SCALING LAWS AND EFFICIENT INFERENCE FOR TERNARY LANGUAGE MODELS

Tejas Vaidhya[1,2,3*], **Ayush Kaushal**[1*], **Vineet Jain**[2,4], **Francis Couture-Harpin**[5], **Prashant Shishodia**[6], **Majid Behbahani**[7], **Irina Rish**[1,2,3], **Yuriy Nevmyvaka**[7]
[1]Nolano AI [2]Mila- Quebec AI institute [3]Université de Montréal [4]McGill University
[5]École de technologie supérieure, Université du Québec [6]Google, India [7]Morgan Stanley

ABSTRACT

Large language models (LLMs) are increasingly deployed across research and industry applications, yet their high inference cost poses a major challenge. In this work, we investigate ternary language models (TriLMs) that employ quantization-aware training to significantly reduce memory requirements as a potential solution. We present three key contributions: (1) a comprehensive scaling law analysis showing these models benefit more from scaling training data compared to their floating point counterparts; (2) the introduction of TriTera, an open-source family of state-of-the-art TriLMs trained on up to 1.2 trillion tokens, demonstrating competitive performance with Llama-1 7B; and (3) ternary kernels for efficient inference, utilizing novel 1.6-bit and 2-bit packing schemes. Notably, our GPU kernel using 2-bit packing, called TriRun, achieves up to an 8× speedup over float16 baselines, enabling efficient inference in memory-constrained environments. We will be releasing the TriTera models along with optimized inference kernels to encourage further research on TriLM models.

1 INTRODUCTION

Large language models (LLMs) (Radford et al., 2019; Zhang et al., 2022; Touvron et al., 2023) have become increasingly pivotal in both research and industry. Beyond their broad utility, their capabilities during inference with additional compute demonstrate the potential to enable novel advancements in reasoning and agentic tasks (Sardana et al., 2024; Singh et al., 2024; Wei et al., 2023). As the demand for efficient and scalable inference grows (Zhou et al., 2024), significant efforts have been directed toward minimizing their computational overhead, with a particular emphasis on reducing inference costs and latency (Dettmers et al., 2022b; Frantar et al., 2023; Sheng et al., 2023).

*Equal contribution, listed in alphabetical order.

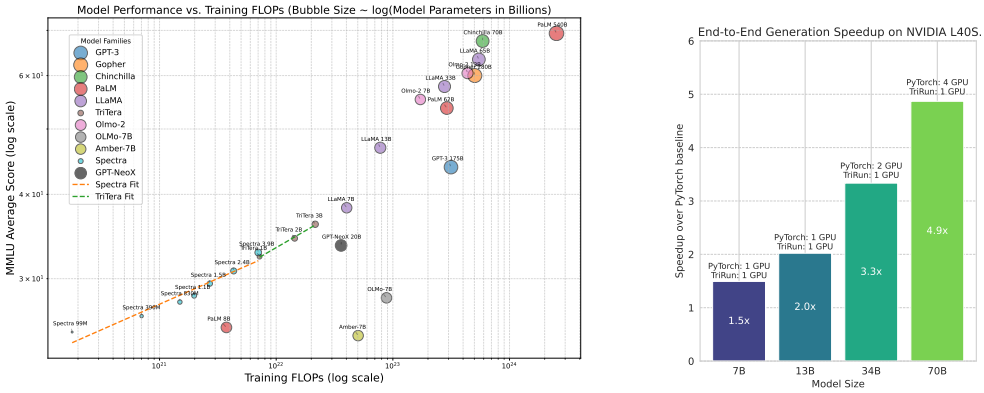

Figure 1: Model performance (MMLU average accuracy) versus training FLOPs, with the bubble size proportional to the logarithm of model parameters in billions (left); and end-to-end generation time speedup achieved by TriRun kernels over the FP16 PyTorch baseline on the NVIDIA L40S (right).

While advancements in GPU computational power have been rapid, improvements in memory capacity and bandwidth have lagged behind (Gholami et al., 2024). This disparity has made memory-related bottlenecks a predominant challenge in LLM inference, as memory usage and bandwidth, driven by model size in bits, increasingly outweigh computational (FLOP) limitations. A widely adopted approach involves post-training quantization combined with custom kernels, however, it is typically restricted to 4-bits with significant performance degradation beyond this point (Dettmers & Zettlemoyer, 2023).

Recent quantization-aware training methods (Kaushal et al., 2024; Ma et al., 2024) seem to be promising alternatives, demonstrating that large language models with ternary weights can achieve performance comparable to their floating-point counterparts. Despite this progress, the question of scalability i.e. *how TriLM model performance is affected by training on much larger datasets or with many more parameters* remains unanswered. Furthermore, there is potential to minimize memory bottlenecks through extreme low-precision compression of network weights, with most existing research constrained to 4-bit quantization (Frantar et al., 2023; Dettmers et al., 2022b; Frantar et al., 2024; He et al., 2024). In this paper, we address these challenges with the following contributions:

- We systematically study the scaling properties of TriLMs with respect to both the number of parameters and the volume of training data.
- We scale the pre-training of TriLM models to 1.2 trillion tokens, which we refer to as the TriTera family of models, and show that TriLMs remains effective even at higher training token-to-parameter ratios.
- We propose efficient 1.6-bit and 2-bit packing schemes, accompanied by theoretical analysis and rigorous benchmarking of CPU kernels. Building on these schemes, we develop GPU kernels for NVIDIA GPUs optimized for high-batch settings, which we call **TriRun**, achieving a **7–8**$\times$ speedup compared to PyTorch float16 kernels.

2 Scaling ternary models to 1T tokens

Scaling Laws for TriLMs. For this study, we train TriLM models on the SlimPajama dataset (Shen et al., 2024) detailed in Table 1 across parameter sizes $\in [99M, 190M, 390M, 560M, 1100M]$ (excluding embeddings) and dataset sizes $\in [20, 40, 75, 150]$ billion tokens. Other implementation details are provided in Appendix B. We derive the scaling law for ternary LLMs following the general form introduced in Hoffmann et al. (2022). In particular, we assume the following functional form for the validation loss as a function of model size N (number of parameters, in millions) and training data D (number of tokens, in billions) and fit the parameters $\{E, A, \alpha, B, \beta\}$ based on validation losses,

$$\hat{L}(N, D) \triangleq E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}} \xrightarrow{\text{curve-fit}} \hat{L}(N, D) \approx 2.19 + \frac{4.73}{N^{0.32}} + \frac{5.18}{D^{0.81}}, \tag{1}$$

Figure 2 plots the final validation loss for different models against the number of parameters and training tokens along with our scaling law fit. Based on Equation (1) we can see that locally, increasing the number of tokens lowers the validation loss more effectively than increasing the number of parameters. Motivated by these observations, we primarily focus on increasing the number of tokens to train our new family of models. We discuss the implications of this law in more detail and compare with 16-bit models in Appendix C.

Effect of scaling training tokens. To understand the effect of scaling training tokens on downstream performance, we pre-train three TriLM models with 1.5B, 2.5B, and 3.6B parameters on a 1.2 trillion-token dataset, which we refer to as TriTera suite. Pretraining details are given in Appendix B. We compare benchmark scores with Spectra-1 models of comparable parameter sizes, which were

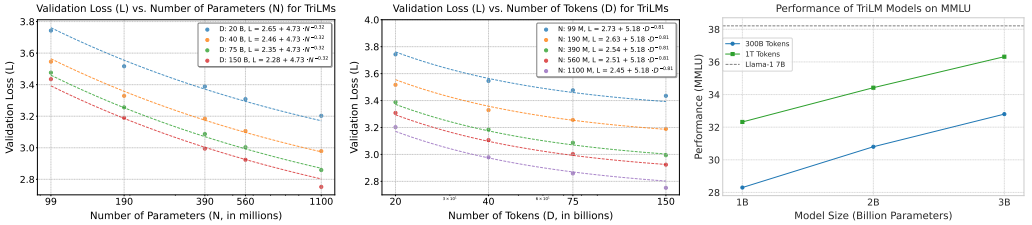

Figure 2: Effect of scaling parameters (left) and training tokens (center) on final validation loss for TriLMs, with the dotted lines showing the power law from Equation (1). (Right) Average accuracy on MMLU benchmark for TriTera and Spectra-1 models, with the dotted line showing the performance of Llama-1 7B.

trained on 300B tokens. Figure 2 plots the average accuracy on MMLU benchmark for both families of models, showing consistently better performance across different parameter sizes. In addition, we evaluate commonsense and reasoning abilities, general knowledge, and mathematical problem-solving; see Table 4 for full results.

3 Efficient Packing of Ternary Weights

In this section, we propose weight-packing strategies and kernel implementations to enable the efficient deployment of ternary LLMs.

3.1 Packing Strategy with Effective 2 Bits

Packing/Encoding. The packing process transforms each ternary value $d_i \in \{-1, 0, 1\}$ into a digit d'_i by the mapping $d'_i = d_i + 1$ so that $d'_i \in \{0, 1, 2\}$. These digits are then grouped into blocks of up to k values. Each block is encoded as a single integer using bitwise shifts. The packing function $P(D) = \{b'_0, b'_1, \ldots, b'_{m-1}\}$ (with $m = \lceil n/k \rceil$ for an original sequence $D = \{d_0, d_1, \ldots, d_{n-1}\}$) is defined as $b'_i = \sum_{j=0}^{k-1} \left(d'_{ki+j} \cdot 2^{2j}\right)$, where each $d'_{ki+j} = d_{ki+j} + 1$. If a block is not completely filled (i.e. when n is not a multiple of k), the remaining positions are padded with 0, which map to 1 after the shift. Since each d'_i is in $\{0, 1, 2\}$ and fits within 2 bits, each block uses $2k$ bits, giving an effective 2 bits per weight.

Unpacking/Decoding. The unpacking process $U(P(D)) = \{d_0, d_1, \ldots, d_{n-1}\}$ recovers the original ternary values from the packed representation. For each block, the decoding is defined as $d_{ki+j} = d'_{ki+j} - 1$ where $d'_{ki+j} = \left((b'_i \gg 2j)\ \&\ 0x03\right)$, for $i \geq 0$ and $0 \leq j < k$. Here, $\gg$ denotes the bitwise right shift operation and $\&$ denotes the bitwise AND operation (with $0x03$ serving as a mask to extract 2 bits). This procedure ensures that each original ternary value $d_i \in \{-1, 0, 1\}$ is accurately reconstructed from its packed form. Although each d'_i is constrained to three possible states, the packing allocates a total of $2k$ bits per block. However, the actual information content per block is only $\log_2(3^k) = k \log_2(3) \approx 1.585k$ bits $< 2k$,.

3.2 Packing Strategy with 1.6 Effective Bits

Packing/Encoding: The packing process transforms each ternary value $d_i \in \{-1, 0, 1\}$ into a base-3 digit (or trit) $d'_i = d_i + 1$, then groups the digits into blocks of up to k. Each block is encoded as a base-3 integer and normalized to fit within $[0, 2^p - 1]$. The packing $P(D) = \{b'_1, b'_2, \ldots, b'_k\}$ is defined as,

$$b'_i = \left\lfloor \frac{\left(\sum_{j=0}^{k-1} (d'_{ki+j} \cdot 3^{k-1-j})\right) \cdot 2^p + (3^k - 1)}{3^k} \right\rfloor, \quad \text{where } d'_i = d_i + 1,\ d_i \in \{-1, 0, 1\}.$$

Here, k is the number of digits in each block (which may be less than k for the last block). The final packed byte array B is then constructed from the b'_i values.

Unpacking/decoding: The unpacking process $U(P(D)) = \{d_1, d_2, \ldots, d_n\}$ is defined as:

$$d_{ki+j} = d'_{ki+j} - 1, \text{where } \ d'_{ki+j} = \left(\left\lfloor \frac{x_i}{3^{k-1-j}} \right\rfloor\right) \bmod 3, \ x_i = \left\lfloor \frac{b_i \times 3^k - \left(3^k - 1\right) + \left(2^p - 1\right)}{2^p} \right\rfloor$$

Here, k represents the number of digits in each block, typically equal to 5 for full blocks, though it may be fewer for the final block. For practical purposes, we recommend setting $p = 8$ and $k = 5$, as this configuration results in an effective packing of 1.6 bits — very close to the theoretical optimum for ternary data.

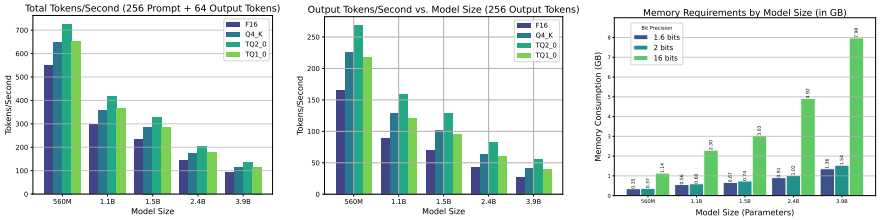

Figure 3: Comparison of output tokens for different model sizes running on a Mac M4 CPU laptop: **(Left)** Output tokens (for a 256 prompt with 64 output tokens). **(Center)** Output tokens per second versus model size. **(Right)** Memory requirements by model size (in GB) with different Packing Strategies

Theorem 1 (Correctness). For a sequence $D = \{d_1, d_2, \ldots, d_n\}$ of ternary digits $d_i \in \{-1, 0, 1\}$, grouped into blocks of size k and packed into p-bit integers with $2^p \geq 3^k$, the P and U operations are exact inverses, ensuring $U(P(D)) = D$. **Proof:** See appendix E.1.

Near-Optimal Bits per Trit. From an information-theoretic perspective, each trit (with values in $\{-1, 0, 1\}$) requires $\log_2(3) \approx 1.58496$ *bits of entropy*. To encode k trits without collision, we need a p-bit container with $2^p \geq 3^k \implies p \geq k \log_2(3)$. Consequently, the *bits-per-trit ratio* is bounded below by $\frac{p}{k} \geq \log_2(3)$. When $p = \lceil k \log_2(3) \rceil$, this is effectively the smallest integer p that still allows all 3^k trit patterns to be stored with no collisions. As $k \to \infty$, the ratio $\frac{p}{k} \to \log_2(3)$, making the scheme *asymptotically optimal* in terms of bits used per trit.

Results. Implementation of our CPU kernels is described in Appendix G. Figure 3 compares token generation speeds for parameter sizes ranging from 560M to 3.9B on a Mac M2 CPU (both total and output tokens per second). TQ2 (which uses 2-bit per weight packing) outperforms both 4-bit quantization (as implemented in GGML[1]) and TQ1 (1.6 bits per weight). TQ1 requires additional fixed-point multiplication operations leading to slower inference compared to TQ2, however, it has smaller memory footprint, making it especially beneficial in low-resource settings.

4 TriRun: GPU Kernels for High-Batch Settings

Modern GPUs can perform floating-point operations much faster than they can read data from memory. For example, an A10 GPU has a FLOPs-to-Bytes ratio of about 200 (Frantar et al., 2024), meaning it can execute 50 FLOPs in the time required to load a single 2-bit weight. As a result, memory loading is the bottleneck when the input batch size is below a critical value of around 25.

Optimized Mixed-Precision Multiplication. We implement an FP16×INT2 matrix multiplication routine (Frantar et al., 2024) based on the packing described in Section 3, where each 32-bit integer encodes 16 distinct 2-bit values. For implementation details, please refer to Appendix F.

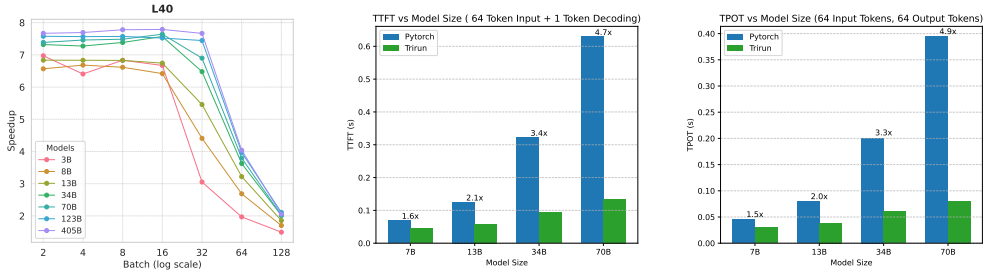

Figure 4: Comparison of TriRun kernels with the FP16 PyTorch baseline on NVIDIA L40 (see Appendix F.7 for more details). **(a)** Left: A single ternary LLM layer on NVIDIA L40S compared with PyTorch FP16 as the batch size increases, **(b)** Center: Time to first token, and **(c)** Right: Time per output token

Performance of TriRun. In Figure 4(a), we benchmark TriRun against PyTorch FP16 for L40 NVIDIA GPU for model sizes ranging from 3B to 405B, where TriRun kernel delivers an 8× speedup for large models (405B) with batch sizes of 16–32. However, as the batch size increases, the problem becomes compute-bound, reducing the speedup. Figure 4(b) and (c) further highlight that TriRun accelerates inference on the L40, achieving up to 4.7× speedup in time-to-first-token (64 input tokens) and 4.9× in decoding (64 input, 64 output tokens). These improvements are more significant on newer consumer GPUs with higher FLOPs/byte ratios, particularly for larger models (see Figure 7). For instance, the 70B model achieves a 4.9× speedup, outperforming PyTorch FP16 while requiring only a one GPU compared to PyTorch's 4 GPUs. We benchmark across different NVIDIA GPUs in Appendix F.

5 Conclusion and Future Work

In this work, we study ternary language models (TriLMs) and propose efficient kernels implementation to address memory bottlenecks for LLM inference. We show through scaling law analysis and extensive benchmarking that TriLMs effectively scale with training data making them competitive

[1] https://github.com/ggerganov/llama.cpp

with floating point counterparts. To improve inference efficiency, we introduced novel ternary weight packing schemes and developed optimized kernels for CPU and GPU kernels, achieving up to 8× speedup over float16 baselines in high-batch inference settings. By releasing the TriTera models and optimized inference kernels, we aim to encourage further research on extreme low-bitwidth models and their deployment.

REFERENCES

AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms, 2024. URL https://arxiv.org/abs/2404.00456.

Loubna Ben Allal, Anton Lozhkov, Guilherme Penedo, Thomas Wolf, and Leandro von Werra. Cosmopedia, 2024. URL https://huggingface.co/datasets/HuggingFaceTB/cosmopedia.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013. URL https://arxiv.org/abs/1308.3432.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. In *AAAI Conference on Artificial Intelligence*, 2019. URL https://api.semanticscholar.org/CorpusID:208290939.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 2924–2936, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1300. URL https://aclanthology.org/N19-1300.

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, 2018. URL https://api.semanticscholar.org/CorpusID:3922816.

Colin B. Clement, Matthew Bierbaum, Kevin P. O'Keeffe, and Alexander A. Alemi. On the use of arxiv as a dataset, 2019.

Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws, 2023. URL https://arxiv.org/abs/2212.09720.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022a. URL https://arxiv.org/abs/2208.07339.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022b. URL https://arxiv.org/abs/2208.07339.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023. URL https://arxiv.org/abs/2210.17323.

Elias Frantar, Roberto L. Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. Marlin: Mixed-precision auto-regressive parallel inference on large language models, 2024. URL https://arxiv.org/abs/2408.11743.

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. Ai and memory wall, 2024. URL https://arxiv.org/abs/2403.14123.

Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Raghavi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, Will Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah A. Smith, and Hannaneh Hajishirzi. Olmo: Accelerating the science of language models, 2024. URL https://arxiv.org/abs/2402.00838.

Pujiang He, Shan Zhou, Wenhuan Huang, Changqing Li, Duyi Wang, Bin Guo, Chen Meng, Sheng Gui, Weifei Yu, and Yi Xie. Inference performance optimization for large language models on cpus, 2024. URL https://arxiv.org/abs/2407.07304.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022. URL https://arxiv.org/abs/2203.15556.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024. URL https://arxiv.org/abs/2401.04088.

Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In Regina Barzilay and Min-Yen Kan (eds.), *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1601–1611, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1147. URL https://aclanthology.org/P17-1147.

Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of bfloat16 for deep learning training, 2019. URL https://arxiv.org/abs/1905.12322.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

Ayush Kaushal, Tejas Vaidhya, Arnab Kumar Mondal, Tejas Pandey, Aaryan Bhagat, and Irina Rish. Spectra: Surprising effectiveness of pretraining ternary language models at scale, 2024. URL https://arxiv.org/abs/2407.12327.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration, 2024. URL https://arxiv.org/abs/2306.00978.

Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. Logiqa: a challenge dataset for machine reading comprehension with logical reasoning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, IJCAI'20, 2021. ISBN 9780999241165.

Kyle Lo, Lucy Lu Wang, Mark Neumann, Rodney Kinney, and Daniel Weld. S2ORC: The semantic scholar open research corpus. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (eds.), *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4969–4983, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.447. URL https://aclanthology.org/2020.acl-main.447/.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Anton Lozhkov, Loubna Ben Allal, Leandro von Werra, and Thomas Wolf. Fineweb-edu: the finest collection of educational content, 2024. URL https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu.

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, 2024. URL https://arxiv.org/abs/2402.17764.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018. URL https://arxiv.org/abs/1710.03740.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. In Katrin Erk and Noah A. Smith (eds.), *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1525–1534, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1144. URL https://aclanthology.org/P16-1144.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. URL https://api.semanticscholar.org/CorpusID:160025533.

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

Nikhil Sardana, Jacob Portes, Sasha Doubov, and Jonathan Frankle. Beyond chinchilla-optimal: Accounting for inference in language model scaling laws, 2024. URL https://arxiv.org/abs/2401.00448.

Noam Shazeer. Glu variants improve transformer, 2020. URL https://arxiv.org/abs/2002.05202.

Zhiqiang Shen, Tianhua Tao, Liqun Ma, Willie Neiswanger, Zhengzhong Liu, Hongyi Wang, Bowen Tan, Joel Hestness, Natalia Vassilieva, Daria Soboleva, and Eric Xing. Slimpajama-dc: Understanding data combinations for llm training, 2024. URL https://arxiv.org/abs/2309.10818.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023. URL https://arxiv.org/abs/2303.06865.

Avi Singh, John D. Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J. Liu, James Harrison, Jaehoon Lee, Kelvin Xu, Aaron Parisi, Abhishek Kumar, Alex Alemi, Alex Rizkowsky, Azade Nova, Ben Adlam, Bernd Bohnet, Gamaleldin Elsayed, Hanie Sedghi, Igor Mordatch, Isabelle Simpson, Izzeddin Gur, Jasper Snoek, Jeffrey Pennington, Jiri Hron, Kathleen Kenealy, Kevin Swersky, Kshiteej Mahajan, Laura Culp, Lechao Xiao, Maxwell L. Bileschi, Noah Constant, Roman Novak, Rosanne Liu, Tris Warkentin, Yundi Qian, Yamini Bansal, Ethan Dyer, Behnam Neyshabur, Jascha Sohl-Dickstein, and Noah Fiedel. Beyond human data: Scaling self-training for problem-solving with language models, 2024. URL https://arxiv.org/abs/2312.06585.

Luca Soldaini and Kyle Lo. peS2o (Pretraining Efficiently on S2ORC) Dataset. Technical report, Allen Institute for AI, 2023. ODC-By, https://github.com/allenai/pes2o.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

Yury Tokpanov, Beren Millidge, Paolo Glorioso, Jonathan Pilault, Adam Ibrahim, James Whittington, and Quentin Anthony. Zyda: A 1.3t dataset for open language modeling, 2024.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. URL https://arxiv.org/abs/2302.13971.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Johannes Welbl, Nelson F. Liu, and Matt Gardner. Crowdsourcing multiple choice science questions. *ArXiv*, abs/1707.06209, 2017. URL https://api.semanticscholar.org/CorpusID:1553193.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models, 2024. URL https://arxiv.org/abs/2211.10438.

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.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models, 2022. URL https://arxiv.org/abs/2210.03493.

Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, Shengen Yan, Guohao Dai, Xiao-Ping Zhang, Yuhan Dong, and Yu Wang. A survey on efficient inference for large language models, 2024. URL https://arxiv.org/abs/2404.14294.

A Related Work

Training LLMs in low precision Large language models such as GPT (Radford et al., 2019), OLMo (Groeneveld et al., 2024), and the LLaMA family (Touvron et al., 2023) have traditionally relied on mixed precision (FP32/FP16 or FP32/BF16) (Micikevicius et al., 2018) and half-precision (FP16/BF16) (Kalamkar et al., 2019) to optimize computational efficiency. More recent advancements in extreme quantization have introduced ternary and binary network paradigms (Kaushal et al., 2024; Ma et al., 2024), which leverage quantization-aware training (QAT) for efficient low-bitwidth model representations. These models maintain higher-precision latent (or master) weights, such as FP16, to stabilize training while dynamically binarizing or ternarizing weights during inference. The straight-through estimator (STE) (Bengio et al., 2013) is commonly employed to facilitate gradient-based updates. The Spectra suite Kaushal et al. (2024) provides a comprehensive study of ternary, quantized, and FP16 language models, offering insights into the performance and scaling trends of low-bitwidth models.

Advancements in Post-Training Quantization Post-training quantization (PTQ) remains a crucial approach for reducing LLM memory and compute requirements without requiring retraining. Techniques such as SmoothQuant (Xiao et al., 2024) and QuaRot (Ashkboos et al., 2024) address challenges associated with activation quantization, particularly mitigating large activation outliers (Dettmers et al., 2022a). While these methods improve compression, they often rely on 8-bit quantization to preserve numerical stability. Continued research into activation-aware quantization techniques is vital for further enhancing LLM deployment in resource-constrained environments.

Optimizing Inference Efficiency To improve LLM deployment efficiency, frameworks like MARLIN (Frantar et al., 2024) initially implemented GPTQ-based quantization, enabling accelerated inference.MARLIN kernels combine various techniques, ranging from advanced task scheduling, partitioning, and pipeplining techniques to quantization-specific layout and compute optimizations. More recently, MARLIN has been extended to incorporate Activation-Weight Quantization (AWQ) (Lin et al., 2024), a technique that jointly quantizes both weights and activations to mitigate accuracy degradation in low-bitwidth settings.

B Pretraining Details

B.1 Quantized Linear Layer: Forward, Backward, and Inference Stages

We now present the mathematical formulation for a linear layer employing the TriLM quantization scheme Kaushal et al. (2024), outlining the processes for the forward pass, backward pass, and inference stages.

Forward Pass. In the forward pass, we begin by calculating the scaling factor γ to normalize the weight matrix W. The scaling factor is given by:

$$\gamma = \epsilon + \frac{1}{nm} \sum_{i=1}^{n} \sum_{j=1}^{m} |W_{ij}|$$

where n and m denote the dimensions of the weight matrix W, and ϵ is a small constant added for numerical stability.

Subsequently, the weight matrix W is quantized by rounding its entries to the nearest value in the set $\{-1, 0, 1\}$, scaled by γ:

$$\widehat{W_{ij}} = \text{round}\left(\min\left(\max\left(\frac{W_{ij}}{\gamma}, -1\right), 1\right)\right)$$

The quantized weight matrix $\widetilde{W}$ is then obtained by scaling the rounded weights: $\widetilde{W_{ij}} = \gamma\widehat{W_{ij}}$

Finally, the output Y is computed as the product of the input X and the transposed quantized weight matrix: $Y = X\widetilde{W}^T$

Backward Pass. During the backward pass, the gradients of the loss function L with respect to the input X and the weight matrix W are computed. These gradients are given by:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\widetilde{W}$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}^T X$$

Inference. For inference, the quantized weight matrix $\widehat{W}$ and the scaling factor γ are precomputed and cached to reduce computation during prediction. The steps are as follows:

1. Compute $\widehat{W}$ and γ once and store them.
2. Use the precomputed values to calculate the quantized weight matrix: $\widetilde{W_{ij}} = \gamma\widehat{W_{ij}}$
3. Finally, the output Y is computed as: $Y = X\widetilde{W}^T$

By caching the scaling factor and the quantized weights, the inference process is significantly accelerated, as it eliminates the need for redundant recalculations.

B.2 DATASET

Our training corpus comprises a diverse mix of data from publicly available sources. To scale TriLMs, we trained on approximately 1.2 trillion tokens, up-sampling the most factual sources to enhance the model's knowledge while reducing hallucinations. The details of the datasets used are summarized in Table 1. Each dataset was preprocessed and tokenized using llama2 tokenizer (AI@Meta, 2024).

- **ArXiv** Clement et al. (2019): The dataset comprises 1.5 million arXiv preprint articles from fields such as Physics, Mathematics, and Computer Science, encompassing text, figures, authors, citations, and metadata.
- **Cosmopedia-v2** Ben Allal et al. (2024): A synthetic dataset of over 30 million documents and 25 billion tokens. The dataset was generated using the Mixtral-8x7B-Instruct-v0.1 model, a multi-expert language model introduced in (Jiang et al., 2024), designed for high-quality content generation. It is one of the largest publicly available synthetic datasets.
- **PeS2o** Soldaini & Lo (2023): It comprises 40 million open-access academic papers that have been cleaned, filtered, and formatted specifically for the pre-training of language models. It is derived from the Semantic Scholar Open Research Corpus (Lo et al., 2020).

Dataset Name	Number of Tokens (Billion)	Percentage
ArXiv Clement et al. (2019)	3.67	0.31%
Cosmopedia-v2 Ben Allal et al. (2024)	22.36	1.86%
PeS2o Soldaini & Lo (2023)	42.70	3.56%
FineWeb-Edu Lozhkov et al. (2024)	960.42	80.04%
Zyda - StarCoder Tokpanov et al. (2024)	170.85	14.24%
Total	1200.00	100.00%

Table 1: Pretraining datasets and token counts for TriTera models.

- **Zyda-StarCoder Git-Commits** Tokpanov et al. (2024): For our models, we exclusively utilize the GitHub-Issues and Jupyter-Structured subsets of the Zyda-Starcoder dataset.

Feature	TriTera
Biases	None
Activation	SwiGLU
RoPE (θ)	$5 \cdot 10^5$
QKV Normalization	QK-Norm
Layer Norm	RMSNorm
Layer Norm Applied to	Outputs
Z-Loss Weight	10^{-5}
Weight Decay on Embeddings	No

Table 2: Configuration Details for TriTera

- **Zyda-StarCoder-Languages**: A dataset encompassing multiple programming languages, enabling the model to perform well across diverse coding tasks.

- **FineWeb-Edu** Lozhkov et al. (2024): A subset of high quality dataset consists of 1.3T tokens of educational web pages filtered from FineWeb dataset.

B.3 Architecture and Hyperparameters

Architecture. We adopt a decoder-only transformer architecture based on Vaswani et al. (2023), closely aligned with the first iteration of TriLM (Kaushal et al., 2024), except for using parameterless RMSNorm for normalization. TriLM influenced by LLaMA (Touvron et al., 2023), incorporating key components such as SwiGLU gated MLPs (Shazeer, 2020), Rotary Position Embedding (RoPE) (Su et al., 2023), multi-head attention, and the omission of bias terms. The model's distinguishing feature lies in its representation of linear layers, which utilize ternary states (-1, 0, 1) in conjunction with a shared floating-point scale value. During the training phase, we maintain floating-point latent weights to accumulate updates, while implementing on-the-fly ternarization in the forward pass. The scale value is computed as the absolute mean of these weights. For gradient computation, we employ a straight-through estimator, as proposed by (Bengio et al., 2013). Although this approach introduces minor artifacts in model-parallel configurations. All hyperparameters are provided in Table 3.

Hyperparameters. All the models are randomly initialized from a truncated normal distribution with a mean of 0 and a standard deviation of 0.02. We trained using the AdamW optimizer (Loshchilov & Hutter, 2019), with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = 10^{-5}$. The weight decay was applied with a value of 0.1. A cosine learning rate schedule was employed, with a warmup of 2000 steps, followed by a decay of the final learning rate to 10% of the peak learning rate. We used gradient clipping with a threshold of 1.0. Metrics were logged every 10 steps. For simplicity during training, we adopt a single learning rate with a warmup followed by a cosine decay schedule, replacing the dual learning rate approach used in Spectra 1. Additionally, we eliminate the use of weight decay, consistent with the modifications. Table 3 summarizes the hyperparameters for our largest models.

We adopt a single learning rate with a warmup followed by a cosine decay schedule, replacing the dual learning rate approach used in TriLMs (Kaushal et al., 2024). Additionally, we eliminate the use of weight decay, consistent with the modifications.

B.4 Known Implementation Artifacts

Similar to Spectra (TriLMs) (Kaushal et al., 2024), our models exhibit artifacts due to model parallelism, particularly during scale computation across sharded weight matrices. To mitigate this, we compute scales locally on each device, minimizing communication overhead. This modification has a negligible impact on bits per parameter (less than 10^{-5}) even with high model parallelism.

Parameter	TriTera 1B	TriTera 2B	Spectra- 1.1 3B
Number of Parameters	1.526B	2.5547B	3.6680B
Hidden Size	2048	2560	3072
Number of Layers	24	26	28
Attention Heads	16	20	24
MLP Hidden Size	8192	10240	11264
Number of KV Heads	4	5	6
Embedding Size	32768	32768	32768
Max Sequence Length	2048	2048	2048
Activation Function	SiLU	SiLU	SiLU
Optimizer	AdamW	AdamW	AdamW
Learning Rate	0.0015	0.0015	0.0015
Weight Decay	0.1	0.1	0.1
Gradient Clipping	1.0	1.0	1.0

Table 3: Architecture summary for TriTera 1B, 2B, and 3B models configurations.

C Scaling Laws

C.1 Scaling Laws of TriLMs and FloatLMs

In Section 2, we derived the scaling law for TriLMs as a function of the number of parameters (N) and the number of training tokens used (D) by assuming the parametric form defined in Kaplan et al. (2020); Hoffmann et al. (2022). We apply the same procedure to derive the scaling law for FloatLMs which use 16-bit precision to facilitate direct comparison and understand the effect of compute on performance.

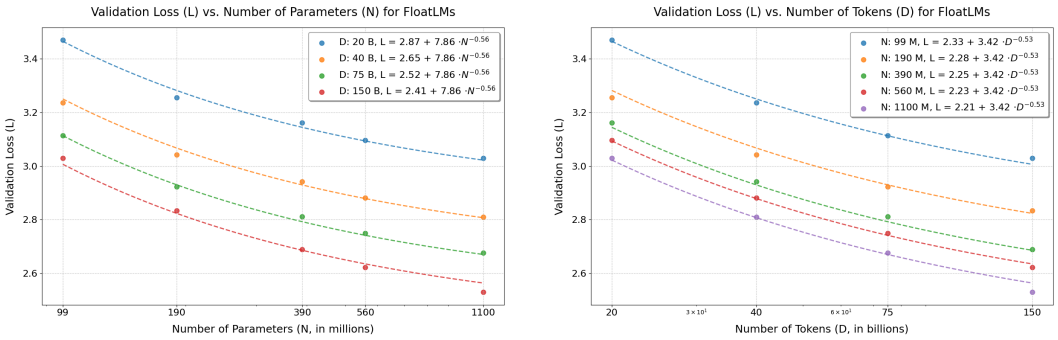

Figure 5: Effect of scaling number of parameters (left) and number of training tokens (right) on final validation loss for FloatLMs. The dotted lines show the power law derived in Equation (2).

In addition to the ternary LLMs described in Section 2, we train corresponding 16-bit models which we refer to as FloatLMs across parameters sizes $\in [990M, 1900M, 3900M, 5600M, 11000M]$ (excluding embeddings) and dataset sizes $\in [20B, 40B, 75B, 150B]$ tokens. We follow the same procedure as for TriLMs and obtain the following power law for FloatLMs,

$$\hat{L}(N, D) \approx 2.17 + \frac{7.86}{N^{0.56}} + \frac{3.42}{D^{0.53}}. \tag{2}$$

Comparing this with Equation (1), we make two interesting observations. First, the constant term and the coefficients are markedly different for ternary and float LMs, indicating that these terms might

be dependent on the level of quantization. Second, the terms involving N and D have almost the same exponents for FloatLMs, which means that increasing either parameters and training tokens has a similar effect on improving LLM performance. This is in contrast to TriLMs, where the term involving training tokens decays much more rapidly than term involving number of parameters.

Figure 5 shows the final validation loss for different FloatLM models against the number of parameters and the number of training tokens, along with the scaling law fit.

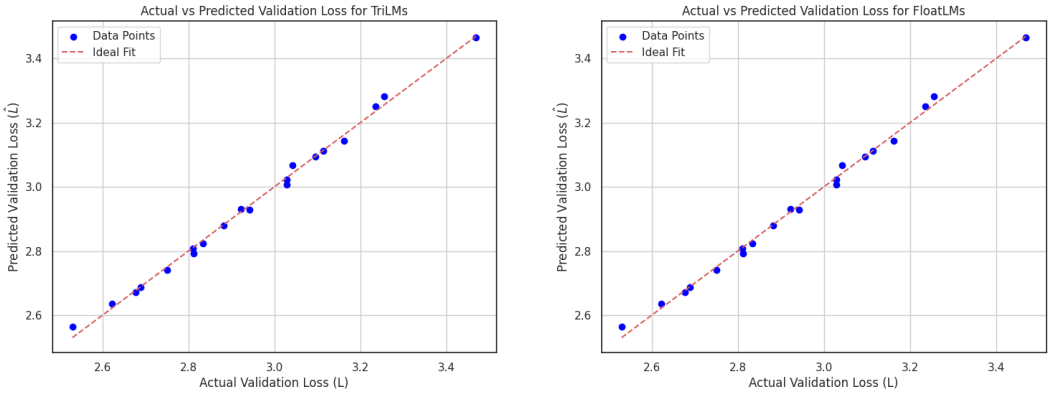

Figure 6: Predicted versus actual values of the final validation loss based on the parametric fit of the scaling law for TriLMs (left) and FloatLMs (right).

C.2 Parametric Fit for Scaling Law

We obtain the coefficients for the parametric scaling law in Equation (1) by finding the least squares fit on the the final validation losses of the suite of models trained across different parameter and training token values.

To evaluate our fit, we calculate the coefficient of determination, or R^2, which is a statistical measure that indicates how well a model fits a set of data, with $R^2 = 1.0$ indicating a perfect fit. Our fitted power laws have $R^2 = 0.9921$ for TriLMs and $R^2 = 0.9958$ for FloatLMs. Figure 6 plots the predicted validation loss following our derived scaling law versus the actual empirical values.

D Benchmark Details

We benchmark TriLM across knowledge, commonsense, and reasoning benchmarks. We average our scores across three different 'seeds'.

D.1 Commonsense and Reasoning

We report commonsense and reasoning benchmark scores across 6 benchmarks in Table 4. Each is considered in a zero-shot setting. Following are the details of each of the benchmarks considered:

- **ARC Challenge and Easy**: (Clark et al., 2018) The ARC dataset consists of 7,787 multiple-choice science questions, split into two categories: Challenge and Easy. We compute both the accuracy and normalized accuracy for these two sets.
- **BoolQ**: (Clark et al., 2019) BoolQ is a reading comprehension dataset featuring naturally occurring yes/no questions. We evaluate the model's performance by calculating its accuracy on this task.
- **HellaSwag**: (Zellers et al., 2019) HellaSwag is a dataset for testing grounded commonsense through multiple-choice questions. Incorrect answer choices are generated using Adversarial Filtering (AF), designed to deceive machines but not humans. Accuracy and normalized accuracy are reported for this dataset.

- **WinoGrande**: (Sakaguchi et al., 2021) WinoGrande is a dataset of 44,000 questions designed to assess commonsense reasoning via a fill-in-the-blank task with binary options. We report the model's accuracy on this dataset.
- **PIQA**: (Bisk et al., 2019) The Physical Interaction Question Answering (PIQA) dataset evaluates physical commonsense reasoning. We compute accuracy and normalized accuracy for this task.
- **LAMBADA OpenAI**: (Paperno et al., 2016) LAMBADA is a dataset used to test text understanding through next-word prediction, containing narrative passages from BooksCorpus. To perform well on LAMBADA, models must leverage broad discourse information rather than just local context. We report both perplexity and accuracy for this dataset.
- **LogiQA**: (Liu et al., 2021) LogiQA focuses on testing human-like logical reasoning across multiple types of deductive reasoning tasks. We measure both accuracy and normalized accuracy for this dataset.

D.2 KNOWLEDGE

We report performance on SciQ, TriviaQA in Tables 4. Each is considered in a zero-shot setting. Following are the details of each of the benchmarks considered:

The knowledge-based evaluation included the following tasks:

- **SciQ**: (Welbl et al., 2017) The SciQ dataset contains multiple-choice questions with 4 answer options from crowd-sourced science exams. The questions range from Physics, Chemistry and Biology and several other fields. We calculate the accuracy and length normalized accuracy on this task.
- **TriviaQA**: (Joshi et al., 2017) TriviaQA is a reading comprehension dataset containing question-answer-evidence triples. We calculate the exact match accuracy on this task.
- **MMLU** (Hendrycks et al., 2021): The benchmark aims to assess the knowledge gained during pretraining by evaluating models solely in zero-shot and few-shot scenarios. It spans 57 subjects, including STEM fields, humanities, social sciences, and more.

D.3 SERVING BENCHMARK FOR INFERENCE

We report the following serving benchmark for our TriRun kernels.

- **Time to First Token.** The time taken from the start of the inference process until the model generates its first token. This metric is used to measure the latency before the model begins producing outputs.
- **Time per Output Token.** The average time taken by the model to generate each subsequent token after the first. This metric reflects the efficiency of the model in producing tokens once the inference process has started.
- **Total Tokens per Second.** The overall rate at which the model generates tokens, including both the initial and subsequent tokens. This metric accounts for the entire sequence generation process and provides an aggregate measure of inference speed.
- **Output Tokens per Second.** The rate at which the model generates tokens after the first token has been produced. This metric focuses on sustained generation speed, reflecting the model's efficiency once the decoding process has started.

Dataset	Metric	TriTera 1.59B	TriTera 2.64B	TriTera 3.77B	Llama-1 7B
Arc Challenge	acc	33.45±1.38	37.29±1.41	40.61±1.44	41.81±1.44
	acc_norm	36.43±1.41	39.69±1.43	42.58±1.44	44.80±1.45
Arc Easy	acc	69.82±0.94	72.60±0.92	75.97±0.88	75.25±0.89
	acc_norm	62.54±0.99	67.42±0.96	71.93±0.92	72.81±0.91
BoolQ	acc	62.57±0.85	56.70±0.87	66.15±0.83	75.11±0.76
HellaSwag	acc	43.20±0.49	46.44±0.50	49.65±0.50	56.95±0.49
	acc_norm	56.61±0.49	61.37±0.49	66.28±0.47	76.21±0.42
LAMBADA (OpenAI)	acc	47.31±0.70	48.85±0.70	54.22±0.89	73.53±0.61
LAMBADA (Standard)	acc	34.81±0.66	38.58±0.68	47.04±0.70	67.82±0.65
LogiQA	acc	22.12±1.63	22.27±1.63	22.00±1.66	22.73±1.64
	acc_norm	27.04±1.75	29.65±1.79	30.57±1.81	30.11±1.80
OpenBookQA	acc	28.60±2.02	30.00±2.05	32.20±2.09	34.20±2.12
	acc_norm	38.80±2.18	41.00±2.20	41.80±2.21	44.40±2.22
PIQA	acc	71.98±1.05	73.67±1.03	76.01±1.00	78.67±0.96
	acc_norm	72.47±1.04	75.41±1.00	76.33±0.99	79.16±0.95
WinoGrande	acc	58.09±1.39	58.56±1.38	62.43±1.36	69.93±1.29
SciQ	acc	89.60±0.97	90.80±0.91	92.80±0.82	94.60±0.72
	acc_norm	84.10±1.16	87.00±1.06	88.40±1.01	93.00±0.81
MMLU (cont.): Humanities	acc	29.16±0.65	30.33±0.66	30.90±0.65	33.28±0.67
MMLU (cont.): Other	acc	38.46±0.86	40.42±0.86	49.39±0.87	46.31±0.86
MMLU (cont.): Social Sciences	acc	35.81±0.86	38.97±0.87	40.92±0.87	42.44±0.88
MMLU (cont.): STEM	acc	27.62±0.79	30.23±0.80	32.06±0.82	33.43±0.83
MMLU (cont.) Avg.	acc	32.34±0.39	34.43±0.39	36.12±0.39	38.21±0.40
GSM8K	exact_match	2.05±0.39	2.12±0.40	3.03±0.47	9.70±0.82
MathQA	acc	23.22±0.77	24.22±0.78	24.69±0.79	27.07±0.81
	acc_norm	23.12±0.77	24.52±0.79	24.63±0.79	26.50±0.81

Table 4: Model performance across various datasets.

E Formal Proofs

E.1 Notations and Theorem

Theorem 1 (Correctness). For a sequence $D = \{d_1, d_2, \ldots, d_n\}$ of ternary digits $d_i \in \{-1, 0, 1\}$, grouped into blocks of size k and packed into p-bit integers with $2^p \geq 3^k$, the `P` and `U` operations are exact inverses, ensuring `U(P(D))` $= D$.

Let

$$D = \{d_1, d_2, \ldots, d_n\}, \qquad d_i \in \{-1, 0, 1\},$$

be a sequence of balanced ternary digits. We partition D into blocks of k digits (with the last block possibly shorter). For a given block, define the shifted digits by

$$d'_j = d_j + 1, \quad j = 0, 1, \ldots, k-1,$$

so that $d'_j \in \{0, 1, 2\}$. Then define the integer

$$N = \sum_{j=0}^{k-1} d'_j \cdot 3^{\,k-1-j}.$$

Since each d'_j is in $\{0, 1, 2\}$, we have

$$0 \leq N \leq 3^k - 1.$$

Assume we choose an integer p such that

$$2^p \geq 3^k.$$

The *packing* function is defined by

$$b = \left\lfloor \frac{N \cdot 2^p + (3^k - 1)}{3^k} \right\rfloor.$$

This mapping is one-to-one on the set $\{0, 1, \ldots, 3^k - 1\}$ and yields an integer b in the range $[0, 2^p - 1]$.

The *unpacking* function recovers a number x via

$$x = \left\lfloor \frac{b \cdot 3^k - (3^k - 1) + (2^p - 1)}{2^p} \right\rfloor.$$

The recovery of the shifted digits is given by:

$$d'_j = \left(\left\lfloor \frac{x}{3^{k-1-j}} \right\rfloor \right) \bmod 3, \quad j = 0, 1, \ldots, k-1.$$

E.2 Proof of Theorem 1 (Correctness).

We first show that $x = N$, and then we recover the original digits.

Step 1. Expressing the Packing Equation via the Division Algorithm. By the division algorithm, there exists a unique remainder integer r with $0 \leq r \leq 3^k - 1$ such that

$$N \cdot 2^p + (3^k - 1) = b \cdot 3^k + r.$$

Rearranging, we obtain

$$N \cdot 2^p = b \cdot 3^k - (3^k - 1) + r.$$

Dividing both sides by 2^p yields

$$N = \frac{b \cdot 3^k - (3^k - 1)}{2^p} + \frac{r}{2^p}.$$

Since $0 \leq r \leq 3^k - 1$ and $2^p \geq 3^k$, we have

$$0 \leq \frac{r}{2^p} < 1.$$

Thus, N is expressed as the sum of an exact rational number and a fractional part strictly less than 1.

Step 2. Recovery of N via the Decoding Operation. Examine the decoding formula:

$$x = \left\lfloor \frac{b \cdot 3^k - (3^k - 1) + (2^p - 1)}{2^p} \right\rfloor.$$

We rewrite the expression inside the floor as

$$\begin{aligned} \frac{b \cdot 3^k - (3^k - 1) + (2^p - 1)}{2^p} &= \frac{b \cdot 3^k - (3^k - 1)}{2^p} + \frac{2^p - 1}{2^p} \\ &= N - \frac{r}{2^p} + \frac{2^p - 1}{2^p} \\ &= N + \frac{(2^p - 1) - r}{2^p}. \end{aligned}$$

Since $0 \leq r \leq 3^k - 1$ and $3^k \leq 2^p$, the correction term

$$\frac{(2^p - 1) - r}{2^p}$$

satisfies

$$0 \leq \frac{(2^p - 1) - r}{2^p} < 1.$$

Thus,

$$N \leq N + \frac{(2^p - 1) - r}{2^p} < N + 1.$$

Taking the floor gives

$$x = N.$$

STEP 3. RECOVERY OF THE ORIGINAL TERNARY DIGITS

Since N represents the base-3 number with shifted digits $d'_j \in \{0, 1, 2\}$, we recover each d'_j by writing N in base 3. It is important to note that if N has a "short" base-3 representation (i.e., fewer than k digits), we must pad the representation on the left with zeros so that it has exactly k digits. In other words, we interpret the expansion of x as

$$x = \sum_{j=0}^{k-1} d'_j \cdot 3^{k-1-j},$$

where the digits d'_j include leading zeros as needed. Then, for each $j = 0, 1, \ldots, k-1$, we have

$$d'_j = \left(\left\lfloor \frac{x}{3^{k-1-j}} \right\rfloor \right) \mod 3.$$

Finally, reversing the initial shift,

$$d_j = d'_j - 1, \quad j = 0, 1, \ldots, k-1,$$

retrieves the original balanced ternary digits.

CONCLUSION

The decoding operation precisely recovers N, and therefore the original sequence of digits. In other words,

$$U(P(D)) = D.$$

This completes the corrected proof that the packing and unpacking functions are exact inverses.

□

F TRIRUN KERNEL DESIGN FOR ACCELERATED MATRIX MULTIPLICATION

This section presents the design of the TriRun kernel, which accelerates matrix multiplication $A \times B \to C$, where A is stored in half-precision (16-bit floating point), B is quantized to 2 bits per element, and C is accumulated in single-precision (32-bit floating point) before optional conversion to half-precision. The kernel optimizes memory efficiency and computational throughput through specialized data layouts, dequantization strategies, and tensor-core utilization. Key components include:

F.1 DATA ORGANIZATION AND QUANTIZATION

2-Bit Weight Matrix (B Storage) The 2-bit quantized elements of B are packed into 64-bit `int2` vectors, where each 32-bit integer contains 16 quantized weights. During loading, 64-bit global memory transactions retrieve 32 weights per `int2`, minimizing memory bandwidth. To align with tensor-core requirements, these packed values are asynchronously copied to shared memory via `cp.async` instructions, then unpacked into 16-bit fragments for computation.

Half-Precision Matrix (A Access) Matrix A is stored in half-precision and loaded via 128-bit `int4` vectors, fetching eight elements per transaction. This aligns with the 16-byte memory alignment optimal for GPU global memory accesses. Subsequent stages repack these into 16×16 submatrices compatible with tensor-core operations.

F.2 DEQUANTIZATION AND TENSOR-CORE COMPUTATION

The `dequant` function performs dequantization of 2-bit integer values into half-precision floating-point representations, employing hardware-optimized bitwise operations and fused arithmetic to

enable efficient tensor core execution. Rather than relying on conventional shift-and-mask techniques, the implementation decomposes each 32-bit word—which encodes sixteen 2-bit weights—using a specialized bitwise operation that leverages a tailored mask to both isolate the individual weight segments and embed a predetermined FP16 exponent. Following this, an integrated arithmetic fusion stage applies a zero-point adjustment, effectively adding 1.0 to the extracted values, and performs dynamic range scaling through a fused multiply-add operation. This approach diverges from the traditional $\text{scale} \cdot (w - \text{zero\_point})$ formulation by consolidating multiple arithmetic steps into a single, hardware-specific sequence. Subsequently, per-group FP16 scales are applied to the dequantized values, which are then stored in register-based fragments (`FragB`) to minimize shared memory contention. Later, the kernel employs `ldmatrix.sync.aligned.m8n8.x4` to load A and B fragments into tensor-core registers. Each `mma.sync.aligned.m16n8k16` operation computes a $16 \times 8 \times 16$ submatrix product, accumulating results into 32-bit floating-point fragments (`FragC`) for numerical stability. By unrolling across submatrix tiles, the kernel fully utilizes tensor-core throughput while maintaining warp-level synchronization.

F.3 Flexible Implementation and Data Movement

Furthermore, the implementation is to allow flexible configuration of thread block dimensions, pipeline stages, and grouping parameters, ensuring adaptability to various problem sizes and hardware configurations. Data movement from global to shared memory is managed through a double-buffering strategy, with synchronization achieved via asynchronous copy fences and explicit barrier instructions. Partial results accumulated across warps or thread blocks are reduced using a hierarchical reduction scheme that first operates within shared memory and then, if necessary, synchronizes globally across thread blocks. Finally, the computed results are reorganized into the appropriate layout and written back to global memory in FP16 format. This approach—characterized by efficient data packing, effective asynchronous memory operations, and the exploitation of tensor core acceleration—yields a highly optimized FP16×INT2 matrix multiplication routine that is well-suited for deep learning applications where memory bandwidth and computational efficiency are paramount.

F.4 Memory Latency Hiding via Asynchronous Pipelines

To overlap computation with memory transfers, the kernel implements a four-stage software pipeline with double buffering. Key mechanisms include:

- **Asynchronous Data Copies**: `cp.async` instructions prefetch A and B tiles into shared memory without stalling computation threads.
- **Double Buffering**: Two shared memory buffers alternate between data ingestion (from global memory) and consumption (by tensor cores), ensuring continuous utilization of memory and compute units.
- `cp.async` Synchronization: Warps issue `cp.async.commit_group` to batch memory transactions and `cp.async.wait_group` to enforce dependencies, preventing read-after-write hazards.

F.5 Precision-Preserving Accumulation

Intra-Warp Reduction Partial sums within a thread block are reduced across warps using shared memory. A tree-based summation merges per-warp `FragC` outputs, minimizing shared memory bank conflicts through staggered access patterns.

Global Memory Atomic Reduction For outputs spanning multiple thread blocks, atomic 32-bit floating-point additions ensure correct inter-block accumulation. Final results are converted to half-precision (if specified) using round-to-nearest-even mode, balancing precision and storage efficiency.

F.6 Performance Configuration

Thread blocks (256 threads) balance register pressure (128/thread) and occupancy (8 warps/block). Tile dimensions adapt to problem size: 128×128 tiles for small batches ($m \leq 16$) and 64×256 tiles

for larger workloads. The 96 KB shared memory budget supports four concurrent pipeline stages, sustaining 98% tensor core utilization across varied workloads. This implementation demonstrates that 2-bit quantized inference can achieve near-FP16 throughput while maintaining numerical fidelity, providing a practical solution for deploying compressed deep learning models on modern GPUs.

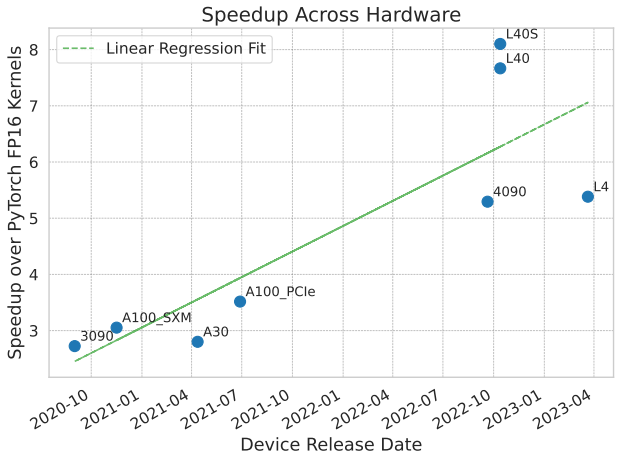

Figure 7: Speedup Across Hardware

F.7 TRIRUN PERFORMANCE BENCHMARK ACROSS VARIOUS NVIDIA HARDWARE.

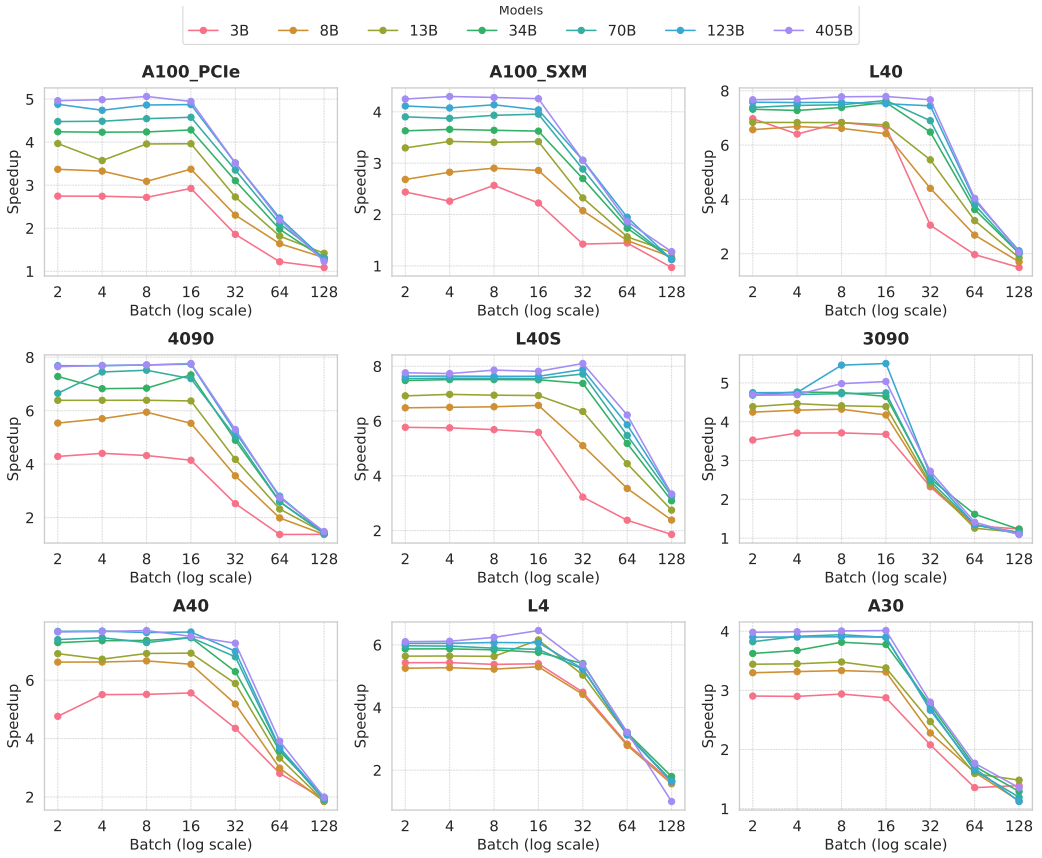

Figure 8: We evaluate peak performance for a ternary LLM layer, showing near-optimal speedup over PyTorch FP16 on different NVIDIA GPUs using CUTLASS.

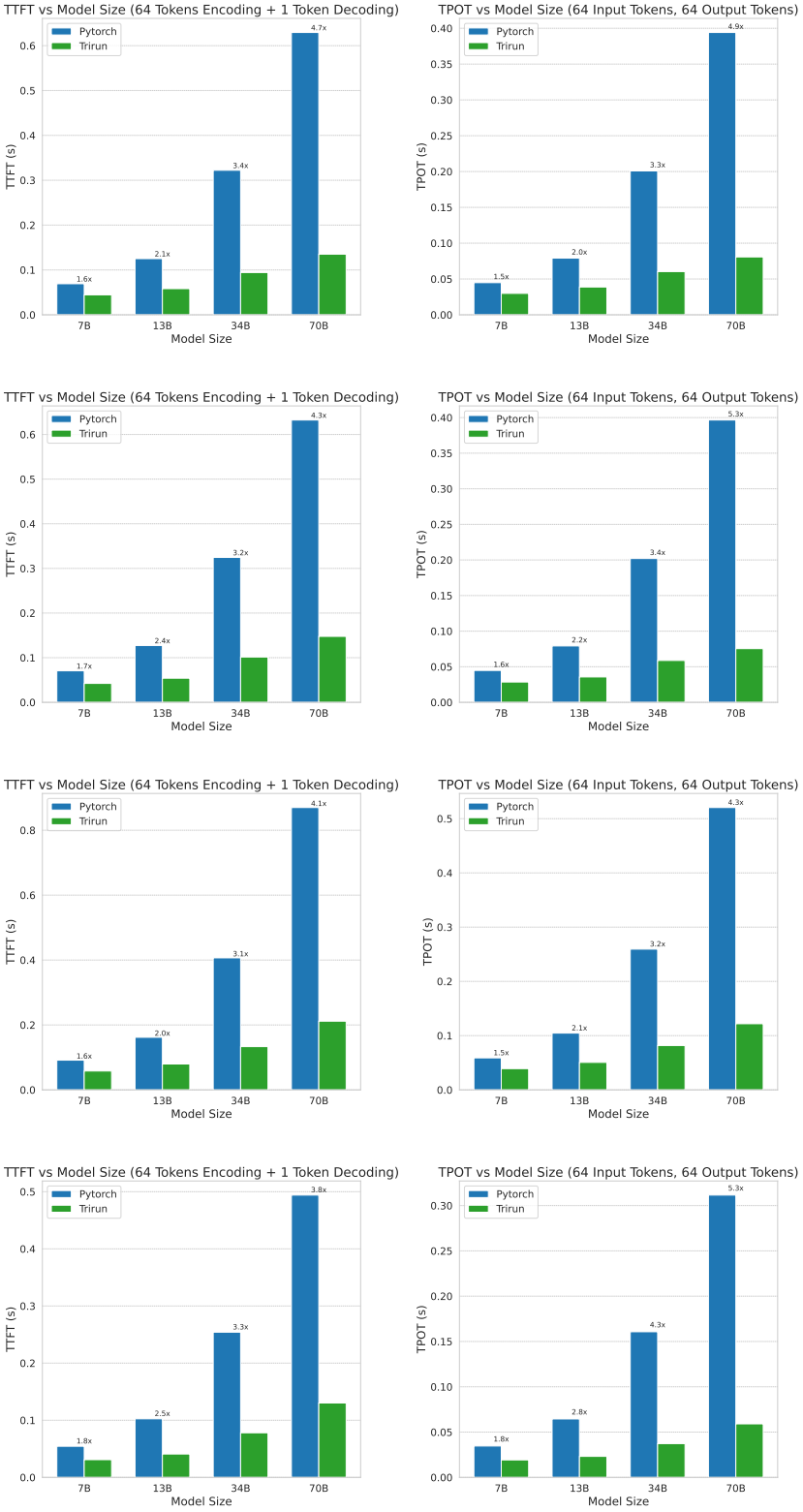

Figure 9: Comparison of TriRun kernels with the FP16 pytorch baseline on NVIDIA L40S, L40, A40 and 4090 (top to bottom). **(a)** Left: Time to first token, **(b)** Right: Time per output token.

G INFERENCE IMPLEMENTATION ON CPUS AND BENCHMARKING ACROSS HARDWARE.

G.1 CPU INFERENCE WITH EFFICIENT PACKING

To assess the effectiveness of our packings, we implemented both packing methods in `ggml.cpp`[2], a framework optimized for running large language models (LLMs) on CPUs. While additional optimizations are possible, our primary focus is on reducing a model's memory footprint and accelerating memory-bound workloads. This is achieved by statically compressing pretrained weights and decompressing them on-the-fly during inference. We begin by demonstrating the efficiency of the CPU implementations for the 1.6-bit packing, referred to as TQ1, and the 2-bit packing, referred to as TQ2. Subsequently, we explore the extension of these methods to high-batch GPU environments.

TQ2: Implementing effective 2 bit for TriLMs. The quantization and dequantization scheme begins with partitioning the input tensor into contiguous, non-overlapping blocks, each containing 256 elements. For each block, a scaling factor (floating-point numbers associated with TriLMs) d_i is calculated as the maximum absolute value of the elements within the block, i.e., $d_i = \max(|b_{ij}|)$, where b_{ij} denotes the j-th element in the i-th block. The inverse scaling factor $\hat{d}_i$ is then defined as $\hat{d}_i = \frac{1}{d_i}$. Each element b_{ij} in the block is quantized to a ternary value by multiplying it by the inverse scaling factor and rounding the result:$q_{ij} = \text{round}(b_{ij} \cdot \hat{d}_i)$. To enable unsigned integer packing, the quantized values are shifted, resulting in $q_{ij} \in \{0, 1, 2\}$. The quantized elements are then packed into a single byte, where every four quantized values are represented using base-4 positional encoding. As a result, each block of 256 elements is stored in 64 bytes, with the scaling factor d_i stored in 2 bytes in float16 format, yielding a total of 66 bytes per block. The dequantization process begins by extracting the scaling factor d_i from its 16-bit floating-point representation, restoring it to full precision. Each packed byte is unpacked by reversing the base-4 encoding to recover the four ternary elements (see G.2). The elements are then adjusted back to their signed values by subtracting 1, resulting in $q_{ij} \in \{-1, 0, 1\}$. Finally, the original block is reconstructed by multiplying each quantized value by the corresponding scaling factor: $\hat{b}_{ij} = d_i \cdot q_{ij}$, where $\hat{b}_{ij}$ denotes the dequantized approximation of b_{ij}. This quantization scheme achieves significant memory efficiency by compressing 256 floating-point values (1024 bytes at 32-bit precision) into just 66 bytes.

TQ1: Implementing effective 1.6 bit for TriLMs. In our implementation, we encode $k = 5$ ternary digits (trits) into $p = 8$ bits, achieving an effective bit rate of 1.6 bits per trit. A key challenge arises in efficiently decoding these packed trits for SIMD-optimized operations, as traditional decoding methods rely on division and modulo operations, which are computationally expensive and ill-suited for vectorization. The conventional approach to decoding a packed byte b involves computing a base-3 integer x using the formula: $x = \frac{\lfloor b \cdot 3^5 - (3^5 - 1) \rfloor}{2^8}$, where $3^5 = 243$ and $2^8 = 256$. Each trit $d_i \in \{-1, 0, 1\}$ is then extracted through the operation: $d_{i+1} = \left\lfloor \frac{x}{3^{4-i}} \right\rfloor \mod 3 \quad \text{for} \quad i = 0, \ldots, 4$. This method incurs high computational costs due to the repeated divisions and modulo operations, which hinder SIMD parallelism. To address these inefficiencies, we exploit the near-equivalence $3^5 \approx 2^8$, enabling a multiplication-based scheme that iteratively extracts trits without explicit division or modulo operations. (See Appendix G.3 for the detailed iterative procedure and its SIMD advantages.)

G.2 ADDITIONAL IMPLEMENTATION DETAILS OF TQ-2

For quantization, the packed value calculation and detailed encoding steps are as follows:

- **Packing**: $q_{\text{packed}} = q_0 + 4q_1 + 16q_2 + 64q_3$.
- **Storage**: 64 bytes for quantized elements + 2 bytes for the float16 scaling factor d_i, totaling 66 bytes per block.

[2]https://github.com/ggerganov/llama.cpp

For dequantization, the explicit unpacking procedure involves:

$$q_0 = q_{\text{packed}} \mod 4, \quad q_1 = \left\lfloor \frac{q_{\text{packed}}}{4} \right\rfloor \mod 4, \quad q_2 = \left\lfloor \frac{q_{\text{packed}}}{16} \right\rfloor \mod 4, \quad q_3 = \left\lfloor \frac{q_{\text{packed}}}{64} \right\rfloor \mod 4.$$

The ternary storage method uses only 2 bits per element, with minimal overhead from the float16 scale per block. The process relies on hardware-friendly bitwise operations for fast packing and unpacking, making it suitable for large-scale deployments in memory-constrained environments while maintaining a balance between numerical fidelity and storage efficiency.

G.3 Additional Implementation details of TQ-1.

In this optimized decoding approach, for each packed byte b, the procedure begins by setting $b_0 = b$. The trits are then extracted iteratively by multiplying b_i by 3, yielding a 9-bit intermediate value. The high byte of this value is then extracted to obtain $d'_i = \left\lfloor \frac{b_i \cdot 3}{2^8} \right\rfloor$, which corresponds to a ternary digit from the set $\{0, 1, 2\}$. The remainder is then updated for the next iteration with the operation:

$$b_{i+1} = (b_i \cdot 3) \& 0xFF.$$

Finally, the values d'_i are normalized by subtracting 1, mapping $\{0, 1, 2\}$ to $\{-1, 0, 1\}$. This iterative decoding method offers significant advantages for SIMD implementation. It replaces non-vectorizable division and modulo operations with fixed-point arithmetic and bitwise masking, both of which are highly SIMD-friendly. The iterative structure minimizes data dependencies, enabling the parallel extraction of trits across multiple packed bytes. By leveraging the numerical proximity of 3^5 and 2^8, this method achieves efficient decoding of ternary values with a computational complexity linear in k. The avoidance of costly arithmetic operations and compatibility with SIMD architectures make this approach particularly well-suited for high-performance applications involving ternary arithmetic.

G.4 Benchmarking across various hardware

Configuration			Model Size				
Tokens	Kernel	Bits	560M	1.1B	1.5B	2.4B	3.9B
Prompt Tokens/Second							
32	F16	16	729.97 ± 6.48	417.6 ± 17.3	295.92 ± 2.38	152.43 ± 0.24	91.2 ± 0.69
	Q4_K	4	490.44 ± 3.17	270.03 ± 1.61	195.22 ± 0.89	106.15 ± 0.55	61.81 ± 0.27
	TQ2_0	2	543.04 ± 3.51	305.1 ± 1.09	221.87 ± 0.60	118.25 ± 0.91	68.62 ± 0.28
	TQ1_0	1.6	617.94 ± 2.78	362.67 ± 2.48	276.51 ± 5.73	145.49 ± 1.36	84.03 ± 0.05
64	F16	16	1223.75 ± 21.71	640.8 ± 2.06	440.41 ± 0.84	247.14 ± 0.48	144.75 ± 0.45
	Q4_K	4	886.14 ± 7.08	450.96 ± 1.26	320.72 ± 0.47	180.05 ± 1.09	105.52 ± 0.41
	TQ2_0	2	951.09 ± 10.60	501.71 ± 1.09	362.98 ± 1.18	198.74 ± 0.35	116.17 ± 0.26
	TQ1_0	1.6	1104.45 ± 11.50	578.31 ± 9.17	435.69 ± 0.40	235.16 ± 0.73	136.06 ± 0.42
Output Tokens/Second							
8	F16	16	170.86 ± 0.82	92.65 ± 0.06	71.51 ± 0.03	44.48 ± 0.04	28.09 ± 0.03
	Q4_K	4	237.84 ± 0.27	134.82 ± 0.26	107.23 ± 0.32	64.79 ± 0.58	43.27 ± 0.68
	TQ2_0	2	278.69 ± 0.59	167.45 ± 0.13	134.02 ± 0.17	86.60 ± 0.03	57.65 ± 0.07
	TQ1_0	1.6	228.83 ± 0.53	125.68 ± 0.08	99.60 ± 0.13	62.31 ± 0.11	39.99 ± 0.05
Total Tokens/Second							
256/8	F16	16	1142.38 ± 7.01	628.91 ± 1.37	489.93 ± 1.45	296.44 ± 0.68	194.58 ± 0.26
	Q4_K	4	1165.63 ± 2.60	620.84 ± 2.17	481.84 ± 1.40	288.85 ± 0.53	186.65 ± 0.92
	TQ2_0	2	1234.71 ± 0.91	668.92 ± 1.26	513.46 ± 1.13	308.62 ± 0.28	202.10 ± 0.52
	TQ1_0	1.6	1241.55 ± 4.71	665.48 ± 4.89	517.62 ± 1.16	311.72 ± 0.51	205.35 ± 0.25

Table 5: Tokens per Second for Different Model Sizes and Quantization Kernels M4 Max (14CPU Coresz). Values represent mean ± standard deviation.

Size	Kernel	#GPU	1	2	4	8	16	32	64
7B	Pytorch	1	0.0210	0.0218	0.0219	0.0221	0.0225	0.0237	0.0244
7B	Trirun	1	0.0135	0.0146	0.0146	0.0145	0.0146	0.0146	0.0146
7B	Speedup	-	1.5556	1.4932	1.5000	1.5241	1.5411	1.6233	1.6712
13B	Pytorch	1	0.0380	0.0401	0.0402	0.0405	0.0411	0.0431	0.0461
13B	Trirun	1	0.0184	0.0195	0.0193	0.0194	0.0207	0.0195	0.0195
13B	Speedup	-	2.0652	2.0564	2.0829	2.0876	1.9855	2.2103	2.3641
34B	Pytorch	2	0.0986	0.1025	0.1027	0.1036	0.1051	0.1126	0.1213
34B	Trirun	1	0.0277	0.0292	0.0288	0.0287	0.0288	0.0288	0.0339
34B	Speedup	-	3.5596	3.5103	3.5660	3.6098	3.6493	3.9097	3.5782
70B	Pytorch	4	0.1952	0.2062	0.2066	0.2076	0.2093	0.2300	0.2352
70B	Trirun	1	0.0381	0.0399	0.0397	0.0396	0.0397	0.0403	0.0544
70B	Speedup	-	5.1234	5.1679	5.2040	5.2424	5.2720	5.7072	4.3235
123B	Trirun	1	0.0544	0.0556	0.0557	0.0559	0.0566	0.0617	0.0850

Table 6: End-2-end inference speedup (Encoding) across varying number of tokens on L40S with our Kernels over Pytorch's Fp16

Size	Kernel	#GPU	1	2	4	8	16	32	64
7B	Pytorch	1	0.0229	0.0449	0.0889	0.1768	0.3529	0.7047	1.4133
7B	Trirun	1	0.0152	0.0299	0.0596	0.1193	0.2374	0.4748	0.9463
7B	Speedup	-	1.5066	1.5017	1.4916	1.4820	1.4865	1.4842	1.4935
13B	Pytorch	1	0.0399	0.0791	0.1573	0.3132	0.6250	1.2494	2.5005
13B	Trirun	1	0.0196	0.0388	0.0777	0.1555	0.3114	0.6219	1.2411
13B	Speedup	-	2.0357	2.0387	2.0245	2.0141	2.0071	2.0090	2.0147
34B	Pytorch	2	0.1009	0.2009	0.4011	0.8014	1.6022	3.2050	6.4134
34B	Trirun	1	0.0300	0.0603	0.1203	0.2408	0.4815	0.9632	1.9272
34B	Speedup	-	3.3633	3.3317	3.3342	3.3281	3.3275	3.3275	3.3278
70B	Pytorch	4	0.1975	0.3941	0.7877	1.5746	3.1491	6.2989	12.6034
70B	Trirun	1	0.0400	0.0808	0.1615	0.3244	0.6501	1.3005	2.6025
70B	Speedup	-	4.9375	4.8775	4.8774	4.8539	4.8440	4.8434	4.8428
123B	Trirun	1	0.0566	0.1126	0.2246	0.4488	0.8976	1.7936	3.5924

Table 7: End-2-end inference speedup (Decoding) across varying number of tokens on L40S with our Kernels over Pytorch's Fp16