Machine Learning Systems with Reduced Memory Requirements

Franklin Huang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-120
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-120.html

May 17, 2024

Copyright © 2024, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I'd like to acknowledge many people who have continuously assisted me in my short year to complete this thesis, without them none of this would happen. First and foremost my research advisor Professor Nikolic, second reader Professor Shao, and also PhD candidate Coleman Hooper for helping to scope achievable topics. Additionally, staff and Professors on CS 252A, EECS 251B, and EE C249A helped me discover the complexities of how algorithms interact with every level of hardware. Most importantly, Tapeout and Bringup members of BearlyML22 \& 23 for designing a homegrown chip that can run a small language model for the first time. Finally, all members of SLICE and BWRC lab whom we relied on consultation of Chipyard and PCB analog magic matters.

Machine Learning Systems with Reduced Memory Requirements

by

Hongyi (Franklin) Huang

Research Project submitted in partial satisfaction of the

requirements for the degree of

Masters of Science, Plan II

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Borivoje Nikolic, Research Advisor
Professor Sophia Shao, Second Reader

Spring 2024

The Research Project of Hongyi (Franklin) Huang, titled Machine Learning Systems with Reduced Memory Requirements, is approved:

Research Advisor ________________ Date May 16, 2024.

Second Reader Sophia Shao Date 05/17/2024

University of California, Berkeley

Machine Learning Systems with Reduced Memory Requirements

Copyright 2024
by
Hongyi (Franklin) Huang

Abstract

Machine Learning Systems with Reduced Memory Requirements

by

Hongyi (Franklin) Huang

Masters of Science, Plan II in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Borivoje Nikolic, Research Advisor

Machine learning systems today are developing in two opposites that require an increasing amount of hardware awareness to make productization feasible. On one hand, a stable scaling law in large language lodels (LLMs) pushes the system scale to be bigger; on the other hand, robotics and wearable applications require neural networks to fit in small systems that have an extremely low amount of compute, memory, and power budget.

Even though Moore's Law and architectural innovations have been sustaining compute performance growth, there is a widening processor-memory gap that requires imminent innovation. While anticipated hardware advancements such as GDDR7, HBM4, and UCIe are expected to alleviate this gap, challenges in the memory hierarchy will persist. Therefore, it is crucial to design kernels that enhance inference throughput by efficiently utilizing and managing the memory hierarchy.

This technical report explores improvements in the compression of a broad range of language models and compares them to the state-of-the-art. While quantization benefits all models, this thesis finds that sparsity and entropy methods are particularly effective for smaller models, reducing the bitrate to as low as 1.96 bits per weight with minimal accuracy loss. In contrast, larger language models derive greater advantages from enhanced data reuse and page-based memory management techniques.

Specifically, in CodeGen applications where parallel sampling enhances accuracy, these strategies have demonstrated the potential to reduce the memory capacity and bandwidth requirements of attention kernels by 15x. When evaluating problem-solving capacity, parallel sampling effectively matches the capabilities of a single sampled larger model with a tenfold reduction in memory and parameter count. These achievements unlock the possibility of local deployment for both real-time embedded systems and language model applications.

To my family mostly full of engineers,

And to those who share my passion for making technology more human.

Contents

Contents **ii**

List of Figures **iii**

List of Tables **iv**

1 On-Device TinyML **1**
- 1.1 Quantization Basics . . . 1
- 1.2 Memory Traffic & Runtime Analysis . . . 3
- 1.3 Single Batch Small Language Model Inference . . . 4

2 Entropy-Based Lossless Compression **8**
- 2.1 Overview . . . 8
- 2.2 Fundamentals of Asymmetrical Numerical Systems . . . 9
- 2.3 Compression Rate on Various Models . . . 11
- 2.4 Accelerating Decompression Software . . . 11
- 2.5 Difficulties of Entropy Compression in Large Language Models . . . 12
- 2.6 Related and Concurrent Works . . . 14

3 CodeGen: Parallelism in Language Models **16**
- 3.1 Introduction . . . 16
- 3.2 Memory Traffic of Large Language Models Inference . . . 17
- 3.3 Outlier Migration through Group Quantization . . . 17
- 3.4 Application Specific Parallelism . . . 18
- 3.5 KV-Cache Compression: Shared Prompt & PagedAttention . . . 22

4 Conclusion **28**

Bibliography **29**

List of Figures

1.1 Left: Sparse compute kernel. Right: Relufied Llama architecture. 5

2.1 rANS Decompression and Threading Illustration. 12
2.2 Weight distribution in Llama 2 7B. 13

3.1 Group-wise quantization vs. tensor-wise quantization. 18
3.2 Custom fused kernel performances compared to naive PyTorch implementation, benchmarked on RTX 4080. 19
3.3 KV-cache and inference batch size, comparing naive, shared prompt, and paging. 20
3.4 Weight can achieve memory reuse through high batch, but KV-cache traditionally does not. 21
3.5 Blue: Tokens/s produced, Orange: Tokens/s used in the program (eliminating special tokens and trajectories that ended early). 22
3.6 Llama-7B on MBPP, 0-shot prompt pass rates @1, @10, @100 with varying temperatures, top-k=32. The annealing schedule is defined below. For data see Tables 3.1 and 3.2. 23
3.7 Average batch size over time: on average, memory and compute reduction equals the area above the orange line till y=100. 24
3.8 ChunkedAttention (blue), shared prompt only comparing against FlashAttention (green). 26
3.9 ChunkAttention (with non-shared KV-Cache), tested with half shared-prompt and half divergent-context. TreeAttention is tested with an extra branch of 1 per time step along with half shared-prompt. 26

List of Tables

1.1 Common fixed point storage format for quantized numbers. 1
1.2 Density of activation across different models. 7
1.3 Non-zero activation % across consecutive tokens for llama 260k (TinyStories). . 7

2.1 Compression ratio & effective bitrate per weight. 11
2.2 Single-core speed benchmark and overhead relative to reading in quantized bits from Flash or DRAM. On M1 silicon, a state-of-the-art 1.5-1.8 Giga int4 Symbols/s is achieved per core. Alternatives including branchless programming and CPU vectorization were attempted but did not yield speedups. 14

3.1 Pass rates of CodeLlama 7B on MBPP. 24
3.2 Avg seconds per problem for CodeLlama 7B on MBPP. 24

Acknowledgments

I'd like to acknowledge many people who have continuously assisted me in my short year to complete this thesis, without them none of this would happen. First and foremost my research advisor Professor Nikolic, second reader Professor Shao, and also PhD candidate Coleman Hooper for helping to scope achievable topics. Additionally, staff and Professors on CS 252A, EECS 251B, and EE C249A helped me discover the complexities of how algorithms interact with every level of hardware. Most importantly, Tapeout and Bringup members of BearlyML22 & 23 for designing a homegrown chip that can run a small language model for the first time. Also, Gert Christen and members of TensorZipper whom have found problems and challenges in making a startup on embedded system machine learning, even though we eventually realized the product would not be very scalable and pivoted. Finally, all members of SLICE and BWRC lab whom we relied on consultation of Chipyard and PCB analog magic matters. Special shoutout to Yufeng Chi, Edison Wang, and Richard Yan for the 'Emeryville Satellite Campus' of SLICE.

Chapter 1

On-Device TinyML

TinyML fits small machine learning systems into mobile or even embedded systems for on-device real-time inference. These are often highly quantized models with negligible training costs but need to run in tight deadlines or power budgets. The primary difficulty of these systems lies in writing kernels that compute efficiently while minimizing off-chip bandwidth.

This chapter discusses basic quantization methods in ML and how more advanced optimizations can achieve speedups in mobile and embedded systems.

1.1 Quantization Basics

Quantization reduces a floating point number down to a fixed point with a shared scale value across many numbers to reduce precision at the cost of rounding errors. Common storage formats are in Table 1.1, with higher precision 1:7:8 (int16) and 1:3:4 (int8) often used for intermediate activation layers, even effective doing PID control. 1:3:0 (int4) and 1:7:0 (int8) are often effective for storing weight. Previous works establish fused kernel formulas and quantization down to 4 bits using shared tensor-wise scaling factors [1, 2, 3].

Doing arithmetic on various formats of $C = A * B$ involves multiplying the $A * B$ into a higher precision, then saturating the max & min of integer to prevent overflow, and restoring fraction bits to C with proper rounding mechanism. For example, multiplying two int16 numbers that represent 1:7:8 would be int16 * int16 = int16(int32 $\gg$ 8); which has a +/-128 range, and precision of ≈ 0.004. An addition operator between the same format would only require saturation. Common rounding mechanisms include round-to-even and round-to-ceil when the fraction bit is exactly 0.5, while the former is statistically more stable and

Table 1.1: Common fixed point storage format for quantized numbers.

int16	1:7:8	[1] Sign	[7] Integer	[8] Fraction
int8	1:3:4	[1] Sign	[3] Integer	[4] Fraction
int4	1:3:0	[1] Sign	[3] Integer	[0] Fraction

implemented in PyTorch, round-to-even is commonly supported on all embedded hardware and faster.

1.1.1 Quantization-Aware Training

To minimize accuracy loss when quantizing models, Jacob et al [1] demonstrate quantization-aware training (QAT) of neural networks by inserting back-to-back quantization and dequantization layer. This method establishes a way to uniformly quantize a neural net without accuracy loss as its effects are already accounted for during training.

1.1.2 Uniform Scalar or Affine Quantization

To round weight or activation, scaling factor S and zero point Z are obtained, then we quantize and dequantize according to the following function Q. The saturation function takes in additional A and B, which are max and min representations of rounded numerical precision to prevent overflow or underflow. In practice, $Z = 0$ or scalar quantization simplifies the fused arithmetic step greatly so it is used more often, affine quantization uses a non-zero offset but requires more complicated fused kernel arithmetic, see the same Quantization-Aware Training paper [1] for fused kernel affine quantization.

$$Z = [max(X) - min(X)]/2$$

$$S = max(max(X - Z), -min(X - Z))$$

$$Q(x, s, z) = saturate(x/s + z, A, B)$$

$$Q^{-1}(x, s, z) = round(s(x - z))$$

Note that the scaling factor and zeros here are determined as a whole for the entire matrix or tensor. This will only work for small models, larger models as explained in Chapter 3 will require quantization every 64 groups or so to deal with outliers.

1.1.3 Fusing Dequantization and Arithmetic

A simple method that achieves speedup for any model is to fuse the arithmetic and dequantization kernels. Only with fused kernels can hardware utilize near theoretical peak FLOPS. Otherwise, there will be up to 3x more memory traffic in cache or DRAM that bottlenecks everything: dequantization, intermediate dot product result, and final activation function.

To avoid the need for dequantization, weight and activations are multiplied entirely in fixed-point and shifted as described previously. Scaling activation directly to the next layer's scaling factor is tricky to do quickly as it normally involves division arithmetic. Fortunately,

the same quantization-aware training paper [1] finds that M can be expressed into a simple fixed point and shift.

$$M = \frac{S_W S_{A_l}}{S_{A_{l+1}}} = 2^{-n} M_0 = M_0 \gg n$$

This insight allows for a kernel fusion that quickly reduces the dot product down to the next layer in one pass of memory with nothing but fixed point multiplier, shifters, and adders. Which are commonly accessible and efficient in any ISA or architecture.

$$A_{l+1}^{(i,k)} = ReLU6([M_0 * \sum_{j=1}^{N} W_l^{(i,j)} A_l^{(j,k)}] \gg n)$$

Note that since everything is now in fixed-point, using activation function ReLU6 in the model (a ReLU function with a ceil of 6) prevents activation quantization error caused by overflow and reduces accuracy degradation when porting to a fully quantized kernel. ReLU6 is also noted to be helpful in previous fixed-point quantization works [2], and experimentally, int4 and int8 weights achieve $< 1\%$ accuracy loss on MNIST.

1.2 Memory Traffic & Runtime Analysis

Two separate projects below implemented a fully quantized feed-forward neural network using all the above-optimized logic. Both of these are small enough to fit on-chip scratchpad when quantized, demonstrating the effectiveness of fixed-point fused kernel in embedded systems.

One of the chips, BearlyML 2022 [1], is an embedded machine-learning SoC in Intel 16 finFET. It is equipped with a custom sparse-dense inference accelerator attached to Chipyard's four rocket cores, 512 KB L2 cache, 16 KB scratchpad, all connected by a unidirectional ring network-on-chip bus. A downscaled MNIST network of (196x32x10) was run on BearlyML 2022, achieving a 12x speedup by using the above method along the sparse-dense accelerator. See according optimizations performance on Figure 1.1(a). Note that half of the time was still spent on the activation function and rescaling after using the sparse-dense accelerator.

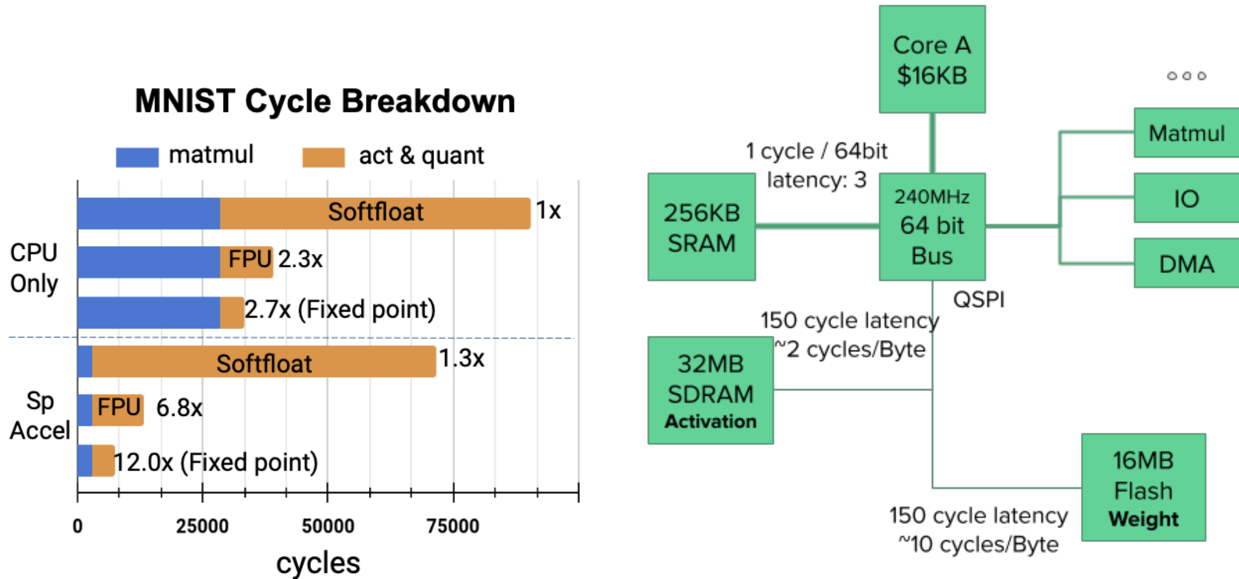
The other project ran a neural net of size 8x32x32x2 simulating a PID controller on atmega32u4 [2] to balance a two-wheeled balboa bot. This small network ran 266 Hz on the simple chip, well meeting the real-time deadline requirement of 100 Hz.

The memory traffic pattern differs if the neural net cannot fit on an on-chip scratchpad. In this case, the program would be forced to repeatedly read weights from a QSPI flash, making memory access the primary bottleneck and running the neural net at an unbearably slow rate. For most in-order CPUs, only 10% of the time is spent computing. This problem becomes much more exaggerated on ESP32S3, which has two vector cores and a peak of

[1] Part of the EECS 194/290C class effort to design and tape-out a custom SoC.

[2] For an EECS C249A project.

7 GFLOPS when coded optimally, but all of this depends on a tiny QSPI RAM and flash for memory access. While switching to convolution would turn the problem into compute-bound, many models based on attention or diffusion still rely entirely on feed-forward layers. Hence the motivation for Section 1.3 and Chapter 2 is to find a way to reduce memory bandwidth incurred by reading weights.



(a) BearlyML 2022 Cycle Benchmarks.

(b) Typical Embedded System Architecture.

1.3 Single Batch Small Language Model Inference

A single batch language model inference is fundamentally limited by the bandwidth of weight reads as the process is entirely autoregressive. One would need to either increase the batch size or decrease the bandwidth of weight reads to obtain an inference speedup. Recent orthogonal SOTA methods include:

- Speculative decoding to increase batch size by using a small model to look ahead and guess.
- Activation sparsity induction by using ReLU activations in the perceptron layers, which requires network fine-tuning.

As language models only predict similar outputs when parameter counts are sufficiently large, here we explore activation sparsity induction to double throughput effectively by ignoring weight reads when activations are zeroed.

1.3.1 Dynamic Activation Sparsity

Recent papers on LLM in a Flash [4] and ReLU Strikes Back [5] fine-tuned transformer models to use ReLU(x-1) instead of SiLU to induce activation sparsity. This is done by training a language model with SiLU, then replacing the activation function into various forms of ReLU for sparsity and re-training it using less than 10% of the original tokens to recover accuracy.

As a demonstration, a llama and OPT architecture [6] based story model was trained on TinyStories dataset [7]. We were able to achieve a 50% speed up on the llama variant compared to non-sparse activation on both Apple Silicon M1 and a newer class tapeout chip BearlyML 2023. The kernel fuses quantization methods outlined in Section 1.1, while adding an if statement to skip if activation is zero. The kernel and memory packing of weights is illustrated in Figure 1.1. [3]

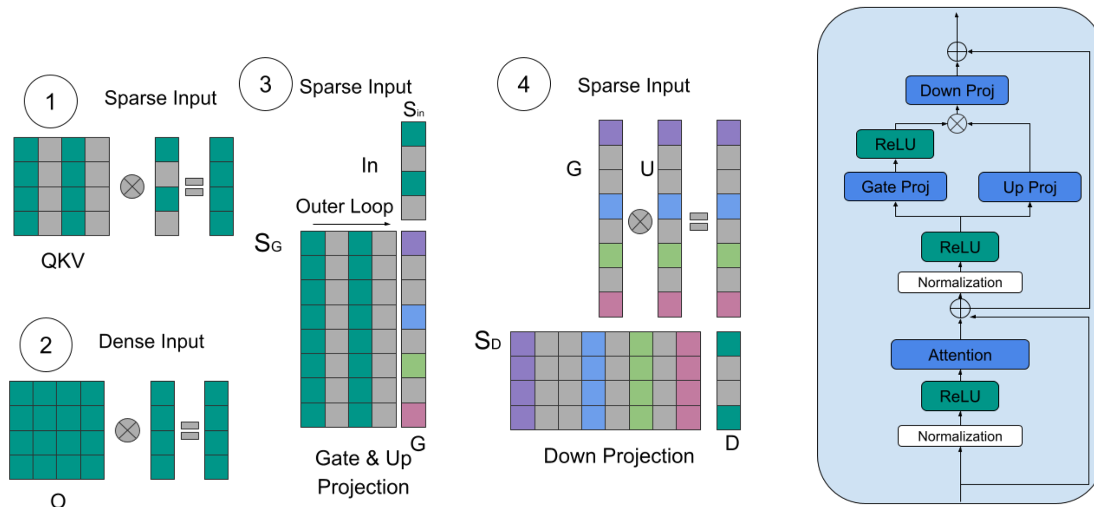


Figure 1.1: Left: Sparse compute kernel. Right: Relufied Llama architecture.

1.3.2 Activation Sparsity Challenges at Larger Scale

When we attempt to implement this on a larger scale, the drawback of needing to fine-tune appears: even for a 7B model, it would require $\approx 1-10\%$ of the original training compute to recover accuracy [5]. Specifically for Llama 2 7B, requires 2.5-25 months of A100 [6] hours to fine-tune.

[3]The sparse story models are open-sourced at https://github.com/hongyihuang/llama2.c

From experiments by attempting to fine-tune the Llama 2 7B model and also retrain/fine-tune a Llama 260k model, the following non-trivial drawbacks were observed which raises concerns:

- For Llama 2 7B model fine-tuning:
  - Convergence stagnates within 30-100 iterations when utilizing LoRA [8], only full rank training works which is extremely memory intensive. Possible recent performance-efficient fine-tuning mitigations include using ReLoRA [9] or GaLore [10].
  - As batch size increases, aggregate sparsity across batches reduces, eventually leading to reading in $\approx 73\%$ of weights when $B = 64$ (See Table 1.3). The only utility of the sparsity-induced fine-tuned model is single batch inference, which may not justify the fine-tuning cost.
  - Fine-tuning the base model to induce sparsity requires the recipe for the dataset and must be drawn from that distribution to avoid knowledge collapse. There are also no known procedures for how to select a dataset if the model we are fine-tuning is not a base model, but also a fine-tuned model of a base model such as CodeLlama.
- For Llama 260k tinystories model, with density % on Table 1.2 and 1.3:
  - As fine-tuning iterations increase, the sparse activation slowly becomes denser and hits the density levels outlined in tables
  - Training from scratch with a $relu(x-1)$ achieves the same sparsity levels as $relu$. We hypothesize that during full-rank training the weights learn to mitigate the offset and activate more to achieve a higher accuracy.
  - Given the first two points, there is a concern that as fine-tuning iterations continue to increase, density levels may also continue to increase. This is most likely an architecture problem of Llama's simultaneous gate and up projection, which allows gradients from up and gate projection to pass smoothly. While this increases overall accuracy, it does reduce the sparsity effects of ReLU. OPT only has a single-up projection followed by ReLU, it demonstrates reliable sparsity levels without spiking significantly during retraining.

Table 1.2: Density of activation across different models.

Model Type	Llama 260k (TinyStories)			Relu strikes back Llama 7B		OPT 220k
Act Type	relu	relu(x-.25)	relu(x-1)	relu	relu(x-1)	relu
D Proj	46%	23%	0%	35%	3%	32.15%
QKV	60%	27%	25%	49%	?	49.77%
FFN	63%	32%	12%	33%	?	52.01%
Val Loss	2.155	2.079	2.383	-	-	2.083
Train Loss	2.158	2.080	2.382	-	-	2.085

Table 1.3: Non-zero activation % across consecutive tokens for llama 260k (TinyStories).

Window Size	Active %
1	23%
2	36%
64	73%

Chapter 2

Entropy-Based Lossless Compression

Experiments show that lossless compression algorithm could be applied to these small models to reduce up to 51% of the original storage in int4, achieving an effective int2 bitrate. There are caveats though, including:

- Lossless compression is only useful for small models and is harder to utilize for large models that have occasional outliers.
- It is hard to achieve high throughput decompression without occupying useful hardware units.

2.1 Overview

Quantization is a lossy way to compress bits by rounding. Its average error can be determined by $avg(abs(X - Q(X)))$. Lossless compression in comparison takes advantage of the uneven statistical distribution of quantized symbols to further reduce bits per symbol, its theoretical limit is the entropy of symbols [11]. Given $s \in S$ symbols, the average bits per symbol in a distribution is equivalent to $\sum_s^S P(s)log_2(1/P(s))$, where $P(s)$ is the probability of symbol occurrence.

Huffman [12], Arithmetic Coding (AC) [13], and Asymmetrical Numerical Systems (ANS) [14] are three of such lossless compression algorithms, essentially using more bits to represent rare symbols and fewer bits to represent frequent symbols to achieve compression in aggregate. The more skewed the distributions are, the better the compression rate. Huffman is widely used in a variety of compression algorithms as it is simple to decompress and is highly effective when probabilities are extremely skewed. AC and ANS both achieve near Shannon optimal compression ratio by packing fractional bit information into a streaming range of numbers, thereby improving over Huffman. AC however, is computationally intense and slow beyond binary symbols; while tANS, when converted to a simple lookup table achieve fractional bit packing at a throughput similar to that of Huffman.

Note that even though entropy-based compression algorithms are lossless by themselves, it is always applied *after quantization* to further squeeze the bitrate of compressed objects. Hence, entropy-based methods are not useful by themselves but are used in conjunction with quantization to put the accuracy-bitrate tradeoff to a near Shannon optimal level.

2.2 Fundamentals of Asymmetrical Numerical Systems

The basic encoding formula of rANS, as in the original paper [14] is as follows:

$$X_t = Encoder_{rANS}(X_{t-1}, s_t) = \lfloor \frac{X_{t-1}}{F_{s_t}} \rfloor * M + C_{s_t} + mod(X_{t-1}, F_{s_t}) \tag{2.1}$$

where M is the total count of symbols, often chosen to be 2^N to make multiplication/division arithmetic as simple as a shift; F is the frequency or probability distribution function (PDF) of symbols; and C is the cumulative frequency or cumulative distribution function (CDF) of symbols.

Decoding formula outputs symbol s_t and state X_{t-1} given state X_t. Where the inverse of cumulative frequency is $C_{inv}(y) = i$ if $C_i \leq y < C_{i+1}$:

$$s_t = C_{inv}(mod(X_t, M)) \tag{2.2}$$

$$X_{t-1} = \lfloor \frac{X_t}{M} \rfloor * F_{s_t} + mod(X_t, M) - C_{s_t} \tag{2.3}$$

Note that the output of symbols is the reverse sequence of the encoder, hence during implementation, it is practical to reverse the encoding sequence and output bitstream whenever possible, or reverse the decoding sequence and bitstream ingestion process. Both overheads are minimal, but this work has chosen to reverse both during encoding to make decoding less convoluted to optimize.

Additionally, note F and C are chosen to be large enough to approximate the distribution, which is often 4-8 bits larger than $log_2(M)$. In this work, 8 bits of F, C, and C_{inv} were chosen for 4 bits of symbol s, which is enough for machine learning quantization.

2.2.1 Streaming rANS

While the math works out above, X grows to a large number. A more compact way of streaming out chunks of information (here in bytes) is simpler to program and enables table lookup. The original ANS [14] paper establishes that as long as an intermediate bitstream of X, represented as I is in the range of $[lM, 2lM - 1]$ for any choice of integer l, we can extract at least 1 symbol from the bitstream. If there isn't enough information in I, we simply add more bytes to restore it to range, then decompress more. The encoding and decoding algorithm is listed at 1 and 2.

Algorithm 1 rANS Streaming Compression Algorithm, int4 symbols, int8 PDF/CDF

```
1: procedure STREAMING ENCODE rANS(buf, data, buf_size, tables)
2:     Point to tables: inv_f, PDF, CDF each of size 2^8, 2^4, 2^4 respectively
3:     state ← 256
4:     j ← 0
5:     for i = 0 to buf_size do
6:         symbol ← data[size − 1 − i]                    ▷ Reverse sequence
7:         while state >= (PDF[symbol] ≪ 8) do
8:             data[j + +] ← state&0xFF
9:             state ← state ≫ 8
10:        end while
11:        state ← ((state/PDF[symbol]) ≪ 8) + (state%PDF[symbol])
12:        state ← state + (CDF[symbol] ≫ 8)
13:    end for
        return reverse(buf)
14: end procedure
```

Algorithm 2 rANS Streaming Decompression Algorithm, int4 symbols, int8 PDF/CDF

```
1: procedure STREAMING DECODE rANS(data, buf, buf_size, tables)
2:     Find pointers embedded in tables: inv_f, PDF, CDF each of size 2^8, 2^4, 2^4 respectively
3:     for i = 0 to buf_size do
4:         slot ← state&0xFF                              ▷ Mask the lowest byte
5:         symbol ← (int8_t)inv_f[slot]
6:         buf[i] ← symbol − 8               ▷ Re-center and export decompressed byte
7:         state ← (state ≫ 8) × PDF[symbol] + slot − CDF[symbol]
8:         if j < data_size and state < 256 then
9:             state ← (state ≪ 8) + data[j + +]
10:        end if
11:    end for
        return buf
12: end procedure
```

Table 2.1: Compression ratio & effective bitrate per weight.

Model/Data	MNIST	CIFAR-10
Linear (8b)	72% 5.76b	82% 6.56b
Linear (4b)	62% 2.48b	72% 2.88b
Conv (8b)	-	91% 7.28b
Conv (4b)	-	49% 1.96b

2.3 Compression Rate on Various Models

Table 2.1 demonstrates various models' average bits per weight after compression when trained with Quantization Aware Training (QAT). There is no accuracy loss compared to the original quantized as entropy compression is lossless. Floating point vs int8 vs int4 quantization observed less than 1% accuracy loss as QAT was used.

2.4 Accelerating Decompression Software

2.4.1 Achieving Parallelism

To achieve parallelism, one must be able to index into a variable length code. A common trick is to add checkpoint states and positions as illustrated in Figure 2.1.

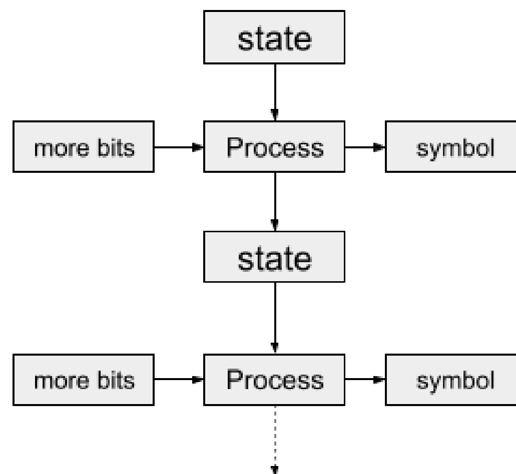
2.4.2 Tablulated ANS (tANS)

The highest throughput achievable through software is the tabulated version. Many things can be stored to accelerate the above computation, the minimal table needed would be storing C_{inv}, PDF, and CDF (total of $\tilde{0}$.3KB) as in rANS, which would only need minimal computation when realized in hardware logic directly. To completely bypass any arithmetic, simply store the tables for $X_t => X_{t-1}, s_t$ with the smallest range l for a reasonable size of table. In this case X_t has size $uint16_t$ and would take 2B*64K = 128KB, and $s_t = inv_f[slot]$ would only be 256B.

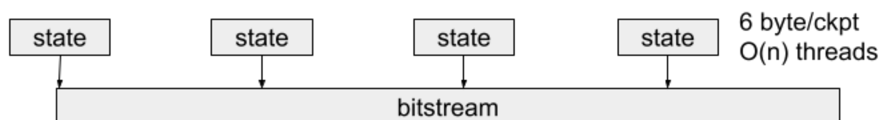
Part a) of figure 2.1 illustrates this process. Additionally, table 2.2 shows the speed and memory footprint of various software optimizations implemented [1].

Despite the 2^4 symbols decompression software kernel is still too slow to achieve any real wall-clock gains, Shannon optimal entropy-based methods can fit 18−51% larger models onto embedded or mobile systems with 2.6x or 5x runtime overhead, making this only potentially useful for accelerating loading from flash by heavily threading the decompression to many cores. To achieve real wall clock gains without using cores that can be doing matmul, a dedicated hardware decompression unit must be implemented, or much fewer symbols (2-4)

[1] See https://github.com/hongyihuang/tensorzipper/tree/main/C for code



(a) rANS Decompression Process.



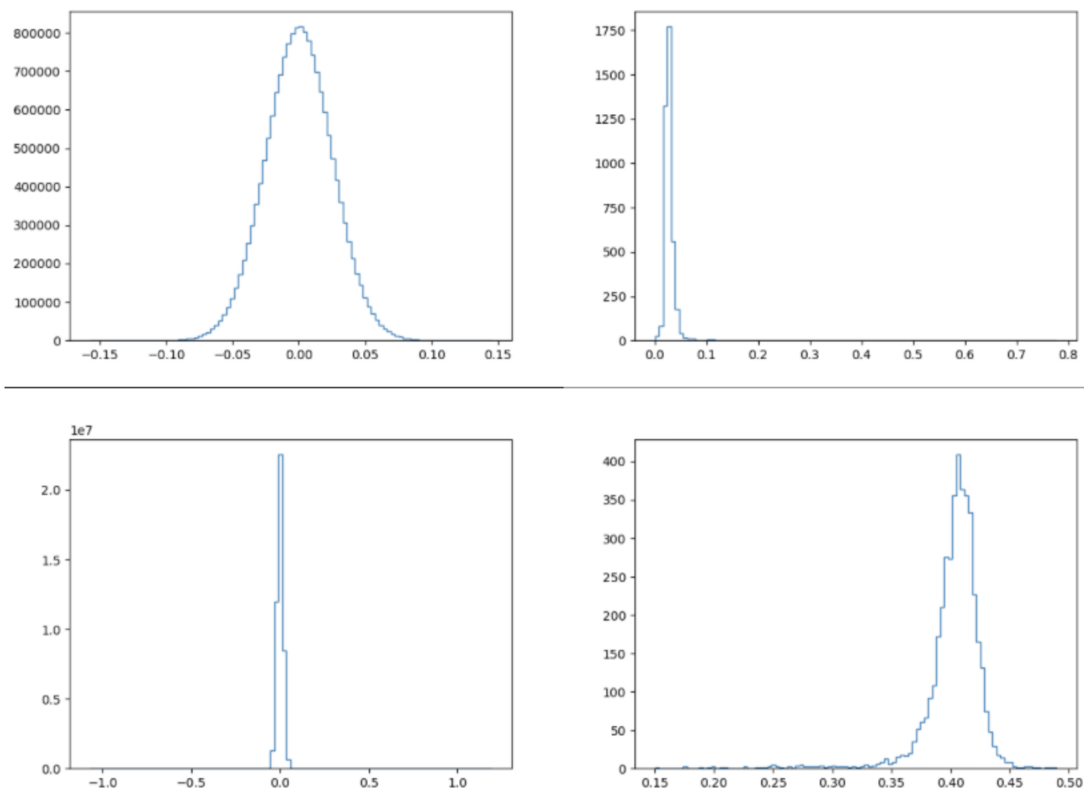
(b) Threading by adding checkpoint states and position.

Figure 2.1: rANS Decompression and Threading Illustration.

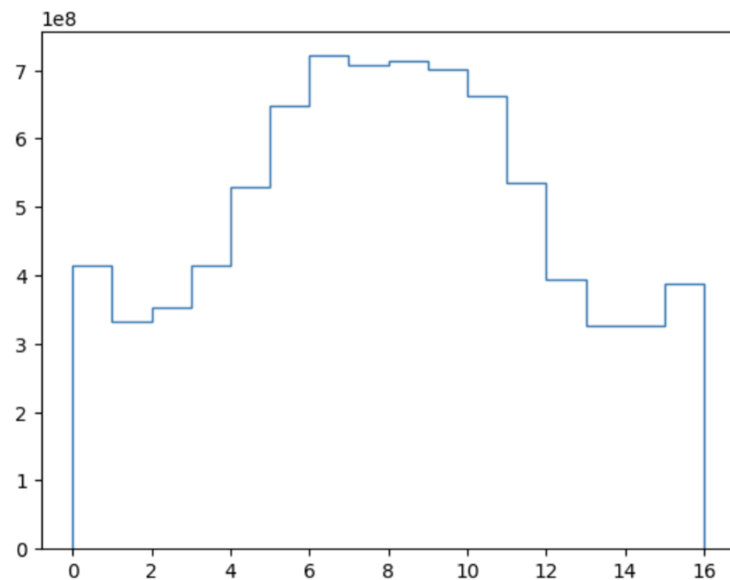
must occur for lookup tables to output more symbols at once. The next section 2.5 outlines a more serious problem that makes entropy methods only gain minor improvements for current large language models without quantization-aware training methods.

2.5 Difficulties of Entropy Compression in Large Language Models

In Figure 2.2, we show that despite LLMs having a normal weight distribution or even with more extreme outliers, recent methods focus on equalizing the probability distribution functions of symbols to recover more information and regain accuracy. On average, group quantization methods are only 5-6% further compressible using entropy methods (ranges from 3-9% across layers).



(a) Fp16 weight distribution from different layers.



(b) Aggregate weight distribution after quantization group of 64.

Figure 2.2: Weight distribution in Llama 2 7B.

Table 2.2: Single-core speed benchmark and overhead relative to reading in quantized bits from Flash or DRAM. On M1 silicon, a state-of-the-art 1.5-1.8 Giga int4 Symbols/s is achieved per core. Alternatives including branchless programming and CPU vectorization were attempted but did not yield speedups.

Optimization/Chip	ESP32-s3 (40nm/5 stage)		M1 (5nm/OoO)	
	Cycles per symbol	Overhead	32KB Chunk	Overhead
0.3KB LUT rANS	58 cycles	3x	40-44ms	20-22x
Coalesced 0.3KB LUT rANS	52 cycles	2.6x	38-40ms	18-20x
128KB tANS	200 cycles	10x	22ms	10x
Loop unrolled 4x	-	-	11ms	5x

2.6 Related and Concurrent Works

2.6.1 Pruning/Sparsity

Previous works such as deep compression [15] force the weights to have more zero symbols using pruning-based re-training methods. Our method currently focuses on the benefit of not needing retraining or sparsity induction, as many large models and convolution networks are difficult to prune. Methods that rely on pruning do have the notable benefit of higher decompression speed and the ability to skip compute as it has many zeros in its entries. That being said, the rANS algorithm outlined here would work as a replacement for Huffman in the final stage of the deep compression.

2.6.2 Sensitivity Based Non-uniform Bins

Recent works such as SqueezeLLM [16] and KV-Quant [17] utilize non-uniform symbol bins to increase the accuracy of quantization, as previous quantization methods bring unacceptable losses to the model. Since symbol bins are non-uniform, the objective here is to equalize the probability of each symbol. Non-uniform bin encoding works against entropy-based compression as it relies on a skewed symbol PDF, making it not further compressible.

While non-uniform bin methods recover accuracy lost by quantization well in LLMs, smaller models: a) do not have outliers as it is caused by softmax in the attention mechanism and b) can mitigate problems of quantization caused by accuracy lost by using quantization-aware training.

Furthermore, dequantized values are of floating point or a much higher precision fixed point, which would increase the compute footprint significantly for TinyML. Note that as quantization methods advance beyond 2 bits, a ternary system achieves more optimal numerical balance than binary, but is difficult to pack and requires entropy-based methods to pack in memory. The next subsection 2.6.3 outlines recent works that show progress on ternary systems and may require entropy-based compression to realize gains.

2.6.3 Ternary LLM

A series of ternary weight [18] language models were recently trained from scratch and showed impressive accuracy when compared to floating point, which converges as models get larger. The obvious drawback to this method is the need to train the large model from scratch to be quantization aware, which takes 21 A100 GPU-years even for a small Llama 7B. Despite the drawbacks, if institutions were to open source Ternary LLMs, one can losslessly compress Ternary LLMs extremely well using methods laid out in this Chapter 2.

Concurrently, QMoE [19] demonstrates a ternary 1B Switch Transformer can be losslessly decompressed through dictionary lookups quickly on GPU. As software decompression speed is the primary concern, we concur that dictionary lookup is essential. Streaming rANS implementation utilizing direct lookup 2.4.2 is also fast, efficient, and small when there are only 3 symbols instead of 16. The primary difference between the lookup mechanism of *rANS* and a simple *stateless dictionary* outlined in QMoE is that rANS outputs an extra state that encodes residual information left from the last byte, and therefore is more compression optimal but requires more lookup space.

Chapter 3

CodeGen: Parallelism in Language Models

3.1 Introduction

In this chapter, we evaluate CodeLlama [20] using the Mostly Basic Python Programming (MBPP) dataset [21] and introduce novel algorithms that optimize trade-offs between accuracy, runtime, and device constraints. To the best of my knowledge, this is the first study to explore the effects of PagedAttention [22] on CodeGen. In this application, parallelism can be substantial, and branching unfolds gradually rather than abruptly. Through page management of KV-cache, an economy of scale is achieved where parallel sampling by a factor of 100 is, on average, only 6.6 times more resource-intensive than single sampling on the MBPP dataset. Combined with data from CodeLlama showing that a 100x sampled 7B model achieves the problem-solving performance of a single sampled 70B model, we have effectively matched the problem-solving capabilities of a larger model with a tenfold reduction in memory requirements. This achievement opens up the potential for local deployment.

```
# Write a function to find the shared elements from the given two lists.

# Test cases:
assert set(similar_elements((3, 4, 5, 6),(5, 7, 4, 10))) == set((4, 5))
assert set(similar_elements((1, 2, 3, 4),(5, 4, 3, 7))) == set((3, 4))
assert set(similar_elements((11, 12, 14, 13),(17, 15, 14, 13))) == set((13, 14))

# Solution
def similar_elements(test_tup1, test_tup2):
    res = tuple(set(test_tup1) & set(test_tup2))
    return (res)
```

3.2 Memory Traffic of Large Language Models Inference

A large language model is an autoregressive generator sampling from learned probability $P_\theta(Token_t|Token_{1...t-1})$. This structure forces the model to calculate token by token without the ability to generate future tokens in one go. For transformers specifically, the memory traffic would include both weight and KV-Cache [23] in the attention mechanism [24], which when combined in the orders of GB.

However, regular quantization methods and batch scaling methods are harder to apply to these models. These difficulties largely stem from the attention mechanism that can query information from past tokens, creating two new problems:

1. Softmax within the attention layer causes occasional outliers in both activations and weights, significantly increasing the dynamic range of the signals.

2. Querying past information from KV-Cache in the attention mechanism incurs off-chip memory traffic, which does not scale efficiently with batch size during inference when the histories of tokens are independent.

We start by addressing how to quantize large tensors with outliers through group quantization in section 3.3. Section 3.4 and 3.5 will explore methods to reduce memory traffic of weights and reduce KV-Cache footprint in the attention mechanism when batch size is large when exploiting application-specific parallelism.

3.3 Outlier Migration through Group Quantization

In order to deal with outliers as demonstrated in Figure 3.1, scale s is given to a group of 64-256 integers to reduce the probability of maximum reaching any outliers, reducing the bin width and quantization error. In conjunction with other concurrent works, weight and KV-cache can be stored in int4 or less with little accuracy loss. This format has been recently standardized known as MX or microscaling format [25].

Fused CUDA kernels are ideally written for all the layers in a transformer to avoid extra memory traffic during the dequantization process. However, this is a time-consuming process to realize. Triton [26] offers a good alternative that translates high-level Python expressible APIs down to CUDA code, allowing for writing custom fused kernels with reasonable engineering effort. The following examples demonstrate and benchmark custom kernels that fuse shift, multiply, and reshape operators, even into matrix multiplication kernels.

3.3.1 Group Quantization Kernels

Even a simple group quantization or dequantization kernel implemented in native GPU avoids PyTorch's repeated read and write operations. Group quantization implemented in

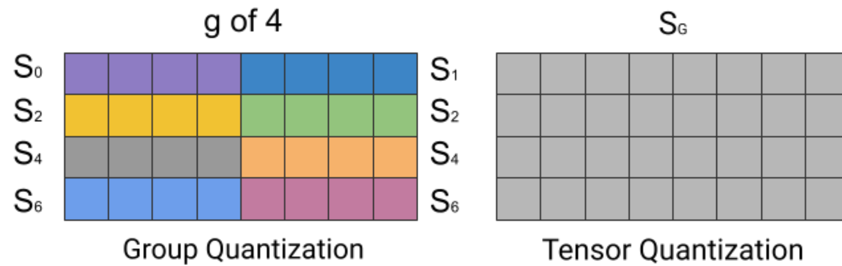


Figure 3.1: Group-wise quantization vs. tensor-wise quantization.

PyTorch causes excessive traffic as it writes intermediate shift and scale operations into DRAM, making two passes of the data in memory traffic. A simple custom kernel in Triton fuses the operators in one pass. Note that quantization is always slower as finding the max scale takes extra effort, for PyTorch this is especially so as it occurs an extra pass in DRAM.

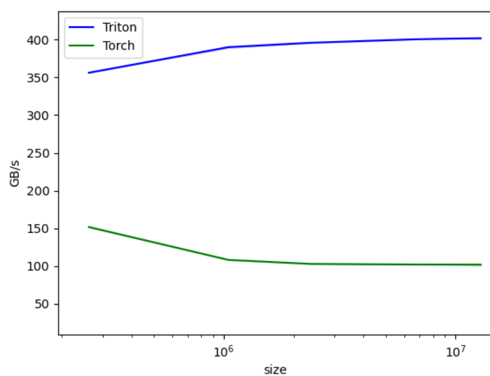
3.3.2 W4A16 Matmul Kernels

In Figure 3.2 we benchmarked three setups of matrix multiplication with quantized numbers with dimensions activation (Batch x M) @ weight (M x M), where activation is in fp16 and Weight is quantized as int4 with group of 64. First is a PyTorch implementation of dequantization and matmul, second a Triton dequantization kernel with PyTorch matmul, and third a fully Triton custom kernel that fuses dequantization with matmul.

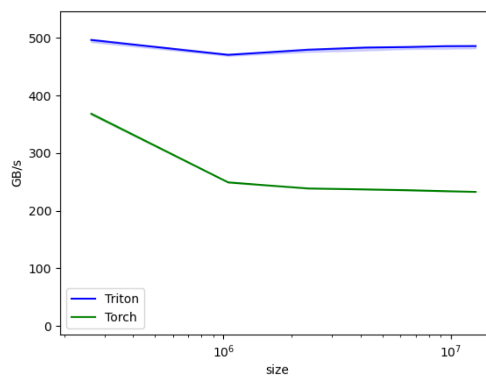
A consistent throughput doubling is observed for the second approach, while the fully custom kernel sees a 5x speedup for matrix size of 4096. Triton's matmul kernels are not as efficiently tuned as PyTorch's for small matrix sizes. While the performance gains from reduced bandwidth may be minor against PyTorch's matmul with small matrices, for M = 4096, the size of CodeLlama 7B matrices, we obtain a 5-6x speedup.

3.4 Application Specific Parallelism

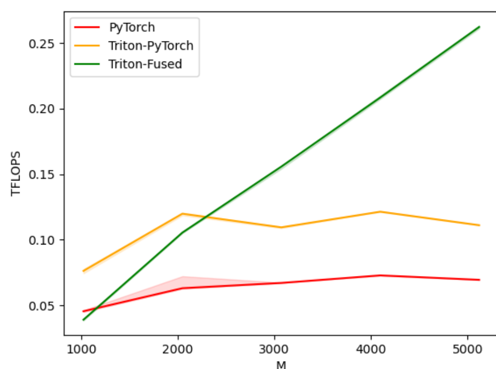
Coding task accuracy rates benefit significantly from running parallel samples. Figure 3.6 shows a consistent pass rate increase as parallel samples increase from 1 to 10 to 100 with increasing temperature settings. These are expected behaviors from CodeLlama [20], though the pass rate is about 10% less as we used 0-shot prompts instead of 3-shot prompts. What is unique about this workload is it breaks the traditional assumption that a single user's inference is single-batched. This allows us to increase the batch size to exploit data reuse of weight traffic in both the feedforward and attention layers of transformers.



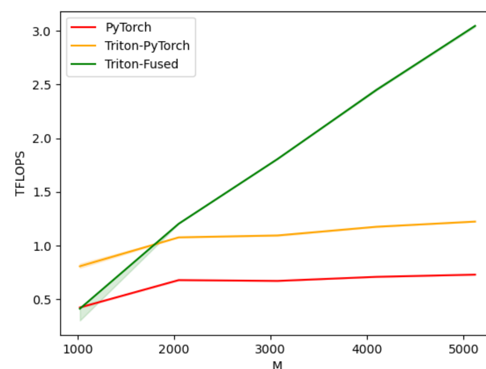
(a) 4x Quantization Speedup.



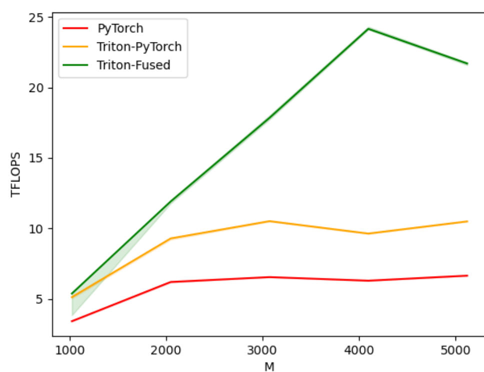
(b) 2x De-quantization Speedup.



(c) Matmul Batch = 1, up to 5x speedup.



(d) Matmul Batch = 10, up to 6x speedup.



(e) Matmul Batch = 100, up to 5x speedup.

Figure 3.2: Custom fused kernel performances compared to naive PyTorch implementation, benchmarked on RTX 4080.

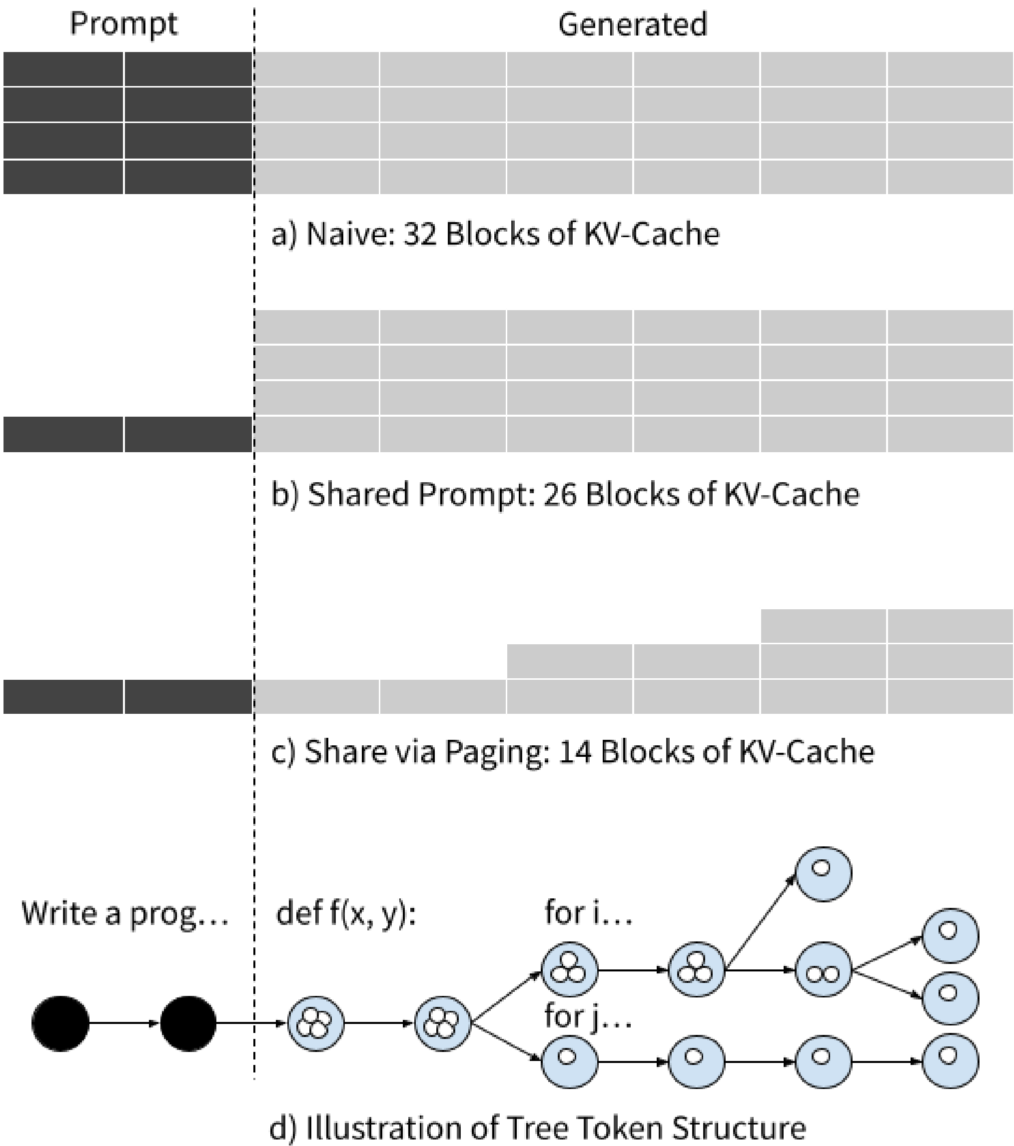


Figure 3.3: KV-cache and inference batch size, comparing naive, shared prompt, and paging.

However, the attention mechanism traditionally scales poorly even with large batches. Due to its need to recall its own context as demonstrated in Figure 3.4, KV-cache memory footprint and traffic usually scale linearly by batch size. By analyzing the dependencies graph for this specific application, we find that paged based attention kernel can effectively unlock parallelism and sequence-dependent memory optimizations allowing for the reuse and amortization of shared contexts. See Figure 3.3 for a visual example.

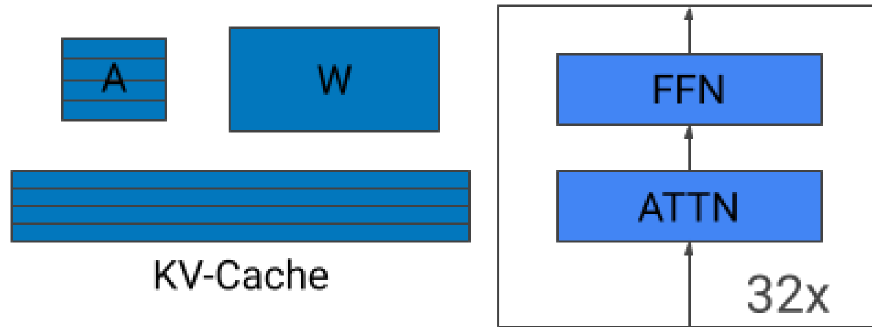


Figure 3.4: Weight can achieve memory reuse through high batch, but KV-cache traditionally does not.

3.4.1 Particle Sampling

The fundamental characteristic of a planning-infused LLM is its ability to allocate more compute and memory resources as the problem difficulty requires. This subsection outlines a novel method that does the equivalent of multiple parallel sampling without actually spending the compute or memory if trajectories overlap, effectively forming a tree with a limited branching budget that caps at the max batch size.

3.4.1.1 Particles & Forking

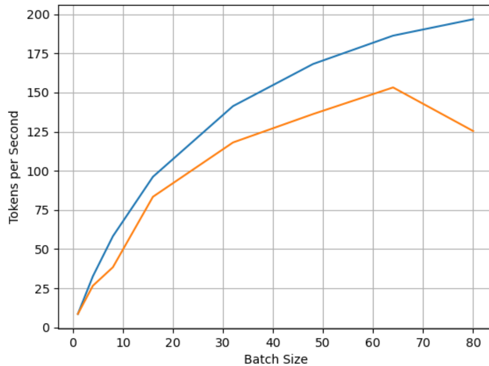
Particle count is initialized as the maximum batch size, all occupying the same root token node. The node will then sample the model action particle times, and upon forking, duplicate KV-cache or the paged KV-cache as needed. Figure 3.5 shows the effective speedup when utilizing this method.

3.4.1.2 Fine Grained Temperature Annealing

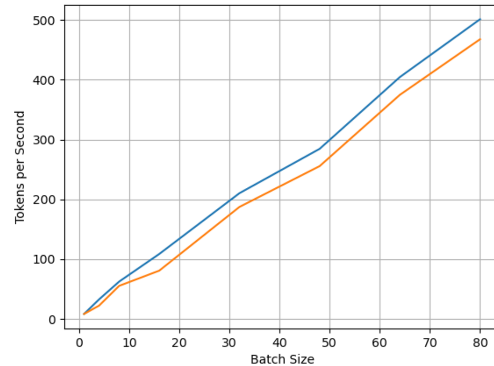
The token sampling algorithm needs improvement to balance exploration and exploitation trade-offs that are aware of hardware costs. Normally, a constant temperature T regulates the confidence of token sampling, with lower temperature results in a more confident distribution that reduces diverse output.

$$softmax(y_i) = \frac{e^{y_i/T}}{\sum_{j=1}^{n} e^{y_j/T}}$$

A simple temperature increase may be able to increase exploration of trajectories, but it is not aware that KV-cache and batch size must be limited for a reasonable runtime. Constant temperature results in risky exploration near the end if most particles have been



(a) Effective Tokens/s without Particles



(b) Effective Tokens/s with Particles

Figure 3.5: Blue: Tokens/s produced, Orange: Tokens/s used in the program (eliminating special tokens and trajectories that ended early).

spent already. In Figure 3.6, higher temperature @1 shows a deteriorating pass rate. It is hence reasonable to reduce temperature if there are limited particles left on this path, forcing a good solution to converge given a limited budget.

A simple linear relation between $(1, T_{min})$ and (B_{max}, T_{max}) effectively maintains sample diversity while not spending excessive time to converge on a good solution. In Fig 3.6, lines with Annealing consistently show on par pass rate while spending less time, and $T_{min} = 0.1$ while the originally given temperature becomes T_{max}.

$$T_{t,i} = (T_{max} - T_{min})/(B_{max} - 1) * (B_{t,i} - 1) + T_{min}$$

3.5 KV-Cache Compression: Shared Prompt & PagedAttention

Even after quantization, the memory footprint of reserving 768 tokens for 100 batches is 10GB. We hence need to find ways to a) share the storage of KV-cache that has a common history and b) not reserve memory for all the tokens unless materialized as the end length varies significantly during sampling. PagedAttention [22] effectively manages all these scenarios. In our specific application, most compression comes from simply sharing the problem description prompt, by adopting prompt-sharing for a 512-token prompt + 256-token answer format, now only less than 3.5GB is needed.

From Figure 3.7, we can expect more improvements from paging + particle sampling, as it only increases batch size when needed. Given prompt storage overhead is negligible and

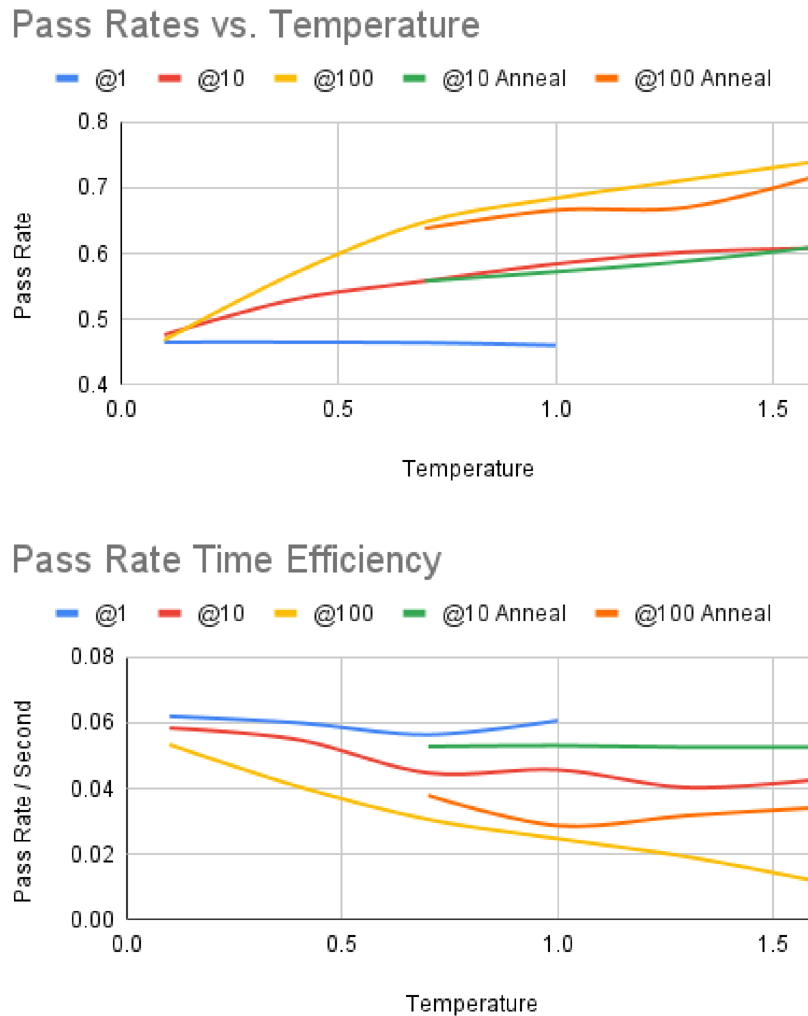


Figure 3.6: Llama-7B on MBPP, 0-shot prompt pass rates @1, @10, @100 with varying temperatures, top-k=32. The annealing schedule is defined below. For data see Tables 3.1 and 3.2.

only 20% of the batches are active on average, only 0.7GB is needed, which is only 20% of the 3.5GB of weights. This allows us to run CodeLlama 7B with max prompt length of 512 tokens, a max code length of 256 tokens, and a paged batch size of 100 with ease on a 2080Ti with only 11GB of VRAM.

Table 3.1: Pass rates of CodeLlama 7B on MBPP.

Temperature	Pass @1	Pass @10	Pass @100	Pass@10 Anneal	Pass@100 Anneal
0.1	0.4649	0.476	0.468		
0.4	0.4649	0.530	0.57		
0.7	0.464	0.558	0.648	0.558	0.638
1.0	0.460	0.584	0.684	0.572	0.666
1.3		0.602	0.712	0.588	0.670
1.6		0.608	0.740	0.610	0.718

Table 3.2: Avg seconds per problem for CodeLlama 7B on MBPP.

Temperature	Pass @1	Pass @10	Pass @100	Pass@10 Anneal	Pass@100 Anneal
0.1	7.50	8.15	8.77		
0.4	7.75	9.67	14.07		
0.7	8.23	12.48	21.22	10.57	16.86
1.0	7.58	12.79	27.72	10.78	23.23
1.3		14.95	37.09	11.18	21.12
1.6		14.24	63.09	11.59	21.05

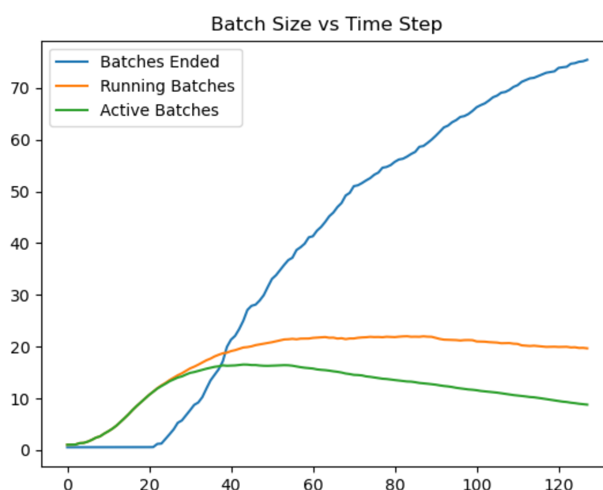


Figure 3.7: Average batch size over time: on average, memory and compute reduction equals the area above the orange line till y=100.

3.5.1 Chunk Attention: Prefix-Aware PagedAttention

To materialize these compression benefits in further speedup, kernels[1] were implemented that allow the prompt KV-cache memory traffic to be shared across samples. It fuses:

- FlashAttention [27], utilizing the online softmax mechanism to fuse the softmax kernel with matrix multiply in one pass.
- PagedAttention [22], allowing for KV-Cache sharing in memory storage.
- ChunkAttention [28], illustrated in Figure 3.5.1, explicitly allowing the shared prompt to achieve an extra degree of parallelism to fully saturate tensor core throughput.
- A novel TreeAttention kernel, illustrated in Figure 3.5.1 saving DRAM traffic for fine-grained branching of KV-cache by making sure repeated elements are resident in L2 cache of GPU.

Upon synthetic benchmarks shown in Figure 3.9, we find that ChunkAttention indeed achieves a speedup proportional to the percentage of shared prompt in the entire context, here is 50%. Furthermore, TreeAttention can further reduce 15.5% RAM traffic when a branching factor of 1 occurs every time step. For modern GPUs that can compute much faster than memory bandwidth, we observe that a TreeAttention-only kernel can achieve a speedup against a ChunkAttention-only kernel without even needing to explicitly parallelize the shared prompt.

On the MBPP dataset specifically, we were not able to integrate these kernels in time, but based on prompt and solution length ratio of 2:1, the synthetic benchmarks suggest on average at least a threefold acceleration of the attention kernel. This is left for future work.

Cumulatively, the 5x reduction of batch size on average through particle sampling and 3x reduction of KV-cache footprint through page management reduces memory footprint and bandwidth by 15x of the original KV-cache size, making scaling by parallel batches more economical.

[1]Kernels are open sourced at https://github.com/hongyihuang/spec-mcts/blob/main/triton_kernels.py

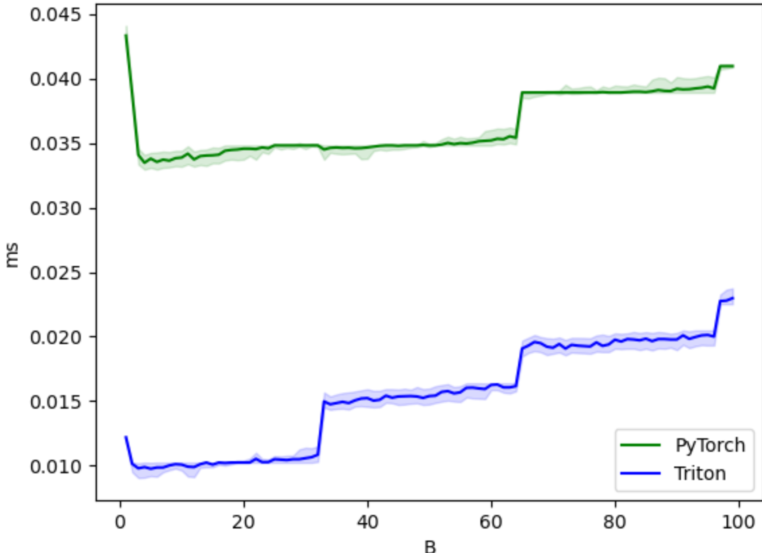


Figure 3.8: ChunkedAttention (blue), shared prompt only comparing against FlashAttention (green).

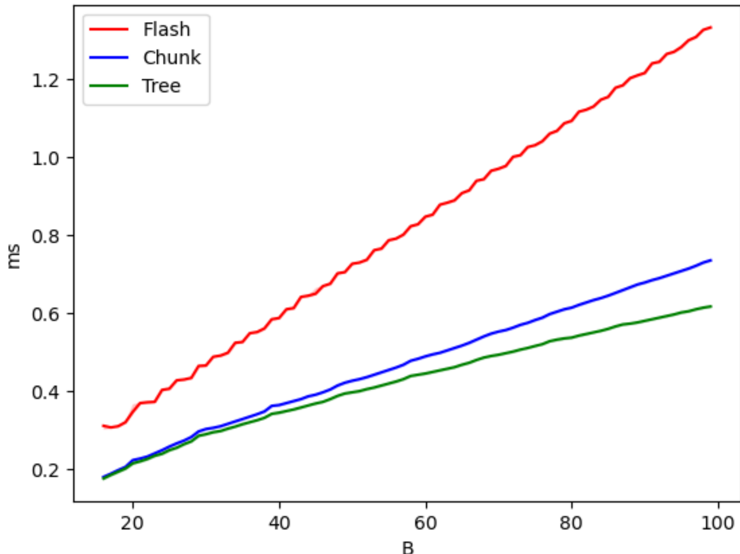
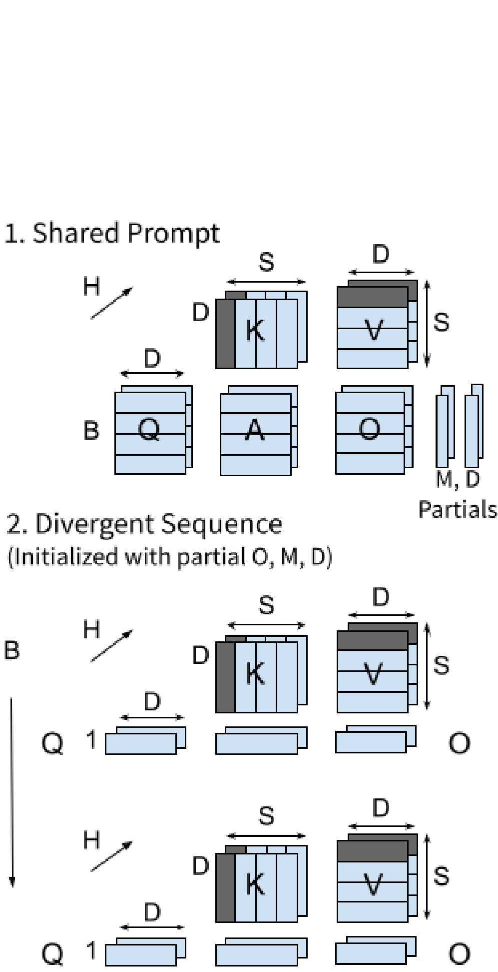
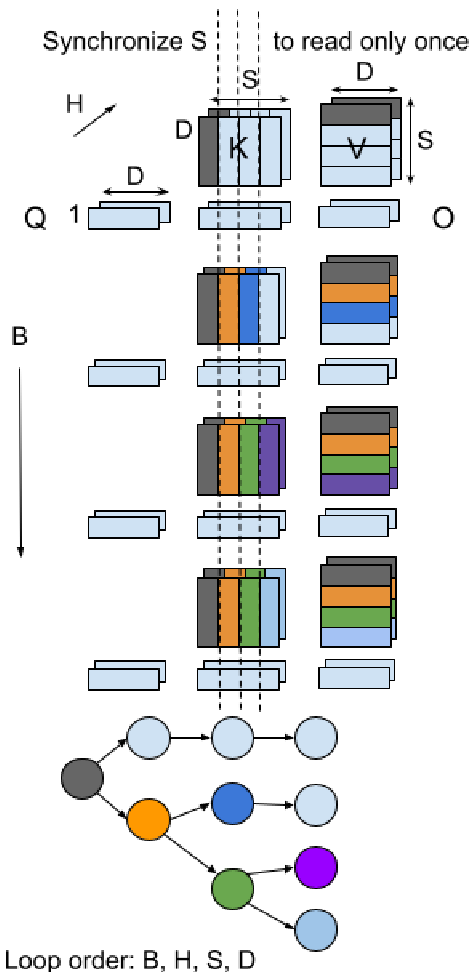


Figure 3.9: ChunkAttention (with non-shared KV-Cache), tested with half shared-prompt and half divergent-context. TreeAttention is tested with an extra branch of 1 per time step along with half shared-prompt.



(a) Chunk attention kernel illustration.



(b) Tree attention kernel illustration.

Chapter 4

Conclusion

In summary, this thesis adds to the field the use of entropy-based compression and activation sparsity for small language models that fit on embedded systems, which both can potentially double inference performance compared to simple int4 quantization. We demonstrate that asymmetrical numerical systems can optimally compress weights for models with less dynamic range with minimum overhead, achieving 2 bits per weight bitrate. Additionally, activation sparsity can be induced via inserting ReLU activations during training, and skipping weight reads for sparse activations can also reduce 50% of off-chip bandwidth.

In contrast, large language models have weights of large dynamic range. Parallel sampling, paging-based memory management, and novel attention kernel for specific applications such as code generation to achieve performance equal to tenfold larger models and 15x less KV-cache footprint. This thesis contributes by discovering the effects of paging-based memory management on CodeGen tasks specifically and adding a novel tree attention kernel that can fuse the attention layer arithmetic and optimally share KV-cache traffic.

These methods allow for effective local or distributed deployment in both embedded systems and language model applications that are real-time or privacy critical. Most importantly, for models of any scale, writing fused kernels that avoid extra memory passes between multiple operations is demonstrated to be the most reliable way to obtain speedup. Compilers that can automatically fuse rudimentary operators without needing to manually program may prove useful.

There are still many future research avenues for these three methods. In the case of activation sparsity, solving the various challenges of scaling up the sparse model remains an open problem discussed in Section 1.3.2. While entropy-based compression is only useful for small models or non-transformer models, a speedup can be realized when a hardware decompression unit is implemented to minimize resource consumption. We also await the open-source of any BitNet or Ternary LLMs for further experimentation of entropy-based compression. Finally, for large language models in applications that benefit from parallel sampling, more sophisticated planning and search that resembles tree search AlphaGo may improve performance further against larger models.

Bibliography

[1] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *CoRR* abs/1712.05877 (2017). arXiv: 1712.05877. URL: http://arxiv.org/abs/1712.05877.

[2] Raghuraman Krishnamoorthi. *Quantizing deep convolutional networks for efficient inference: A whitepaper*. 2018. arXiv: 1806.08342 [cs.LG].

[3] Hao Wu et al. *Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation*. 2020. arXiv: 2004.09602 [cs.LG].

[4] Keivan Alizadeh et al. *LLM in a flash: Efficient Large Language Model Inference with Limited Memory*. 2024. arXiv: 2312.11514 [cs.CL].

[5] Iman Mirzadeh et al. *ReLU Strikes Back: Exploiting Activation Sparsity in Large Language Models*. 2023. arXiv: 2310.04564 [cs.LG].

[6] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL].

[7] Ronen Eldan and Yuanzhi Li. *TinyStories: How Small Can Language Models Be and Still Speak Coherent English?* 2023. arXiv: 2305.07759 [cs.CL].

[8] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL].

[9] Vladislav Lialin et al. *ReLoRA: High-Rank Training Through Low-Rank Updates*. 2023. arXiv: 2307.05695 [cs.CL].

[10] Jiawei Zhao et al. *GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection*. 2024. arXiv: 2403.03507 [cs.LG].

[11] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[12] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[13] G. G. Langdon. "An Introduction to Arithmetic Coding". In: *IBM Journal of Research and Development* 28.2 (1984), pp. 135–149. DOI: 10.1147/rd.282.0135.

[14] Jarek Duda. *Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding*. 2014. arXiv: 1311.2540 [cs.IT].

[15] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV].

[16] Sehoon Kim et al. *SqueezeLLM: Dense-and-Sparse Quantization*. 2024. arXiv: 2306.07629 [cs.CL].

[17] Coleman Hooper et al. *KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization*. 2024. arXiv: 2401.18079 [cs.LG].

[18] Shuming Ma et al. *The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits*. 2024. arXiv: 2402.17764 [cs.CL].

[19] Elias Frantar and Dan Alistarh. *QMoE: Practical Sub-1-Bit Compression of Trillion-Parameter Models*. 2023. arXiv: 2310.16795 [cs.LG].

[20] Baptiste Rozière et al. "Code Llama: Open Foundation Models for Code". In: *Meta AI* (2023).

[21] Jacob Austin et al. "Program Synthesis with Large Language Models". In: *arXiv preprint arXiv:2108.07732* (2021).

[22] Woosuk Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. 2023. arXiv: 2309.06180 [cs.LG].

[23] Reiner Pope et al. *Efficiently Scaling Transformer Inference*. 2022. arXiv: 2211.05102 [cs.LG].

[24] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].

[25] Bita Darvish Rouhani et al. *Microscaling Data Formats for Deep Learning*. 2023. arXiv: 2310.10537 [cs.LG].

[26] Philippe Tillet, H. T. Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19. ISBN: 9781450367196. DOI: 10.1145/3315508.3329973. URL: https://doi.org/10.1145/3315508.3329973.

[27] Tri Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022. arXiv: 2205.14135 [cs.LG].

[28] Lu Ye et al. *ChunkAttention: Efficient Attention on KV Cache with Chunking Sharing and Batching*. 2024. URL: https://openreview.net/forum?id=9k27IITeAZ.