Optimize and deploy large language model on laptop

Shuai Wang, sw23v
Florida State University
Tallahassee, USA
sw23v@fsu.edu

Ruoyu Li, rl13m
Florida State University
Tallahassee, USA
rli@cs.fsu.edu

Longge Yuan, ly23a
Florida State University
Tallahassee, USA
ly23a@fsu.edu

Abstract

This study addresses the challenge of deploying large language models (LLMs) on resource-constrained devices like laptops by implementing and evaluating advanced optimization techniques to improve inference speed. Specifically, we applied loop unrolling, multithreading, and SIMD (Single Instruction, Multiple Data) techniques, achieving a substantial reduction in inference time from 12.4 seconds per token to 1.4 seconds per token. This improvement renders the LLM's performance on laptops viable for real-world applications. Looking ahead, potential enhancements include utilizing CUDA for further leveraging GPU capabilities, which are well-suited for parallel processing tasks inherent in LLMs. Additionally, exploring loop tiling could optimize memory access patterns and reduce inference times even more effectively. Extending our optimization strategies to various LLM architectures will also be crucial in assessing the generalizability of our approach and identifying model-specific optimizations. This study not only demonstrates significant performance enhancements but also sets the stage for further innovations in the efficient deployment of LLMs on edge devices.

1 INTRODUCTION

1.1 Motivation

To provide a deeper exploration into the motivation behind deploying large language models (LLMs) like GPT on edge devices such as laptops, it's imperative to analyze the specific challenges and opportunities that this technological shift entails. The migration from centralized cloud servers to decentralized edge computing environments represents a significant paradigm shift in the field of artificial intelligence and machine learning. This transition is driven not just by the technological advancements that enable such deployments, but also by a confluence of factors that address both technical limitations and broader socio-economic impacts.

- **Deeper Dive into Technical Challenges and Solutions**
  The deployment of LLMs on edge devices introduces several technical challenges that need to be addressed to ensure successful implementation. These include managing the computational demands of LLMs with the relatively limited processing power and memory of laptops compared to cloud servers. Innovations in model compression techniques, such as quantization and pruning, have made it feasible to reduce the size of these models without significantly compromising their performance. Further, the development of specialized hardware accelerators for machine learning tasks, such as GPUs and TPUs available in consumer devices, helps in bridging the gap between the computational requirements of LLMs and the capabilities of edge devices.

- **Exploring the Connectivity Aspect**
  Another critical aspect of edge deployment is its ability to operate independently of a constant internet connection. This capability is crucial in environments where connectivity is unreliable or expensive. Edge computing allows LLMs to function autonomously, making real-time decisions without the need to query a distant server. This not only enhances the reliability and speed of applications but also opens up new possibilities for AI applications in remote and underserved areas, thus broadening the impact of advanced technologies.

- **Addressing Privacy Concerns More Rigorously**
  Privacy concerns are particularly acute in the context of AI, as models often need to process sensitive data. By processing data locally on an edge device, the exposure of sensitive information to external networks is minimized, significantly enhancing data security. Local data processing aligns with global trends toward stricter data privacy regulations, offering a strategic advantage by inherently designing systems that prioritize user privacy and data security.

- **The Bigger Picture: Sustainable AI and Ethical Considerations**
  Resource optimization in edge computing doesn't only refer to computational resources but also encompasses network bandwidth and energy consumption. By localizing data processing, significant reductions in data transmission can be achieved, which not only saves bandwidth but also reduces the energy footprint associated with data transfer over vast network infrastructures. Furthermore, this model promotes a more sustainable approach to deploying AI systems by decreasing reliance on large, centralized data centers, which

are energy-intensive and contribute significantly to the operational costs.

- **Latency and Real-time Processing**
  The shift towards edge deployment also resonates with broader objectives of sustainability and ethical responsibility in technology deployment. It encourages the development of systems that consume fewer resources and generate less carbon footprint. Moreover, by processing data locally, edge AI fosters greater transparency and user control over their data, a fundamental aspect of ethical AI practices.

In conclusion, expanding the deployment of LLMs onto edge devices such as laptops is not merely a technical challenge; it's a multifaceted endeavor that addresses essential aspects of functionality, accessibility, privacy, and sustainability. The evolution towards edge computing signifies a more inclusive, secure, and environmentally conscious approach to AI deployment, ultimately enhancing the interaction between humans and intelligent systems in myriad everyday contexts. This detailed exploration into the motivations behind edge deployment elucidates the profound impacts and transformative potential of localizing AI processing on personal devices.

1.2 Improvement

To further improve the efficiency and performance of large language models (LLMs) such as GPT deployed on edge devices such as laptops, several advanced techniques can be leveraged. These techniques include utilizing caching mechanisms, adopting parallel computing strategies, and using compression and quantization models. Let's try using the following function. Each of these approaches plays a crucial role in optimizing LLM deployment for edge computing.

1.2.1 Leverage caching and reduce overhead.

Leveraging advanced programming techniques such as loop reordering, loop tiling, and loop unrolling can further enhance the efficiency of caching strategies and reduce computational overhead when deploying large language models (LLMs) on edge devices like laptops. These techniques are primarily used to optimize the execution of loops, which are common in the processing tasks associated with LLMs, especially in the context of data preparation, model training, and inference.

- **Loop Reordering**
  Loop reordering involves rearranging the order of nested loops to optimize memory access patterns and enhance cache performance. By adjusting the loop order, people can ensure that data accessed in memory is as close as possible to previously accessed data, which maximizes cache hits and minimizes cache misses. For instance, if data is stored in a row-major format, reordering loops to process data row-wise rather than column-wise can lead to significant improvements in cache utilization. This can reduce the number of cache lines loaded, decrease memory bandwidth usage, and speed up the overall execution time.

- **Loop Tiling**
  Loop tiling, also known as loop blocking, is another effective technique used to optimize memory usage and improve cache performance. This method involves dividing a loop's iteration space into smaller, more manageable blocks or "tiles". These tiles are sized so that the data processed by one tile fits into the cache, preventing the cache from being flushed between iterations. This reduces the latency associated with accessing data from main memory and can significantly boost performance, especially in large-scale computations typical of LLM operations. Loop tiling not only helps in improving cache efficiency but also can be tailored to the specific architecture of a device, allowing for customized performance enhancements.

- **Loop Unrolling**
  Loop unrolling is a technique to enhance execution speed by reducing the overhead of loop control instructions (such as incrementing and condition checking). By replicating the loop body multiple times and decreasing the frequency of loop control checks, loop unrolling minimizes the computational overhead associated with each iteration. Although this increases the size of the code, it can lead to faster execution times by enabling more efficient use of instruction pipelines and reducing the number of jumps or branch instructions. When combined with vectorization, where multiple data points are processed simultaneously using single instruction, multiple data (SIMD) capabilities of modern processors, loop unrolling can substantially speed up data processing tasks.

By incorporating these loop optimization techniques loop reordering, loop tiling, and loop unrolling—into the deployment strategy for LLMs on laptops, developers can significantly reduce the computational overhead and enhance the performance of these models. These optimizations help in making more efficient use of the device's cache and processing power, which is crucial for running advanced AI models in resource-constrained environments. Additionally, when these techniques are applied thoughtfully, taking into account the specific characteristics and limitations of the hardware, they can lead to a noticeable improvement in the

responsiveness and efficiency of LLM applications on edge devices.

1.2.2 Parallel Computing.

Using SIMD (Single Instruction, Multiple Data) and multithreading in parallel computing significantly enhances the efficiency and performance of large language models (LLMs) when deployed on edge devices like laptops. These techniques leverage the capabilities of modern processors to execute multiple operations simultaneously, making them extremely valuable for handling the computationally intensive tasks associated with LLMs.

- **SIMD: Boosting Data Processing Efficiency**
  SIMD is a type of data parallelism that allows a single instruction to process multiple data points simultaneously. This is particularly useful for the matrix operations common in neural network computations, where the same operation is repeatedly applied across large datasets. By employing SIMD operations, CPUs and GPUs can dramatically accelerate vector and matrix arithmetic, a core component of the computational workload for training and inference in LLMs.

  Modern processors, including those found in laptops, often come equipped with SIMD instruction sets like Intel's AVX (Advanced Vector Extensions) or ARM's NEON. These extensions can handle multiple data points per clock cycle, which speeds up the overall computation process by several folds. When SIMD is effectively utilized, it can reduce the cycle count required to process data, decrease latency, and increase throughput, all of which are crucial for the performance of LLMs in real-time applications.

- **Multithreading: Maximizing Core Utilization**
  Multithreading, on the other hand, involves the execution of multiple threads of execution concurrently, making use of the multiple cores of modern CPUs and GPUs. This approach allows different parts of a program to run in parallel, thereby improving the overall efficiency of the application. For LLMs, multithreading can be used to parallelize different operations such as data loading, preprocessing, and different parts of the model's computation.

  By distributing the tasks of an LLM across multiple threads, people can make better use of the available computational resources. For instance, while one thread handles input/output operations, another could perform model updates, and yet another could manage asynchronous data preprocessing. This division of labor helps in reducing idle time for CPU or GPU resources, ensuring that the device's capabilities are fully utilized, leading to faster processing times and improved responsiveness of the model.

- **Combining SIMD and Multithreading**
  When SIMD and multithreading are combined, they provide a robust solution for optimizing the performance of LLMs on laptops. SIMD handles the efficient execution of data-parallel tasks within a single thread, while multithreading spreads the computational load across multiple threads and cores. This dual approach optimizes both the execution of individual instructions and the overall operation management, leading to a highly efficient parallel computing environment.

For developers looking to deploy LLMs on edge devices, leveraging both SIMD and multithreading is key to achieving high performance. It allows the model to handle more computations in less time, making it feasible to run sophisticated AI applications directly on user-end devices without significant delays. This capability is essential for applications requiring real-time or near-real-time processing capabilities, such as interactive AI systems, mobile applications, and on-device AI solutions where speed and efficiency are paramount.

1.2.3 Compression and quantization models.

Neural architecture search (NAS) is a powerful method to optimize neural network designs, including finding the most efficient architectures for compressed and quantized models. By automating the process of designing neural network architectures, NAS can significantly enhance the performance and efficiency of large language models (LLMs) deployed on edge devices like laptops, especially when resources are limited.

- **NAS for Model Compression and Quantization**
  When applying NAS for model compression and quantization, the search is specifically tailored to identify architectures that maintain high accuracy while being inherently smaller and faster to compute. Compression techniques like pruning and knowledge distillation can be integrated into the NAS process to evaluate architectures that naturally require fewer parameters or less computational power. For example, NAS can discover sparse architectures that eliminate unnecessary connections and weights, achieving a lighter model without extensive manual tuning.

Quantization can also be integrated into the NAS framework. During the architecture search, NAS can evaluate how different architectures perform when their weights and activations are quantized to lower-bit representations. This is

crucial because some architectures might be more robust to quantization than others, maintaining higher accuracy when using reduced precision. NAS can thus help find optimal architectures that are quantization-friendly, ensuring that the performance degradation often associated with quantization is minimized.

Together, these three strategies form a powerful approach to optimizing the deployment of large language models on edge devices. By solving the challenges associated with limited resources, these technologies ensure that the LLM can be effectively and efficiently used in a variety of applications directly from the user's laptop, thereby improving performance and user experience. In terms of results, we reduced the average inference speed of LLM on a laptop from 12.4 seconds per token to 1.4 seconds per token. This reduces the speed by nearly 10 times.

2 RELATED WORK

2.1 Large Language Model Meta AI

LLaMA, developed by Meta AI, represents a noteworthy advancement in the realm of large language models. According to the fig 7 [14], as an LLM, LLaMA has been designed to function effectively across a range of sizes, making it particularly relevant for studies involving model scaling and efficiency—key considerations when deploying models on edge devices. LLaMA ranges parameters from 7B to 65B. LLaMA-13B outperforms GPT-3 in most benchmarks despite being 10 times smaller.[14] LLaMA's architecture and training methodologies provide a useful benchmark for discussing the compression and optimization of LLMs for limited-resource environments such as laptops and other personal computing devices.

Architecture and Performance: LLaMA is built on a transformer architecture, similar to other prominent models like GPT and BERT, but with optimizations that potentially reduce computational overhead without a substantial loss in performance.[14] This makes it an exemplary model for exploring ways to balance between model complexity and efficiency, a crucial aspect when dealing with hardware limitations on edge devices.

Model Scaling: One of the remarkable aspects of LLaMA is its scalability. The model has been trained at various scales, providing insights into the trade-offs between model size, computational demands, and performance. LLaMA's pretraining data consists mainly of web pages (over 80%), with another 6.5% code-intensive data from GitHub and StackExchange, 4.5% from books, and 2.5% scientific data from arXiv, which has become the basis for training general large language models (LLMs).[18] This scaling characteristic is

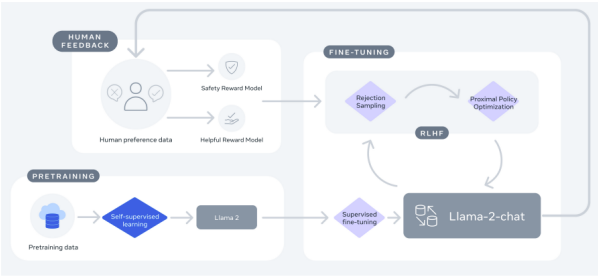

Figure 1. Multi-threading

instrumental for applications on edge devices, where the available computational resources can vary significantly.

Research and Development Implications: The development of LLaMA has spurred a variety of research initiatives focused on improving the efficiency and applicability of LLMs in real-world scenarios. Studies based on LLaMA have explored issues such as reducing latency, enhancing data privacy, and minimizing energy consumption—each of which is critical for successful edge deployment.[16]

Comparison with Other Models: In the broader landscape of LLMs, LLaMA can be contrasted with models like GPT-3, BERT, and others that are typically used in cloud-based environments. LLaMA with 13B parameters and more outperforms other language models.[14] Comparable performance to GPT-4 is achieved without reinforcement learning.[17] The comparative analysis of these models in terms of performance, efficiency, and deployment feasibility on edge devices can provide deeper insights into the optimal strategies for model deployment in constrained environments.

2.2 Quantization

Quantization is a well-established technique in the field of machine learning that enhances model inference efficiency by reducing the computational complexity and resource requirements of neural networks. [6] By converting floating-point values into a lower-precision integer format, quantization decreases the model's memory footprint and speeds up the computation, thus enabling faster inference times with minimal impact on accuracy. This method is particularly valuable for deploying advanced machine learning models like LLMs on edge devices, where computational resources are limited.

Quantization Techniques: Various approaches to quantization have been studied and implemented, ranging from post-training quantization (PTQ) to quantization-aware training (QAT). PTQ involves applying quantization to a pre-trained model without the need for retraining, which is straightforward and quick to implement. On the other hand,

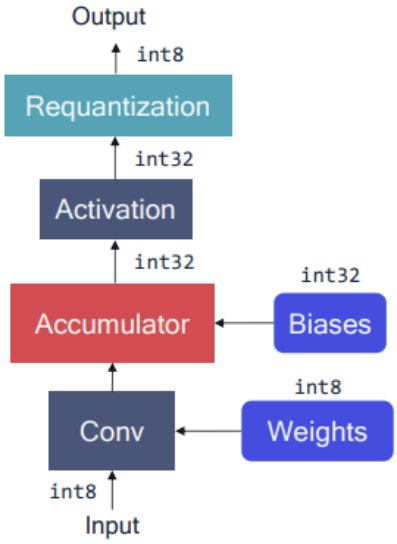

Figure 2. Simulated quantized on-device inference

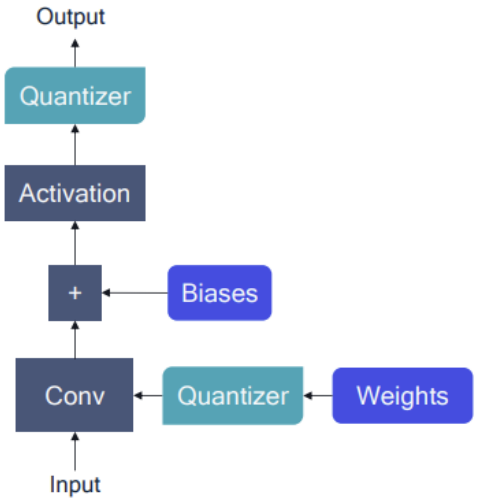

Figure 3. Simulated quantization by using floating-point operations

QAT incorporates quantization directly into the training process, which often results in better model accuracy as the model learns to adjust to the reduced precision during its training. Moving from floating-point representation to low-precision fixed integer values represented with four or fewer bits has the potential to reduce memory footprint and latency by a factor of 16.[6] Both techniques are essential in the context of deploying LLMs on laptops and other edge devices, as they provide options depending on the available resources and required model accuracy.

Applications in Edge Computing: The application of quantization is particularly relevant in the context of edge computing. By reducing the demands on memory and processing power, quantized models can run more efficiently on edge devices such as smartphones, IoT devices, and laptops. This capability is critical for applications requiring real-time processing and decision-making where sending data to the cloud for inference would be too slow or impractical. In the context of the Internet of Things, if a large amount of data generated by connected devices is all transmitted to the cloud, cloud computing will cause a huge load.[4]

Comparison and Integration with Other Techniques: In addition to standing alone as a method to reduce model size and speed up inference, Figure 2 [11] quantization is often used in conjunction with other optimization techniques such as pruning and knowledge distillation. Quantization generally outperforms pruning for neural networks. Figure 3 [11]They allow users to efficiently test various quantization options and support GPU acceleration for quantization-aware training[8]This integrated approach maximizes the efficiency of LLMs, making it feasible to deploy sophisticated AI models in constrained environments without significant loss of functionality. Such simulations are significantly easier to implement than running experiments on actual quantization hardware or using quantization kernels.

Related Research and Developments: The ongoing research in quantization continues to push the boundaries of what is possible with this technique. Innovations such as dynamic quantization and adaptive quantization are being explored, which adjust the precision of calculations based on the specific needs of the task at hand. These developments show promise in further bridging the gap between model performance and computational efficiency.

2.3 Parallel Computing

Parallel computing has emerged as a fundamental strategy for maximizing the computational efficiency of LLMs, particularly when deployed on laptops and other edge devices. This approach leverages multiple computing resources simultaneously to speed up the processing of large-scale machine learning tasks, which is crucial for real-time applications and energy-efficient computing.

Cache locality: Improving cache locality is a key consideration in parallel computing, especially for memory-intensive applications such as LLM. Efficient cache management ensures that data is stored closer to the processor, reducing the time and energy consumed in retrieving data from main memory. [1] A figure 5 is a canonical processor without a pipeline. It takes five clock cycles to complete an instruction. [3] we employs techniques such as loop tiling

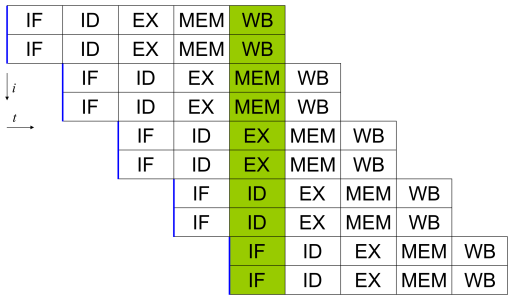

Figure 4. A canonical five-stage pipelined processor

Figure 5. A canonical processor without pipeline

and data reorganization to enhance cache locality, thereby minimizing cache misses and reducing latency. [1] Figure 4 is a canonical five-stage pipeline processor. In the best case, it takes one clock cycle to complete an instruction, so the processor can deliver scalar performance. Each level of the five-stage pipeline processor's storage hierarchy organizes its data in blocks to simplify management and reduce overhead. This is critical for edge devices where memory bandwidth and speed are limited.

- **Temporal locality**
  Temporal locality refers to the repeated use of specific data or resources within a relatively short period of time. In the context of an LL.M., if specific weights or data points are used in calculations, it is likely that the same project will be required again soon. By keeping frequently accessed data in cache, the processor reduces the number of time-consuming memory read operations. This is particularly beneficial for LLMs in tasks such as iterative optimization algorithms, where the same parameters are repeatedly adjusted over multiple training cycles.

- **Spatial Locality**
  Spatial locality, on the other hand, refers to the use of data elements within close storage locations. In neural networks, data is often accessed in predictable patterns. For example, when processing common matrix operations in deep learning, adjacent matrix elements are usually processed together. By storing contiguous blocks of data in cache, spatial locality minimizes the overhead of fetching data from main memory, thereby increasing the execution speed of matrix calculations.

Data-level Parallelism: Data-level parallelism, particularly through SIMD (Single Instruction, Multiple Data) operations, allows a single instruction to be performed on multiple data points simultaneously. This is particularly effective for the vector and matrix operations that are common in neural network computations. In mainstream CPUs, vectorization is provided by a large number of powerful SIMD extensions, which continue to grow not only in vector size, but also in the complexity of the instruction sets provided. [7]By leveraging SIMD, processors can dramatically increase throughput, reduce processing times, and decrease power consumption—all of which are vital for the efficient operation of LLMs on edge devices.

Multithreading: Multithreading takes advantage of the multiple cores typically available in modern processors to perform various computational tasks in parallel. By distributing different aspects of LLM inference across multiple threads, the workload is balanced more effectively, which can lead to significant improvements in performance. Data prediction is used to alleviate dependency constraints and enable lookahead execution of the threads.[2] This is especially useful in scenarios where multiple users or processes need to access the AI model simultaneously, as it allows the model to handle multiple requests without bottlenecking.

Integration with LLM Deployment: The application of parallel computing techniques in the deployment of LLMs is crucial for optimizing both the speed and energy efficiency of these models on edge devices. Scaling up transformer-based language models in terms of model size, training data, and training compute has been shown to predictably improve performance on a wide range of downstream NLP tasks. [10]Researchers and developers focus on designing parallel algorithms and infrastructure that minimize computational overhead, enhance real-time processing capabilities, and maintain model accuracy even under constrained conditions.

3 METHOD

3.1 TinyChat Engine

As indicated by the literature in the field, numerous endeavors have been made to deploy and accelerate LLM models on edge devices, resulting in significant advancements. From the array of available frameworks, we have selected TinyChatEngine [13] as our foundational framework for developing optimization techniques aimed at accelerating the inference speed of large language models.

TinyChatEngine is an advanced inference engine designed specifically for LLMs. It offers broad platform compatibility, supporting x86 architecture (Intel/AMD), ARM architecture

(Apple M1/M2, Raspberry Pi), as well as CUDA for Nvidia GPUs. This wide range of compatibility ensures that the engine can be seamlessly deployed across various devices and systems, catering to diverse computing environments.

One of the notable advantages of TinyChatEngine is its independent library design. Developed from scratch in C/C++, the engine doesn't rely on any external dependencies, making it highly self-contained and efficient. This not only simplifies the deployment process but also reduces the chances of compatibility issues arising from conflicting libraries or versions.

TinyChatEngine stands out for its impressive performance, capable of achieving real-time inference on popular devices such as Macbook and GeForce laptops. This high-performance capability enables smooth and responsive interactions with the LLM, ensuring quick and efficient generation of responses.

Additionally, TinyChatEngine boasts a user-friendly approach. With a straightforward setup process, users can easily download the engine, compile it, and start utilizing it immediately. This simplicity allows developers and researchers to focus on their tasks without being burdened by complex installation procedures or intricate configurations.

Also, the TinyChatEngine relied on TinyEngine [9], a lightweight inference engine. TinyEngine represents a significant advancement in comparison to existing inference libraries, primarily due to its innovative compilation method based on code generation. This approach effectively eliminates memory overhead, resulting in more efficient memory utilization during the inference process.

Furthermore, TinyEngine introduces model-adaptive memory scheduling, departing from the traditional layer-wise optimization approach. Instead, it analyzes the overall network topology and devises a tailored memory scheduling strategy that maximizes efficiency and performance. This holistic approach ensures a more optimized and effective memory management system.

Moreover, TinyEngine incorporates specialized computation kernel optimization techniques, specifically tailored to different layers of the model. By applying these optimizations, TinyEngine further enhances the inference speed, providing a significant boost in performance.

In summary, TinyEngine distinguishes itself from existing inference libraries through its code generator-based compilation method, memory scheduling based on network topology, and specialized computation kernel optimizations. These advancements collectively result in improved inference efficiency, reduced memory overhead, and accelerated inference speed for large language models.

The TinyEngine offers convenient utilization of pre-configured techniques or the implementation of custom optimization techniques to accelerate inference. Here are some examples:

- In-place depth-wise convolution: This unique data placement technique for depth-wise convolution overwrites input data with intermediate/output data, effectively reducing peak SRAM memory usage.
- Patch-based inference: A versatile scheduling approach that performs inference patch-by-patch, focusing only on a small spatial region of the feature map. This method significantly reduces peak memory requirements.
- Operator fusion: An optimization method that enhances performance by merging one operator into another, allowing them to be executed together without the need for a roundtrip to memory.
- SIMD (Single instruction, multiple data) programming: A computational technique that performs identical operations on multiple data points simultaneously, maximizing efficiency.
- HWC to CHW weight format transformation: A technique for transforming weight formats, increasing the cache hit ratio specifically for in-place depth-wise convolution.
- Image to Column (Im2col) convolution: An implementation technique that computes convolution operations using general matrix multiplication (GEMM) operations. This approach improves computational efficiency.
- Loop reordering: A loop transformation technique that optimizes a program's execution speed by rearranging or interchanging the sequence of loops.
- Loop unrolling: A loop transformation technique that improves a program's execution speed at the cost of larger binary size. This approach involves a space-time tradeoff.
- Loop tiling: A loop transformation technique that aims to reduce memory access latency by partitioning a loop's iteration space into smaller chunks or blocks. This helps ensure that data used in a loop remains in the cache until it is needed again.

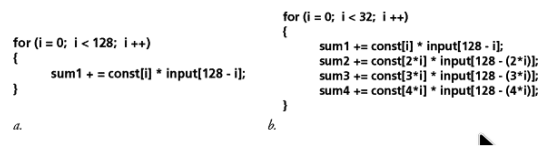

Figure 6. Loop unrolling

By leveraging TinyChatEngine as weel as TinyEngine, we aim to enhance the performance and efficiency of our targeted model in order to achieve faster inference times. This choice aligns with the broader trend of optimizing language models for deployment on edge devices, ensuring that our accelerated model can deliver prompt and responsive results.

3.2 Our Goal (Research Questions)

Since matrix multiplication forms the core of the inference computation, our focus is on techniques to accelerate its execution.

Therefore, our research question is as follows: Given a pre-quantized large language model (LLM) within an edge-based inference framework (TinyChat Engine), our objective is to implement system acceleration techniques that reduce inference latency. We also aim to compare the effects of each technique on the overall performance. Here are the techniques we employed.

Firstly, we optimized cache locality and minimized branching overhead through loop unrolling. In matrix multiplication, the loop control introduces significant overhead, particularly when checking terminating conditions. We unrolled the loop in the code and reduce the associated overheads, like cache misses.

Also, by utilizing SIMD (Single Instruction, Multiple Data), we took advantage of data-level parallelism. This approach includes executing the same CPU instruction on different data concurrently. We assigned the matrix's values to vector registers and utilizing CPU-specific operations to perform computations together. In our experiment, because we used macbook, we implemented the Arm version of SIMD.

Finally, we divided the metric into smaller components and employed the Pthreads library to execute all optimizations in parallel. This parallelization approach facilitated concurrent execution of the optimization techniques we mentioned before.

3.3 Loop Unrolling

Loop unrolling, or loop unwinding, as showed in Figure 6, is a technique for modifying loops that aims to enhance a program's running speed at the cost of increasing its binary size, a concept referred to as the space-time tradeoff. This transformation can be performed either by the programmer manually or automatically by an optimizing compiler. However, on contemporary processors, loop unrolling can sometimes be detrimental, as the larger code size may lead to more cache misses

The objective of loop unwinding is to accelerate a program by minimizing or removing loop management instructions, like pointer calculations and "end of loop" checks for each cycle[15]. This technique also aims to lessen branching penalties and mask delays, such as those associated with data retrieval from memory[12]. To reduce this overhead, loops can be reformulated as a series of repeated, similar, and independent statements.

Through Loop unrolling, we can achieve following improvement:

- Notable performance improvements are attainable when the fewer instructions executed outweigh any negative impact on performance due to the program's larger size.
- This approach also leads to a reduction in branch penalties[5].
- Operations within a loop that do not depend on each other can be executed in parallel to enhance efficiency.
- This technique can be dynamically applied in situations where the array size is not known at compile time, as exemplified by Duff's device.

However, Loop unrolling may also causes server drawbacks as following:

- A larger program size, which can be problematic especially in embedded systems, may lead to more instruction cache misses and negatively impact performance
- The clarity of the code might suffer if loop unrolling is not automatically done by a compiler.
- When loops contain function calls, merging loop unrolling with function inlining could be impractical due to a potentially substantial increase in code size, presenting a dilemma between two optimization strategies.
- There's a potential for increased usage of registers for temporary data within a loop iteration, which might hinder performance, although this is subject to further optimization techniques.
- For complex code beyond simple loops, unrolled loops with branching may be less efficient than recursive implementations.

On the one hand, since we use loop unrolling to perform tight matrix calculations, we can fully utilize the advantages

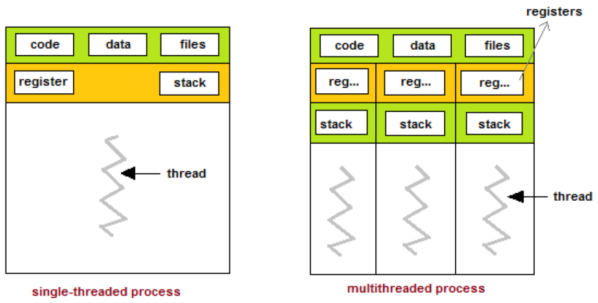

Figure 7. Multi-threading

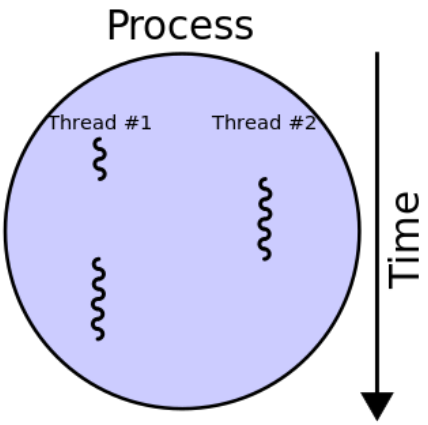

Figure 8. A process with two threads of execution, running on a single processor.

of loop unrolling; on the other hand, the experimental environment is fixed and the independence between each calculation is high, which can effectively avoid loop unrolling. disadvantages. So overall, loop unrolling was chosen due to its good compatibility in our project.

3.4 Multi Threading

Multithreading, as showed in Figure 7, in the realm of computer architecture refers to the capability of a central processing unit (CPU), or just one core within a multi-core processor, to execute several threads at the same time, with this functionality being facilitated by the operating system. This concept is distinct from multiprocessing. Within a multithreaded program, the threads concurrently utilize the resources of one or more cores, which encompasses the computational units, the CPU's cache system, and the translation lookaside buffer (TLB).

Contrasting with multiprocessing systems, which have several complete processors within single or multiple cores, multithreading seeks to enhance the efficiency of a singular core by enabling parallelism at both the thread level and the instruction level. Since these two methods are synergistic, modern computing systems typically merge them, utilizing CPUs that support multithreading on their multiple cores.

Since the late 1990s, advancements in leveraging instruction-level parallelism have plateaued, leading to a resurgence in the adoption of multithreading approaches. This shift has allowed the principle of throughput computing to gain traction beyond its initial niche within transaction processing. Despite the challenges in accelerating the performance of an individual thread or program, most computer systems are routinely engaged in juggling multiple threads or programs simultaneously. Therefore, methods that elevate the throughput across the spectrum of tasks can contribute to improvements in the system's overall performance. And figure has illustrates a simple 2-threads process in time sequence 8.

Multi-threads has several advantages as below:

- If one thread experiences a high number of cache misses, other threads can continue to utilize the unused computing resources, potentially speeding up overall execution since these resources would otherwise remain idle.
- If a thread cannot fully utilize the CPU's resources due to interdependent instructions, running additional threads can help in keeping those resources from going idle.

Multi-threads has some disadvantages as below:

- Multiple threads may interfere with each other when they share hardware resources like caches or translation lookaside buffers (TLBs), which can worsen the performance of individual threads due to increased contention.
- Execution times for a single thread may not improve and can even degrade, potentially affected by lower operational frequencies or additional pipeline stages required to support thread-switching hardware.
- The overall efficiency of multithreading varies widely. For instance, Intel's Hyper-Threading Technology claims up to a 30% improvement, but the actual gain can depend heavily on the nature of the tasks being run.
- Hardware multithreading might not benefit software that is highly optimized, such as hand-tuned assembly programs that use SIMD extensions and perform data prefetches. These programs may actually see a decrease in performance due to competition for shared resources.
- From a software perspective, hardware support for multithreading is more visible and demands significant changes in application programs and operating

systems, paralleling the software techniques for computer multitasking. Thread scheduling becomes a critical issue in managing multithreading.

Based on the above advantages and disadvantages, multithreading can be well adapted to our matrix calculation tasks and give full play to its strengths.

In terms of multithreading, OpenMP(Open Multi-Processing), as an Application Programming Interface (API) that supports multi-platform shared-memory multiprocessing programming, provides an efficient way to leverage the computational power of modern multi-core processors. It employs a multithreading approach where a primary thread (the thread executing a sequence of instructions) forks a specified number of sub-threads and distributes tasks among them. These threads then run concurrently, with the runtime environment allocating threads to different processors as needed. OpenMP's structure is shown in 9

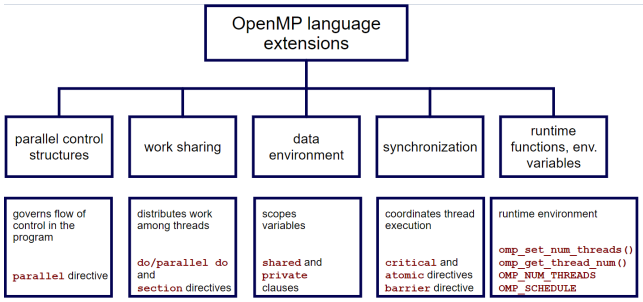

Figure 9. Chart of OpenMP constructs.

OpenMP's multithreading model is based on the fork-join model, in which sections of code intended for parallel execution are marked with compiler directives to ensure multiple threads are formed before the section executes. Each thread has a unique ID that can be accessed using a function (such as omp_get_thread_num()). After the parallelized code has been executed, all sub-threads rejoin the primary thread, which continues to the end of the program.

Furthermore, OpenMP supports work-sharing constructs, allowing developers to specify how independent tasks are assigned among multiple threads. For instance, the "omp for" or "omp do" can be used to distribute loop iterations across threads, and "sections" can assign separate but independent blocks of code to different threads.

These features make OpenMP a powerful tool for developing multithreaded applications, especially when rapid development of parallel processing capabilities is needed in areas like scientific computing, engineering simulations, and data analysis.

3.5 SIMD

Single Instruction, Multiple Data (SIMD) is a parallel processing paradigm identified in Flynn's Taxonomy, designed to perform the same operation on multiple data points simultaneously. This architecture is incorporated directly into hardware and can be accessed through instruction set architectures (ISAs). SIMD processors are used to enhance performance in operations common in multimedia tasks, such as adjusting image contrast or audio volume. Modern CPUs and GPUs often include SIMD instructions to facilitate efficient multimedia processing. 10.

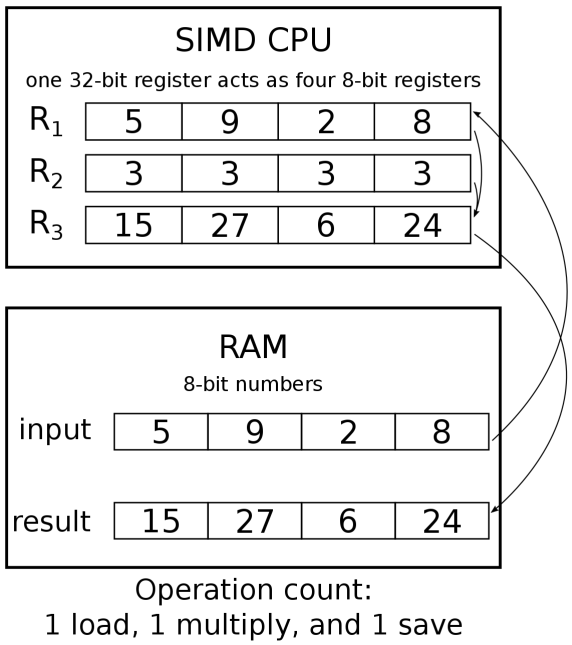

Figure 10. A simple demo of SIMD.

The primary advantage of SIMD is its ability to process multiple data points simultaneously, significantly speeding up tasks where the same operation is applied to large volumes of data, such as pixel adjustments in images or simultaneous channel adjustments in audio processing. This parallel data handling capability allows SIMD to process blocks of data in one go, which is more efficient than processing each element sequentially. This method reduces the time required for data retrieval and operation execution, thereby improving overall system performance for suitable tasks.

Despite its benefits, SIMD is not universally applicable to all types of computing tasks. Some algorithms, particularly those involving heavy control flow like code parsing, do not lend themselves well to vectorization. Programming with SIMD can involve complex challenges, including managing data alignment and dealing with architecture-specific instruction sets and register sizes. Furthermore, implementing SIMD often requires manual effort, as most compilers do not automatically generate SIMD instructions from standard code. The need for multiple implementations to cover various architectures and the large register files also increase power consumption and chip area requirements.

According to those features, due to the condition that the project is running in same machine with defined structure ARM, if we can achieve SIMD properly, it would be a useful technique.

In addition, since there are two main different architecture ARM and X86, SIMD has two different systems:

1. Instruction Sets
   - ARM: ARM architecture uses a SIMD technology called NEON, also referred to as "Advanced SIMD". NEON provides SIMD instructions for both floating-point and integer data, allowing multiple data operations in a single instruction cycle. NEON is typically used in mobile and embedded systems and supports both 64-bit and 128-bit SIMD operations.

   - x86: In the x86 architecture, SIMD instruction sets started with MMX and evolved into the SSE (Streaming SIMD Extensions) series, and now to the current AVX (Advanced Vector Extensions). AVX, and the more advanced AVX-512, offer wider vector units (up to 512 bits), allowing larger scale parallel data processing.

2. Hardware Support
   - ARM: NEON is supported across a variety of ARM processors, especially in the Cortex-A series. Due to the diversity of the ARM architecture, the implementation and support of NEON can vary by specific processor models.

   - x86: Nearly all modern x86 processors support at least one form of SIMD instruction sets, from early SSE to the latest AVX-512. However, it's noteworthy that AVX-512 is not supported by all modern x86 processors; it's usually available only in high-end and the most recent processors.

3. Performance and Cost
   - ARM's NEON: NEON is optimized for low-power environments, where power consumption is a crucial consideration for mobile devices. NEON is designed with efficiency and energy control in mind.

   - x86's SSE, AVX: x86 SIMD instruction sets like SSE and AVX are designed for higher performance applications, commonly seen in personal computers, servers, and high-performance computing environments. AVX-512, in particular, provides extremely high data throughput, suitable for applications demanding intense computational resources.

Since our project is based on ARM, the performance is not as good as X86 in speed. Thereby, the deployment of LLM would be more challenging.

4 EXPERIMENT

4.1 Running Environment

All of the experiment was run in MAC as below:

- Model Name: MacBook Pro 2023 model
- CPU Chip: Apple M3 Pro
- Total Number of Cores: 11 (5 performance and 6 efficiency)
- Memory: 18 GB
- Type: LPDDR5
- Metal Support: Metal 3

And all of the improvement related coding is done in QM ARM, written in C++.

ARM is an architecture for processor architectures that is different with X86. ARM (Advanced RISC Machine) architecture is a reduced instruction set computing (RISC) architecture primarily used in mobile devices, embedded systems, and low-power devices. It focuses on energy efficiency and performance-per-watt. ARM processors are known for their power efficiency, allowing for longer battery life in portable devices. They are designed to execute multiple instructions simultaneously, enabling efficient multitasking and parallel processing. ARM-based systems often excel in power-constrained scenarios and are commonly found in smartphones, tablets, and IoT devices. X86 architecture, also known as the Intel architecture, is a complex instruction set computing (CISC) architecture commonly used in desktop and laptop computers. It emphasizes performance and compatibility with a wide range of software. x86 processors are designed for high performance and are commonly found in personal computers and servers. They offer extensive instruction sets and sophisticated features, making them suitable for demanding applications such as gaming, multimedia editing,

and intensive computational tasks.

Apple Silicon is Apple's custom-designed system-on-a-chip (SoC) based on the ARM architecture. It offers several advantages:

- Performance and Efficiency: Apple Silicon processors are known for their excellent balance between performance and power efficiency. They leverage advanced manufacturing processes and custom-designed cores to deliver high-performance computing while consuming less power, resulting in improved battery life.
- Integration: Apple Silicon integrates multiple components, including the CPU, GPU, Neural Engine, and other specialized accelerators, onto a single chip. This integration enhances performance, reduces latency, and enables efficient communication between different subsystems, resulting in improved overall system performance.
- Unified Ecosystem: With Apple Silicon, Apple aims to create a unified ecosystem across its devices. Applications designed for ARM-based systems, such as those developed for iPhones and iPads, can run natively on Apple Silicon Macs without requiring emulation or translation. This compatibility streamlines software development and allows for seamless integration across different Apple devices.

LPDDR5 (Low Power Double Data Rate 5) is the latest generation of mobile DRAM (Dynamic Random Access Memory) technology.

```
for (int ch = 0; ch < k;) {
    // pointer of the int8 activation
    const signed char *a_int8 = &A->int8_data_ptr[row * k + ch];
    // pointer of the int4 weights
    uint8_t *w0_int4 = &B->int4_data_ptr[(col * k + ch) / 2];
    uint8_t *w1_int4 = &B->int4_data_ptr[((col + 1) * k + ch) / 2];
    uint8_t *w2_int4 = &B->int4_data_ptr[((col + 2) * k + ch) / 2];
    uint8_t *w3_int4 = &B->int4_data_ptr[((col + 3) * k + ch) / 2];
    float s_a = params->A_scales[(row * k + ch) / block_size];
    float s_w0 = params->scales[(col * k + ch) / block_size];
    float s_w1 = params->scales[((col + 1) * k + ch) / block_size];
    float s_w2 = params->scales[((col + 2) * k + ch) / block_size];
    float s_w3 = params->scales[((col + 3) * k + ch) / block_size];
_ARM
    int intermediate_sum0 = 0, intermediate_sum1 = 0
    int intermediate_sum2 = 0, intermediate_sum3 = 0;
    for (int qj = 0; qj < 16; qj++) {
        uint8_t packed_int4_0 = w0_int4[qj];
        uint8_t packed_int4_1 = w1_int4[qj];
        uint8_t packed_int4_2 = w2_int4[qj];
        uint8_t packed_int4_3 = w3_int4[qj];
        signed char w_de_0 = (packed_int4_0 & 0x0F) - 8.0;
        signed char w_de_1 = (packed_int4_1 & 0x0F) - 8.0;
        signed char w_de_2 = (packed_int4_2 & 0x0F) - 8.0;
        signed char w_de_3 = (packed_int4_3 & 0x0F) - 8.0;
        signed char w_de_160 = (packed_int4_0 >> 4) - 8.0;
        signed char w_de_161 = (packed_int4_1 >> 4) - 8.0;
        signed char w_de_162 = (packed_int4_2 >> 4) - 8.0;
        signed char w_de_163 = (packed_int4_3 >> 4) - 8.0;
        intermediate_sum0 += a_int8[qj] * w_de_0;
        intermediate_sum1 += a_int8[qj] * w_de_1;
        intermediate_sum2 += a_int8[qj] * w_de_2;
        intermediate_sum3 += a_int8[qj] * w_de_3;
        intermediate_sum0 += a_int8[qj + 16] * w_de_160;
        intermediate_sum1 += a_int8[qj + 16] * w_de_161;
        intermediate_sum2 += a_int8[qj + 16] * w_de_162;
        intermediate_sum3 += a_int8[qj + 16] * w_de_163;
    }
    acc0 += (float)intermediate_sum0 * s_a * s_w0;
    acc1 += (float)intermediate_sum1 * s_a * s_w1;
    acc2 += (float)intermediate_sum2 * s_a * s_w2;
    acc3 += (float)intermediate_sum3 * s_a * s_w3;
    ch += block_size;
```

Figure 11. Core code of Loop Unrolling.

4.2 Loop Unrolling

AS figure11 shows, the code utilizes loop unrolling to process more data per iteration. By expanding the operations 4 time for each loop, we can improve the speed. The detail could be five steps as following:

1. Loop Over Channels: The code iterates over a set number of channels (ch) within a loop, accessing quantized activation and weight data for computation.
2. Access Quantized Data: Pointers are used to reference both 8-bit quantized activation values (a_int8) and 4-bit quantized weight values (w_int4).
3. Compute Scales: Scales for the quantized values are calculated, which will later be used to adjust the values post-multiplication, converting them back to a representational float format.
4. Unpack and Compute: Within a nested loop, the code unpacks the 4-bit weights into separate 8-bit integers, performs multiplications with the activations, and accumulates these products into intermediate sums.
5. Final Accumulation: The intermediate sums are then scaled and added to accumulator variables (acc0 to acc3), which contributes to the output of a neural network layer.

4.3 Multi-Threading

AS figure12 shows, the code creates 4-12 threads based on different settings. Then calculate which column should that process run. Finally the matrix calculation would be distributed into different threads to be calculated. And the steps

```cpp
struct w4a8_thread_args {
    int start_j, end_j;
    const struct matmul_params *params;
};
namespace matmul {
void MatmulOperator::mat_mul_all_techniques(struct matmul_params *params) {
    int i, j, k;
    const struct matrix *A = &params->A, *B = &params->B, *C = &params->C;
    const int block_size = params->block_size;
    float *scale = params->scales, *offset = params->offset;

    assert(params->block_size % 32 == 0);
    assert(A->row == C->row);

    quantize_fp32_to_int8(A->data_ptr, A->int8_data_ptr,
        params->A_scales, A->row * A->column, block_size);

    const int num_thread = 8;
    pthread_t thread_pool[num_thread];
    struct w4a8_thread_args threads_args[num_thread];
    assert(params->block_size == 32);  // support block size 32 for now

    // TODO: Thread creation
    int n = C->column;
    for (int i = 0; i < num_thread; ++i) {
        n = C->column;
        threads_args[i].start_j = i * (n / num_thread);
        threads_args[i].end_j = (i+1) * (n / num_thread);
        threads_args[i].params = params;
        int rc = pthread_create(&thread_pool[i], NULL,
            all_techniques_worker_func, (void *)&threads_args[i]);
        if (rc) {
            printf("Error: unable to create thread %d\n", rc);
            exit(-1);
        }
    }
    // TODO: Join threads
    for (int i = 0; i < num_thread; ++i) {
        pthread_join(thread_pool[i], NULL);
    }
};
}  // namespace matmul
```

Figure 12. Core code of Multi-Threading.

of the coding can be divided into 3 steps:

1. Distribute the different parts of matrix.
2. Put those parts into corresponded threads and calculate them.
3. Finish the calculation and join the main thread.

4.4 SIMD

AS figure13 and figure14shows, SIMD is more complex than Loop unrolling and multi-threading. In overview, the code can be divided into serveral parts:

1. Use ‘vandq_u8‘ with the mask_low4bit to get the lower half data.
2. Use ‘vshrq_n_u8‘ to right shift 4 bits and get the upper half.
3. Use ‘vreinterpretq_s8_u8‘ to interpret the vector as int8.
4. Apply zero_point to weights and convert the range from (0, 15) to (-8, 7).
5. Load 32 8-bit activation.
6. Perform dot product and store the result into the intermediate sum, int_sum0.
7. Perform the dot product.

```cpp
for (int q = 0; q < num_block; q += 4) {
    // load 32x4bit (16 bytes) weight
    const uint8x16_t w0 = vld1q_u8(w_start);       // 32 4bit weight
    const uint8x16_t w1 = vld1q_u8(w_start + 16);  // 32 4bit weight
    const uint8x16_t w2 = vld1q_u8(w_start + 32);  // 32 4bit weight
    const uint8x16_t w3 = vld1q_u8(w_start + 48);  // 32 4bit weight
    w_start += 64;

    // TODO: decode each uint8x16_t weight vector into the lower and upper half
    // Hint:
    // (1) use `vandq_u8` with the mask_low4bit to get the lower half
    // (2) use `vshrq_n_u8` to right shift 4 bits and get the upper half
    // (3) use `vreinterpretq_s8_u8` to interpret the  vector as int8
    // lowbit mask
    const uint8x16_t mask_low4bit = vdupq_n_u8(0xf);
    const uint8x16_t lower_half_u8_0 = vandq_u8(w0, mask_low4bit);
    const uint8x16_t upper_half_u8_0 = vshrq_n_u8(w0, 4);
    const int8x16_t lower_half_s8_0 = vreinterpretq_s8_u8(lower_half_u8_0);
    const int8x16_t upper_half_s8_0 = vreinterpretq_s8_u8(upper_half_u8_0);
    const uint8x16_t lower_half_u8_1 = vandq_u8(w1, mask_low4bit);
    const uint8x16_t upper_half_u8_1 = vshrq_n_u8(w1, 4);
    const int8x16_t lower_half_s8_1 = vreinterpretq_s8_u8(lower_half_u8_1);
    const int8x16_t upper_half_s8_1 = vreinterpretq_s8_u8(upper_half_u8_1);
    const uint8x16_t lower_half_u8_2 = vandq_u8(w2, mask_low4bit);
    const uint8x16_t upper_half_u8_2 = vshrq_n_u8(w2, 4);
    const int8x16_t lower_half_s8_2 = vreinterpretq_s8_u8(lower_half_u8_2);
    const int8x16_t upper_half_s8_2 = vreinterpretq_s8_u8(upper_half_u8_2);
    const uint8x16_t lower_half_u8_3 = vandq_u8(w3, mask_low4bit);
    const uint8x16_t upper_half_u8_3 = vshrq_n_u8(w3, 4);
    const int8x16_t lower_half_s8_3 = vreinterpretq_s8_u8(lower_half_u8_3);
    const int8x16_t upper_half_s8_3 = vreinterpretq_s8_u8(upper_half_u8_3);


    // TODO: apply zero_point to weights and convert the range from (0, 15) to (
    // Hint: using `vsubq_s8` to the lower-half and upper-half vectors of weight
    const int8x16_t offsets = vdupq_n_s8(8);
    const int8x16_t adjusted_lower_half_0 = vsubq_s8(lower_half_s8_0, offsets);
    const int8x16_t adjusted_upper_half_0 = vsubq_s8(upper_half_s8_0, offsets);
    const int8x16_t adjusted_lower_half_1 = vsubq_s8(lower_half_s8_1, offsets);
    const int8x16_t adjusted_upper_half_1 = vsubq_s8(upper_half_s8_1, offsets);
    const int8x16_t adjusted_lower_half_2 = vsubq_s8(lower_half_s8_2, offsets);
    const int8x16_t adjusted_upper_half_2 = vsubq_s8(upper_half_s8_2, offsets);
    const int8x16_t adjusted_lower_half_3 = vsubq_s8(lower_half_s8_3, offsets);
    const int8x16_t adjusted_upper_half_3 = vsubq_s8(upper_half_s8_3, offsets);
```

Figure 13. Core code of SIMD part 1.

4.5 Ablation Test

those all techniques, all the performances are shown in 1. And their improvements are following:

1. **Loop Unrolling**:Average time reduced from 12.44s to 12.11s, approximately a 2.7% decrease.

2. **Multi-threading**:Average time reduced from 12.44s to 3.18s, approximately a 74.4% decrease.

3. **SIMD**:Average time reduced from 12.44s to 8.64s, approximately a 30.5% decrease.

4. **Multi-threading + Loop Unrolling**:Average time reduced from 12.44s to 2.57s, approximately a 79.3% decrease.

	Total time(s)	Average time(s)	Count(word)
Baseline	1642.32	12.44	132
Loop unrolling	1598.42	12.11	132
Multi-threading	419.92	3.18	132
SIMD	1278.30	8.64	148
Multi-threading + Loop Unrolling	339.64	2.57	132
All techniques	207.79	1.40	148

Table 1. Performance Comparison

```
const int8x16_t offsets = vdupq_n_s8(8);
const int8x16_t adjusted_lower_half_0 = vsubq_s8(lower_half_s8_0, offsets);
const int8x16_t adjusted_upper_half_0 = vsubq_s8(upper_half_s8_0, offsets);
const int8x16_t adjusted_lower_half_1 = vsubq_s8(lower_half_s8_1, offsets);
const int8x16_t adjusted_upper_half_1 = vsubq_s8(upper_half_s8_1, offsets);
const int8x16_t adjusted_lower_half_2 = vsubq_s8(lower_half_s8_2, offsets);
const int8x16_t adjusted_upper_half_2 = vsubq_s8(upper_half_s8_2, offsets);
const int8x16_t adjusted_lower_half_3 = vsubq_s8(lower_half_s8_3, offsets);
const int8x16_t adjusted_upper_half_3 = vsubq_s8(upper_half_s8_3, offsets);

// load 128 8-bit activation
const int8x16_t a0 = vld1q_s8(a_start);
const int8x16_t a1 = vld1q_s8(a_start + 16);
const int8x16_t a2 = vld1q_s8(a_start + 32);
const int8x16_t a3 = vld1q_s8(a_start + 48);
const int8x16_t a4 = vld1q_s8(a_start + 64);
const int8x16_t a5 = vld1q_s8(a_start + 80);
const int8x16_t a6 = vld1q_s8(a_start + 96);
const int8x16_t a7 = vld1q_s8(a_start + 112);
a_start += 128;

// TODO: perform dot product and store the result into the intermediate sum
// Hint: use `vdotq_s32` and store the sum for each block in int_sum{0-3}
int32x4_t int_sum0 = vdupq_n_s32(0), int_sum1 = vdupq_n_s32(0),
    int_sum2 = vdupq_n_s32(0), int_sum3 = vdupq_n_s32(0);

int_sum0 = vdotq_s32(int_sum0, a0, adjusted_lower_half_0);
int_sum0 = vdotq_s32(int_sum0, a1, adjusted_upper_half_0);
int_sum1 = vdotq_s32(int_sum1, a2, adjusted_lower_half_1);
int_sum1 = vdotq_s32(int_sum1, a3, adjusted_upper_half_1);
int_sum2 = vdotq_s32(int_sum2, a4, adjusted_lower_half_2);
int_sum2 = vdotq_s32(int_sum2, a5, adjusted_upper_half_2);
int_sum3 = vdotq_s32(int_sum3, a6, adjusted_lower_half_3);
int_sum3 = vdotq_s32(int_sum3, a7, adjusted_upper_half_3);

float s_0 = *s_a++ * *s_w++;
float s_1 = *s_a++ * *s_w++;
float s_2 = *s_a++ * *s_w++;
float s_3 = *s_a++ * *s_w++;

sumv0 = vmlaq_n_f32(sumv0, vcvtq_f32_s32(int_sum0), s_0);
sumv0 = vmlaq_n_f32(sumv0, vcvtq_f32_s32(int_sum1), s_1);
sumv0 = vmlaq_n_f32(sumv0, vcvtq_f32_s32(int_sum2), s_2);
sumv0 = vmlaq_n_f32(sumv0, vcvtq_f32_s32(int_sum3), s_3);
```

Figure 14. Core code of SIMD part 2.

5. **All techniques**:Average time reduced from 12.44s to 1.40s, approximately an 88.7% decrease.

5 DISCUSSION

5.1 Graph Comparison

As shown in the figure, among all the improvement methods, when viewed individually, the improvement of multithreading is the most obvious, followed by SIMD, and Loop unrolling has the smallest improvement. If combined, these

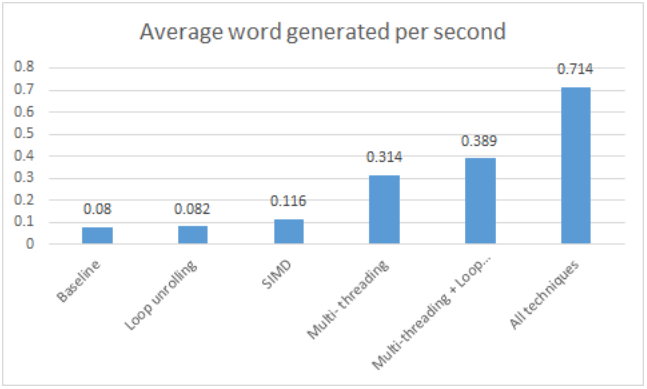

Figure 15. The speed of different models.

methods will not be mutually exclusive, but will complement each other, greatly increasing the speed.

Although the improvement of SIMD and Loop unrolling is not as significant as that of multithreading, it is accomplished without consuming more computer CPU, which means that it will be very effective under any ARM architecture.

In comparison, the improvement of multithreading greatly depends on the number of CPU cores. Our CPU core number is 11, so in a setting of less than or equal to 12, the speed increases rapidly with the number of threads.

5.2 Result Demonstration

We tested our model on a MacBook Pro laptop. The system's information is as follows. The model comes with Apple Silicon, which means it utilizes the ARM architecture. Additionally, it is equipped with 11 CPU cores and 18 GB of LPDDR5 memory.

Regarding the model we discussed earlier, we are utilizing a quantized model called LLaMA2, which consists of 7 billion parameters. This model has undergone quantization, specifically to Int4 data format, and is now prepared for integration with our optimization process.

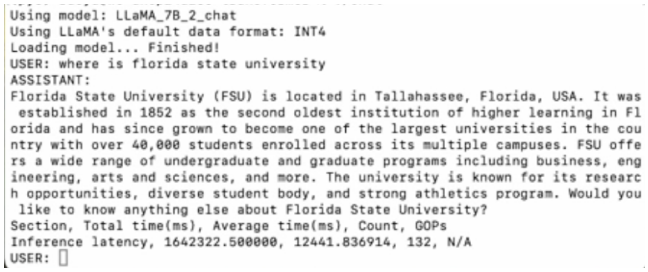

Figure 16. Baseline: Model without any optimizations

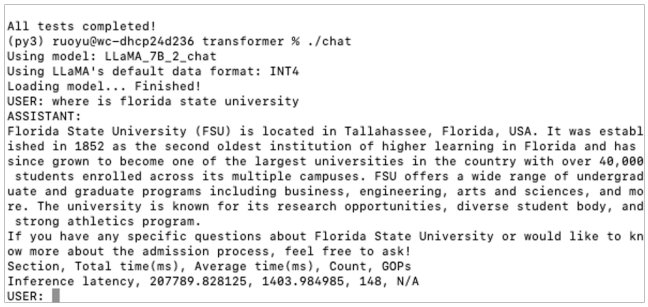

Figure 17. All techniques applied: Model with all optimizations

The demonstration result is showed in Figure 16 and Figure 17.During the demonstration, we posed the question "Where is Florida State University?" to the model with the anticipation of receiving an appropriate response in a terminal chat window. As the results indicated, the baseline model took approximately 30 minutes to generate a response, with an average response time of around 12.4 seconds. However, after implementing all the system-level optimizations, the improved model exhibited a remarkable reduction in latency, completing the task in just about 2 minutes in total, with response times averaging around 1.4 seconds. This substantial improvement transformed the previously unacceptable latency into a relatively tolerable timeframe.

6 Conclusion and Future Work

In this article, we applied the techniques of loop unrolling, multithreading, and SIMD to accelerate the inference of the large language model (LLM) on a laptop. As a result, the average inference speed of the LLM on the laptop was reduced from 12.4 seconds per token to 1.4 seconds per token. This significant reduction in latency makes the LLM's performance on a laptop acceptable. However, there are still several avenues for further improvement that we can explore in the future.

One potential direction for future work is to leverage CUDA, a parallel computing platform and programming model, to achieve additional speed improvements. By utilizing the computational power of GPUs, we can offload certain computations from the CPU to the GPU, which is optimized for parallel processing. This approach has shown promising results in accelerating various deep learning models, and it may provide further performance gains for the LLM. By carefully optimizing the GPU implementation and exploiting parallelism at different levels, we can potentially reduce the inference time even further.

Additionally, another technique worth investigating is loop tiling. By partitioning the input data into smaller, more manageable tiles, we can reduce cache misses and improve data locality. This can lead to more efficient memory access patterns, resulting in faster inference times. Exploring different tiling strategies and evaluating their impact on the LLM's performance can provide insights into optimizing the computational efficiency and reducing memory latency.

Furthermore, it would be valuable to extend this approach to other types of LLMs to validate its general effectiveness. Large language models come in various architectures, sizes, and configurations, and each may have unique characteristics that affect their inference performance. By applying the proposed techniques to different LLMs, we can assess the generalizability of our approach and identify any specific optimizations that may be required for different models. This broader evaluation can help establish the effectiveness and applicability of our acceleration techniques across a wider range of LLM architectures.

In conclusion, while we have achieved significant improvements in the inference speed of the LLM on a laptop through loop unrolling, multithreading, and SIMD techniques, there are still promising avenues for future work. Leveraging CUDA, exploring loop tiling techniques, and extending this approach to other LLMs can potentially unlock further performance gains and contribute to the ongoing efforts in optimizing the inference of large language models.

References

[1] 2024. (Apr. 2024). https://en.wikipedia.org/wiki/Parallel_computing (cit. on pp. 5, 6).

[2] H. Akkary and M.A. Driscoll. 1998. A dynamic multithreading processor. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 226–236. DOI: 10.1109/MICRO.1998.742784 (cit. on p. 6).

[3] Nathan Beckmann, Phillip B Gibbons, and Charles McGuffey. 2022. Spatial locality and granularity change in caching. (2022). arXiv: 2205.14543 [cs.DS] (cit. on p. 5).

[4] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. 2020. An overview on edge computing research. *IEEE Access*, 8, 85714–85728. DOI: 10.1109/ACCESS.2020.2991734 (cit. on p. 5).

[5] Agner Fog. 2012. Optimizing subroutines in assembly language. Copenhagen University College of Engineering. PDF. Retrieved 2012-09-22. (Feb. 2012). http://agner.org/optimize/optimizing_assembly.pdf (cit. on p. 8).

[6] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A survey of quantization methods for efficient neural network inference. (2021). arXiv: 2103.13630 [cs.CV] (cit. on pp. 4, 5).

[7] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6, 9, (July 2013), 709–720. DOI: 10.14778/2536360.2536370 (cit. on p. 6).

[8] Andrey Kuzmin, Markus Nagel, Mart van Baalen, Arash Behboodi, and Tijmen Blankevoort. 2024. Pruning vs quantization: which is better? (2024). arXiv: 2307.02973 [cs.LG] (cit. on p. 5).

[9] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. Mcunet: tiny deep learning on iot devices. (2020). arXiv: 2007.10319 [cs.CV] (cit. on p. 7).

[10] H Lyu, N. Sha, S. Qin, M. Yan, Y. Xie, and R. Wang. [n. d.] Advances in neural information processing systems. *Advances in neural information processing systems*, 32. https://par.nsf.gov/biblio/10195511 (cit. on p. 6).

[11] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. 2021. A white paper on neural network quantization. (2021). arXiv: 2106.08295 [cs.LG] (cit. on p. 5).

[12] W.P. Petersen and P. Arbenz. 2004. *Introduction to Parallel Computing*. Oxford University Press (cit. on p. 8).

[13] Haotian Tang, Shang Yang, Ji Lin, Jiaming Tang, Wei-Ming Chen, Wei-Chen Wang, and Song Han. 2003. (Sept. 2003). https://hanlab.mit.edu/blog/tinychat (cit. on p. 6).

[14] Hugo Touvron et al. 2023. Llama: open and efficient foundation language models. (2023). arXiv: 2302.13971 [cs.CL] (cit. on p. 4).

[15] Jeffrey D. Ullman and Alfred V. Aho. 1977. *Principles of Compiler Design*. Addison-Wesley Pub. Co., Reading, Mass, 471–472. ISBN: 0-201-10073-8 (cit. on p. 8).

[16] Sai H. Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. 2024. Chatgpt for robotics: design principles and model abilities. *IEEE Access*, 12, 55682–55696. DOI: 10.1109/ACCESS.2024.3387941 (cit. on p. 4).

[17] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. 2024. Explainability for large language models: a survey. *ACM Trans. Intell. Syst. Technol.*, 15, 2, Article 20, (Feb. 2024), 38 pages. DOI: 10.1145/3639372 (cit. on p. 4).

[18] Wayne Xin Zhao et al. 2023. A survey of large language models. (2023). arXiv: 2303.18223 [cs.CL] (cit. on p. 4).