LLaMA2 from Scratch: An own Implementation

1st Christian Bernhard
M. Sc. Computer Science
Ludwig-Maximilians-Universität
Munich, Germany
c.bernhard@campus.lmu.de

2nd Thomas Ziereis
M. Sc. Computer Science
Ludwig-Maximilians-Universität
Munich, Germany
thomas.ziereis@campus.lmu.de

***Abstract*— This project aims to demystify the inner workings of modern Large Language Models (LLMs) by implementing Llama2 from scratch. The focus is on understanding the end-to-end process from training to inference by rewriting the components that contribute to the model's ability to comprehend and generate human-like text. By doing so, we aim to understand the technological and theoretical principles that make such models effective.**

***Index Terms*—Llama2, Large Language Models, Natural Language Processing, Attention**

I. Introduction

In the past few years, significant progress has been made in the field of natural language processing (NLP), mainly through the development of large language models (LLMs). These models have significantly improved the ability to understand and generate human-like text, opening new paths for research and application. This paper is dedicated to a detailed exploration of one such model, LLaMA2, by demystifying its architecture and implementation from the ground up. The study is organized into three main parts:

1) Highlighting the key modifications that distinguish the LLaMA architecture from the original Transformer model proposed in "Attention Is All You Need" (Vaswani et al., 2017) [1].
2) The process of loading pretrained weights from Meta's release and performing inference, demonstrating the practical application of the model and different sampeling strategies.
3) Optimization techniques applied to LLMs, focusing on improving inference time through methods such as quantization.

Even though sequential models such as RNNs and their evolved counterparts, LSTMs, were the go-to solution for processing text in AI for a long time, they came with several drawbacks. These models struggled with slow computation times for long sequences, vanishing or exploding gradients, and difficulties in accessing information from distant tokens. The introduction of the Transformer model in 2017 marked a paradigm shift by addressing these issues, primarily through its innovative self-attention mechanism. Building upon this foundation, the LLaMA architecture introduces several key adjustments to the vanilla Transformer architecture from 2017.

This paper assumes the reader has a foundational understanding of the principles underlying LLMs and the Transformer architecture, including concepts such as neural network training, embeddings, and self-attention mechanisms. As such, detailed explanations of these foundational concepts will not be provided.

The complete codebase developed during this project, along with detailed documentation, is available for public access at https://github.com/ziereis/simple-llama.

II. LLaMA2 Key Concepts

The exploration of LLaMA2's architecture requires an understanding of several key concepts that differentiate it from its predecessor Transformers 1. This section delves into these innovations, emphasizing their contributions to the model's performance and efficiency. For our investigation, we utilize the LLaMA2-7b and LLaMA2-7b-chat variants. The former serves as a general document generator/completer, while the latter is specifically optimized for chat-based Q&A interactions. Our results (see Section V) show that the chat version demonstrates significantly more stable overall performance. Our repository provides implementations in PyTorch, C/C++, and CUDA, with the PyTorch variant offering a more accessible entry point. To make the model easier to comprehend and lightweight, we removed features such as parallel batch processing, resulting in simpler code that demands fewer resources. By employing further techniques such as quantization (as discussed in Section IV), we achieved the ability to run this model on standard consumer hardware.

In comparison, LLaMA2 introduces several architectural adjustments that refine the original Transformer design (Fig. 1). For example, the normalization processes in LLaMA2 precede both the attention layers and the feedforward blocks, distinguishing it from the post-layer normalization routine seen in traditional Transformer architectures. Further distinctions or new concepts are explained in this section.

A. Rotary Positional Embeddings

In the original Transformer architecture, positional embeddings were introduced to preserve the sequential information of input sequences [1]. However, these embeddings have some limitations, particularly when dealing with long-range

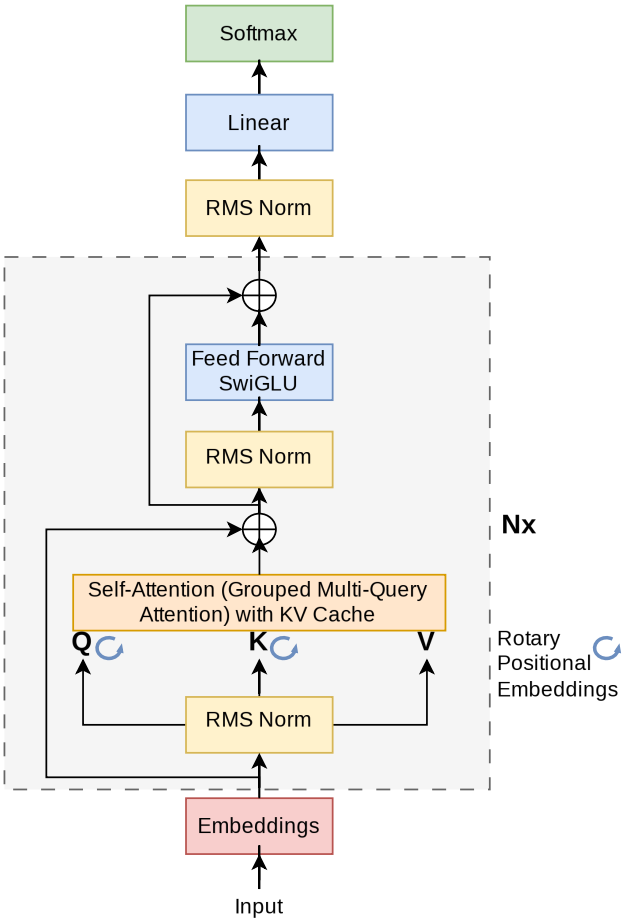

Fig. 1. LLaMA2 Architecture with its key components

dependencies. To address this, the LLaMA language model employs Rotary Positional Embeddings (RoPE) [2].

In contrast to the original Transformer architecture, where positional embeddings are a fixed set of learned vectors, RoPE represents positional information as a rotation in vector space. This rotation is not learned during training and is typically precomputed.

A key difference between these two embeddings is that the original Transformer's positional embeddings represent absolute positions in a sequence. Specifically, every token at position x uses the positional embedding for position x. In contrast, RoPE encodes the relative distance between tokens, capturing the relationships between them in a more nuanced way. This means that for a token at position x, each previous token is rotated by an amount determined by its relative distance from the current token. A token that is the direct predecessor of the current token would only be rotated a marginal amount, whereas a token further in the past of the sequence would be rotated an exponentially larger amount.

The advantages of this approach are twofold. First, during training and inference, a max_seq_len times embedding_dim matrix needs to be maintained less, which reduces memory consumption, a valuable resource, especially when running on GPUs. Additionally, since the embeddings do not need to be learned and can be precomputed, there are fewer gradients to maintain during training, resulting in less computation [2].

B. RMS Normalization

The quest for computational efficiency without compromising model performance has led to the adoption of Root Mean Square Normalization (RMSNorm) in the LLaMA2 architecture. This technique emerges as an alternative to the conventional Layer Normalization, with distinct advantages that make it suitable for large-scale language models.

RMS norm has several advantages over traditional LayerNorm. First, RMS norm is computationally more efficient than traditional LayerNorm, particularly for large input sequences. This is because the RMS norm only requires calculating the square root of the sum of the squares of the input activations, which is faster than calculating the mean and standard deviation. Second, empirical evidence has shown that the efficiency gain does not come at the cost of performance, since it performs as good as traditional LayerNorm for various architecture including transformers [3].

RMSNorm operates by rescaling the input activations based solely on their root mean square value. This modification maintains the core principle of normalizing the scale of activations, allowing for stable gradients during backpropagation, but eliminates the mean centering step.

$$\hat{a_i} = \frac{a_i}{\text{RMS}(a)}, \tag{1}$$

where RMS(a) is calculated as follows:

$$\text{RMS}(a) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}. \tag{2}$$

In LLaMA2's implementation, RMSNorm comes with the addition of a learnable parameter, analogous to the gamma in Layer Normalization, which is applied post-normalization to retain model flexibility.

C. Self Attention with KV Cache

In the inference stage of LLaMA2, the utilization of a Key-Value (KV) Cache is an innovative strategy to enhance computational efficiency. The premise is straightforward: when generating text, the model's current focus is on predicting the next token. Each token produced is contingent on the entire preceding sequence—the context or 'prompt' from which it must draw inference.

Traditional self-attention mechanisms require recalculating the attention across all tokens, including those previously processed, which is computationally redundant. LLaMA2 addresses this by implementing a KV Cache system, which preserves the results of attention computations for previously seen tokens. During each inference step, the model retrieves relevant information from this cache, thereby reducing the computational load significantly.

Therefore, our generate_text function has two primary stages. First, it inputs the initial prompt to populate the KV

Cache without utilizing the model's output, setting the stage for subsequent inference steps. The second stage generates the actual text completion by sampling new tokens by using the KV Cache.

```
def generate_text():
...
# Populate KV Cache, ignore models output
output_tokens = []
for token in prompt:
  _ = model.forward(token)
  output_tokens.append(token)

# Completion Sampling, use models output
while len(output_tokens) < max_toks:
  latest_token = output_tokens[-1]
  output = model.forward(latest_token)
  output_tokens.append(sample(output))
...
```

D. Attention Types: Grouped Query, Multi Query, Multi-head

The advancements in GPU technology have led to a scenario where memory access and data transfer, rather than raw computational power, often become the performance bottlenecks in large language models. To address this, LLaMA2 optimizes the self-attention mechanism's structure.

Traditionally, each attention head in a multi-head attention mechanism computes its own set of Key (K) and Value (V) matrices (Fig. 2 left). This approach, while effective in capturing diverse features of the input, is memory-intensive.

An efficiency-driven variation is the multi-query attenion. In this approach only one set of K and V matrices is computed, and the diversity of attention is maintained by having multiple Query (Q) matrices (Fig. 2 right). This significantly reduces memory transfer overhead in exchange for some of the models performance.

A hybrid between both approaches that larger variants of LLaMA2 employ is the grouped multi-query attention (Fig. 2 middle). This method offers computational advantages by reducing the number of heads for keys and values, thus reducing memory requirements without significantly affecting the quality of attention, as is the case with multi-query. However this was only used in the bigger LLaMA2 Models, not in the 7b version we are using. Our implementation therefore represents the traditional multi-head-attention architecture [4].

E. Feedforward Network with SwiGLU Activation

LLaMA2 introduces a variation in the feedforward network (FFN) layer with the adoption of the SwiGLU activation function. This change implements an additional parameter matrix, resulting in three matrices W, V, W_2 as opposed to the original FFN's two, yet ingeniously maintains the overall parameter budget by reducing the dimensionality of these matrices.

The traditional FFN layer in the Transformer architecture, as described in "Attention Is All You Need", applies the Rectified Linear Unit (ReLU) activation function, defined as:

$$\text{FFN}_{\text{Transformer}}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (3)$$

In contrast, the LLaMA model utilizes:

$$\text{FFN}_{\text{SwiGLU}}(x, W, V, W_2) = (\text{Swish}_1(xW) \odot xV)W_2 \quad (4)$$

where the Swish function, defined as $\text{swish}(x) = x \cdot \sigma(\beta x)$, employs a sigmoid function to introduce a non-zero gradient for negative inputs, as opposed to the immediate cutoff at zero by ReLU. Here β is typically set to 1, making it the Sigmoid Linear Unit (SiLU).

The practical benefits of SwiGLU in LLaMA2 are not entirely understood in theory. However, empirical results show that it outperforms ReLU in certain contexts, particularly by allowing the network to retain and learn from information that would otherwise be disregarded by ReLU's zeroing effect on negative inputs [6].

F. Model Forward Pass

Combining all the mentioned elements, you end up with a state of the art LLM architecture for text generation 1. The procedure begins with the conversion of raw input into a sequence of tokens, which are subsequently passed through an embedding layer to attain a dense vector representation. These embeddings are then processed by a sequentially series of N_x Encoder Blocks, each enhancing the contextual understanding of the token sequence.

After the embeddings have traversed the encoder blocks, they encounter a final FNN. This network takes the contextual embeddings and maps them onto a vocabulary-size output space. It is from this output that the model samples the prediction for the next token with the help of the previously introduced generate_text method.

The next token prediction becomes the input for the subsequent iteration of this generative loop, enabling the model to produce text one token at a time, until a ¡EOS¿ token or the limit of output tokens is reached. The specifics of this inference loop will be further elaborated in the following section III.

```
def forward(self, token: int, pos: int):
  embed = self.tok_embeddings(token)
  for layer in self.layers:
    embed = EncoderBlock(embed, pos)
  embedding_norm = self.norm(embed)
  output = self.output(embedding_norm)
  return output
```

III. Inference

A. Decoding Strategies

Decoding strategies are crucial in determining how language models generate text. Each strategy influences the variability of the generated text. Here, we discuss several strategies, including Greedy, Temperature Scaling, Random Sampling, Top-K, and our primarily employed Top-P Sampling.

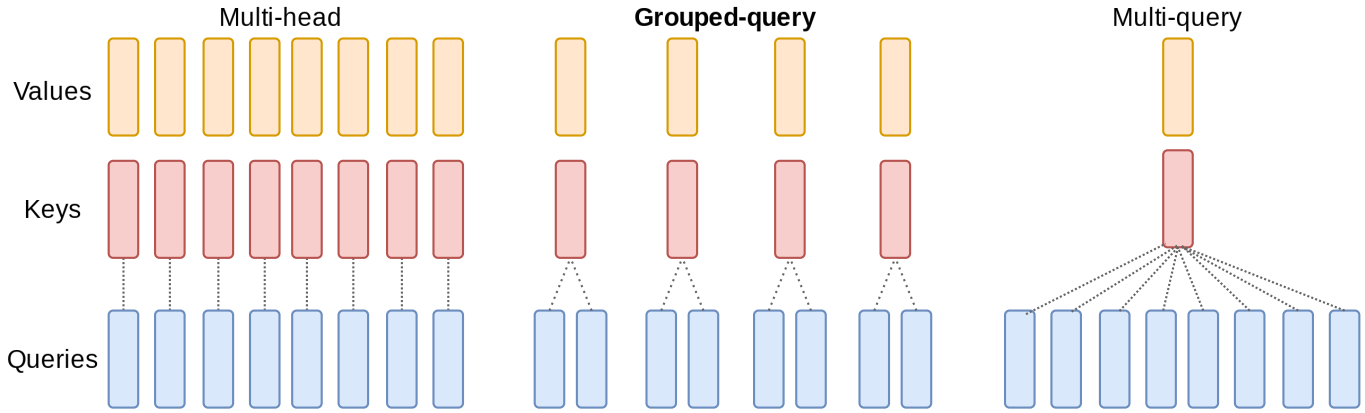

Fig. 2. Different Attention Types: Multi-head, Grouped-query, Multi-query [5]

B. Greedy

Greedy decoding is the simplest form of text generation in which the model selects the most likely next token at each step. This deterministic approach can quickly resolve highly probable sequences but often leads to repetitive or predictable text as it fails to explore less obvious alternatives that might offer more interesting or contextually appropriate outputs [7].

C. Top K

Building on the concept of minimizing less likely outputs, as seen in greedy decoding, Top-K sampling further refines the selection process. Top-K sampling addresses some of the randomness by restricting the sampling pool to the K most likely next tokens. This limits the risk of selecting highly improbable tokens, thus maintaining a balance between randomness and relevance. However, this method can still occasionally produce less relevant tokens, especially if K is set too high [7].

D. Temperature Scaling

For a more nuanced control over the selection process, we employ temperature scaling. Temperature scaling modifies the logits of the output probabilities before applying the softmax function. By adjusting the temperature parameter, we can control the concentration level of the probability distribution. A lower temperature (e.g., 0.1) makes the model outputs more confident and less varied, whereas a higher temperature (e.g., 1.0) increases diversity in the choices by flattening the probability distribution, making less likely tokens more probable [7].

E. Top P (Nucleus) Sampling

Finally, to effectively manage the balance between diversity and coherence, we implement Top-P (Nucleus) Sampling. Top-P (Nucleus) Sampling enhances text generation by dynamically adjusting the set of potential tokens based on their cumulative probabilities exceeding a threshold p. This method flexibly adapts to the model's confidence level: in uncertain scenarios, it considers a broader array of tokens, while in more confident scenarios, it narrows the selection. This adaptability is crucial for balancing coherence with creative variance in outputs. Our implementation employs a combination with temperature scaling to further fine-tune the probability distribution. A lower top_p setting tightens the narrative by focusing on more probable tokens, enhancing relevance and coherence, and a higher setting increases narrative variety by including a wider range of tokens [7]. The generate_text functions let's you simply choose between all mentioned sampling strategies.

IV. Optimization

A. Performance Considerations

Before optimizing the inference speed of large language models (LLMs), it is crucial to understand what dominates the execution time of LLM inference. In general, the inference process of LLaMA2 can be divided into two parts: prompt evaluation and token generation. When prompting LLaMA2, the first part of the execution involves feeding the actual prompt to provide the necessary context for generating further tokens. This stage consists of feeding an array of tokens, namely the initial prompt, into LLaMA2 to populate its KV Cache. The next part of the execution is the actual token generation stage, where single tokens, represented as integer IDs, are fed into the network, and a single token is predicted as output [8].

It is essential to consider these two stages separately, as they have distinct performance characteristics. Specifically, feeding the initial prompt is compute-bound, whereas the token generation stage is memory-bound. When feeding a vector of tokens into the forward pass, we query multiple embeddings from the embedding matrix, and our current activations, which operate on the forward pass, will always be a prompt_len times embedding_dim matrix. Since the majority of the weights are matrices (e.g., attention and FFN weights), the primary unit of computation will be matrix-matrix multiplication, which, if implemented properly, is bounded by the number of operations a system can perform and not the time spent loading memory into registers [9].

On the other hand, in the generation stage, we always feed a single token in the forward pass, which means our activations will always be a vector. This leads to the fact that all matrix-matrix multiplications turn into matrix-vector multiplications, which are primarily memory-bound. We implement the prompting phase by feeding single tokens and ignoring the output until the KV Cache is populated. Although this approach has only disadvantages in terms of runtime, it makes the implementation easier to understand and more readable, which was our goal [10].

This analysis leads to the conclusion that the primary and absolute dominant factor for our runtime performance will be determined by the size of the model and the speed at which we can transfer memory. Understanding this should be sufficient to make it clear that the major factor for improving LLaMA2 inference speed is quantization, as it is a very efficient way to reduce model size.

B. Quantization

Quantization is a technique used to reduce the precision of the model's weights from floating-point to lower-bit-width integers. This process is integral to optimizing the LLaMA2 model for faster inference and decreased model size, making it more accessible for deployment on our edge devices.

The quantization process involved grouping weights to compute shared scale factors, reducing the granularity of adjustments needed per weight. This grouping was crucial to maintaining a balance between compression efficiency and model performance. Specifically, larger group sizes allowed for more uniform scale factors, which can simplify the computational overhead but may introduce greater error. The error introduced by quantization was systematically measured to ensure it remained within acceptable limits.

We employed a specific quantization strategy known as the absolute maximum (absmax) quantization [11]. In this method, the absolute maximum value within each group of weights is used to determine the scale factor for that group. The scale factor, S, is calculated as follows:

$$S = \frac{\text{max\_int\_value}}{\text{absmax}}$$

where max_int_value is the maximum storable value for the quantized data type (127 for int8 and 7 for int4). Each weight, W, is then quantized to Q by [11]:

$$Q = \text{round}\left(\frac{W}{S}\right)$$

and subsequently dequantized back to W' using [11]:

$$W' = Q \times S$$

The error, E, introduced by this quantization process is defined as the maximum absolute difference between the original and the dequantized weights:

$$E = \max(|W - W'|)$$

This measure helps to ensure that quantization errors are kept to a minimum and remain within acceptable bounds.

We employed two different quantization schemes: 8-bit (int8) and 4-bit (int4). The original model weights, provided in a half-precision float datatype (bf16) from the consolidated checkpoint.pth file by Meta, occupy 12.6 GB in a common PyTorch format. Since studying how to parse this file would be a major undertaking, we exported this file into our own binary data format, which we can easily parse.

This has one disadvantage, as our CPUs do not support the bf16 type natively. Therefore, we used the fp32 to store the base model, leading to a model size of roughly 26 GB. Transitioning to quantized formats, the sizes of the quantized models were significantly reduced: the q4.bin model is only 3.3 GB, and the q8.bin model is 6.5 GB. This represents a substantial decrease in storage requirements, by approximately 87% for the 4-bit quantized model and 75% for the 8-bit quantized model, compared to the non-quantized .bin format.

While the reduction in size and increase in inference speed are advantageous, naive absmax quantization does not come without its trade-offs. As the results section will show, especially the int4 quantization, despite its storage and speed benefits, leads to a more noticeable degradation in the quality of the model's language generation capabilities compared to the original FP16 weights. Int8 offers a more balanced compromise, featuring a noticeable boost in inference speed with a less significant drop in output quality.

Certain operations in neural networks, particularly those involving normalization and activation functions, require high precision to maintain accuracy [10]. Another reason for not quantizing these layers is that they occupy very little space compared to other layers, since the weights are one-dimensional. Most layers are two-dimensional, like the Q weights for the attention, which occupy 512 MB in fp32 format. Therefore, it is usually a good trade-off to not quantize these layers and trade a little bit larger model for more accuracy.

One way to run a forward pass on the quantized weights is to dequantize them on the fly in the forward pass and run operations as usual. However, this approach does not provide any runtime benefit, as the amount of data that needs to be transferred between the CPU and DRAM remains the same. To really exploit the potential performance benefit of quantization, one must implement a matrix-vector multiplication that can directly operate on the quantized weights. A reference implementation of a naive Q8MatVecMul is displayed in figure 3 [10] [8].

The general structure of this is very similar to normal matrix-vector multiplication, with the key difference that we perform the inner dot product between one row of the weights w and the activations x on a by-group basis. This enables us to first accumulate the intermediate result using integer operations before multiplying it by the scales of the group. One thing to note is that it is important to upcast the i8 values to a higher precision before multiplying them to avoid overflowing the integers.

The benefit of using a quantization-aware matrix-vector multiply is twofold. Firstly, we directly operate on the quan-

```
fn Q8MatVecMul(w: QMatrix, x: QVector,
               group_size: int) -> Vector:
  groups_per_row = w.n_rows / group_size

  out: Vector = NewVector(w.n_rows)

  for (r_idx in range(w.n_rows)):
    faccum: f32 = 0.0
    for (g_idx in range(groups_per_row)):
      iaccum: i32 = 0
      begin: i32 = g_idx * group_size;
      end: i32 = (g_idx + 1) * group_size;
      for (idx in range(begin, end)):
        // upcast to avoid overflow
        iaccum += i32(w[r_idx][idx])
                  * i32(x[idx])
      // scale result
      scale =  w.scales[r_idx][g_idx]
               * x.scales[g_idx]
      faccum += f32(iaccum) * scale
    out[r_idx] = faccum
  return out
```

Fig. 3. Quantized Matrix-Vector Multiplication

tized values and thus have to transfer less data to the CPU. Secondly, the majority of our operations are now integer operations, which have a lower instruction latency than floating-point operations.

It is important to note that there is still some amount of overhead involved in the quantized forward pass, as some operations like the RMS norm expect unquantized activation values and return unquantized values, which have to be (de)quantized on the fly. However, these operations are fairly cheap, as the activations are always one-dimensional vectors, and the quantization/dequantization process can be parallelized to a fairly high degree.

V. Results

One of our primary objectives in this paper was to investigate the computational processes involved in running inference on Large Language Models (LLMs), particularly for LLaMA2. As hinted in the previous section, we assume that most of our time will be spent on matrix-vector multiplications and that our computation will be memory-bound. To verify our claims and explore the impact of quantization on inference time, we conducted benchmarking experiments. Specifically, we examined which operations the model spends most of its time on, the achievable speedup through quantization, and the inference speed gain by running the model on a GPU. Additionally, to provide a reference point, we compared our simplistic implementation to llama.cpp, a high-performance LLM inference engine [12]. All our benchmarks were performed on an AMD Ryzen 5600, a 6-core, 12-thread processor with an AMD Zen4 architecture. Our GPU used for the benchmarks was an NVIDIA RTX 3070 with 8GB of VRAM running CUDA 11.5.

A. Comparison to llama.cpp

Since our implementation is relatively simplistic and lacks optimization beyond parallelization with OpenMP, we compared our implementation to a highly optimized LLM engine, namely llama.cpp.

TABLE I
Comparison of Prompt Evaluation Times

Model	Q4 (Tok/s)	Q8 (Tok/s)
llama.cpp	33,10	26,75
Ours	7.06	4.37
Speedup (%)	368.83 %	512.12%

As evident in Table I, llama.cpp significantly outperforms our implementation in prompt evaluation. This is expected, given that we perform prompt evaluation on a token-by-token basis, whereas llama.cpp utilizes matrix-matrix multiplications for prompt evaluation and can evaluate all prompt tokens in a single forward pass [9]. Notably, the prompt evaluation time for llama.cpp shows minimal differences between Q8 and Q4, further indicating that prompt evaluation is compute-bound, as discussed in Section IV.

TABLE II
Comparison of Text Generation Times

Model	Q4 (Tok/s)	Q8 (Tok/s)
llama.cpp	10.01	5.71
Ours	7.06	4.37
Speedup (%)	41.78 %	30.66%

For our implementation, the more relevant results are presented in Table II, which displays the text generation times. Here, llama.cpp shows a moderate performance advantage over our implementation, with a speedup of approximately 42% for Q4 and 31% for Q8. The disparity in speedup between Q8 and Q4 is likely attributed to the fact that, for larger models, the runtime is increasingly dominated by memory transfer speeds, which are hardware-dependent. Consequently, the optimizations in terms of runtime are less pronounced for larger models, as most of the computation time is spent on memory transfers, which should be equally fast for our implementation and llama.cpp.

B. Overall performance

To benchmark our overall performance, we did generate 200 tokens and measured the following metrics:

- (TTFT) Time to first token in seconds.
- (Tok/s) Tokens per second

It is to note that we use the unix syscall mmap() for lazy loading the weights, so the TTFT does not equal the total loading time of the weights, since not all weights have to be loaded to generate the first token. Nevertheless, we made

TABLE III
PERFORMANCE BY PRECISION WITH PERCENTAGE CHANGE

Type	TTFT (s)		Tok/s	
	Value	% Change	Value	% Change
F32	10.68	-	1.32	-
Q8	2.74	-74.34%	4.37	+231.06%
Q4	1.49	-45.62%	7.01	+60.41%
Q4 GPU	2.07	+38.93%	15.79	+125.25%

sure to flush the page cache before running the experiments to ensure a fair evaluation.

As shown in Table III, quantization leads to significant improvements in both TTFT and tokens per second Tok/s. We observe that removing 8 bits of precision corresponds to an inference speed gain of roughly 58-60%. Additionally, the table shows inference speed for our GPU implementation, which demonstrates impressive speedups compared to running on the CPU. At the same level of quantization, we see a 125% speedup in Tok/s. This speedup is likely due not only to the massively parallel execution of GPUs, but also to their use of high bandwidth memory (HBM), which achieves substantially higher bandwidth than standard DRAM.

C. Computational Hotspots

To gain a clear understanding of which operations consume the most inference time, we conducted benchmarks for every operation executed by LLaMA2 to generate a single token. Table IV shows the results, detailing the runtime of each kernel in milliseconds and the percentage of total runtime spent in each. Note that the Attention kernel does not include the matrix-vector multiplications for Q, K, V, and O, as these are counted towards the MatVec kernel.

This table shows some interesting results. In particular, inspecting the percentage of total runtime for all three precisions reveals that the smaller the model, the less the runtime is dominated by the MatVec kernel. This highlights that runtime is primarily dominated by memory transfer speed. Additionally, it's visible that the second most expensive kernel for the quantized forward pass is the quantization kernel itself, indicating a significant overhead in the quantization/dequantization process during model execution. Running the model on the GPU yields similar results to the CPU, although the quantization overhead is reduced, likely due to its highly parallelizable nature.

D. Quantization

As discussed in Section IV, we apply quantization to reduce the model's quality, achieving better performance. This is a common trade-off in LLM inference, where minimal quality loss can lead to significantly faster runtime speeds. However, it remains crucial for the models to produce accurate and useful results. Thus, no matter how rapid the processing, quality should not be overlooked.

In this chapter, we highlight the impact quantization has on the model's quality. A popular benchmark for evaluating LLM quality is the MMLU (Massive Multitask Language Understanding) [13]. This benchmark includes a wide variety of questions across different topics, making it well-suited to assess our model. However, conducting such a benchmark rigorously is beyond the scope of this project.

Instead, we selected one question from each of four categories: College Computer Science, College Physics, Global Facts, and Machine Learning. The choice of categories reflects our interest in these topics. For sampling, we employed the greedy method to ensure the reproducibility of our results.

The questions are:

1) **Global facts:** What is the median international income as of 2020?
   - Choices: ["$300", "$1,000", "$10,000", "$30,000"]
   - Answer: $10,000
2) **Computer Science:** Which of the following is the name of the data structure in a compiler that is responsible for managing information about variables and their attributes?
   - Choices: ["Abstract Syntax Tree (AST)", "Attribute Grammar", "Symbol Table", "Semantic Stack"]
   - Answer: Symbol Table
3) **Physics:**A meter stick with a speed of 0.8c moves past an observer. In the observer's reference frame, how long does it take the stick to pass the observer?
   - Choices: ["1.6 ns", "2.5 ns", "4.2 ns", "6.9 ns"]
   - Answer: 2.5 ns
4) **Machine Learning:** Given two Boolean random variables, A and B, where $P(A) = \frac{1}{2}$, $P(B) = \frac{1}{3}$, and $P(A \mid \neg B) = \frac{1}{4}$, what is $P(A \mid B)$?
   - Choices: ["$\frac{1}{6}$", "$\frac{1}{4}$", "$\frac{3}{4}$", "1"]
   - Answer: 1

The results of our benchmark are shown in Table V. From these results, there doesn't appear to be any performance degradation for the selected questions. However, upon inspecting the answers not just for their correctness, but also for their general quality, a significant drop-off is noticeable in the Q4 model. It's common for it to randomly produce German words; for instance, when attempting to generate the word "difference," it often produces the German word "unterschied," then veers off on a tangent. However, this issue typically lasts only for a few words, after which it returns to correctly predicting tokens.

For Q8 and F32, no noticeable differences are observed, likely because the model is trained on F16 values, reducing only by 8 bits rather than 24 as the names suggest.

VI. CONCLUSION OUTLOOK

The landscape of LLM inference is a highly active field, particularly as major tech companies are open-sourcing model weights for newer, larger, and more powerful models. The demand for running these models on one's own hardware continues to grow. In this paper, we explored LLaMA2, but by the time this work is completed, LLaMA3, Microsoft's Phi-3, and others will have been released. To stay within the scope of

TABLE IV
KERNEL TIMES

Kernel	Q4		Q4-GPU		Q8		F32	
	Time (ms)	% Total	Time (ms)	% Total	Time (ms)	% Total	Time (ms)	% Total
Total	**129.84**	100.00	**56.07**	100.00	**231.52**	100.00	**685.80**	100.00
RMS	0.09	0.07	1.34	2.39	0.09	0.04	0.11	0.02
MatVec	**118.18**	**91.02**	**50.89**	**90.76**	**218.75**	**94.48**	**683.30**	**99.64**
Quantize	**6.76**	**5.21**	**2.14**	**3.83**	**9.27**	**4.01**	-	-
Dequantize	0.92	0.71	0.02	0.03	0.23	0.10	-	-
Attention	2.51	1.93	0.57	1.02	1.81	0.78	0.93	0.14
Rotate	1.02	0.79	0.37	0.67	1.00	0.43	1.01	0.15
SiLU	0.26	0.20	0.19	0.35	0.27	0.11	0.27	0.04
Residual	0.03	0.03	0.38	0.68	0.04	0.02	0.05	0.01

TABLE V
QUESTION RESULTS

Question	Q4	Q8	F32
1	✓	✓	✓
2	✓	✓	✓
3	✗	✗	✗
4	✗	✗	✗

this project, we prioritized simplicity over the most optimized techniques for runtime speed and quality.

For example, advanced quantization techniques such as GPTQ [14] provide evidence that more sophisticated quantization methods lead to minimal performance degradation, even for up to 3-bit quantization. Another capability not explored in this paper, but often employed in practice, is splitting the model's encoder layers to run in a hybrid fashion across CPU and GPU. By offloading as many layers as possible onto the consumer GPU and keeping the remaining computation on the CPU, this provides a relatively simple technique to better utilize available hardware [12].

Furthermore various sampling strategies were explored such as greedy, top k, and top p. However, there are other strategies like beam search, which offers a balance between precision and diversity by exploring multiple paths simultaneously, selecting the most likely sequence from a set of candidates. This helps recover from an unfortunate token sampling that has a bad cumulative probability over multiple tokens, ensuring better long-term performance.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[2] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," 2023.

[3] B. Zhang and R. Sennrich, "Root mean square layer normalization," 2019.

[4] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[5] U. Jamil, "pytorch-llama," https://github.com/hkproj/pytorch-llama, 2024.

[6] N. Shazeer, "Glu variants improve transformer," *arXiv preprint arXiv:2002.05202*, 2020.

[7] C. Shi, H. Yang, D. Cai, Z. Zhang, Y. Wang, Y. Yang, and W. Lam, "A thorough examination of decoding methods in the era of llms," 2024.

[8] T. Pegolotti, E. Frantar, D. Alistarh, and M. Püschel, "Generating efficient kernels for quantized inference on large language models," in *Workshop on Efficient Systems for Foundation Models@ ICML2023*, 2023.

[9] J. Tunney, "llamafile," https://github.com/mozilla-Ocho/llamafile, 2024.

[10] A. Karpathy, "llama.c," https://github.com/karpathy/llama2.c, 2024.

[11] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale," *Advances in Neural Information Processing Systems*, vol. 35, pp. 30 318–30 332, 2022.

[12] G. Gerganov, "llama.cpp," https://github.com/ggerganov/llama.cpp, 2024.

[13] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," 2021.

[14] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," 2023.