory bandwidth technology have been significantly slow, compared to the improvements in computes, leading to the phenomenon known as the Memory Wall [207, 72]. Consequently, researchers have turned their attention to exploring algorithmic methods to overcome this challenge.

One promising approach is quantization, where model parameters are stored at lower precision, instead of the typical 16 or 32-bit precision used for training. For instance, it has been demonstrated that LLM models can be stored in 8-bit precision without performance degradation [311], where 8-bit quantization not only improves the storage requirements by half but also has the potential to improve inference latency and throughput. As a result, there has been significant research interest in quantizing models to even lower precisions. A pioneering approach is GPTQ [61] which uses a training-free quantization technique that

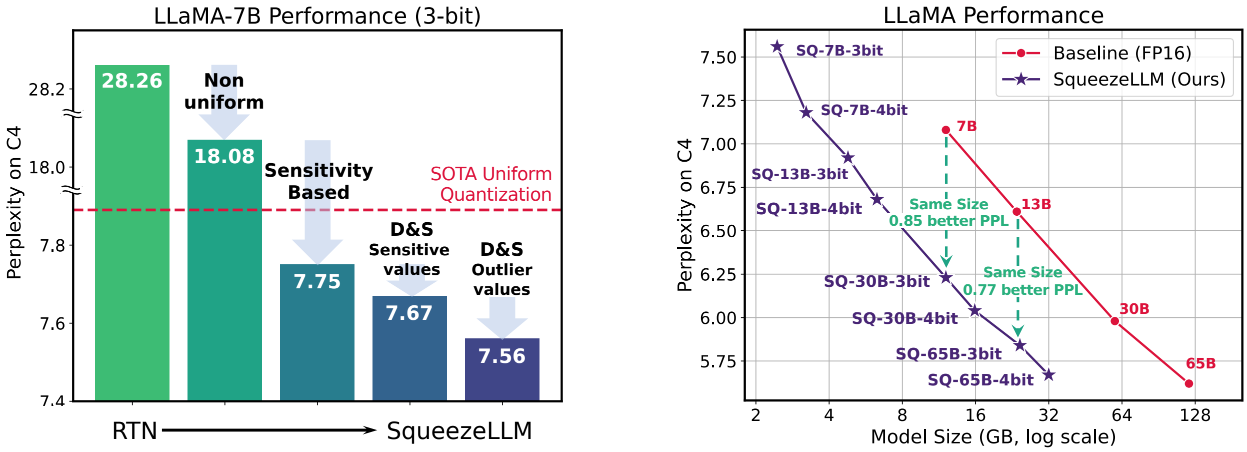


Figure 3.1: (Left) SqueezeLLM incorporates two key approaches: (i) sensitivity-based non-uniform quantization (Section 3.3), where quantization bins are allocated closer to sensitive values, and (ii) the Dense-and-Sparse decomposition (Section 3.3), which retains both sensitive values and outlier values as full-precision sparse format. When applied to LLaMA-7B with 3-bit quantization, our method outperforms the state-of-the-art methods [61, 166] by a large perplexity margin of over 0.3 on the C4 benchmark. (Right) By applying our methods to LLaMA models of varying sizes, we can achieve improved trade-offs between perplexity and model size.

achieves near-lossless 4-bit quantization for LLM models with over tens of billions of parameters. However, achieving high quantization performance remains challenging, particularly with lower bit precision and for relatively smaller models (e.g., $<$ 50B parameters).

Contributions. In this paper, we conduct an extensive study of low-bit precision quantization, and we identify limitations in existing approaches. Based on the insight that *the memory, rather than the compute, is the primary bottleneck* in LLM inference with generative tasks, we introduce SqueezeLLM, a post-training quantization framework with a novel *sensitivity-based non-uniform quantization* and *Dense-and-Sparse decomposition*. These techniques enable lossless compression even at precisions as low as 3 bits with reduced model sizes and faster inference without compromising model performance. Our detailed contributions include:

- **Sensitivity-based Non-Uniform Quantization:** We demonstrate that uniform quantization of prior works is sub-optimal for LLM inference for two reasons. First, the weight distributions in LLMs exhibit clear non-uniform patterns (Figure 3.3). Second, the inference computation in prior works does not fully benefit from uniform quantization as the arithmetic is performed in FP16 precision, not in reduced precision. To address these, we propose a novel sensitivity-based non-uniform quantization method to achieve more optimal LLM quantization, which significantly improves the perplexity of 3-bit LLaMA-7B from 28.26 of uniform quantization to 7.75 on C4 (Section 3.3).

- **Dense-and-Sparse Quantization:** The weights in LLMs contain significant outliers, making low-bit quantization extremely challenging. To address this, we propose a simple solution that decomposes weights into dense and sparse components. The sparse part holds outlier values in full precision using efficient sparse storage methods, and the dense part can have a more compact range to aid quantization. By extracting only 0.45% of the weight values as the sparse component, we further improve the perplexity of LLaMA-7B from 7.75 to 7.58 on C4 (Section 3.3).
- **Evaluation:** We extensively test SqueezeLLM on various models on language modeling tasks using the C4 and WikiText2 datasets as well as on the MMLU [97] and Vicuna benchmarks [27] (Section 3.4). Furthermore, our deployed models on A6000 GPUs also exhibit significant latency gains of up to 2.4× compared to the FP16 baseline, showcasing the effectiveness of our method in terms of both quantization performance and inference efficiency (Section 3.4).

3.2 Memory Wall

Inference behavior broadly falls into two categories: *compute-bound* inference that is limited by computational throughput, and *memory-bound* inference that is bottlenecked by the rate at which data can be fed into the processing cores from memory. *Arithmetic intensity*, the ratio of compute to memory operations, is a typical metric used to assess this behavior. High and low arithmetic intensity indicates a compute-bound and memory-bound problem, respectively. For memory-bound problems, the speedup can be achieved by reducing the memory traffic rather than compute since the compute units in hardware are often under-utilized waiting to receive data from memory.

Generative LLM inference exhibits extremely low arithmetic intensity compared to other workloads[1] [133]. This is because it consists almost entirely of matrix-vector operations, which limits the data reuse as each weight load can only process a single vector for a single token, and cannot be amortized across the multiple vectors for different tokens. This low arithmetic intensity needs to be contrasted with the compute operations on a typical GPU which is orders of magnitude higher than the memory operations.[2] The disparity between compute and memory bandwidth, along with the growing memory requirements of deep learning, has been termed the *Memory Wall* problem [72]. To further illustrate this problem, we used a simple roofline-based performance modeling approach [133] to study LLaMA-7B's runtime on an A5000 GPU with different bit precisions (Figure 3.2). While we assume that all computations are kept at FP16, we see that the latency decreases linearly as we reduce the bit precision, indicating that the main bottleneck is memory, not compute.

[1] To be precise, we limit this discussion to single batch inference where the arithmetic involves matrix-vector operations. For large batch inference, compute can become important.

[2] For instance, A5000 GPU has peak computational throughput of 222 TeraFLOPs per second, which is 290× higher than the peak memory bandwidth of 768 GigaBytes per second.

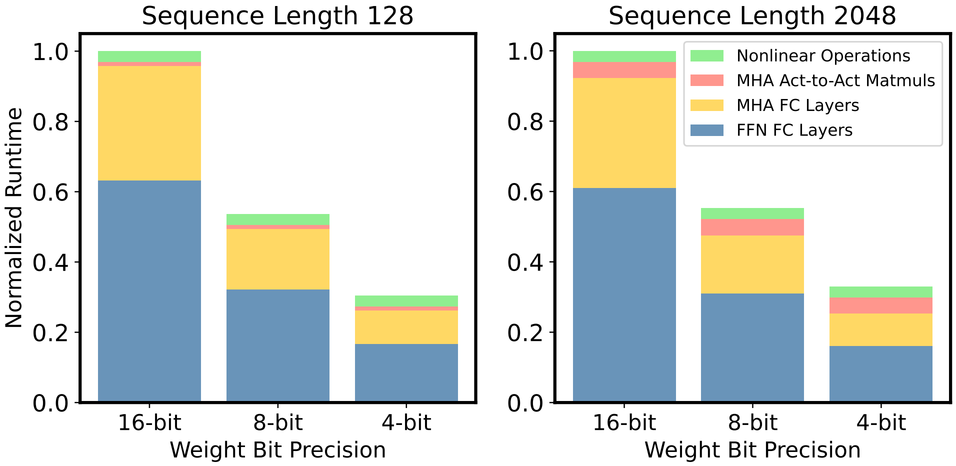


Figure 3.2: Normalized runtime for LLaMA-7B when reducing the bit precision for the weights with sequence lengths of 128 (left) and 2048 (right). Results were obtained using a roofline-based performance model for an A5000 GPU. Reducing only the precision of the weights (and not the activations) is sufficient to obtain significant latency reductions.

In summary, in generative LLM inference, loading weights into memory is the primary bottleneck, while the cost of dequantization and FP16 computation is relatively small. Thus, by quantizing just the weights to lower precision, while leaving the activations in full precision, we can attain significant speedup as well as reduced model size. Given this insight, the appropriate strategy is to *minimize the memory size even if it may add overhead to arithmetic operations.*

3.3 Methodology

Sensitivity-Based Non-uniform Quantization

As in Figure 3.3 (Top), weight distributions in LLMs demonstrate non-uniform patterns. The main task for quantization is to find an optimal way to allocate distinct quantized values (e.g., 8 for 3 bits) in a way that preserves model performance. A widely used approach in LLM quantization works is uniform quantization, where the weight range is evenly divided into bins. This has two main issues. First, uniformly distributing quantized values is sub-optimal as weight distributions are typically non-uniform. Second, while the main advantage of uniform quantization is efficient integer computation, this does not lead to end-to-end latency improvement in memory-bound LLM inference. Therefore, we have chosen non-uniform quantization, which allows for a more flexible allocation of the representative values.

Finding an optimal non-uniform quantization configuration translates into solving a k-means problem. Given a weight distribution, the goal is to determine k centroids that best represent the values (e.g., k=8 for 3-bit). This optimization problem for non-uniform

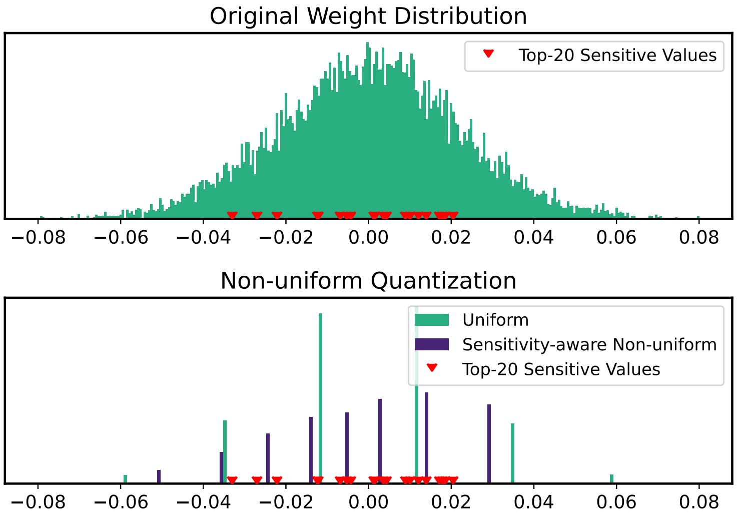


Figure 3.3: (Top) The weight distribution of one output channel in LLaMA-7B. The top-20 sensitive values are marked in red. (Bottom) Weight distributions after 3-bit quantization using uniform and sensitivity-based non-uniform quantization. In the latter case, the quantized values are clustered around the sensitive values.

quantization can be formulated as

$$Q(w)^* = \arg\min_Q \|W - W_Q\|_2^2, \tag{3.1}$$

where W denotes the weights and W_Q is the corresponding quantized weights (i.e., $[Q(w)$ for $w \in W]$), represented by k distinct values $\{q_1, \cdots, q_k\}$. Here, the optimal solution $Q(w)^*$ can be obtained by 1-dimensional k-means clustering, which clusters the parameters into k clusters and assign the centroid of each cluster as q_j's. While this already outperforms uniform quantization, we propose an improved *sensitivity-based* clustering algorithm.

Sensitivity-Based K-means Clustering. The quantization objective is to represent the model weights with low-bit precision with minimal perturbation in the model output [47]. While quantization introduces perturbations in each layer, we need to minimize the overall perturbation with respect to the *final loss*, rather than focusing on individual layers, as it provides a more direct measure of the end-to-end performance degradation after quantization [151]. To achieve this, we need to place the k-means centroids near the values that are more sensitive with respect to the final loss, rather than treating all weight values equally, as in Equation 3.1. To determine more sensitive values, we perform Taylor expansion to analyze how the loss changes in response to perturbations in the weights W:

$$\mathcal{L}(W_Q) \simeq \mathcal{L}(W) - g^\top (W - W_Q) \tag{3.2}$$

$$+ \frac{1}{2}(W - W_Q)^\top H (W - W_Q) \tag{3.3}$$

where g and $H = \mathbb{E}[\frac{\partial^2}{\partial W^2}\mathcal{L}(W)]$ are the gradient and Hessian of the loss at W. Assuming that the model has converged, the gradient g can be approximated as zero which gives us the following formula for computing how much the model gets perturbed after quantization:

$$Q(w)^* = \arg\min_{Q}(W - W_Q)^\top H(W - W_Q). \tag{3.4}$$

In the new optimization target, as compared to Equation 3.1, the perturbation of each weight after quantization, i.e., $W - W_Q$, is weighted by the scaling factor introduced by the second-order derivative, H. This highlights the importance of minimizing perturbations for weights with large Hessian values, as they have a greater impact on the overall perturbation of the final output. In other words, the second-order derivative serves as a measure of importance for each weight value.

Due to the cost of computing the Hessian, we use an approximation to the Hessian based on the Fisher information matrix $\mathcal{F}$, which can be calculated over a sample dataset D as $H \simeq \mathcal{F} = \frac{1}{|D|}\sum_{d\in D} g_d g_d{}^\top$. This only requires computing gradient for a set of samples, which can be calculated efficiently with existing frameworks. To make the optimization objective in Equation 3.4 more feasible, we further approximate the Fisher information matrix as a diagonal matrix by assuming that the cross-weight interactions are negligible. This simplifies our objective target as follows:

$$Q(w)^* \simeq \arg\min_{Q}(W - W_Q)^\top \mathrm{diag}(\mathcal{F})(W - W_Q) \tag{3.5}$$

$$= \arg\min_{Q}\sum_{i=1}^{N} \mathcal{F}_{ii}\big(w_i - Q(w_i)\big)^2. \tag{3.6}$$

An important consequence of Equation 3.5 is the *weighted* k-means clustering setting, where the centroids will be pulled closer to these sensitive weight values. In Figure 3.3, we illustrate the top-20 sensitive values based on the Fisher information of the exemplary weight distribution. At the bottom, the quantized values assigned by uniform quantization (green) are compared to those assigned by the sensitivity-based k-means approach (purple), which achieves a better trade-off by placing centroids near sensitive values, effectively minimizing quantization error. With 3-bit LLaMA-7B, sensitivity-based non-uniform quantization achieves much lower perplexity of 7.75 compared to the 28.26 perplexity of round-to-nearest uniform quantization on C4 (Figure 3.1 and Section 3.4)

Dense-and-Sparse Quantization

Another challenge in low-bit LLM quantization is outlier values [10, 40, 285, 286]. In Figure 3.4, we plot the normalized weight distributions of different layers in LLaMA-7B, which demonstrate that ∼99.9% of the weights are concentrated in a narrow range of ∼10% of the entire distribution. Naively quantizing the weights with a large range will significantly degrade performance, especially at low precisions. However, this also implies opportunity as the range of the weight values can be contracted by a factor of 10 simply by removing a

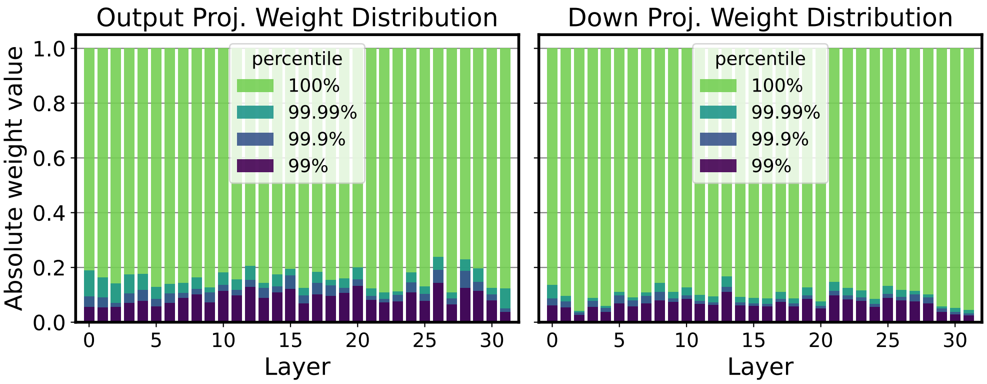


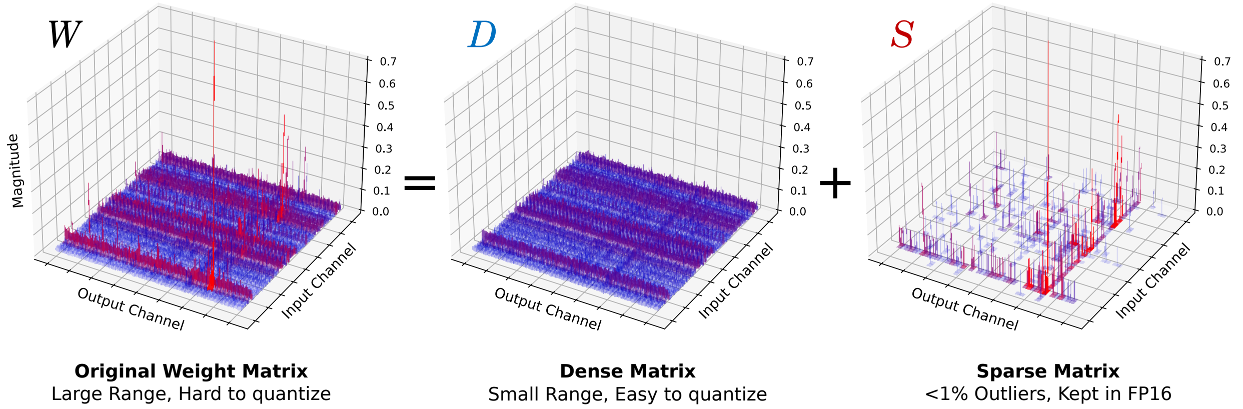
Figure 3.4: The distributions of the (normalized) absolute weight values, for the output layers in MHA and the down layers in FFN across different layers in LLaMA-7B. Note that the distributions exhibit outlier patterns across all layers, with 99% of the values clustered within ∼10% of the entire range.

small number of outlier values (e.g., 0.1%), yielding a significant improvement in quantization resolution. This will then help the sensitivity-based k-means centroids to focus more on the sensitive values rather than a few outliers.

Motivated by this, we introduce a method to filter out outliers from the weight matrix W by performing a simple yet effective decomposition into a sparse matrix (S) containing the outliers and the remaining dense matrix (D) that can be quantized much more effectively thanks to its significantly reduced range of values. That is, $W = D+S$ where $D = W[T_{\min} \leq w \leq T_{\max}]$ and $S = W[w < T_{\min}$ or $w > T_{\max}]$. Here, $T_{\min/\max}$ are thresholds that define outliers based on the percentile of the distribution. This Dense-and-Sparse decomposition process is visually illustrated in Figure 3.5.

Importantly, the overhead of this decomposition is minimal, since the number of outlier values is small (e.g., 0.5% of the entire values). Therefore, the sparse matrix can be stored efficiently using methods like the compressed sparse row (CSR) format. Inference is also straightforward with the decomposition as in $WX = DX + SX$, two kernels for dense and sparse multiplication can be overlapped, and the sparse part (SX) can benefit from sparse kernels (Section 3.3).

Sensitivity-Based Sparse Matrix. In addition to isolating outliers into a sparse matrix, we've also discovered the advantage of precisely representing a small number of highly sensitive weight matrix values. These values can be easily identified based on the Fisher information (Section 3.3). This not only maintains sensitive values with FP16 to avoid their impact on the model output, but also prevents the centroids of Equation 3.5 from skewing towards the sensitive values. We have observed that extracting only 0.05% of these sensitive values across layers substantially enhances quantization performance (Appendix B.2). Altogether, with 3-bit LLaMA-7B, extracting 0.45% of outlier and sensitive values further reduces the perplexity from 7.67 to 7.56 (Figure 3.1 and Section 3.4).



Original Weight Matrix
Large Range, Hard to quantize

Dense Matrix
Small Range, Easy to quantize

Sparse Matrix
<1% Outliers, Kept in FP16

Figure 3.5: The illustration of the Dense-and-Sparse decomposition. The left figure plots the magnitude of a weight matrix (W) in the LLaMA 65B model, which contains a few outliers. These outliers contribute to the large range of values in the original weight matrix which significantly degrades the quantization performance. This matrix, however, can be decomposed into a sparse matrix S (Right) that contains the outliers and the remaining dense matrix D (Middle). The dense matrix D then exhibits a significantly smaller range, making accurate quantization much easier. The sparse matrix S can be kept in full precision with minimal memory and runtime overhead.

Dense-and-Sparse Kernel Implementation

To efficiently process non-uniformly quantized values, we implement 3/4-bit CUDA LUT-based kernels for matrix-vector multiplication between compressed weight matrices and uncompressed activation vectors. These kernels load the compressed weights and dequantize them piece-by-piece to minimize memory bandwidth utilization. The compressed matrices store 3/4-bit indices, which correspond to LUT entries containing FP16 values associated with the bins obtained from non-uniform quantization. After dequantization, all arithmetic is performed in FP16.

To optimize the handling of our Dense-and-Sparse representation, we develop kernels for sparse matrix-vector multiplication that load a matrix in CSR format and a dense activation vector, inspired by [54]. Since the non-zero entry distributions are highly skewed across rows (Appendix B.1), assigning a single thread per row can be inefficient due to uneven workload distribution among threads. Thus, we implement *balanced hybrid kernels* based on [58] by assigning an equal number of nonzeros per thread; this leads to additional synchronization across threads due to rows being processed by multiple threads, but leads to a more balanced work assignment. We set the number of threads such that there were 10 nonzeros per thread. The dense non-uniform kernel and balanced sparse kernels are launched in one call to avoid overhead from summing the outputs from these separate operations.

3.4 Evaluations

Experiment Setup

Models and Datasets. We have conducted comprehensive evaluations of SqueezeLLM using various models on different tasks. First, in the language modeling evaluation, we apply SqueezeLLM to the LLaMA [262], LLaMA2 [261] and OPT [325] models and measure the perplexity of the quantized models on the C4 [216] and WikiText2 [182] datasets with a chunk size of 2048. We also evaluate the domain-specific knowledge and problem-solving ability through the MMLU benchmark [97] using the instruction-tuned Vicuna (v1.1 and v1.3) models. We used the Language Model Evaluation Harness to run zero-shot evaluation across all tasks [64]. Finally, we evaluate the instruction following ability following the methodology presented in [27]. To do so, we generate answers for 80 sample questions and compared them to the answers generated by the FP16 counterpart using the GPT-4 score. To minimize the ordering effect, we provide the answers to GPT-4 in both orders, resulting in a total of 160 queries.

Baseline Methods. We compare SqueezeLLM against PTQ methods for LLMs including RTN as well as state-of-the-art methods including GPTQ [61], AWQ [166] and SpQR [42]. To ensure a fair comparison, we use GPTQ *with* activation ordering throughout all experiments unless specified, which addresses the significant performance drop that would otherwise occur. For AWQ, we use official quantized models or reproduce using their official code if they are not available except for LLaMA 65B with group size 256, which ran into OOM even on A100-80G. Evaluations are then conducted based on our perplexity method. For SpQR, we rely on the paper's reported numbers since their perplexity evaluation methodology is identical to ours. SpQR aims to enhance 3-bit and 4-bit models by introducing grouping, bi-level quantization, and sparsity, making them approximately 4 and 4.6 bits on average for LLaMA. In contrast, SqueezeLLM aims to preserve 3 and 4-bit as closely as possible, minimizing any extra model size overhead. Therefore, we present our best-effort comparison of SpQR and SqueezeLLM by comparing 3-bit SpQR models, which average around 4 bits, and our 4-bit models, both of which possess similar model sizes.

Quantization Details. For SqueezeLLM, we adopt channel-wise quantization where each output channel is assigned a separate lookup table. We use 2 different sparsity levels: 0% (dense-only) and 0.45% (0.05% sensitive values and 0.4% outlier values, as discussed in Section 3.3). For measuring sensitivity, we use 100 random samples from the Vicuna training set for Vicuna models and C4 training set for the others. While grouping can also be incorporated with our method, we found it sub-optimal as compared to extracting sensitive/outlier values with sparsity (Appendix B.2).

Latency Profiling. We measure the latency and peak memory usage for generating 128 and 1024 tokens on an A6000 machine using the Torch CUDA profiler. As an official implementation of GPTQ (in particular, the grouped version) is not available, we implement an optimized kernel for single-batch inference based on the most active open-source codebase [76].

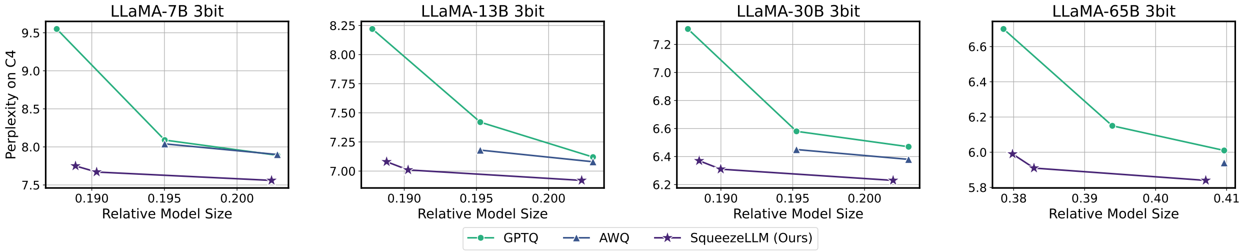


Figure 3.6: Perplexity comparison PTQ methods for 3-bit LLaMA quantization, evaluated on C4. The x-axes are the relative model sizes with respect to the model size in FP16. Different size-perplexity trade-offs are achieved by adjusting the group size for GPTQ and AWQ and the sparsity level for ours. Our quantization method consistently and significantly outperforms GPTQ and AWQ across all model size regimes, with a more pronounced gap in lower-bit and smaller model sizes.

To compare latency with SpQR, we rely on their reported speedup numbers to make our best-effort comparison, as their kernel implementation is not publicly available. Regarding AWQ, we use the GPTQ kernel without activation ordering since they exhibit identical behavior during inference. Although AWQ has released their own kernel implementation, their 3-bit kernels are not publicly available. Furthermore, they have incorporated optimizations that are unrelated to quantization, such as LayerNorm and positional embedding, which are universally applicable to other quantization methods. To ensure a fair comparison with other methods, we refrained from using their released kernels.

Main Results

Table 3.1 shows quantization results for LLaMA along with the baseline methods. The models are grouped based on their size to better compare size-perplexity trade-offs. See Figure 3.6 for a visual illustration. Below we use LLaMA-7B as the main example to discuss the impact of dense-only and Dense-and-Sparse quantization, and we then discuss how these trends extend to larger models. We provide the full evaluation result on all LLaMA models in Table B.13.

Dense-only Quantization. In Table 3.1 (Top), we compare dense-only SqueezeLLM with 0% sparsity level and GPTQ without grouping. With 4-bit quantization, our method exhibits minimal degradation compared to the FP16 baseline, with only ∼0.1 perplexity degradation on C4 and WikiText2, while reducing the model size by 3.95×. Moreover, when compared to non-grouped GPTQ our method shows significant perplexity improvement of up to 0.22.

The performance gap between the two methods becomes more pronounced with 3-bit quantization. SqueezeLLM outperforms GPTQ by a substantial margin of 1.80/1.22 points on C4/WikiText2 with a 5.29× compression rate. This is only 0.67/0.55 points off from the FP16 baseline. This demonstrates the effectiveness of the sensitivity-based non-uniform method for ultra-low-bit quantization.

Table 3.1: Perplexity comparison of LLaMA models quantized into 3 and 4 bits using different methods including RTN, GPTQ, AWQ and SpQR on C4 and WikiText-2. We compare the performance of different methodologies by grouping them based on their model sizes. In the first group, we compare dense-only SqueezeLLM with non-grouped GPTQ. In the second group, we compare SqueezeLLM with a sparsity level of 0.45% to GPTQ and AWQ with a group size of 128. For comparison, we add speedup and peak memory usage numbers, which we provide more details in Table 3.3. Further results for LLaMA-30/65B and other models including LLaMA-2 7/13/70B are provided in Appendix B.6. † Since SpQR does not release their kernel implementation, we conduct our best-effort comparison using their reported speedup numbers. See Section 3.4 for details. ‡ GPTQ with activation ordering incurs a significant latency penalty as elements in the same channel are associated with different scaling factors, resulting in distributed memory accesses (Section 3.4). GPTQ *without* activation ordering alleviates the latency issue at the cost of a substantial perplexity degradation.

LLaMA-7B	**3-bit**					**4-bit**				
Method	**Avg. Bits** (comp. rate)	**PPL** C4	Wiki	**Speed up**	**Mem** (GB)	**Avg. Bits** (comp. rate)	**PPL** C4	Wiki	**Speed up**	**Mem** (GB)
Baseline	16	7.08	5.68	1×	12.7	16	7.08	5.68	1×	12.7
RTN	3 (5.33)	28.26	25.61	2.3×	2.9	4 (4.00)	7.73	6.29	2.0×	3.7
GPTQ	3 (5.33)	9.55	7.55	2.3×	2.9	4 (4.00)	7.43	5.94	2.0×	3.7
SpQR	-	-	-	-	-	3.94 (4.06)	7.28	5.87	1.2×†	N/A
SqueezeLLM	3.02 (5.29)	**7.75**	**6.32**	2.1×	2.9	4.05 (3.95)	**7.21**	**5.79**	1.8×	3.8
GPTQ (g128, no reorder)‡	3.24 (4.93)	10.09	8.85	2.0×	3.0	4.24 (3.77)	7.80	6.07	1.6×	3.8
GPTQ (g128)‡	3.24 (4.93)	7.89	6.27	0.2×	3.0	4.24 (3.77)	7.21	5.78	0.4×	3.8
AWQ (g128)	3.24 (4.93)	7.90	6.44	2.0×	3.0	4.24 (3.77)	7.22	5.82	1.6×	3.8
SqueezeLLM(0.45%)	3.24 (4.93)	**7.56**	**6.13**	1.9×	3.1	4.27 (3.75)	**7.18**	**5.77**	1.7×	4.0

LLaMA-13B	**3-bit**					**4-bit**				
Method	**Avg. Bits** (comp. rate)	**PPL** C4	Wiki	**Speed up**	**Mem** (GB)	**Avg. Bits** (comp. rate)	**PPL** C4	Wiki	**Speed up**	**Mem** (GB)
Baseline	16	6.61	5.09	1×	24.6	16	6.61	5.09	1×	24.6
RTN	3 (5.33)	13.24	11.78	2.7×	5.3	4 (4.00)	6.99	5.53	2.3×	6.8
GPTQ	3 (5.33)	8.22	6.22	2.7×	5.3	4 (4.00)	6.84	5.29	2.3×	6.8
SpQR	-	-	-	-	-	3.96 (4.04)	6.72	5.22	1.2×†	N/A
SqueezeLLM	3.02 (5.30)	**7.08**	**5.60**	2.4×	5.4	4.04 (3.96)	**6.71**	**5.18**	2.0×	6.9
GPTQ (g128, no reorder)‡	3.25 (4.92)	7.16	5.53	2.2×	5.7	4.25 (3.77)	6.71	5.18	1.9×	7.2
GPTQ (g128)‡	3.25 (4.92)	7.12	5.47	0.2×	5.6	4.25 (3.77)	6.70	5.17	0.4×	7.0
AWQ (g128)	3.25 (4.92)	7.08	5.52	2.2×	5.7	4.25 (3.77)	6.70	5.21	1.9×	7.2
SqueezeLLM (0.45%)	3.24 (4.94)	**6.92**	**5.45**	2.2×	5.8	4.26 (3.76)	**6.68**	**5.17**	1.9×	7.3

Dense-and-Sparse Quantization. By leveraging the Dense-and-Sparse quantization, we achieve a further reduction in the perplexity gap from the FP16 baseline, as shown in Ta-

Table 3.2: Comparison of PTQ methods on zero-shot MMLU accuracy applied to Vicuna v1.1 and v1.3. We add peak memory usage in GB for comparison. Additional results on 5-shot MMLU evaluation can be found in Appendix B.6.

Method	**Avg.**	**7B (v1.1)**		**13B (v1.1)**		**7B (v1.3)**		**13B (v1.3)**		**33B (v1.3)**	
	bit	Acc	Mem	Acc	Mem	Acc	Mem	Acc	Mem	Acc	Mem
Baseline	16	39.1%	12.7	41.2%	24.6	40.2%	12.7	43.3%	24.6	49.5%	OOM
AWQ (g128)	4.25	38.0%	3.8	40.4%	7.2	**39.6%**	3.8	42.2%	7.2	49.5%	17.2
SqueezeLLM	4.05	38.8%	3.8	39.2%	6.9	39.3%	3.8	**44.1%**	6.9	48.0%	17.5
SqueezeLLM (0.45%)	4.26	**39.4%**	4.0	**41.0%**	7.3	39.5%	4.0	43.8%	7.3	**49.9%**	18.7
AWQ (g128)	3.25	36.5%	3.0	37.6%	5.7	37.4%	3.0	40.7%	5.7	46.4%	13.2
SqueezeLLM	3.02	36.0%	2.9	37.2%	5.4	35.1%	2.9	40.5%	5.4	46.2%	12.5
SqueezeLLM (0.45%)	3.24	**37.7%**	3.1	**39.4%**	5.8	**37.6%**	3.1	**40.8%**	5.8	**47.7%**	14.7

ble 3.1. This improvement is particularly significant with 3-bit quantization, where extracting just 0.45% of the values yields around 0.2 perplexity improvement. This enables nearly lossless compression with less than 0.1/0.5 perplexity deviation from the FP16 baseline for 4/3-bit, respectively.

Both GPTQ and AWQ use a grouping strategy to enhance performance with a slight overhead in model size. However, we demonstrate that SqueezeLLM with a sparsity level of 0.45% consistently outperforms both GPTQ/AWQ with a group size of 128 in all scenarios with comparable model sizes. This is more pronounced for 3-bit quantization, where SqueezeLLM with a 0.45% sparsity level outperforms both GPTQ/AWQ with a group size of 128 by up ∼0.3 perplexity.

Results on Larger Models. In Table 3.1 (13B) and Table B.13 (30/65B), we observe that the trend in 7B extends to larger models, where SqueezeLLM consistently outperforms other PTQ methods across all models and bit widths. Such a trend is also illustrated in Figure 3.6 for 3-bit quantization where even *dense-only* SqueezeLLM achieves comparable perplexity to *grouped* GPTQ/AWQ. With sparsity, we can further improve perplexity, reducing the gap from the FP16 baseline to less than 0.1/0.4 perplexity points for 4/3-bit quantization. Notably, with 3-bit quantization, our approach achieves up to a 2.1× reduction in perplexity gap from the FP16 baseline compared to existing methods. Further ablation studies on our design choices are provided in Appendix B.2, and additional results on the LLaMA2 and OPT models can be found in Appendix B.6.

Quantization of Instruction Following Models

Instruction tuning has emerged as a method for improving the model's ability to respond to user commands. We explore the quantization of instruction-following models to demonstrate the benefits of SqueezeLLM in terms of accuracy preservation by applying it to the Vicuna models, and evaluating the performance with the following approaches.

MMLU Evaluation. We first evaluate the baseline and quantized models on the MMLU benchmark where the weighted accuracy in the zero-shot setting is provided in Table 3.2 for

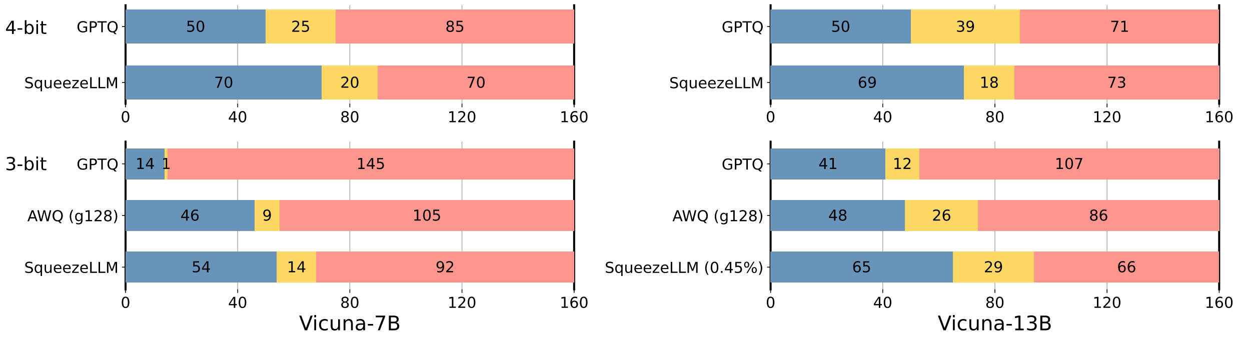


Figure 3.7: Comparison of PTQ methods applied to Vicuna v1.1. Blue / yellow / red represent the number of times that the quantized model won / tied / lost against the baseline FP16 model. This evaluation was performed using the methodology from Vicuna.

Vicuna models. As we can see, 3-bit SqueezeLLM achieves higher accuracy for all models compared to AWQ and also preserves the FP16 baseline accuracy with 4-bit quantization. 5-shot results are provided in Appendix B.6.

Instruction-Following Ability. Another approach for evaluating instruction-following ability is to ask GPT-4 to rank the generated responses as presented in [27]. As shown in Figure 3.7, SqueezeLLM without sparsity achieves near-perfect performance (i.e., 50/50 split) with 4-bit quantization for both Vicuna-7B and 13B, outperforming GPTQ with the same model size. In the case of 3-bit quantization, SqueezeLLM outperforms both GPTQ and AWQ with comparable model sizes. In the case of the Vicuna-13B model, achieving a near-perfect 50/50 split for 3-bit quantization.

Hardware Deployment and Profiling

We show the latency and peak GPU memory usage of SqueezeLLM in Table 3.3 on an A6000 GPU for different configurations when generating 128 tokens. We observe that the LUT-based non-uniform approach in SqueezeLLM (3rd row) shows up to 2.4× speedup compared to the FP16 baseline, and exhibits comparable latency and peak memory usage to the uniform quantization of non-grouped GPTQ (2nd row). This indicates that the overhead associated with LUT-based dequantization is small, especially considering the significant perplexity gains it enables.

Additionally, when incorporating sparsity, we still observed latency gains relative to the FP16 baseline. As shown in Table 3.3, keeping 0.45% of parameters in FP16 (4th row) only adds around 10% latency overhead relative to the dense-only implementation, while still resulting in up to 2.2× speed up compared to the FP16 baseline. In contrast, when accounting for permutation, the GPTQ runtime is degraded heavily (5th row). This latency penalty is due to permutation, which means that elements in the same channel need to be scaled using different scaling factors (which are accessed using group indices); it is challenging for these distributed memory accesses to be performed efficiently, as GPUs rely heavily on

Table 3.3: Latency (s) and peak memory usage (GB) of 3-bit LLaMA when generating 128 tokens on an A6000 GPU. The table compares the FP16 baseline, non-grouped and grouped GPTQ with activation ordering, and SqueezeLLM with different sparsity levels. For comparison, we include bitwidth and perplexity (PPL) on the C4 benchmark. See Table B.11 for additional results on generating 1024 tokens, and see Table B.12 for additional benchmarking results on an A100 GPU.

Method	**Avg.**	**7B**			**13B**			**30B**			**65B**		
	bit	PPL	Lat	Mem	PPL	Lat	Mem	PPL	Lat	Mem	PPL	Lat	Mem
Baseline	16	7.08	3.2	12.7	6.61	5.6	24.6	5.98	OOM	OOM	5.62	OOM	OOM
GPTQ	3	9.55	1.4	2.9	8.22	2.1	5.3	7.31	4.0	12.3	6.70	6.7	24.0
SqueezeLLM	3.02	7.75	1.5	2.9	7.08	2.4	5.4	6.37	4.0	12.5	5.99	7.6	24.5
GPTQ (g128)	3.25	7.89	13.7	3.0	7.12	24.2	5.6	6.47	61.9	12.9	6.01	117.8	25.1
SqueezeLLM (0.45%)	3.24	7.56	1.7	3.1	6.92	2.5	5.8	6.23	4.4	14.7	5.84	8.8	28.0

coalesced memory accesses in order to optimally use memory bandwidth. This shows how our Dense-and-Sparse quantization methodology allows for both higher accuracy as well as better performance relative to GPTQ. Additional evaluation results on generating 1024 tokens are provided in Table B.11, where we observe a similar trend.

3.5 Related Work

Transformer Quantization. Quantization methods can be broadly categorized based on whether retraining is required or not [70]. Quantization-Aware Training (QAT) requires retraining the model to adapt its weights to help recover accuracy after quantization [318, 236, 134, 328, 326, 6], whereas Post-Training Quantization (PTQ) does not involve retraining [332, 14, 241, 194, 160]. While QAT often results in better accuracy, it is often infeasible for LLMs due to the expensive retraining cost and/or lack of access to the training data and infrastructure. As such, most works on LLM quantization have focused on PTQ [311, 40, 61, 296, 316, 166]. Our work also focuses on the PTQ approach.

Quantization methods can be also classified as uniform or non-uniform [70]. Uniform quantization [61, 166, 42, 318, 236, 134, 113, 170], which uniformly divides weight ranges into bins, has gained popularity since it allows faster computation by using quantized precision arithmetic. However, recent hardware trends indicate that faster computation does not necessarily translate to improved end-to-end latency or throughput [72], particularly in memory-bound tasks like generative LLM inference (Section 3.2). Furthermore, uniform quantization can be sub-optimal when the weight distribution is non-uniform, as in LLMs (Figure 3.3).

Hence, we focus on non-uniform quantization, which non-uniformly allocates quantization bins without constraints for a more accurate representation of weights and smaller quantization errors. While it does not support integer arithmetic for computational acceleration, this drawback is not significant for memory-bound problems, as in our focus, where the primary bottleneck lies in memory bandwidth rather than computation. Among non-uniform quantization methods [120, 31], the most similar work to ours is GOBO [317], which introduces a k-means clustering-based look-up table approach. Our work introduces two novel methods as compared to GOBO: (i) sensitivity-based methods; and (ii) Dense-and-Sparse quantization methodologies, which yield substantial improvements within the k-means-based non-uniform quantization framework.

LLM Quantization. With the increasing popularity of LLMs, *weight-only quantization* has surfaced as a promising approach to reduce memory consumption and enhance inference efficiency. GPTQ [61] has been a pioneering work, and AWQ [166] and SpQR [42] have also suggested the weight-only quantization schemes concurrent to our work. Our work, however, is different in two key aspects. First, our work employs non-uniform quantization, as opposed to uniform quantization of the aforementioned works. In particular, our sensitivity-based non-uniform quantization not only better represents non-uniform distributions of weights, but it also strategically reduces the impact on more sensitive values, thereby enabling more aggressive quantization without performance degradation. Second, while previous works quantize weights in a way that layer-wise output activations remain unaffected, our approach targets preserving the model's final output. This strategy of minimizing the final loss, as shown in Appendix B.2, leads to better quantization performance since it is a direct measure of the end-to-end performance degradation after quantization.

Non-uniform Quantization. For low-bit LLM quantization, [41] has recently introduced the NF datatype, highlighting the importance of non-uniform quantization. However, our approach differs by offering a more dynamic non-uniform representation that accounts for both weight distributions and sensitivity of values, as opposed to the static, hard-coded NF datatype that assumes the normal distribution of the weights. While previous studies [88, 301] have used k-means clustering in quantization, our work pioneers its application in LLM quantization. Furthermore, we introduce the novel sensitivity-based weighted k-means clustering strategy, enabling lossless sub-4-bit quantization by significantly reducing performance degradation, in contrast to the sensitivity-agnostic counterpart (Figure 3.1).

Outlier-Aware Quantization. Among the various challenges in low-bit Transformer quantization, one key issue is the presence of outliers [140], which can unnecessarily increase the quantization range. To address this issue, outlier-aware quantization methods have been investigated [10, 40, 285, 286, 296]. Notably, [40] keeps outlier activations in floating-point, while [285] transfers outlier factors to later layers without affecting functionality. These focus on activations, which is not a concern in our work where all activations are in floating-point. Our Dense-and-Sparse quantization instead tackles *weight* outliers for low-bit LLM quantization.

Concurrently to our work, SpQR [42] also explores outlier extraction in the context of weight quantization. While SpQR has shown a promising result on outlier extraction, SqueezeLLM, leveraging sensitivity-based non-uniform quantization, achieves precise quantization with significantly lower (e.g., 0.05%) or even zero sparsity levels. This is critical for both reducing model size and improving inference speed, as higher sparsity often degrades latency. Furthermore, SqueezeLLM uses outlier extraction as a direct solution to prevent outliers from negatively impacting quantization performance, bypassing the need for using the grouping strategy as an indirect solution. This contrasts with SpQR, which relies on fine-grained grouping that leads to increased model size and a more complex quantization process such as the bi-level quantization scheme.

Dense-and-Sparse Decomposition. Matrix decomposition into dense and sparse components has been explored in attention map decomposition [19, 37], leveraging the fact that attention patterns often present low-rank characteristics with a few outliers. To the best of our knowledge, however, our research is the first to apply the dense-and-sparse decomposition strategy to weight matrices to improve quantization performance. Additionally, we uniquely incorporate both outlier and sensitive values within the sparse matrix, which yields considerable improvement in post-quantization performance.

3.6 Conclusion

We have presented SqueezeLLM which attempts to address the Memory Wall problem associated with generative LLM inference that is memory-bound. SqueezeLLM incorporates two novel ideas that allow ultra-low precision quantization of LLMs with negligible degradation in generation performance: the sensitivity-based non-uniform quantization method; and the Dense-and-Sparse decomposition that resolves the outlier issue. We have evaluated SqueezeLLM on a wide range of models and datasets that assess language modeling, problem-solving, and instruction-following capabilities of quantized models, where we have demonstrated that our quantization method can consistently outperform the previous state-of-the-art methodologies.

Chapter 4

Efficient Inference Method: Speculative Decoding with Big Little Decoder

4.1 Introduction

In recent years, the Transformer [266] has become the *de-facto* model architecture for a wide range of Natural Language Processing tasks. The potential of the Transformer architecture has been further enhanced by the emergence of Large Language Models (LLMs) with up to hundreds of billions of parameters trained on massive text corpora [12, 216, 229, 50, 101, 29, 244, 325, 260]. Despite their performance, efficiently running these models for inference is a challenge due to their large model size and runtime complexity. This limits their use in many applications that require real-time responses.

These computational inefficiencies are particularly pronounced in *autoregressive* generative tasks such as machine translation [16, 9], summarization [99], and language modeling [182]. For these tasks, models need to run iteratively to generate tokens sequentially, as each token is dependent on the previously generated tokens. This requires the models to load weight matrices, as well as the cached keys and values of previously generated tokens [210], for each token generation, thus preventing parallelization of the loaded values across multiple tokens. This makes autoregressive text generation memory bandwidth constrained during inference [124]. As a consequence, autoregressive generative tasks suffer from low hardware utilization as well as high inference latency [133]. In contrast, non-autoregressive tasks, such as text classification [269], can process the entire input sequence with a single weight load, which is then shared across all input tokens in parallel. Given the increasing popularity of text generation tasks, in light of advancements in LLMs, it is critical to improve the inference latency and runtime efficiency of autoregressive decoding processes despite the potential sacrifice in generation quality.

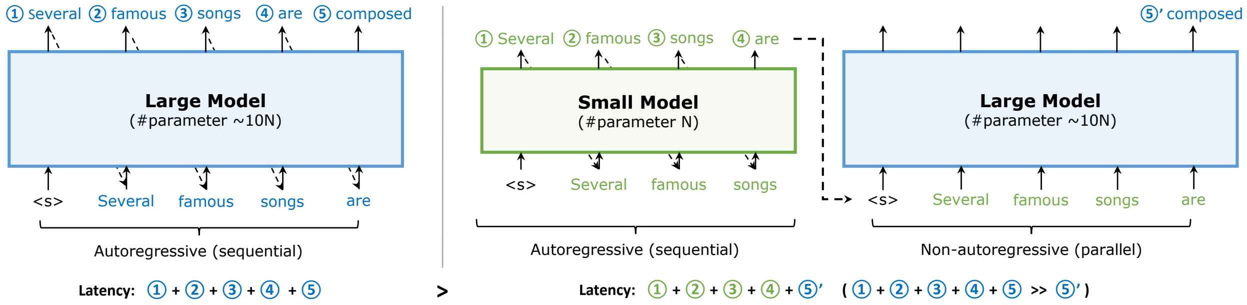

① several ② famous ③ songs ④ are ⑤ composed
Large Model
(#parameter ~10N)
<s> Several famous songs are
Autoregressive (sequential)
Latency: ① + ② + ③ + ④ + ⑤
>
① Several ② famous ③ songs ④ are
Small Model
(#parameter N)
<s> Several famous songs
Autoregressive (sequential)
⑤' composed
Large Model
(#parameter ~10N)
<s> Several famous songs are
Non-autoregressive (parallel)
Latency: ① + ② + ③ + ④ + ⑤' (① + ② + ③ + ④ + ⑤ >> ⑤')

Figure 4.1: Illustration of (Left) the normal autoregressive decoding procedure of a large model and (Right) BiLD that consists of a small model and a large model. In BiLD, the small model generates tokens autoregressively (i.e., sequentially) until it hands over control to the large model. The large model then takes as input the tokens generated by the small model in parallel, allowing for non-autoregressive (i.e., parallel) execution to generate the next token. This improves end-to-end latency by allowing for more efficient utilization of underlying hardware.

To overcome this, non-autoregressive decoding [81, 278, 163, 254, 283, 233, 152, 69, 84] has been explored to maximize token-level parallelization and reduce the inference latency of generative tasks by generating multiple tokens simultaneously. This approach can be more computationally efficient than the regular autoregressive process. However, non-autoregressive decoding suffers from text generation quality issues due to its assumption of conditional independence between output tokens [127]. In order to achieve comparable performance to that of autoregressive processes, it generally requires complex and often task-dependent training strategies, supplementary hinting information that guides the decoding process [278, 163, 254, 283, 233], and knowledge distillation [335].

In this paper, we introduce a novel framework named Big Little Decoder (BiLD) that can be applied to various text generation scenarios to reduce inference latency *without* additional training iterations or modifications to the existing training pipeline or model architecture. As illustrated in Figure 4.1 (Right), the BiLD framework consists of two decoder models, a large model and small model, that work collaboratively to generate text sequences. In particular, only the small model is executed autoregressively to generate the majority of the text, taking advantage of its small runtime overhead. The large model only engages occasionally to refine the small model's inaccurate predictions, thus allowing for efficient non-autoregressive execution. This *autoregressive small, non-autoregressive large* scheme results in a substantial improvement of up to ~2× in end-to-end inference latency, compared to regular autoregressive execution, while maintaining similar or better generation quality. The effectiveness of our framework is also supported by our observation that the predictions made by small and large models only slightly disagree, and thus the small model can match the performance of the large model with a minimal refinement of its own predictions (Figure 4.2, Section 4.2).

In summary, our main contributions are as follows:

- We introduce BiLD, a general framework that allows faster inference of various text generation applications. Our framework is designed to coordinate a large model and a small model such that the large model is only executed infrequently and efficiently in a non-autoregressive manner to refine the small model's inaccurate predictions.
- We propose two policies for BiLD: the fallback policy that allows the small model to hand control over to the large model if it is not confident enough (Section 4.2), and the rollback policy that allows the large model to review and correct the small model's inaccurate predictions (Section 4.2).
- We introduce a simple yet effective technique for aligning the predictions of the small model with those of the large model. By incorporating this *prediction alignment* technique into the BiLD framework, we can further enhance its performance with minimal additional effort (Section 4.2).
- We apply BiLD for 4 different text generation scenarios including IWSLT 2017 De-En [16] and WMT 2014 De-En [9] for machine translation, XSUM [188] and CNN/ DailyMail [99] for summarization. Compared to the full autoregressive execution, BiLD achieved a speedup of up to 1.85× without generation quality degradation and 2.12× allowing ∼1 point degradation on an NVIDIA T4 GPU (Section 4.3).

4.2 Methodology

Motivating Examples

Although large models tend to produce higher-quality text, they also result in longer end-to-end latencies, which can be further exacerbated by the regressive process of predicting one token at a time. However, in many text generation scenarios we demonstrate that a model that is an order of magnitude smaller than a larger model can achieve comparable generation quality to the larger model, provided that a few erroneous predictions are corrected. This implies that only a small fraction of the small model's predictions deviate from those of the larger model. To validate this claim, we evaluate two different generative scenarios, machine translation with mT5 [302] on WMT 2014 De-En [9] and summarization with T5 [216] on CNN/DailyMail [99] by running the large model along the small model for every decoding iteration. See Section 4.3 for more details on these models. Then, we measure the likelihood of the large model predicting the same token that the small model generates. If the likelihood is below a certain threshold, we assume that the small model's prediction is not accurate enough, and we replace it with the large model's prediction. By controlling the threshold, we can adjust the proportion of the large model's engagement.

Figure 4.2 plots the text generation quality on the validation dataset of each benchmark for different proportions of the large model's engagement. The results exhibit a clear trend across the tasks where the small models with ∼10× smaller sizes can retain the large model's generation quality only if approximately 20% of their inaccurate predictions were substituted

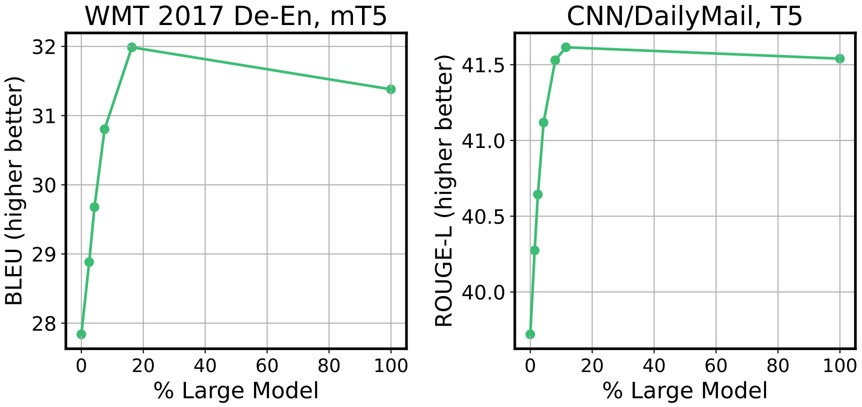


Figure 4.2: Quality of text generation for different proportions of the large model's engagement on the small model's prediction, evaluated on the validation datasets of (Left) WMT 2014 De-En translation [9]; and (Right) CNN/DailyMail summarization [99]. We see that the small models can achieve a comparable or better generation quality to the large models if ∼20% of their incorrect predictions were substituted.

by the large model. While this experiment assumes an ideal case where the predictions of the large model are available as ground truth in every iteration, it nonetheless demonstrates the feasibility of achieving the text generation quality of the large model while maintaining the low inference latency of the small model.

Problem Formulation

At nth decoding iteration, the small model and the large model each take as input a partially generated output text $y_{1:n-1} = (y_1, \cdots, y_{n-1})$, and then generate a probability distribution over entire vocabulary $p_S(y|y_{1:n-1})$ and $p_L(y|y_{1:n-1})$, respectively. Then, the next token $y_{n,S}$ and $y_{n,L}$ are sampled from the probability distributions,

$$y_{n,S} \sim p_S(y|y_{1:n-1}) \quad \text{and} \quad y_{n,L} \sim p_L(y|y_{1:n-1}). \tag{4.1}$$

Depending on whether to use the small model or the large model for the nth decoding step, the nth token y_n can be either $y_{n,S}$ or $y_{n,L}$. When deciding which model to use, it is not feasible to run the large model along with the small model for every decoding step to verify the predictions of the small model, as in the experiments in Section 4.2. Thus, it is necessary to hand over the control to the large model only when the small model is likely to make an inaccurate prediction based on a policy $\pi(y_{1:n-1})$ that returns a boolean value $\{0, 1\}$ indicating whether to use the large model:

$$y_n = \begin{cases} y_{n,S} & \text{if } \pi(y_{1:n-1}) = 0 \\ y_{n,L} & \text{if } \pi(y_{1:n-1}) = 1. \end{cases} \tag{4.2}$$

The objective, therefore, is to design a lightweight policy π that leads to high text generation quality with the minimum end-to-end latency by invoking the large model only when necessary. In order to illustrate the mechanism by which latency is reduced, consider a simple case where the small model has generated tokens y_1 through y_n autoregressively. If the large model takes over the control and predicts the next token, y_{n+1}, it can now take multiple input tokens (y_1 through y_n) in parallel, thus allowing for *non-autoregressive* inference. It is worth noting that this non-autoregressive approach would require the same amount of FLOPs as a regressive approach that predicts y_1 through y_{n+1} sequentially; however, it is much faster on hardware due to its token-level parallelism and increased arithmetic intensity [289]. In other words, processing multiple tokens in a single memory operation is more efficient than individually processing tokens with separate memory operations, as memory accesses can be more costly than arithmetic operations in decoding tasks [133]. If the latency saving from running the large model non-autoregressively outweights the additional cost of running the small model, there is a net latency reduction. Therefore, the aim of this approach is *not* to reduce the number of FLOPs, but rather to improve the hardware utilization and arithmetic intensity of the decoding process. More detailed analysis on this can be found in Appendix C.5. Figure 4.1 provides a high-level overview of how the small and the large models in BiLD coordinates for text generation.

We now focus on constructing an appropriate policy π for our framework. Here, we introduce two simple policies, the fallback and rollback policies, which (despite their simplicity) result in high performance with significant latency reduction. We discuss the details in the following subsections.

Fallback Policy: Small Model Knows When to Stop Predictions

The first principle of the policy is that the small model should be able to determine when to hand over control to the large model. Whenever the small model lacks confidence in its prediction, it is better to allow the large model to take over. Confidence (or uncertainty, in reverse) quantification has been an active research area [62, 128], and any lightweight confidence metric can serve as a potential candidate. Here, we find it sufficient with a simple policy based on the maximum prediction probability, i.e., $\max_y p_S(y|y_{1:n-1})$, similar to the observations made in [94]. If the maximum prediction probability is lower than a certain threshold α_{FB}, then the small model's prediction is regarded to be not confident enough, and we *fallback* to the large model to generate the next token. Note that this does not entail a runtime overhead. Figure 4.3 (Top) illustrates the fallback policy.

Fallback Policy: If $\max_y p_S(y|y_{1:n-1}) < \alpha_{FB}$, then fallback to the large model and set $y_n = y_{n,L}$.

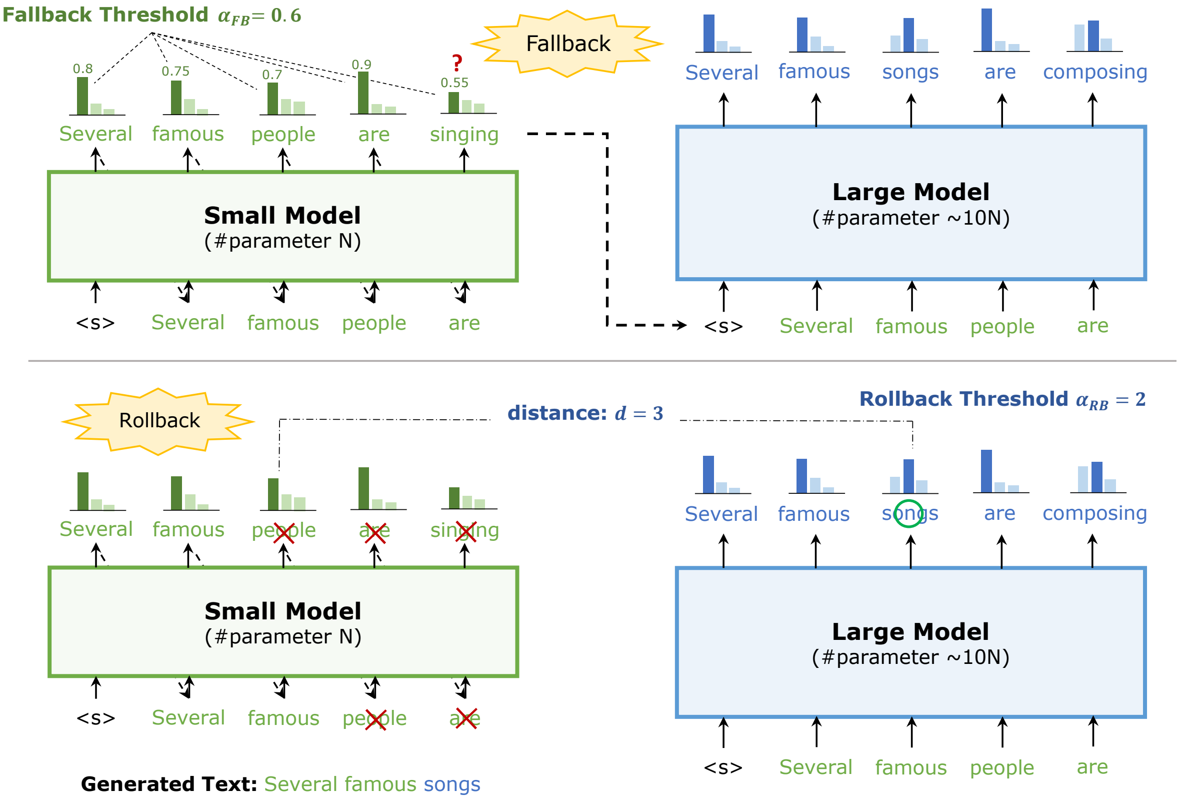


Figure 4.3: (Top) The fallback policy. When the small model generates tokens autoregressively, if the prediction probability of a specific token is below the predefined fallback threshold value α_{FB}, the prediction is deemed to be not confident enough, and control is then shifted to the larger model to produce the corresponding token. (Bottom) The rollback policy. If the large model takes over the control, it produces its own predictions for all previous tokens, as well as the current token. If the prediction probability from the large model for a previously generated token deviates from the small model's prediction by a distance metric d exceeding the predetermined rollback threshold α_{RB}, the small model's prediction is regarded as incorrect. In such a case, we roll back all predictions made by the small model that follow the corresponding token.

Rollback Policy: Large Model Knows When to Revert Predictions

While the fallback policy allows the large model to take over when the small model is not confident enough, it is still possible that the small model is over-confident in its incorrect predictions [83]. Moreover, a single incorrect prediction at an early decoding iteration can lead to a catastrophic effect [232], as it will affect all subsequent token predictions. To avoid such cases, it is desirable to have the large model review the small model's predictions and ensure the validity of each prediction. In our framework, this comes without any extra cost.

When the large model is provided with the tokens generated by the small model for its non-autoregressive prediction of the next token, it also produces its own predictions for all the previous decoding steps. That said, given the partially generated text $y_{1:n}$, it generates $p_L(y|y_{1:m})$ for all previous and current decoding steps $m = 1, \cdots, n$, which can be used to validate the small model's previous predictions.

Therefore, for some distance metric $d(\cdot, \cdot)$ that compares two probability distributions, we find the smallest decoding step m such that

$$d(p_S(y|y_{1:m}), p_L(y|y_{1:m})) > \alpha_{RB} \tag{4.3}$$

for a predetermined threshold α_{RB}. If such m exists, we regard the small model's previous prediction y_m to be inaccurate, and we *rollback* all predictions that follow, i.e., y_m through y_n, since they are all dependent on the wrong prediction. We then replace y_m with $y_{m,L}$ of the large model. We will discuss in Section 4.3 that the cross-entropy loss between the small model's hard label and the large model's soft label (which measures the likelihood of obtaining the small model's prediction from the large model's output) is a good choice for the metric d. Rollback may incur additional latency due to the need for duplicated computation for the reverted tokens. However, we demonstrate in Section 4.3 the net advantage of rollback as the improved text generation quality outweighs the additional latency. See Figure 4.3 (Bottom) for a detailed illustration of the rollback policy.

Rollback Policy: If there exists a minimum $m \in [1, n-1]$ such that $d(p_S(y|y_{1:m}), p_L(y|y_{1:m})) > \alpha_{RB}$, then rollback the predictions $(y_m, \cdots, y_n)$ and set $y_m = y_{m,L}$.

Big Little Decoder

Taken together, the Big Little Decoder (BiLD) framework consists of one small model, one large model, and a policy that determines which model to use for each decoding iteration. The policy comprises two components: the fallback policy to fall back to the large model when the small model's prediction is not confident enough; and a rollback policy to roll back the small model's predictions if they deviate from the predictions of the large model. Algorithm 5 provides a summary of the end-to-end algorithm.

Model Prediction Alignment

BiLD is a general framework that imposes no restriction on the selection of small and large models as long as they use the same vocabulary. Therefore, as will be demonstrated in Section 4.3, two independently trained models can compose BiLD to achieve a significant latency improvement. Nevertheless, when two models are trained separately, they may generate sequences with similar or identical semantic meanings but using different vocabularies. For instance, one model may produce the phrase "writing is hard" while the other may generate "writing is difficult". Because the BiLD policy relies on the degree of agreement

Algorithm 5 Big Little Decoder

```
$y \leftarrow [\text{bos}]$
while $y[-1] \neq \text{eos}$
    $p_S \leftarrow \text{SmallModel}(y)$
    if $\max(p_S[-1]) > \alpha_{FB}$
        # Use the small model's predicton
        $y \leftarrow y + [\text{sample}(p_S[-1])]$
    else
        # Fallback to the large model
        $p_L \leftarrow \text{LargeModel}(y)$
        $m \leftarrow$ min. index such that $d(p_L[m], p_S[m]) > \alpha_{RB}$
        if $m$ exists
            # Rollback: use the large model's prediction
            $y \leftarrow y[:m] + [\text{sample}(p_L[m])]$
        else
            # Don't rollback: use the large model's prediction
            $y \leftarrow y + [\text{sample}(p_L[-1])]$
return $y$
```

between the large and small models, such a vocabulary-level discrepancy can result in unnecessary disagreements that roll back the small model's prediction without any improvement in generation quality.

In order to address this issue and further optimize the BiLD performance, we present a simple approach called *model prediction alignment* that aids in aligning the predictions produced by the small and large models. To achieve this, we leverage a calibration dataset $\mathcal{X}_{\text{cal}} = \{x^{(i)}\}$ that well represents the input sentence distribution. We then generate the corresponding output sequence for each input sequence using the large model, resulting in $\mathcal{Y}_{\text{cal}} = \{y^{(i)}\}$ where $y^{(i)} = \arg\max p_L(y|x^{(i)})$. Subsequently, we fine-tune the small model using the calibration examples $(x_{\text{cal}}, y_{\text{cal}}) \in (\mathcal{X}_{\text{cal}}, \mathcal{Y}_{\text{cal}})$.

The underlying rationale of this approach is to increase the likelihood of the small model generating sequences that would have been generated by the large model. This can minimize the distance between the small model and the large model's predictions per each token, i.e., $d(p_S(y|x, y_{1:m}), p_L(y|x, y_{1:m}))$, throughout the decoding process, thereby avoiding unnecessary rollbacks. Despite its simplicity, our experiments in Section 4.3 demonstrate that this approach can be incorporated into the BiLD framework with minimal effort to significantly enhance the performance. We further emphasize that this method does not introduce any additional complexity or hyperparameters to the normal training pipeline. This is comparable to knowledge distillation [100], an alternative method for aligning model predictions, which requires modifications to the training pipeline, access to internal hidden states such as logits, and additional hyperparameter tuning.

4.3 Evaluations

Experiment Setup

Models and Datasets. To access the generalizability and validity of BiLD in various text generation settings, we have selected IWSLT 2017 De-En [16] and WMT 2014 De-En [9] for machine translation benchmarks and XSUM [188] and CNN/DailyMail [99] for summarization benchmarks. We used mT5-large and small [302] for machine translation and T5-large and small [216] for summarization as our target models, where the size of the models differ by approximately a factor of 20. Our framework is built on top of PyTorch [204] and the HuggingFace Transformers library [290] along with their pre-trained checkpoints.

Training. We fine-tune the pre-trained models on the target benchmarks for 500k steps to obtain the *baseline* small and large models. To train the *aligned* small models via the prediction alignment method (Section 4.2), we generate output sequences from the input sequences of the training datasets using the fully trained large models to create calibration datasets. We then fine-tune the pre-trained small models on the calibration dataset using the same training recipes and the number of steps as the baseline small models. More training details can be found in Appendix C.1. Throughout the paper, we refer to BiLD with the baseline and aligned small models as *unaligned* and *aligned* BiLD, respectively.

Inference. All inference evaluations are conducted on a single NVIDIA T4 GPU of a GCP n1-standard-4 instance, using a batch size 1, which is a common use case for online serving [232]. For the distance metric d in Equation 4.3 for the rollback policy, we use the cross-entropy loss between the small model's hard label and the large model's soft label. For BiLD inference, we sweep over different fallback and rollback thresholds to explore different trade-offs between generation quality and latency. More evaluation details can be found in Appendix C.1.

Main Results

The main results are illustrated in Figure 4.4, which shows the trade-off between text generation quality and average end-to-end latency per example, normalized by the vanilla inference latency of the pure large baseline models. The trade-offs are obtained by controlling the fallback and rollback thresholds. Table 4.1 summarizes the results, with the second and third rows corresponding to unaligned BiLD. When coupled with the normally fine-tuned baseline small models, BiLD achieves an average speedup of 1.50× across all benchmarks, with up to 1.71× speedup on CNN/DailyMail without any degradation in text generation quality (2nd row). By allowing $\sim$1 point degradation, BiLD achieves an average speedup of 1.70×, with up to 2.05× speedup (3rd row). Note that unaligned BiLD is a *pure* plug-and-play solution that does not require additional training effort or cost beyond preparing small and large models independently.

In addition, Figure 4.4 shows the efficacy of the prediction alignment method, leading to a consistent improvement of aligned BiLD over unaligned BiLD. As summarized in the

Table 4.1: The summary of Figure 4.4 which compares the generation quality and latency speedup of BiLD against vanilla inference with large baseline models. The first row reports the vanilla inference, and the second and third rows report unaligned BiLD. The fourth and fifth rows report aligned BiLD. In both cases of unaligned and aligned BiLD, we report the speedup with minimal BLEU/ROUGE-L score degradation (second and fourth rows), and within ∼1 point degradation (third and fifth rows).

Task (Model)	Machine Translation (mT5)				Summarization (T5)			
Dataset	IWSLT		WMT		XSUM		CNN/DailyMail	
	BLEU	Speedup	BLEU	Speedup	ROUGE-L	Speedup	ROUGE-L	Speedup
Vanilla Inference	40.32	-	31.38	-	35.08	-	41.54	-
BiLD (Unaligned)	40.33	1.43×	31.28	1.34×	35.12	1.48×	41.44	1.71×
	39.44	1.58×	30.47	1.43×	34.02	1.72×	40.57	2.05×
BiLD (Aligned)	40.24	1.62×	31.26	1.47×	35.05	1.50×	41.52	1.85×
	39.13	1.78×	30.33	1.70×	33.95	1.80×	40.96	2.12×

forth and fifth rows of Table 4.1, aligned BiLD that incorporates the aligned small models yields an average speedup of 1.61×, with up to 1.85× speedup (4th row). Within ∼1 point degradation, it achieves an average speedup of 1.85×, with up to 2.12× speedup (5th row). The results also demonstrate that both unaligned and aligned BiLD outperform the baseline BLEU/ROUGE-L scores in the high-latency regime, which can be attributed to the ensembling effect of using two different models, as also studied in prior work [181]. In Appendix C.5, we provide examples of text sequences generated by BiLD, which demonstrate that the large model's engagement in BiLD decoding not only improves the prediction accuracy but also prevents incorrect predictions from impacting the future ones.

We have additionally conducted a performance comparison of our method with the speculative sampling method proposed in [20] on the IWSLT 2017 De-En and XSUM benchmarks. We implement and evaluate it in the same environment as our main BiLD experiments using the same baseline large and small models. We apply a fixed window size of [3, 10]. On the IWSLT benchmark, speculative sampling achieves a BLEU score of 39.93 with a 1.28× speedup, while BiLD (unaligned) achieves a 0.61 higher BLEU score with similar speedup, or a 0.21× more latency gain with a similar BLEU score. On the XSUM benchmark, speculative sampling achieves a ROUGE-L score of 35.00 with a 1.25× speedup. In contrast, BiLD achieves up to a 0.30 ROUGE-L score gain with a faster latency, or up to 0.22× more latency gain with a better ROUGE-L score. We provide more detailed comparisons in Appendix C.3.

Ablation Studies

We have further conducted two ablation studies to validate the individual components of BiLD by (1) removing the rollback policy, and (2) removing the fallback policy. When removing the rollback policy, we use the same fallback thresholds as the main experiments

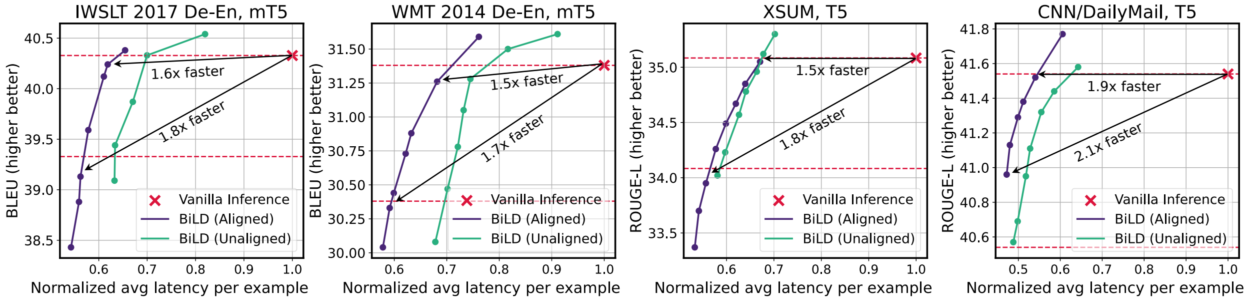


Figure 4.4: Generation quality and average end-to-end latency of processing a single example on 4 different benchmarks. We report BLEU for machine translation and ROUGE-L for summarization as performance metrics. The green and blue lines are unaligned and aligned BiLD, respectively. The X marks are the vanilla inference with the baseline large models. For comparison, two horizontal lines are plotted to indicate the BLEU/ROUGE-L score of the vanilla inference and 1 point degradation from it. The latency on the x-axis is normalized by the baseline latency.

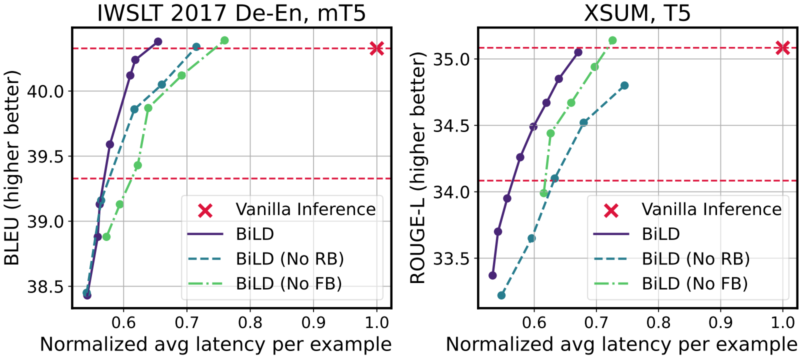


Figure 4.5: Ablation study results for BiLD on (Left) IWSLT 2017 De-En translation and (Right) XSUM summarization tasks without the rollback or fallback policy. Aligned small models were used in all cases. The result demonstrates that BiLD experiences significant performance degradation without either policy in both tasks. The horizontal lines indicate the vanilla inference score and 1 point degradation from it.

to control the generation quality and latency trade-off. When removing the fallback policy, we use the same rollback thresholds as the main experiments. Additionally, we apply fallback after a fixed number of small model executions (swept over [3, 10]), similar to [20].

Figure 4.5 illustrates the results of these ablation studies on IWSLT 2017 De-En for machine translation and XSUM for summarization with aligned BiLD. The results show that the rollback policy consistently produces better generation quality across all latency regimes, particularly in the high-BLEU/ROUGE regime where the large model's engagement via rollback is crucial in correcting small model's wrong predictions. This demonstrates that,

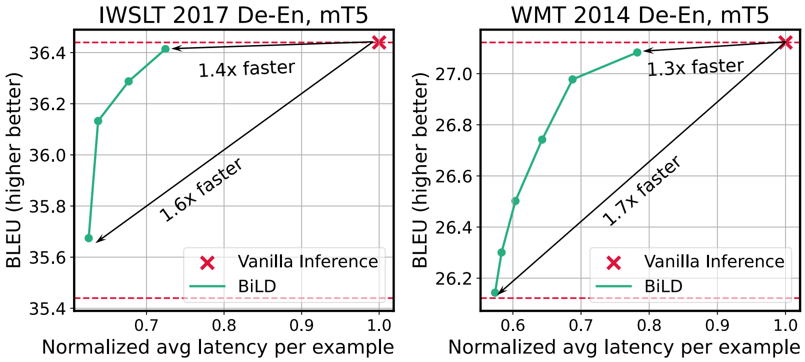


Figure 4.6: Application of the BiLD framework to the early exit problem using the mT5-small model as the large model and its first layer as the small model, evaluated on (Left) the IWSLT 2017 De-En and (Right) WMT 2014 De-En benchmarks. The × marks indicate the latency and BLEU score of the mT5-small models. The horizontal lines indicate the vanilla inference score and 1 point degradation from it.

despite the additional latency overhead from the duplicated computation of reverted tokens, the improvement in text generation quality outweighs this cost. Similarly, removing the fallback policy and periodically handing over control to the large model after a fixed number of token generations leads to significant performance degradation. Taken together, these results highlight that both policies are critical components of BiLD.

Early Exiting Strategy in the BiLD Framework

So far, we have demonstrated how BiLD can be used as a general framework for accelerating the text generation process by incorporating a small model and a large model. However, having two separate models is not a limitation as they can be combined into a single model by using a subset of a larger model, such as a few of its early layers, as a smaller model. This approach resembles the early exit strategy, which is a popular method for accelerating the decoding process [232]. This section demonstrates how the early exiting strategy can be reframed within the BiLD framework.

To demonstrate the applicability of using the early exiting strategy within the BiLD framework, we use mT5-small model as the large model and the first (out of 8) layer as the small model, and evaluate it on two machine translation benchmarks: IWSLT 2017 De-En and WMT 2014 De-En. To ensure consistency between the prediction made after the first layer and the one made after the last layer, we train the model with the average loss of these two layers, similar to [52, 232]. The prediction head is shared for these two layers. More training and evaluation details can be found in Appendix C.2.

Figure 4.6 illustrates the results, where for each benchmark, BiLD achieves up to 1.60× and 1.74× speedup within less than one point BLEU score drop, respectively. This demonstrates the extensibility of the BiLD framework to early exit problems. In Appendix C.2, we further provide a detailed comparison of our results with CALM [232], another framework that incorporates early exiting for fast Transformer decoding. Compared to CALM, BiLD offers two advantages that contribute to better generation quality: (1) in BiLD, even if an early exited prediction (i.e., prediction made by the smaller model) is incorrect, it can be corrected and replaced using the rollback policy; (2) the key and value caches for skipped layers are filled with actual values instead of being computed from the exiting layer's hidden states, leading to reduced error propagation and improved decoding stability. As a result, when tested on IWSLT 2017 De-En and WMT 2014 De-En using mT5-small, BiLD achieves a BLEU score improvement of up to ∼2 points over CALM in both datasets (Figure C.1).

4.4 Related Work

Efficient Transformer Decoding Inference

A variety of approaches have been proposed to increase the speed and reduce the overall inference costs of Transformers. Well-known approaches include efficient architecture design [115, 138, 149, 255, 275, 295], quantization [134, 236, 317, 318, 311, 294, 40], pruning [63, 227, 144, 183, 268, 55, 146], and neural architecture search [21, 245, 246, 271, 299, 312]. While these methods are generally suitable for Transformer-based tasks, some of the works have been focused on efficient decoding mechanisms to reduce the cost of autoregressive tasks.

One popular line of research that shares similarity to our work is *non-autoregressive decoding*. Non-autoregressive decoding, also known as parallel decoding, was first introduced in [81] as a method to reduce inference latency by producing multiple output tokens in parallel, thus avoiding sequential text generation. Subsequent work has further improved the performance of non-autoregressive models by incorporating auxiliary or hinting information [278, 163, 254, 283, 233] to ensure more accurate parallel decoding, or by allowing multiple additional iterations to refine any inaccurate predictions [152, 69, 84]. Such a multi-iteration decoding scheme has also been proposed in [287, 80, 250], which generates texts with fewer steps than the autoregressive scheme by inserting or deleting multiple tokens per iteration. However, these works require complex and often task-dependent training strategies and/or auxiliary information to achieve comparable performance to that of autoregressive models. In contrast, our methodology aims for a plug-and-play solution that does not require any complex training pipeline.

Our work is also related to the approaches that reduce the decoding cost by making decoders shallow. [127] demonstrates that increasing the depth of encoders and decreasing the depth of decoders can reduce decoding latency while still preserving performance. CALM [232] recently introduces *early exiting*, which dynamically adjusts the depth of the decoder for each token generation by terminating the inference at a middle layer, rather than executing until the end layer. While our method shares the same goal of accelerating decoding, we take a different approach by improving decoding parallelism rather than by skipping unnecessary computation. In addition, our framework offers several advantages over

CALM: (1) our method is a fully black box approach that does not involve any modifications to model structures, while CALM requires modifications such as state propagation for the skipped layers; (2) our approach does not require changes to the training pipeline, whereas CALM requires averaged loss across all layers to ensure layer consistency; (3) our approach can be also applied without any training which is critical in various LLM use cases where training is either infeasible or prohibitively expensive. In Section 4.3, we also show that the early exiting strategy can be implemented in our framework to yield significantly better generation quality, further demonstrating the generalizability of our method to a wider range of problems.

Use of Multiple Models

Coordinating the use of multiple models has also been explored in knowledge distillation and ensemble learning. *Knowledge distillation* is a widely adopted methodology for enhancing the performance of smaller models by training them to replicate the behavior of larger, more complex models [100]. When applied to the Transformer architecture, this approach involves distilling the final logits [228, 258] and/or hidden states of a larger model, such as the attention map [255, 122, 276]. In contrast to knowledge distillation, which leverages the knowledge of a large model solely during the training time to improve the training of a smaller model, our method is a run-time solution applied during the decoding process. Therefore, our approach can be more adaptive to run-time behaviors and does not add complexity to training.

Ensemble learning is another approach for coordinating multiple models, wherein multiple models are trained independently and their predictions are combined to improve overall performance. Ensemble learning has been found to yield promising results for Transformer inference [199, 114, 300, 181, 102], particularly when the models aggregated are pre-trained on different datasets and use different techniques. However, ensemble learning generally requires running multiple models and combining their predictions at run-time, which can be computationally expensive and not optimized for latency. Our research aims to optimize both model performance and run-time latency.

Concurrently and independently of our work, [156, 20] also propose an interesting algorithm to accelerate generative inference using a more powerful model to score and speculatively sample predictions from a less powerful model. While [156, 20] offer unbiased estimators that match the stronger model's probability distributions, our extensive empirical evaluation shows that our approach can deliver superior latency-performance trade-offs, due to its non-random rollback (i.e., rejection) policy as well as the dynamic fallback window size. See Section 4.3 and Appendix C.3 for an in-depth comparison.

4.5 Conclusion

In this work, we have introduced Big Little Decoder (BiLD), a framework that reduces end-to-end inference latency for a wide variety of text generation tasks without the need for training or modifying the existing models. In essence, our framework couples a large and small decoder model together to generate text more efficiently. In particular, we start inference with a small model which runs autoregressively for the majority of the time to generate text with a low inference cost, while the large model is executed non-autoregressively to refine the small model's inaccurate predictions. BiLD incorporates two policies, the fallback policy, which hands control to the large model when the small model is uncertain, and the rollback policy, which allows the large model to revert the small model's inaccurate predictions. Our framework is evaluated across various text generation scenarios, including machine translation, summarization, and language modeling. Running on an NVIDIA Titan Xp GPU, with no performance drop BiLD achieved an average speedup of 1.52×, with improvements of up to 2.18× on some tasks. Furthermore, when a 1 point degradation in performance was allowed, BiLD achieved an average speedup of 1.76× with speedups of up to 2.38× on some tasks.

Chapter 5

Efficient Model Architecture: Efficient Transformer for Automatic Speech Recognition

5.1 Introduction

The increasing success of end-to-end neural network models has been a huge driving force for the drastic advancements in various automatic speech recognition (ASR) tasks. While both convolutional neural networks (CNN) [329, 159, 141, 178, 90] and Transformers [322, 167, 321, 165, 125] have drawn attention as popular backbone architectures for ASR models, each of them has several limitations. Generally, CNN models lack the ability to capture global contexts and Transformers involve prohibitive computing and memory overhead. To overcome these shortcomings, Conformer [82] has recently proposed a novel convolution-augmented Transformer architecture. Due to its ability to synchronously capture global and local features from audio signals, Conformer has become the *de facto* model not only for ASR tasks, but also for various end-to-end speech processing tasks [85]. Furthermore, it has also achieved state-of-the-art performance in combination with recent developments in self-supervised learning methodologies as well [330, 189]. While the Conformer architecture was introduced as an autoregressive RNN-Transducer (RNN-T) [77] model in its original setting, it has been adopted with less critique to non-autoregressive schemes such as Connectionist Temporal Classification (CTC) [78] as well [191].

Despite being a key architecture in speech processing tasks, the Conformer architecture has some limitations that can be improved upon. First, Conformer still suffers from the quadratic complexity of the attention mechanism limiting its efficiency on long sequence lengths. This problem is further highlighted by the long sequence lengths of typical audio inputs as also pointed out in [238]. Furthermore, the Conformer architecture is relatively more complicated than Transformer architectures used in other domains such as in natural language processing [43, 266, 213] or computer vision [49, 57, 263]. For instance, the

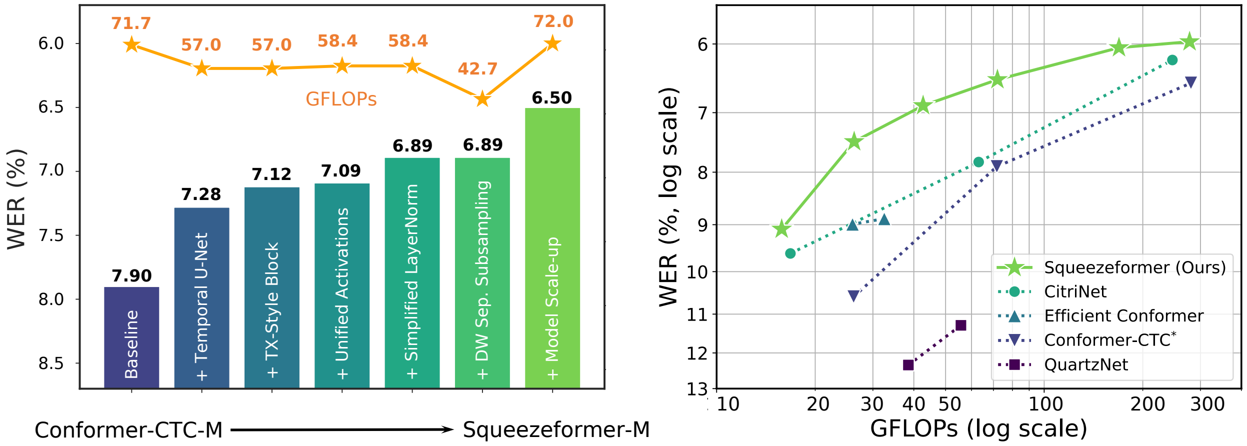


Figure 5.1: (Left) We perform a series of systematic studies on macro and micro architecture to redesign the Conformer architecture towards our Squeezeformer architecture. The bars and the line indicate the WER on LibriSpeech test-other dataset and the FLOPs, respectively. For each design modification, we strictly improve WER until our final Squeezeformer model outperforms Conformer by 1.40% WER improvement with the same number of FLOPs. See Table 5.1 for the details. (Right) LibriSpeech test-other WER vs. FLOPs for Squeezeformer and other state-of-the-art ASR models. Conformer-CTC* is our own reproduction to the best performance as possible and the others are the reported numbers in their papers [141, 178, 13]. Our architecture scales well to smaller and larger models to constantly outperform other models by a large margin throughout the entire FLOPs range. See Table 5.3 for the details. For both plots, the lower the WER, the better; however, we plotted in reverse for better visualization.

Conformer architecture incorporates multiple different normalization schemes and activation functions, the Macaron structure [174], as well as back-to-back multi-head attention (MHA) and convolution modules. This level of complexity makes it difficult to efficiently deploy the model on dedicated hardware platforms for inference [134, 192, 314]. More importantly, this raises the question of whether such design choices are necessary and optimal for achieving good performance in ASR tasks.

In this paper, we perform a careful and systematic analysis of each of the design choices with the goal of achieving lower word-error-rate (WER) for a given computational budget. We developed a much simpler and more efficient hybrid attention-convolution architecture in both its macro and micro-design that consistently outperforms the state-of-the-art ASR models. In particular, we make the following contributions in our proposed Squeezeformer model:

- We find a high temporal redundancy in the learned feature representations of neighboring speech frames especially deeper in the network, which results in unnecessary computational overhead. To address this, we incorporate the temporal U-Net structure in which a downsampling layer halves the sampling rate at the middle of the network, and a light upsampling layer recovers the temporal resolution at the end for training stability (Section 5.2).

- We redesign the hybrid attention-convolution architecture based on our observation that the back-to-back MHA and convolution modules with the Macaron structure are suboptimal. In particular, we propose a simpler block structure similar to the standard Transformer block [266, 43], where the MHA and convolution modules are each directly followed by a single feed forward module (Section 5.2).
- We finely examine the micro-architecture of the network and found several modifications that simplify the model overall and greatly improve the accuracy and efficiency. This includes (i) activation unification that replaces GLU activations with Swish (Section 5.2), (ii) Layer Normalization simplification by replacing redundant pre-Layer Normalization layers with a scaled post-Layer Normalization which incorporates a learnable scaling for the residual path that can be merged with other layers to be zero-cost during inference (Section 5.2), and (iii) incorporation of a depthwise separable convolution for the first sub-sampling layer that results in a significant floating point operations (FLOPs) reduction (Section 5.2).
- We show that the Squeezeformer architecture scales well with both smaller and larger models and consistently outperforms other state-of-the-art ASR models when trained under the same settings (Table 5.3, Section 5.3). Furthermore, we justify the final model architecture of Squeezeformer with a reverse ablation study for the design choices (Table 5.4, Section 5.3).

5.2 Architecture Design

The Conformer architecture has been widely adopted by the speech community and is used as a backbone for different speech tasks. At a macro-level, Conformer incorporates the Macaron structure [174] comprised of four modules per block, as shown in Figure 5.2 (Left). These blocks are stacked multiple times to construct the Conformer architecture. In this work, we carefully reexamine the design choices in Conformer, starting first with its macro-architecture, and then its micro-architecture design. We choose Conformer-CTC-M as the baseline model for the case study, and we compare word-error-rate (WER) on LibriSpeech test-other as a performance metric for each architecture. Furthermore, we measure FLOPs on a 30s audio input as a proxy for model efficiency. While we acknowledge that FLOPs may not always be a linear indicator of hardware and runtime efficiency, we choose FLOPs as it is hardware agnostic and is statically computable. However, we do measure the final end-to-end throughput of our changes, ensuring up to 1.34× consistent improvement in runtime for different versions of Squeezeformer (Table 5.3).

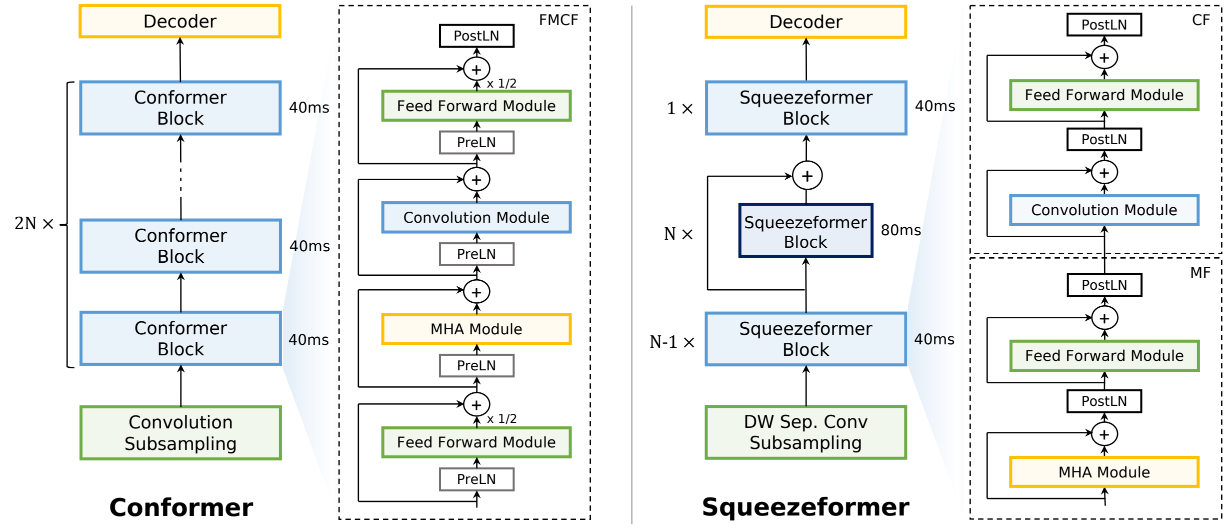


Figure 5.2: (Left) The Conformer architecutre and (Right) the Squeezeformer architecture which comprises of the Temporal U-Net structure for downsampling and upsampling of the sampling rate, the standard Transformer-style block structure that only uses Post-Layer Normalization, and the depthwise separable subsampling layer.

Table 5.1: Starting from Conformer as the baseline, we redesign the architecture towards Squeezeformer through a series of systematic studies on macro and micro architecture. Note that for each design change, the WER on LibriSpeech test-clean and test-other datasets improves consistently. For comparison, we include the number of parameters and FLOPs for a 30s input in the last two columns.

Model	Design change	test-clean	test-other	Params (M)	GFLOPs
Conformer-CTC-M	Baseline	3.20	7.90	27.4	71.7
	+ Temporal U-Net (Section 5.2)	2.97	7.28	27.5	57.0
	+ Transformer-style Block (Section 5.2)	2.93	7.12	27.5	57.0
	+ Unified activations (Section 5.2)	2.88	7.09	28.7	58.4
	+ Simplified LayerNorm (Section 5.2)	2.85	6.89	28.7	58.4
Squeezeformer-SM	+ DW sep. subsampling (Section 5.2)	2.79	6.89	28.2	42.7
Squeezeformer-M	+ Model scale-up (Section 5.2)	2.56	6.50	55.6	72.0

Macro-Architecture Design

We first focus on designing the macro structure of Squeezeformer, i.e., how the blocks and modules are organized in a global scale.

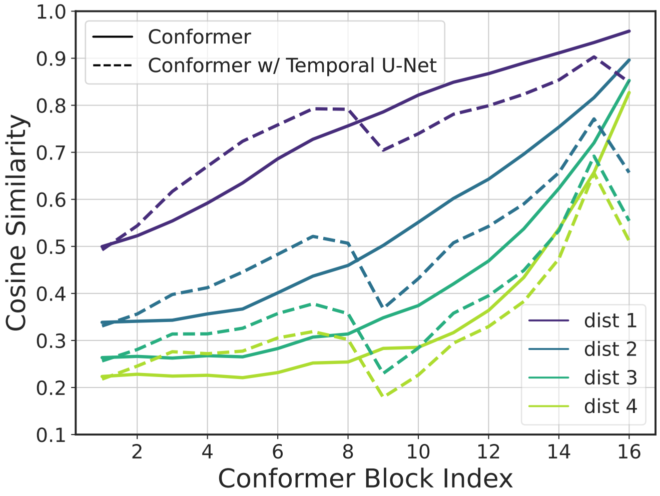


Figure 5.3: Cosine similarity between two embedding vectors of neighboring speech frames with varying adjacency distances across the Conformer blocks. The temporal dimension is downsampled after the 7th block and upsampled before the 16th block in the Temporal U-Net structure.

Temporal U-Net Architecture

The hybrid attention-convolution structure enables Conformer to capture both global and local interactions. However, the attention operation has a quadratic FLOPs complexity with respect to the input sequence length. We propose to lighten this extra overhead by computing attention over a reduced sequence length. In the Conformer model itself, the input sampling rate is reduced from 10ms to 40ms with a convolutional subsampling block at the base of the network. However, this rate is kept constant throughout the network, with all the attention and convolution operations operating at a constant temporal scale.

To this end, we begin by studying the temporal redundancy in the learned feature representations. In particular, we analyze how the learned feature embeddings per speech frame are differentiated through the Conformer model depth. We randomly sample 100 audio signals from LibriSpeech's dev-other dataset, and process them through the Conformer blocks, recording their per-block activations. We then measure the average cosine similarity between two neighboring embedding vectors. The results are plotted as the solid lines in Figure 5.3. We observe that the embeddings for the speech frames directly next to each other have an average similarity of 95% at the topmost layer, and even those 4 speech frames away from each other have a similarity of more than 80%. This reveals that there is an increasing temporal redundancy as inputs are processed through the Conformer blocks deeper in the network. We hypothesize that this redundancy in feature embedding vectors causes unnecessary computational overhead and that the sequence length can be reduced deeper in the network without loss in accuracy.

As our first macro-architecture improvement step, we change the Conformer model to incorporate subsampling of the embedding vectors after it has been processed by the early blocks of the model. In particular, we keep the sample rate to be 40ms up to the 7th block, and afterward we subsample to a rate of 80ms per input sequence by using a pooling layer. For the pooling layer we use a depthwise separable convolution with stride 2 and kernel size 3 to merge the redundancies across neighboring embeddings. This decreases the attention complexity by 4× and also reduces the redundancies of the features. This temporal downsampling shares similarities with computer vision models, which often downsample the input image spatially to save compute and develop hierarchical level features [93, 243, 57, 161], and with the approach of Efficient Conformer [13].

However, the temporal downsampling alone leads to an unstable and diverging training behaviour (Section 5.3). One possible reason for this is the lack of enough resolution for the decoder after subsampling the rate to 80ms. The decoder maps an embedding for each speech frame into a single label, e.g., character, and therefore requires sufficient resolution for successful decoding of the full sequence. Inspired from successful architectures for dense prediction in computer vision such as U-Net [222], we incorporate the Temporal U-Net structure to recover the resolution at the end of the network through an upsampling layer as shown in Figure 5.2. This upsampling block takes the embedding vectors processed by the 40ms and 80ms sampling rates, and produces an embedding with a rate of 40ms by adding them together via a skip connection. To the best of our knowledge, the closest work to our Temporal U-Net is the approach proposed in [208], in which the U-Net structure is incorporated into a fully-convolutional model to downsample sleep signals.

This change not only reduces the total FLOPs by 20% compared to Conformer[1], but also improves the test-other WER by 0.62% from 7.90% to 7.28% (Table 5.1, 2nd row). Furthermore, analyzing the cosine similarity shows that the Temporal U-Net architecture prevents the neighboring embeddings from becoming too similar to each others at the later blocks, in particular at the final block directly connected to the decoder, as shown in Figure 5.3 as the dashed lines.

Transformer-Style Block

The Conformer block consists of a sequence of feed-forward ('F'), multi-head attention (MHA, 'M'), convolution ('C'), and another feed-forward module ('F'). We denote this as the FMCF structure. Note that the convolutional kernel sizes in ASR models are rather large, e.g., 31 in Conformer, which makes its behaviour similar to attention in mixing global information. This is stark contrast to convolutional kernels in computer vision, which often have small 3×3 kernels and hence benefit greatly from attention's global processing. As such,

[1] The total FLOPs is for the entire model. If we just study the attention block, the Temporal U-Net structure reduces the FLOPs by 2.31× and 2.53× FLOPs reduction for processing 30s and 60s audio signals as compared to Conformer-CTC-M baseline, respectively.

placing the convolution and MHA module with a similar functionality back-to-back (i.e., the MC substructure) does not seem prudent. Hence, we consider an MF/CF structure, which is motivated by considering the convolution module as a local MHA module. Furthermore, we drop the Macaron structure [174], as MHA modules followed by feed-forward modules have been more widely adopted in the literature [266, 213, 43, 49]. In a nutshell, we simplify the architecture to be similar to the standard Transformer network and denote the blocks MF and CF substructures, as shown in Figure 5.2. This modification further improves the test-other WER by 0.16% from 7.28% to 7.12% and marginally improves the test-clean WER without affecting the FLOPs (Table 5.1, 3rd row).

Micro-Architecture Design

So far we have designed the macro structure of Squeezeformer by incorporating seminal architecture principles from computer vision and natural language processing into Conformer. In this subsection, we now focus on optimizing the micro structure of the individual modules. We show that we can further simplify the module architectures while improving both efficiency and performance.

Unified Activations

Conformer uses Swish activation for most of the blocks. However, it switches to a Gated Linear Unit (GLU) for its convolution module. Such a heterogeneous design seems overcomplicated and unnecessary. From a practical standpoint, the use of multiple activations complicates hardware deployment, as an efficient implementation requires dedicated logic design, look up tables, or custom approximations [314, 192, 134]. For instance, on low-end edge devices with no dedicated vector processing unit, supporting additional non-linear operations would require additional look up tables or advanced algorithms [68, 67]. To address this, we propose to replace the GLU activation with Swish, unifying the choice of activation function throughout the entire model. We keep the expansion rate for the convolution modules. As shown in the 4th row of Table 5.1, this change does not entail noticeable changes in WER and FLOPs but only simplifies the architecture.

Simplified Layer Normalizations

Continuing our micro-architecture improvements, we note that the Conformer model incorporates redundant Layer Normalizations (LayerNorm), as shown in Figure 5.4 (Left). This is because the Conformer model contains both a post-LayerNorm (postLN) that applies LayerNorm in between the residual blocks, as well as pre-LayerNorm (preLN) which applies LayerNorm inside the residual connection. While it is hypothesized that preLN stabilizes training and postLN benefits performance [272], these two modules used together lead to redundant back-to-back operations. Aside from the architectural redundancy, LayerNorm can be computationally expensive [314, 134] due to its global reduction operations.

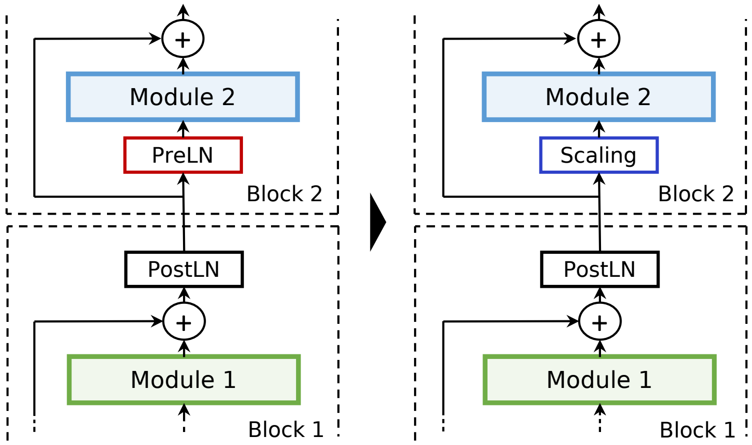


Figure 5.4: (Left) Back-to-back preLN and postLN at the boundary of the blocks. (Right) The preLN can be replaced with the learned scaling that readjusts the magnitude of the activation that goes into the subsequent module.

However, we found that naïvely removing the preLN or postLN leads to training instability and convergence failure (Section 5.3). Investigating the cause of failure, we observe that a typical trained Conformer model has orders of magnitude differences in the norms of the learnable scale variables of the back-to-back preLN and postLN. In particular, we found that the preLN would scale down the input signal by a large value, giving more weight to the skip connection. Therefore, it is important to use a scaling layer when replacing the preLN component to allow the network to control this weight. This idea is also on par with several training stabilization strategies in other domains. For instance, NF-Net [11] proposed adaptive (i.e., learnable) scaling before and after the residual blocks to stabilize training without normalization. Furthermore, DeepNet [272] also recently proposed to add non-trainable rule-based scaling to the skip connections to stabilize preLN in Transformers.

Inspired by these computer vision advancements, we propose to replace preLN with a learnable scaling layer that scales and shifts the activations, $\text{Scaling}(x) = \gamma x + \beta$, with learnable scale and bias vectors γ and β of the size of feature dimension. For homogeneity of architectural design, we then replace the preLN throughout all the modules with the postLN-then-scaling as illustrated in Figure 5.2 (Right) and make the entire model postLN-only. Note that the learned scaling parameters can be merged into the weights of the subsequent linear layer, as the architecture illustrated in Figure 5.2 (Right), and hence have zero inference cost. With the learned scaling, our model further improves the test-other WER by 0.20% from 7.09% to 6.89% (Table 5.1, 5th row).

Depthwise Separable Subsampling

We now shift our focus from the Conformer blocks to the subsampling block. While it is easy to overlook this single module at the beginning of the architecture, we note that it accounts for a significant portion of the overall FLOPs count, up to 28% for Conformer-CTC-M with a 30-second input. This is because the subsampling layer uses two vanilla convolution operations each of which has a stride 2. To reduce the overhead of this layer, we replace the second convolution operation with a depthwise separable convolution while keeping the kernel size and stride the same. We leave the first convolution operation as is since it is equivalent to a depthwise convolution with the input dimension 1. This saves an additional 22% of the baseline FLOPs without a test-other WER drop and even a 0.06% improvement in test-clean WER (Table 5.1, 6th row). An important point to note here is that generally depthwise separable convolutions are hard to efficiently map to hardware accelerators, in part due its low arithmetic intensity. However, given the large FLOPs reduction, we consistently observe an overall improvement in the total inference throughput of up to 1.34× as reported in Table 5.3, as compared to the baseline Conformer models.

We name our final model with all these improvements as *Squeezeformer-SM*. Compared to Conformer-CTC-M, our initial baseline, Squeezeformer-SM improves WER by 1.01% from 7.90% to 6.89% with 40% less FLOPs. Given the smaller FLOPs of Squeezeformer-SM, we also scale up the model to a similar FLOPs cost as Conformer-CTC-M. In particular, we scale both depth and width of the model together following the practice in [46]. Scaling up the model achieves additional test-other WER gain of 0.39% from 6.89% to 6.50% (Table 5.1, 7th row), and we name this architecture *Squeezeformer-M*.

5.3 Results

Experiment Setup

Models. Following the procedure described in Section 5.2, we construct Squeezeformer variants with different size and FLOPs: we apply the macro and micro-architecture changes in Section 5.2 to construct Squeezeformer-XS, SM, and ML from Conformer-S, M, and L, retaining the model size. Afterwards, we construct Squeezeformer-S, M, and L by scaling up each model to match the FLOPs of the corresponding Conformer. The detailed architecture configurations are described in Table 5.2.

While there are multiple options available for the decoder such as RNN-Transducer (RNN-T) [77] and Connectionist Temporal Classification (CTC) [78], we use a CTC decoder whose non-autoregressive decoding method benefits training and inference latency [178]. However, the main focus of this work is the model architecture design of the encoder, which can be orthogonal to the decoder type.

Another subtlety when evaluating models is the use of external language models (LM). In many prior works [125, 280, 86, 321, 175, 200], decoders are often augmented with external LMs such as pre-trained 4-gram or Transformer, which boost the final WER by re-scoring

Table 5.2: Detailed architecture configurations for Conformer-CTC (baseline) and Squeezeformer. For comparison, we include the number of parameters and FLOPs for a 30s input in the last two columns.

Model	# Layers	Dimension	# Heads	Params (M)	GFLOPs
Conformer-CTC-S	16	144	4	8.7	26.2
Squeezeformer-XS	16	144	4	9.0	15.8
Squeezeformer-S	18	196	4	18.6	26.3
Conformer-CTC-M	16	256	4	27.4	71.7
Squeezeformer-SM	16	256	4	28.2	42.7
Squeezeformer-M	20	324	4	55.6	72.0
Conformer-CTC-L	18	512	8	121.5	280.6
Squeezeformer-ML	18	512	8	125.1	169.2
Squeezeformer-L	22	640	8	236.3	277.9

the outputs in a more lexically accurate manner. However, we compare the results *without* external LMs to fairly compare the true representation power of the model architectures alone − external LMs can be incorporated as an orthogonal optimization afterward.

Training Details. Because the training recipes and codes for Conformer have not been open-sourced, we train it to reproduce the best performance numbers as possible. We train both Conformer-CTC and Squeezeformer on the LibriSpeech-960hr [201] for 500 epochs on Google's cloud TPUs v3 with batch size 1024 for the small and medium variants and 2048 for the large variants. We use AdamW [173] optimizer with weight decay 5e-4 for all models.

For learning rate scheduling, we extend the widely used Noam Annealing [266] to additionally support the number of steps to maintain the peak learning rate T_{peak} [238] and the decay rate d. That is, $\text{lr}(t) = \frac{\text{lr}_{\text{peak}} t}{T_0}$ for $t < T_0$, lr_{peak} for $T_0 \leq t < T_0 + T_{\text{peak}}$, and $\frac{\text{lr}_{\text{peak}} T_0^d}{(t - T_{\text{peak}})^d}$ for $t \geq T_0 + T_{\text{peak}}$, where t is the step number, lr_{peak} is the peak learning rate, and T_0 is the warmup steps. Note that the Noam annealing is a special case with $d = 0.5$ and $T_{\text{peak}} = 0$. We find warming up for 20 epochs, maintaining the peak learning rate for additional 160 epochs, and decaying with $d = 1$ work well in many cases, and fix these values throughout all experiments. We use the peak learning rate 2e-3, 1.5e-3, and {1, 0.5}e-3 for the small, medium, and large variants, respectively. We use SentencePiece [143] tokenizer with the vocabulary size 128, and the same dropout setting as in [82]. Finally, for data augmentation, we only use SpecAugment [202] with 2 frequency masks in [0, 27], and 5 (for all the small variants, Conformer-M and Squeezeformer-SM), 7 (for Squeezeformer-M) or 10 (for the large variants) time masks with the masking ratio of [0, 0.05].

Evaluation Details. We evaluate the final models on both clean and other datasets using CTC greedy decoding. For both Conformer-CTC and Squeezeformer, we additionally measure the throughput on a single NVIDIA's Tesla A100 GPU machine (GCP a2-highgpu-1g instance) using 30s audio inputs as an indicator of hardware performance. Here, we use

Table 5.3: WER (%) comparison on LibriSpeech dev and test datasets for Squeezeformer and other state-of-the-art CTC models for ASR including Conformer-CTC, QuartzNet [141], CitriNet [178], Transformer-CTC [165], and Efficient Conformer-CTC [13]. For comparison, we include the number of parameters, FLOPs, and throughput (Thp) on a single NVIDIA Tesla A100 GPU for a 30s input in the last three columns. *The performance numbers for Conformer-CTC are based on our own reproduction to the best performance as possible. All the other performance numbers are from the corresponding papers. †With and ‡without the grouped attention.

Model	dev-clean	dev-other	test-clean	test-other	Params (M)	GFLOPs	Thp (ex/s)
Conformer-CTC-S* [82]	4.21	10.54	4.06	10.58	8.7	26.2	613
QuartzNet 5x5 [141]	5.39	15.69	-	-	6.7	20.2	-
Citrinet 256 [178]	-	-	3.78	9.60	10.3	16.8	-
Squeezeformer-XS	**3.63**	**9.30**	**3.74**	**9.09**	9.0	15.8	763
Conformer-CTC-M* [82]	2.94	7.80	3.20	7.90	27.4	71.7	463
QuartzNet 5x10 [141]	4.14	12.33	-	-	12.8	38.5	-
QuartzNet 5x15 [141]	3.98	11.58	3.90	11.28	18.9	55.7	-
Citrinet 512 [178]	-	-	3.11	7.82	37.0	63.1	-
Eff. Conformer-CTC† [13]	-	-	3.57	8.99	13.2	26.0	-
Eff. Conformer-CTC‡ [13]	-	-	3.58	8.88	13.2	32.5	-
Squeezeformer-S	2.80	7.49	3.08	7.47	18.6	26.3	602
Squeezeformer-SM	**2.71**	**6.98**	**2.79**	**6.89**	28.2	42.7	558
Conformer-CTC-L* [82]	2.61	6.45	2.80	6.55	121.5	280.6	200
Citrinet 1024 [178]	-	-	**2.52**	6.22	143.1	246.3	-
Squeezeformer-M	2.43	6.51	2.56	6.50	55.6	72.0	431
Squeezeformer-ML	**2.34**	**6.08**	2.61	**6.05**	125.1	169.2	268
Transformer-CTC [165]	2.6	7.0	2.7	6.8	255.2	621.1	-
Squeezeformer-L	**2.27**	**5.77**	**2.47**	**5.97**	236.3	277.9	207

CUDA 11.5 and Tensorflow 2.5, and test with the largest possible batch size that saturates the machine.

Main Results

In Table 5.3 we compare the WER of Squeezeformer with Conformer-CTC and other state-of-the-art CTC-based ASR models including QuartzNet [141], CitriNet [178], Transformer [165], and Efficient Conformer [13] on the clean and other datasets. Note that the performance numbers for Conformer-CTC[2] are based on our own reproduction to the best performance as possible due to the absence of public training recipes or codes. For simplicity, we denote WER as test-clean/test-other without % throughout the section.

[2] The WER results exhibit some differences from the original paper [82] due to the difference in decoder. The original Conformer uses RNN-T decoder, which is known to generally result in better WER than CTC [327, 13, 178].

Table 5.4: Ablation studies for the design choices made in Squeezeformer, including Temporal U-Net, LayerNorm, and activation in the convolution module. *Without the upsampling layer, the model fails to converge.

Ablation	Model	dev-clean	dev-other
Ours	Squeezeformer-M	2.43	6.51
Temporal U-Net (Section 5.2),	No skip connection	2.78	7.38
	No upsampling	N/A*	N/A*
LayerNorm (Section 5.2)	PostLN only	5.60	14.00
	PreLN only	3.02	8.27
Convolution module (Section 5.2)	No Swish	2.53	6.73

Squeezeformer vs. Conformer. Our smallest model Squeezeformer-XS outperforms Conformer-CTC-S by 0.32/1.49 (3.74/9.09 vs. 4.06/10.58) with 1.66× FLOPs reduction. Compared with Conformer-CTC-M, Squeezeformer-S achieves 0.12/0.43 WER improvement (3.08/7.47 vs. 3.20/7.90) with 1.47× smaller size and 2.73× less FLOPs, and Squeezeformer-SM further improves WER by 0.41/1.01 (2.79/6.89 vs. 3.20/7.90) with a comparable size and 1.70× less FLOPs. Compared with Conformer-CTC-L, Squeezeformer-M shows 0.24/0.05 WER improvement (2.56/6.50 vs. 2.80/6.55) with significant size and FLOPs reductions of 2.18× and 3.90×, respectively, and Squeezeformer-ML shows 0.19/0.50 WER improvement (2.61/6.05 vs. 2.80/6.55) with a similar size and 1.66× less FLOPs. Finally, our largest model Squeezeformer-L improves WER by 0.33/0.58 upon Conformer-CTC-L with the same FLOPs count, achieving the state-of-the-art result of 2.47/5.97.

Squeezeformer vs. Other ASR Models. As can be seen in Table 5.3, our model consistently outperforms QuartzNet, CitriNet, and Transformer with comparable or smaller model sizes and FLOPs counts. A notable result is a comparison against Efficient-Conformer: our model outperforms the efficiently-designed Efficient Conformer by a large margin of 0.79/1.99 (2.79/6.89 vs. 3.58/8.88) with the same FLOPs count. The overall results are summarized as a plot in Figure 5.1 (Right) where Squeezeformer consistently outperforms other models across all FLOPs regimes.

Ablation Studies

In this section, we provide additional ablation studies for the design choices made for individual architecture components using Squeezeformer-M as the base model. See Table 5.4. Unless specified, we use the same hyperparameter settings as in the main experiment.

Temporal U-Net. In the 2nd row of Table 5.4, the model clearly underperforms by 0.35/0.87 without the skip connection from the downsampling layer to the upsampling layer. This shows that the high-resolution information collected in the early layers is critical for successful decoding. The 3rd row in Table 5.4 shows that our model completely fails to

Table 5.5: WER (%) comparison on TIMIT test split for Squeezeformer and Conformer-CTC that are trained on LibriSpeech with and without finetuning. For comparison, we also include the number of parameters and FLOPs.

Model	without finetuning	with finetuning	Params (M)	GFLOPs
Conformer-S	18.09	13.41	8.7	26.2
Squeezeformer-XS	16.31	12.89	9.0	15.8
Conformer-M	13.91	10.95	27.4	71.7
Squeezeformer-S	13.78	11.26	18.6	26.3
Squeezeformer-SM	13.65	10.50	28.2	42.7
Conformer-L	13.41	10.03	121.5	280.6
Squeezeformer-M	13.44	10.32	55.6	72.0
Squeezeformer-ML	11.35	9.96	125.1	169.2
Squeezeformer-L	12.92	9.76	236.3	277.9

converge without the upsampling layer due to training stability, even with several different peak learning rates of {0.5, 1.0, 1.5}e-3.

LayerNorm. In the 4th line of Table 5.4, we show that WER drops significantly by 3.17/7.49 when we apply the PostLN-only scheme without the learned scaling layer. Another alternative design choice is to apply the PreLN-only scheme without the learned scaling, which also results in a noticeable WER degradation of 0.59/1.76 as shown in the 5th line of Table 5.4. In both cases, the model fails to converge, so we report the best WER before divergence. The results suggest that the learned scaling layer plays a key role for training stabilization and better WER.

Convolution Module. When ablating the GLU activation in the convolution modules, another possible design choice is to drop it without replacing it with the Swish activation. This, however, results in 0.10/0.22 worse WER as shown in the last line of Table 5.4.

Transferability to Unseen Datasets

In Table 5.5, we additionally evaluate the transferability of Squeezeformer trained on LibriSpeech to unseen TIMIT [66] dataset with and without finetuning. In both cases, we used the same SentencePiece tokenizer as Librispeech training. For finetuning, we used the same learning rate scheduler as in Section 5.3 with the peak learning rate $\mathrm{lr}_{\mathrm{peak}}$ in {0.5, 1, 2, 5}e-4, 2 epochs of warmup (T_0), and 0 epoch of maintaining the peak learning rate (T_{peak}). All the other training recipes are the same as Section 5.3. We use Conformer-CTC as the baseline model to compare against, and we report WER measured on the test split. As can be seen in the table, the general trend aligns with the LibriSpeech results in Table 5.3: under smaller or same FLOPs and parameter counts, Squeezeformer outperforms Conformer-CTC, both with and without finetuning.

5.4 Related Work

The recent advancements in end-to-end ASR can be broadly categorized into (1) model architecture and (2) training methods.

Model Architecture for End-to-end ASR. The recent end-to-end ASR models are typically composed of an *encoder*, which takes as input a speech signal (i.e., sequence of speech frames) and extracts high-level acoustic features, and a *decoder*, which converts the extracted features from the encoder into a sequence of text. The model architecture of the encoder determines the representational power of an ASR model and its ability to extract acoustic features from input signals. Therefore, a strong architecture is critical for overall performance.

One of the popular choices for a backbone model architecture is convolutional neural network (CNN). End-to-end deep CNN models have been first explored in [329, 159], and further improved by introducing depth-wise separable convolution [242, 265, 107] in QuartzNet [141] and the Squeeze-and-Excitation module [112] in CitriNet [178] and ContextNet [90]. However, since CNNs often fail to capture global contexts, Transformer [266] models have also been widely adopted in backbone architectures due to their ability to capture long-range dependencies between speech frames [322, 167, 321, 165, 125]. Recently, [82] has proposed a novel model architecture named Conformer, which augments Transformers with convolutions to model both global and local dependencies efficiently. With the Conformer architecture as our starting point, we focus on designing a next-generation model architecture for ASR that is simpler, more accurate, and more efficient.

The hybrid attention-convolution architecture of Conformer has enabled the state-of-the-art results in many speech tasks. However, the quadratic complexity of the attention layer still proves to be cost prohibitive at larger sequence lengths. While different approaches have been proposed to reduce the cost of MHA in ASR [323, 324, 17, 238], their main focus is not changing the overall architecture design, and their optimizations can also be applied to our model, as they are orthogonal to the developments for Squeezeformer. Efficient-Conformer [13] introduces the progressive downsampling scheme and grouped attention to reduce the training and inference costs of Conformer. Our work incorporates a similar progressive downsampling, but also introduces an up-sampling mechanism with skip connections from the earlier layers inspired by the U-Net [222] architecture in computer vision and the U-Time [208] architecture for sleep signal analysis. We find this to be critical for training stability and overall performance. In addition, through systematic experiments, we completely refactor the Conformer block by carefully redesigning both the macro and micro-architectures.

Training Methodology for End-to-end ASR. In the past few years, various self-supervised learning methodologies based on contrastive learning [5, 330, 270] or masked prediction [108, 24, 4] have been proposed to push forward the ASR performance. While a model pre-trained with self-supervised tasks generally outperforms when finetuned on a target ASR task, training strategies are not the main focus in this work as they can be applied independently to the underlying architecture.

5.5 Conclusions

In this work, we performed a series of systematic ablation studies on the macro and micro architecture of the Conformer architecture, and we proposed a novel hybrid attention-convolution architecture that is simpler and consistently achieves better performance than other models for a wide range of computational budgets. The key novel components of Squeezeformer's macro-architecture is the incorporation of the Temporal U-Net structure which downsamples audio signals in the second half of the network to reduce the temporal redundancy between adjacent features and save compute, as well as the MF/CF block structure similar to the standard Transformer-style which simplifies the architecture and improves performance. Furthermore, the micro-architecture of Squeezeformer simplifies the activations throughout the model and replaces redundant LayerNorms with the scaled postLN, which is more efficient and leads to better accuracy. We also drastically reduce the subsampling cost at the beginning of the model by incorporating a depthwise separable convolution. We perform extensive testing of the proposed architecture and find that Squeezeformer scales very well across different model sizes and FLOPs regimes, surpassing prior model architectures when trained under the same settings. Our code along with the checkpoints for all of the trained models is open-sourced and available online [131].

Chapter 6

Efficiency in Agentic Applications: LLM Compiler for Parallel Function Calling

6.1 Introduction

Recent advances in the reasoning capability of Large Language Models (LLMs) have expanded the applicability of LLMs beyond content generation to solving complex problems [284, 139, 308, 7, 277, 336, 26, 306, 65]; and recent works have also shown how this reasoning capability can be helpful in improving accuracy for solving complex and logical tasks. The reasoning capability has also allowed function (i.e., tool) calling capability, where LLMs can invoke provided functions and use the function outputs to help complete their tasks. These functions range from a simple calculator that can invoke arithmetic operations to more complex LLM-based functions.

The ability of LLMs to integrate various tools and function calls could enable a fundamental shift in how we develop LLM-based software. However, this brings up an important challenge: *what is the most effective approach to incorporate multiple function calls?* A notable approach has been introduced in ReAct [307], where the LLM calls a function, analyzes the outcomes, and then reasons about the next action, which involves a subsequent function call. For a simple example illustrated in Figure 6.1 (Left), where the LLM is asked if Scott Derrickson and Ed Wood have the same nationality, ReAct initially analyzes the query and decides to use a search tool to search for Scott Derrickson. The result of this search (i.e., observation) is then concatenated back to the original prompt for the LLM to reason about the next action, which invokes another search tool to gather information about Ed Wood.

ReAct has been a pioneering work in enabling function calling, and it has been integrated into several frameworks [150, 169]. However, scaling this approach for more complex applications requires considerable optimizations. This is due to the sequential nature of ReAct, where it executes function calls and reasons about their observations one after the other.

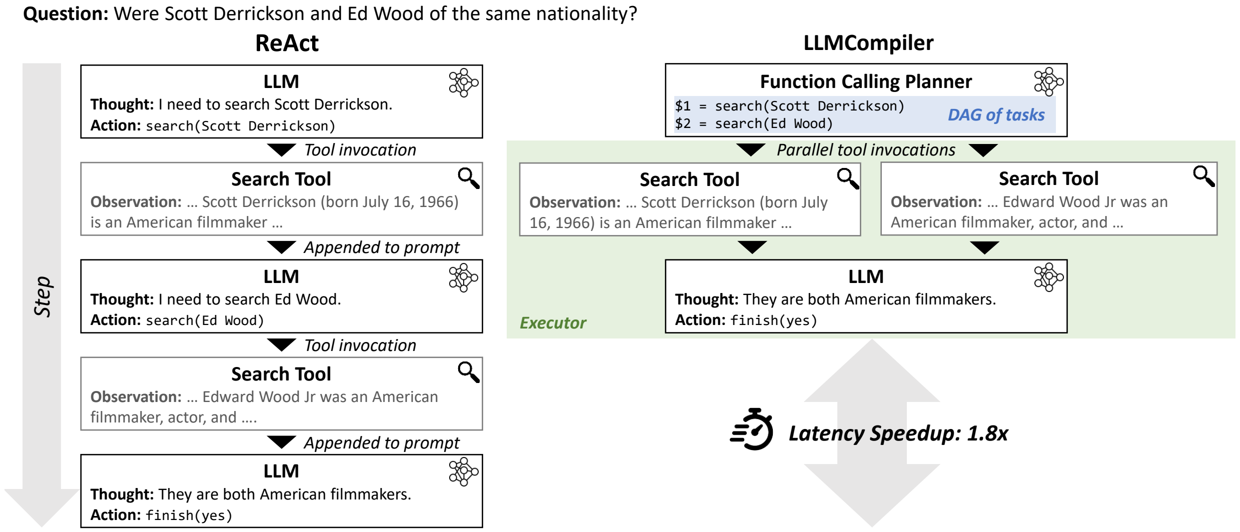

Question: Were Scott Derrickson and Ed Wood of the same nationality?
ReAct
LLM
Thought: I need to search Scott Derrickson.
Action: search(Scott Derrickson)
Tool invocation
Search Tool
Observation: … Scott Derrickson (born July 16, 1966) is an American filmmaker …
Appended to prompt
LLM
Thought: I need to search Ed Wood.
Action: search(Ed Wood)
Tool invocation
Search Tool
Observation: … Edward Wood Jr was an American filmmaker, actor, and ….
Appended to prompt
LLM
Thought: They are both American filmmakers.
Action: finish(yes)
Step
LLMCompiler
Function Calling Planner
$1 = search(Scott Derrickson)
$2 = search(Ed Wood)
DAG of tasks
Parallel tool invocations
Search Tool
Observation: … Scott Derrickson (born July 16, 1966) is an American filmmaker …
Search Tool
Observation: … Edward Wood Jr was an American filmmaker, actor, and …
LLM
Thought: They are both American filmmakers.
Action: finish(yes)
Executor
Latency Speedup: 1.8x

Figure 6.1: An illustration of the runtime dynamics of LLMCompiler, in comparison with ReAct [307], given a sample question from the HotpotQA benchmark [304]. In LLMCompiler (Right), the Planner first decomposes the query into several tasks with inter-dependencies. The Executor then executes multiple tasks in parallel, respecting their dependencies. Finally, LLMCompiler joins all observations from the tool executions to produce the final response. In contrast, sequential tool execution of the existing frameworks like ReAct (Left) leads to longer execution latency. In this example, LLMCompiler attains a latency speedup of 1.8× on the HotpotQA benchmark. While a 2-way parallelizable question from HotpotQA is presented here for the sake of simple visual illustration, LLMCompiler is capable of managing tasks with more complex dependency patterns (Figure 6.2 and Section 6.4).

This approach, along with the agent systems that extend ReAct [129, 308, 212, 225, 252], may lead to inefficiencies in latency and cost, due to the sequential function calling and repetitive LLM invocations for each reasoning and action step. Furthermore, while dynamic reasoning about the observations has benefits in certain cases, concatenating the outcomes of intermediate function calls could disrupt the LLM's execution flow, potentially reducing accuracy [297]. Common failure cases include repetitive invocation of the same function, which is also highlighted in the original paper [307], and early stopping based on the partial intermediate results, as will be further discussed in Section 6.4 and Section 6.5.

To address this challenge, we draw inspiration from classical compilers, where optimizing instruction executions in traditional programming languages has been extensively explored. A key optimization technique in compilers involves identifying instructions that can be executed in parallel and effectively managing their dependencies. Similarly, one can envision a compiler, tailored for LLM function calling, which can efficiently orchestrate various function calls and their dependencies. This shares a similar philosophy with the recent studies that align LLMs with computer systems [126, 198]. To this end, we introduce LLMCompiler, a novel framework that enables parallel multi-tool execution of LLMs across different models and workloads. To the best of our knowledge, LLMCompiler is the first framework to optimize the orchestration of LLM function calling that can not only improve latency and cost, but also accuracy, by minimizing interference from the outputs of intermediate function calls. In more detail, we make the following contributions:

- We introduce LLMCompiler, an LLM compiler that optimizes the parallel function calling performance of LLMs. At a high level, this is achieved by introducing three key components: (i) a Function Calling Planner (Section 6.2) that identifies an execution flow; (ii) a Task Fetching Unit (Section 6.2) that dispatches the function calls in parallel; (iii) an Executor (Section 6.2) that executes the dispatched tasks using the associated functions.
- We evaluate LLMCompiler on *embarrassingly parallel* patterns using HotpotQA [304] and Movie Recommendation [249], where we observe 1.80×/3.74× speedup and 3.37×/ 6.73× cost reduction compared to ReAct (Section 6.4).
- To test the performance on more complex patterns, we introduce a new benchmark called ParallelQA which includes various non-trival function calling patterns. We show up to 2.27× speedup, 4.65× cost reduction, and 9% improved accuracy compared to ReAct (Section 6.4).
- We evaluate LLMCompiler's capability in dynamic replanning, which is achieved through a feedback loop from the Executor back to our Function Calling Planner. For the Game of 24 [308], which requires repeated replanning based on the intermediate results, LLMCompiler demonstrates a 2× speedup compared to Tree-of-Thoughts (Section 6.4).
- We show that LLMCompiler can explore the interactive decision-making environment effectively and efficiently. On WebShop, LLMCompiler achieves up to 101.7× speedup and 25.7% improved success rate compared to the baselines (Section 6.4).

6.2 Methodology

To illustrate the components of LLMCompiler, we use a simple 2-way parallel example in Figure 6.2. To answer "How much does Microsoft's market cap need to increase to exceed Apple's market cap?," the LLM first needs to conduct web searches for both companies' market caps, followed by a division operation. While the existing frameworks, including ReAct, perform these tasks sequentially, it is evident that they can be executed in parallel. The key question is how to automatically determine which tasks are parallelizable and which are interdependent, so we can orchestrate the execution of the different tasks accordingly. LLMCompiler accomplishes this through a system that consists of the following three components: a Function Calling Planner (Section 6.2) that generates a sequence of tasks and their dependencies; a Task Fetching Unit (Section 6.2) that replaces arguments based on intermediate results and fetches the tasks; and an Executor (Section 6.2) that executes the tasks with associated tools. To use LLMCompiler, users are only required to provide tool definitions, and optional in-context examples for the Planner, as will be further discussed in Section 6.3.

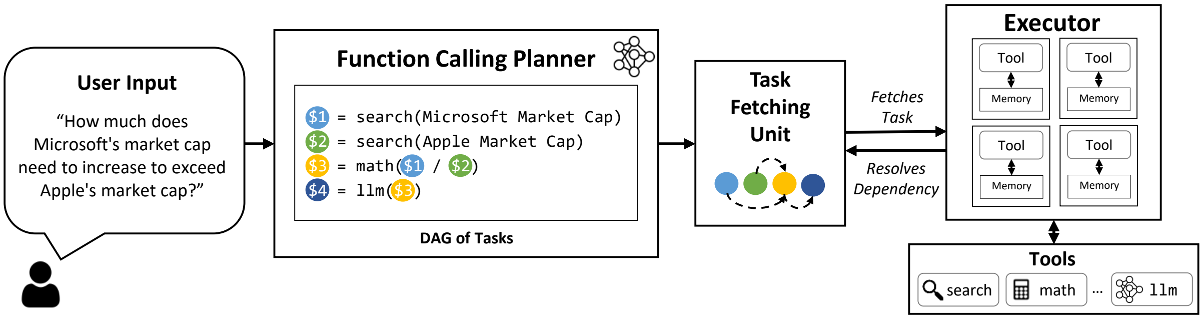


Figure 6.2: Overview of the `LLMCompiler` framework. The Function Calling Planner generates a DAG of tasks with their inter-dependencies. These tasks are then dispatched by the Task Fetching Unit to the Executor in parallel based on their dependencies. In this example, Task `$1` and `$2` are fetched together for parallel execution of two independent search tasks. After each task is performed, the results are forwarded back to the Task Fetching Unit to unblock the dependent tasks after replacing their placeholder variables (e.g., the variable `$1` and `$2` in Task `$3`) with actual values. Once all tasks have been executed, the final answer is delivered to the user.

Function Calling Planner

The Function Calling Planner is responsible for generating a sequence of tasks to be executed along with any dependency among them. For instance, Tasks `$1` and `$2` in Figure 6.2 are two independent searches that can be performed in parallel. However, Task `$3` has a dependency on the outcomes of the first and second searches. Therefore, the Planner's role is to automatically identify the necessary tasks, their input arguments, as well as their inter-dependencies using the sophisticated reasoning capability of LLMs, essentially forming a directed acyclic graph of task dependencies. If a task is dependent on a preceding task, it incorporates a placeholder variable, such as `$1` in Task 3 of Figure 6.2, which will later be substituted with the actual output from the preceding task (Section 6.2).

The Planner in `LLMCompiler` leverages LLMs' reasoning capability to decompose tasks from natural language inputs. To achieve this, the Planner LLM incorporates a pre-defined prompt that guides it on how to create dependency graphs and to ensure correct syntax (see Appendix D.5 for details). Besides this, users also need to supply tool definitions and optional in-context examples for the Planner. These examples provide detailed demonstrations of task decomposition specific to a problem, helping the Planner to better understand the rules. Further details on user-supplied information for `LLMCompiler` are elaborated in Section 6.3. In Section 6.3, we introduce an additional optimization for the Planner that streams tasks as soon as they are created, instead of waiting to complete the entire planning process.

Task Fetching Unit

The Task Fetching Unit, inspired by the instruction fetching units in modern computer architectures, fetches tasks to the Executor as soon as they are ready for (parallel) execution based on a greedy policy. Another key functionality is to replace variables with the actual outputs from preceding tasks, which were initially set as placeholders by the Planner. For the example in Figure 6.2, the variable `$1` and `$2` in Task `$3` would be replaced with the actual market cap of Microsoft and Apple. This can be implemented with a simple fetching and queuing mechanism without a dedicated LLM.

Executor

The Executor asynchronously executes tasks fetched from the Task Fetching Unit. As the Task Fetching Unit guarantees that all the tasks dispatched to the Executor are independent, it can simply execute them concurrently. The Executor is equipped with user-provided tools, and it delegates the task to the associated tool. These tools can be simple functions like a calculator, Wikipedia search, or API calls, or they can even be LLM agents that are tailored for a specific task. As depicted in the Executor block of Figure 6.2, each task has dedicated memory to store its intermediate outcomes, similar to what typical sequential frameworks do when aggregating observations as a single prompt [307]. Upon completion of the task, the final results are forwarded as input to the tasks dependent on them.

Dynamic Replanning

In various applications, the execution graph may need to adapt based on intermediate results that are a priori unknown. An analogy in programming is branching, where the path of execution is determined only during runtime, depending on which branch conditions are satisfied. Such dynamic execution patterns can also appear with LLM function calling. For simple branching (e.g., if-else statements) one could statically compile the execution flow and choose the right dynamically based on the intermediate results. However, for more complex branching it may be better to do a recompilation or replanning based on the intermediate results.

When replanning, the intermediate results are sent back from the Executor to the Function Calling Planner which then generates a new set of tasks with their associated dependencies. These tasks are then sent to the Task Fetching Unit and subsequently to the Executor. This cycle continues until the desired final result is achieved and can be delivered to the user. We show an example use case of this in Section 6.4 for solving the Game of 24 using the Tree-of-Thoughts approach.

6.3 LLMCompiler Details

User-Supplied Information

LLMCompiler requires two inputs from the user:

1. **Tool Definitions**: Users need to specify the tools that LLMs can use, including their descriptions and argument specifications. This is essentially the same requirement as other frameworks like ReAct and OpenAI function calling.
2. **In-context Examples for the Planner**: Optionally, users can provide LLMCompiler with examples of how the Planner should behave. For instance, in the case of Figure 6.2, users may provide examples illustrating expected inter-task dependencies for certain queries. These examples can assist the Planner LLM in understanding how to use various tools and generate the appropriate dependency graph for incoming inputs in the correct format. In Appendix D.4, we include the examples that we used in our evaluations.

Streamed Planner

The Planner may incur a non-trivial overhead for user queries that involve a lot of tasks as it blocks the Task Fetching Unit and the Executor, which must wait for the Planner output before initiating their processes. However, analogous to instruction pipelining in modern computer systems, this can be mitigated by enabling the Planner to asynchronously stream the dependency graph, thereby allowing each task to be immediately processed by the Executor as soon as its dependencies are all resolved. In Table D.1, we present a latency comparison of LLMCompiler with and without the streaming mechanism across different benchmarks. The results demonstrate consistent latency improvements with streaming. Particularly, in the ParallelQA benchmark, the streaming feature leads to a latency gain of up to 1.3×. This is attributed to the math tool's longer execution time for ParallelQA, which can effectively hide the Planner's latency in generating subsequent tasks, unlike the shorter execution times of the search tool used in HotpotQA and Movie Recommendation.

6.4 Results

In this section, we evaluate LLMCompiler using a variety of models and problem types. We use both the proprietary GPT models and the open-source LLaMA-2 model, with the latter demonstrating LLMCompiler's capability in enabling parallel function calling in open-source models. Furthermore, there are various types of parallel function calling patterns that can be addressed with LLMs. This ranges from embarrassingly parallel patterns, where all tasks can be executed in parallel without any dependencies between them, to more complex dependency patterns, as illustrated in Figure 6.3. Importantly, we also assess LLMCompiler on the Game of 24 benchmark, which involves dynamic replanning based on intermediate results,

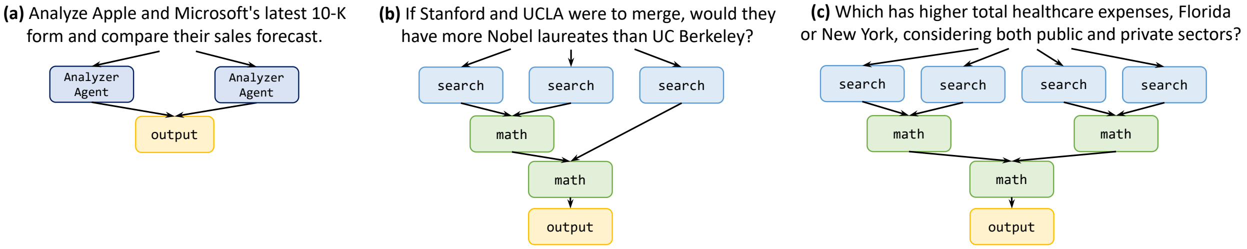

(a) Analyze Apple and Microsoft's latest 10-K form and compare their sales forecast.
Analyzer Agent
Analyzer Agent
output
(b) If Stanford and UCLA were to merge, would they have more Nobel laureates than UC Berkeley?
search
search
search
math
math
output
(c) Which has higher total healthcare expenses, Florida or New York, considering both public and private sectors?
search
search
search
search
math
math
math
output

Figure 6.3: Examples of questions with different function calling patterns and their dependency graphs. HotpotQA and Movie Recommendation datasets exhibit pattern (a), and ParallelQA dataset exhibits patterns (b) and (c), among other patterns. In (a), we need to analyze each company's latest 10-K. In (b), we need three searches for each school, followed by one addition and one comparison operation. In (c), we need to search for each state's annual healthcare spending in each sector, sum each state's spending, and then perform a comparison.

highlighting its adaptability to dynamic dependency graphs. Finally, we apply `LLMCompiler` to the WebShop benchmark to showcase its potential in decision-making tasks. Overall, we start presenting results for simple execution patterns, and then we move to more complex ones.

Embarrassingly Parallel Function Calling

The simplest scenario involves an LLM using a tool repeatedly for independent tasks such as conducting parallel searches or analyses to gather information on different topics, like the pattern depicted in Figure 6.3 (a). While these tasks are independent of each other and can be executed in parallel, ReAct, along with other LLM solutions as they stand, would need to run sequentially. This leads to increased latency and token consumption due to its frequent LLM invocations for each tool usage, as also illustrated in Figure 6.1. In this section, we demonstrate how `LLMCompiler` can identify parallelizable patterns and execute independent tasks concurrently to resolve this issue. To do so, we use the following two benchmarks:

- **HotpotQA:** A dataset that evaluates multi-hop reasoning [304]. We only use the comparison dev set. This contains 1.5k questions comparing two different entities, thus exhibiting a 2-way embarrassingly parallel execution pattern. An example question is shown in Figure 6.1.
- **Movie Recommendation:** A dataset with 500 examples that asks to identify the most similar movie out of four options to another set of four movies, exhibiting an 8-way embarrassingly parallel pattern [249].

Experimental Setups. As a baseline method, we compare `LLMCompiler` with ReAct. We follow the ReAct setup [307] using the same Wikipedia search tool that LLMs can use to search for information. We did not include the lookup tool since it is not relevant to our

Table 6.1: Accuracy and latency comparison of `LLMCompiler` compared to the baseline on different benchmarks, including HotpotQA, Movie Recommendation, our custom dataset named ParallelQA, and the Game of 24. For HotpotQA and Movie Recommendation, we frequently observe looping and early stopping (Section 6.4). To minimize these behaviors as much as possible, we incorporated ReAct-specific prompting which we denote as ReAct†. ReAct (without †) indicates the original results without this prompting. We do not include the latency for the original ReAct since looping and early stopping make precise latency measurement difficult.

Benchmark	**Method**	**GPT (Closed-source)**			**LLaMA-2 70B (Open-source)**		
		Acc (%)	Latency (s)	Speedup	Acc (%)	Latency (s)	Speedup
HotpotQA	ReAct	61.52	-	-	54.74	-	-
	ReAct†	62.47	7.12	1.00×	54.40	13.44	1.00×
	OAI Parallel Func.	62.05	4.42	1.61×	-	-	-
	`LLMCompiler`	62.00	**3.95**	**1.80×**	57.83	**9.58**	**1.40×**
Movie Rec.	ReAct	68.60	-	-	70.00	-	-
	ReAct†	72.47	20.47	1.00×	70.60	33.37	1.00×
	OAI Parallel Func.	77.00	7.42	2.76×	-	-	-
	`LLMCompiler`	77.13	**5.47**	**3.74×**	77.80	**11.83**	**2.82×**
ParallelQA	ReAct	89.09	35.90	1.00×	59.59	15.47	1.00×
	OAI Parallel Func.	87.32	19.29	1.86×	-	-	-
	`LLMCompiler`	89.38	**16.69**	**2.15×**	68.14	**26.20**	**2.27×**
Game of 24	Tree-of-Thoughts	74.00	241.2	1.00×	30.00	952.06	1.00×
	`LLMCompiler`	75.33	**83.6**	**2.89×**	32.00	**456.02**	**2.09×**

Table 6.2: Input and output token consumption as well as the estimated cost on HotpotQA, Movie Recommendation, and our custom dataset named ParallelQA. The cost is computed based on the pricing table of the GPT models used for each benchmark.

Benchmark	Method	Tokens		Cost	Cost
		In.	Out.	($/1k)	Red.
HotpotQA	ReAct	2900	120	5.00	1.00×
	OAI Para. Func.	2500	63	2.66	1.87×
	`LLMCompiler`	1300	80	**1.47**	**3.37×**
Movie Rec.	ReAct	20000	230	20.46	1.00×
	OAI Para. Func.	5800	160	6.14	3.33×
	`LLMCompiler`	2800	115	**3.04**	**6.73×**
ParallelQA	ReAct	46000	470	480	1.00×
	OAI Para. Func.	25000	370	260	1.81×
	`LLMCompiler`	9200	340	**103**	**4.65×**

problem setting. We have optimized the prompt and in-context examples for both ReAct and `LLMCompiler` to the best of our abilities. For all experiments across these datasets, we use gpt-3.5-turbo (1106 release). For the experiments using GPT, we additionally report the results using OpenAI's parallel function calling capability, which was announced concurrently with our work. We also show how `LLMCompiler` can be effectively combined with the open-source LLaMA-2 70B model to provide the model with parallel function calling capabilities. For all experiments, we have measured accuracy, end-to-end latency, as well as input and output token usage. See Appendix D.1 for details on experimental setups.

Accuracy and Latency. We report the accuracy, end-to-end latency, and relative speedup of `LLMCompiler` compared to ReAct in Table 6.1. First, we observe that ReAct consistently achieves lower accuracy compared to OpenAI parallel function calling and `LLMCompiler`. We identify two main failure modes in ReAct: (1) the tendency for redundant generation of prior function calls, a point also noted in the original ReAct paper [307]; and (2) premature early stopping based on the incomplete intermediate results. In Section 6.5, we offer a detailed analysis demonstrating how these two prevalent failure cases significantly hurt ReAct's accuracy, and how they can be resolved with `LLMCompiler`, leading to an accuracy enhancement of up to 7 – 8%. Furthermore, we have conducted interventional experiments in which we incorporated ReAct-specific prompts to avoid repetitive function calls and early stopping. ReAct† in Table 6.1 refers to ReAct *with* this ReAct-specific prompt. The ReAct-specific prompt yields a general accuracy improvement with ReAct† as compared to the original ReAct. Nevertheless, `LLMCompiler` still demonstrates on-par and better accuracy than ReAct†, as such prompting does not serve as a perfect solution to completely avoiding the erroneous behavior of ReAct.

Additionally, when compared to ReAct†, `LLMCompiler` demonstrates a noticeable speedup of 1.80× and 1.40× on HotpotQA with GPT and LLaMA, respectively. Similarly, `LLMCompiler` demonstrates 3.74× and 2.82× speedup on Movie Recommendation with each model. Note that we benchmark the latency of `LLMCompiler` against that of ReAct† since the repeating and early stopping behavior of the original ReAct as discussed above makes its latency unpredictable and unsuitable for a fair comparison. `LLMCompiler` demonstrates a speedup of up to 35% compared to OpenAI parallel function calling whose latency gain over ReAct is 1.61× and 2.76× on each benchmark.[1]

Costs. Another important consideration of using LLMs is cost, which depends on the input and output token usage. The costs for GPT experiments are provided in Table 6.2. `LLMCompiler` is more cost-efficient than ReAct for cost, as it involves less frequent LLM invocations. Interestingly, `LLMCompiler` also outperforms the recent OpenAI parallel function calling in cost efficiency. This is because `LLMCompiler`'s planning phase is more prompt

[1] Unfortunately, we are unable to conclude why this is the case, as OpenAI has not publicly disclosed any details about their function calling mechanism. One speculation is that there might be additional overheads to validate the function and argument names and to convert them into a system prompt. Nevertheless, we have seen a consistent trend with multiple runs over several days.

length efficient than that of OpenAI parallel function calling since our Planner's in-context examples are rather short and only include plans, not observations (see Appendix D.5).

Parallel Function Calling with Dependencies

The cases considered above are rather simple, as only one tool is used and all tasks can be executed independently of one another. However, similar to code execution in traditional code blocks, we may encounter function calling scenarios that involve more complex dependencies. To systematically evaluate the capability to plan out function calling in scenarios that involve complex task dependencies, we have designed a custom benchmark called ParallelQA. This benchmark is designed to incorporate non-trivial function calling patterns, including three different types of patterns in Figure 6.3 (b) and (c). Inspired by the IfQA benchmark [315], ParallelQA contains 113 examples that involve mathematical questions on factual attributes of various entities. In particular, completing the task requires using two tools (i.e., search and math tools), with the second tool's argument depending on the result of the first tool's output. We have meticulously included questions that are answerable only with information from Wikipedia's first paragraph, effectively factoring out the failure cases due to unsuccessful searches. See Appendix D.6 for more details in ParallelQA.

Experimental Setups. Similar to Section 6.4, we use ReAct [307] as the main baseline. Here, both ReAct and `LLMCompiler` are equipped with two tools: (1) the search tool, identical to the one mentioned in Section 6.4; and (2) the math tool, which solves mathematical problems. The math tool is inspired by the Langchain [150]'s `LLMMathChain`, which uses an LLM as an agent that interprets input queries and invokes the `numexpr` function with the appropriate formula. This enables the math chain to address a broad spectrum of math problems that are written both in mathematical and verbal form. See Appendix D.1 for more details on experimental setups.

Accuracy and Latency. As shown in the ParallelQA row of Table 6.1, `LLMCompiler` arrives at the final answer with an average speedup of 2.15× with gpt-4-turbo and 2.27× with LLaMA-2 70B, by avoiding sequential execution of the dependency graphs. Beyond the latency speedup, we observe higher accuracy of `LLMCompiler` with the LLaMA-2 model as compared to that of ReAct, due to the reasons discussed in Section 6.4. Particularly in the LLaMA-2 experiment, where `LLMCompiler` achieves around a 9% increase in accuracy, we note that ∼20% of the examples experienced repetitive function calls with ReAct, aligning with our observations from the accuracy analysis detailed in Section 6.5. Additionally, a comprehensive analysis of `LLMCompiler`'s failure cases is provided in Section 6.5, where we note minimal Planner failures, highlighting `LLMCompiler`'s effectiveness in breaking down problems into complex multi-task dependencies.

Cost. Similar to Section 6.4, `LLMCompiler` demonstrates substantial cost reductions of 4.65× and 2.57× compared to ReAct and OpenAI's parallel function calling, respectively, as indicated in Table 6.2. This efficiency stems from `LLMCompiler`'s reduced frequency of LLM invocations, which is also the case with OpenAI's parallel function calling, which is limited to planning out immediate parallelizable tasks, not the entire dependency graph. For example, in Figure 6.3 (c), OpenAI's method would necessitate three distinct LLM calls for initial search tasks, following math tasks, and the final math task. In contrast, `LLMCompiler` achieves this with a single LLM call, planning all tasks concurrently.

Parallel Function Calling with Replanning

In the previous sections, we have discussed cases in which dependency graphs can be determined statically. However, there are cases where dependency graphs need to be constructed dynamically depending on intermediate observations. Here, we consider one such dynamic approach in the context of the Game of 24 with the Tree-of-Thoughts (ToT) strategy proposed in [308]. The Game of 24 is a task to generate 24 using a set of four numbers and basic arithmetic operations. For example, from the numbers 2, 4, 4, and 7, a solution could be $4 \times (7 - 4) \times 2 = 24$. ToT approaches this task through two iterative LLM processes: (i) the thought proposer generates candidate partial solutions by selecting two numbers and applying an operation (e.g. 2, 3, 7 from 2, 4, 4, 7 by calculating 7 - 4); (ii) the state evaluator assesses the potential of each candidate. Only the promising candidates are then processed in subsequent iterations of the thought proposer and state evaluator until 24 is reached. Details about the Game of 24 benchmark and the ToT strategy can be found in Appendix D.7.

While ToT achieves significant improvement at solving the Game of 24, its sequential, breadth-first search approach through the state tree can be time-consuming. `LLMCompiler` offers a faster alternative by enabling parallel execution of the thought proposer and the subsequent feasibility evaluator, akin to a parallel beam search method.

Experimental Setups. Although `LLMCompiler` offers latency advantages, solving this problem with a single static graph is not feasible, as the Planner cannot plan out the thought proposing stage before identifying the selected candidates from the state evaluator of the previous iteration. Consequently, the Planner is limited to planning only within one iteration at a time. To address this, we resort to `LLMCompiler`'s replanning capability. In particular, `LLMCompiler` is equipped with three tools: `thought_proposer` and `state_evaluator`, which are both LLMs adapted from the original ToT framework, and `top_k_select`, which chooses the top k candidates from the `thought_proposer` based on the `state_evaluator`'s assessment. After all these tools are executed, `LLMCompiler` can decide to "replan" if no proposal reaches 24, triggering the Planner to devise new plans using the shortlisted states from `top_k_select` of the previous iteration. In this way, `LLMCompiler` can dynamically regenerate plans of each iteration, being able to tackle highly complex tasks that require iterative replanning based on the outcomes of previous plans.

To evaluate `LLMCompiler`'s performance on the Game of 24, we use 100 different instances of the game. For each problem, we consider the output as successful if its operations are valid and yield 24 while also using the provided numbers exactly once each. Further details on experiment setups are outlined in Appendix D.1.

Success Rate and Latency. In the last two rows of Table 6.1, we explore the latency and success rate of `LLMCompiler` in comparison to the baseline described in [308] on the Game of 24 benchmark. With the gpt-4 model, `LLMCompiler` demonstrates a 2.89× enhancement in latency while slightly improving the success rate compared to the baseline. Similarly, when applied with the LLaMA-2 model, `LLMCompiler` shows a 2.01× improvement in latency, again without compromising on success rate. These results demonstrate not only a significant latency reduction without quality degradation, but also the replanning capability of `LLMCompiler` for solving complex problems.

Application: `LLMCompiler` in Interactive Decision Making Tasks

In this section, we demonstrate that `LLMCompiler` can explore language-based interactive environments effectively by benchmarking `LLMCompiler` on WebShop [309]. As highlighted in [239, 307, 309], WebShop exhibits considerable diversity, which requires extensive exploration to purchase the most appropriate item. While recent work feature advanced exploration strategies and show promising results [334, 176], their approaches are largely based on a sequential and extensive tree search that incurs significant latency penalties. Here, `LLMCompiler` showcases an exploration strategy that is both effective and efficient with the use of parallel function calling. Our method enables broader exploration of items in the environment, which improves success rate compared to ReAct. At the same time, this exploration can be parallelized, yielding up to 101.7× speedup against baselines that perform sequential exploration.

Experimental Setups. We evaluate `LLMCompiler` against three baselines on this benchmark, ReAct [307], LATS [334], and LASER [176], using 500 WebShop instructions. The evaluation metrics are success rate, average score, and latency. More details of the WebShop environment and the baseline methods are provided in Appendix D.8. For this experiment, `LLMCompiler` is equipped with two tools: `search` and `explore`. The `search` function triggers the model to generate and dispatch a query that returns a list of typically ten items from the Webshop environment. The `explore` function then clicks through links for each of the found items and retrieves information about options, prices, attributes, and features that are available. Finally, based on the gathered information, `LLMCompiler` decides on the item that best matches the input instruction for purchasing. Further details on experiments can be found in Appendix D.1.

Performance and Latency. Our approach significantly outperforms all baseline models as shown in Table 6.3. When using gpt-3.5-turbo, `LLMCompiler` achieves a 28.4% and 6% improvement in success rate against ReAct and LATS; with gpt-4, our method improves upon ReAct and LASER by 20.4% and 5.6%, respectively. In terms of latency,

Table 6.3: Performance and Latency Analysis for WebShop. We evaluate LLMCompiler with two models: gpt-4 and gpt-3.5-turbo and compare LLMCompiler against three baselines: ReAct, LATS, and LASER. We report success rate and average score in percentage. We reproduce the success rate and average score for ReAct, while those for LATS and LASER are from their papers. N denotes the number of examples used for evaluation.

Model	Method	Succ. Rate	Score	Latency (s)	N
gpt-3.5-turbo	ReAct	19.8	54.2	5.98	500
	LATS	38.0	75.9	1066	50
	LLMCompiler	44.0	72.8	10.72	50
	LLMCompiler	48.2	74.2	10.48	500
gpt-4-0613	ReAct	35.2	58.8	19.90	500
	LASER	50.0	75.6	72.16	500
	LLMCompiler	55.6	77.1	26.73	500

LLMCompiler exhibits a 101.7× and 2.69× speedup against LATS and LASER. While we note that LLMCompiler execution is slightly slower than ReAct on this benchmark, mainly due to the Planner overhead, we also highlight that the gains in success rate far outweigh the minor latency penalty.

We further delve into why LLMCompiler attains such an improved success rate and score compared to ReAct. Based on our observations, we discover that the ReAct agent tends to commit to a decision with imperfect information, a scenario that can arise when the agent has not gathered sufficient details about the features and options available for items. This observation was also noted in [239] – without exploring more items in the environment, the agent struggles to differentiate between seemingly similar choices, ultimately failing to make the correct decision. In contrast, LLMCompiler undergoes further exploration by visiting all ten items found by search and retrieving relevant information about each item. We find that employing an effective search strategy is critical to decision-making tasks such as the WebShop benchmark.

The relatively high performance of LATS can also be explained in terms of its exploration scheme. In this framework, the agent executes a brute-force search through the state and action space of Webshop, exploring as many as 30 trajectories before making the final purchase. While this approach provides richer information for decision-making, the end-to-end execution becomes prohibitively slow.

We report that our method, LLMCompiler, outperforms LASER by an average score of 1.5. When compared to LATS, this score is within the standard deviation range of our method. The average score for LLMCompiler, along with its standard deviation, is 72.8 ± 4.01 for gpt-3.5-turbo. Further note that while the performance differences are marginal, our method exhibits significant execution speedup, 101.7× over LATS and 2.69× over LASER.

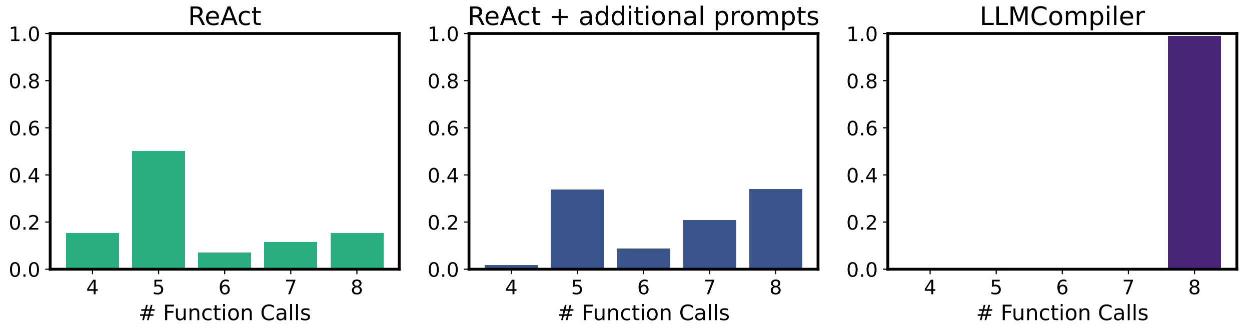


Figure 6.4: Distributions of the number of function calls when running the Movie Recommendation benchmark on ReAct (Left), ReAct with specific prompts to avoid early stopping (Middle, corresponding to ReAct† in Table 6.1), and `LLMCompiler` (Right). `LLMCompiler` (Right) consistently completes the search for all 8 movies, whereas ReAct (Left) often exit early, demonstrated by about 85% of examples stopping early. Although the custom prompts shift ReAct's histogram to higher function calls (Middle), they still fall short of ensuring comprehensive searches for all movies. gpt-3.5-turbo is used for the experiment.

6.5 Discussion

Accuracy Analysis: ReAct vs. `LLMCompiler`

In this section, we conduct a detailed analysis that compares the accuracy of both ReAct and `LLMCompiler`, highlighting two failure cases that are prevalent in ReAct: (i) premature early stopping; and (ii) repetitive function calls. Furthermore, we demonstrate that while those failure cases negatively impact the ReAct accuracy, they can be effectively addressed by `LLMCompiler`, thereby yielding the improved accuracy of our framework. We analyze two specific scenarios: the Movie Recommendation evaluation with GPT, where ReAct often prematurely stops, leading to significantly lower accuracy compared to `LLMCompiler` (68.60 vs. 77.13 in Table 6.1); and the HotpotQA evaluation with LLaMA-2 70B, where ReAct's repetitive function calls result in a notable accuracy degradation compared to `LLMCompiler` (70.00 vs. 77.80 in Table 6.1).

Premature Early Stopping of ReAct. ReAct frequently suffers from premature early stopping, ceasing function calls too early, and making decisions based on incomplete information. A clear example of this is observed in the Movie Recommendation benchmark, where ReAct often searches for fewer than the required 8 movies before delivering its final answer. In Figure 6.4 (Left), we illustrate the distribution of the number of function calls within ReAct (using GPT) across thhe Movie Recommendation benchmark. Here, we observe around 85% of the examples exhibit early stopping, making decisions without completing all 8 movie searches. This contrasts with `LLMCompiler` (Right), where almost all examples (99%) com-

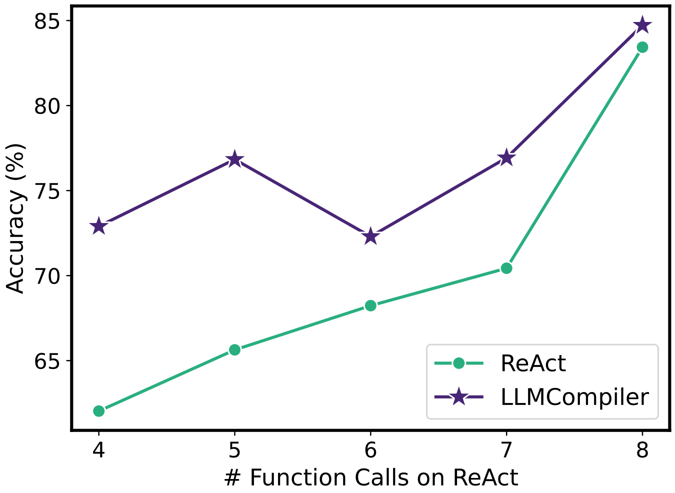


Figure 6.5: The Movie Recommendation accuracy of the examples that are categorized by the number of function calls on ReAct, measured both on ReAct and `LLMCompiler`. The plot indicates that in ReAct, a decrease in the number of function calls correlates with lower accuracy, indicating that premature exits lead to reduced accuracy. In contrast, when the same examples are evaluated using `LLMCompiler`, which ensures complete searches for all eight movies before reaching a decision, they achieve higher and more consistant accuracy than those processed by ReAct. gpt-3.5-turbo is used for the experiment, and the results are averaged over 3 different runs.

plete the full search of 8 movies. Although adding specific prompts to ReAct to prevent early stopping shifts the distribution towards more function calls (Figure 6.4, Middle), resulting in an accuracy improvement from 68.60 to 72.47 (ReAct† in Table 6.1), it is nevertheless an imperfect solution.

To further assess how early stopping negatively impacts accuracy, we categorize Movie Recommendation benchmark examples by their number of function calls in ReAct. We then evaluated these groups using `LLMCompiler`, ensuring complete search results for all 8 movies. Figure 6.5 reveals that fewer function calls in ReAct correlate with lower average accuracy (green line). Conversely, if these examples were processed through `LLMCompiler`, with complete searches for all eight movies, they consistently attained higher accuracy (purple line). This not only indicates that ReAct struggles with premature exits (which is not fully addressed by prompting), but the earlier it stops, the greater the decline in accuracy, contributing to the overall accuracy drop observed in Table 6.1. In contrast, `LLMCompiler` effectively addresses this issue.

Repetitive Function Calls of ReAct. Another common failure case of ReAct is its tendency for repetitive function calls, often leading to infinite loops or exceeding the context

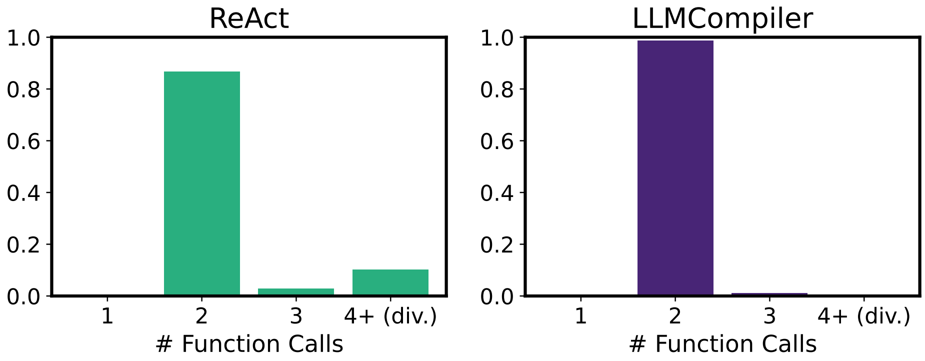


Figure 6.6: Distributions of the number of function calls when running the HotpotQA benchmark on ReAct (Left) and `LLMCompiler` (Right). While `LLMCompiler` (Right) consistently completes the task within 2 function calls, which is expected as HotpotQA exhibits a 2-way parallelizable pattern, ReAct (Left) shows that around 10% of the examples undergo repetitive (>4) function calls, resulting in a diverging behavior of the framework. LLaMA-2 70B is used for the experiment.

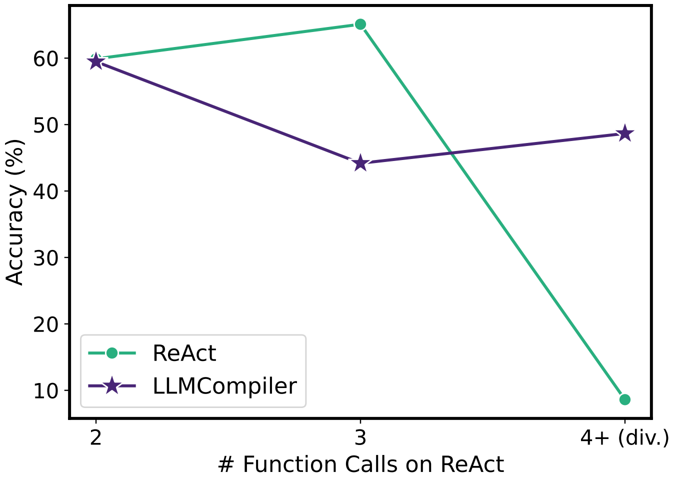


Figure 6.7: The HotpotQA accuracy of the examples that are categorized by the number of function calls on ReAct, measured both on ReAct and `LLMCompiler`. The plot indicates that in ReAct, repetitive function calls of more than or equal to four times can result in a significant accuracy degradation due to its infinite looping and diverging behavior. On the other hand, when the same examples are evaluated using `LLMCompiler`, which ensures only two searches per example, they achieve a higher of around 50%. LLaMA-2 70B is used for the experiment.

length limit. This problem is particularly noticeable in the HotpotQA benchmark where ReAct repeatedly calls the same function if the Wikipedia search returns insufficient information about the searched entity. Although HotpotQA is inherently 2-way parallelizable, as illustrated in Figure 6.6, we observe that about 10% of its examples require more than four function calls in ReAct, usually resulting in an infinite loop or a divergent behavior. In contrast, `LLMCompiler` executes only two function calls for most examples.

To show how the repetitive function calls impact the overall accuracy, we conduct an accuracy analysis similar to the previous case. In Figure 6.7, we categorize HotpotQA benchmark examples by the number of function calls in ReAct, and then we compare their accuracy on both ReAct and `LLMCompiler`. The analysis reveals that examples that launch two function calls in ReAct maintain the same accuracy in `LLMCompiler`. However, cases with more than four function calls in ReAct, which often lead to divergent behavior, show less than 10% accuracy in ReAct. On the other hand, when these examples are processed with `LLMCompiler`, they achieve around 50% accuracy by circumventing repetitive calls. It is worth noting that there are instances with three function calls in ReAct, where an extra search can lead to improved accuracy by retrying with an alternate entity name when the initial search fails, yielding a better accuracy than `LLMCompiler`. While this shows a potential adaptability advantage of ReAct, such instances represent less than 3% of cases.

Failure Case Analysis of `LLMCompiler`

This section delves into a qualitative analysis of `LLMCompiler`'s failure cases on the ParallelQA benchmark, which can be broadly attributed to failures in the Planner, Executor, or the final output process. Failures in the final output process refer to cases when LLMs are unable to use the observations collected from tool execution (which are incorporated into the context) to deliver the correct answer to the user. Among the 10.6% (36 examples) of `LLMCompiler`'s total failures reported in Table 6.1, we have noted that the Planner, Executor, and final output process contributed to 8%, 64%, and 28% of the failures, respectively. The Planner's 8% failure rate is exclusive to `LLMCompiler`. For instance, the Planner would incorrectly map inputs and outputs by assigning a wrong identifier as an input to a subsequent task, thereby forming an incorrect DAG. However, with adequate tool definitions and in-context examples, Planner errors are significantly reduced (only 3 instances in total throughout our evaluation), highlighting the LLM's capability to decompose problems into complex multi-task dependencies.

The remaining 92% of the total failures are attributed to the Executor and the final output process. The Executor accounts for most of these failures (64%), with common issues like the `math` tool choosing wrong attributes or mishandling unit conversions. For the final output process (28% of failures), errors include incorrect conclusions from the gathered observations, such as failing to pick the smallest attribute from the collected data. It's worth noting that these problems are not exclusive to `LLMCompiler`, but they also occur in ReAct. Nevertheless, `LLMCompiler` tends to have slightly fewer failures in these areas than ReAct, as it provides only relevant contexts to each tool, aiding in more accurate information extraction. We believe that optimizing the structure of the agent scratchpad, rather than simply appending observations, could further reduce failures in the final output process.

6.6 Related Work

Latency Optimization in LLMs

Various studies have focused on optimizing model design [137, 60, 166, 42, 146, 59, 135, 20, 156] and systems [147, 313, 109, 110] for efficient LLM inference. Optimizations at the application level, however, are less explored. This is critical from a practical point of view for situations involving black-box LLM models and services where modifications to the models and the underlying inference pipeline are highly restricted.

Skeleton-of-Thought [190] recently proposed to reduce latency through application-level parallel decoding. This method involves a two-step process of an initial skeleton generation phase, followed by parallel execution of skeleton items. However, it is primarily designed for embarrassingly parallel workloads and does not support problems that have inherently interdependent tasks, as it assumes no dependencies between skeleton tasks. This limits its applicability in complex scenarios such as coding [22, 177, 96, 2] or math [98, 97] problems, as also stated in the paper [190]. `LLMCompiler` addresses this by translating an input query into a series of tasks with inter-dependencies, thereby expanding the spectrum of problems it can handle.

Concurrently to our work, OpenAI has recently introduced a parallel function calling feature in their 1106 release, enhancing user query processing through the simultaneous generation of multiple function calls [196]. Despite its potential for reducing LLM execution time, this feature has certain limitations, as it is exclusively available for OpenAI's proprietary models. However, there is a growing demand for using open-source models driven by the increasing number of open-source LLMs as well as parameter-efficient training techniques [154, 111, 105] for finetuning and customization. `LLMCompiler` enables efficient parallel function calling for open-source models, and also, as we will show later in Section 6.4, it can potentially achieve better latency and cost.

Plan and Solve Strategy

Several studies [291, 205, 211, 336, 91] have explored prompting methods of breaking down complex queries into various levels of detail to solve them, thereby improving LLM's performance in reasoning tasks. Specifically, Decomposed Prompting [129] tackles complex tasks by decomposing them into simpler sub-tasks, each optimized through LLMs with dedicated prompts. Step-Back Prompting [333] enables LLMs to abstract high-level concepts from details to enhance reasoning abilities across various tasks. Plan-and-Solve Prompting [274] segments multi-step reasoning tasks into subtasks to minimize errors and improve task accuracy without manual prompting. However, these methods primarily focus on improving the accuracy of reasoning benchmarks. In contrast, `LLMCompiler` uses a planner to identify parallelizable patterns within queries, aiming to reduce latency while maintaining accuracy.

In addition to the aforementioned works, ViperGPT [256] TPTU [224], and HuggingGPT [237] and have introduced end-to-end plan-and-solve frameworks. `LLMCompiler` sets itself apart by providing a general framework that enables efficient and accurate function calling in a broader range of problems. This stems from `LLMCompiler`'s capabilities in (i) planning and replanning; (ii) parallel execution; and (iii) addressing a wider range of problem domains, which will be discussed in more detail in Appendix D.3.

Another notable work is ReWOO [297] which employs a planner to separate the reasoning process from the execution and observation phases to decrease token usage and cost as compared to ReAct. Our approach is different from ReWOO in multiple aspects. First, `LLMCompiler` allows parallel function calling which can reduce latency as well as cost. Second, `LLMCompiler` supports dynamic replanning which is important for problems whose execution flow cannot be determined statically in the beginning (Section 6.4).

Tool-Augmented LLMs

A notable work is Toolformer [230], which produces a custom LLM output to let the LLM decide what the inputs for calling the functions should be and where to insert the result. This approach has inspired various tool calling frameworks [164, 237]. ReAct [307] proposed to have LLMs interact with external environments through reasoning and action generation for improved performance. Gorilla [206] introduced a finetuned LLM designed for function calling, and ToolLLM [212] and RestGPT [248] have extended LLMs to support real-world APIs. Moreover, OpenAI [195] released their own function calling capabilities, allowing their LLMs to return formatted JSON for execution.

6.7 Conclusions

Existing methods for invoking multiple functions with LLMs resort to sequential and dynamic reasoning. As a result, they suffer from inefficiencies in latency, cost, and accuracy. As a solution, we introduced `LLMCompiler`, a compiler-inspired framework that enables efficient parallel function calling across various LLMs, including open-source models like LLaMA-2 and OpenAI's GPT series. By decomposing user inputs into tasks with defined interdependencies and executing these tasks concurrently through its Planner, Task Fetching Unit, and Executor components, `LLMCompiler` demonstrates substantial improvements in latency (up to 3.7×), cost efficiency (up to 6.7×), and accuracy (up to $\sim$9%), even outperforming OpenAI's parallel function calling feature in latency gains. We look forward to future work building upon `LLMCompiler` that will improve both the capabilities and efficiencies of LLMs in executing complex, large-scale tasks, thus transforming the future development of LLM-based applications.

Chapter 7

Conclusion

7.1 Review

While the common approach of scaling model and dataset sizes has driven remarkable advances in AI technologies over the past few years, it has also introduced significant inference-time overheads, including increased latency, memory demands, and power consumption, all of which hinder the efficient deployment of state-of-the-art models. In this thesis, we presented a full-stack view of improving the efficiency of large-scale AI models, addressing optimization at four different levels within the inference stack.

Model Optimization. Model optimization is a key strategy for efficient model deployment by enabling more effective utilization of hardware resources, such as compute and memory. In this thesis, we focused on quantization, a widely adopted method for model optimization that represents model weights and activations using lower-bit precision (e.g. 8-bit or lower). Quantization not only reduces static storage and peak run-time memory requirements by using smaller bit widths to represent model weights but also enables efficient integer computation, which can further improve latency and power consumption of model inference.

In Chapter 2, we introduced I-BERT, a quantization scheme designed to optimize *compute efficiency* during Transformer inference. In particular, I-BERT proposes integer-only quantization that allows the entire inference process to be executed using integer arithmetic. By leveraging integer-only quantization, I-BERT enables the deployment of Transformer models on integer-only logical units or processors, such as Turing Tensor Cores in NVIDIA GPUs and ARM processors [1], resulting in up to a 3.5× inference speedup for encoder models like BERT [43] and RoBERTa [171].

In Chapter 3, we introduced SqueezeLLM, a quantization scheme designed to optimize *memory efficiency* during Transformer inference. While I-BERT demonstrated significant improvements in compute-bound workloads, the autoregressive generation process in decoder-based LLMs tends to be rather memory-bound. In such cases, optimizing computational resources (e.g., integer-processing units) has a limited impact on overall latency, as the primary bottleneck will be the time spent loading weight matrices from memory.

Based on this observation, SqueezeLLM proposes sensitivity-based nonuniform quantization and Dense-and-Sparse decomposition, allowing for more aggressive bit precision reduction with minimal accuracy degradation. By minimizing memory loading times, SqueezeLLM achieves a 2× speedup with near-lossless prediction performance, using 3-bit quantization on memory-bound LLMs such as LLaMA [262].

Inference Methods. For efficient inference, it is often necessary to revisit inference methods, i.e., how models are executed, to reduce redundant operations and maximize resource utilization. This is becoming increasingly important with the growing popularity of LLMs that rely on autoregressive generation, which is highly memory-inefficient due to its repetitive and unparallelizable memory operations. As a result, there is an increasing demand for optimized inference methods that better amortize memory load costs across more compute operations, thereby enhancing overall inference latency.

In Chapter 4, we introduced a speculative decoding framework named Big Little Decoder (BiLD) that addresses the inefficiencies of memory operations during autoregressive inference. BiLD proposes a collaborative generation strategy using two models of different sizes: the smaller model quickly generates multiple tokens, while the larger model periodically reviews and refines those predictions. This approach enables the larger model to perform non-autoregressive execution, processing multiple tokens in a single iteration and yielding a 2× speedup in inference without compromising the generation quality of the large model.

Model Architectures. Model architecture design is another indispensable component in achieving efficient inference. Inductive bias [267], the assumptions made by a learning algorithm to generalize from finite training data to a general model of the domain, have played a key role in guiding model design. While Transformers have shown significant performance improvements when trained on large datasets even with minimal inductive bias, incorporating domain-specific inductive biases can be essential when designing more compact models or working in data-scarce environments.

In Chapter 5, we introduced Squeezeformer, a more efficient model architecture family for automatic speech recognition. Unlike the discrete inputs in the written language domain, from which the Transformer architecture originated, the speech domain consists of continuous input signals. To leverage this domain-specific characteristic, Squeezeformer incorporates a Temporal U-Net structure, which downsamples along the temporal axis to eliminate redundancy in speech inputs. This approach significantly reduces sequence length, thereby cutting computational costs and improving inference efficiency.

AI Applications. Finally, in Chapter 6, we expanded the scope of efficiency from optimizing individual models to also considering the external components (e.g., functions) they interact with in agentic applications. With recent advances in the reasoning capabilities of LLMs, these models have been integrated with external functions to create agentic applications. In such applications, LLMs actively engage with their environments by executing actions and retrieving information using external functions to complete complex user tasks autonomously.

As a result, enhancing the efficiency of dynamic interactions between LLMs and these external functions has become crucial for developing more efficient, scalable, and responsive agentic applications. This motivated the introduction of the `LLMCompiler` framework that efficiently orchestrates multiple function calls by decomposing user inputs into executable tasks and their interdependencies. `LLMCompiler` has demonstrated not only a significant reduction in execution latency and costs by running independent tasks in parallel, but also improved robustness by breaking down complex user inputs into smaller, manageable tasks.

7.2 Impact of Our Work

This thesis is a compilation of my research publications – I-BERT [130], SqueezeLLM [137], BiLD [135], Squeezeformer [136], and `LLMCompiler` [132] – which were presented at major AI conferences, ICML and NeurIPS, from 2021 to 2024. Beyond their academic recognition, these works have significantly impacted both research communities and the industry, as well as open-source communities. In this section, we summarize some of the impacts of our work.

I-BERT [134], introduced in January 2021 on arXiv, was the first integer-only quantization scheme for the Transformer architecture, enabling Transformer inference to be carried out entirely using integer arithmetic. This contribution led to I-BERT being selected for an Oral presentation at ICML 2021. Additionally, I-BERT was immediately integrated into the HuggingFace Transformers library as one of the earliest officially supported models, contributing to the broader adoption of efficient Transformer inference in open-source and industrial applications.

Two years after the introduction of I-BERT, we have witnessed a significant shift in AI models and applications, moving from million-scale Transformer models to billion-scale LLMs, and from encoder-based models (e.g., BERT) to generative decoder models (e.g., GPT and LLaMA). This shift has also changed the workload characteristics from being primarily compute-bound to more memory-bound. In response to this change, SqueezeLLM [137] was introduced in June 2023 on arXiv as a new quantization scheme specifically designed for generative LLMs to better optimize their memory-bound characteristic. Not only was this work accepted to ICML 2024, but it also became one of the pioneering low-bit quantization methods for LLMs. As quantization has since become indispensable for efficient LLM inference, SqueezeLLM has laid the foundation for numerous subsequent studies on LLM quantization that followed.

Squeezeformer [136], introduced in June 2022 on arXiv and accepted in NeurIPS 2022, is an efficient automatic speech recognition model that achieved up to a 3% improvement in word-error-rate compared to state-of-the-art models including Conformer [82] with the same computational budget. It was also integrated into NVIDIA's official NeMo codebase, one of the most widely adopted open-source frameworks for speech applications. This highlights the growing demand for efficient speech recognition solutions as well as its contribution to the speech recognition field within both academic and industry contexts.

LLMCompiler [132] was introduced in December 2023 on arXiv as the pioneering attempt to improve the efficiency of LLM-based agentic applications. Accepted to ICML 2024, LLMCompiler demonstrated significant performance gains, achieving up to 4× speedup, 7× cost reduction, and 9% accuracy improvement across various function-calling benchmarks compared to the widely used ReAct-based solutions [307]. LLMCompiler has also gained substantial traction in the open-source community, accruing 1.5k stars on GitHub in a few months after its publication. Additionally, the LLMCompiler framework has been integrated into the most widely used open-source LLM frameworks such as LlamaIndex and LangChain, further showcasing its broad applicability and high demand for building scalable, robust agentic applications in diverse LLM-based communities.

Finally, this thesis has also been inspired by several key survey papers we published. In 2021, [71] was published as a book chapter in Low-Power Computer Vision, providing a comprehensive overview of recent quantization techniques and their impact. This work became pivotal in introducing quantization and its benefits to a broader audience. In February 2023, [133] was introduced as a survey paper that emphasized the importance of the full-stack view of making Transformer inference more efficient to address the growing compute and memory demands of large-scale models – a central theme of this thesis. Additionally, "AI and Memory Wall" [73], initially published as a blog post in early 2021, raised awareness among the broader public about the increasing trend of AI workloads becoming memory-bound (where we introduced the earlier version of Figure 1.1). As memory bottleneck became a critical issue for large-scale models, the paper was revisited and invited to the IEEE MICRO Journal Special Issue in 2024, inspiring various efforts to address memory challenges in LLM inference, including our own works [135, 137].

7.3 Future Directions

In this section, we discuss potential future extensions of the full-stack view of designing solutions for efficient and scalable AI algorithms and systems (Figure 7.1).

Memory-Optimized AI Model and System Design. With the increasing popularity of autoregressive and large-scale models [220, 168], model inference is becoming increasingly memory-bound. As illustrated in Figure 1.1, the rapid expansion of state-of-the-art Transformer models (growing 410× every two years) contrasts with the much slower improvement in memory bandwidth of underlying hardware (2× every two years). The memory-bound inference behavior is further exacerbated by the wide adoption of long-context models and their applications. For instance, with context length over 32k, the KV cache can surface as a dominant contributor to memory traffic, potentially consuming more memory bandwidth than model weights [103].

To bridge the gap between model requirements and hardware constraints, further investigation into memory optimization techniques for both models and systems is necessary. As demonstrated in our previous works [137, 103, 135, 104], reevaluating the memory-compute trade-offs by prioritizing memory efficiency, even at the cost of additional computation, will

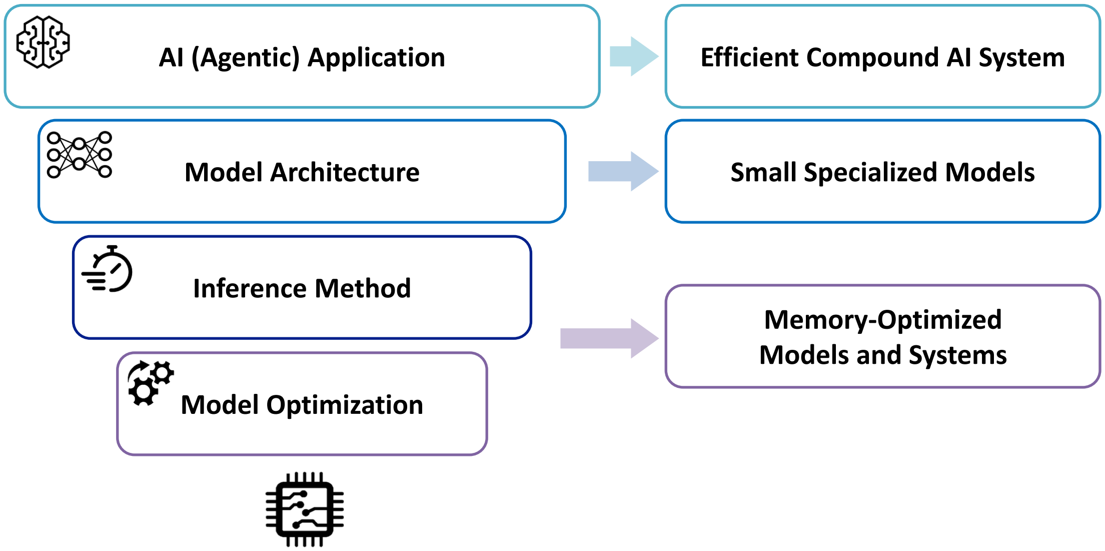


Figure 7.1: Potential future extensions of the full-stack view of designing solutions for efficient and scalable AI algorithms and systems.

be crucial for alleviating the memory pressure of future workloads. This may call for more aggressive compression techniques, such as variable-length encoding or binary/ternary quantization, alongside system architectures designed to support them. Additionally, developing more memory-efficient methods for managing long contexts will be an important research direction. These approaches could include advanced quantization techniques [103, 172], KV cache eviction and partial retrieval mechanisms [331, 162], or even paradigm shifts in model architectures to manage long contexts with less memory overheads [79, 36].

Small Models for Specialized Domains and Tasks. LLMs these days demonstrate high problem-solving capabilities in general tasks. This breakthrough, however, often relies on a dramatic scale-up of the model size and training data, which can be impractical for designing smaller models. To address this, the following two directions can be considered for developing small specialist models.

(i) *Domain and task-specific model architecture*: When designing models for particular domains, considering the unique characteristics of each domain can enable more efficient architectures than simply applying the existing Transformer architecture [136]. Even within the NLP domain, current language models may not necessarily be optimal for emerging tasks such as retrieval-augmented generation (RAG), multi-turn conversation, and function calling. Different tasks have unique patterns for retrieving and storing important contexts; however, the current Transformer architecture only offers a single interface through the KV cache, which serves as a general, one-size-fits-all solution but may not be optimal for individual task. This motivates that the Transformer architecture needs to be revisited for individual domains and tasks, thereby enhancing their efficiency in various applications.

(ii) *Training dataset synthesis*: Developing small models often faces challenges due to insufficient parametric memory and limited reasoning and generalization capabilities for unseen tasks. This frequently requires additional training on target tasks to ensure the desired performance. Using larger language models to synthesize and augment training datasets can offer a viable solution, enabling rather smaller models to acquire the capabilities to solve various sophisticated tasks [279]. This approach – focusing on a specific problem domain and synthesizing corresponding datasets – along with advancements in model architectures, will be crucial in paving the way for sub-billion-scale specialist models.

Compound AI Systems: Efficiency Beyond a Single Model. Advances in the reasoning capabilities of LLMs, driven by increased model capacity and advanced reasoning methodologies, are expanding their applications beyond content generation to serve as building blocks for *compound AI systems* [319]. For instance, LLMs can collaborate with other models or external functions (e.g., search engines and calculators) to solve more complex tasks or use retrieval methods for more grounded answers (i.e., RAG). These compound AI systems offer a new perspective on systems, extending beyond serving or training a single model to orchestrating multiple models and additional components that the models interact with. For these systems to be practically useful, it is important to identify and optimize inference bottlenecks in various components – such as retrievers, function calling, and the routing/orchestration of multiple models – to improve efficiency in terms of latency, cost, and context length. Our work on `LLMCompiler` [132] was the first attempt to focus on enhancing efficiency, particularly in function calling scenarios, but much work remains to fully realize the potential of these systems.

Bibliography

[1] ARM. *Cortex-M, https://developer.arm.com/ip-products/processors/cortex-m*. 2020.

[2] Jacob Austin et al. *Program Synthesis with Large Language Models*. 2021. arXiv: `2108.07732 [cs.PL]`.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[4] Alexei Baevski et al. "Data2vec: A general framework for self-supervised learning in speech, vision and language". In: *arXiv preprint arXiv:2202.03555* (2022).

[5] Alexei Baevski et al. "wav2vec 2.0: A framework for self-supervised learning of speech representations". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 12449–12460.

[6] Haoli Bai et al. "BinaryBERT: Pushing the Limit of BERT Quantization". In: *arXiv preprint arXiv:2012.15701* (2020).

[7] Maciej Besta et al. *Graph of Thoughts: Solving Elaborate Problems with Large Language Models*. 2023. arXiv: `2308.09687 [cs.CL]`.

[8] Aishwarya Bhandare et al. "Efficient 8-bit quantization of transformer neural machine language translation model". In: *arXiv preprint arXiv:1906.00532* (2019).

[9] Ondrej Bojar et al. "Findings of the 2014 Workshop on Statistical Machine Translation". In: *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Baltimore, Maryland, USA: Association for Computational Linguistics, June 2014, pp. 12–58. URL: `http://www.aclweb.org/anthology/W/W14/W14-3302`.

[10] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. "Understanding and overcoming the challenges of efficient Transformer quantization". In: *arXiv preprint arXiv:2109.12948* (2021).

[11] Andy Brock et al. "High-performance large-scale image recognition without normalization". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 1059–1071.

[12] Tom B Brown et al. "Language models are few-shot learners". In: *arXiv preprint arXiv:2005.14165* (2020).

[13] Maxime Burchi and Valentin Vielzeuf. "Efficient conformer: Progressive downsampling and grouped attention for automatic speech recognition". In: *2021 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE. 2021, pp. 8–15.

[14] Yaohui Cai et al. "ZeroQ: A novel zero shot quantization framework". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 13169–13178.

[15] Daniel Cer et al. "Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation". In: *arXiv preprint arXiv:1708.00055* (2017).

[16] Mauro Cettolo et al. "Overview of the IWSLT 2017 Evaluation Campaign". In: *Proceedings of the 14th International Conference on Spoken Language Translation*. Tokyo, Japan: International Workshop on Spoken Language Translation, Dec. 2017, pp. 2–14. URL: `https://aclanthology.org/2017.iwslt-1.1`.

[17] Xuankai Chang et al. "End-to-end ASR with Adaptive Span Self-Attention." In: *INTERSPEECH*. 2020, pp. 3595–3599.

[18] Jerry Chee et al. "Quip: 2-bit quantization of large language models with guarantees". In: *Advances in Neural Information Processing Systems* 36 (2024).

[19] Beidi Chen et al. "Scatterbrain: Unifying sparse and low-rank attention". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 17413–17426.

[20] Charlie Chen et al. "Accelerating Large Language Model Decoding with Speculative Sampling". In: *arXiv preprint arXiv:2302.01318* (2023). arXiv: `2302.01318 [cs.CL]`.

[21] Daoyuan Chen et al. "AdaBERT: Task-adaptive bert compression with differentiable neural architecture search". In: *arXiv preprint arXiv:2001.04246* (2020).

[22] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: (2021). arXiv: `2107.03374 [cs.LG]`.

[23] Patrick H Chen and Cho-jui Hsieh. "A comparison of second-order methods for deep convolutional neural networks". In: *openreview under ICLR 2018* (2018).

[24] Sanyuan Chen et al. "WavLM: Large-scale self-supervised pre-training for full stack speech processing". In: *arXiv preprint arXiv:2110.13900* (2021).

[25] Tianqi Chen et al. "TVM: An automated end-to-end optimizing compiler for deep learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594.

[26] Wenhu Chen et al. *Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks*. 2023. arXiv: `2211.12588 [cs.CL]`.

[27] Wei-Lin Chiang et al. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. Mar. 2023. URL: `https://lmsys.org/blog/2023-03-30-vicuna/`.

[28] Jungwook Choi et al. “PACT: Parameterized clipping activation for quantized neural networks”. In: *arXiv preprint arXiv:1805.06085* (2018).

[29] Aakanksha Chowdhery et al. “Palm: Scaling language modeling with pathways”. In: *arXiv preprint arXiv:2204.02311* (2022).

[30] Aakanksha Chowdhery et al. “Palm: Scaling language modeling with pathways”. In: *Journal of Machine Learning Research* 24.240 (2023), pp. 1–113.

[31] Insoo Chung et al. “Extremely low bit transformer quantization for on-device neural machine translation”. In: *arXiv preprint arXiv:2009.07453* (2020).

[32] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “BinaryConnect: Training deep neural networks with binary weights during propagations”. In: *Advances in neural information processing systems*. 2015, pp. 3123–3131.

[33] Matthieu Courbariaux et al. “Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1”. In: *arXiv preprint arXiv:1602.02830* (2016).

[34] Richard Crandall and Carl B Pomerance. *Prime numbers: a computational perspective*. Vol. 182. Springer Science & Business Media, 2006.

[35] Ido Dagan, Oren Glickman, and Bernardo Magnini. “The PASCAL recognising textual entailment challenge”. In: *Machine Learning Challenges Workshop*. Springer. 2005, pp. 177–190.

[36] Tri Dao and Albert Gu. “Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality”. In: *arXiv preprint arXiv:2405.21060* (2024).

[37] Jyotikrishna Dass et al. “Vitality: Unifying low-rank and sparse approximation for vision transformer acceleration with a linear taylor attention”. In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 415–428.

[38] Mostafa Dehghani et al. “Universal transformers”. In: *arXiv preprint arXiv:1807.03819* (2018).

[39] Jérémie Detrey and Florent de Dinechin. “A parameterized floating-point exponential function for FPGAs”. In: *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.* IEEE. 2005, pp. 27–34.

[40] Tim Dettmers et al. “GPT3. int8 (): 8-bit Matrix Multiplication for Transformers at Scale”. In: *Advances in Neural Information Processing Systems*. 2022.

[41] Tim Dettmers et al. “Qlora: Efficient finetuning of quantized llms”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[42] Tim Dettmers et al. “SpQR: A Sparse-Quantized Representation for Near-Lossless LLM Weight Compression”. In: *arXiv preprint arXiv:2306.03078* (2023). arXiv: `2306.03078 [cs.CL]`.

[43] Jacob Devlin et al. "BERT: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[44] Jesse Dodge et al. "Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping". In: *arXiv preprint arXiv:2002.06305* (2020).

[45] William B Dolan and Chris Brockett. "Automatically constructing a corpus of sentential paraphrases". In: *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*. 2005.

[46] Piotr Dollár, Mannat Singh, and Ross Girshick. "Fast and accurate model scaling". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 924–932.

[47] Zhen Dong et al. "HAWQ-V2: Hessian Aware trace-Weighted Quantization of Neural Networks". In: *NeurIPS'19 workshop on Beyond First-Order Optimization Methods in Machine Learning.* (2019).

[48] Zhen Dong et al. "HAWQ: Hessian aware quantization of neural networks with mixed-precision". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 293–302.

[49] Alexey Dosovitskiy et al. "An image is worth 16x16 words: Transformers for image recognition at scale". In: *arXiv preprint arXiv:2010.11929* (2020).

[50] Nan Du et al. "GLAM: Efficient scaling of language models with mixture-of-experts". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 5547–5569.

[51] Vage Egiazarian et al. "Extreme Compression of Large Language Models via Additive Quantization". In: *arXiv preprint arXiv:2401.06118* (2024).

[52] Maha Elbayad et al. "Depth-adaptive transformer". In: *arXiv preprint arXiv:1910.10073* (2019).

[53] Steven K Esser et al. "Learned step size quantization". In: *arXiv preprint arXiv:1902.08153* (2019).

[54] Georgii Evtushenko. "Sparse Matrix-Vector Multiplication with CUDA". In: *https://medium.com/analytics-vidhya/sparse-matrix-vector-multiplication-with-cuda-42d191878e8f* (2019).

[55] Angela Fan, Edouard Grave, and Armand Joulin. "Reducing transformer depth on demand with structured dropout". In: *arXiv preprint arXiv:1909.11556* (2019).

[56] Angela Fan et al. "Training with quantization noise for extreme fixed-point compression". In: *arXiv preprint arXiv:2004.07320* (2020).

[57] Haoqi Fan et al. "Multiscale Vision Transformers". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 6824–6835.

[58] Goran Flegar and Enrique S Quintana-Ortí. “Balanced CSR sparse matrix-vector product on graphics processors”. In: *Euro-Par 2017: Parallel Processing: 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28–September 1, 2017, Proceedings 23*. Springer. 2017, pp. 697–709.

[59] Elias Frantar and Dan Alistarh. *SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot*. 2023. arXiv: `2301.00774 [cs.LG]`.

[60] Elias Frantar et al. “GPTQ: Accurate Post-training Compression for Generative Pretrained Transformers”. In: *arXiv preprint arXiv:2210.17323* (2022).

[61] Elias Frantar et al. “GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers”. In: *arXiv preprint arXiv:2210.17323* (2022).

[62] Yarin Gal and Zoubin Ghahramani. “Dropout as a bayesian approximation: Representing model uncertainty in deep learning”. In: *international conference on machine learning*. PMLR. 2016, pp. 1050–1059.

[63] Trevor Gale, Erich Elsen, and Sara Hooker. “The state of sparsity in deep neural networks”. In: *arXiv preprint arXiv:1902.09574* (2019).

[64] Leo Gao et al. *A framework for few-shot language model evaluation*. 2021.

[65] Luyu Gao et al. “PAL: Program-aided Language Models”. In: *arXiv preprint arXiv:2211.10435* (2022).

[66] John S Garofolo. “Timit acoustic phonetic continuous speech corpus”. In: *Linguistic Data Consortium, 1993* (1993).

[67] Xue Geng et al. “Hardware-aware exponential approximation for deep neural network”. In: (2018).

[68] Xue Geng et al. “Hardware-aware softmax approximation for deep neural networks”. In: *Asian Conference on Computer Vision*. Springer. 2018, pp. 107–122.

[69] Marjan Ghazvininejad et al. “Mask-predict: Parallel decoding of conditional masked language models”. In: *arXiv preprint arXiv:1904.09324* (2019).

[70] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference”. In: *arXiv preprint arXiv:2103.13630* (2021).

[71] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference”. In: *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326.

[72] Amir Gholami et al. “AI and Memory Wall”. In: *IEEE Micro* (2024), pp. 1–5.

[73] Amir Gholami et al. “AI and memory wall”. In: *IEEE Micro* (2024).

[74] Amir Gholami et al. “SqueezeNext: Hardware-Aware Neural Network Design”. In: *Workshop paper in CVPR* (2018).

[75] Mitchell A Gordon, Kevin Duh, and Nicholas Andrews. “Compressing bert: Studying the effects of weight pruning on transfer learning”. In: *arXiv preprint arXiv:2002.08307* (2020).

[76] GPTQ-For-LLaMA. *https://github.com/qwopqwop200/GPTQ-for-LLaMa.*

[77] Alex Graves. “Sequence transduction with recurrent neural networks”. In: *arXiv preprint arXiv:1211.3711* (2012).

[78] Alex Graves et al. “Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks”. In: *Proceedings of the 23rd international conference on Machine learning.* 2006, pp. 369–376.

[79] Albert Gu and Tri Dao. “Mamba: Linear-time sequence modeling with selective state spaces”. In: *arXiv preprint arXiv:2312.00752* (2023).

[80] Jiatao Gu, Changhan Wang, and Junbo Zhao. “Levenshtein transformer”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[81] Jiatao Gu et al. “Non-autoregressive neural machine translation”. In: *arXiv preprint arXiv:1711.02281* (2017).

[82] Anmol Gulati et al. “Conformer: Convolution-augmented transformer for speech recognition”. In: *arXiv preprint arXiv:2005.08100* (2020).

[83] Chuan Guo et al. “On calibration of modern neural networks”. In: *International conference on machine learning.* PMLR. 2017, pp. 1321–1330.

[84] Junliang Guo, Linli Xu, and Enhong Chen. “Jointly masked sequence-to-sequence model for non-autoregressive neural machine translation”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics.* 2020, pp. 376–385.

[85] Pengcheng Guo et al. “Recent developments on ESPNet toolkit boosted by Conformer”. In: *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE. 2021, pp. 5874–5878.

[86] Kyu J Han, Ramon Prieto, and Tao Ma. “State-of-the-art speech recognition using multi-stream self-attention with dilated 1d convolutions”. In: *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU).* IEEE. 2019, pp. 54–61.

[87] Song Han and B Dally. “Efficient methods and hardware for deep learning”. In: *University Lecture* (2017).

[88] Song Han, Huizi Mao, and William J Dally. “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding”. In: *International Conference on Learning Representations* (2016).

[89] Song Han et al. “Learning both weights and connections for efficient neural network”. In: *Advances in neural information processing systems.* 2015, pp. 1135–1143.

[90] Wei Han et al. “ContextNet: Improving convolutional neural networks for automatic speech recognition with global context”. In: *arXiv preprint arXiv:2005.03191* (2020).

[91] Shibo Hao et al. *Reasoning with Language Model is Planning with World Model.* 2023. arXiv: `2305.14992 [cs.CL]`.

[92] James W Hauser and Carla N Purdy. "Approximating functions for embedded and ASIC applications". In: *Proceedings of the 44th IEEE 2001 Midwest Symposium on Circuits and Systems. MWSCAS 2001 (Cat. No. 01CH37257).* Vol. 1. IEEE. 2001, pp. 478–481.

[93] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[94] Dan Hendrycks and Kevin Gimpel. "A baseline for detecting misclassified and out-of-distribution examples in neural networks". In: *arXiv preprint arXiv:1610.02136* (2016).

[95] Dan Hendrycks and Kevin Gimpel. "Gaussian error linear units (GELUs)". In: *arXiv preprint arXiv:1606.08415* (2016).

[96] Dan Hendrycks et al. "Measuring Coding Challenge Competence With APPS". In: *NeurIPS* (2021).

[97] Dan Hendrycks et al. "Measuring Massive Multitask Language Understanding". In: *Proceedings of the International Conference on Learning Representations (ICLR)* (2021).

[98] Dan Hendrycks et al. "Measuring Mathematical Problem Solving With the MATH Dataset". In: *NeurIPS* (2021).

[99] Karl Moritz Hermann et al. "Teaching machines to read and comprehend". In: *Advances in neural information processing systems.* 2015, pp. 1693–1701.

[100] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network". In: *Workshop paper in NIPS* (2014).

[101] Jordan Hoffmann et al. "Training Compute-Optimal Large Language Models". In: *arXiv preprint arXiv:2203.15556* (2022).

[102] Chris Hokamp et al. "Dyne: Dynamic ensemble decoding for multi-document summarization". In: *arXiv preprint arXiv:2006.08748* (2020).

[103] Coleman Hooper et al. "Kvquant: Towards 10 million context length llm inference with kv cache quantization". In: *arXiv preprint arXiv:2401.18079* (2024).

[104] Coleman Hooper et al. "Speed: Speculative pipelined execution for efficient decoding". In: *arXiv preprint arXiv:2310.12072* (2023).

[105] Neil Houlsby et al. "Parameter-efficient transfer learning for NLP". In: *International conference on machine learning.* PMLR. 2019, pp. 2790–2799.

[106] Andrew Howard et al. "Searching for MobilenetV3". In: *Proceedings of the IEEE International Conference on Computer Vision.* 2019, pp. 1314–1324.

[107] Andrew G Howard et al. "MobileNets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[108] Wei-Ning Hsu et al. "Hubert: Self-supervised speech representation learning by masked prediction of hidden units". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 29 (2021), pp. 3451–3460.

[109] *https://github.com/NVIDIA/TensorRT-LLM.*

[110] *https://huggingface.co/text-generation-inference.*

[111] Edward J Hu et al. "LoRA: Low-Rank Adaptation of Large Language Models". In: *International Conference on Learning Representations*. 2022.

[112] Jie Hu, Li Shen, and Gang Sun. "Squeeze-and-Excitation networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 7132–7141.

[113] Yafeng Huang et al. "Output Sensitivity-Aware DETR Quantization". In: (2023).

[114] Ngo Quang Huy, Tu Minh Phuong, and Ngo Xuan Bach. "Autoencoding Language Model Based Ensemble Learning for Commonsense Validation and Explanation". In: *arXiv preprint arXiv:2204.03324* (2022).

[115] Forrest N Iandola et al. "SqueezeBERT: What can computer vision teach NLP about efficient neural networks?" In: *arXiv preprint arXiv:2006.11316* (2020).

[116] Forrest N Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size". In: *arXiv preprint arXiv:1602.07360* (2016).

[117] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[118] Shankar Iyer, Nikhil Dandekar, and Kornl Csernai. "First Quora Dataset Release: Question Pairs.(2017)". In: *URL https://data. quora. com/First-Quora-Dataset-Release-Question-Pairs* (2017).

[119] Benoit Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2704–2713.

[120] Yongkweon Jeon et al. "Mr. BiQ: Post-Training Non-Uniform Quantization based on Minimizing the Reconstruction Error". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 12329–12338.

[121] Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. "LLM-Blender: Ensembling Large Language Models with Pairwise Ranking and Generative Fusion". In: *arXiv preprint arXiv:2306.02561* (2023).

[122] Xiaoqi Jiao et al. "Tinybert: Distilling bert for natural language understanding". In: *arXiv preprint arXiv:1909.10351* (2019).

[123] Jing Jin et al. "KDLSQ-BERT: A Quantized Bert Combining Knowledge Distillation with Learned Step Size Quantization". In: *arXiv preprint arXiv:2101.05938* (2021).

[124] Michiel de Jong et al. "FiDO: Fusion-in-Decoder optimized for stronger performance and faster inference". In: *arXiv preprint arXiv:2212.08153* (2022).

[125] Shigeki Karita et al. "A comparative study on Transformer vs RNN in speech applications". In: *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE. 2019, pp. 449–456.

[126] Andrej Karpathy. *Intro to Large Language Models*. 2023.

[127] Jungo Kasai et al. "Deep encoder, shallow decoder: Reevaluating non-autoregressive machine translation". In: *arXiv preprint arXiv:2006.10369* (2020).

[128] Alex Kendall and Yarin Gal. "What uncertainties do we need in bayesian deep learning for computer vision?" In: *Advances in neural information processing systems* 30 (2017).

[129] Tushar Khot et al. "Decomposed Prompting: A Modular Approach for Solving Complex Tasks". In: *The Eleventh International Conference on Learning Representations*. 2023.

[130] Sehoon Kim. *https://github.com/kssteven418/I-BERT*. 2021.

[131] Sehoon Kim. *https://github.com/kssteven418/Squeezeformer*.

[132] Sehoon Kim et al. "An LLM compiler for parallel function calling". In: *arXiv preprint arXiv:2312.04511* (2023).

[133] Sehoon Kim et al. "Full Stack Optimization of Transformer Inference: a Survey". In: *arXiv preprint arXiv:2302.14017* (2023).

[134] Sehoon Kim et al. "I-BERT: Integer-only bert quantization". In: *arXiv preprint arXiv:2101.01321* (2021), pp. 5506–5518.

[135] Sehoon Kim et al. *Speculative decoding with big little decoder*. 2024.

[136] Sehoon Kim et al. "Squeezeformer: An Efficient Transformer for Automatic Speech Recognition". In: *arXiv preprint arXiv:2206.00888* (2022).

[137] Sehoon Kim et al. *SqueezeLLM: Dense-and-Sparse Quantization*. 2023. arXiv: `2306.07629 [cs.CL]`.

[138] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. "Reformer: The Efficient Transformer". In: *International Conference on Learning Representations*. 2019.

[139] Takeshi Kojima et al. *Large Language Models are Zero-Shot Reasoners*. 2023. arXiv: `2205.11916 [cs.CL]`.

[140] Olga Kovaleva et al. "BERT busters: Outlier dimensions that disrupt transformers". In: *arXiv preprint arXiv:2105.06990* (2021).

[141] Samuel Kriman et al. "QuartzNet: Deep automatic speech recognition with 1d time-channel separable convolutions". In: *ICASSP*. IEEE. 2020, pp. 6124–6128.

[142] Raghuraman Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". In: *arXiv preprint arXiv:1806.08342* (2018).

[143] Taku Kudo and John Richardson. "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing". In: *arXiv preprint arXiv:1808.06226* (2018).

[144] Eldar Kurtic et al. "The Optimal BERT Surgeon: Scalable and Accurate Second-Order Pruning for Large Language Models". In: *arXiv preprint arXiv:2203.07259* (2022).

[145] Kiseok Kwon et al. "Co-design of deep neural nets and neural net accelerators for embedded vision applications". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE. 2018, pp. 1–6.

[146] Woosuk Kwon et al. "A Fast Post-Training Pruning Framework for Transformers". In: *arXiv preprint arXiv:2204.09656* (2022). arXiv: `2204.09656 [cs.CL]`.

[147] Woosuk Kwon et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023.

[148] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "CMSIS-NN: Efficient neural network kernels for arm cortex-m cpus". In: *arXiv preprint arXiv:1801.06601* (2018).

[149] Zhenzhong Lan et al. "Albert: A lite bert for self-supervised learning of language representations". In: *arXiv preprint arXiv:1909.11942* (2019).

[150] Langchain. *https://github.com/langchain-ai/langchain*.

[151] Yann LeCun, John S Denker, and Sara A Solla. "Optimal brain damage". In: *Advances in neural information processing systems*. 1990, pp. 598–605.

[152] Jason Lee, Elman Mansimov, and Kyunghyun Cho. "Deterministic non-autoregressive neural sequence modeling by iterative refinement". In: *arXiv preprint arXiv:1802.06901* (2018).

[153] Dmitry Lepikhin et al. "GShard: Scaling giant models with conditional computation and automatic sharding". In: *arXiv preprint arXiv:2006.16668* (2020).

[154] Brian Lester, Rami Al-Rfou, and Noah Constant. *The Power of Scale for Parameter-Efficient Prompt Tuning*. 2021. arXiv: `2104.08691 [cs.CL]`.

[155] Hector Levesque, Ernest Davis, and Leora Morgenstern. "The winograd schema challenge". In: *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*. Citeseer. 2012.

[156] Yaniv Leviathan, Matan Kalman, and Yossi Matias. "Fast inference from transformers via speculative decoding". In: *International Conference on Machine Learning*. PMLR. 2023, pp. 19274–19286. arXiv: `2211.17192 [cs.LG]`.

[157] Fengfu Li, Bo Zhang, and Bin Liu. “Ternary weight networks”. In: *arXiv preprint arXiv:1605.04711* (2016).

[158] Hao Li et al. “Pruning filters for efficient convnets”. In: *arXiv preprint arXiv:1608.08710* (2016).

[159] Jason Li et al. “Jasper: An end-to-end convolutional neural acoustic model”. In: *arXiv preprint arXiv:1904.03288* (2019).

[160] Xiuyu Li et al. “Q-Diffusion: Quantizing Diffusion Models”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2023, pp. 17535–17545.

[161] Yanghao Li et al. “Improved Multiscale Vision Transformers for classification and detection”. In: *arXiv preprint arXiv:2112.01526* (2021).

[162] Yuhong Li et al. “Snapkv: Llm knows what you are looking for before generation”. In: *arXiv preprint arXiv:2404.14469* (2024).

[163] Zhuohan Li et al. “Hint-based training for non-autoregressive machine translation”. In: *arXiv preprint arXiv:1909.06708* (2019).

[164] Yaobo Liang et al. *TaskMatrix.AI: Completing Tasks by Connecting Foundation Models with Millions of APIs*. 2023. arXiv: `2303.16434 [cs.AI]`.

[165] Tatiana Likhomanenko et al. “Rethinking evaluation in ASR: Are our models robust enough?” In: *arXiv preprint arXiv:2010.11745* (2020).

[166] Ji Lin et al. “AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration”. In: (2023). arXiv: `2306.00978 [cs.CL]`.

[167] Chunxi Liu et al. “Improving RNN Transducer based ASR with auxiliary tasks”. In: *2021 IEEE Spoken Language Technology Workshop (SLT)*. IEEE. 2021, pp. 172–179.

[168] Hao Liu et al. “World model on million-length video and language with ringattention”. In: *arXiv preprint arXiv:2402.08268* (2024).

[169] Jerry Liu. *LlamaIndex*. Nov. 2022. DOI: `10.5281/zenodo.1234`. URL: `https://github.com/jerryjliu/llama_index`.

[170] Yijiang Liu et al. “NoisyQuant: Noisy Bias-Enhanced Post-Training Activation Quantization for Vision Transformers”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2023, pp. 20321–20330.

[171] Yinhan Liu et al. “RoBERTa: A robustly optimized bert pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).

[172] Zirui Liu et al. “Kivi: A tuning-free asymmetric 2bit quantization for kv cache”. In: *arXiv preprint arXiv:2402.02750* (2024).

[173] Ilya Loshchilov and Frank Hutter. “Decoupled Weight Decay Regularization”. In: *International Conference on Learning Representations*. 2019.

[174] Yiping Lu et al. "Understanding and improving Transformer from a multi-particle dynamic system point of view". In: *arXiv preprint arXiv:1906.02762* (2019).

[175] Christoph Lüscher et al. "RWTH ASR Systems for LibriSpeech: Hybrid vs Attention–w/o Data Augmentation". In: *arXiv preprint arXiv:1905.03072* (2019).

[176] Kaixin Ma et al. *LASER: LLM Agent with State-Space Exploration for Web Navigation.* 2023. arXiv: `2309.08172 [cs.CL]`.

[177] Aman Madaan et al. *Self-Refine: Iterative Refinement with Self-Feedback.* 2023. arXiv: `2303.17651 [cs.CL]`.

[178] Somshubra Majumdar et al. "Citrinet: Closing the gap between non-autoregressive and autoregressive end-to-end models for automatic speech recognition". In: *arXiv preprint arXiv:2104.01721* (2021).

[179] Huizi Mao et al. "Exploring the regularity of sparse structure in convolutional neural networks". In: *Workshop paper in CVPR* (2017).

[180] Yihuan Mao et al. "Ladabert: Lightweight adaptation of bert through hybrid model compression". In: *arXiv preprint arXiv:2004.04124* (2020).

[181] Yoshitomo Matsubara et al. "Ensemble Transformer for Efficient and Accurate Ranking Tasks: an Application to Question Answering Systems". In: *arXiv preprint arXiv:2201.05767* (2022).

[182] Stephen Merity et al. "Pointer sentinel mixture models". In: *International Conference on Learning Representations.* 2017. arXiv: `1609.07843 [cs.CL]`.

[183] Paul Michel, Omer Levy, and Graham Neubig. "Are sixteen heads really better than one?" In: *arXiv preprint arXiv:1905.10650* (2019).

[184] Asit Mishra and Debbie Marr. "Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy". In: *arXiv preprint arXiv:1711.05852* (2017).

[185] Tom M Mitchell. "The need for biases in learning generalizations". In: (1980).

[186] Pavlo Molchanov et al. "Pruning convolutional neural networks for resource efficient inference". In: *arXiv preprint arXiv:1611.06440* (2016).

[187] Purnendu Mukherjee et al. *Real-Time Natural Language Understanding with BERT Using TensorRT, hhttps://developer.nvidia.com/blog/nlu-with-tensorrt-bert/.* 2019.

[188] Shashi Narayan, Shay B Cohen, and Mirella Lapata. "Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization". In: *arXiv preprint arXiv:1808.08745* (2018).

[189] Edwin G Ng et al. "Pushing the limits of non-autoregressive speech recognition". In: *arXiv preprint arXiv:2104.03416* (2021).

[190] Xuefei Ning et al. *Skeleton-of-Thought: Large Language Models Can Do Parallel Decoding.* 2023. arXiv: `2307.15337 [cs.CL]`.

[191] NVDIA Nemo. *https://github.com/NVIDIA/NeMo.*

[192] NVDLA Primer. *http://nvdla.org/primer.html.* 2021.

[193] NVIDIA. *TensorRT: https://developer.nvidia.com/tensorrt.* 2018.

[194] Sangyun Oh et al. "Non-uniform Step Size Quantization for Accurate Post-training Quantization". In: *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XI.* Springer. 2022, pp. 658–673.

[195] OpenAI. *GPT-4 Technical Report.* 2023. arXiv: `2303.08774 [cs.CL]`.

[196] OpenAI. *New models and developer products announced at DevDay.* 2023.

[197] Myle Ott et al. "FairSeq: A Fast, Extensible Toolkit for Sequence Modeling". In: *Proceedings of NAACL-HLT 2019: Demonstrations.* 2019.

[198] Charles Packer et al. *MemGPT: Towards LLMs as Operating Systems.* 2023. arXiv: `2310.08560 [cs.AI]`.

[199] Liu Pai. "QiaoNing at SemEval-2020 Task 4: Commonsense Validation and Explanation system based on ensemble of language model". In: *Proceedings of the Fourteenth Workshop on Semantic Evaluation.* 2020, pp. 415–421.

[200] Jing Pan et al. "ASAPP-ASR: Multistream CNN and self-attentive SRU for SOTA speech recognition". In: *arXiv preprint arXiv:2005.10469* (2020).

[201] Vassil Panayotov et al. "Librispeech: an ASR corpus based on public domain audio books". In: *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP).* IEEE. 2015, pp. 5206–5210.

[202] Daniel S Park et al. "Specaugment: A simple data augmentation method for automatic speech recognition". In: *arXiv preprint arXiv:1904.08779* (2019).

[203] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. "Value-aware quantization for training and inference of neural networks". In: *Proceedings of the European Conference on Computer Vision (ECCV).* 2018, pp. 580–595.

[204] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[205] Pruthvi Patel et al. "Is a question decomposition unit all we need?" In: 2022.

[206] Shishir G. Patil et al. *Gorilla: Large Language Model Connected with Massive APIs.* 2023. arXiv: `2305.15334 [cs.CL]`.

[207] David A Patterson. "Latency lags bandwith". In: *Communications of the ACM* 47.10 (2004), pp. 71–75.

[208] Mathias Perslev et al. "U-Time: A fully convolutional network for time series segmentation applied to sleep staging". In: *Advances in Neural Information Processing Systems* 32 (2019).

[209] Antonio Polino, Razvan Pascanu, and Dan Alistarh. "Model compression via distillation and quantization". In: *arXiv preprint arXiv:1802.05668* (2018).

[210] Reiner Pope et al. "Efficiently Scaling Transformer Inference". In: *arXiv preprint arXiv:2211.05102* (2022).

[211] Ofir Press et al. *Measuring and Narrowing the Compositionality Gap in Language Models*. 2023. arXiv: `2210.03350 [cs.CL]`.

[212] Yujia Qin et al. "Toolllm: Facilitating large language models to master 16000+ real-world apis". In: *arXiv preprint arXiv:2307.16789* (2023).

[213] Alec Radford et al. *Improving language understanding by generative pre-training.* 2018.

[214] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[215] Colin Raffel et al. "Exploring the limits of transfer learning with a unified text-to-text transformer". In: *arXiv preprint arXiv:1910.10683* (2019).

[216] Colin Raffel et al. "Exploring the limits of transfer learning with a unified text-to-text transformer". In: *The Journal of Machine Learning Research* 21.1 (2020), pp. 5485–5551.

[217] Alessandro Raganato, Yves Scherrer, and Jörg Tiedemann. "Fixed encoder self-attention patterns in transformer-based machine translation". In: *arXiv preprint arXiv:2002.10260* (2020).

[218] Pranav Rajpurkar et al. "SQuAD: 100,000+ questions for machine comprehension of text". In: *arXiv preprint arXiv:1606.05250* (2016).

[219] Mohammad Rastegari et al. "XNOR-Net: Imagenet classification using binary convolutional neural networks". In: *European Conference on Computer Vision*. Springer. 2016, pp. 525–542.

[220] Machel Reid et al. "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context". In: *arXiv preprint arXiv:2403.05530* (2024).

[221] Adriana Romero et al. "FitNets: Hints for thin deep nets". In: *arXiv preprint arXiv:1412.6550* (2014).

[222] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[223] C Rosset. "Turing-NLG: A 17-billion-parameter language model by microsoft". In: *Microsoft Blog* (2019).

[224] Jingqing Ruan et al. "Tptu: Task planning and tool usage of large language model-based ai agents". In: *arXiv preprint arXiv:2308.03427* (2023).

[225] Yangjun Ruan et al. "Identifying the Risks of LM Agents with an LM-Emulated Sandbox". In: *arXiv preprint arXiv:2309.15817* (2023).

[226] Mark Sandler et al. "MobilenetV2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2018, pp. 4510–4520.

[227] Victor Sanh, Thomas Wolf, and Alexander M Rush. "Movement pruning: Adaptive sparsity by fine-tuning". In: *arXiv preprint arXiv:2005.07683* 33 (2020), pp. 20378–20389.

[228] Victor Sanh et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *arXiv preprint arXiv:1910.01108* (2019).

[229] Teven Le Scao et al. "BLOOM: A 176B-Parameter Open-Access Multilingual Language Model". In: *arXiv preprint arXiv:2211.05100* (2022).

[230] Timo Schick et al. "Toolformer: Language models can teach themselves to use tools". In: *arXiv preprint arXiv:2302.04761* (2023).

[231] Nicol N Schraudolph. "A fast, compact approximation of the exponential function". In: *Neural Computation* 11.4 (1999), pp. 853–862.

[232] Tal Schuster et al. "Confident adaptive language modeling". In: *arXiv preprint arXiv:2207.07061* (2022).

[233] Chenze Shao et al. "Minimizing the bag-of-ngrams difference for non-autoregressive neural machine translation". In: *Proceedings of the AAAI conference on artificial intelligence.* Vol. 34. 01. 2020, pp. 198–205.

[234] Wenqi Shao et al. "Omniquant: Omnidirectionally calibrated quantization for large language models". In: *arXiv preprint arXiv:2308.13137* (2023).

[235] Noam Shazeer and Mitchell Stern. "Adafactor: Adaptive Learning Rates with Sublinear Memory Cost". In: *International Conference on Machine Learning.* 2018, pp. 4596–4604.

[236] Sheng Shen et al. "Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT." In: *AAAI.* Vol. 34. 05. 2020, pp. 8815–8821.

[237] Yongliang Shen et al. *HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face.* 2023. arXiv: `2303.17580 [cs.CL]`.

[238] Kyuhong Shim, Jungwook Choi, and Wonyong Sung. "Understanding the role of self attention for efficient speech recognition". In: *International Conference on Learning Representations.* 2021.

[239] Noah Shinn et al. *Reflexion: Language Agents with Verbal Reinforcement Learning.* 2023. arXiv: `2303.11366 [cs.AI]`.

[240] Mohammad Shoeybi et al. "Megatron-LM: Training multi-billion parameter language models using gpu model parallelism". In: *arXiv preprint arXiv:1909.08053* (2019).

[241] Gil Shomron et al. "Post-training sparsity-aware quantization". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 17737–17748.

[242] Laurent Sifre and Stéphane Mallat. “Rigid-motion scattering for texture classification”. In: *arXiv preprint arXiv:1403.1687* (2014).

[243] K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *International Conference on Learning Representations*. 2015.

[244] Shaden Smith et al. “Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model”. In: *arXiv preprint arXiv:2201.11990* (2022).

[245] David So, Quoc Le, and Chen Liang. “The evolved transformer”. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 5877–5886.

[246] David R So et al. “Primer: Searching for efficient transformers for language modeling”. In: *arXiv preprint arXiv:2109.08668* (2021).

[247] Richard Socher et al. “Recursive deep models for semantic compositionality over a sentiment treebank”. In: *Proceedings of the 2013 conference on empirical methods in natural language processing*. 2013, pp. 1631–1642.

[248] Yifan Song et al. *RestGPT: Connecting Large Language Models with Real-World RESTful APIs*. 2023. arXiv: `2306.06624 [cs.CL]`.

[249] Aarohi Srivastava et al. “Beyond the imitation game: Quantifying and extrapolating the capabilities of language models”. In: *arXiv preprint arXiv:2206.04615* (2022).

[250] Mitchell Stern et al. “Insertion transformer: Flexible sequence generation via insertion operations”. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 5976–5985.

[251] Gilbert W Stewart. *Afternotes on numerical analysis*. SIAM, 1996.

[252] Theodore R. Sumers et al. *Cognitive Architectures for Language Agents*. 2023. arXiv: `2309.02427 [cs.AI]`.

[253] Siqi Sun et al. “Patient knowledge distillation for bert model compression”. In: *arXiv preprint arXiv:1908.09355* (2019).

[254] Zhiqing Sun et al. “Fast structured decoding for sequence models”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[255] Zhiqing Sun et al. “Mobilebert: a compact task-agnostic bert for resource-limited devices”. In: *arXiv preprint arXiv:2004.02984* (2020).

[256] Dídac Surís, Sachit Menon, and Carl Vondrick. *ViperGPT: Visual Inference via Python Execution for Reasoning*. 2023. arXiv: `2303.08128 [cs.CV]`.

[257] Mingxing Tan and Quoc V Le. “EfficientNet: Rethinking model scaling for convolutional neural networks”. In: *arXiv preprint arXiv:1905.11946* (2019).

[258] Raphael Tang et al. “Distilling task-specific knowledge from bert into simple neural networks”. In: *arXiv preprint arXiv:1903.12136* (2019).

[259] James W Thomas et al. *The libm library and floatingpoint arithmetic in HP-UX for Itanium-based systems*. Tech. rep. Technical report, Hewlett-Packard Company, Palo Alto, CA, USA, 2004.

[260] Romal Thoppilan et al. "Lamda: Language models for dialog applications". In: *arXiv preprint arXiv:2201.08239* (2022).

[261] Hugo Touvron et al. "Llama 2: Open foundation and fine-tuned chat models". In: *arXiv preprint arXiv:2307.09288* (2023).

[262] Hugo Touvron et al. "LLaMA: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971* (2023). arXiv: `2307.09288 [cs.CL]`.

[263] Hugo Touvron et al. "Training data-efficient image transformers & distillation through attention". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 10347–10357.

[264] Iulia Turc et al. "Well-read students learn better: On the importance of pre-training compact models". In: *arXiv preprint arXiv:1908.08962* (2019).

[265] Vincent Vanhoucke. "Learning visual representations at scale". In: 2014.

[266] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. Vol. 30. 2017, pp. 5998–6008.

[267] Roman Vershynin. "Introduction to the non-asymptotic analysis of random matrices". In: *arXiv preprint arXiv:1011.3027* (2010).

[268] Elena Voita et al. "Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned". In: *arXiv preprint arXiv:1905.09418* (2019).

[269] Alex Wang et al. "GLUE: A multi-task benchmark and analysis platform for natural language understanding". In: *arXiv preprint arXiv:1804.07461* (2018).

[270] Chengyi Wang et al. "Unispeech: Unified speech representation learning with labeled and unlabeled data". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 10937–10947.

[271] Hanrui Wang et al. "HAT: Hardware-aware transformers for efficient natural language processing". In: *arXiv preprint arXiv:2005.14187* (2020).

[272] Hongyu Wang et al. "DeepNet: Scaling Transformers to 1,000 Layers". In: *arXiv preprint arXiv:2203.00555* (2022).

[273] Kuan Wang et al. "HAQ: Hardware-Aware Automated Quantization". In: *In Proceedings of the IEEE conference on computer vision and pattern recognition* (2019).

[274] Lei Wang et al. "Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models". In: *arXiv preprint arXiv:2305.04091* (2023).

[275] Sinong Wang et al. "Linformer: Self-Attention with Linear Complexity". In: *arXiv preprint arXiv:2006.04768* (2020).

[276] Wenhui Wang et al. "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers". In: *arXiv preprint arXiv:2002.10957* (2020).

[277] Xuezhi Wang et al. *Self-Consistency Improves Chain of Thought Reasoning in Language Models*. 2023. arXiv: `2203.11171 [cs.CL]`.

[278] Yiren Wang et al. "Non-autoregressive machine translation with auxiliary regularization". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 5377–5384.

[279] Yizhong Wang et al. "Self-instruct: Aligning language models with self-generated instructions". In: *arXiv preprint arXiv:2212.10560* (2022).

[280] Yongqiang Wang et al. "Transformer-based acoustic modeling for hybrid speech recognition". In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 6874–6878.

[281] Edward Waring. "Vii. problems concerning interpolations". In: *Philosophical transactions of the royal society of London* (1779).

[282] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. "Neural network acceptability judgments". In: *Transactions of the Association for Computational Linguistics* 7 (2019), pp. 625–641.

[283] Bingzhen Wei et al. "Imitation learning for non-autoregressive neural machine translation". In: *arXiv preprint arXiv:1906.02041* (2019).

[284] Jason Wei et al. "Chain-of-thought prompting elicits reasoning in large language models". In: vol. 35. 2022, pp. 24824–24837.

[285] Xiuying Wei et al. "Outlier suppression: Pushing the limit of low-bit transformer language models". In: *arXiv preprint arXiv:2209.13325* (2022).

[286] Xiuying Wei et al. "Outlier Suppression+: Accurate quantization of large language models by equivalent and optimal shifting and scaling". In: *arXiv preprint arXiv:2304.09145* (2023).

[287] Sean Welleck et al. "Non-monotonic sequential text generation". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 6716–6726.

[288] Adina Williams, Nikita Nangia, and Samuel R Bowman. "A broad-coverage challenge corpus for sentence understanding through inference". In: *arXiv preprint arXiv:1704.05426* (2017).

[289] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[290] Thomas Wolf et al. "Transformers: State-of-the-art natural language processing". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 2020, pp. 38–45.

[291] Tomer Wolfson et al. “Break It Down: A Question Understanding Benchmark”. In: *Transactions of the Association for Computational Linguistics* (2020).

[292] Bichen Wu et al. “Mixed precision quantization of convnets via differentiable neural architecture search”. In: *arXiv preprint arXiv:1812.00090* (2018).

[293] Jiaxiang Wu et al. “Quantized convolutional neural networks for mobile devices”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2016, pp. 4820–4828.

[294] Xiaoxia Wu et al. “Extreme Compression for Pre-trained Transformers Made Simple and Efficient”. In: *arXiv preprint arXiv:2206.01859* (2022).

[295] Zhanghao Wu et al. “Lite transformer with long-short range attention”. In: *arXiv preprint arXiv:2004.11886* (2020).

[296] Guangxuan Xiao et al. “SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models”. In: *Proceedings of the 40th International Conference on Machine Learning.* Vol. 202. Proceedings of Machine Learning Research. PMLR, 23–29 Jul 2023, pp. 38087–38099.

[297] Binfeng Xu et al. *ReWOO: Decoupling Reasoning from Observations for Efficient Augmented Language Models.* 2023. arXiv: `2305.18323 [cs.CL]`.

[298] Canwen Xu et al. “Bert-of-theseus: Compressing bert by progressive module replacing”. In: *arXiv preprint arXiv:2002.02925* (2020).

[299] Jin Xu et al. “NAS-BERT: task-agnostic and adaptive-size BERT compression with neural architecture search”. In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining.* 2021, pp. 1933–1943.

[300] Yige Xu et al. “Improving bert fine-tuning via self-ensemble and self-distillation”. In: *arXiv preprint arXiv:2002.10345* (2020).

[301] Yuhui Xu et al. *Deep Neural Network Compression with Single and Multiple Level Quantization.* 2018. arXiv: `1803.03289 [cs.LG]`.

[302] Linting Xue et al. “mT5: A massively multilingual pre-trained text-to-text transformer”. In: *arXiv preprint arXiv:2010.11934* (2020).

[303] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. “Designing energy-efficient convolutional neural networks using energy-aware pruning”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2017, pp. 5687–5695.

[304] Zhilin Yang et al. “HotpotQA: A dataset for diverse, explainable multi-hop question answering”. In: *arXiv preprint arXiv:1809.09600* (2018).

[305] Zhilin Yang et al. “XLNet: Generalized autoregressive pretraining for language understanding”. In: *Advances in neural information processing systems.* 2019, pp. 5753–5763.

[306] Zonglin Yang et al. *Language Models as Inductive Reasoners*. 2022. arXiv: `2212.10923 [cs.CL]`.

[307] Shunyu Yao et al. "React: Synergizing reasoning and acting in language models". In: *arXiv preprint arXiv:2210.03629* (2022).

[308] Shunyu Yao et al. *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. 2023. arXiv: `2305.10601 [cs.CL]`.

[309] Shunyu Yao et al. *WebShop: Towards Scalable Real-World Web Interaction with Grounded Language Agents*. 2023. arXiv: `2207.01206 [cs.CL]`.

[310] Zhewei Yao et al. "HAWQV3: Dyadic Neural Network Quantization". In: *arXiv preprint arXiv:2011.10680* (2020).

[311] Zhewei Yao et al. "ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers". In: *arXiv preprint arXiv:2206.01861* (2022).

[312] Yichun Yin et al. "AutotinyBERT: Automatic hyper-parameter optimization for efficient pre-trained language models". In: *arXiv preprint arXiv:2107.13686* (2021).

[313] Gyeong-In Yu et al. "Orca: A distributed serving system for {Transformer-Based} generative models". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 521–538.

[314] Joonsang Yu et al. "NN-LUT: Neural Approximation of Non-Linear Operations for Efficient Transformer Inference". In: *arXiv preprint arXiv:2112.02191* (2021).

[315] Wenhao Yu et al. *IfQA: A Dataset for Open-domain Question Answering under Counterfactual Presuppositions*. 2023. arXiv: `2305.14010 [cs.CL]`.

[316] Zhihang Yuan et al. "RPTQ: Reorder-based Post-training Quantization for Large Language Models". In: *arXiv preprint arXiv:2304.01089* (2023).

[317] Ali Hadi Zadeh et al. "Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 811–824.

[318] Ofir Zafrir et al. "Q8BERT: Quantized 8bit bert". In: *arXiv preprint arXiv:1910.06188* (2019).

[319] Matei Zaharia et al. "The Shift from Models to Compound AI Systems". In: *BAIR Blog: https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/* (2024).

[320] Dongqing Zhang et al. "LQ-Nets: Learned quantization for highly accurate and compact deep neural networks". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 365–382.

[321] Frank Zhang et al. "Faster, simpler and more accurate hybrid asr systems using wordpieces". In: *arXiv preprint arXiv:2005.09150* (2020).

[322] Qian Zhang et al. “Transformer transducer: A streamable speech recognition model with Transformer encoders and RNN-T loss”. In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 7829–7833.

[323] Shucong Zhang et al. “On the usefulness of self-attention for automatic speech recognition with Transformers”. In: *2021 IEEE Spoken Language Technology Workshop (SLT)*. IEEE. 2021, pp. 89–96.

[324] Shucong Zhang et al. “Stochastic attention head removal: A simple and effective method for improving Transformer based ASR models”. In: *arXiv preprint arXiv:2011.04004* (2020).

[325] Susan Zhang et al. “OPT: Open pre-trained transformer language models”. In: *arXiv preprint arXiv:2205.01068* (2022). arXiv: `2205.01068 [cs.CL]`.

[326] Wei Zhang et al. “Ternarybert: Distillation-aware ultra-low bit bert”. In: *arXiv preprint arXiv:2009.12812* (2020).

[327] Xiaohui Zhang et al. “Benchmarking LF-MMI, CTC And RNN-T Criteria For Streaming ASR”. In: *2021 IEEE Spoken Language Technology Workshop (SLT)*. IEEE. 2021, pp. 46–51.

[328] Yifan Zhang et al. “QD-BEV: Quantization-aware View-guided Distillation for Multi-view 3D Object Detection”. In: (2023).

[329] Ying Zhang et al. “Towards end-to-end speech recognition with deep convolutional neural networks”. In: *arXiv preprint arXiv:1701.02720* (2017).

[330] Yu Zhang et al. “Pushing the limits of semi-supervised learning for automatic speech recognition”. In: *arXiv preprint arXiv:2010.10504* (2020).

[331] Zhenyu Zhang et al. “H2o: Heavy-hitter oracle for efficient generative inference of large language models”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[332] Ritchie Zhao et al. “Improving neural network quantization without retraining using outlier channel splitting”. In: *International conference on machine learning*. PMLR. 2019, pp. 7543–7552.

[333] Huaixiu Steven Zheng et al. *Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models*. 2023. arXiv: `2310.06117 [cs.LG]`.

[334] Andy Zhou et al. *Language Agent Tree Search Unifies Reasoning Acting and Planning in Language Models*. 2023. arXiv: `2310.04406 [cs.AI]`.

[335] Chunting Zhou, Graham Neubig, and Jiatao Gu. “Understanding knowledge distillation in non-autoregressive machine translation”. In: *arXiv preprint arXiv:1911.02727* (2019).

[336] Denny Zhou et al. “Least-to-Most Prompting Enables Complex Reasoning in Large Language Models”. In: *The Eleventh International Conference on Learning Representations*. 2023.

[337] Shuchang Zhou et al. “DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients”. In: *arXiv preprint arXiv:1606.06160* (2016).

Appendix A

Compute Optimization: Integer-only Transformer Quantization

A.1 Quantization Methods

Symmetric and Asymmetric Quantization

Symmetric and asymmetric quantization are two different methods for uniform quantization. Uniform quantization is a uniform mapping from floating point $x \in [x_{\min}, x_{\max}]$ to b-bit integer $q \in [-2^{b-1}, 2^{b-1}-1]$. Before the mapping, input x that does not fall into the range of $[x_{\min}, x_{\max}]$ should be clipped. In asymmetric quantization, the left and the right side of the clipping range can be different, i.e., $-x_{\min} \neq x_{\max}$. However, this results in a bias term that needs to be considered when performing multiplication or convolution operations [119]. For this reason, we only use symmetric quantization in this work. In symmetric quantization, the left and the right side of the clipping range must be equal, i.e., $-x_{\min} = x_{\max} = \alpha$, and the mapping can be represented as Equation 2.1.

Static and Dynamic Quantization

There is a subtle but important factor to consider when computing the scaling factor, S. Computing this scaling factor requires determining the range of parameters/activations (i.e., α parameter in Equation 2.1). Since the model parameters are fixed during inference, their range and the corresponding scaling factor can be precomputed. However, activations vary across different inputs, and thus their range varies. One way to address this issue is to use dynamic quantization, where the activation range and the scaling factor are calculated during inference. However, computing the range of activation is costly as it requires a scan over the entire data and often results in significant overhead. Static quantization avoids this runtime computation by precomputing a fixed range based on the statistics of activations during training, and then uses that fixed range during inference. As such, it does not have the runtime overhead of computing the range of activations. For maximum efficiency, we adopt static quantization, with all the scaling factors fixed during inference.

A.2 Error Term of Equation 2.3

As one can see, the polynomial approximation of Equation 2.3 exactly matches the data at the interpolating points (x_j, f_j). The error between a target function $f(x)$ and the polynomial approximation $L(x)$ is then:

$$|f(x) - L(x)| = \left| \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)\dots(x - x_n) \right|, \tag{A.1}$$

where ξ is some number that lies in the smallest interval containing $x_0, ..., x_n$. In general, this error reduces for large n (for a properly selected set of interpolating points). Therefore, a sufficiently high-order polynomial that interpolates a target function is guaranteed to be a good approximation for it. We refer interested readers to [251] for more details on polynomial interpolation.

A.3 Experimental Details

Implementation

In I-BERT, all the MatMul operations are performed with INT8 precision, and are accumulated to INT32 precision. Furthermore, the Embedding layer is kept at INT8 precision. Moreover, the non-linear operations (i.e., GELU, Softmax, and LayerNorm) are processed with INT32 precision, as we found that keeping them at high precision is important to ensure no accuracy degradation after quantization. Importantly, note that using INT32 for computing these operations has little overhead, as input data is already accumulated with INT32 precision, and these non-linear operations have linear computational complexity. We perform Requantization [310] operation after these operations to bring the precision down from INT32 back to INT8 so that the follow up operations (e.g., next MatMuls) can be performed with low precision.

Training

We evaluate I-BERT on the GLUE benchmark [269], which is a set of 9 natural language understanding tasks, including sentimental analysis, entailment, and question answering. We first train the pre-trained RoBERTa model on the different GLUE downstream tasks until the model achieves the best result on the development set. We report this as the baseline accuracy. We then quantize the model and perform quantization-aware fine-tuning to recover the accuracy degradation caused by quantization. We refer the readers to [310] for more details about the quantization-aware fine-tuning method for integer-only quantization. We search the optimal hyperparameters in a search space of learning rate $\{5\mathrm{e}-7, 1\mathrm{e}-6, 1.5\mathrm{e}-6, 2\mathrm{e}-6\}$, self-attention layer dropout $\{0.0, 0.1\}$, and fully-connected layer dropout $\{0.1, 0.2\}$, except for the one after GELU activation that is fixed to 0.0. We fine-tune up to 6 epochs for larger datasets (e.g., MNLI and QQP), and 12 epochs for the smaller datasets. We report the best accuracy of the resulting quantized model on the development set as I-BERT accuracy.

Accuracy Evaluation on the GLUE Tasks

For evaluating the results, we use the standard metrics for each task in GLUE. In particular, we use classification accuracy and F1 score for QQP [118] and MRPC [45], Pearson Correlation and Spearman Correlation for STS-B [15], and Mathews Correlation Coefficient for CoLA [282]. For the remaining tasks [288, 218, 247, 35], we use classification accuracy. For the tasks with multiple metrics, we report the average of them. Since there are two development sets for MNLI [288], i.e., MNLI-match (MNLI-m) for in-domain evaluation, and MNLI-mismatch (MNLI-mm) for cross-domain evaluation, and we report the accuracy on both datasets. We exclude WNLI [155] as it has relatively small dataset and shows an unstable behaviour [44].

Environment Setup for Latency Evaluation

We use TensorRT 7.2.1 to deploy and tune the latency of BERT-Base and BERT-Large models (both INT8 and FP32) on Google Cloud Platform virtual machine with a single Tesla T4 GPU, CUDA 11.1, and cuDNN 8.0.

We should also mention that the most efficient way of implementing BERT with TensorRT is to use NVIDIA's plugins [187] that optimize and accelerate key operations in the Transformer architecture via operation fusion. Our estimates are that INT8 inference using NVIDIA's plugins is about 2 times faster than naïvely using TensorRT APIs. However, we cannot modify those plugins to support our integer-only kernels as they are partially closed sourced and pre-compiled. Therefore, our latency evaluation is conducted without fully utilizing NVIDIA's plugins. This leaves us a chance for further optimization to achieve our latency speedup relative to FP32 even more significant. As such, one could expect the potential for a further $\sim 2\times$ speed up with INT8 quantization.

Appendix B

Memory Optimization: Dense-and-Sparse Quantization for Large Language Models

B.1 Data Skew in Per-channel Sparsity Pattern

Figure B.1 provides the distribution of nonzero entries per output channel across different linear layers in the first LLaMA-7B block. This plot shows that the nonzero distribution is heavily skewed, with a few channels containing a much larger proportion of nonzero values. This skewed distribution makes it challenging to efficiently perform computations using the sparse matrix, as it is difficult to distribute the nonzero elements evenly across parallel processing units. This motivates our modified kernel for handling channels with a large number of outliers in order to reduce the runtime overhead of the sparse matrices. As outlined in Table B.1, we observed over 100% added runtime overhead when employing a standard CSR-based kernel. However, if we allocate each thread to process a fixed number of nonzeros (rather than having each thread process an entire row) we were able to drastically reduce the runtime overhead to 10-20% with both sensitive values and outliers.

B.2 Ablation Studies

Sensitivity-Based Quantization.

In our ablation study, we investigate the impact of sensitivity-based weighted clustering on the performance of non-uniform quantization. In Table B.2, we compared the performance of sensitivity-based and sensitivity-agnostic approaches in the context of 3-bit quantization of the LLaMA-7B model. For sensitivity-agnostic quantization, we apply non-weighted k-means clustering at sparsity levels of 0%, 0.05%, and 0.45%. The results demonstrate that while non-uniform quantization alone can reduce the perplexity from 28.26 (of RTN uniform

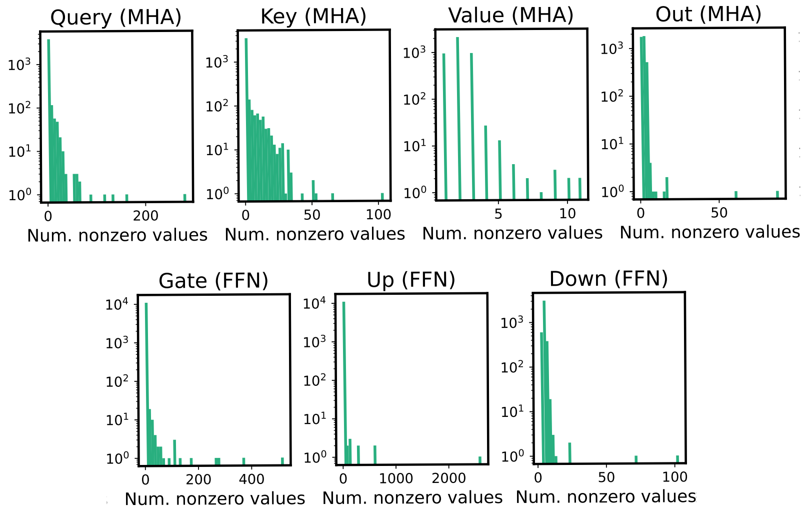


Figure B.1: Histograms of the number of non-zero entries per output channel in 7 different linear layers in the first LLaMA-7B block. The histograms reveal the presence of a few channels that contain significantly more non-zero entries than others, highlighting the skew in the sparsity patterns across different channels within the linear layers.

Table B.1: Hardware profiling of latency and memory usage using different kernel implementations for LLaMA 7B, 13B, 30B, and 65B quantized into 3-bit when generating 128 tokens on an A6000 GPU. The first row shows the performance of SqueezeLLM without sparsity as a reference. The second row shows the performance of SqueezeLLM with a sparsity level of 0.45% using a standard kernel for processing a CSR matrix. The third row shows the performance of SqueezeLLM with a sparsity level of 0.45% using a balanced sparse kernel that allocates 10 nonzeros per thread, thereby more efficiently handling skewed sparse matrices.

Sparse Kernel	**Sparsity Level**	**Latency (Seconds)**				**Peak Memory (GB)**			
		7B	13B	30B	65B	7B	13B	30B	65B
-	0%	1.5	2.4	4.0	7.6	2.9	5.4	12.5	24.5
Standard	0.45%	3.9	6.2	12.5	14.4	3.2	5.8	13.7	28.0
Balanced	0.45%	1.7	2.6	4.4	8.8	3.1	5.8	14.7	28.0

quantization) to 18.08 without considering sensitivity, incorporating sensitivity-based clustering is critical in reducing the perplexity to 7.75. This improvement is consistent across all sparsity levels.

Table B.2: Ablation study comparing sensitivity-agnostic and sensitivity-based non-uniform quantization on the LLaMA-7B model with 3-bit quantization, measured by perplexity on the C4 benchmark. The baseline model in FP16 achieves a perplexity of 7.08.

Method	**Sensitivity-Agnostic** (↓)	**Sensitivity-Based** (↓)
SqueezeLLM	18.08	**7.75**
SqueezeLLM (0.05%)	8.10	**7.67**
SqueezeLLM (0.45%)	7.61	**7.56**

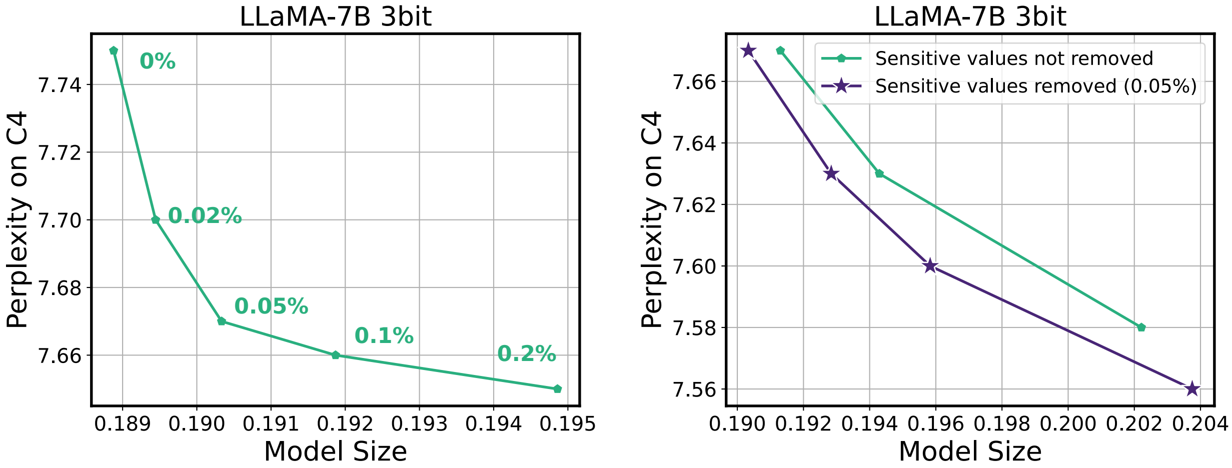


Figure B.2: (Left) Model size (normalized by the size of the FP16 model) and perplexity trade-off with different percentages of sensitive values included in the sparse matrix. Here, no outlier values are included in the sparse matrix. (Right) Comparison of the performance when the sensitive values are not removed as the sparse matrix (only outlier values are removed) to the case where 0.05% of the sensitive values are removed. In both cases, the trade-offs are obtained by controlling the percentage of outlier values included in the sparse matrix.

Impact of Sparsity Levels on SqueezeLLM

In Figure B.2 (Left), we present the perplexity results of the 3-bit quantized LLaMA-7B model on the C4 benchmarks, with varying percentages of sensitive values extracted as the sparse matrix, ranging from 0% to 0.2%. The plot demonstrates that the perplexity gain diminishes as the sparsity level of the sensitive values exceeds 0.05%. Therefore, we maintain a fixed sparsity level of 0.05% for the sensitive values throughout all experiments.

Furthermore, in Figure B.2 (Right), we compare the performance when the sensitive values are not removed as the sparse matrix (only outlier values are removed) to the case where 0.05% of the sensitive values are removed. In both scenarios, we control the sparsity level by increasing the percentage of outlier values included in the sparse matrix to obtain

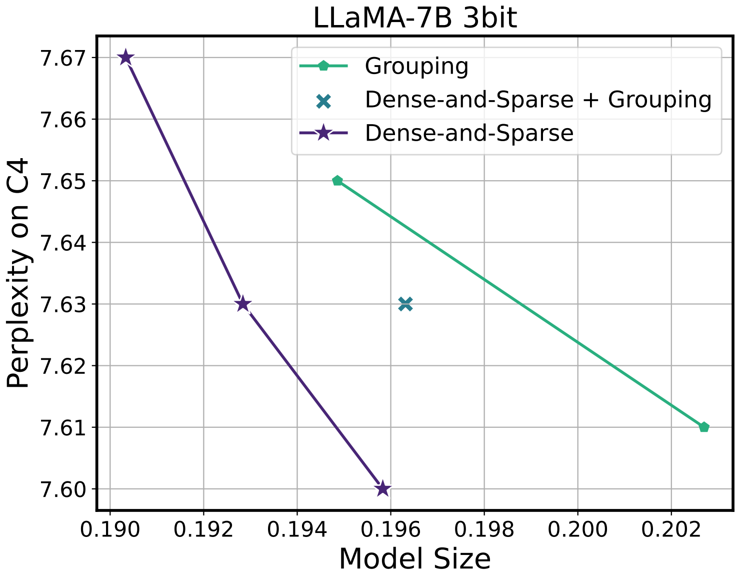


Figure B.3: Model size (normalized by the size of the FP16 model) and perplexity trade-offs of grouping and the Dense-and-Sparse decomposition on 3-bit quantization of LLaMA-7B. Here, we compare SqueezeLLM with (i) grouping using group sizes of 1024 and 512 (green), (ii) a hybrid approach that combines a group size of 1024 with a sparsity level of 0.05% (blue), and (iii) the Dense-and-Sparse decomposition approach with varying sparsity levels (violet). The pure Dense-and-Sparse decomposition always outperforms both grouping and the hybrid approach.

the trade-off curves. The results indicate that the sparsity configuration with both sensitive values and outlier values consistently outperforms the configuration with only outlier values.

Impact of Grouping on SqueezeLLM

In Figure B.3, we explore the effectiveness of incorporating grouping into SqueezeLLM as an alternative approach to improve quantization performance. We compare three configurations: SqueezeLLM with (i) grouping using group sizes of 1024 and 512 (green), (ii) a hybrid approach that combines a group size of 1024 with a sparsity level of 0.05% (blue), and (iii) the Dense-and-Sparse decomposition approach with varying sparsity levels (violet), where 0.05% of sensitive values are kept and the percentage of outlier values is adjusted. The results clearly demonstrate that both grouping and the hybrid approach result in suboptimal trade-offs compared to the pure Dense-and-Sparse decomposition approach.

This can be attributed to two factors. First, the Dense-and-Sparse decomposition is a direct solution to the outlier issue. In contrast, while grouping can mitigate the impact of outliers to some extent by isolating them within individual groups, it does not provide a direct solution to this issue. Second, grouping can introduce significant overhead in terms

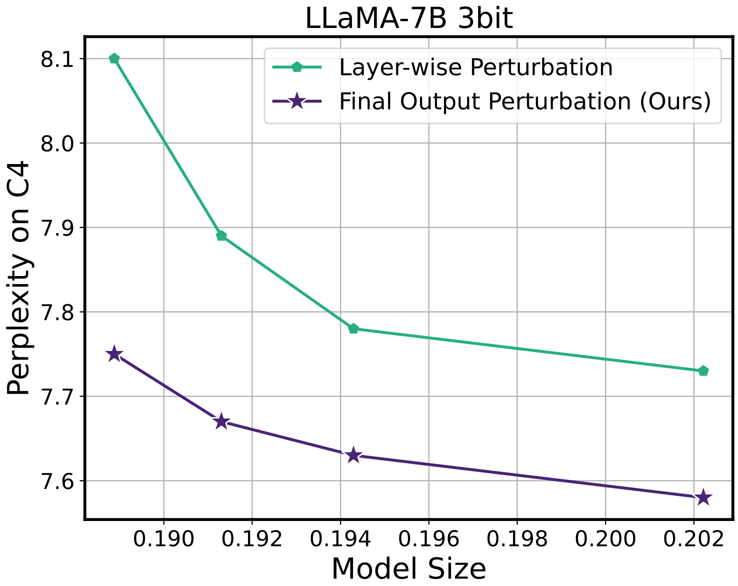


Figure B.4: Model size (normalized by the size of the FP16 model) and perplexity trade-offs for 3-bit quantization of the LLaMA-7B model using layer-wise perturbation minimization versus final output perturbation minimization as a non-uniform quantization objective. The trade-off is obtained by adjusting the sparsity level of the outliers being extracted. Across all sparsity levels, the OBD framework, which is the foundation for SqueezeLLM, consistently outperforms the OBS framework as an alternative approach.

of storage requirements when combined with non-uniform quantization, since it needs to store one LUT per group. This can be a considerable overhead compared to the uniform quantization approach, where only a scaling and zero point value per group need to be stored.

Comparison of Optimization Objectives for Non-uniform Quantization: Minimizing Layer-wise Perturbation versus Final Output Perturbation

While our method targets minimizing the perturbation of the final output of the model during quantization, it is worth noting that minimizing the layer-wise perturbation can also be considered as an alternative. Most existing solutions for LLM quantization including GPTQ [61], AWQ [166], and SpQR [42] have used the latter objective, which aims to minimize the perturbation of output activations in individual layers. In this ablation study, we demonstrate that minimizing the final output perturbation is a superior objective to minimizing the layer-wise perturbation.

Table B.3: Perplexity scores on Wikitext2 for the LLaMA-2 7B model, quantized using non-uniform (SqueezeLLM's sensitivity-based quantization) and uniform (RTN) approaches with 3 and 4-bit precision with varying levels of sparsity.

Bit Width	**Sparsity Level (%)**	**Avg. Bit Width**	**Uniform (PPL)**	**Nonuniform (PPL)**
16-bit	0	16	5.47	**5.47**
4-bit	0	4.04	6.12	**5.62**
	0.05	4.09	5.95	**5.59**
	0.45	4.26	5.95	**5.57**
	2	5.01	5.95	**5.55**
	4.5	6.20	5.94	**5.53**
3-bit	0	3.02	542.00	**6.18**
	0.05	3.07	27.38	**6.05**
	0.45	3.24	26.58	**5.96**
	1.5	3.98	25.97	**5.81**
	4.5	5.18	23.58	**5.73**

When minimizing the layer-wise perturbation, the optimization objective for determining the non-uniform quantization configuration can be reformulated as $\arg\min_Q \|WX - W_Q X\|_2^2$, where X denotes a batch of input activations. This object can be approximated as a weighted k-means clustering problem, where each weight is weighted by the square of the corresponding input activation size. This indeed results in the activation-based sensitivity/importance metric as in the AWQ framework [166].

In Figure B.4, we compare the perplexity on the C4 dataset for 3-bit quantization of the LLaMA-7B model using both objectives. Across all sparsity levels obtained by adjusting the number of outliers being extracted, SqueezeLLM based on final loss perturbation minimization outperforms the alternative of using layer-wise perturbation minimization by a large margin of up to around 0.3 perplexity points.

Impact of Non-uniform Quantization versus Dense-and-Sparse Decomposition

In Table B.3, we perform a detailed analysis to further disambiguate the impact of non-uniform quantization and the Dense-and-Sparse decomposition.

Uniform vs. Non-uniform Quantization. As can be seen in Table B.3, across all bitwidths and sparsity levels, our non-uniform quantization has noticeable improvements over uniform quantization.

Sparsity Levels. Furthermore, we also report the results with varying sparsity levels of the Dense-and-Sparse decomposition in Table B.3. As expected, higher levels of sparsity consistently result in improved performance in any scenario. However, there are diminishing returns for larger values of sparse decomposition since only a small portion of the weight

Table B.4: Perplexity scores on C4 and WikiText2 for the LLaMA-2 7B model, quantized using SqueezeLLM with 4-bit and 3-bit with different sparsity level. In particular, the sparsity levels of 3-bit quantization are selected to match their average bit widths to that of 4-bit quantization without sparsity.

Bit Width	Sparsity Level (%)	Avg. Bit Width	C4 (PPL)	WikiText2 (PPL)
16-bit	0	16	6.97	5.47
4-bit	0	4.04	7.12	5.62
3-bit	1.5	3.98	7.35	5.81
	2.5	4.22	7.32	5.80

Table B.5: Peak memory requirement in GB when quantizing different LLaMA models.

Model	Peak Memory (GB)
LLaMA-7B	33
LLaMA-13B	61
LLaMA-30B	149
LLaMA-65B	292

values are outliers or sensitive. As a consequence, saving additional values into the sparse format does not help as much beyond a certain level, and instead results in higher average bitwidth. This is in line with the conclusions in the main experiments where we found a sparsity level of 0.45% sufficient for the performance gain.

Impact of Dense-and-Sparse Decomposition versus Precision

In Table B.4, we additionally demonstrate that increasing the bit width of the dense component results in higher improvement in perplexity compared to increasing the sparsity level. Note that 4-bit LLaMA-2 7B model without any sparsity outperforms the 3-bit counterparts with sparsity levels of 1.5% and 2.5% that have similar or even larger model sizes. This observation aligns with the sensitivity level ablation study in Appendix B.2, since the Dense-and-Sparse decomposition is only effective to the extent of removing the outliers and sensitive values from the parameters. Increasing the sparsity level beyond that will not be effective and results in diminishing returns.

B.3 Quantization Cost Analysis

Memory Requirement

In Table B.5, we report the memory requirement of SqueezeLLM when quantizing different model sizes from 7B to 65B. Note that our method can have a higher memory requirement

Table B.6: End-to-end latency breakdown of quantizing different LLaMA models. Latency is broken down into (i) Fisher information computation on a A100 system and (ii) sensitivity-based k-means clustering on Intel Xeon Gold 6126 with 48 cores. In the last column, we provide the end-to-end time for GPTQ as reported in the original paper.

Model	Fisher Computation (min)	K-means (min)	GPTQ (min)
LLaMA-7B	0.3	11	10
LLaMA-13B	0.6	17	21
LLaMA-30B	1.3	45	45
LLaMA-65B	2.5	80	96

Table B.7: Perplexity on C4 and Wikitext2 of the LLaMA2 7B model after 4-bit quantization, with varying sizes of the calibration dataset used for computing the Fisher information matrix.

# Data Examples	C4	Wikitext2
1	7.89	6.41
2	7.81	6.22
5	7.73	6.20
10	7.72	6.17
20	7.72	6.16
100	7.72	6.18

than GPTQ. This is because SqueezeLLM performs quantization based on minimizing the perturbation to the loss function of the model which requires computing the Fisher information matrix. GPTQ, on the other hand, performs quantization by minimizing the perturbation to the output activation of the individual layer, which does not require back-propagating the gradient through the model to compute the Fisher information matrix. However, this is a one-time cost, and as demonstrated below, this gradient computation process is fast, taking only 2-3 minutes even for the largest 65B model.

Quantization Time

In Table B.6, we additionally assess the end-to-end time for (i) computing the Fisher information on an A100 system and (ii) performing sensitivity-based K-means clustering on Intel Xeon Gold 6126 with 48 cores, which are two major procedures in SqueezeLLM. Note that the time for computing the Fisher information matrix is minimal, taking only 2.5 minutes with the largest 65B model. K-mean clustering can take 11 min for the 7B model and up to 80 min for the 65B model. Overall, the computational time requirement of SqueezeLLM is on par with that of GPTQ.

Table B.8: Perplexity on Wikitext2 of the LLaMA2 13B and 70B models quantized into 4, 3, and 2 bits using SqueezeLLM and QuIP [18]. For QuIP, we use the perplexity numbers that are reported in the original paper as well as our own reproduction using the official codebase. Following the perplexity evaluation method of the QuIP paper, we use sequence length of 4096 (different from other experiments that use sequence length of 2048).

Model	**Config.**	**Avg. Bit Width**	**LLaMA2-13B**	**LLaMA2-70B**
QuIP (original paper)	4-bit	4	-	3.53
QuIP (our repr)	4-bit	4	4.81	3.65
SqueezeLLM	4-bit	4.05	**4.67**	**3.21**
QuIP (original paper)	3-bit	3	-	3.85
QuIP (our repr)	3-bit	3	5.25	3.84
SqueezeLLM	3-bit	3.02	**5.01**	**3.55**
QuIP (original paper)	2-bit	2	-	6.33
QuIP (our repr)	2-bit	2	**20.54**	**6.20**
SqueezeLLM	2-bit	2.01	61.25	10.86
SqueezeLLM	2-bit + 0.1%	2.05	**7.91**	**5.04**
SqueezeLLM	2-bit + 0.45%	2.22	**7.43**	**4.71**

Data Efficiency

In Table B.7, we provide data efficiency analysis in terms of the number of data samples to calculate the Fisher information matrix (gradients) for sensitivity-based non-uniform quantization. While we used a calibration set of 100 data samples throughout the paper, a calibration set with as few as 10 examples is typically sufficient to achieve the desired quantization performance. Note that both GPTQ and AWQ require 100-200 data points for calibration as reported in the AWQ paper [166].

B.4 Comparison with Other Weight-only Quantization Methods

In this section, we compare SqueezeLLM with more recent weight-only quantization methods including QuIP [18] and OmniQuant [234].

Comparison with QuIP

Here, we provide a quantitative comparison of our method to QUIP. Given that the QuIP paper only reports performance evaluation of LLaMA2-70B among all LLaMA models, we enrich our comparison by additionally incorporating our own reproduction based on their official codebase. Different from other experiments that use sequence length of 2048, we use sequence length of 4096, following the perplexity evaluation method of the QuIP paper. In Table B.8, we compare the perplexity scores on Wikitext2 for LLaMA2 13B and 70B models

Table B.9: Perplexity on Wikitext2 of all LLaMA and LLaMA2 models quantized into 4 and 3 bits using SqueezeLLM and OmniQuant [18]. For OmniQuant, we directly use the perplexity numbers that are reported in the original paper.

Model	**Config.**	**Avg. Bit Width**	**7B**	**13B**	**30B**	**65B**	**2-7B**	**2-13B**	**2-70B**
Baseline	16-bit	16	5.68	5.09	4.1	3.53	5.47	4.88	3.32
Omniquant	4-bit	4	5.86	5.21	4.25	**3.71**	5.74	5.02	3.47
SqueezeLLM	4-bit	4.05	**5.79**	**5.18**	**4.22**	3.76	**5.62**	**4.99**	**3.41**
Omniquant	4-bit (g128)	4.24	5.77	5.17	4.19	**3.62**	5.58	**4.95**	3.4
SqueezeLLM	4-bit (0.45%)	4.27	**5.77**	**5.17**	**4.18**	3.63	**5.57**	4.96	**3.39**
Omniquant	3-bit	3	6.49	5.68	4.74	4.04	6.58	5.58	3.92
SqueezeLLM	3-bit	3.02	**6.32**	**5.60**	**4.66**	**4.05**	**6.18**	**5.36**	**3.77**
Omniquant	3-bit (g128)	3.24	6.15	**5.44**	4.56	3.94	6.03	5.28	3.78
SqueezeLLM	3-bit (0.45%)	3.24	**6.13**	5.45	**4.44**	**3.88**	**5.96**	**5.23**	**3.63**

Table B.10: Perplexity on Wikitext2 of all LLaMA2 models quantized into 2 bits using SqueezeLLM and OmniQuant [18]. For OmniQuant, we directly use the perplexity numbers that are reported in the original paper.

Model	**Config.**	**Avg. Bit Width**	**2-7B**	**2-13B**	**2-70B**
Baseline	16-bit	16	5.47	4.88	3.32
OmniQuant	2-bit	2	37.37	**17.21**	**7.81**
SqueezeLLM	2-bit	2.01	**35.49**	41.02	9.44
SqueezeLLM	2-bit (0.1%)	2.05	**13.64**	**8.56**	**5.38**
OmniQuant	2-bit (g128)	2.24	11.06	8.26	6.55
SqueezeLLM	2-bit (0.45%)	2.22	**10.79**	**7.91**	**4.99**

quantized to 4, 3, and 2-bit. Note that we did not include a comparison on LLaMA2 7B as we were unable to achieve reasonable performance with QuIP, as was also reported in [51].

The table indicates that dense-only SqueezeLLM consistently achieves superior performance over QUIP, across all model sizes and quantization bitwidth. With 2bit quantization, we noticed that solely relying on dense-only quantization may not yield results as competitive as those of QuIP. However, by incorporating just 0.1% sparsity (additional 0.05 bit; 0.05% outlier values + 0.05% sensitive values), SqueezeLLM significantly outperforms QuIP by a considerable margin.

Comparison with OmniQuant

In Table B.9, we compare the perplexity of our method to OmniQuant on WikiText2 using sequence length of 2048. In particular, the table reports the perplexity numbers of 4 and 3-bit quantized models across all LLaMA and LLaMA2 models. For OmniQuant, we directly use the numbers reported in the original paper. Omniquant and SqueezeLLM are grouped in the table so that their model sizes are roughly the same. This comparison demonstrates

Table B.11: Latency (s) and peak memory usage (GB) of 3-bit LLaMA when generating 1024 tokens on an A6000 GPU. The table compares the FP16 baseline, non-grouped and grouped GPTQ with activation ordering, and SqueezeLLM with different sparsity levels. For comparison, we include bitwidth and perplexity on the C4 benchmark.

Method	**Avg.**	**7B**			**13B**			**30B**			**65B**		
	bit	PPL	Lat	Mem	PPL	Lat	Mem	PPL	Lat	Mem	PPL	Lat	Mem
Baseline	16	7.08	26.5	13.1	6.61	47.0	25.2	5.98	OOM	OOM	5.62	OOM	OOM
GPTQ	3	7.55	12.6	3.3	6.22	19.1	6.0	5.76	36.8	13.8	5.58	60.2	26.2
SqueezeLLM	3.02	6.32	13.6	3.4	5.60	21.2	6.1	4.66	37.8	16.1	4.05	66.9	29.9
GPTQ (g128)	3.25	6.27	110.7	3.4	5.47	176.1	6.2	4.83	500.8	14.3	4.55	955.2	27.3
SqueezeLLM (0.45%)	3.24	6.13	14.6	3.6	5.45	22.2	6.5	4.44	42.5	17.4	3.88	82.35	32.4

that SqueezeLLM generally outperforms OmniQuant with the same model size and memory constraints.

Additionally, Table B.10 demonstrates the same comparison using 2-bit quantization. With 2-bit quantization, the table shows that OmniQuant without grouping outperforms dense-only SqueezeLLM on the 13B and 70B models. This can be attributed to OmniQuant's learnable clipping ranges via a few iterations of training that effectively account for outliers. SqueezeLLM's sensitivity-based nonuniform quantization alone does not inherently address this. Handling outliers can be particularly critical for 2-bit quantization where weights should be represented with only four values. Nevertheless, introducing a 0.1% sparsity remarkably enhances SqueezeLLM's performance with a minimal memory overhead increase of 0.05 bit. This perplexity improvement is also persistent when comparing OmniQuant with a group size 128 and SqueezeLLM at a 0.45% sparsity level with roughly the same size.

B.5 Additional Hardware Profiling Results

In Table B.11, we provide additional hardware profiling results using a sequence length of 1024. All the experimental setups and details are identical to Section 3.4 and Table 3.3.

Additionally, in Table B.12, we demonstrate that our custom CUDA kernels (both including and without including outliers) attain significant speedups of 1.5-2.5× relative to the fp16 baseline. These results were obtained without any additional optimizations or tuning specifically for the A100, demonstrating how our kernels are easily portable across different GPUs and do not introduce complexity.

Table B.12: Matrix-vector kernel runtime (in seconds) for generating 128 tokens, benchmarked on an A100 GPU. Our kernel implementation attains 1.5-2.5× performance speedups relative to the fp16 matrix-vector multiply kernel across different model sizes without any additional optimizations or tuning. We include GPTQ (with group size 128) without reordering for comparison against the latency of uniform quantization with grouping.

Method	**Bit Width**	**Model**		
		7B	13B	30B
Baseline	16	1.21	2.32	5.56
GPTQ (g128)	4	0.92	1.51	3.24
SqueezeLLM	4	0.83	1.52	3.66
SqueezeLLM (0.45%)	4	1.09	1.87	4.25
GPTQ (g128)	3	0.62	1.03	2.39
SqueezeLLM	3	0.56	0.97	2.26
SqueezeLLM (0.45%)	3	0.83	1.32	2.86

B.6 Additional Experiment Results

Perplexity Evaluation

In Table B.13, we provide the full experimental results on LLaMA [262]. Furthermore, in Table B.14, B.15 and B.16, we provide additional experimental results on LLaMA2 [261] and OPT [325] models.

5-shot MMLU Evaluation

In Table B.17, we provide additional results on 5-shot MMLU evaluation using the Vicuna v1.1 (7/13B) and Vicuna v1.3 (7/13/33B) models. We see a similar trend as the zero-shot MMLU evaluation results where SqueezeLLM consistently outperforms the baseline quantization methods with the same model size.

B.7 Limitations

While our empirical results primarily focus on generation tasks, the proposed ideas in this work are not inherently limited to decoder architectures. However, we have not yet conducted thorough assessments of our framework's effectiveness on encoder-only or encoder-decoder architectures, as well as other neural network architectures. Additionally, it is important to note that our hardware performance modeling approach relies on a simulation-based method using a roofline model, which entails making simplified assumptions about the hardware's inference pipeline.

Table B.13: Perplexity comparison of LLaMA-30B and 65B models quantized into 4 and 3 bits using different methods including RTN, GPTQ, AWQ and SpQR on C4 and WikiText-2. We compare the performance of GPTQ, AWQ, and SqueezeLLM in groups based on similar model sizes. In the first group, we compare dense-only SqueezeLLM with non-grouped GPTQ. In the subsequent groups, we compare SqueezeLLM with different levels of sparsity to GPTQ and AWQ with different group sizes. [†] SpQR does not report their near-3-bit performance. However, in the case of 65B model, its 3-bit perplexity on Wikitext-2 can be inferred from the trade-off curve in Figure 8 of their paper. This comparison indicates that the gap between SpQR and SqueezeLLM can be larger in the lower-bitwidth regimes.

LLaMA-30B	**3-bit**			**4-bit**		
Method	**Avg. Bits**	**PPL** (↓)		**Avg. Bits**	**PPL** (↓)	
	(comp. rate)	C4	Wiki	(comp. rate)	C4	Wiki
Baseline	16	5.98	4.10	16	5.98	4.10
RTN	3 (5.33)	28.53	14.89	4 (4.00)	6.33	4.54
GPTQ	3 (5.33)	7.31	5.76	4 (4.00)	6.20	4.43
SpQR	-	-	-	3.89 (4.11)	6.08	4.25
SqueezeLLM	3.02 (5.31)	**6.37**	**4.66**	4.03 (3.97)	**6.06**	**4.22**
GPTQ (g128)	3.25 (4.92)	6.47	4.83	4.25 (3.77)	6.07	4.24
AWQ (g128)	3.25 (4.92)	6.38	4.63	4.25 (3.77)	6.05	4.21
SqueezeLLM (0.45%)	3.25 (4.92)	**6.23**	**4.44**	4.25 (3.77)	**6.04**	**4.18**

LLaMA-65B	**3-bit**			**4-bit**		
Method	**Avg. Bits**	**PPL** (↓)		**Avg. Bits**	**PPL** (↓)	
	(comp. rate)	C4	Wiki	(comp. rate)	C4	Wiki
Baseline	16	5.62	3.53	16	5.62	3.53
RTN	3 (5.33)	12.77	10.59	4 (4.00)	5.86	3.92
GPTQ	3 (5.33)	6.70	5.58	4 (4.00)	5.81	4.11
SpQR	3 (5.33)	-	4.2[†]	3.90 (4.10)	5.70	**3.68**
SqueezeLLM	3.02 (5.30)	**5.99**	**4.05**	4.04 (3.96)	**5.69**	3.76
GPTQ (g128)	3.25 (4.92)	6.01	4.55	4.25 (3.77)	5.69	3.76
AWQ (g128)	3.25 (4.92)	5.94	4.00	4.25 (3.77)	5.68	3.67
SqueezeLLM (0.45%)	3.24 (4.94)	**5.84**	**3.88**	4.26 (3.76)	**5.67**	**3.63**

Table B.14: Perplexity comparison of LLaMA2 models quantized into 4 and 3 bits using different methods including RTN, GPTQ, AWQ and SpQR on C4 and WikiText-2. We compare the performance of GPTQ, AWQ, and SqueezeLLM in groups based on similar model sizes. In the first group, we compare dense-only SqueezeLLM with non-grouped GPTQ. In the subsequent groups, we compare SqueezeLLM with different levels of sparsity to GPTQ and AWQ with different group sizes. Note that all GPTQ results are with activation reordering.

LLaMA2-7B	**3-bit**			**4-bit**		
Method	**Avg. Bits**	**PPL** (↓)		**Avg. Bits**	**PPL** (↓)	
	(comp. rate)	C4	Wiki	(comp. rate)	C4	Wiki
Baseline	16	6.97	5.47	16	6.97	5.47
RTN	3 (5.33)	404.45	542.86	4 (4.00)	7.72	6.12
GPTQ	3 (5.33)	10.45	8.97	4 (4.00)	7.42	5.90
SqueezeLLM	3.02 (5.29)	**7.72**	**6.18**	4.05 (3.95)	**7.12**	**5.62**
GPTQ (g128)	3.24 (4.93)	7.97	6.25	4.24 (3.77)	7.23	5.72
AWQ (g128)	3.24 (4.93)	7.84	6.24	4.24 (3.77)	7.13	5.72
SqueezeLLM (0.45%)	3.24 (4.93)	**7.51**	**5.96**	4.27 (3.75)	**7.08**	**5.57**

LLaMA2-13B	**3-bit**			**4-bit**		
Method	**Avg. Bits**	**PPL** (↓)		**Avg. Bits**	**PPL** (↓)	
	(comp. rate)	C4	Wiki	(comp. rate)	C4	Wiki
Baseline	16	6.47	4.88	16	6.47	4.88
RTN	3 (5.33)	12.50	10.68	4 (4.00)	6.83	5.20
GPTQ	3 (5.33)	8.27	6.17	4 (4.00)	6.74	5.08
SqueezeLLM	3.02 (5.30)	**6.97**	**5.36**	4.04 (3.96)	**6.57**	**4.99**
GPTQ (g128)	3.25 (4.92)	7.06	5.31	4.25 (3.77)	6.57	**4.96**
AWQ (g128)	3.25 (4.92)	6.94	5.32	4.25 (3.77)	6.56	4.97
SqueezeLLM (0.45%)	3.24 (4.94)	**6.82**	**5.23**	4.26 (3.76)	**6.54**	**4.96**

LLaMA2-70B	**3-bit**			**4-bit**		
Method	**Avg. Bits**	**PPL** (↓)		**Avg. Bits**	**PPL** (↓)	
	(comp. rate)	C4	Wiki	(comp. rate)	C4	Wiki
Baseline	16	5.52	3.32	16	5.52	3.32
RTN	3 (5.33)	10.02	7.52	4 (4.00)	5.80	3.67
GPTQ	3 (5.33)	6.69	4.86	4 (4.00)	5.70	3.59
SqueezeLLM	3.02 (5.30)	**5.83**	**3.77**	4.04 (3.96)	**5.58**	**3.41**
GPTQ (g128)	3.25 (4.92)	5.87	3.88	4.25 (3.77)	5.59	3.42
AWQ (g128)	3.25 (4.92)	5.81	3.74	4.25 (3.77)	5.58	3.41
SqueezeLLM (0.45%)	3.24 (4.94)	**5.73**	**3.63**	4.26 (3.76)	**5.57**	**3.39**

Table B.15: Perplexity comparison of OPT 1.3B, 2.7B, and 6.7B models quantized into 4 and 3 bits using different methods including RTN, GPTQ, AWQ and SpQR on C4 and WikiText-2. We compare the performance of GPTQ, AWQ, and SqueezeLLM in groups based on similar model sizes. In the first group, we compare dense-only SqueezeLLM with non-grouped GPTQ. In the subsequent groups, we compare SqueezeLLM with different levels of sparsity to GPTQ and AWQ with different group sizes. Note that all GPTQ results are with activation reordering. "div" means that the perplexity is diverged.

OPT-1.3B	**3-bit**			**4-bit**		
Method	**Avg. Bits**	**PPL** (↓)		**Avg. Bits**	**PPL** (↓)	
	(comp. rate)	C4	Wiki	(comp. rate)	C4	Wiki
Baseline	16	14.72	14.62	16	14.72	14.62
RTN	3 (5.43)	div.	div.	4 (4)	24.68	48.19
SqueezeLLM	3.04 (5.26)	**16.42**	**16.30**	4.09 (3.91)	**15.01**	**14.94**
AWQ (g128)	3.25 (4.93)	16.28	16.32	4.25 (3.77)	15.04	14.95
SqueezeLLM (0.5%)	3.25 (4.92)	**15.84**	**15.76**	4.30 (3.72)	**14.94**	**14.83**

OPT-2.7B	**3-bit**			**4-bit**		
Method	**Avg. Bits**	**PPL** (↓)		**Avg. Bits**	**PPL** (↓)	
	(comp. rate)	C4	Wiki	(comp. rate)	C4	Wiki
Baseline	16	13.17	12.47	16	13.17	12.47
RTN	3 (5.33)	div.	div.	4 (4)	17.52	16.92
SqueezeLLM	3.04 (5.26)	**14.45**	**13.85**	4.07 (3.93)	**13.38**	**12.80**
AWQ (g128)	3.25 (4.93)	16.28	16.32	4.25 (3.77)	13.39	12.73
SqueezeLLM (0.5%)	3.25 (4.92)	**13.88**	**13.43**	4.29 (3.73)	**13.30**	**12.60**

OPT-6.7B	**3-bit**			**4-bit**		
Method	**Avg. Bits**	**PPL** (↓)		**Avg. Bits**	**PPL** (↓)	
	(comp. rate)	C4	Wiki	(comp. rate)	C4	Wiki
Baseline	16	11.74	10.86	16	11.74	10.86
RTN	3 (5.33)	div.	div.	4 (4)	13.38	12.10
SpQR	-	-	-	3.94 (4.06)	11.98	11.04
SqueezeLLM	3.02 (5.29)	**12.44**	**11.70**	4.05 (3.96)	**11.85**	**11.03**
SpQR	-	-	-	4.27 (3.74)	11.88	**10.91**
AWQ (g128)	3.25 (4.92)	12.30	11.41	4.25 (3.77)	11.86	10.93
SqueezeLLM (0.5%)	3.26 (4.90)	**12.18**	**11.31**	4.28 (3.73)	**11.83**	10.92

Table B.16: Perplexity comparison of OPT 13B and 30B models quantized into 4 and 3 bits using different methods including RTN, GPTQ, AWQ and SpQR on C4 and WikiText-2. We compare the performance of GPTQ, AWQ, and SqueezeLLM in groups based on similar model sizes. In the first group, we compare dense-only SqueezeLLM with non-grouped GPTQ. In the subsequent groups, we compare SqueezeLLM with different levels of sparsity to GPTQ and AWQ with different group sizes. Note that all GPTQ results are with activation reordering. "div" means that the perplexity is diverged.

OPT-13B	**3-bit**			**4-bit**		
Method	**Avg. Bits** (comp. rate)	**PPL** (↓) C4	Wiki	**Avg. Bits** (comp. rate)	**PPL** (↓) C4	Wiki
Baseline	16	11.20	10.12	16	11.20	10.12
RTN	3 (5.33)	div.	div.	4 (4)	12.35	11.32
SpQR	-	-	-	3.93 (4.07)	11.34	10.28
SqueezeLLM	3.02 (5.29)	**12.69**	**11.76**	4.05 (3.96)	**11.29**	**10.24**
SpQR	-	-	-	4.27 (3.74)	11.27	**10.22**
AWQ (g128)	3.25 (4.92)	12.61	10.67	4.25 (3.77)	11.28	**10.22**
SqueezeLLM (0.5%)	3.26 (4.90)	**11.57**	**10.54**	4.28 (3.73)	**11.26**	**10.22**

OPT-30B	**3-bit**			**4-bit**		
Method	**Avg. Bits** (comp. rate)	**PPL** (↓) C4	Wiki	**Avg. Bits** (comp. rate)	**PPL** (↓) C4	Wiki
Baseline	16	10.69	9.56	16	10.69	9.56
RTN	3 (5.33)	div.	div.	4 (4)	11.90	10.98
SpQR	-	-	-	3.94 (4.06)	10.78	**9.54**
SqueezeLLM	3.01 (5.31)	**11.10**	**10.17**	4.03 (3.97)	**10.75**	9.65
SpQR	-	-	-	4.26 (3.76)	10.73	**9.50**
AWQ (g128)	3.25 (4.92)	10.96	9.85	4.25 (3.77)	10.75	9.59
SqueezeLLM (0.5%)	3.26 (4.90)	**10.93**	**9.77**	4.28 (3.73)	**10.72**	9.61

Table B.17: Comparison of PTQ methods on five-shot MMLU accuracy applied to Vicuna v1.1 and v1.3. We add peak memory usage in GB for comparison.

Method	**Avg. bit**	**7B (v1.1)** Acc	Mem	**13B (v1.1)** Acc	Mem	**7B (v1.3)** Acc	Mem	**13B (v1.3)** Acc	Mem	**33B (v1.3)** Acc	Mem
Baseline	16	45.3%	12.7	50.0%	24.6	45.6%	12.7	51.6%	24.6	60.1%	OOM
AWQ (g128)	4.25	44.1%	3.8	48.8%	7.2	44.8%	3.8	50.7%	7.2	59.6%	17.2
SqueezeLLM	4.05	44.3%	3.8	48.4%	6.9	44.3%	3.8	50.5%	6.9	59.6%	16.5
SqueezeLLM (0.45%)	4.26	**44.7%**	4.0	**49.7%**	7.3	**44.9%**	4.0	**51.4%**	7.3	**60.0%**	17.7
AWQ (g128)	3.25	41.4%	3.0	46.3%	5.7	42.5%	3.0	48.4%	5.7	56.3%	13.3
SqueezeLLM	3.02	40.4%	2.9	45.6%	5.4	41.0%	2.9	47.4%	5.4	55.7%	12.4
SqueezeLLM (0.45%)	3.24	**42.2%**	3.1	**48.2%**	5.8	**43.2%**	3.1	**48.8%**	5.8	**58.2%**	13.7

Appendix C

Efficient Inference Method: Speculative Decoding with Big Little Decoder

C.1 Experimental Details

Training Details

For machine translation, we use IWSLT 2017 German-English [16] and WMT 2014 German-English [9] as target benchmarks, and mT5 [302] as a target model. We use the 8-layer mT5-small and the 24-layer mT5-large as the small and large models. For summarization, we use XSUM [188] and CNN/DailyMail [99] as target benchmarks, and T5 [216] as a target model. We use T5-small and T5-large with 6 and 24 layers, respectively, for the small and large models. Table C.1 summarizes the size and configuration of each model. All the models are fine-tuned from the pre-trained checkpoints of the HuggingFace library [290] for 500k steps using a batch size of 16. We use Adafactor optimizer [235] with constant learning rate of {0.5, 1, 2, 5}e$-$4 for the small models and {0.5, 1}e$-$4 for the large models. We refer to the normally fine-tuned models on the validation datasets as the *baseline* small and large models.

When training *aligned* small models via the prediction alignment method described in Section 4.2, we first generate calibration datasets using the input sequences from the training datasets of each benchmark. We then use the fully trained large model to generate output sequences through greedy sampling with a beam size of 1. To ensure a fair comparison, we fine-tune pre-trained small models (rather than the baseline small models that are already fine-tuned on the training datasets) on the calibration datasets using the same training recipes and the number of training steps as described above. This decision is based on our observation that fine-tuning a baseline model using the calibration dataset tends to improve generation quality, likely due to the increased number of training examples and data augmentation effects, which makes it difficult to make a fair comparison between unaligned

Table C.1: Model configurations of the large and small models for each evaluation task. For comparison, the number of layers, hidden dimension, FFN dimension, and the number of decoder parameters (without embeddings) for each model are provided.

Task	**Model**	**# Layers**	**dim**	**FFN dim**	**# Params**
Machine Translation	mT5-large [302]	24	1024	2816	409M
	mT5-small [302]	8	512	1024	25M
Summarization	T5-large [216]	24	1024	4096	402M
	T5-small [216]	6	512	2048	25M

BiLD and aligned BiLD. However, in practice, one can obtain aligned models by applying the prediction alignment method directly to the fine-tuned baseline small models to achieve the best performance.

Evaluation Details

All inference evaluations including latency measurement are conducted on a single NVIDIA T4 GPU of a GCP n1-standard-4 instance with 4 vCPUs and 15GB memory. For inference, we use batch size 1, which is a common use case for online serving [232]. For the distance metric d in Equation 4.3 for the rollback policy, we use the cross-entropy loss between the small model's hard label and the large model's soft label. This measures the (negative log) likelihood of obtaining the small model's prediction from the large model's output. For BiLD inference, we sweep over different fallback and rollback thresholds to explore different trade-offs between generation quality and latency. For the machine translation tasks, we use fallback thresholds in [0.5, 0.9] and rollback thresholds in [1, 10]. For the summarization tasks, fallback thresholds in [0.2, 0.6] and rollback thresholds in [2, 6]. We keep the maximum generation length of the small model to 10 to avoid high rollback costs. In Appendix C.5, we provide a detailed analysis of how varying the fallback and rollback thresholds impacts the trade-offs between generation quality and latency in the BiLD framework.

C.2 Details of Early Exiting Strategy in the BiLD Framework

Training and Evaluation Details

The training and evaluation details for BiLD as well as for CALM are as follows.

BiLD. We use the mT5-small model as the large model and the first (out of 8) layer as the small model, and evaluate it on two machine translation benchmarks: IWSLT 2017 De-En and WMT 2014 De-En. To ensure consistency between the prediction made after the first layer and the one made after the last layer, we fine-tune the pre-trained mT5 model using the average loss of the first and the final layers, similar to [52, 232]. That is, $\mathcal{L} = \frac{1}{2}(\mathcal{L}_1 + \mathcal{L}_{-1})$ where $\mathcal{L}_1$ and $\mathcal{L}_{-1}$ are the negative log-likelihood loss after the first layer and the final layer. The prediction head is shared for these two layers. We fine the pre-trained mT5-small model on each benchmark for 500k steps using a batch size of 16. Similar to the main experiments, we use Adafactor optimizer [235] with constant learning rate of {0.5, 1, 2, 5}e−4. For evaluation, we use fallback thresholds in [0.2, 0.8] and rollback thresholds in [0.5, 1.5].

CALM. To reproduce CALM [232] in our experimental setup, we have fine-tuned the pre-trained mT5-small model on IWSLT 2017 De-En and WMT 2014 De-En datasets. We employ the averaged loss across all layers, i.e., $\mathcal{L} = \sum_{i=1}^{L} w_i \mathcal{L} i$, where $w_i = i / \sum_{j=1}^{L} j$, which was introduced in the paper to ensure the layer consistency. We use Adafactor optimizer [235] with constant learning rate of {0.5, 1, 2, 5}e−4 for 500k training steps. To make a fair comparison, we match the BLEU score of the fine-tuned model to that of BiLD's models Among the two training-free confidence measures introduced in the CALM paper, softmax-based and hidden-state saturation-based measures, we have chosen to use the latter approach as an early exiting criterion. That said, if the cosine similarity between the current layer's hidden states and the previous layer's hidden states exceeds a certain threshold, we perform early exiting. We have found that the softmax-based alternative is not applicable in our evaluation scenario due to the large output vocabulary (more than 200k for mT5, which is $\sim 10\times$ larger than T5), which significantly increases latency overhead. As described in the paper, when early exiting happens, the hidden states of the exited layer are propagated down to the remaining layers to compute the key and value caches. To achieve different trade-offs between latency and generation quality, we sweep over λ in [0.7, 0.98] and t in {0, 1, 2, 4, 8} in the decaying threshold function.

Performance Comparison between BiLD and CALM

Figure C.1 illustrates the BLEU score and latency curves of BiLD compared to CALM in the early exiting setting. In both tasks, our method achieves significantly better BLEU scores with the same latency speedup, yielding up to around 2 point better BLEU score in the $\sim 1.5\times$ speedup regime. This can be attributed to two factors. First, in BiLD, even if an early exited prediction (i.e., prediction made by the smaller model) is incorrect, it can be corrected and replaced using the rollback policy. Therefore, an error in the early exited layer is propagated less drastically to the future prediction. Second, the key and value caches for skipped layers are filled with actual values instead of being computed from the exiting layer's hidden states. This also leads to reduced error propagation and improved decoding stability.

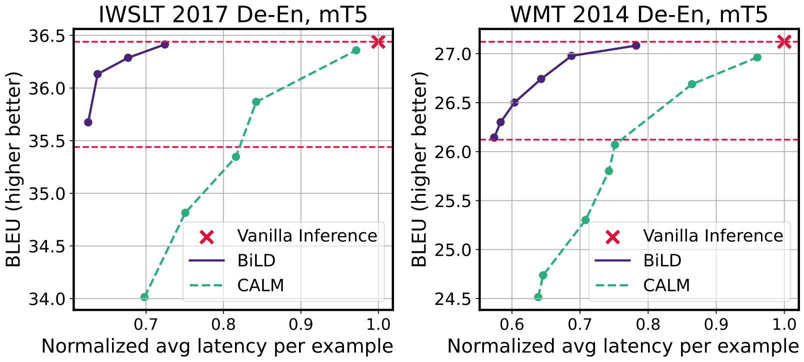


Figure C.1: The trade-off curves between inference latency and BLEU score for BiLD and CALM in the early exiting setting for (Left) IWSLT 2017 De-En and (Right) WMT 2014 De-En. The × marks indicate the vanilla inference latency and BLEU score of the mT5-small models. The horizontal lines indicate the vanilla inference score and 1 point degradation from it. BiLD outperforms CALM across all speedup regimes by up to 2 $\sim$ 2.5 points better BLEU score, demonstrating the effectiveness of our approach for the early exiting strategy.

C.3 Comparison with Other Speculative Decoding Frameworks

Concurrently and independently of our work, [156, 20] also propose an algorithm to accelerate generative inference using a more powerful model to score and speculatively sample predictions from a less powerful model. While the rejection sampling-based approach in [156, 20] offers unbiased estimators that match the stronger model's probability distributions, our extensive empirical evaluation shows that our approach can deliver superior latency-performance trade-offs, due to its non-random rollback (i.e., rejection) policy as well as the dynamic fallback window size. Below, we provide distinctions in detailed methodologies and quantitative comparison, as well as our insights on better latency and performance of our approach.

Differences in Methodology

While the idea of using two models with different sizes can be deemed similar to the speculative decoding frameworks in [156, 20], we have clear distinctions in detailed methodologies.

(1) Non-Random Prediction Rollback Approach: The primary difference lies in how we decide the rollback (e.g., rejection) of predictions from the small model. In our rollback policy, we propose to make the rejection decision based on the *distance* between the small and large model predictions, which differs from the rejection sampling policy outlined in [156, 20]. While [156, 20] propose an unbiased estimator on the large model's prediction, Figure 4.2 demonstrates that combining predictions from both models through our distance-

Table C.2: Comparison of BiLD to other rejection sampling based speculative sampling methods proposed in [156, 20] on IWSLT and XSUM. For BiLD, we include two BiLD configurations: the one that matches latency and the other that matches BLEU/ROUGE-L scores as compared to the rejection sampling based methods. Note that BiLD consistently outperforms other methods by achieving either (1) improved BLEU/ROUGE-L scores with equivalent latency gains, or (2) improved latency gains while retaining the same performance score.

Dataset	IWSLT		XSUM	
	BLEU	Speedup	ROUGE-L	Speedup
Vanilla Inference	40.32	-	35.08	-
Rejection Sampling Based [156, 20]	39.93	1.28×	35.00	1.25×
BiLD (Match Latency)	**40.54**	1.23×	**35.30**	1.42×
BiLD (Match BLEU/ROUGE-L)	39.87	**1.49**×	34.96	**1.50**×

based rejection approach can surpass the exclusive utilization of the large model's prediction probability. BiLD seeks to find and utilize this optimal performance point without introducing much runtime cost. We have a further discussion below about how our rejection policy benefits text-generation performance.

(2) Dynamic Fallback Window Size: Additionally, we introduce the *dynamic fallback window size* in our fallback policy. In [156, 20], the window size remains a fixed hyperparameter; however, it is also highlighted in [20] that the window size can have a noticeable impact on end-to-end latency. Our approach offers an efficient and robust solution: adjusting the window size at runtime based on the small model's confidence level in run-time. Our ablation study (Figure 4.5) demonstrates that omitting the fallback policy and periodically transitioning control to the large model, as proposed in [156, 20], can result in notable latency degradation.

(3) Model Alignment Enhancement: Beyond the core framework, we introduce a model alignment method to align the small model's predictions with those of the large model. This enhances the framework by reducing unnecessary rejections and can be incorporated with minimal adjustments to the training pipeline.

Quantitative Comparisons

In Table C.2, we provide a comprehensive quantitative comparison between our method and [156, 20] across two different datasets: IWSLT for machine translation and XSum for summarization. In order to ensure a fair comparison that isolates the impact of the frameworks themselves, we employ the baseline (non-aligned) small model for all experiments. We maintained the same evaluation setup and hyperparameter space that are outlined in Appendix C.1.

Table C.3: Comparison of the percentage of fallback and rollback (rejection) occurrences of BiLD and other rejection sampling based speculative sampling methods [156, 20]. While achieving even better BLEU/ROUGE-L scores in IWSLT and XSUM, BiLD involves noticeably fewer number of fallbacks and rollbacks, resulting in a significantly better latency speedup.

Task	Method	BLEU/ROUGE-L	Speedup	% Fallback	% Rollback
IWSLT	Rejection Sampling Based [156, 23]	39.93	1.28×	23.24%	9.81%
	BiLD (Better BLEU)	40.33	1.43×	**21.09%**	**1.56%**
XSUM	Rejection Sampling Based [156, 23]	35.00	1.25×	36.84%	24.24%
	BiLD (Better ROUGE-L)	35.12	1.48×	**32.33%**	**6.41%**

Table C.2 includes two BiLD configurations: the one that matches latency and the other that matches BLEU/ROUGE-L scores as compared to the rejection sampling-based methods. Across all experiments, BiLD consistently outperforms speculative decoding. It achieves either (1) notably improved BLEU/ROUGE-L scores with equivalent latency gains, or (2) superior latency gains while retaining the same BLEU/ROUGE-L scores.

Insights on Better Latency and Performance

Quantitative analysis reveals a consistent trend where our method, when compared to other speculative decoding frameworks, effectively enhances both text generation quality and latency. We provide insights and explanations into why our approach surpasses speculative decoding frameworks.

(1) Better text generation quality

Ensembling effect: The power of blending outputs from multiple models has been well-explored in various fields. This is also the case in open-source LLM models which exhibit diverse strengths and weaknesses due to variations in data, architectures, and hyperparameters, making different models complementary to each other [121]. In fact, we show such effects of blending multiple model outputs in Figure 2, where a combination of 20% of the large model's prediction with the small model's prediction outperforms the exact imitation of the large model's behavior. Our approach offers fallback and rollback policies that efficiently exploit optimal ensemble point, which produces superior output quality compared to the unbiased estimate of the large model in [156, 20].

Rollback policy that leads to higher performance: Our method adopts a rejection policy that completely discards the small model's prediction if it significantly deviates from the large model's counterpart, based on cross entropy-based distance metric. This contrasts with speculative decoding, where the decision involves a stochastic rejection sampling process. We empirically observe that BiLD 's *hard rejection policy* allows a better

BLEU/ROUGE-L score with significantly fewer number of rollbacks (rejections) than the *stocastic rejection policy* of speculative decoding as described in Table C.3. We hypothesize that this boost in predictive performance stems from our hard rollback policy, which prevents potentially erroneous predictions by ruling out stochasticity. We additionally hypothesize that such a strategy can address exposure bias, mitigating the impact of a single early-stage misprediction on subsequent predictions.

(2) Lower end-to-end latency Furthermore, our fallback policy introduces a dynamic fallback window size (i.e. number of small model's consecutive iterations) that is determined based on the run-time prediction confidence of the small model. This is in contrast with speculative decoding which adopts a static window size. The advantages of the dynamic window size are two-fold:

Less fallbacks: The dynamic window size enables the small model to persist in making predictions when it is confident, thus minimizing the unnecessary engagement of the large model. This is supported by Table C.3 where BiLD involves fewer number of fallbacks (23.24% → 21.09% and 36.84% → 32.33%) than [156, 20] while achieving better performance.

Less rollbacks/rejections: The dynamic window size further enables preemption of the small model when it is uncertain, which avoids rollback of the small model's wrong predictions. This is also supported by Table C.3 where BiLD involves significantly fewer number of rollbacks (9.81% → 1.56% and 24.24% → 6.41%) than [156, 20] while achieving better performance.

Minimizing both fallbacks and rollbacks/rejections reduces unnecessary computation which directly translates to end-to-end latency improvement.

C.4 BiLD with Sampling

Our approach isn't restricted to greedy decoding, but it can seamlessly extend to sampling methods. The only modification is to perform random sampling instead of greedy sampling when drawing a token from both the small model and the large model while using the same fallback and rollback policy. This is because both the fallback and rollback policies, based on the maximum prediction probability, serve as an effective indicator of the small model's uncertainty in prediction and potential inaccuracies, regardless of the sampling method. The following table illustrates the latency versus performance trade-off of the sampling-based approach, specifically using nucleus sampling with p=0.8, similar to [23]. This evaluation follows the same environment as other experiments outlined in the paper, and both cases involve aligned small models.

Table C.4 exhibits the BLEU/ROUGE-L score of BiLD on the IWSLT and XSUM benchmarks as well as their relative speedup. As can be seen in the table, our method exhibits a similar trend to the greedy decoding case. It achieves a ∼1.5× speedup without compromising performance and a ∼1.8× speedup with a modest 1-point BLEU/ROUGE score reduction.

Table C.4: BiLD with nucleus sampling (p=0.8) on IWSLT and XSUM. Similar to the greedy decoding case, our method achieves a ∼1.5× speedup without compromising performance and a ∼1.8× speedup with a modest 1-point BLEU/ROUGE score reduction with sampling.

Dataset	IWSLT		XSUM	
	BLEU	Speedup	ROUGE-L	Speedup
Vanilla Inference	39.24	-	34.00	-
BiLD	39.72 (+0.48)	1.51×	34.34 (+0.34)	1.22×
	39.26 (+0.02)	1.63×	34.04 (+0.04)	1.45×
	38.27 (−0.97)	1.80×	33.10 (−0.90)	1.85×

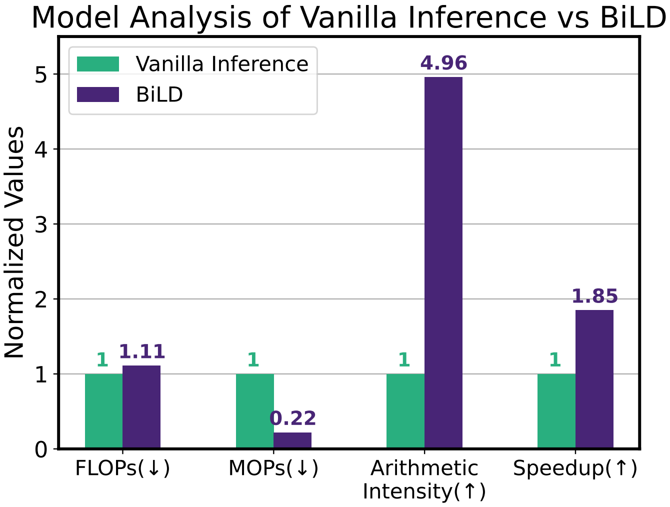
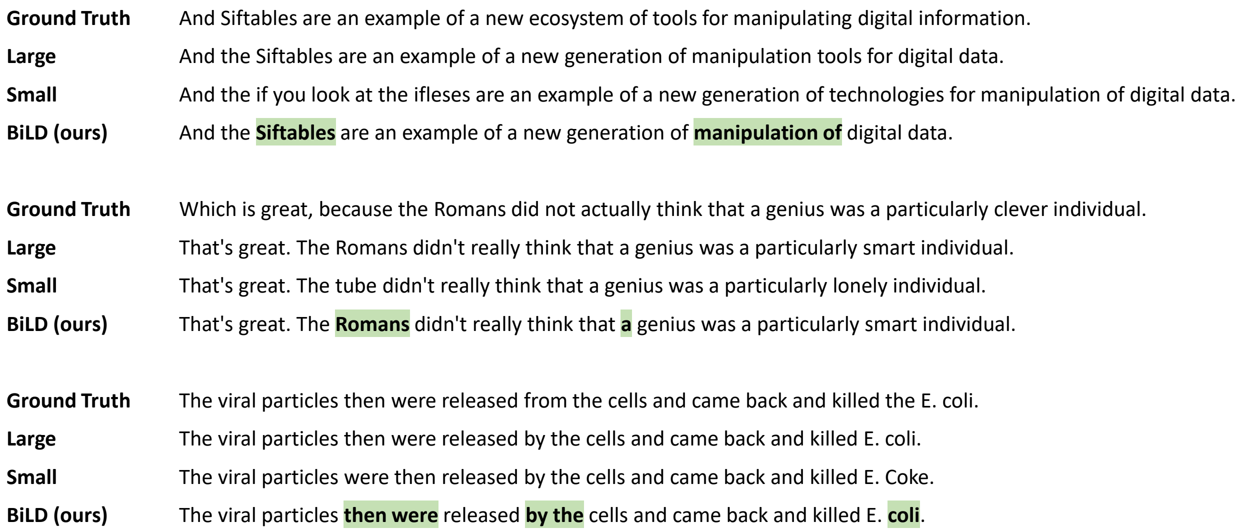


Figure C.2: FLOPs, MOPs (memory operations), arithmetic intensity, and latency speedup comparison of vanilla inference and BiLD on the CNN/DailyMail benchmark. BiLD approach results in a remarkable reduction in MOPs due to the improved token-level parallelism, resulting in significantly higher arithmetic intensity.

C.5 Additional Analysis

Model Analysis of BiLD: FLOPs, MOPs, and Arithmetic Intensity

Figure C.2 compares average FLOPs, MOPs (memory operations), arithmetic intensity, and the latency speedup of the vanilla inference and BiLD on the CNN/DailyMail benchmarks. For BiLD, we use the model with roughly the same ROUGE-L score as the vanilla inference, and all the numbers are normalized by the numbers of the vanilla inference. The figure illustrates that BiLD exhibits slightly higher FLOPs compared to the vanilla inference. This is due to the fact that the autoregressive and non-autoregressive executions have the same



Ground Truth	And Siftables are an example of a new ecosystem of tools for manipulating digital information.
Large	And the Siftables are an example of a new generation of manipulation tools for digital data.
Small	And the if you look at the ifleses are an example of a new generation of technologies for manipulation of digital data.
BiLD (ours)	And the **Siftables** are an example of a new generation of **manipulation of** digital data.
Ground Truth	Which is great, because the Romans did not actually think that a genius was a particularly clever individual.
Large	That's great. The Romans didn't really think that a genius was a particularly smart individual.
Small	That's great. The tube didn't really think that a genius was a particularly lonely individual.
BiLD (ours)	That's great. The **Romans** didn't really think that **a** genius was a particularly smart individual.
Ground Truth	The viral particles then were released from the cells and came back and killed the E. coli.
Large	The viral particles then were released by the cells and came back and killed E. coli.
Small	The viral particles were then released by the cells and came back and killed E. Coke.
BiLD (ours)	The viral particles **then were** released **by the** cells and came back and killed E. **coli**.

Figure C.3: Example text sequences that BiLD generates with the validation set of IWSLT 2017 De-En, compared to the ground truths and the outputs of the large and small baselines. For BiLD, tokens generated by the large model are highlighted in red, while all the other tokens are generated by the small model. This illustrates that with a small engagement of the large model, BiLD can correct not only inaccurate vocabulary but also wrong semantics of the text that the small model would have otherwise generated.

amount of FLOPs, and BiLD involves additional overhead of running the small model alongside. However, in the case of MOPs, BiLD demonstrates a significant ∼5× reduction of memory operations. This can be attributed to the capability of BiLD to process multiple tokens with a single weight load, thereby enhancing token-level parallelism and maximizing data reuse. In contrast, this is not the case in the vanilla inference where a single weight load can only process a single token. Consequently, BiLD achieves a significantly higher arithmetic intensity, which is approximately 5 times larger than the vanilla inference. Arithmetic intensity [289] measures the number of arithmetic operations that can be performed per memory operation. Given that memory operations can contribute more to the overall inference latency than arithmetic operations in many Transformer decoding scenarios [133], decreasing memory operations and increasing arithmetic intensity can effectively alleviate the inference bottleneck. This leads to an overall latency speedup of 1.85× on actual hardware.

Examples of Generated Sequences

Figure C.3 provides examples of text sequences generated by BiLD on the validation set of IWSLT 2017 De-En, along with the ground truths (i.e., labels) and outputs of the pure large and small baseline models. The tokens generated from the large model of BiLD are highlighted in green, while all the other tokens are generated by the small model. The results

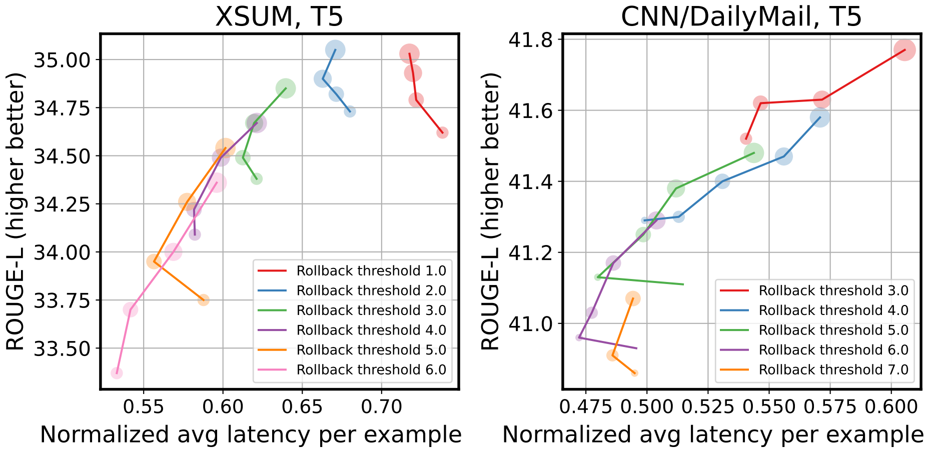


Figure C.4: The trade-off between latency and generation quality (ROUGE-L) for the aligned BiLD model on two summarization tasks: (Left) XSUM and (Right) CNN/DailyMail. Each curve represents a different rollback threshold, with smaller thresholds indicating more rollbacks. The trade-off can be further obtained within each curve with different fallback thresholds, where larger scatter sizes indicate larger fallback thresholds. A larger fallback threshold implies more fallbacks.

illustrate that the small model often produces low-quality texts, by predicting inaccurate tokens which can alter the meaning of the entire sentence. To contrast, it is observed from the examples that BiLD is able to improve the text generation quality by letting the large model interrupt when the small model generates incorrect tokens. Particularly, in the examples provided, BiLD tends to be as strong as the large model at predicting terminologies. Overall, the large model's engagement in BiLD decoding not only improves the prediction accuracy but also prevents incorrect predictions from impacting the future ones.

Impact of Fallback and Rollback on Performance

We have explored how the BiLD framework can achieve different trade-offs between latency and generation quality by adjusting fallback and rollback thresholds. In this section, we present a detailed analysis of how these thresholds affect the performance using the aligned BiLD model on two different summarization tasks, XSUM and CNN/DailyMail, as illustrated in Figure C.4. Different curves in the plot represent different rollback thresholds, and each scatter point within the curve represents different fallback thresholds. Note that a small rollback threshold implies more rollback, while a larger fallback threshold implies more fallback.

We observe a general trend where smaller rollback thresholds (i.e., more rollbacks) result in better generation quality but longer latency. This trend is expected because, with more rollback, we preempt more small model's predictions that can be potentially inaccurate by sacrificing the latency. Similarly, there is also a general trend that smaller fallback thresholds (i.e., fewer fallbacks) result in faster latency but a worse generation quality. However, we observed that lowering the fallback rates beyond a certain point can actually hurt both the latency and generation quality. This is because inaccurate predictions that the small model should have fallen back are later rolled back, incurring an extra 'flush' cost for the tokens that follow.

Appendix D

Efficiency in Agentic Applications: LLM Compiler for Parallel Function Calling

D.1 Experimental Details

Our experiments evaluate two different common scenarios: (1) using API-based closed-source models; and (2) using open-source models with an in-house serving framework. We use OpenAI's GPT models as closed-source models, in particular, gpt-3.5-turbo (1106 release) for HotpotQA and Movie Recommendation, gpt-4-turbo (1106 release) for ParallelQA, and gpt-4 (0613 release) for Game of 24. Experiments on HotpotQA, Movie Recommendation, and ParallelQA were all conducted in November 2023 after the 1106 release. The Game of 24 experiments were conducted over a two-month period from September to October 2023. For an open-source model, we use LLaMA-2 [262], which was hosted on 2 A100-80GB GPUs using the vLLM [147] framework. All the runs have been carried out with zero temperature, except for `thought_proposer` and `state_evaluator` for the Game of 24 evaluation, where the temperature is set to 0.7. Since OpenAI has randomness in outputs even with temperature 0, we have conducted 3 runs, and we reported the average accuracy. Across ReAct, OpenAI parallel function calling, and `LLMCompiler`, we perform 3, 1, and 5-shot learning for HotpotQA, Movie Recommendation, and ParallelQA, respectively; the same examples across different methods were used to ensure a fair comparison. For the Game of 24, we use 2 in-context examples for the Planner. We use the same instruction prompts across different methods for a fair comparison, except for ReAct† in Section 6.4 with additional ReAct-specific prompts. For WebShop experiment, we use gpt-4-0613 with 8k context window and gpt-3.5-turbo model with 16k context window.

D.2 Analysis

Parallel Speedup Modeling

While `LLMCompiler` shows noticeable latency gain in various workloads, it is not achieving the $N\times$ latency speedup for N-way parallel workloads. This is mostly due to the overhead associated with `LLMCompiler`'s Planner and final answering process that cannot be parallelized. In our Movie Recommendation experiment, `LLMCompiler`'s Planner and the answering process have an overhead of 1.88 and 1.62 seconds on average, respectively, whose combined overhead already comprises more than half of `LLMCompiler`'s overall latency in Table 6.1. Another source of overhead is the straggler effect among the parallel tasks when they need to join together. We observe the average latency of the slowest `search` to be 1.13 seconds, which is nearly 2× the average latency of all tasks, which is 0.61 seconds. Below, we provide an analytical latency modeling of ReAct, `LLMCompiler`, and `LLMCompiler` with streaming, and we provide an analysis of achievable latency speedup.

In this section, our focus is on *embarrassingly parallelizable* workload (pattern Figure 6.3(a)), as this allows for a clearer understanding of the impact of each component on potential latency gains. For the precise latency analysis, we consider three key components: the Planner, the Task Fetching Unit, and the Executor, in Figure 6.2. Assume that the Planner generates N different tasks to be done. We define P_i as the Planner's output corresponding to the i-th atomic task. Each P_i is a blueprint for a specific atomic task, which we refer to as E_i. The execution of E_i involves a specific function call using the appropriate tool. The latency function of each unit in the system is defined to quantify the time taken for specific operations. For the Planner, the latency is denoted as $T_P(P_i)$, representing the time taken by the Planner to generate the plan P_i. Similarly, for the Executor, the latency, $T_E(E_i)$, corresponds to the time required to complete the task E_i. We ignore the latency of Task Formulation Unit, as it is negligible in this section. Our focus here is on comparing the latency models of ReAct [307], and `LLMCompiler`.

To begin our analysis of ReAct's latency, we express its total latency as:

$$T^R = \sum_{i=1}^{N} \left(T_P^R(P_i) + T_E(E_i) \right). \tag{D.1}$$

Here, the superscript R refers to ReAct. In the ReAct agent system, the process typically involves initial thought generation, followed by action generation and the acquisition of observations through function calls associated with the tool. The creation of both thought and action are collectively considered as part of generating P_i. It is important to note that while the Planner's latency is denoted with a superscript (indicating ReAct), the Executor's latency does not have such a superscript. This is because the function calling and the tools execution remain the same between ReAct and `LLMCompiler`.

For `LLMCompiler`, where all parallelizable tasks are processed concurrently, the total latency is determined by the slowest task among these tasks. Hence, the latency model for

LLMCompiler can be represented as:

$$T^C = \sum_{i=1}^{N} T_P^C(P_i) + \max_{k \in 1,\ldots,N} T_E(E_k). \tag{D.2}$$

This expression captures the sum of all planning times plus the execution time of the longest task, reflecting the system's focus on parallel execution.

Further, if the Planner employs streaming of the dependency graph, the latency model undergoes a modification and can be expressed as:

$$T^{SC} = \sum_{i=1}^{N} T_P^C(P_i) + T_E(E_N). \tag{D.3}$$

It is important to note that $T^{SC} \leq T^C$. This implies that the streaming mechanism allows for a more efficient handling of task dependencies, potentially reducing overall latency.

In evaluating the potential speedup achievable with the LLMCompiler framework compared to ReAct, the speedup metric, denoted as γ, is defined as follows:

$$\gamma = \frac{T^R}{T^C} = \frac{\sum_{i=1}^{N} \left(T_P^R(P_i) + T_E(E_i) \right)}{\sum_{i=1}^{N} T_P^C(P_i) + \max_{k \in 1,\ldots,N} T_E(E_k)}. \tag{D.4}$$

This ratio represents the comparative efficiency of LLMCompiler over ReAct, considering both planning and execution latencies.

To estimate the upper bound of this speedup, $\gamma_{\max}$, we assume that the executor latency $T_E(E_i)$ is dominant over the planning latency $T_P(P_i)$ and all the latencies of executing tasks remain the same. Under this assumption, the upper bound is calculated as:

$$\gamma_{\max} \approx \frac{\sum_{i=1}^{N} T_E(E_i)}{\max_{k \in 1,\ldots,N} T_E(E_k)} = N, \tag{D.5}$$

indicating the theoretical maximum speedup, $\gamma_{\max}$, is equal to the number of tasks, N.

On the other hand, the lower bound of the speedup, γ, is observed when the planning latency is the predominant factor. Given that the planning latencies of both ReAct and LLMCompiler are generally similar, the minimum speedup is approximated as:

$$\gamma_{\min} \approx \frac{\sum_{i=1}^{N} T_P^R(P_i)}{\sum_{i=1}^{N} T_P^C(P_i)} \approx 1. \tag{D.6}$$

From these observations, we can conclude that to achieve significant latency gains with LLMCompiler, it is crucial to (i) reduce the planner overhead and (ii) minimize the occurrence of stragglers.

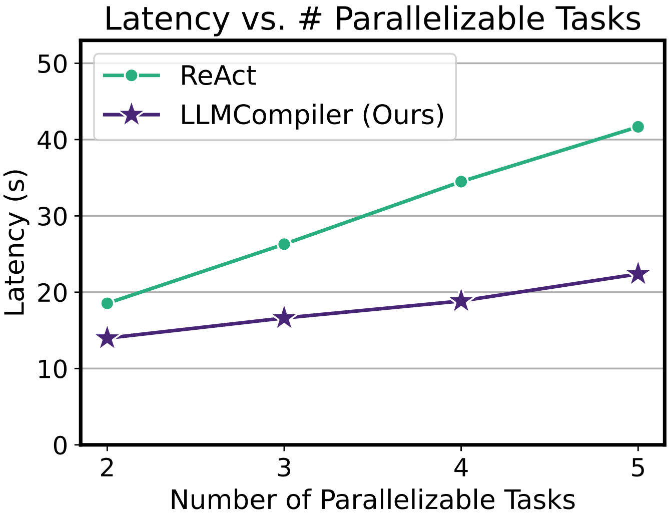


Figure D.1: Latency on the ParallelQA benchmark grouped by the number of maximum parallelizable tasks.

Latency versus Number of Parallelizable Tasks

In Figure D.1, we also report a more detailed latency breakdown on ParallelQA where we show the end-to-end latency as a function of the number of parallel tasks. This is often referred to as weak-scaling in high-performance computing, where the ideal behavior is to have a constant latency as the number of tasks is increased. We can see that ReAct's latency increases proportionally to the number of tasks, which is expected as it executes the tasks sequentially. In contrast, the latency of `LLMCompiler` increases at a much smaller rate, as it can perform multiple function calls in parallel when possible. The reason the end-to-end latency increases slightly with `LLMCompiler` is due to the overhead of the Planner, which needs to generate plans initially, and which cannot be parallelized. We provide a further analysis of this in Appendix D.2.

Latency Comparison Between Using and not Using Streamed Planner

In Table D.1, we present a latency comparison of `LLMCompiler` with and without the streaming mechanism of the Planner across different benchmarks.

Additional Experiments on the HotpotQA Bridge Benchmark

In our main experiments in Section 6.4, we used the comparison benchmark in HotpotQA to demonstrate the capability of `LLMCompiler` in efficiently executing 2-way parallelizable workloads. The other part of the benchmark, called 'bridge,' involves sequential tasks such as

Table D.1: A latency comparison between using and not using streaming in the Planner. Streaming yields consistent latency improvement across different benchmarks, as it enables the Task Fetching Unit to start task execution immediately as each task is produced by the Planner. The impact of streaming is especially notable in the ParallelQA benchmark, where tool execution times are long enough to effectively hide the Planner's execution time.

Benchmark	w/o streaming (s)	w/ streaming (s)	Latency speedup
HotpotQA	4.00	**3.95**	1.01×
Movie Rec.	5.64	**5.47**	1.03×
ParallelQA	21.72	**16.69**	1.30×

Table D.2: Accuracy and latency comparison of `LLMCompiler` compared to ReAct on the HotpotQA bridge benchmark. ReAct† denotes ReAct with additional prompting that minimizes looping and early stopping, similar to Table 6.1.

Method	Accuracy (%)	Latency (s)
ReAct	22.7	7.07
ReAct†	23.1	6.42
`LLMCompiler`	**26.3**	**4.70**

"What government position was held by the woman who portrayed Corliss Archer in the film Kiss and Tell?" `LLMCompiler` is not limited to the comparison benchmark, but it can also be applied to the bridge benchmark due to its replanning capability: initially, it searches for the woman who played Corliss Archer in the film Kiss and Tell, and then, through replanning, searches the government position held by this woman for the example above.

Similar to our experiments with the comparison benchmark, Table D.2 compares `LLMCompiler` against ReAct and ReAct with the additional prompt that avoids repetitive function calling and early stopping (ReAct†) on the bridge benchmark. We observe 4 and 3% accuracy improvement, respectively, which is attributed to ReAct's repetitive function invocation – even with the additional prompt (ReAct†), we have still observed 5% of the examples failing with this issue. Furthermore, such repetitive function call also accounts for the slightly higher latency of ReAct compared to ours. This experiment demonstrates that `LLMCompiler` allows for efficient and accurate function calling for both parallel and sequential workloads.

Table D.3: Qualitative comparison between `LLMCompiler` and other frameworks including ReAct [307], TPTU (SA for Sequential Agent and OA for One-step Agent) [224], ViperGPT [256] and HuggingGPT [237].

Method	Planning	Replanning	Parallel Execution	Domain
ReAct	X	-	X	All
TPTU-SA	X	-	X	All
TPTU-OA	O	X	X	All
ViperGPT	O	X	X	Limited
HuggingGPT	O	X	O	Limited
`LLMCompiler`	O	O	O	All

D.3 Additional Discussions about Related Works

TPTU [224], HuggingGPT [237], and ViperGPT [256] have introduced end-to-end plan-and-solve frameworks. In this section, we discuss how `LLMCompiler` distinguishes itself from other frameworks from various angles, including the capabilities in (i) planning and replanning; (ii) parallel execution; and (iii) addressing a wider range of problem domains. Refer to Table D.3 for the summary.

Parallel Execution: Parallel execution is a critical feature in the `LLMCompiler` framework that allows for efficient function calling and job completion. While the One-step Agent in TPTU (i.e., TPTU-OA) incorporates planning, it does not enable parallel function calling, as it only decomposes a user input into a sequence of functions and the associated arguments without their inter-dependencies. ViperGPT generates Python codes. However, ViperGPT, by itself, does not support parallel execution without a dedicated parallel processing engine since the standard Python interpreter lacks support for parallel execution. While HuggingGPT enables parallel execution, it strictly targets models in HuggingFace, making it hard to apply in a wide range of problems and domains that LLMCompiler supports.

Planning and Replanning: The TPTU's Sequential Agent (i.e., TPTU-SA) is an iterative framework like ReAct [307] that executes one action per iteration. While TPTU-OA, HuggingGPT, and ViperGPT are all planning-based frameworks that plan out multiple actions prior to execution, they lack replanning capabilities. `LLMCompiler`, in contrast, incorporates the replanning mechanism to generate a new set of tasks when the previous plans are not sufficient enough to deliver the response back to the user. This enables `LLMCompiler` to adapt plans based on intermediate results that are a priori unknown, without the need for introducing complex branching logic, thereby extending the scope of problems that it can address.

Problem Domains: ViperGPT and HuggingGPT aim for vision tasks via Python code generation and models in HuggingFace, respectively, showing significant promise in these specific areas. In contrast, `LLMCompiler` targets a general framework that enables efficient

Table D.4: Accuracy and latency speedup comparison of `LLMCompiler` compared to ReAct and TPTU (SA for Sequential Agent and OA for One-step Agent) on the HotpotQA comparison benchmark using gpt-3.5-turbo. ReAct† and TPTU-SA† denote ReAct and TPTU-SA with additional prompting that minimizes looping and early stopping, respectively, similar to Table 6.1.

Method	Accuracy (%)	Speedup
ReAct	61.52	-
ReAct†	62.47	1×
TPTU-SA	34.16	-
TPTU-SA†	44.59	1.09×
TPTU-OA	57.50	1.35×
`LLMCompiler`	**62.00**	**1.51×**

and accurate function calling in a wide range of problem domains, rather than restricting itself to specific fields.

Quantitative Comparison between `LLMCompiler` and TPTU

Additionally, in Table D.4, we additionally provide accuracy and latency speedup of `LLMCompiler` against TPTU-SA and TPTU-OA. Since the official implementation of TPTU is not available, we implemented TPTU-SA and TPTU-OA based on the prompts provided in the original paper. As can be seen in the table, the results clearly demonstrate `LLMCompiler`'s latency and accuracy benefit over both TPTU-SA and TPTU-OA. Compared with TPTU-SA, `LLMCompiler` exhibits a significant accuracy improvement due to TPTU's prevalent issue with repetitive function calls. Note that this issue is not fully mitigated even with better prompting (TPTU-SA†), leading to ∼15% of examples failing with repetitive function calls. Compared with both TPTU-SA and TPTU-OA, `LLMCompiler` also benefits from reduced latency through parallel task execution. Overall, the results are consistent with the main experiments and analysis against other baseline methods (i.e., ReAct and OpenAI's parallel function calling).

D.4 User-Supplied Examples for `LLMCompiler` Configuration

`LLMCompiler` provides a simple interface that allows for tailoring the framework to different use cases by providing tool definitions as well as optional in-context examples for the Planner. Below, we provide the Planner example prompts that are used to set up the framework for the Movie Recommendation and Game of 24 benchmarks with only a few lines of prompts.

Movie Recommendation Example Prompts

```
Question: Find a movie similar to Mission Impossible, The Silence of
the Lambs, American Beauty, Star Wars Episode IV - A New Hope
Options:
Austin Powers International Man of Mystery
Alesha Popvich and Tugarin the Dragon
In Cold Blood
Rosetta

1. search("Mission Impossible")
2. search("The Silence of the Lambs")
3. search("American Beauty")
4. search("Star Wars Episode IV - A New Hope")
5. search("Austin Powers International Man of Mystery")
6. search("Alesha Popvich and Tugarin the Dragon")
7. search("In Cold Blood")
8. search("Rosetta")
Thought: I can answer the question now.
9. finish()
###
```

Game of 24 Example Prompts

```
Question: "1 2 3 4", state_list: [""]
$1 = thought_proposer("1 2 3 4", "")
$2 = state_evaluator("1 2 3 4", "$1")
$3 = top_k_select("1 2 3 4", ["$1"], ["$2"])
$4 = finish()
###
Question: "1 2 3 4", state_list: ["1+2=3(left:3 3 4)","2-1=1(left:1 3
4)","3-1=2(left:2 2 4)","4-1=3(left:2 3 3)","2*1=2(left:2 3 4)"]
$1 = thought_proposer("1 2 3 4", "1+2=3(left:3 3 4)")
$2 = thought_proposer("1 2 3 4", "2-1=1(left:1 3 4)")
$3 = thought_proposer("1 2 3 4", "3-1=2(left:2 2 4)")
$4 = thought_proposer("1 2 3 4", "4-1=3(left:2 3 3)")
$5 = thought_proposer("1 2 3 4", "2*1=2(left:2 3 4)")
$6 = state_evaluator("1 2 3 4", "$1")
$7 = state_evaluator("1 2 3 4", "$2")
$8 = state_evaluator("1 2 3 4", "$3")
```

```
$9 = state_evaluator("1 2 3 4", "$4")
$10 = state_evaluator("1 2 3 4", "$5")
$11 = top_k_select("1 2 3 4", ["$1", "$2", "$3", "$4", "$5"], ["$6",
"$7", "$8", "$9", "$10"])
$12 = finish()
###
```

D.5 Pre-defined `LLMCompiler` Planner Prompts

The pre-defined `LLMCompiler` Planner prompt provides it with specific instructions on how to break down tasks and generate dependency graphs while ensuring that the associated syntax is formatted correctly. This prompt contains specific rules such as assigning each task to a new line, beginning each task with a numerical identifier, and using the `$` sign to denote intermediate variables.

```
- Each action described above contains input/output types and
descriptions.
- You must strictly adhere to the input and output types for each
action.
- The action descriptions contain the guidelines. You MUST strictly
follow those guidelines when you use the actions.
- Each action in the plan should strictly be one of the above types.
Follow the Python conventions for each action.
- Each action MUST have a unique ID, which is strictly increasing.
- Inputs for actions can either be constants or outputs from preceding
actions. In the latter case, use the format $id to denote the ID of
the previous action whose output will be the input.
- Ensure the plan maximizes parallelizability.
- Only use the provided action types. If a query cannot be addressed
using these, invoke the finish action for the next steps.
- Never explain the plan with comments (e.g. #).
- Never introduce new actions other than the ones provided.
```

In addition to user-provided functions, the Planner includes a special, hard-coded `finish` function. The Planner uses this function either when the plan is sufficient to address the user query or when it can no longer proceed with planning before executing the current plan, i.e., when it deems replanning necessary. When the Planner outputs the `finish` function, its plan generation stops. Refer to Appendix D.4 for examples of the Planner's usage of the `finish` function in planning. The definition of the `finish` function is as below and is included as a prompt to the Planner along with the definitions of other user-provided functions.

```
finish():
 - Collects and combines results from prior actions.
 - A LLM agent is called upon invoking join to either finalize the user
query or wait until the plans are executed.
 - join should always be the last action in the plan, and will be
called in two scenarios:
 (a) if the answer can be determined by gathering the outputs from
tasks to generate the final response.
 (b) if the answer cannot be determined in the planning phase before
you execute the plans.
```

D.6 ParallelQA Benchmark Generation

Inspired by the IfQA benchmark [315], our custom benchmark ParallelQA contains 113 examples that are designed to use mathematical questions on factual details of different entities to answer questions, thus requiring a mix of search and mathematical operations that are interdependent in various ways. For instance, the benchmark includes examples like "If Texas and Florida were to merge and become one state, as well as California and Michigan, what would be the largest population density among these 2 new states?" requires four parallel search tasks, followed by math tasks dependent on the search outcomes, that can be executed in parallel.

The main objective of the benchmark is to quantify the framework's ability to decompose an input into multiple tasks to derive an answer. Therefore, we have meticulously selected 56 distinct entities across various domains whose attributes can be accessible from Wikipedia search. By minimizing tool execution (i.e., Wikipedia search) failures, we have aimed our benchmark to effectively assess the frameworks' abilities to decompose questions into multiple tasks, plan them out, and derive final answers based on observations. Furthermore, to incorporate diverse execution patterns, we crafted various dependency patterns that perform unary and binary math operations after searching for additional information about entities in a given question. We have also curated different questions that accommodate different numbers of maximally parallelizable tasks, ranging from 2 to 5, and we have included varying numbers of joins between parallel function calls as well to increase problem complexity. For instance, we have 2 and 3 joins in Figure 6.3 (b) and (c), respectively. The benchmark contains 113 different examples, that were populated by GPT-4 based on the aforementioned criteria and labeled by humans afterward.

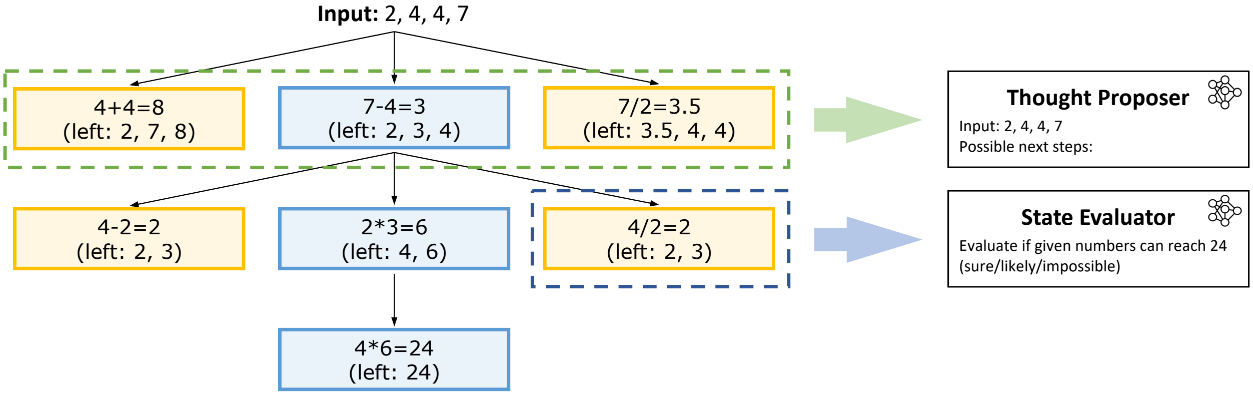


Figure D.2: Visualization of the Tree of Thoughts (ToT) in the Game of 24. Each node represents a distinct proposal, beginning with the root node and branching out through the application of single operations by the thought proposer. Subsequent states are evaluated by the state evaluator for their potential to reach the target number 24. The ToT retains the top-5 states according to their values.

D.7 Details of the Game of 24 and the Tree-of-Thoughts Approach

The Game of 24 is a mathematical reasoning game that challenges players to manipulate a given set of four numbers, using the basic arithmetic operations of addition, subtraction, multiplication, and division, to arrive at the number 24. The rule of this game is that the given numbers must be used only once. For instance, given the numbers 2, 4, 4, and 7, one possible solution is $4 \times (7-4) \times 2 = 24$. This is a non-trivial reasoning benchmark for LLMs, highlighted by the fact that even advanced models like GPT-4 exhibit only a 4% success rate, even when using chain-of-thought prompting [308].

In ToT, the problem is solved in several steps. At each step, the LLM, referred to as the thought proposer, generates thoughts. Each thought is a partial solution that consists of two numbers and an arithmetic operation between them. Then, these thoughts are fed into the state evaluator which assigns a label for each of them. These labels are ‘sure,’ ‘likely,’ and ‘impossible,’ which are given to thoughts to denote how likely they could produce 24 with additional arithmetic operations between the result and the remaining numbers. Only the thoughts that are likely to produce 24 continue onto the next step. This process is illustrated in Figure D.2.

D.8 Details of WebShop Experiments

WebShop Environment

The WebShop environment simulates an online shopping platform. Tasks are designed for the agent to find the item that best matches the given instruction. For instance, if the instruction specifies, “I am looking for a queen-sized bed that is black, and priced lower than 140.00 dollars,” the agent’s task is to pinpoint the bed that precisely fits these criteria: “queen-sized,” “black,” and “priced under 140.00 dollars.” For each item, there is an associated reward measuring how well this item matches the instruction based on price, item options, and other details contained in the item page. The evaluation metrics are the success rate—the proportion of episodes where the selected product satisfies all requirements—and the average score—the mean reward across episodes.

Baseline Methods

In addition to ReAct, we use LASER [176] and LATS [334] as baseline methods to compare against `LLMCompiler`. LASER [176] solves tasks through a state-exploration approach. In the context of WebShop, the possible environment pages are encoded as different states (e.g., search page, item page, and item detail subpage). Actions are used to transition between these states, such as executing a search query, selecting an item, checking the item detail, navigating the next search page and so on. The Webshop exploration is therefore reduced to a search problem on the given state-space graph.

Using a variant of Monte Carlo Tree Search, LATS [334] plans its actions by constructing a decision tree, evaluating potential moves based on their likelihood of success, and selecting actions through a balance of exploration and exploitation. The agent then adapts its strategy based on feedback from the environment, learning from both successes and failures to refine its decision-making process. This iterative approach allows LATS to navigate complex online shopping tasks, albeit much more slowly due to its exhaustive tree search.