NEUROMORPHIC PRINCIPLES FOR EFFICIENT LARGE LANGUAGE MODELS ON INTEL LOIHI 2

Steven Abreu
University of Groningen & Intel Labs
s.abreu@rug.nl

Sumit Bam Shrestha
Intel Labs
sumit.bam.shrestha@intel.com

Rui-Jie Zhu
University of California, Santa Cruz
rzhu48@ucsc.edu

Jason Eshraghian
University of California, Santa Cruz
jsn@ucsc.edu

ABSTRACT

Large language models (LLMs) deliver impressive performance but require large amounts of energy. In this work, we present a MatMul-free LLM architecture adapted for Intel's neuromorphic processor, Loihi 2. Our approach leverages Loihi 2's support for low-precision, event-driven computation and stateful processing. Our hardware-aware quantized model on GPU demonstrates that a 370M-parameter MatMul-free model can be quantized with no accuracy loss. Based on preliminary results, we report up to 3× higher throughput with 2× less energy, compared to transformer-based LLMs on an edge GPU, with significantly better scaling. Further hardware optimizations will increase throughput and decrease energy consumption. These results show the potential of neuromorphic hardware for efficient inference and pave the way for efficient reasoning models capable of generating complex, long-form text rapidly and cost-effectively.

1 INTRODUCTION

Large language models (LLMs) have revolutionized machine learning—but their computational and energy demands are enormous. This challenge motivates the development of efficient and scalable foundation models that are optimized not only algorithmically but co-designed with novel hardware architectures. In this paper, we propose a hardware-aware approach that integrates an efficient LLM architecture with Intel's neuromorphic processor, Loihi 2 (Davies et al., 2021). Although originally designed for event-based, sparse computations in spiking neural networks, Loihi 2's support for low-precision arithmetic and unstructured weight sparsity makes it an attractive platform for reducing energy consumption and latency in LLM inference. See Appendix A.1 for more details on Loihi 2.

Although LLMs have been dominated by self-attention (Vaswani et al., 2017) with quadratic runtime complexity, LLMs based on state space models (SSMs) offer linear scaling with competitive performance (Gu & Dao, 2023). SSMs rely on element-wise recurrence (Gupta et al., 2022), and use stateful neurons that align well with compute-near-memory architectures like Loihi 2. Advances in quantization of LLMs (Dettmers et al., 2022; Frantar et al., 2023; Xiao et al., 2024) have culminated in extreme quantization at scale with LLMs using binary activations (Zhu et al., 2023) or ternary weight matrices as seen in BitNet (Ma et al., 2024), piecewise affine transformers (Kosson & Jaggi, 2023), ShiftAddLLM (You et al., 2024), and earlier work on binarized neural networks (Courbariaux et al., 2016). Building on these ideas, Zhu et al. (2024) introduced the MatMul-free LLM, which replaces traditional matrix multiplications (MatMuls) with ternary matrices and element-wise operations, while also using a subquadratic SSM layer based on the HGRN model (Qin et al., 2023; 2024).

Loihi 2 is optimized for sequence-based processing, element-wise recurrence, low-precision arithmetic, and weight sparsity, all of which are features of the MatMul-free model. These benefits have been demonstrated on signal processing tasks (Orchard et al., 2021b; Shrestha et al., 2024) with orders of magnitude better latency and efficiency than state-of-the-art solutions. This paper presents a work-in-progress of adapting and deploying the MatMul-free language model from Zhu

et al. (2024) to Intel Loihi 2, opening a pathway that bridges neuromorphic computing with state-of-the-art efficient LLMs. This required a microcode implementation to map the MMF model to Loihi 2's architecture, along with a detailed ablation study to evaluate the optimal bit-precision for all operators in the MMF model. We are able to run the MMF model fully on-chip, using fixed-point arithmetic to optimize for energy and latency.

2 Model Architecture

The model architecture is based on the 370M parameter[1] MatMul-free language model (Zhu et al., 2024). It uses a combination of ternary weights and specialized layers to replace all matrix multiplications with additions, bit shifts, and elementwise operators. The overall model architecture follows the Metaformer (Yu et al., 2022) paradigm, consisting of alternating token mixers and channel mixers. Figure 1 provides a high-level overview of this structure.

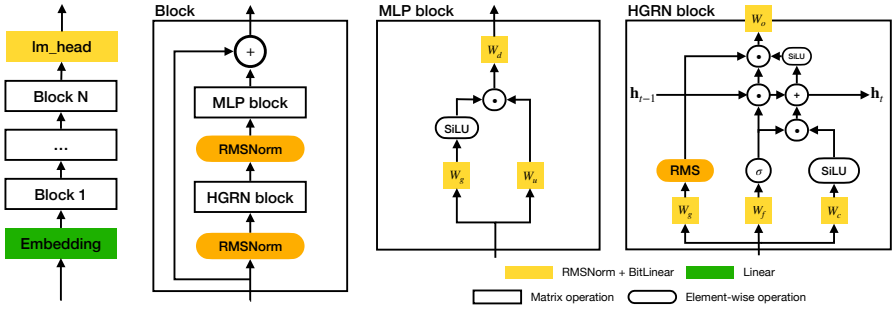

Figure 1: Model architecture of the MatMul-free language model from Zhu et al. (2024).

The model introduces two key innovations. The **BitLinear Layer** combines a ternarized linear transformation with a preceding RMSNorm operation, to stabilize the activation distribution (Ma et al., 2024; Zhang et al., 2023; Zhu et al., 2024). Formally, the input vector $x \in \mathbb{R}^d$ (where $d = 1,024$ for the 370M model), $\mathbf{W} \in \{-c, 0, c\}^{d \times d'}$ is the ternary weight matrix with a scaling factor $c \in \mathbb{R}$. The RMSNorm (Zhang & Sennrich, 2019) and subsequent BitLinear layer are implemented as:

$$\text{RMSNorm}(x; g, \epsilon) = \frac{x}{\sqrt{\epsilon + \sum_i^d x_i^2}} \odot g \tag{1}$$

$$\text{BitLinear}(x; \mathbf{W}, g, \epsilon) = \text{RMSNorm}(x; g, \epsilon) \circledast \mathbf{W} \tag{2}$$

where $\odot$ denotes element-wise multiplication, $\epsilon = 10^{-6}$ is a small constant, $g \in \mathbb{R}^d$ is a learned scaling parameter, and $\circledast$ performs the accumulation of ternary weights and inputs[2]. The use of ternary weights naturally leads to synaptic sparsity; the 370M MMF model has $35.4\% \pm 2.5\%$ weights of magnitude zero across all ternary weight matrices[3].

The MatMul-free language model further uses a BitLinear version of the GLU (Gated Linear Unit (Dauphin et al., 2017)):

$$g_t = x_t \circledast \mathbf{W}_g, \quad u_t = x_t \circledast \mathbf{W}_u, \quad p_t = \tau(g_t) \odot u_t, \quad d_t = p_t \circledast \mathbf{W}_d$$

where $g_t, u_t, p_t \in \mathbb{R}^d$ are intermediate activations, and τ denotes the SiLU activation function $\tau(x) = x \odot \sigma(x)$ where $\sigma(x) = 1/(1 + e^x)$ is the sigmoid function.

The **MLGRU** (MatMul-free Linear Gated Recurrent Unit) acts as the token mixer, replacing the computationally expensive self-attention mechanism in traditional transformers. It utilizes a variant

[1] Available on HuggingFace: huggingface.co/ridger/MMfreeLM-370M.

[2] As shown by Kosson & Jaggi (2023), this can be done using only additions and negations.

[3] Note that this synaptic sparsity does not yield memory savings because sparse encoding of a 1024×1024 ternary matrix takes $\approx 7.5\times$ more memory than dense encoding. However, zero weights do allow for energy savings by skipping calculations.

of the GRU (Cho et al., 2014), inspired by the HGRN (Qin et al., 2023; 2024), modified to use only additions and element-wise products. This is achieved by employing BitLinear layers for all linear transformations within the MLGRU cell. Formally, we denote $h_t \in \mathbb{R}^d$ as the hidden state at time t and $x_t \in \mathbb{R}^d$ as the input at time t. The dynamics of the MLGRU layer at time step t are given by:

$$
\begin{aligned}
f_t &= \sigma\left(\text{BitLinear}(x_t; \mathbf{W}_f, g_f, \epsilon)\right), \\
c_t &= \tau\left(\text{BitLinear}(x_t; \mathbf{W}_c, g_c, \epsilon)\right), \\
h_t &= f_t \odot h_{t-1} + (1 - f_t) \odot c_t, \\
g_t &= \text{RMSNorm}\left(\text{BitLinear}(x_t; \mathbf{W}_g, g_g, \epsilon); g_{g'}, \epsilon\right), \\
o'_t &= g_t \odot \tau(h_t), \\
o_t &= \text{BitLinear}(o'_t; \mathbf{W}_o, g_o, \epsilon)
\end{aligned}
$$

where $\mathbf{W}_c, \mathbf{W}_f, \mathbf{W}_o, \mathbf{W}_g \in \mathbb{R}^{d \times d}$ are ternary weights, f_t, g_t, c_t, o'_t are intermediate activations at time step t, h_t is the hidden state, and o_t is the final output at time step t. The initial hidden state h_0 is set to zero. Similarly to the HGRN (Qin et al., 2023), the MLGRU also employs the `cummax` operation to bias the forget gate values in deeper layers closer to 1, which we omit for brevity.

3 Model Adaptation for Loihi 2

Quantization of weights and activations As a first step, we quantize the original half-precision model from Zhu et al. (2024). Loihi 2 supports 8-bit weights and 24-bit activations. As done by previous work (Ma et al., 2024; Zhu et al., 2024), we evaluate the zero-shot performance of our quantized model on a range of language tasks, including ARC (Easy & Challenge) (Clark et al., 2018), Hellaswag (Zellers et al., 2019), Winogrande (Sakaguchi et al., 2021), PIQA (Bisk et al., 2020), and OpenbookQA (Mihaylov et al., 2018). We report the baseline models from Zhu et al. (2024) for the MatMul-free LLM that we adopt, and their transformer baseline. For comparison, we also report performance of the Qwen-2.5 500M model from Qwen Team (2024). Although the model has only 35% more parameters than the transformer baseline, it performs significantly better. We expect that the baselines from Zhu et al. (2024) could reach similar performance with the training procedure from Qwen Team (2024).

Table 1: Results from quantization of the 370M MatMul-free language model on GPU. *Baseline*: optimized models from Zhu et al. (2024) and Qwen Team (2024). *PT*: PyTorch-only implementation. *Ax / Wx*: activations / RMSNorm weights quantized to x-bit integers. $\epsilon_{\text{rms}} \uparrow$: setting the value for ϵ_{rms} to 10^{-3} from previously $\epsilon_{\text{rms}} = 10^{-6}$. †: difference relative to MatMul-free baseline.

Configuration	ARCc	ARCe	HS	OQA	PQ	WG	Avg	Diff.†
MatMul-free baseline	22.8	42.1	32.4	28.4	62.6	49.4	39.6	(0.0%)
Transformer baseline	24.0	45.0	34.3	29.2	64.0	49.9	41.1	(3.8%)
Qwen2-500M	31.0	64.6	49.1	35.2	70.3	56.5	51.1	(29.0%)
PT	22.7	42.2	32.5	28.4	62.4	48.5	39.4	-0.4%
PT + W8	23.2	41.8	32.4	28.0	62.4	49.5	39.5	-0.2%
PT + A8	22.7	40.0	31.5	27.6	61.0	50.0	38.8	-2.0%
PT + A16	22.7	42.5	32.5	29.0	63.2	49.9	40.0	**0.9%**
PT + W8A8	22.3	40.3	31.9	27.2	59.9	49.1	38.5	-2.9%
PT + W8A16	22.7	42.3	32.3	28.0	63.1	49.3	39.6	0.0%
PT + W8A8 + $\epsilon_{\text{rms}} \uparrow$	28.3	26.8	26.1	27.0	52.7	51.5	35.4	-10.7%
PT + W8A16 + $\epsilon_{\text{rms}} \uparrow$	23.0	42.4	32.4	27.8	63.0	50.1	39.8%	**0.4%**

We re-implemented the model and replace the GPU-optimized Triton kernels with simple PyTorch operations that are easier to quantize and to verify against our Loihi 2 hardware implementation. Minor numerical differences add up to a relative performance drop of 0.4%, see Table 1 (MatMul-free baseline vs. PyTorch). We further quantize weights and activations, to ensure compatibility with fixed point computation, as required by Loihi 2. Quantization is applied symmetrically per tensor, with scaling factors restricted to powers of two so that rescaling can be done efficiently using

bit-shift operations. We use "fake quantization" in PyTorch, in that all operations use floating-point numbers which are quantized and de-quantized.

Our results in Table 1 show that 8-bit weight quantization leads to only a 0.2% performance decrease relative to the baseline model. Previous work has demonstrated low quantization errors for state space models when using W8A16 (8-bit weights, 16-bit activations) with significantly higher drops for W8A8 (Pierro & Abreu, 2024; Abreu et al., 2024; Chiang et al., 2024). Indeed, quantization to W8A8 and W8A16 of our model show a relative performance drop of 2.9% and 0.0%, respectively. We thus use W8A16 for our hardware implementation.

Although the baseline model uses half precision (FP16), the RMSNorm is still computed in full precision (FP32) for numerical stability. We quantize activations inside the RMSNorm layer to 24-bit integers with 12 fractional bits. We further increase the ϵ value from 10^{-6} (which underflows with 12 fractional bits) to 10^{-3}. The resulting performance of all interventions mentioned thus far is shown in the last two rows of Table 1. The W8A8 and W8A16 quantization schemes with modified ϵ show a relative performance change of -10.7% and +0.4%, respectively. We chose the W8A16 quantization scheme as it is fully compatible with Loihi 2. Therefore, our final quantized model on GPU shows no performance loss compared to the baseline FP16 model.

Fixed-point implementation Two operations used in the MatMul-free LM are not defined on integers, namely the sigmoid activation function σ and the inverse-square-root in the RMSNorm. We employ a look-up-table (LUT) for a fixed-point approximation of the logistic sigmoid function, $\sigma(x) = 1/(1 + e^{-x})$. For the inverse-square-root in the RMSNorm layer, we adapted a well-known "fast inverse square root" algorithm to operate in fixed-point arithmetic. See Appendix A.2 for details.

Mapping the model to Loihi 2 The Loihi 2 implementation represents the model as a network of neurons interconnected via synapses. Each neuron is implemented as a simple microcode program that is executed asynchronously on one of 120 neuro cores on each Loihi 2 chip, after which its output is transmitted to other neurons through synaptic connections. A global time step is maintained through barrier synchronization between all neuro cores. Since each neuron only maintains its own state, aggregate operations–such as computing the sum of squares over an activation vector–must be realized by dedicated neurons that receive inputs from all neurons within the corresponding layer. This is the case for the RMSNorm, where the sum of squares over all neurons in a given layer is calculated, see Equation 2. Figure A.5 (*right*) illustrates the computational graph that implements the RMSNorm operation on Loihi 2. Layer and operator fusion are well-established strategies to minimize redundant computations (Waeijen et al., 2021; Niu et al., 2021), thereby reducing latency and enhancing energy efficiency. In our implementation, we perform Loihi-specific layer fusion to consolidate operations into a streamlined computational graph, as depicted in Figure A.5 (*left*). We also derive the fusion of two subsequent RMSNorm layers into a single operator for further acceleration, see Appendix A.3.

4 RESULTS

We implemented a single block of the MatMul-free language model on a single Loihi 2 chip. We parallelize the workload to all 120 available neuro cores on the chip. Verification of the model on Loihi 2 indicates close alignment with the quantized PyTorch simulation.

Note: All current comparisons are performed with FP16 baselines on non-Loihi hardware.

We contrast the estimated performance of the 370M MatMul-free model on Loihi 2 against transformer-based baselines running on an NVIDIA Jetson Orin Nano. We selected the Jetson Orin Nano (8GB) as our comparison platform because it represents a state-of-the-art edge AI device with 1024 CUDA cores, 32 Tensor cores, and a maximum power consumption of 15W, making it a relevant benchmark for energy-efficient AI applications. The Orin Nano is designed specifically for edge deployment scenarios similar to those targeted by neuromorphic hardware, enabling a fair comparison between platforms intended for similar operational environments. Efficiency and throughput metrics for Loihi 2 are estimates based on a preliminary implementation that is not fully optimized, see Appendix A.4.1 for details. We compare the MatMul-free LLM on Loihi 2 against two similarly-sized transformer-based LLMs available on HuggingFace. As a Llama series rep-

resentative model, we use Alireo-400M (Montebovi, 2024), a 400M parameter transformer-based LLM with 24 layers and a context window of 8,192. It should be noted that the Alireo model was trained specifically on Italian text, so its performance is not included in Table 1 as it would not be representative of a competitive general-purpose transformer-based model at this parameter scale. We further use Qwen2.5-500M (Yang et al., 2024; Qwen Team, 2024), a 500M parameter transformer-based LLM with 24 layers and a context window of 32,768 whose performance is also included in Table 1. Both models run in half-precision (FP16). We did not benchmark the MatMul-free LLM or the Transformer baseline from Zhu et al. (2024) because the Jetson Orin Nano does not support Triton.

Table 2 shows results for the comparison of the MatMul-free LLM on Loihi 2 and transformer-based LLMs on the NVIDIA Jetson Orin Nano, also including results for the MatMul-free LLM on an H100 GPU and the Transformer++ baseline from Zhu et al. (2024) on a single H100 GPU. The results demonstrate that the MatMul-free LLM on GPU improves in throughput and efficiency with longer sequence lengths, due to linear scaling of the token mixing and better utilization of the GPU at higher sequence lengths. In contrast, the transformer++ baseline on GPU increases in throughput for short sequence lengths due to better utilization of the hardware, and then deteriorates in throughput and efficiency for even longer sequences because of the quadratic scaling of self-attention.

Table 2: Throughput and energy efficiency for two transformer-based language models running on the NVIDIA Jetson Orin Nano and H100 compared to our MatMul-free LM running on Intel's Loihi 2, across different sequence lengths for prefill and generation. The best-performing sequence length for each model and metric is underlined. Metrics for Loihi 2 are based on preliminary experiments and subject to further performance optimization, see Appendix A.4.1. **Gen**: autoregressive generation, **Prefill**: prefill mode. * Llama representative model from Montebovi (2024).

			Throughput (↑ tokens/sec)					Efficiency (↓ mJ/token)				
	Sequence length		500	1000	4000	8000	16000	500	1000	4000	8000	16000
Generate	**MMF (370M)**	**Loihi 2**†	**41.5**	**41.5**	**41.5**	**41.5**	**41.5**	**405**	**405**	**405**	**405**	**405**
	MMF (370M)	H100	13.4	13.3	13.5	13.2	13.5	10.1k	10.1k	10.0k	9.9k	9.8k
	TF++ (370M)	H100	22.4	22.9	21.7	21.3	20.9	5.5k	5.6k	6.2k	6.8k	8.2k
	Llama* (400M)	Jetson‡	14.3	14.9	14.7	15.2	12.8	723	719	853	812	1.2k
	Qwen2 (500M)	Jetson‡	13.4	14.0	14.1	15.4	12.6	791	785	912	839	1.2k
Prefill	**MMF (370M)**	**Loihi 2**†	**6632**	**6632**	**6632**	**6632**	**6632**	**3.7**	**3.7**	**3.7**	**3.7**	**3.7**
	MMF (370M)	H100	11.4k	13.1k	30.6k	51.6k	84.6k	6.1	5.3	2.5	1.4	0.9
	TF++ (370M)	H100	21.6k	32.7k	44.3k	55.4k	60.5k	11.3	7.3	5.4	4.3	3.8
	Llama* (400M)	Jetson‡	849	1620	3153	2258	1440	11.7	7.8	6.8	7.6	11.5
	Qwen2 (500M)	Jetson‡	627	909	2639	3861	3617	17.9	13.9	6.7	4.4	5.3

† The MatMul-free LM on Loihi 2 was characterized on a 32-chip Alia Point Loihi 2 system (N3C1 silicon) running NxKernel v0.2.0 and NxCore v2.5.8 (accessible to Intel Neuromorphic Research Community members). Appendix A.4.1 compares results for single-chip and multi-chip scaling.

‡ Transformer LMs were characterized on NVIDIA Jetson Orin Nano 8GB using the MAXN power mode running Jetpack 6.2, TensorRT 10.3.0, CUDA 12.4. Energy values include CPU_GPU_CV, SOC, and IO components as reported by jtop 4.3.0.

Performance results are based on testing as of Jan 2025 and may not reflect all publicly available security updates. Results may vary.

In our experiments, we focus on single-batch inference and we further differentiate between two operational modes. "Prefill" refers to the phase where a long input sequence is ingested, allowing for parallel processing of multiple tokens, which naturally yields higher throughput and energy efficiency. In contrast, "autoregressive generation" denotes the sequential production of tokens, where each token must be generated and received before the next can be processed. Profiling both modes separately highlights these differences in performance and efficiency. The execution modes that enable this distinct processing on Loihi 2 are described in Appendix A.1.1.

During *prefill*, Loihi 2 shows at least **2× higher throughput** with approximately **2× less energy** per token. During *auto-regressive generation*, the advantage of Loihi 2 grows to having almost **3× higher throughput** with approximately **2× less energy** per token. Due to the MatMul-free model's subquadratic architecture, this advantage is expected to grow for longer sequence lengths. We highlight that power and throughput of the MatMul-free model on Loihi 2 is *constant across sequence length*. This is due to the linear scaling of the recurrent token mixer and the fact that Loihi 2 has its parameters and hidden states stored locally inside neuro-cores thus requiring significantly

less memory movement than GPUs. Further optimizations are underway to increase throughput and reduce power consumption of the MatMul-free language model on Loihi 2, see Appendix A.4.

Table 2 highlights how transformers on an edge GPU are consistently slower and less efficient during generation, and scale unfavorably to longer sequence lengths. During prefill, transformers show low throughput and efficiency for shorter sequences due to under-utilization of the GPU, then reach optimal performance between 4000-8000 tokens, after which both throughput and efficiency decline. Table 2 also compares the MatMul-free model on Loihi against the same model running on an H100 GPU. Loihi 2 delivers $\mathbf{3}\times$ **higher throughput** during generation with at least $\mathbf{14}\times$ **less energy per token**. During prefill, the H100 delivers higher throughput than Loihi 2, but outperforms in energy efficiency only for large sequence lengths.

Latency characteristics are particularly important for interactive applications at the edge. Our experiments reveal that for batch size 1, which is typical in edge deployment scenarios, the MatMul-free model on Loihi 2 demonstrates significant latency advantages, with a $6.6\times$ **lower time-to-first token** on a 500-token input sequence (99ms for the MatMul-free model on the Loihi 2 vs. 659ms for the Llama-style model on the Jetson). This advantage increases with sequence length due to the linear scaling properties of our approach versus the quadratic complexity of transformer models. For real-time applications like voice assistants or mobile chatbots, this latency reduction directly translates to more responsive user experiences while maintaining significantly lower energy consumption.

While we benchmarked against the Jetson Orin Nano as a representative edge GPU platform, we expect our approach to show similar advantages compared to other edge computing solutions. Platforms such as mobile SoCs, FPGAs, and various edge TPUs all face similar challenges with transformer-based models: quadratic scaling with sequence length and significant memory movement costs. Since our neuromorphic approach addresses both fundamental limitations through linear scaling and closer compute-memory integration, we anticipate the relative throughput and energy efficiency advantages to persist across other edge computing architectures. The benefits would be most pronounced for resource-constrained IoT or mobile platforms where energy efficiency is paramount, with potentially more modest gains against specialized NPUs that have already optimized for recurrent operations.

5 Conclusion

We demonstrated that neuromorphic principles and co-design can be leveraged to build efficient LLMs. By merging a MatMul-free architecture with Intel's Loihi 2–leveraging state-based computation, extreme quantization, and operator fusion–we built a strong 370M-parameter model with significantly improved throughput and energy efficiency. Our experiments reveal that the inherent parallelism and low-power processing of Loihi 2 translate into substantial gains in throughput and energy efficiency.

The key innovations of our approach include: (1) the first demonstration of a modern LLM architecture on neuromorphic hardware, establishing a pathway for efficient AI at the edge; (2) a hardware-aware quantization methodology that maintains model accuracy while enabling fixed-point computation; (3) a novel microcode implementation of the MatMul-free architecture that exploits Loihi 2's asynchronous, event-driven computing paradigm; and (4) custom operator fusion techniques specifically designed for neuromorphic computation, including our double RMSNorm derivation. Unlike previous approaches that targeted specific neural primitives for neuromorphic systems, our work shows that complete, competitive language models can be adapted to leverage the unique characteristics of neuromorphic hardware while maintaining performance.

On the hardware side, our results demonstrate the potential of neuromorphic processors as platforms for scalable efficient inference, suggesting that future architectures can be co-designed with model innovations to further push performance limits. Our approach offers a promising pathway to enable adaptive language processing without the prohibitive energy costs associated with traditional LLMs. Given the rising importance of reasoning models that require extended chain-of-thought rollouts (DeepSeek-AI et al., 2025), efficient and high-throughput autoregressive generation is more critical than ever. Our design excels especially in this mode, paving the way for scalable foundation models capable of reasoning faster and more efficiently.

REFERENCES

Steven Abreu, Jens E. Pedersen, Kade M. Heckel, and Alessandro Pierro. Q-S5: Towards Quantized State Space Models, June 2024. URL http://arxiv.org/abs/2406.09477. arXiv:2406.09477 [cs].

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about Physical Commonsense in Natural Language. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):7432–7439, April 2020. ISSN 2374-3468. doi: 10.1609/aaai.v34i05.6239. URL https://ojs.aaai.org/index.php/AAAI/article/view/6239. Number: 05.

Hung-Yueh Chiang, Chi-Chih Chang, Natalia Frumkin, Kai-Chiang Wu, and Diana Marculescu. Quamba: A Post-Training Quantization Recipe for Selective State Space Models, October 2024. URL http://arxiv.org/abs/2410.13229. arXiv:2410.13229.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans (eds.), *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1179. URL https://aclanthology.org/D14-1179.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge, March 2018. URL http://arxiv.org/abs/1803.05457. arXiv:1803.05457.

Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1, March 2016. URL http://arxiv.org/abs/1602.02830. arXiv:1602.02830 [cs].

Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 933–941, Sydney, NSW, Australia, August 2017. JMLR.org.

Mike Davies, Andreas Wild, Garrick Orchard, Yulia Sandamirskaya, Gabriel A. Fonseca Guerra, Prasad Joshi, Philipp Plank, and Sumedh R. Risbud. Advancing Neuromorphic Computing With Loihi: A Survey of Results and Outlook. *Proceedings of the IEEE*, 109(5):911–934, May 2021. doi: 10.1109/jproc.2021.3067593. Publisher: Institute of Electrical and Electronics Engineers (IEEE).

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia

Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning, January 2025. URL http://arxiv.org/abs/2501.12948. arXiv:2501.12948 [cs].

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale, November 2022. URL http://arxiv.org/abs/2208.07339. arXiv:2208.07339 [cs].

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers, March 2023. URL http://arxiv.org/abs/2210.17323. arXiv:2210.17323 [cs].

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal state spaces are as effective as structured state spaces. *Advances in Neural Information Processing Systems*, 35:22982–22994, 2022.

Atli Kosson and Martin Jaggi. Multiplication-Free Transformer Training via Piecewise Affine Operations, October 2023. URL http://arxiv.org/abs/2305.17190. arXiv:2305.17190 [cs].

Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits, February 2024. URL http://arxiv.org/abs/2402.17764. arXiv:2402.17764 [cs].

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a Suit of Armor Conduct Electricity? A New Dataset for Open Book Question Answering. In *EMNLP*, 2018.

Michele Montebovi. Alireo-400m: A lightweight italian language model, 2024. URL https://huggingface.co/DeepMount00/Alireo-400m-instruct-v0.1.

Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 883–898, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454083. URL https://doi.org/10.1145/3453483.3454083.

Garrick Orchard, E Paxon Frady, Daniel Ben Dayan Rubin, Sophia Sanborn, Sumit Bam Shrestha, Friedrich T Sommer, and Mike Davies. Efficient neuromorphic signal processing with loihi 2. In *2021 IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 254–259. IEEE, 2021a.

Garrick Orchard, E. Paxon Frady, Daniel Ben Dayan Rubin, Sophia Sanborn, Sumit Bam Shrestha, Friedrich T. Sommer, and Mike Davies. Efficient Neuromorphic Signal Processing with Loihi 2. In *2021 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, October 2021b. doi: 10.1109/sips52927.2021.00053.

Alessandro Pierro and Steven Abreu. Mamba-PTQ: Outlier Channels in Recurrent Large Language Models, July 2024. URL http://arxiv.org/abs/2407.12397. arXiv:2407.12397 [cs].

Zhen Qin, Songlin Yang, and Yiran Zhong. Hierarchically Gated Recurrent Neural Network for Sequence Modeling. *Advances in Neural Information Processing Systems*, 36:33202–33221, December 2023.

Zhen Qin, Songlin Yang, Weixuan Sun, Xuyang Shen, Dong Li, Weigao Sun, and Yiran Zhong. HGRN2: Gated Linear RNNs with State Expansion, April 2024. URL http://arxiv.org/abs/2404.07904. arXiv:2404.07904 [cs].

Qwen Team. Qwen2.5: A party of foundation models, September 2024. URL https://qwenlm.github.io/blog/qwen2.5/.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. WinoGrande: an adversarial winograd schema challenge at scale. *Commun. ACM*, 64(9):99–106, August 2021. ISSN 0001-0782. doi: 10.1145/3474381. URL https://dl.acm.org/doi/10.1145/3474381.

Sumit Bam Shrestha, Jonathan Timcheck, Paxon Frady, Leobardo Campos-Macias, and Mike Davies. Efficient Video and Audio Processing with Loihi 2. In *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 13481–13485, April 2024. doi: 10.1109/ICASSP48485.2024.10448003. URL https://ieeexplore.ieee.org/abstract/document/10448003. ISSN: 2379-190X.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. June 2017. URL https://nlp.seas.harvard.edu/2018/04/03/attention.html. _eprint: 1706.03762.

Luc Waeijen, Savvas Sioutas, Maurice Peemen, Menno Lindwer, and Henk Corporaal. Convfusion: A model for layer fusion in convolutional neural networks. *IEEE Access*, 9:168245–168267, 2021. doi: 10.1109/ACCESS.2021.3134930.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models, March 2024. URL http://arxiv.org/abs/2211.10438. arXiv:2211.10438 [cs].

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

Haoran You, Yipin Guo, Yichao Fu, Wei Zhou, Huihong Shi, Xiaofan Zhang, Souvik Kundu, Amir Yazdanbakhsh, and Yingyan Celine Lin. ShiftAddLLM: Accelerating Pretrained LLMs via Post-Training Multiplication-Less Reparameterization, July 2024. URL http://arxiv.org/abs/2406.05981. arXiv:2406.05981 [cs] version: 3.

Weihao Yu, Mi Luo, Pan Zhou, Chenyang Si, Yichen Zhou, Xinchao Wang, Jiashi Feng, and Shuicheng Yan. Metaformer is actually what you need for vision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 10819–10829, June 2022.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a Machine Really Finish Your Sentence? In Anna Korhonen, David Traum, and Lluís Màrquez (eds.), *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4791–4800, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1472. URL https://aclanthology.org/P19-1472.

Biao Zhang and Rico Sennrich. Root Mean Square Layer Normalization. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Yichi Zhang, Ankush Garg, Yuan Cao, Lukasz Lew, Behrooz Ghorbani, Zhiru Zhang, and Orhan Firat. Binarized Neural Machine Translation, February 2023. URL http://arxiv.org/abs/2302.04907. arXiv:2302.04907 [cs].

Rui-Jie Zhu, Qihang Zhao, Guoqi Li, and Jason K Eshraghian. Spikegpt: Generative pre-trained language model with spiking neural networks. *arXiv preprint arXiv:2302.13939*, 2023.

Rui-Jie Zhu, Yu Zhang, Ethan Sifferman, Tyler Sheaves, Yiqiao Wang, Dustin Richmond, Peng Zhou, and Jason K. Eshraghian. Scalable MatMul-free Language Modeling, 2024. URL http://arxiv.org/abs/2406.02528. arXiv:2406.02528 [cs].

A APPENDIX

A.1 LOIHI 2 HARDWARE ARCHITECTURE

Loihi 2 is the second-generation of Intel's neuromorphic research processor that was designed for sparse, event-based neural networks (Orchard et al., 2021a). On the Loihi 2 chip, a neural network is processed by massively parallel compute units called *neuro-cores*, with 120 such neuro-cores per chip. Multiple Loihi 2 chips can be stacked together into various larger systems with up to 1,152 chips, see Figure 2. The neuro-cores compute and communicate asynchronously, but a global algorithmic time step is maintained through barrier synchronization. The neuro-cores are co-located with memory and can thus efficiently update local states, simulating up to 8192 stateful neurons per core. Each neuron can be programmed by the user to realize a variety of temporal dynamics through assembly code, and can use a variable amount of memory–each neuro-core has a fixed amount of memory but one can implement neurons with more memory by trading off the number of neurons that each core implements. Input from and output to external hosts and sensors is provided with up to 160 million 32-bit integer messages per second (Shrestha et al., 2024). Loihi 2 can scale to real-world workloads of various sizes with up to 1 billion neurons and 128 billion synapses, using fully-digital stacked systems (Hala Point, Figure 2).

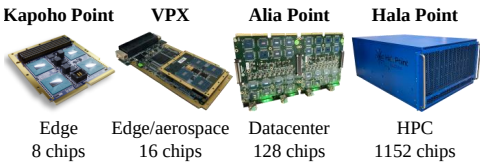

Figure 2: Different Loihi 2 systems are available to cover a wide range of applications from the edge to HPC with up to 1 billion neurons.

The architectural features of Loihi 2 offer unique opportunities to compress and optimize deep learning models. Neural networks running on Loihi must be quantized to low-precision using fixed-point arithmetic–8 bits for synaptic weights[4] and up to 32 bits for messages[5]. Unlike GPUs, Loihi 2 is optimized for computations local within neurons, a common focus of neuromorphic processors. First, it allows fast and efficient updates of neuronal states with recurrent dynamics with minimal data movement, due to the local memory of each neuro-core. Second, the asynchronous event-driven architecture of Loihi 2 allows it to efficiently process unstructured sparse weight matrices. Third, the neuro cores can leverage sparsified activation between neurons, as the asynchronous communication transfers only non-zero messages.

A.1.1 EXECUTION MODES ON LOIHI 2

Loihi 2's asynchronous architecture enables a trade off between throughput and latency, as illustrated in Figure 3. For optimal throughput, new input is provided every time step and forwarded through the neuronal layers in a *pipelined mode*. For optimal latency we use *fall-through mode*, where new input is injected only once the previous input has been processed by, or fallen through, the network as fast as possible. Naturally, when processing a long sequence of prefill text in an LLM, we use pipeline mode for optimal throughput. When generating new text in auto-regressive generation of an LLM, we have to wait for the token at time t to be output before we can begin processing the next token at time $t + 1$, thus making this a natural fit for Loihi 2's fall-through mode.

[4]This is not a hard limit, as one can implement an $8n$-bit synapse through n separate 8-bit synapses that are added together with different fixed-point exponents.

[5]Local states are not restricted in precision, and one may also transmit messages with more than 32 bits in an analogous way to what is described above for synaptic weights.

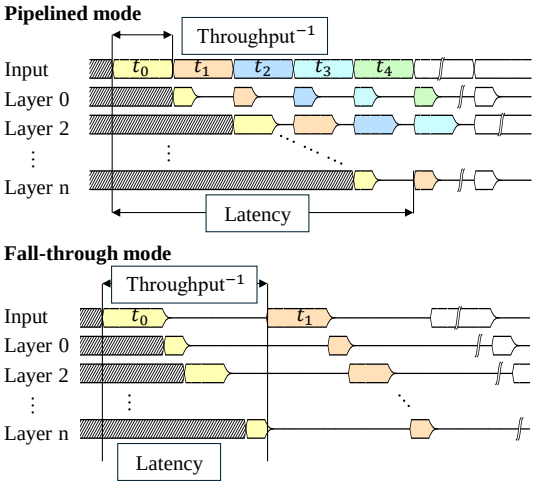

Figure 3: Different execution modes on Loihi 2 that either optimize throughput or latency. In the *pipelined mode*, a new data point is inserted in each time step, to use all processing cores and maximize the throughput–at the expense of latency because equal time bins $t_0 = t_1 = \dots$ are enforced. In the *fall-through mode*, a new data points is only provided once the last data point has been fully processed with minimum latency. Only a single neuronal layer is active at any step as data travels through the network. The time per step is thus minimized as traffic is reduced and potentially more complex neuronal layers are not updated.

A.2 Fixed-Point Implementation Details

A.2.1 Fixed-Point Implementation of the Sigmoid Function

We employ a look-up-table (LUT) for a fixed-point approximation of the logistic sigmoid function, $\sigma(x) = 1/(1 + e^{-x})$, as discussed in Section 3. Specifically, we scale the floating-point input x by $2^{x_{\text{exp}}}$ where $x_{\text{exp}} = 6$ determines the accepted input precision of the fixed-point sigmoid implementation. We quantize $x2^{x_{\text{exp}}}$ to an integer domain $x_{\text{fxp}} = \lfloor x2^{x_{\text{exp}}} \rceil$, and store precomputed values in a LUT $\left(x_{\text{int}}^{\text{lut,i}} \mapsto y_{\text{int}}^{\text{lut,i}}\right)_{i \in \{0,\dots N_\sigma\}}$, where $N_\sigma = 8$ determines the number of LUT entries:

$$y_{\text{int}}^{\text{lut,i}} = \lfloor \sigma\left(\frac{x_{\text{int}}^{\text{lut,i}}}{2^{x_{\text{exp}}}}\right) \cdot 2^{y_{\text{exp}}} \rfloor \tag{3}$$

This LUT stores only entries for positive inputs. For negative inputs, we exploit $\sigma(-x) = 1 - \sigma(x)$, thus requiring only half-range values. During inference, a piecewise linear interpolation between adjacent LUT entries refines the output. This design offers efficient computation and controllable approximation error by tuning x_{exp} and N_σ.

A.2.2 Fixed point implementation of the inverse square root

For the inverse-square-root in the RMSNorm layer, we adapted a well-known "fast inverse square root" algorithm `FastInvSqrt` to operate entirely in fixed-point arithmetic, as discussed in Section 3. Our method treats the input $\tilde{x}$ as an integer paired with a fixed exponent, and uses a LUT with 24 values to produce an initial guess for $\sqrt{\tilde{x}}$. This estimate is then refined using five iterations of the Newton-Raphson method, all in a fixed-point format.

A.3 DOUBLE RMSNORM DERIVATION

Let $x \in \mathbb{R}^d$ and $y = \text{RMSNorm}(x)$ and $z = \text{RMSNorm}(y)$, in expanded form:

$$y = \frac{x}{\sqrt{\epsilon + \sum_i^d x_i^2}} \odot g_1 \tag{4}$$

$$z = \frac{y}{\sqrt{\epsilon + \sum_i^d y_i^2}} \odot g_2 \tag{5}$$

We wish to derive an equation for $z = \text{DoubleRMSNorm}(x) = \text{RMSNorm}(\text{RMSNorm}(x))$.

First, we express $\mu_{\mathbf{y}}$ in terms of $\mu_{\mathbf{x}}$:

$$\mu_{\mathbf{y}} = \frac{1}{D}\sum_{i=1}^{D} y_i^2 = \frac{1}{D}\sum_{i=1}^{D}\left(x_i \cdot \frac{g_1}{\sqrt{\mu_{\mathbf{x}} + \varepsilon}}\right)^2 \tag{6}$$

$$= \left(\frac{g_1^2}{\mu_{\mathbf{x}} + \varepsilon}\right) \cdot \left(\frac{1}{D}\sum_{i=1}^{D} x_i^2\right) = \frac{g_1^2 \mu_{\mathbf{x}}}{\mu_{\mathbf{x}} + \varepsilon}. \tag{7}$$

We then express $\mathbf{z}$ in terms of $\mathbf{x}$ by plugging in the equation for $\mathbf{y}$:

$$\mathbf{z} = \mathbf{y} \cdot \frac{g_2}{\sqrt{\mu_{\mathbf{y}} + \varepsilon}} = \left(\mathbf{x} \cdot \frac{g_1}{\sqrt{\mu_{\mathbf{x}} + \varepsilon}}\right) \cdot \frac{g_2}{\sqrt{\mu_{\mathbf{y}} + \varepsilon}} \tag{8}$$

$$= \mathbf{x} \cdot \frac{g_1 g_2}{\sqrt{(\mu_{\mathbf{x}} + \varepsilon)(\mu_{\mathbf{y}} + \varepsilon)}}. \tag{9}$$

We simplify the denominator:

$$\sqrt{(\mu_{\mathbf{x}} + \varepsilon)(\mu_{\mathbf{y}} + \varepsilon)} = \sqrt{(\mu_{\mathbf{x}} + \varepsilon) \cdot \frac{g_1^2 \mu_{\mathbf{x}}}{\mu_{\mathbf{x}} + \varepsilon} + \varepsilon} \tag{10}$$

$$= \sqrt{(\mu_{\mathbf{x}} + \varepsilon) \cdot \frac{\mu_{\mathbf{x}}(g_1^2 + \varepsilon) + \varepsilon^2}{\mu_{\mathbf{x}} + \varepsilon}} \tag{11}$$

$$= \sqrt{\mu_{\mathbf{x}}(g_1^2 + \varepsilon) + \varepsilon^2}. \tag{12}$$

We then derive the final expression for $\mathbf{z}$:

$$\mathbf{z} = \mathbf{x} \cdot \frac{g_1 g_2}{\sqrt{\mu_{\mathbf{x}}(g_1^2 + \varepsilon) + \varepsilon^2}}. \tag{13}$$

This provides the combined RMSNorm operation with different scaling parameters g_1 and g_2. By combining two RMSNorm operations with different scaling parameters, we arrive at a single normalization step:

$$\mathbf{z} = \mathbf{x} \cdot \frac{g_{\text{combined}}}{\sqrt{\mu_{\mathbf{x}} + \varepsilon_{\text{combined}}}}, \tag{14}$$

where:

$$g_{\text{combined}} = \frac{g_1 g_2}{\sqrt{g_1^2 + \varepsilon}}, \quad \varepsilon_{\text{combined}} = \frac{\varepsilon^2}{g_1^2 + \varepsilon}. \tag{15}$$

Alternatively, since the denominator depends on $\mu_{\mathbf{x}}$, it may not be possible to express $\varepsilon_{\text{combined}}$ independently without further approximations.

A.4 DETAILED HARDWARE RESULTS

A.4.1 DETAILED LOIHI 2 RESULTS

Single chip experiments As described in Section 4, the energy and throughput metrics for Loihi 2 were estimated based on a preliminary implementation. We first implemented a single MatMul-free LM block on a Oheo Gulch single-chip Loihi 2 system, see Table 3 (*1-chip*). We measured

the average time per step (TPS, T_{TPS}), *i.e.*, the time that a single execution time step takes. Given the number of time steps per block, $N_{\text{steps/block}}$, we can compute the total latency of the model, or the *time-to-first-token* T_{ttft}, as $T_{\text{ttft}} = N_{\text{blocks}} \times N_{\text{steps/block}} \times T_{\text{TPS}}$ where $N_{\text{blocks}} = 24$ for the 370M MatMul-free language model.

In prefill we use pipelined mode where the TPS is constant over time because equal time bins are enforced (see Appendix A.1.1 for an explanation). We calculate the prefill throughput as $f_{\text{prefill}} = T_{\text{TPS}}^{-1}$ because a new token is processed in the interval T_{TPS}. We also measure the power of the single-chip system as the sum of a static power and dynamic power component: $P^{\text{1-chip}} = \tilde{P}^{\text{1-chip}} + \bar{P}^{\text{1-chip}}$ where $\bar{P}$ denotes static power and $\tilde{P}$ denotes dynamic power. We estimate the total prefill power as $\hat{P}_{\text{prefill}} = 24 \times P^{\text{1-chip}}$. We finally estimate the energy per token as $\hat{E}_{\text{prefill}} = \hat{P}_{\text{prefill}} * T_{\text{TPS}}$. Figure 4 shows the dynamic and static power of the single-chip experiment.

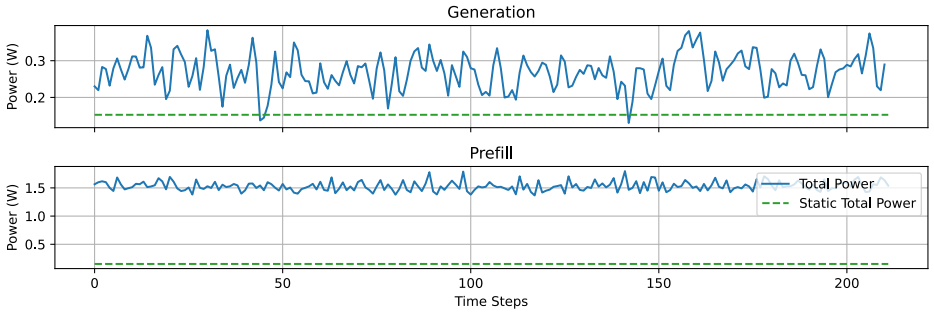

Figure 4: Power of one MatMul-free block on a single-chip Loihi 2 system.

In generation we use fallthrough mode where the TPS varies over time and directly reflects the latency of the operation that is done at the current time step (see Appendix A.1.1 for an explanation). We average over all TPS values across at least 1000 time steps to get T_{TPS}. We then calculate the generation throughput as $f_{\text{generate}} = T_{\text{ttft}}^{-1}$. This is typically significantly lower than f_{prefill}. We measure power in the same way as in prefill mode. However, to estimate the total power for the full model with all 24 blocks, we use $\hat{P}_{\text{generate}} = \tilde{P}^{\text{1-chip}} + 24 \times \bar{P}^{\text{1-chip}}$ because at any given time only a single chip will be processing information and drawing dynamic power–all other chips will be idling and drawing only static power. We estimate the energy per token as $\hat{E}_{\text{generate}} = \hat{P}_{\text{generate}} * T_{\text{ttft}}$.

Table 3: Throughput and energy efficiency estimates for our MatMul-free LM running on Intel's Loihi 2, based on a 1-chip implementation and a 24-chip implementation. Each chip implements a single block of the language model. **GEN**: autoregressive generation, **PF**: prefill mode.

		Throughput (↑ tokens/sec)	Efficiency (↓ mJ/token)
GEN	**Ours (370M)** (24-chip)	**41.5**	**405**
	Ours (370M) (1-chip)	**71.3**	**59**
PF	**Ours (370M)** (24-chip)	**6632**	**3.7**
	Ours (370M) (1-chip)	**13965**	**2.8**

* The MatMul-free LM on Loihi 2 was characterized on (1) Oheo Gulch single-chip Loihi 2 system, (2) Alia Point 32-chip Loihi 2 system (both: N3C1 silicon) running NxKernel v0.2.0 and NxCore v2.5.8 (accessible to Intel Neuromorphic Research Community members). Detailed results for single-chip and multi-chip scaling are presented in Appendix A.4.1.

‡ Transformer LMs were characterized on NVIDIA Jetson Orin Nano 8GB using the MAXN power mode running Jetpack 6.2, TensorRT 10.3.0, CUDA 12.4. Energy values include CPU_GPU_CV, SOC, and IO components as reported by jtop 4.3.0.

Performance results are based on testing as of Jan 2025 and may not reflect all publicly available security updates. Results may vary.

Multi-chip experiments Naturally, our estimates based on single-chip experiments ignore additional latency and power that comes from inter-chip communication. Therefore, we implement all 24 blocks of the MatMul-free LM on a larger multi-chip system. We use the Alia Point system (see Figure 2), where we run only 32 of all 128 chips. Each block of the MatMul-free LM is mapped to exactly one chip, thus we run the entire model on 24 out of the 32 chips on our Alia Point system. We calculate the throughput as for the single-chip experiments, but power and energy per token are

now measured directly, rather than being estimates. Table 3 shows the comparison of our single-chip estimates against the full 24-chip implementation of all MatMul-free blocks on a 32-chip Alia Point system.

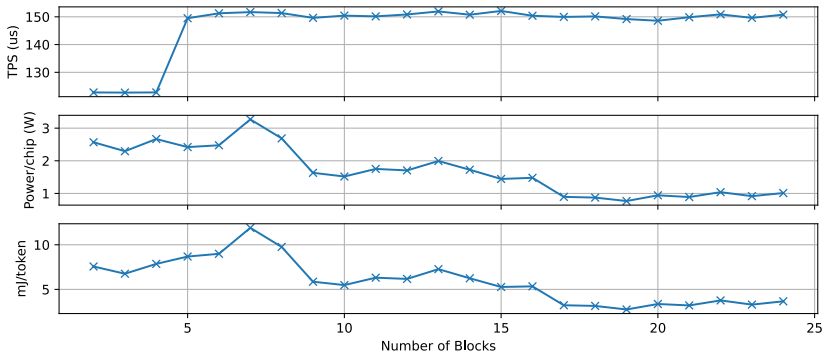

Figure 5: Scaling of time per step (inversely proportional to throughput, see text), power per chip and energy per token, as more chips are utilized in a 32-chip Alia Point Loihi 2 system. Each block of the MatMul-free LLM is implemented on a single Loihi 2 chip.

Throughput is reduced by $\approx 2.1\times$ for prefill and by $\approx 1.7\times$ for generation. Part of this slowdown comes from inter-chip communication. Figure 5 shows how the time per step (TPS) changes as more blocks are implemented on the multi-chip Loihi 2 system. The slowdown is apparent but flattens out and becomes constant for ≥ 5 chips. This is expected as only a single activation vector has to be sent from one chip to the next at every time step (or every $N_{\text{steps/block}}$ steps in generation mode).

Based on the scaling shown in Figure 5, we expect larger MatMul-free models to scale favorably on Loihi 2 systems, even as more layers are added.

Embedding and un-embedding layers In our experiments, we have not implemented the embedding and un-embedding layers on the Loihi 2 chip that map between the embedding vector space $\mathbb{R}^{1024}$ and the token vocabulary of size $V = 32,000$. We plan on implementing the embedding layer as a simple look-up table (LUT). The un-embedding layer is a ternary weight matrix of size $1024 \times V$ is the vocabulary size. In preliminary experiments, we mapped the un-embedding layer to 7 Loihi 2 chips, thus requiring a total of 31 Loihi 2 chips for the 370M MatMul-free language model and thus fitting onto a 32-chip system. As both embedding and un-embedding layers are single layers, they will not add significant latency to our model, thus we expect throughput to stay as reported in our experiments. The system will draw more power due to the additional layers, but we expect further performance optimizations to outweigh the power of two extra layers.

Limitations and further optimizations We are further optimizing latency and power of the MatMul-free LM on Loihi 2 and expect that the energy per token for the 24-chip system can approach our estimate based on a single-chip system, see Table 3.

A.4.2 Detailed NVIDIA Jetson Results

We evaluated the inference performance and energy efficiency of several state-of-the-art transformer language models running on the NVIDIA Jetson Orin Nano 8GB platform. All experiments were conducted with the MAXN power mode enabled (using Jetpack 6.2, TensorRT 10.3.0, and CUDA 12.4), and power measurements were obtained from integrated profiling tools that capture overall system consumption—including contributions from the GPU, SOC, and I/O subsystems.

Our evaluation focused on two distinct inference modes. In **prefill mode**, the model processes the input prompt in parallel. In **generate mode**, the model generates text in an autoregressive manner where tokens are produced sequentially. Due to the inherent dependencies between tokens, processing cannot be parallelized across the sequence length, resulting in lower overall throughput compared to prefill mode.

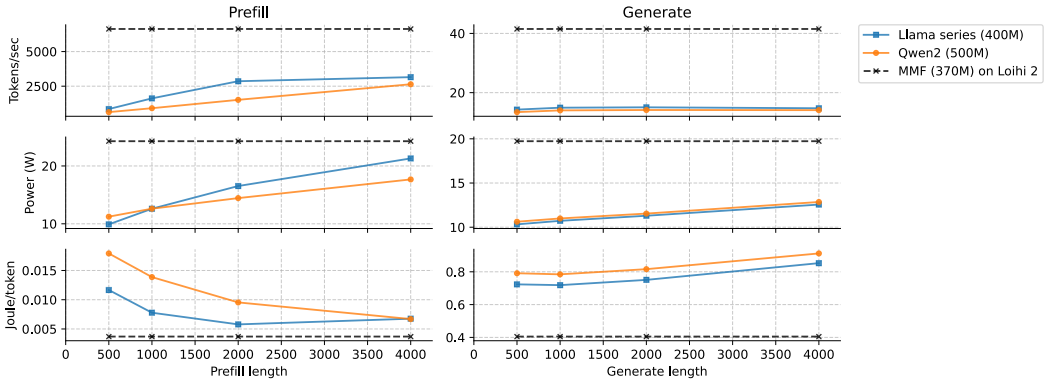

Figure 6: Hardware results for transformer-based LLMs running on the NVIDIA Jetson Orin Nano. *Left*: prefill mode where text is ingested by the LLM. *Right*: generate mode where text is generated in an auto-regressive loop. *Top*: throughput in tokens per second. *Middle*: average power in Watts. *Bottom*: energy per token. All results are averaged over time for 30 inference runs. Results for the MatMul-free LLM running on Loihi 2 are based on estimates, as explained in Appendix A.4.1.

For both modes, we performed a series of inference runs (with each set averaged over 30 iterations) to ensure stable statistical estimates. The key performance metrics–throughput in tokens per second, average power consumption (in Watts), and energy per token (in Joules)–were computed by combining the timing and power measurements recorded during these runs using the `jtop` utility.

In prefill mode, our measurements indicate very high throughput, reaching several thousand tokens per second. This is attributed to the parallel processing of tokens and effective pipelining of operations. In contrast, generate mode exhibits lower throughput since the sequential nature of token generation imposes latency limitations.

The dynamic power consumption is measured during active inference, reflecting contributions from all major system components. While the power draw is higher in prefill mode due to the continuous high-performance computation, generate mode incurs a more variable power profile as the system alternates between processing tasks and idling between token outputs.

Combining the timing and power data provides an estimate of the energy consumed per token. Although prefill mode can achieve high throughput, it also consumes more energy per token. In generate mode, the increased latency contributes to higher total energy per generated token, despite lower instantaneous power draw.

Collectively, these results provide a robust baseline for comparison with Loihi 2, our neuromorphic system. While transformer-based models on the Jetson Orin Nano achieve high throughput in certain configurations, they incur higher energy costs per token compared to the estimated figures for the MatMul-free LM implemented on Loihi 2. Figure 6 visually summarizes these metrics, displaying throughput, average power consumption, and energy per token for both prefill and generate modes. This assessment highlights the weakness inherent in conventional transformer architectures when compared with efficient alternatives on suitable hardware.

A.5 Model architecture on Loihi 2

The architecture of the MatMul-free LLM by Zhu et al. (2024) is shown in Figure 1. As explained in Section 3, the model was mapped to the Loihi 2 architecture and the resulting computational graph is shown in Figure 7.

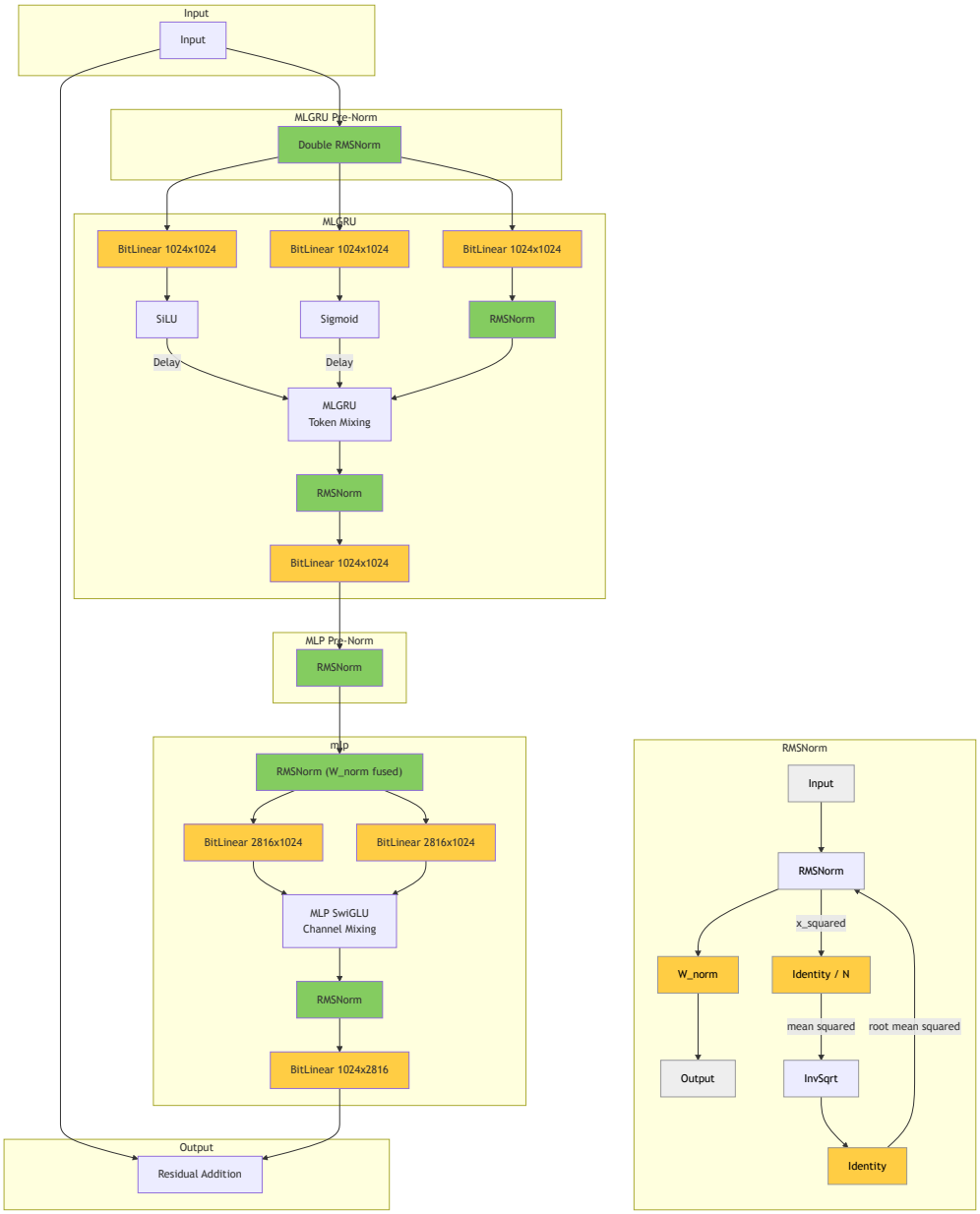

Figure 7: **Left**: Computational graph of a single MatMul-free LM layer, simplified from the actual computational graph that is mapped on the Loihi 2 chip. The RMSNorm is visualized as a single node. **Right**: Computational graph of the RMSNorm layer implemented on the Loihi 2 chip. For explanation, see main text.