Guided Research with Intel

Technische Universität München
Department of Informatics

Efficient and Sustainable LLMs on Intel® Xeon® with Intel® AMX

Authors

Hoang Le Doan (TUM)

Dr. -Ing. Mohamed Elsaid (Intel-Germany)
Cloud Solutions Architect, Supervisor

Annatam Dey (Intel-India)
Cloud Solutions Architect, Supervisor

Prof. Dr. Michael Gerndt (TUM)
Advisor

Audience

SMG, AI TSSs, AI Software groups, DCG BU

Submission Date

23.4.2025

Table of Contents

Abstract

Large Language Models (LLMs) have gained significant attention due to their remarkable performance across various tasks. These systems learn from massive datasets, enabling them to perform tasks, from creating conversational chatbots to generating complex content. The ability to handle diverse linguistic patterns makes them attractive in many applications. However, the substantial computational and memory requirements of LLM training and inference pose significant challenges for resource-constrained devices.

Although GPUs are a common choice for LLMs due to their capacity for parallel processing to handle matrix multiplication, they face significant challenges with high cost and power consumption. Intel's recent CPU architectures offer a compelling alternative through Intel® Advanced Matrix Extensions (Intel® AMX). This technology integrates specialized matrix multiplication units directly into the processor silicon, targeting the small LLMs that are less than 10 Billion parameters for model training and 20 Billion parameters for model inference. Combined with the larger RAM capacity available to CPU systems, Intel AMX-enabled processors create new possibilities for efficient LLM training and deployment. This research evaluates Meta's Llama-2 7B model, a representative decoder-only architecture, on Intel® Xeon® processors equipped with Intel AMX technology. By using post-training quantization methods specifically designed to leverage Intel AMX's computational patterns, the study demonstrates substantial improvements in both inference speed and energy efficiency compared to standard CPU execution.

1 Introduction

1.1 Overview of Large Language Models (LLMs)

Language serves as the cornerstone of human communication, self-expression, and machine interaction. The rise of generalized models responds to the increasing need for machines to manage sophisticated language tasks: translation, summarization, information retrieval, and conversation. Recent advances in large language models (LLMs), built on the Transformer architecture [16], demonstrate remarkable capabilities in context searching and reasoning [11].

According to [11], LLMs can be categorized into 4 types according to their architecture around the transformer:

- **Encoder-decoder:** Uses separate components to process input and generate output. The encoder comprehends context, while the decoder produces responses. Examples include T5 [12] and BART [10], which excel at tasks that require deep understanding before generation.
- **Causal decoder:** Generates text sequentially, with each token prediction based on previous tokens. GPT [17] and Llama [15] use this approach, making them effective for text completion and generation tasks.
- **Prefix decoder:** Allows bidirectional attention over input tokens. This hybrid approach balances understanding and generation capabilities.
- **Mixture of experts:** Using multiple specialized neural networks (experts) that are activated selectively based on input. This architecture improves efficiency by routing computation through relevant pathways rather than the entire model.

Causal decoder architectures have gained significant popularity for their simplicity, computational efficiency, and strong performance across various generation tasks. In this research, we primarily focus on this architecture.

1.2 Transformer architecture

Every state-of-the-art LLM today is built on the Transformer architecture. The self-attention mechanism in this architecture allows the model to weigh the importance of different words in a sequence when processing each specific word, enabling it to capture long-range dependencies effectively.

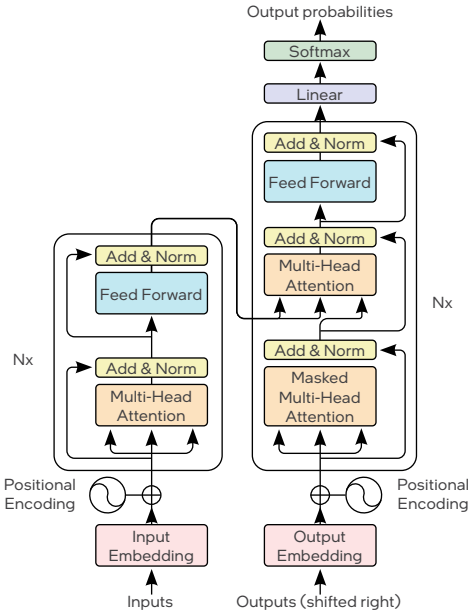

Figure 1: The Transformer - model architecture.

Figure 1: Attention architecture reproduced from [16].

The self-attention mechanism is expressed as in [16]:

$$\text{Attention}(Q, K, V) = \text{softmax}\frac{QK^T}{\sqrt{d_k}}V \quad (1)$$

where $Q \in R^{n \times d_k}$, $K \in R^{n \times d_k}$, and $V \in R^{n \times d_v}$ are the query, key, and value matrices, respectively. Q (Querry) represents what we are searching for, K (Key) represents what we are matching against, and V (Value) contains the information we want to retrieve. n represents the sequence length (i.e., the number of tokens in the input), d_k is the embedding dimensionality of both the query and key vectors (and also serves as a scaling factor in the denominator), and d_v is the embedding dimensionality of the value vectors. Finally, the softmax function normalizes the attention scores into a probability distribution over all tokens in the sequence.

Rather than relying on a single attention pattern, transformers use multi-head attention [16], which allows the model to capture the information from different representation subspaces:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \quad (2)$$

where each attention head operates in a different projection space:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \quad (3)$$

Here, the input matrices Q, K, and V are linearly projected h times using different projection matrices $W_i^Q, W_i^K \in R^{d\text{model} \times d_k}$ and $W_i^V \in R^{d\text{model} \times d_v}$, where $d_k = d_v = d_{model}/h$ so that the model can capture relationships in different subspaces (or dimensions) through multiple attention heads.

The computational complexity of transformers comes primarily from the self-attention mechanism, which scales quadratically with the sequence length due to the QK^T operation that computes pairwise dot products between all query and key vectors. For a sequence of length n, the dot product operation results in a complexity of $O(n^2)$. Since we repeat this process d times, where d is the embedding dimension, the total complexity for the self-attention mechanism is $O(n^2d)$. Additionally, the matrix multiplications involving the Q, K, and V matrices with their corresponding projection matrices W^Q, W^K, and W_i^V contribute $O(nd^2)$ operations.

These operations dominate the computational cost during inference, making them prime targets for optimization.

1.3 Llama

In this research project, we chose Llama (Large Language Model Meta AI) for our experiments due to its efficiency and open-source availability. Llama is a family of transformer-based models with a decoder-only architecture, similar to GPT models, but with optimizations that improve scalability, inference speed, and training stability:

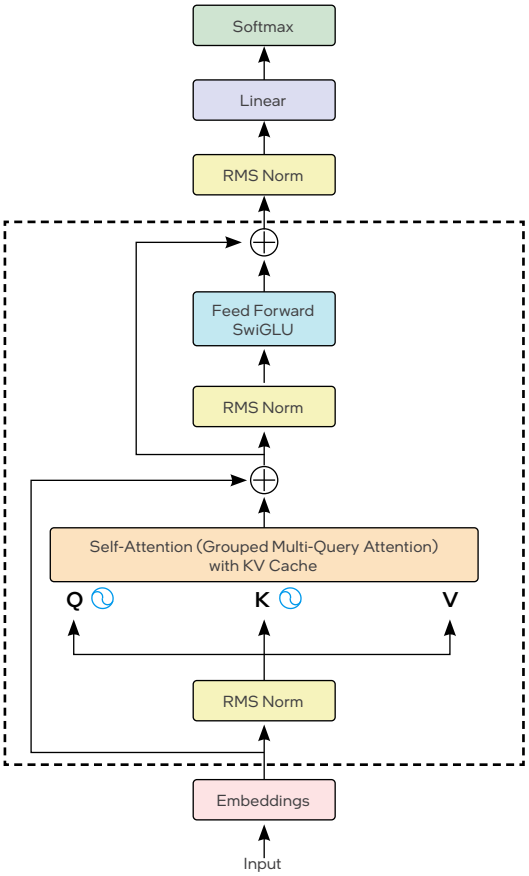

Figure 2: Llama Architecture

1. Token Embeddings Llama uses a SentencePiece-based tokenizer [8], which segments input text into subword units. These tokens are then mapped to high-dimensional vector embeddings, forming the model's input representation.

2. Positional Encoding Instead of using absolute positional encodings like in traditional transformers, Llama uses Rotary Positional Embeddings (RoPE) [14]. RoPE encodes relative positional information directly into the self-attention mechanism, improving the model's ability to handle long-range dependencies.

3. Grouped Query Attention (GQA) Instead of using Multi-Head Attention, which is mentioned in equation 2, Llama uses Grouped Query Attention, where multiple query heads share the same key and value projections. This reduces computational overhead while maintaining the accuracy of the attention mechanism.

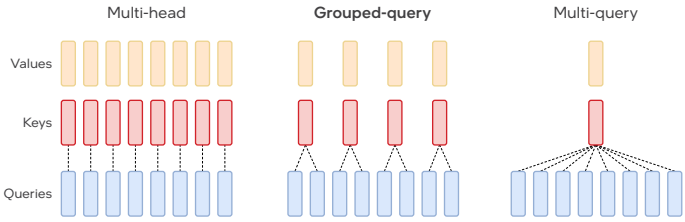

Figure 3: Group Querry Attention

4. Feedforward Network (FFN) Each transformer block in Llama includes a feedforward network consisting of two linear transformations with a SwiGLU activation [13] function in between. The FFN is responsible for transforming feature representations and adding non-linearity.

5. RMS Norm Unlike traditional models that apply post-layer normalization, Llama utilizes Pre-Layer Normalization (Pre-LN) with RMSNorm [18]. This enhances training stability and gradient flow.

6. Causal Masking Since Llama is a decoder-only model, it uses causal masking in self-attention layers. This ensures that each token attends only to previous tokens, enabling left-to-right autoregressive generation.

1.4 Optimization Techniques for Efficient Inference

1.4.1 KV Cache

During inference, a decoder-only model generates the next token sequentially, using the input token and all previously generated tokens [19]. At time step t, the model already has generated tokens from positions 1 to $t-1$ and wants to predict token at t. To do it, the model computes the attention scores [16]:

$$O_{1:t-1} = \text{softmax}\left(\frac{Q_{1:t-1}K_{1:t-1}^{T}}{\sqrt{d_k}}\right)V_{1:t-1} \quad (4)$$

This is done subsequently throughout the attention blocks in the model's architecture. The model then projects the output to the vocabulary space, producing a probability distribution $P(x_t|x_1, x_2, ..., x_{t-1}) = \text{softmax}(W_{vocab} \cdot h_{t-1})$ over the vocabulary for position t, where h_{t-1} is the final hidden state and W_{vocab} is the vocabulary projection matrix. The token with the highest probability becomes token t.

This process becomes extremely time-consuming, as the model must recompute the dot product of Q, K, V vectors for every token at each step [19]. As the length of the input sequence grows, this results in increasingly expensive calculations. The KV cache optimizes this process by storing the key and value vectors for all previously processed tokens. Once computed, these vectors are cached and can be reused in subsequent time steps. When generating a new token, only the current Q, vectors are computed, while the previous key K and V vectors are retrieved and used in the attention computation.

The use of the KV cache introduces an additional memory overhead that grows linearly with the sequence length [4]. The cache size scales linearly with both the model's embedding size, the context length and the batch size. For long-context sequences, this memory requirement become significant bottleneck.

For example, consider the Llama-2-7B model, which has 32 Multi-Head Attention layers (L), each has 32 attention heads (h), and a model dimension of 4096 (d_{model}) with per-head dimension of 128 ($d_k = d_{model}/h$). Using float32 precision (4 bytes per value), a batch size of 8 (B), and a sequence length of 128 (n), the total memory required for storing the KV cache can be calculated as follows:

$$\text{Memory} = L \times B \times n \times h \times d_k \times 2 \times \text{bytes per value} \quad (5)$$

$$= 32 \times 8 \times 128 \times 32 \times 128 \times 2 \times 4 \text{ bytes} \quad (6)$$

$$= 1 \text{ GB} \quad (7)$$

where the factor of 2 accounts for storing both (K) and (V) in the cache. Consider a larger model, such as Llama-70B, which is estimated to have over 70 billion parameters and significantly more attention heads and layers. Suppose a variant of Llama-70B has 96 layers (L), 96 attention heads (h), and a hidden dimension of 12288 (d_{model}), with per-head dimension $d_k = d_{model}/h = 128$. If we assume a batch size of 8, a sequence length of 512, and float32 precision (4 bytes per value), the memory required for the KV cache becomes:

$$\text{Memory} = L \times B \times n \times h \times d_k \times 2 \times \text{bytes per value} \quad (8)$$

$$= 96 \times 8 \times 512 \times 96 \times 128 \times 2 \times 4 \text{ bytes} \quad (9)$$

$$= 48 \text{ GB} \quad (10)$$

This shows that KV cache memory grows linearly with sequence length and batch size. So AI infrastructure memory should afford the required capacity for long-context inference of LLMs.

1.4.2 Post-Training Quantization (PTQ)

Post-Training Quantization (PTQ) is a technique that reduces the precision of model weights and activations after training. Usually with LLM, people don't use Quantization Aware Training because of its expensiveness. Traditionally, LLMs use 32-bit floating-point (FP32) or 16-bit floating-point (FP16) precision for weights and activations. PTQ converts these high-precision values to lower-precision formats such as 8-bit integers (INT8), 4-bit integers (INT4). PTQ offers several advantages for LLM inference:

- Reduced memory footprint: Lower-precision representations require less memory, enabling deployment of larger models or longer context windows.
- Improved computational efficiency: Integer operations are typically faster than floating-point operations.
- Reduced energy consumption: Lower-precision arithmetic consumes less power.

Recent advancements in quantization techniques have demonstrated that Llama models can be quantized with this technique to be 8 or even 4 bits with very few degradation in quality, making it possible to run billion-parameter models locally [3].

1.5 Other optimization techniques

There are several optimization techniques for transformers architecture, for instance: pruning or knowledge distillation. However, this would require huge computational resources to restore accuracy. CPUs are only suitable for training LLMs with fewer than 10 billion parameters and for inference with models under 20 billion parameters. Therefore in this project, we will not explore those mentioned.

1.6 Utilizing CPUs for LLM Inference

Using CPUs only with Intel AMX accelerator for AI workloads is an efficient solution for small GenAI models with up to 10B parameters in model training and up to 20B parameters in model inference [20] and [22]. In this Intel Xeon CPU/AMX based infrastructure setup, the AI solution can be easier to setup and in certain usage scenarios may become more cost efficient and more power optimized than using GPUs or NPUs to accelerate the AI compute for small GenAI models [22]. We prove in this paper that using Intel Xeon AMX can also save energy consumption versus using the standard CPU cores only.

Usage of small GenAI models can fit many customers use-cases that rely on using AI models for specific tasks, edge deployments or for RAG systems where the LLM models are trained and developed on the organization data and files only. Example of LLM models that are less than 10B parameters are: Llama 3.1, Mistral, Phi-3, Gemma and Cerebras-GPT.

1.7 Intel AMX

Intel AMX is a built-in accelerator that is available from 4th generation Intel Xeon. It is integrated into each CPU core, placed near system memory, designed to enhance matrix operations for AI workloads [7].

The Intel AMX consists of 2 main components[1]:

- The first component is tiles. Tiles consist of eight two-dimensional registers, each 1 kilobyte in size. They store large chunks of data.
- The second component is Tile Matrix Multiplication (TMUL). TMUL is an accelerator engine attached to the tiles that performs matrix- multiplication computations for AI.

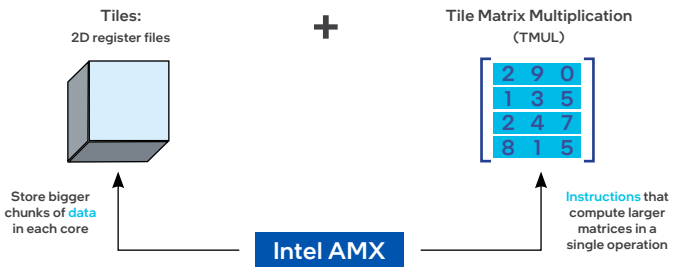

Figure 4: Intel AMX Overview [1]

The CPU controls how programs run while sending matrix calculations to Intel AMX units (See Fig. 5). These parts communicate in three main ways: 1) through commands, where the CPU sends instructions and receives status updates, 2) through data transfers to move information between main memory and special tile registers, and 3) through configuration settings that tell the tile registers how to structure themselves using the TILECFG register. The CPU handles the overall program flow and memory management, while Intel AMX performs matrix operations at the same time as regular instructions, without needing separate processes. Intel AMX keeps track of two types of information: how tiles are configured (their size and organization) and what data is stored in them. This design allows Intel AMX instructions to mix freely with regular CPU instructions without competing for resources, even with other systems like AVX-512. This makes Intel AMX work as a natural extension of what the CPU can do, rather than as a separate system, maintaining the familiar programming approach while adding specialized matrix calculation abilities [2].

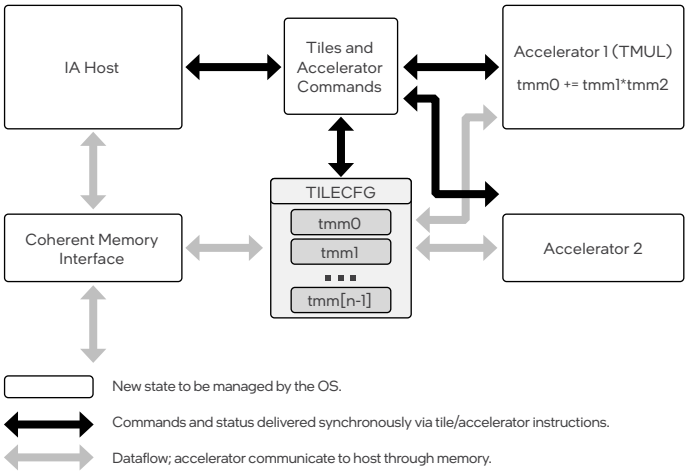

Figure 5: Intel AMX Workflow [2]

Intel AMX accelerates deep learning workloads by supporting multiple specialized data types, including INT8 and BF16. The accelerator TMUL, which consists of a grid of Fused Multiply-Add (FMA) units optimized for matrix operations. When performing matrix multiplication (**C** = **A** × **B**), Intel AMX employs several key acceleration techniques [2]:

- **Parallel Execution:** The TMUL unit processes multiple elements simultaneously using its FMA grid, enabling parallel computation of dot products.
- **Data Locality:** The tile register architecture minimizes memory accesses by keeping matrix data close to the computation units. Each tile register can store up to 16 rows by 64 columns.
- **Data Reuse:** Once loaded, tiles from matrix **A** can be reused for later operations, optimizing efficiency.
- **Reduced Instruction Overhead:** Intel AMX simplifies complex matrix multiplication by encapsulating operations into specialized instructions, eliminating the need for explicit loop management and address calculations.

1.8 Intel AVX-512 VNNI

The Intel's AVX-512 instruction set is specifically designed to accelerate deep learning inference workloads using integer-based computations. This is applied to INT8 quantized neural networks by implementing fused multiply- accumulate (FMAC) operations that allow a single instruction to perform vectorized INT8 multiplication with INT32 accumulation, gaining significant performance for AI inference tasks, particularly in lightweight scenarios with low batch inference requirements.

The difference between AVX-512 VNNI and Intel AMX lies in their approach to matrix operations. While Intel AMX employs a 2D tile-based methodology that processes entire matrix blocks simultaneously, AVX-512 VNNI relies on traditional SIMD (Single Instruction, Multiple Data) processing that decomposes matrices into vector chunks. This SIMD approach performs element-wise dot products followed by accumulation across these decomposed vectors. Although this method enhances parallelism compared to scalar

operations, it introduces overhead from repeated vector loading/storing operations and the computational cost of iteratively traversing the matrix structure. These inefficiencies become particularly bigger when handling the large matrices typical in modern transformer-based language models.

2 Experiment

2.1 Setup

In our experimental setup, we use the Intel Xeon Platinum 8460Y+ processor, part of Intel's 4th generation Xeon Scalable series (codename Sapphire Rapids), which has a dual-socket with 40 physical cores and 80 threads per socket, operating at a base frequency of 2.0 GHz with boost capabilities up to 3.7 GHz [5]. Each processor has a Thermal Design Power (TDP) rating of 300W (600W total for the dual-socket system).

Storage was provided by a 960 GB Intel SATA SSD (model:INTEL SSDSC2KG96), and the server was connected to a 10G network. The operating system used was Ubuntu 24.04 LTS, running on the GNU/Linux 6.8.0-55-generic x86 64 kernel. For energy measurements, we use Intel's Performance Counter Monitor (PCM) framework, which leverages the processor's built-in Running Average Power Limit (RAPL) interface to access on-die energy meters that measure actual energy consumption across multiple power domains (CPU cores and DRAM) through Model-Specific Registers (MSRs) that track cumulative energy in joules with microsecond-level resolution, allowing us to calculate power consumption [6].

Since we only have one CPU, we choose a light weight, open source LLM model for inference: Llama2-7b-chat. For the inference process, we apply weight-only quantization to convert the model weights from Float32 to Int8 and Int4, which are supported by both Intel AMX and AVX-512 VNNI. Specifically, for AVX-512 VNNI, the weights are multiplied in the Int8/Int4 format and accumulated using the Float32 activation function.

For Intel AMX, the weights are multiplied in Int8/Int4 format and accumulated using the BF16 activation function, which is supported by Intel AMX. Note that we do not perform fine-tuning after quantization due to resource limitations. As benchmarked by [3] and [9], weight-only quantization can still maintain more than 99% of model accuracy, and using BF16 activation can restore 100% of the model's original accuracy.

The model is then compiled to the OpenVINO IR format to leverage the full potential of Intel hardware using OpenVINO. OpenVINO is an open- source toolkit developed by Intel for optimizing and deploying deep learning models, especially for inference on Intel hardware. We then measure the total time for the model to run inference at a specific time point, which corresponds to a noticeable energy surge, as recorded in the PCM logs. The total time from this point until inference completion is used to calculate the overall energy consumption during the inference process.

The experiment measures performance across different batch sizes (1, 8, 64, 256, 512) and input sequence lengths (64, 128, 256, 512 tokens) to evaluate how the model handles different workload. For training, the batch size defines how many training samples are processed before the model's weights are updated. For inference, it represents the number of user requests that the LLM can serve simultaneously. For instance, with a batch size of 8 and an input sequence length of 128, 8 sequences are concatenated into a single tensor of shape [8, 128], which is then passed into the model. The model then embeds the input, for Llama 2 7-B this becomes an input tensor of shape [8, 128, 4096]. The CPU distributes the computation across multiple threads to exploit parallelism. On a CPU with 160 cores, threads are assigned to slices of the input tensor: threads [1–20] might handle the first sentence of shape [128, 4096], threads [21–40] the next, and so on. Within each thread, the large matrices are further divided into smaller tile blocks with shape [16, 64] that fit into Intel AMX tile registers. Each tile register holds a small block of the matrix data and passes it to the TMUL accelerator engine for matrix multiplication. Intermediate results are also stored directly in the tile registers, allowing partial sums to be accumulated locally without repeatedly accessing memory or waiting for new data from the thread. After every operation (Attention score, Feedforward, etc.) each thread's output must be synchronized. A global barrier ensures that all threads complete their computations before proceeding to the next transformer layer. Because each layer's computation depends on the full output of the previous layer, there is no pipelining across layers; the model must finish processing the current token batch fully before moving forward. This ensures that every sentence within the batch is processed concurrently and consistently, preserving the sequential dependencies in the model's architecture.

2.2 With vs Without Intel AMX:

In this paper, we compare the same LLM inference benchmarking setup with the same hardware configuration that is mentioned in the previous section 2.1 one time with the enablement of the LLM model to utilize Intel AMX versus disabling this Intel AMX feature to show case the performance and energy consumption impact of using Intel AMX versus using the CPU core only.

2.3 Metrices

The inference process of LLMs can be divided into two stages [19]:

- **Prefilling stage:** The model computes and caches the initial Key-Value pairs for input tokens, establishing the KV cache necessary for subsequent processing. Simultaneously, it generates the first output token through transformer block computations as defined in Equation 4.
- **Decoding stage:** The model sequentially generates subsequent tokens, utilizing and continuously updating the KV cache with new key-value pairs corresponding to each newly produced token. When evaluating Large Language Models (LLMs), we focus on three critical performance metrics:
  - **First-token latency (ms):** Time required to generate the initial out- put token during prefilling.
  - **Throughput (token/s):** Generation time for each subsequent token. This is measured from the second token.
  - **Total energy (J):** Total energy consumption during the complete inference process.

3 Benchmark

We used same server configurations with controlled thermal and power conditions to ensure measurement consistency. We only show here the benchmark of 128 and 256 input token, as the 64 and 512 token have the same properties. Each measurement we repeat 5 times and take the average.

Model	Batch size	Intel AMX (AVX10 1 512 AMX)			No Intel AMX (AVX512 CORE VNNI)		
		Total Energy (J)	First token latency per batch (ms)	Throughput (tokens/s)	Total Energy (J)	First token latency per batch (ms)	Throughput (tokens/s)
Llama-2-7b-chat-hf (int8)	1	3978.32	114.06	19.84	3818.83	253.99	20.31
	8	5306.08	496.66	118.43	5586.23	1589.04	128.06
	64	11863.69	4844.84	419.74	23850.23	13387.81	312.37
	256	31489.06	15571.33	872.20	74139.04	45852.10	422.27
	512	61394.88	32356.71	950.36	138118.58	87843.27	462.74
Llama-2-7b-chat-hf (int4)	1	2626.18	113.27	27.07	2300.90	211.08	28.40
	8	4106.07	457.58	139.85	4637.76	1308.49	160.46
	64	11682.87	4731.73	495.34	17845.70	10997.24	377.99
	256	32829.68	14053.26	731.25	65818.74	44224.18	483.81
	512	66601.19	36394.54	717.44	118671.50	74866.78	535.29

Table 1: Performance Comparison between Intel AMX and AVX512 VNNI Instructions for 128 input token and 128 output tokens

Model	Batch size	Intel AMX (AVX10 1 512 AMX)			No Intel AMX (AVX512 CORE VNNI)		
		Total Energy (J)	First token latency per batch (ms)	Throughput (tokens/s)	Total Energy (J)	First token latency per batch (ms)	Throughput (tokens/s)
Llama-2-7b-chat-hf (int8)	1	3502.92	185.57	19.97	4008.41	534.04	20.59
	8	5813.79	903.99	115.83	5939.43	3593.67	125.92
	64	14892.16	8705.08	452.47	30889.42	27026.73	328.06
	256	54431.07	24837.87	519.32	113629.44	96942.65	347.35
	512	79355.07	56714.681	729.50	216814.6	181572.29	330.42
Llama-2-7b-chat-hf (int4)	1	3466.78	174.02	24.51	2787.37	363.66	27.81
	8	4721.5	1021.67	139.79	5355.35	2761.12	158.58
	64	15237.67	6144.18	406.77	25471.71	21413.24	358.39
	256	50472.68	28522.08	479.51	94535.23	80036.02	413.30
	512	94290.39	59098.04	528.30	177287.86	158628.19	439.86

Table 2: Performance Comparison between Intel AMX and AVX512 VNNI Instructions for 256 input token and 128 output token

Figures below show the benchmark of Int8 and Int4 models for 256 input token:

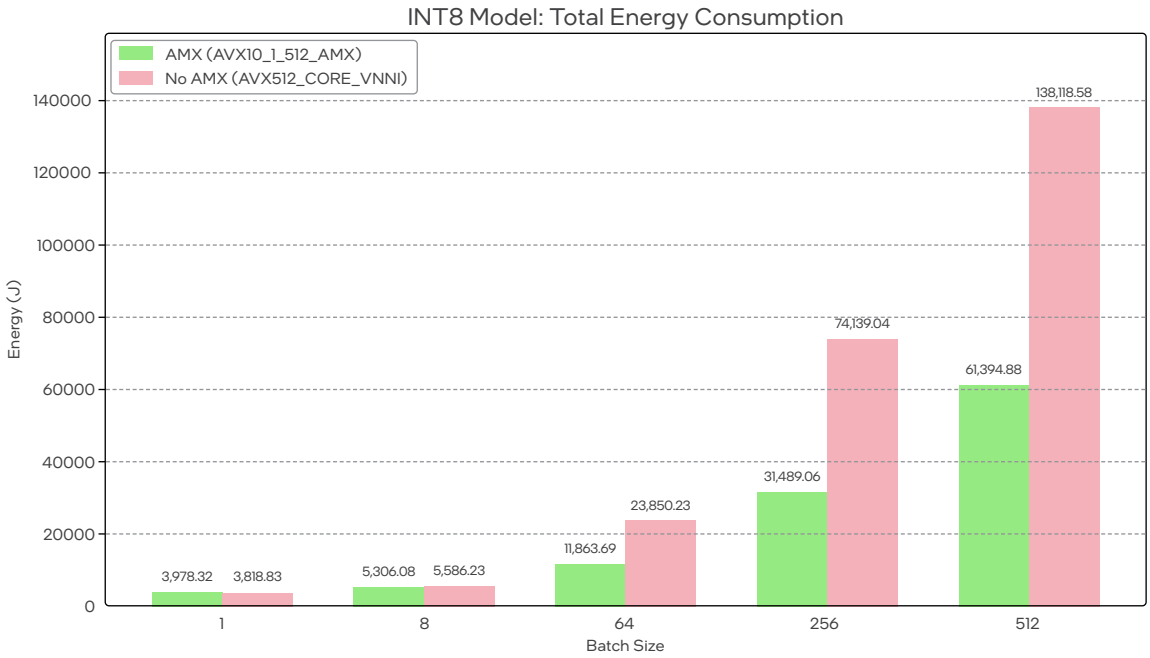

Figure 6: Int8 Energy Consumption

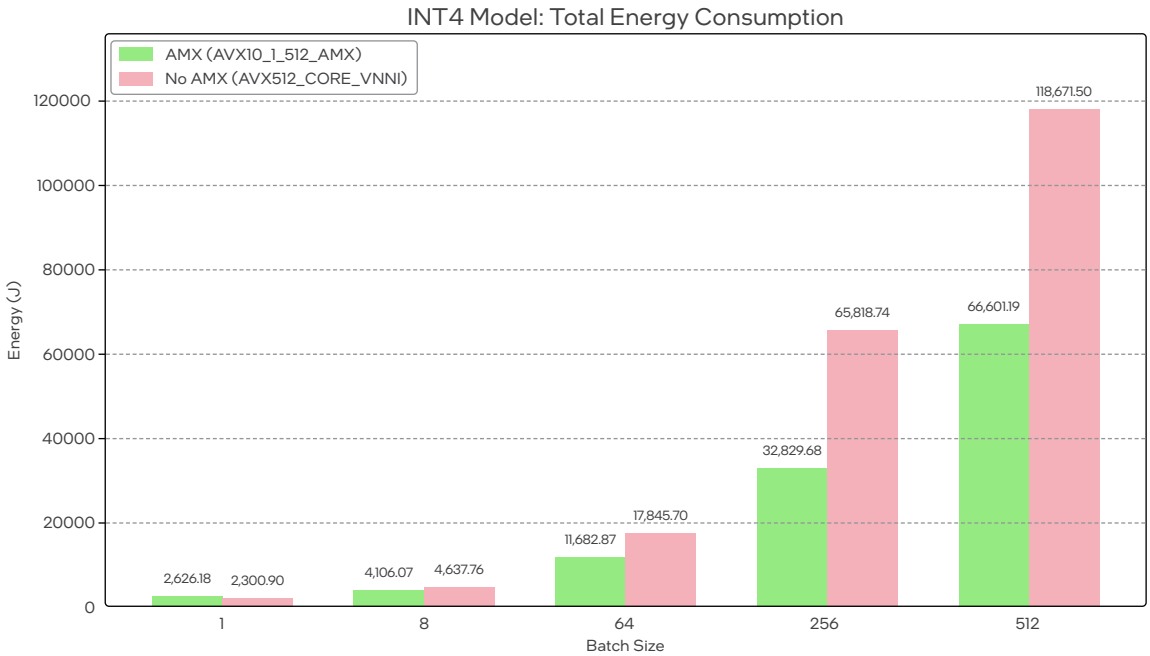

Figure 7: Int4 Energy Consumption

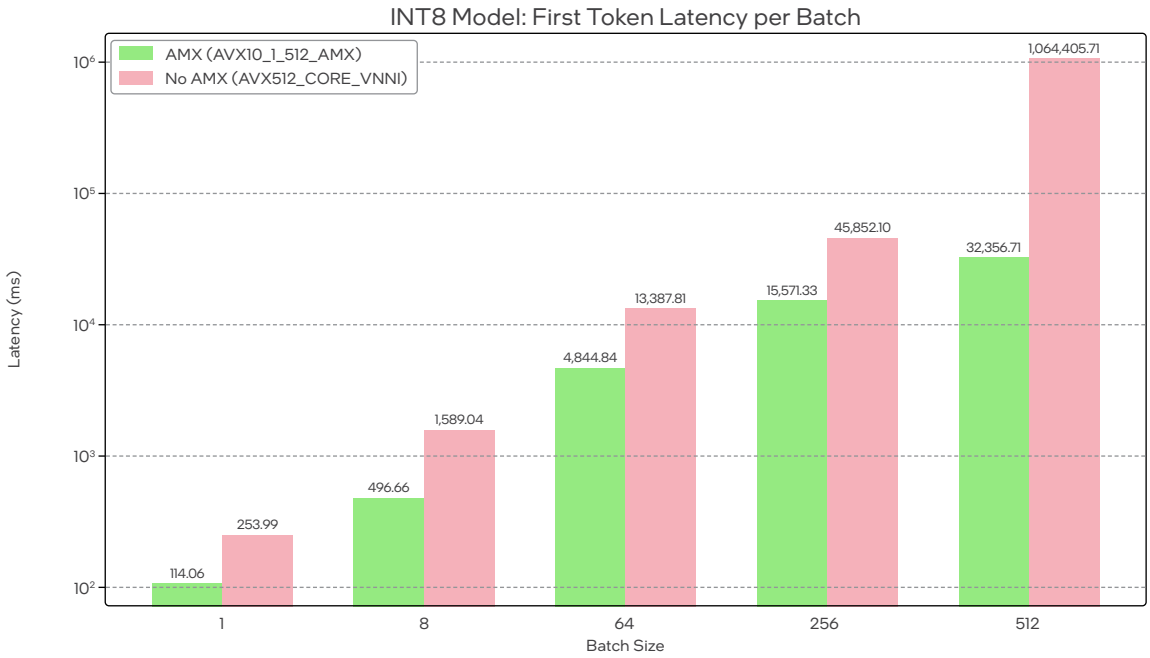

Figure 8: Int8 First Token Latency in LOG scale per Batch

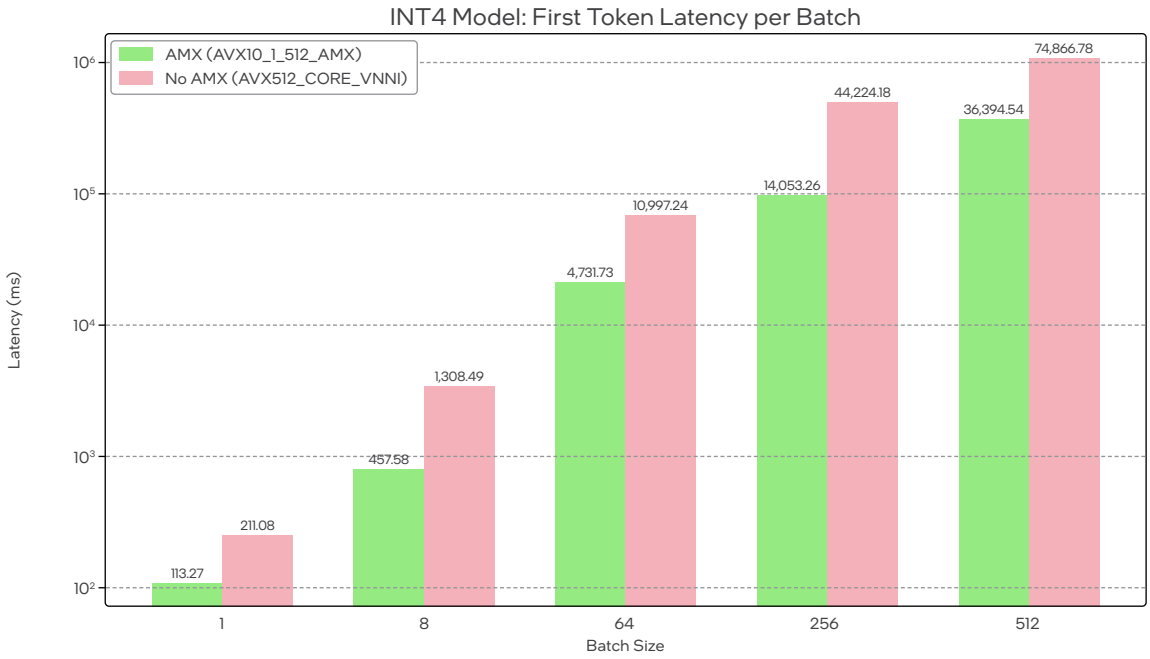

Figure 9: Int4 First Token Latency in LOG scale per Batch

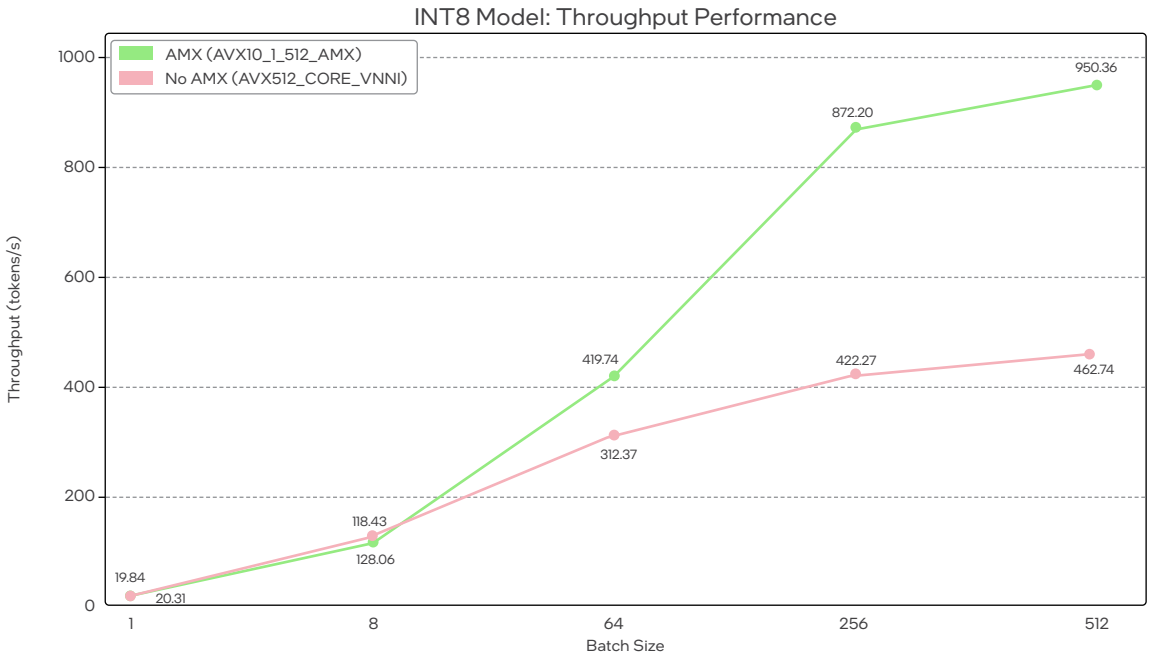

Figure 10: Int8 Throughput

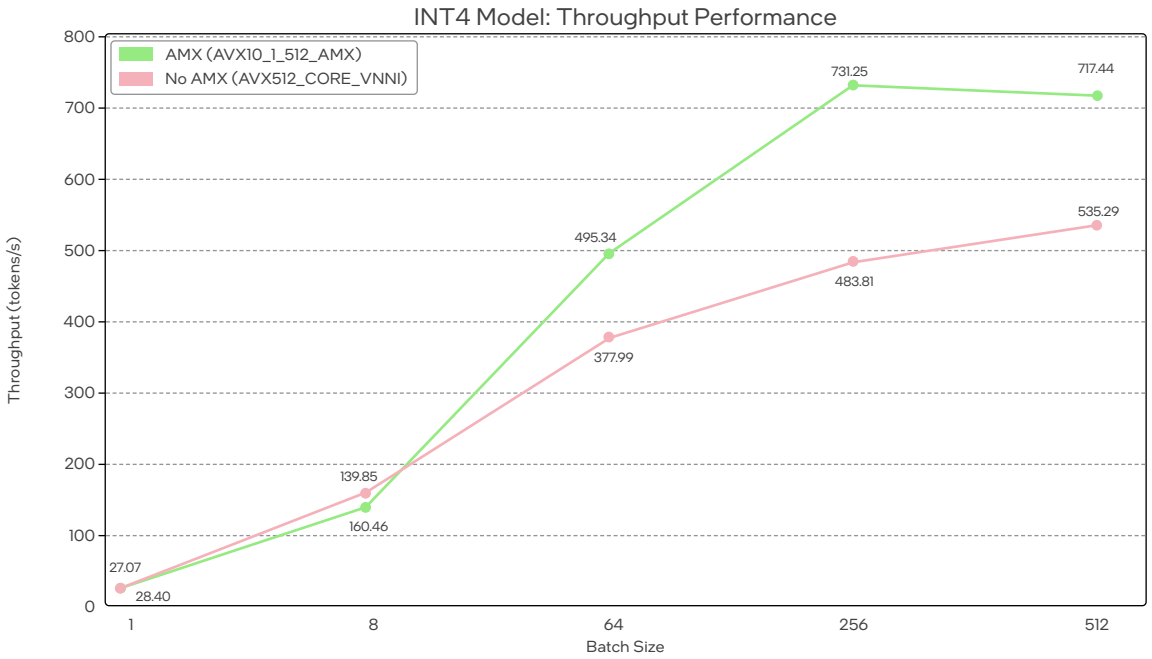

Figure 11: Int4 Throughput

4 Performance Analysis

Our benchmark evaluation of the Llama-2-7b-chat-hf model, comparing Intel AMX (AVX10 1 512 AMX) and AVX512 VNNI (AVX512 CORE VNNI) instruction sets across varying batch sizes and token input length, reveals several significant performance characteristics.

4.1 First-Token Latency

Intel AMX demonstrates consistent and significant advantages in first-token latency across all configurations. This latency benefit becomes increasingly notable as the batch size increases. As mentioned in Section 1.4.1, the first-token latency is calculated by fully computing the matrix multiplication of the attention mechanism (see Equation 4). The LLM processes a number of prompts equal to the batch size, computing separate attention matrices for each prompt in parallel. The total time is taken until the last attention matrix is complete in every sentence. This highlights Intel AMX strength in accelerating matrix multiplication workloads. In contrast, AVX512 VNNI operates on vectors, resulting in a more linear scaling pattern with increasing batch size. The latency reduction offered by Intel AMX is particularly valuable for interactive applications that require rapid initial responses.

4.2 Energy Efficiency

Energy consumption measurements reveal an interesting efficiency pattern. At minimal batch sizes (particularly batch size 1), Intel AMX exhibits marginally higher energy consumption than AVX512 VNNI, likely attributable to initialization overhead of the specialized instruction set, loading to the tile register and moving data to Intel AMX accelerator. However, as batch size increases, Intel AMX demonstrates progressively superior energy efficiency. These efficiency gains have significant implications for operational costs and environmental impact in large-scale deployments. The power consumption of both Intel AMX and AVX512 VNNI does not differ across batch size and input token length.

4.3 Throughput Performance

Throughput measurements from the second generated token and it shows a change depending on the batch size. As explain in Subsection 2.1, we use batch size to exploit multi-threading in the CPU. For small batches (1-8), AVX512 VNNI and Intel AMX perform similarly. However, as the batch size increases, Intel AMX significantly outperforms AVX512 VNNI. This performance gap highlights Intel AMX optimization for parallelized matrix operations, which becomes increasingly beneficial with larger computational workloads.

4.4 Quantization Impact

INT8 quantization consistently achieves higher throughput than INT4 across both Intel AMX and AVX-512 VNNI instruction sets at similar batch sizes. However, the relative performance advantage of Intel AMX over AVX-512 is reduced when using INT4 quantization. This is primarily because Intel AMX natively supports only INT8 and BF16 data types. As a result, when using INT4 weights, the data must first be cast to INT8 before computation can occur. After matrix multiplication, the results are then re-quantized back to INT4 for storage. This additional casting step introduces overhead, diminishing the potential throughput gains. These findings suggest that while lower-bit quantization (such as INT4) can reduce memory and compute requirements, it may also offset the benefits of hardware accelerators that do not natively support such format.

5 Conclusion

GenAI and LLMs are potential topics in business, industry and academic domains. Using GPUs for LLMs is not mandatory; especially for GenAI models that are less than 10B parameters. Intel AMX is a Intel Xeon CPU technology that can accelerate AI models training and inference performance. In this research, we presented Intel AMX value proposition for a GenAI small model; which is Llama2-7b-chat LLM model. The study results show Intel AMX can optimize GenAI model inference performance and energy consumption. Characteristics reveal a clear dependency on batch size. Without Intel AMX, throughput is similar for smaller batches (1-8), while Intel AMX outperforms at larger batch sizes (64, 256, and 512). The performance differ between batch sizes 8 and 64 suggests that Intel AMX incurs some initialization and context-switching overhead, which becomes less noticeable with larger workloads. Intel AMX also consistently provides better energy efficiency, consuming 10-60% less energy than without Intel AMX across all configurations, along with reduced first-token latency. These results suggest that Intel AMX is an potential AI accelerator for GenAI small models that are less than 20B parameters. Particularly for smaller LLMs, where it offers great energy efficiency, low latency, and optimized performance across a range of batch sizes, making it an attractive option for energy-conscious deployments.

Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more on the Performance Index site.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure. Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Your costs and results may vary. Workloads and configurations. Results may vary.

Intel technologies may require enabled hardware, software or service activation.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Appendix

Testing Hardware Configuration:

Dual socket server with Intel Xeon Platinum 8460Y+ (codename Sapphire Rapids). The CPU has 40 physical cores and 80 threads with 2.0 GHz Base-frequency [5]. Each processor has a Thermal Design Power (TDP) rating of 300W (600W total for the dual-socket system).

Storage was provided by a 960 GB Intel SATA SSD (model:NTEL SSDSC2KG96), and the server was connected to a 10G network.

Testing Software Configuration:

The operating system used was Ubuntu 24.04 LTS, running on the GNU/Linux 6.8.0-55-generic x86 64 kernel. For energy measurements, we use Intel's Performance Counter Monitor (PCM) framework.

We choose a light weight, open source LLM model for inference: Llama2-7b-chat. For the inference process, we apply weight-only quantization to convert the model weights from Float32 to Int8 and Int4, which are supported by both Intel AMX and AVX-512 VNNI. Specifically, for AVX-512 VNNI, the weights are multiplied in the Int8/Int4 format and accumulated using the Float32 activation function.

For Intel AMX, the weights are multiplied in Int8/Int4 format and accumulated using the BF16 activation function, which is supported by Intel AMX. The model is then compiled to the OpenVINO IR format to leverage the full potential of Intel hardware using OpenVINO. OpenVINO is an open- source toolkit developed by Intel for optimizing and deploying deep learning models, especially for inference on Intel hardware. We then measure the total time for the model to run inference at a specific time point, which corresponds to a noticeable energy surge, as recorded in the PCM logs. The total time from this point until inference completion is used to calculate the overall energy consumption during the inference process.

The experiment measures performance across different batch sizes (1, 8, 64, 256, 512) and input sequence lengths (64, 128, 256, 512 tokens) to evaluate how the model handles different workload.

Bibliography

[1] Intel CORPORATION. *Accelerate AI Workloads with Intel Advanced Matrix Extensions*. URL: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-12/accelerate-ai-with-amx-sb.pdf.

[2] Intel CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. Latest Edition. Accessed: 09-Mar-2025. 2025. Chap. 19. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[3] Tim DETTMERS. *Effective Weight-Only Quantization for Large Language Models*. Intel Community Blog. Accessed: 2025-03-10. 2023. URL: https://community.intel.com/t5/Blogs/Tech-Innovation/Artificial-Intelligence-AI/Effective-Weight-Only-Quantization- for-Large-Language-Models/post/1529552.

[4] Jun HAN. *TinyML and Efficient Deep Learning Computing*. Massachusetts Institute of Technology. 2024. URL: https://hanlab.mit.edu/courses/ 2024-fall-65940.

[5] INTEL. *Intel Xeon Platinum 8460Y Processor - Specifications*. Accessed: 2025-03-10. 2023. URL: https://www.intel.de/content/www/de/de/products/sku/231736/intel-xeon-platinum-8460y-processor-105m-cache-2-00-ghz/specifications.html.

[6] INTEL. *Running Average Power Limit Energy Reporting*. Accessed: 2025- 03-10. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[7] INTEL. *What is Intel® Advanced Matrix Extensions (Intel® AMX)?* Accessed: 09-Mar-2025. 2025. URL: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-amx.html.

[8] Taku KUDO and John RICHARDSON. *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*. 2018. arXiv: 1808.06226 [cs.CL]. URL: https://arxiv.org/abs/1808.06226.

[9] Eldar KURTIC et al. "Give Me BF16 or Give Me Death? Accuracy- Performance Trade-Offs in LLM Quantization". In: arXiv preprint arXiv:2304.00251 (2023). Accessed: 2025-03-10. URL: https://arxiv.org/abs/2411.02355.

[10] Mike LEWIS et al. BART: *Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. 2019. arXiv: 1910.13461 [cs.CL]. URL: https://arxiv.org/abs/1910.13461.

[11] Humza NAVEED et al. "A Comprehensive Overview of Large Language Models". In: arXiv preprint (2024). arXiv: https://arxiv.org/abs/2307.06435 [cs.CL].

[12] Colin RAFFEL et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2023. arXiv: 1910.10683 [cs.LG]. URL: https://arxiv.org/abs/1910.10683.

[13] Noam SHAZEER. *GLU Variants Improve Transformer*. 2020. arXiv: 2002. 05202 [cs.LG]. URL: https://arxiv.org/abs/2002.05202.

[14] Jianlin SU et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: 2104.09864 [cs.CL]. URL: https://arxiv. org/abs/2104.09864.

[15] Hugo TOUVRON et al. *Llama: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[16] Ashish VASWANI et al. "Attention Is All You Need". In: arXiv preprint arXiv:1706.03762 (2017). URL: https://arxiv.org/abs/1706.03762.

[17] Gokul YENDURI et al. *Generative Pre-trained Transformer: A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions*. 2023. arXiv: 2305.10435 [cs.CL]. URL: https://arxiv.org/abs/2305.10435.

[18] Biao ZHANG and Rico SENNRICH. *Root Mean Square Layer Normalization*. 2019. arXiv: 1910.07467 [cs.LG]. URL: https://arxiv.org/abs/1910.07467.

[19] Zixuan ZHOU et al. "A Survey on Efficient Inference for Large Language Models". In: arXiv preprint arXiv:2404.14294 (2024). URL: https://arxiv.org/abs/2404.14294.

[20] https://cdrdv2-public.intel.com/817369/Intel-GenAI-Infographic_Final.pdf

[21] What Is Intel® Advanced Matrix Extensions (Intel® AMX)? https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-amx.html

[22] https://www.plainconcepts.com/maximizing-llms-performance-intel-cpus/