Scalable MatMul-free Language Modeling

Rui-Jie Zhu[1], **Yu Zhang**[2], **Ethan Sifferman**[1], **Tyler Sheaves**[3], **Yiqiao Wang**[4], **Dustin Richmond**[1], **Peng Zhou**[1,4], **Jason K. Eshraghian**[1*]

[1]University of California, Santa Cruz [2]Soochow University
[3]University of California, Davis [4]LuxiTech

Abstract

Matrix multiplication (MatMul) typically dominates the overall computational cost of large language models (LLMs). This cost only grows as LLMs scale to larger embedding dimensions and context lengths. In this work, we show that MatMul operations can be completely eliminated from LLMs while maintaining strong performance at billion-parameter scales. Our experiments show that our proposed MatMul-free models achieve performance on-par with state-of-the-art Transformers that require far more memory during inference at a scale up to at least 2.7B parameters. We investigate the scaling laws and find that the performance gap between our MatMul-free models and full precision Transformers narrows as the model size increases. We also provide a GPU-efficient implementation of this model which reduces memory usage by up to 61% over an unoptimized baseline during training. By utilizing an optimized kernel during inference, our model's memory consumption can be reduced by more than 10× compared to unoptimized models. To properly quantify the efficiency of our architecture, we build a custom hardware solution on an FPGA which exploits lightweight operations beyond what GPUs are capable of. We processed billion-parameter scale models at 13W beyond human readable throughput, moving LLMs closer to brain-like efficiency. This work not only shows how far LLMs can be stripped back while still performing effectively, but also points at the types of operations future accelerators should be optimized for in processing the next generation of lightweight LLMs. Our code implementation is available at https://github.com/ridgerchu/matmulfreellm.

1 Introduction

Matrix Multiplication (MatMul) is the dominant operation in most neural networks, where dense layers involve vector-matrix multiplication (VMM), convolutions can be implemented as block-sparse VMMs with shared weights, and self-attention relies on matrix-matrix multiplication (MMM). The prevalence of MatMul is primarily due to Graphics Processing Units (GPUs) being optimized for MatMul operations. By leveraging Compute Unified Device Architecture (CUDA) and highly optimized linear algebra libraries such as cuBLAS, the MatMul operation can be efficiently parallelized and accelerated. This optimization was a key factor in the victory of AlexNet in the ILSVRC2012 competition and a historic marker for the rise of deep learning [1]. AlexNet notably utilized GPUs to boost training speed beyond CPU capabilities, and as such, deep learning won the 'hardware lottery' [2]. It also helped that both training and inference rely on MatMul.

Despite its prevalence in deep learning, MatMul operations account for the dominant portion of computational expense, often consuming the majority of the execution time and memory access during

*Corresponding author, email to: pzhou10@ucsc.edu, jsn@ucsc.edu

Preprint. Under Review.

both training and inference phases. Several works have replaced MatMul with simpler operations through two main strategies. The first strategy involves substituting MatMul with elementary operations, e.g., AdderNet replaces multiplication with signed addition in convolutional neural networks (CNNs) [3]. Given the focus on convolutions, AdderNet is intended for use in computer vision over language modeling.

The second approach employs binary or ternary quantization, simplifying MatMul to operations where values are either flipped or zeroed out before accumulation. Quantization can be applied to either activations or weights: spiking neural networks (SNNs) use binarized activations [4, 5, 6], while binary and ternary neural networks (BNNs and TNNs) use quantized weights [7]. Both methods can also be combined [8, 9].

Recent advances in language modeling, like BitNet [10, 11], demonstrate quantization's scalability, replacing all dense layer weights with binary/ternary values to support up to 3 billion parameters. Despite replacing VMMs with accumulations in all dense layers, BitNet retains the self-attention mechanism which relies on an expensive MMM. Dynamically computed matrices Q (query) and K (key) are multiplied to form the attention map. Since both Q and K matrices are dynamically computed from pre-activation values, achieving optimal hardware efficiency on GPUs requires custom optimizations, such as specialized kernels and advanced memory access patterns. Despite these efforts, such MatMul operations remain resource-intensive on GPUs, as they involve extensive data movement and synchronization which can significantly hinder computational throughput and efficiency [12]. In our experiments, ternary quantization of the attention matrices in BitNet causes a significant drop in performance and failure to reach model convergence (see Fig. 1). This raises the question: is it possible to completely eliminate MatMul from LLMs?

In this work, we develop the first scalable MatMul-free language model (Matmul-free LM) by using additive operations in dense layers and element-wise Hadamard products for self-attention-like functions. Specifically, ternary weights eliminate MatMul in dense layers, similar to BNNs. To remove MatMul from self-attention, we optimize the Gated Recurrent Unit (GRU) [13] to rely solely on element-wise products and show that this model competes with state-of-the-art Transformers while eliminating all MatMul operations.

To quantify the hardware benefits of lightweight models, we provide an optimized GPU implementation in addition to a custom FPGA accelerator. By using fused kernels in the GPU implementation of the ternary dense layers, training is accelerated by 25.6% and memory consumption is reduced by up to 61.0% over an unoptimized baseline on GPU. Furthermore, by employing lower-bit optimized CUDA kernels, inference speed is increased by 4.57 times, and memory usage is reduced by a factor of 10 when the model is scaled up to 13B parameters. This work goes beyond software-only implementations of lightweight models and shows how scalable, yet lightweight, language models can both reduce computational demands and energy use in the real-world.

2 Related Works

Binary, Ternary, and Low-Precision Quantization for Language Models: The effort to quantize language models began with reducing a ternary BERT into a binarized model [14], achieving 41% average accuracy on the GLUE benchmarks with subsequent fine-tuning. Ref. [15] distilled the intermediate outputs from a full precision BERT to a quantized version. Recently, Ref. [16] introduced an incremental quantization approach, progressively quantizing a model from 32-bit to 4-bit, 2-bit, and finally to binary model parameters. Following the quantization of BERT, low-precision language generation models have gained momentum. Ref. [17] used Quantization-Aware Training (QAT) to successfully train a model with 2-bit weights. BitNet pushed this to 3-billion-parameter binary and ternary models while maintaining competitive performance with Llama-like language models [10, 11].

MatMul-free Transformers: The use of MatMul-free Transformers has been largely concentrated in the domain of SNNs. Spikformer led the first integration of the Transformer architecture with SNNs [18, 19], with later work developing alternative Spike-driven Transformers [20, 21]. These techniques demonstrated success in vision tasks. In the language understanding domain, SpikingBERT [22] and SpikeBERT [23] applied SNNs to BERT utilizing knowledge distillation techniques to perform sentiment analysis. In language generation, SpikeGPT trained a 216M-parameter generative model using a spiking RWKV architecture. However, these models remain constrained in size, with

SpikeGPT being the largest, reflecting the challenges of scaling with binarized activations. In addition to SNNs, BNNs have also made significant progress in this area. BinaryViT [24] and BiViT [25] successfully applied Binary Vision Transformers to visual tasks. Beyond these approaches, Kosson et al. [26] achieve multiplication-free training by replacing multiplications, divisions, and non-linearities with piecewise affine approximations while maintaining performance.

3 Method

In this section, we break down the components of the proposed MatMul-free LM. We first describe the MatMul-free dense layers (BitLinear layers) that use ternary weights. By constraining the weights to the set $\{-1, 0, +1\}$ and applying additional quantization techniques, MatMul operations are replaced with addition and negation operations. This reduces computational cost and memory utilization, while preserving the expressiveness of the network. We then provide further detail of our MatMul-free LM architecture, which includes a token mixer for capturing sequential dependencies and a channel mixer for integrating information across embedding dimensions.

The Method section is structured as follows. First, in Sec. 3.1, we provide a comprehensive description of the MatMul-free dense layers with ternary weights, which form the foundation of our approach. Next, Sec. 3.2 introduces our hardware-efficient fused BitLinear layer, designed to optimize the implementation of BitLinear layers. Building upon these components, Sec. 3.3 delves into the details of our MatMul-free LM architecture. We present the MatMul-free token mixer, where we propose the MatMul-free Linear Gated Recurrent Unit (MLGRU), and the MatMul-free channel mixer, which employs the Gated Linear Unit (GLU) with BitLinear layers. By combining the MLGRU token mixer and the GLU channel mixer with ternary weights, our proposed architecture relies solely on addition and element-wise products. Finally, Sec. 3.4 provides an overview of the training details used to optimize our model.

3.1 MatMul-free Dense Layers with Ternary Weights

In a standard dense layer, the MatMul between the input $x \in \mathbb{R}^{1\times d}$ and the weight matrix $W \in \mathbb{R}^{d\times m}$ can be expressed as:

$$y = xW = \sum_{j=1}^{d} x_j W_{ij} \quad \text{for } i = 1, 2, \ldots, m$$

where $y \in \mathbb{R}^{1\times m}$ is the output. To avoid using standard MatMul-based dense layers, we adopt BitNet to replace dense layers containing MatMuls with BitLinear modules, which use ternary weights to transform MatMul operations into pure addition operation with accumulation, i.e., ternary accumulation. When using ternary weights, the elements from the weight matrix W are constrained to values from the set $\{-1, 0, +1\}$. Let $\widetilde{\mathbf{W}}$ denote the ternary weight matrix. The MatMul with ternary weights can be expressed as:

$$\widetilde{\mathbf{Y}} = x \circledast \widetilde{\mathbf{W}} = \sum_{j=1}^{d} x_j \widetilde{\mathbf{W}}_{ij}, \quad \widetilde{\mathbf{W}}_{ij} \in \{-1, 0, +1\}, \quad \text{for } i = 1, 2, \ldots, m$$

where $\widetilde{\mathbf{Y}} \in \mathbb{R}^{1\times m}$ is the output, and $\circledast$ represents a ternary MatMul, which can be simplified to accumulation. Since the ternary weights $\widetilde{\mathbf{W}}_{ij}$ can only take values from $\{-1, 0, +1\}$, the multiplication operation in the MatMul can be replaced by a simple addition or subtraction operation:

$$x_j \widetilde{\mathbf{W}}_{ij} = \begin{cases} x_j, & \text{if } \widetilde{\mathbf{W}}_{ij} = 1, \\ 0, & \text{if } \widetilde{\mathbf{W}}_{ij} = 0, \\ -x_j, & \text{if } \widetilde{\mathbf{W}}_{ij} = -1. \end{cases}$$

Therefore, ternary MatMul can be written as follows:

$$\widetilde{\mathbf{Y}}_i = \sum_{j=1}^{d} x_j \widetilde{\mathbf{W}}_{ij} = \sum_{j:\widetilde{\mathbf{W}}_{ij}=1} x_j - \sum_{j:\widetilde{\mathbf{W}}_{ij}=-1} x_j, \quad \text{for } i = 1, 2, \ldots, m$$

Algorithm 1 Fused RMSNorm and BitLinear Algorithm with Quantization

Define FordwardPass($\mathbf{X}, \mathbf{W}, \boldsymbol{b}, \epsilon$)
$\mathbf{X} \in \mathbb{R}^{M \times N}, \mathbf{W} \in \mathbb{R}^{N \times K}, \boldsymbol{b} \in \mathbb{R}^{K}$

function `forward_pass`($\mathbf{X}, \mathbf{W}, \boldsymbol{b}, \epsilon$)
Load $\mathbf{X}, \mathbf{W}, \boldsymbol{b}, \epsilon$ from HBM
On Chip: $\widetilde{\mathbf{Y}}, \mu, \sigma^2, r \leftarrow$ `rms_norm_fwd`($\mathbf{X}$)
On Chip: $\widetilde{\mathbf{W}} \leftarrow$ `weight_quant`($\mathbf{W}$)
On Chip: $\mathbf{O} \leftarrow \widetilde{\mathbf{Y}} \circledast \widetilde{\mathbf{W}} + \boldsymbol{b}$
Store $\mathbf{O}, \mu, \sigma^2, r$ to HBM
return $\mathbf{O}, \mu, \sigma^2, r$

function `rms_norm_fwd`($\mathbf{X}$)
$\mu, \sigma^2 \leftarrow$ `mean`($\mathbf{X}$), `variance`($\mathbf{X}$)
$r \leftarrow \frac{1}{\sqrt{\sigma^2 + \epsilon}}$
$\widetilde{\mathbf{Y}} \leftarrow$ `activation_quant`($r(\mathbf{X} - \mu)$)
return $\widetilde{\mathbf{Y}}, \mu, \sigma^2, r$

function `activation_quant`($\mathbf{X}$)
$s \leftarrow \frac{127}{\texttt{max}(|\mathbf{X}|)}$ ▷ $\lfloor \cdot \rceil \mid \cdot$ means `round` then `clamp`
$\widetilde{X} \leftarrow \lfloor s\mathbf{X} \rceil \mid_{[-128, 127]} \cdot \frac{1}{s}$
return $\widetilde{X}$

function `weight_quant`($\mathbf{W}$)
$s \leftarrow \frac{1}{\texttt{mean}(|\mathbf{W}|)}$
$\widetilde{\mathbf{W}} \leftarrow \lfloor s\mathbf{X} \rceil \mid_{[-1, 1]} \cdot \frac{1}{s}$
return $\widetilde{\mathbf{W}}$
return $\mathbf{O}$

Define BackwardPass($\mathbf{X}, \mathbf{W}, \boldsymbol{b}, \mathbf{O}, \mathrm{d}\mathbf{O}, \mu, \sigma^2, r$)
$\mathbf{X} \in \mathbb{R}^{M \times N}, \mathbf{W} \in \mathbb{R}^{N \times K}, \boldsymbol{b} \in \mathbb{R}^{K}$
$\mathbf{O} \in \mathbb{R}^{M \times K}, \mathrm{d}\mathbf{O} \in \mathbb{R}^{M \times K}$

function `backward_pass`($\mathbf{X}, \mathbf{W}, \boldsymbol{b}, \mathbf{O}, \mu, \sigma^2, r, \mathrm{d}\mathbf{O}$)
Load $\mathbf{X}, \mathbf{W}, \boldsymbol{b}, \mathbf{O}, \mu, \sigma^2, r, \mathrm{d}\mathbf{O}$ from HBM
On Chip: $\mathrm{d}\mathbf{Y} \leftarrow \mathrm{d}\mathbf{O} \times \mathbf{W}^\top$
On Chip: $\mathbf{dX}, \widetilde{\mathbf{Y}} \leftarrow$ `rms_norm_bwd`($\mathrm{d}\mathbf{Y}, \mathbf{X}, \mu, \sigma^2, r$)
On Chip: $\mathrm{d}\mathbf{W} \leftarrow \mathrm{d}\mathbf{O}^\top \times \widetilde{\mathbf{Y}}$
On Chip: $\mathrm{d}\boldsymbol{b} \leftarrow$ `sum`($\mathrm{d}\mathbf{O}$)
Store $\mathbf{dX}, \mathrm{d}\mathbf{W}, \mathrm{d}\boldsymbol{b}$ to HBM
return $\mathbf{dX}, \mathrm{d}\mathbf{W}, \mathrm{d}\boldsymbol{b}$

function `rms_norm_bwd`($\mathrm{d}\mathbf{Y}, \mathbf{X}, \mu, \sigma^2, r$)
$\widetilde{\mathbf{Y}} \leftarrow$ `activation_quant`($r(\mathbf{X} - \mu)$)
$\mathrm{d}\sigma^2 \leftarrow$ `sum`($\mathrm{d}\mathbf{Y} \times (\mathbf{X} - \mu) \times -0.5 \times r^3$)
$\mathrm{d}\mu \leftarrow$ `sum`($-r\mathrm{d}\mathbf{Y}$) + $\mathrm{d}\sigma^2 \times$ `mean`($\mathbf{X} - \mu$)
$\mathbf{dX} \leftarrow r\mathrm{d}\mathbf{Y} + 2\mathrm{d}\sigma^2(\mathbf{X} - \mu)/N + \mathrm{d}\mu/N$
return $\mathbf{dX}, \widetilde{\mathbf{Y}}$

3.2 Hardware-efficient Fused BitLinear Layer

BitNet showed that stabilizing ternary layers requires an additional RMSNorm before the BitLinear input (for more details, refer to Appendix A). However, the vanilla implementation of BitNet is not efficient. Modern GPUs feature a memory hierarchy with a large, global high-bandwidth memory (HBM) and smaller, faster shared memory (SRAM), and the implementation of BitNet introduced many I/O operations: reading the previous activation into SRAM for RMSNorm, writing it back for quantization, reading it again for quantization, storing it, and reading it once more for the Linear operation. To address this inefficiency, we read the activation only once and apply RMSNorm and quantization as fused operations in SRAM.

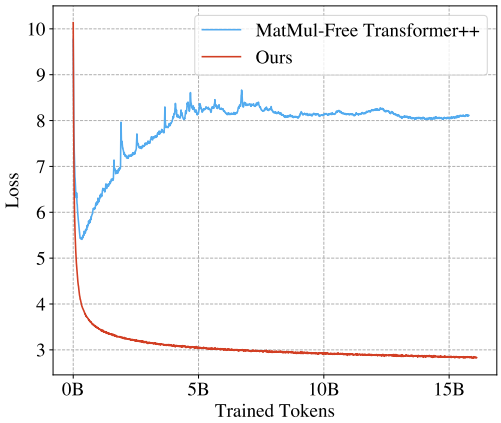


Figure 1: Training loss over steps for the MatMul-free Transformer++ and our proposed method in 370M. The MatMul-free Transformer++ fails to converge, while our method successfully converges under the MatMul-free setting.

Algorithm 1 presents our approach for improving the hardware efficiency of the BitLinear layer by fusing quantized RMSNorm and BitLinear operations. Optimal utilization of SRAM to reduce HBM I/O costs can significantly speed up computations. Since the activations in this model have a larger memory footprint than weights due to ternary weights and the amount of element-wise operations, our optimization efforts focus on activations.

The `forward_pass` function in Algorithm 1 outlines the forward pass of our fused BitLinear layer. It first calls `rms_norm_fwd` to perform RMSNorm on input activations $\mathbf{X}$, loading normalized activations $\mathbf{Y}$, mean μ, variance σ^2, and scaling factor r from HBM. The normalized activations $\mathbf{Y}$ are then quantized to obtain $\widetilde{\mathbf{Y}}$ and the weights $\mathbf{W}$ are quantized using `weight_quant`, with both performed without off-chip data movement. Finally, the output $\mathbf{O}$ is computed on-chip by multiplying

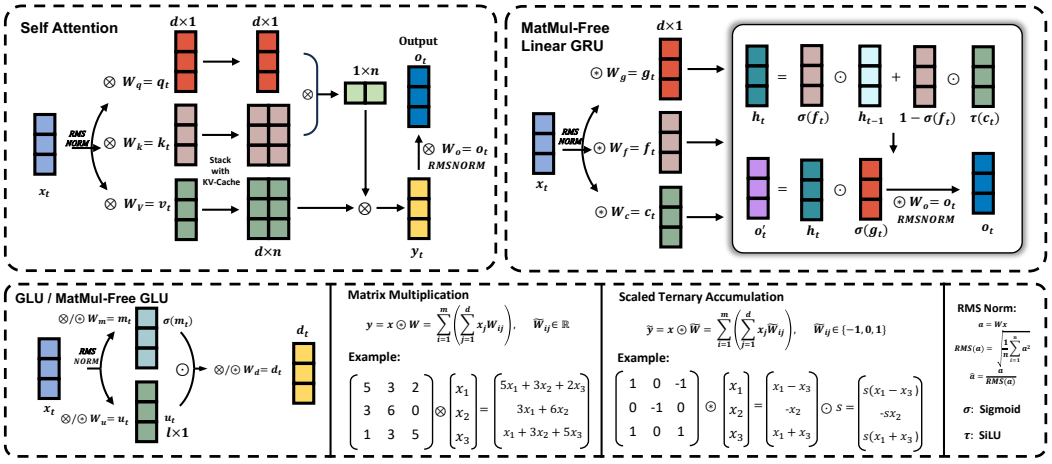


Figure 2: Overview of the MatMul-free LM. The sequence of operations are shown for vanilla self-attention (top-left), the MatMul-free token mixer (top-right), and Ternary Accumulations. The MatMul-free LM employs a MatMul-free token mixer (MLGRU) and a MatMul-free channel mixer (MatMul-free GLU) to maintain the transformer-like architecture while reducing compute cost.

quantized activations $\widetilde{\mathbf{Y}}$ with the ternary weights $\widetilde{\mathbf{W}}$, adding the bias $\boldsymbol{b}$, and then storing the result back to HBM.

The `backward_pass` function first loads $\mathbf{X}$, $\mathbf{W}$, $\boldsymbol{b}$, $\mathbf{O}$, μ, σ^2, r, and the output gradient $\mathrm{d}\mathbf{O}$ from HBM. The gradient $\mathrm{d}\mathbf{Y}$ is then computed on-chip by multiplying the output gradient $\mathrm{d}\mathbf{O}$ with the transposed weight matrix $\mathbf{W}^\top$. Next, it calls `rms_norm_bwd` on-chip to backpropagate through RMSNorm, computing the input gradient $\mathrm{d}\mathbf{X}$. The weight gradient $\mathrm{d}\mathbf{W}$ is calculated on-chip by multiplying the transposed output gradient $\mathrm{d}\mathbf{O}^\top$ with the quantized activations $\widetilde{\mathbf{Y}}$, and the bias gradient $\mathrm{d}\boldsymbol{b}$ is obtained by summing $\mathrm{d}\mathbf{O}$. The computed gradients $\mathbf{dX}$, $\mathrm{d}\mathbf{W}$, and $\mathrm{d}\boldsymbol{b}$ are then stored back to HBM. Sec. 4.4 presents an experimental comparison between Vanilla BitLinear and Fused BitLinear.

3.3 MatMul-free Language Model Architecture

We adopt the perspective from Metaformer [27], which suggests that Transformers consist of a token-mixer (for mixing temporal information, i.e., Self Attention [28], Mamba [29]) and a channel-mixer (for mixing embedding/spatial information, i.e., feed-forward network, Gated Linear Unit (GLU) [30, 31]). A high-level overview of the architecture is shown in Fig. 2.

3.3.1 MatMul-free Token Mixer

Self-attention is the most common token mixer in modern language models, relying on matrix multiplication between three matrices: Q, K, and V. To convert these operations into additions, we binarize or ternarize at least two of the matrices. Assuming all dense layer weights are ternary, we quantize Q and K, resulting in a ternary attention map that eliminates multiplications in self-attention. However, as shown in Fig. 1, the model trained this way fails to converge. One possible explanation is that activations contain outliers crucial for performance but difficult to quantize effectively [32, 33]. To address this challenge, we explore alternative methods for mixing tokens without relying on matrix multiplications.

By resorting to the use of ternary RNNs, which combine element-wise operations and accumulation, it becomes possible to construct a MatMul-free token mixer. Among various RNN architectures, the GRU is noted for its simplicity and efficiency, achieving similar performance to Long Short-Term Memory (LSTM) [34] cells while using fewer gates and having a simpler structure. Thus, we choose the GRU as the foundation for building a MatMul-free token mixer. We first revisit the standard GRU and then demonstrate, step by step, how we derive the MLGRU.

Revisiting the Gated Recurrent Unit The GRU [13] is a widely-used variant of the RNN architecture that is simpler and more computationally efficient compared to the LSTM unit while still maintaining comparable performance. The GRU can be formalized as follows:

$$\boldsymbol{r}_t = \sigma\left(\boldsymbol{x}_t \mathbf{W}_{xr} + \boldsymbol{h}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r\right) \in \mathbb{R}^{1\times d}, \tag{1}$$

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{x}_t \mathbf{W}_{xf} + \boldsymbol{h}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f\right) \in \mathbb{R}^{1\times d}, \tag{2}$$

$$\boldsymbol{c}_t = \tanh\left(\boldsymbol{x}_t \mathbf{W}_{xc} + (\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) \mathbf{W}_{cc} + \mathbf{b}_c\right) \in \mathbb{R}^{1\times d}, \tag{3}$$

$$\boldsymbol{h}_t = \boldsymbol{f}_t \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{f}_t) \odot \boldsymbol{c}_t \in \mathbb{R}^{1\times d}, \tag{4}$$

$$\boldsymbol{o}_t = \boldsymbol{h}_t \tag{5}$$

where $\boldsymbol{x}_t \in \mathbb{R}^{1\times m}$ is the input vector at time step t, $\boldsymbol{h}_{t-1} \in \mathbb{R}^{1\times d}$ is the hidden state vector from the previous time step, σ is the Sigmoid activation function, $\boldsymbol{r}_t \in \mathbb{R}^{1\times d}$ is the reset gate vector, $\boldsymbol{f}_t \in \mathbb{R}^{1\times d}$ is the forget gate vector, $\boldsymbol{c}_t \in \mathbb{R}^{1\times d}$ is the candidate hidden state, $\boldsymbol{h}_t \in \mathbb{R}^{1\times d}$ is the final hidden state vector at time step t, $\boldsymbol{o}_t \in \mathbb{R}^{1\times d}$ is the output vector at time step t, $\mathbf{W}(\cdot) \in \mathbb{R}^{m\times d}$ and $\mathbf{b}(\cdot) \in \mathbb{R}^{1\times d}$ are learnable weight matrices and bias vectors, respectively, $\sigma(\cdot)$ is the sigmoid activation function, and $\odot$ denotes element-wise multiplication.

A key characteristic of the GRU is the coupling of the input gate vector $\boldsymbol{f}_t$ and the forget gate vector $(1 - \boldsymbol{f}_t)$, which together constitute the 'leakage' unit. This leakage unit decays the hidden state $\boldsymbol{h}_{t-1}$ and the candidate hidden state $\boldsymbol{c}_t$ through element-wise multiplication, see Eq. 4. This operation allows the model to adaptively retain information from the previous hidden state $\boldsymbol{h}_{t-1}$ and incorporate new information from the candidate hidden state $\boldsymbol{c}_t$. Importantly, this operation relies solely on element-wise multiplication, avoiding the need for the MatMul. We aim to preserve this property of the GRU while introducing further modifications to create a MatMul-free variant of the model.

MatMul-free Linear Gated Recurrent Unit We first remove hidden-state related weights $\mathbf{W}_{cc}$, $\mathbf{W}_{hr}$, $\mathbf{W}_{hf}$, and the activation between hidden states ($\tanh$). This modification not only makes the model MatMul-free but also enables parallel computation similar to Transformers. This approach is critical for improving computational efficiency, as transcendental functions are expensive to compute accurately, and non-diagonal matrices in the hidden-state would hinder parallel computations. This modification is a key feature of recent RNNs, such as the Linear Recurrent Unit [35], Hawk [36], and RWKV-4 [37]. We then add a data-dependent output gate between $\boldsymbol{h}_t$ and $\boldsymbol{o}_t$, inspired by the LSTM and widely adopted by recent RNN models:

$$\begin{aligned}
\boldsymbol{g}_t &= \sigma(\boldsymbol{x}_t \mathbf{W}_g + \mathbf{b}_g) \in \mathbb{R}^{1\times d}, \\
\boldsymbol{o}'_t &= \boldsymbol{g}_t \odot \boldsymbol{h}_t \in \mathbb{R}^{1\times d}, \\
\boldsymbol{o}_t &= \boldsymbol{o}'_t \mathbf{W}_o + \mathbf{b}_o \in \mathbb{R}^{1\times d}.
\end{aligned}$$

Following the approach of HGRN [38], we further simplify the computation of the candidate hidden state by keeping it as a simple linear transform, rather than coupling it with the hidden state. This can be rewritten as a linear transformation of the input. Finally, we replace all remaining weight matrices with ternary weight matrices, completely removing MatMul operations. The resulting MLGRU architecture can be formalized as follows:

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma\left(\boldsymbol{x}_t \circledast \mathbf{W}_f + \mathbf{b}_f\right) \in \mathbb{R}^{1\times d}, \\
\boldsymbol{c}_t &= \tau\left(\boldsymbol{x}_t \circledast \mathbf{W}_c + \mathbf{b}_c\right) \in \mathbb{R}^{1\times d}, \\
\boldsymbol{h}_t &= \boldsymbol{f}_t \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{f}_t) \odot \boldsymbol{c}_t \in \mathbb{R}^{1\times d}, \\
\boldsymbol{g}_t &= \sigma(\boldsymbol{x}_t \circledast \mathbf{W}_g + \mathbf{b}_g) \in \mathbb{R}^{1\times d}, \\
\boldsymbol{o}'_t &= \boldsymbol{g}_t \odot \boldsymbol{h}_t \in \mathbb{R}^{1\times d}, \\
\boldsymbol{o}_t &= \boldsymbol{o}'_t \circledast \mathbf{W}_o + \mathbf{b}_o \in \mathbb{R}^{1\times d}.
\end{aligned}$$

where $\mathbf{W}_c, \mathbf{W}_f, \mathbf{W}_o, \mathbf{W}_g \in \mathbb{R}^{d\times d}$ are ternary weights, $\boldsymbol{f}_t$ is the forget gate output, σ is the Sigmoid activation function, $\boldsymbol{c}_t$ is the input vector, τ is the SiLU activation function, $\boldsymbol{h}_t$ is the hidden state, $\boldsymbol{g}_t$ is the output gate, $\boldsymbol{o}'_t$ is the intermediate output, and $\boldsymbol{o}_t$ is the final output at time step t. The initial hidden state $\boldsymbol{h}_0$ is set to $\mathbf{0}$. Similarly to HGRN, we also employ the cummax operation to bound the forget gate values in deeper layers closer to 1, though omit this above for brevity. The MLGRU can be viewed as a simplified variant of HGRN that omits complex-valued components and reduces

the hidden state dimension from $2d$ to d. This simplification makes MLGRU more computationally efficient while preserving essential gating mechanisms and ternary weight quantization.

Alternatively to the MLGRU, which employs a data-dependent decay with element-wise product hidden state, the a similarly modified version of the RWKV-4 model can also satisfy the requirement of a MatMul-free token mixer, utilizing static decay and normalization. The performance of using RWKV-4 as a MatMul-free token mixer is discussed in the Experiment section, with a detailed description of the RWKV-4 model provided in Appendix B. However, RWKV-4 introduces exponential and division operations, which are less hardware-efficient compared to the MLGRU.

3.3.2 MatMul-free Channel Mixer

For MatMul-free channel mixing, we use GLU, which is widely adopted in many modern LLMs, including Llama [39, 40, 41], Mistral [42] and RWKV [37], and a BitLinear-adapted version can be expressed as follows:

$$\begin{aligned}
\boldsymbol{g}_t &= \boldsymbol{x}_t \circledast \boldsymbol{W}_g \in \mathbb{R}^{1\times l}, \\
\boldsymbol{u}_t &= \boldsymbol{x}_t \circledast \mathbf{W}_u \in \mathbb{R}^{1\times l}, \\
\boldsymbol{p}_t &= \tau(\boldsymbol{g}_t) \odot \boldsymbol{u}_t \in \mathbb{R}^{1\times l}, \\
\boldsymbol{d}_t &= \boldsymbol{p}_t \circledast \mathbf{W}_d \in \mathbb{R}^{1\times d},
\end{aligned}$$

where τ denotes the SiLU activation function, $\circledast$ represents ternary accumulation, and $\odot$ represents the element-wise product.

The GLU consists of three main steps: 1) *upscaling* the t-step input $\boldsymbol{x}_t \in \mathbb{R}^{1\times d}$ to $\boldsymbol{g}_t, \boldsymbol{u}_t \in \mathbb{R}^{1\times l}$ using weight matrices $\mathbf{W}_g, \mathbf{W}_u \in \mathbb{R}^{d\times l}$ 2) *elementwise gating* $\boldsymbol{u}_t$ with $\boldsymbol{g}_t$ followed by a nonlinearity $f(\cdot)$, where we apply Swish [31]. 3) *Down-scaling* the gated representation $\boldsymbol{p}_t$ back to the original size through a linear transformation $\mathbf{W}_d$. Following Llama [39], we maintain the overall number of parameters of GLU at $8d^2$ by setting the upscaling factor to $\frac{8}{3}d$.

The channel mixer here only consists of dense layers, which are replaced with ternary accumulation operations. By using ternary weights in the BitLinear modules, we can eliminate the need for expensive MatMuls, making the channel mixer more computationally efficient while maintaining its effectiveness in mixing information across channels.

3.4 Training Details

Surrogate Gradient To handle non-differentiable functions such as the Sign and Clip functions during backpropagation, we use the straight-through estimator (STE) [43] as a surrogate function for the gradient. STE allows gradients to flow through the network unaffected by these non-differentiable functions, enabling the training of our quantized model. This technique is widely adopted in BNNs and SNNs.

Large Learning Rate When training a language model with ternary weights, using the same learning rate as regular models can lead to excessively small updates that have no impact on the clipping operation. This prevents weights from being effectively updated and results in biased gradients and update estimates based on the ternary weights. To address this challenge, it is common practice to employ a larger learning rate when training binary or ternary weight language models, as it facilitates faster convergence [44, 45, 11]. In our experiments, we maintain consistent learning rates across both the 370M and 1.3B models, aligning with the approach described in Ref. [46]. Specifically, for the Transformer++ model, we use a learning rate of $3e-4$, while for the MatMul-free LM, we employ a learning rate of $4e-3$, $2.5e-3$, $1.5e-3$ in 370M, 1.5B and 2.7B, respectively. These learning rates are chosen based on the most effective hyperparameter sweeps for faster convergence during the training process.

Learning Rate Scheduler When training conventional Transformers, it is common practice to employ a cosine learning rate scheduler and set a minimal learning rate, typically $0.1\times$ the initial learning rate. We follow this approach when training the full precision Transformer++ model. However, for the MatMul-free LM, the learning dynamics differ from those of conventional Transformer language models, necessitating a different learning strategy. We begin by maintaining the cosine learning rate scheduler and then reduce the learning rate by half midway through the training process.

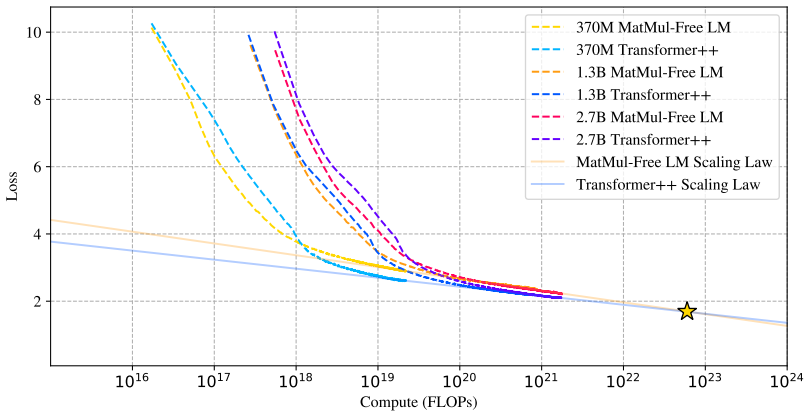


Figure 3: Scaling law comparison between MatMul-free LM and Transformer++ models, depicted through their loss curves. The red lines represent the loss trajectories of the MatMul-free LM, while the blue lines indicate the losses of the Transformer++ models. The star marks the intersection point of the scaling law projection for both model types. MatMul-free LM uses ternary parameters and BF16 activations, whereas Transformer++ uses BF16 parameters and activations.

Interestingly, we observed that during the final training stage, when the network's learning rate approaches 0, the loss decreases significantly, exhibiting an *S*-shaped loss curve. This phenomenon has also been reported by [11, 44] when training binary/ternary language models.

4 Experiments

Our primary focus is testing the MatMul-free LM on moderate-scale language modeling tasks. We compare two variants of our MatMul-free LM against a reproduced advanced Transformer architecture (Transformer++, based on Llama-2) across three model sizes: 370M, 1.3B, and 2.7B parameters. For a fair comparison, all models are pre-trained on the SlimPajama dataset [47], with the 370M model trained on 15 billion tokens, and the 1.3B and 2.7B models trained on 100 billion tokens each. All experiments were conducted using the flash-linear-attention [48] framework, with the Mistral [42] tokenizer (vocab size: 32k) and optimized triton kernel. The training of our models was conducted using 8 NVIDIA H100 GPUs. The training duration was approximately 5 hours for the 370M model, 84 hours for the 1.3B model, and 173 hours for the 2.7B model.

4.1 Scaling Law of MatMul-free LM

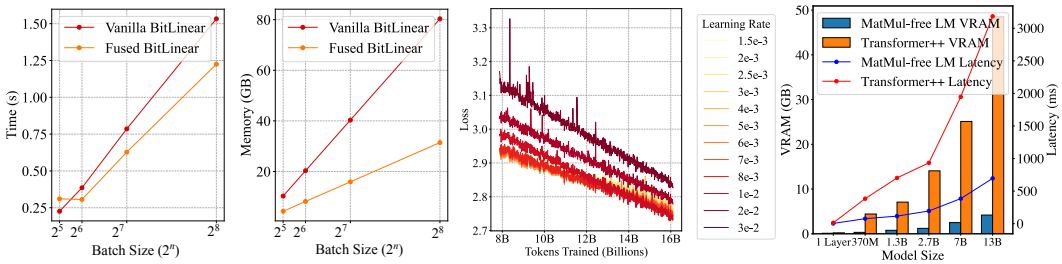
Neural scaling laws posit that model error decreases as a power function of training set size and model size, and have given confidence in performance. Such projections become important as training becomes increasingly expensive with larger models. A widely adopted best practice in LLM training is to first test scalability with smaller models, where scaling laws begin to take effect [49, 50, 51]. The GPT-4 technical report revealed that a prediction model just $1/10,000$ the size of the final model can still accurately forecast the full-sized model performance [52].

We evaluate how the scaling law fits to the 370M, 1.3B and 2.7B parameter models in both Transformer++ and MatMul-free LM, shown in Fig. 3. For a conservative comparison, each operation is treated identically between MatMul-free LM and Transformer++. But note that all weights and activations in Transformer++ are in BF16, while BitLinear layers in MatMul-free LM use ternary parameters, with BF16 activations. As such, an average operation in MatMul-free LM will be computationally cheaper than that of Transformer++.

Interestingly, the scaling projection for the MatMul-free LM exhibits a steeper descent compared to that of Transformer++. This suggests that the MatMul-free LM is more efficient in leveraging additional compute resources to improve performance. As a result, the scaling curve of the MatMul-free LM is projected to intersect with the scaling curve of Transformer++ at approximately

Table 1: Zero-shot accuracy of MatMul-free LM and Transformer++ on benchmark datasets.

Models	Size	ARCe	ARCc	HS	OQ	PQ	WGe	Avg.
370M parameters with 15B training tokens, Layer=24, d=1024								
Transformer++	370M	45.0	24.0	34.3	29.2	64.0	49.9	41.1
MatMul-free RWKV-4	370M	44.7	22.8	31.6	27.8	63.0	50.3	40.0
Ours	370M	42.6	23.8	32.8	28.4	63.0	49.2	40.3
1.3B parameters with 100B training tokens, Layer=24, d=2048								
Transformer++	1.3B	54.1	27.1	49.3	32.4	70.3	54.9	48.0
MatMul-free RWKV-4	1.3B	52.4	25.6	45.1	31.0	68.2	50.5	45.5
Ours	1.3B	54.0	25.9	44.9	31.4	68.4	52.4	46.2
2.7B parameters with 100B training tokens, Layer=32, d=2560								
Transformer++	2.7B	59.7	27.4	54.2	34.4	72.5	56.2	50.7
Ours	2.7B	58.5	29.7	52.3	35.4	71.1	52.1	49.9



(a) Time vs Batch Size: Fused BitLinear vs. Vanilla BitLinear.

(b) Memory vs Batch Size: Fused BitLinear vs. Vanilla BitLinear.

(c) Effect of learning rate on training loss For MatMul-free LM.

(d) Comparison of GPU Memory and Latency in inference between MatMul-free LM and Transformer++.

Figure 4: Performance comparison and analysis of different models and configurations. (a) and (b) show the training performance comparison between Vanilla BitLinear and Fused BitLinear in terms of time and memory consumption as a function of batch size. (c) illustrates the effect of learning rate on training loss for the MatMul-free LM. (d) compares the inference memory consumption and latency between MatMul-free LM and Transformer++ across various model sizes.

10^{23} FLOPs. This compute scale is roughly equivalent to the training FLOPs required for Llama-3 8B (trained with 15 trillion tokens) and Llama-2 70B (trained with 2 trillion tokens), suggesting that MatMul-free LM not only outperforms in efficiency, but can also outperform in terms of loss when scaled up.

4.2 Downstream Tasks

In line with benchmarking in BitNet, we evaluated the zero-shot performance of these models on a range of language tasks, including ARC-Easy [53], ARC-Challenge [53], Hellaswag [54], Winogrande [55], PIQA [56], and OpenbookQA [57]. The results are shown in Tab. 1. Details about the datasets can be found in Appendix C. All evaluations are performed using the LM evaluation harness [58]. The MatMul-free LM models achieve competitive performance compared to the Transformer++ baselines across all tasks, demonstrating its effectiveness in zero-shot learning despite the absence of MatMul operations, and the lower memory required from ternary weights. Notably, the 2.7B MatMul-free LM model outperforms its Transformer++ counterpart on ARC-Challenge and OpenbookQA, while maintaining comparable performance on the other tasks. As the model size increases, the performance gap between MatMul-free LM and Transformer++ narrows, which is consistent with the scaling law. These results highlight that MatMul-free architectures are capable achieving strong zero-shot performance on a diverse set of language tasks, ranging from question answering and commonsense reasoning to physical understanding.

4.3 Learning Rate Analysis

The learning rate is a crucial hyper-parameter in language model training, and models become more sensitive to the learning rate in the ternary/binary weight regime. To determine the optimal learning rate, we conducted a search within the range of $1.5e-3$ to $3e-2$ using our 370M model with a batch size of 50k tokens. The results of this search are shown in Fig. 4(c). The results demonstrate that the final training loss monotonically decreases as the learning rate increases from $1.5e-3$ to $1e-2$. The model only exhibits instability when the learning rate exceeds $2e-2$. This finding suggests that previous works employing ternary weights, such as BitNet, which uses a learning rate of $1.5e-3$, may not be optimal and that higher learning rates could potentially lead to better performance. These findings align with the observations from the Deepseek LLM [59] which found that the optimal learning rate for conventional LLMs is actually larger than the values typically reported in most LLM training setups. Interestingly, we also observed that models trained with larger learning rates at the start of the training process exhibit a more rapid decrease in training loss during the later stages of training compared to those trained with smaller learning rates.

4.4 Training Efficiency Comparison

We evaluate our proposed Fused BitLinear and Vanilla BitLinear implementations in terms of training time and memory usage, shown in Fig.4(a-b). For each experiment, we set the input size and sequence length to 1024. All experiments are conducted using an NVIDIA A100 80GB GPU. Note that during training, the sequence length and batch dimensions are flattened, making the effective batch size the product of these dimensions.

Our experiments show that our fused operator benefits from larger batch sizes in terms of faster training speeds and reduced memory consumption. When the batch size is 2^8, the training speed of the 1.3B parameter model improves from 1.52s to 1.21s per iteration, a 25.6% speedup over the vanilla implementation. Additionally, memory consumption decreases from 82GB to 32GB, a 61.0% reduction in memory usage. The performance of the Fused implementation improves significantly with larger batch sizes, allowing more samples to be processed simultaneously and reducing the total number of iterations.

4.5 Inference Efficiency Comparison

Fig. 4(d) presents a comparison of GPU inference memory consumption and latency between the proposed MatMul-free LM and Transformer++ for various model sizes. In the MatMul-free LM, we employ BitBLAS [60] for acceleration to further improve efficiency. The evaluation is conducted with a batch size of 1 and a sequence length of 2048. The MatMul-free LM consistently demonstrates lower memory usage and latency compared to Transformer++ across all model sizes. For a single layer, the MatMul-free LM requires only 0.12 GB of GPU memory and achieves a latency of 3.79 ms, while Transformer++ consumes 0.21 GB of memory and has a latency of 13.87 ms. As the model size increases, the memory and latency advantages of the MatMul-free LM become more pronounced. It is worth noting that for model sizes larger than 2.7B, the results are simulated using randomly initialized weights. For the largest model size of 13B parameters, the MatMul-free LM uses only 4.19 GB of GPU memory and has a latency of 695.48 ms, whereas Transformer++ requires 48.50 GB of memory and exhibits a latency of 3183.10 ms. These results highlight the efficiency gains achieved by the MatMul-free LM, making it a promising approach for large-scale language modeling tasks, particularly during inference.

5 FPGA Implementation and Results

5.1 Implementation

To test the power usage and effectiveness of the MatMul-free LM on custom hardware that can better exploit ternary operations, we created an FPGA accelerator in SystemVerilog. The overview is shown in Fig. 5.

There are 4 functional units in this design: "Row-wise Operation," "Root Mean Square," "Load Store," and "Ternary Matrix Multiplication," and they each allow for simple out-of-order execution.

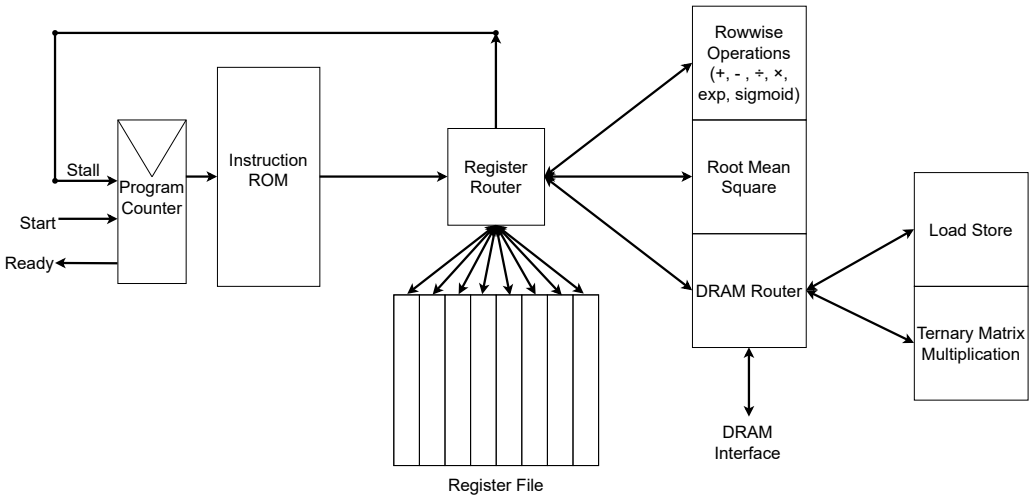


Figure 5: RTL implementation for running MatMul-free token generation

We wrote a custom assembler for our custom instruction set, which was used to convert assembly files into an instruction ROM. The custom instruction set is given below:

- LDV: LoaD Vector from memory
- SV: Store Vector to memory
- ADD: row-wise ADDition
- SUB: row-wise SUBtraction
- MUL: row-wise MULtiplication
- DIV: row-wise DIVision
- EXP: row-wise EXPonential function
- SIG: row-wise SIGmoid
- NORM: NORMalization with root-mean-square
- TMATMUL: Ternary MATrix MULtiplication

The register router delegates incoming instructions to available registers. The register file consists of 8 registers, each storing 1 vector in a separate SRAM array. Each register SRAM array has a read and write port that are delegated to at most one instruction at a time. If an instruction requests access to a functional unit or a register that is busy, the program counter will stall until the functional unit or register has been freed. If two instructions do not block each other, they execute simultaneously.

The "Root Mean Square" functional unit uses a specialized hardware algorithm to preserve precision, and runs in 3 stages. Stage 1 will copy the target vector to an internal-temporary register, and perform a square on each element using a lookup-table. Stage 2 will divide-and-conquer to average neighboring vector elements, generating the Root-Mean-Square result. Stage 3 will perform normalization by dividing each element in the original vector by the Root-Mean-Square result. By using divide-and-conquer for averaging, instead of a typical rolling sum then large divide, rounding errors are significantly reduced.

The "Ternary Matrix Multiplication" functional unit takes in a DRAM address for a ternary matrix, then performs a TMATMUL on the specified vector. Our architecture entirely places the ternary matrices in DRAM. While running a TMATMUL instruction, an SRAM FIFO is simultaneously filled with sequential DRAM fetch results, and emptied by a power-efficient ternary-add operation. At the moment, the three required TMATMUL instructions take up nearly all of the total execution time. In future work, we will introduce parallelism and caching to improve TMATMUL execution time.

Table 2: MatMul-free token generation FPGA core resource utilization and performance metrics. The ternary matrix multiplication operation dominates latency for the current implementation and there is not an observed bottleneck in the local DDR4 bridge. In future implementations, this functional unit will be optimized and the DDR interface will likely become the primary bottleneck.

	%ALMs		%M20Ks		Avg Power (W)		Latency (ms)	
Core Count	**Core**	**Total**	**Core**	**Total**	**Core Active**	**Core Idle**	**Core**	**DDR4**
1	2.9	9	0.01	2.87	13.67	13.68	46.36	0.09
8	23.21	26.9	0.08	3.06	39.78	39.94	46.36	0.18
16	46.43	50.1	0.15	5.13	75.25	73.97	46.36	0.72
26	75.45	100	0.25	22.64	166.30	149.66	46.36	5.76

5.2 Results

The RTL implementation of the MatMul-free token generation core is deployed on a D5005 Stratix 10 programmable acceleration card (PAC) in the Intel FPGA Devcloud. The core completes a forward-pass of a block in 43ms at $d = 512$ and achieves a clock rate of 60MHz. The resource utilization, power and performance of the single-core implementation of a single block ($N = 1$) are shown in Tab. 2. '% ALM Core' refers to the percentage of the total adaptive logic modules used by the core logic, and '%ALM Total' includes the core, the additional interconnect/arbitration logic, and "shell" logic for the FPGA Interface Manager. 'M20K' refers to the utilization of the memory blocks, and indicates that the number of cores are constrained by ALMs, and not on-chip memory (for this DDR implementation). We implement a single token generation core, and estimate the total number of cores that could fit on the platform and the corresponding power, performance and area impact. This is the simplest case where the core only receives 8 bits at a time from memory.

The single core implementation exhibits extremely low dynamic power that is hardly distinguishable from power measured while the core is inactive. Each core requires access to a DDR4 interface and MMIO bridges for host control. In this implementation, the majority of resources are dedicated to the provided shell logic and only 0.4% of programmable logic resources are dedicated to logic for core interconnect and arbitration to DDR4 interfaces/MMIO. As described above, the core latency is primarily due to the larger execution time of the ternary matrix multiply functional unit.

By instead using the full 512-bit DDR4 interface and parallelizing the TMATMUL functional unit, which dominates 99% of core processing time, a further speed-up of approximately 64× is projected, while maintaining the same clock rate without additional optimizations or pipelining, as shown in Table 3. Given the 370M parameter model where $L = 24$, $d = 512$, the total projected runtime is 16.08ms, and a throughput of approximately 62 tokens per second. The 1.3B parameter model, where $L = 24$ and $d = 2048$, has a projected runtime of 42ms, and a throughput of 23.8 tokens per second. This reaches human reading speed at an efficiency that is on par with the power consumption of the human brain. This is for the case of a single core with a single batch of data, and can be scaled up significantly through batch processing by pipelining the single core with a negligible increase in average power, or alternatively, by increasing the core count with an increase in power (Table 2).

Estimates of multi-core implementation latencies are generated by scaling the overheads of the single core implementation and factoring in the growth of logic to accommodate contention on the DDR4 channels. Each core connects to one of four DDR4 channels, and each additional core connected to a channel will double the required arbitration and buffering logic for that channel. As both the host and core share DDR4 channels, this overhead will scale proportional to the number of cores attached to the channel. To mitigate this, future work could bring additional caching optimizations to the core and functional units. Core latency is the compute time of the core from start to ready and DDR4 latency is the required time to transfer input vectors from the host to the PAC local DDR4.

Estimates of multi-core implementation power are calculated by scaling the measured power of a single-core implementation. Idle power is estimated by scaling the total estimated resource overhead of all additional logic added to a constant estimate of idle power consumed by the platform shell. The

Table 3: FPGA Performance Metrics for Different Embedding Dimensions (d)

d	Runtime (ms)	Projected Runtime w/Bursting (ms)	Power (W)		ALM Utilization (%)		Clock (MHz)
			Idle	**Active**	**Core**	**Total**	
512	43	0.67	13.36	13.39	2.8	9	60
1024	112	1.75	13.64	13.65	5.7	11	54
2048	456	7.13	13.92	13.93	11	16	52

single-core active power is scaled by the additional arbitration, interconnect and core overhead. We assume a constant clock rate for all implementations.

We note that the FPGA implementation is done in RTL from top to bottom, and there are many optimizations that could be added. For example, we are not using any vendor-provided IPs, and we are not bursting DDR transactions, both of which would significantly accelerate operation. This approach is to achieve the most generic and cross-platform evaluation possible.

6 Conclusion

We have demonstrated the feasibility and effectiveness of the first scalable MatMul-free language model. Our work challenges the paradigm that MatMul operations are indispensable for building high-performing language models and paves the way for the development of more efficient and hardware-friendly architectures. We achieve performance on par with state-of-the-art Transformers while eliminating the need for MatMul operations, with an optimized implementation that significantly enhances both training and inference efficiency, reducing both memory usage and latency. As the demand for deploying language models on various platforms grows, MatMul-free LMs present a promising direction for creating models that are both effective and resource-efficient. However, one limitation of our work is that the MatMul-free LM has not been tested on extremely large-scale models (e.g., 100B+ parameters) due to computational constraints. This work serves as a call to action for institutions and organizations that have the resources to build the largest language models to invest in accelerating lightweight models. By prioritizing the development and deployment of MatMul-free architectures such as this one, the future of LLMs will only become more accessible, efficient, and sustainable.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[2] Sara Hooker. The hardware lottery. *Communications of the ACM*, 64(12):58–65, 2021.

[3] Hanting Chen, Yunhe Wang, Chunjing Xu, Zhaohui Yang, Chuanjian Liu, Boxin Shi, Chao Xu, Chunfeng Xu, and Qi Tian. The addernet: Do we really need multiplications in deep learning? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1468–1477, 2020.

[4] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[5] Jason K Eshraghian, Max Ward, Emre O Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D Lu. Training spiking neural networks using lessons from deep learning. *Proceedings of the IEEE*, 2023.

[6] 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.

[7] 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. *arXiv preprint arXiv:1602.02830*, 2016.

[8] Sreyes Venkatesh, Razvan Marinescu, and Jason K Eshraghian. Squat: Stateful quantization-aware training in recurrent spiking neural networks. *arXiv preprint arXiv:2404.19668*, 2024.

[9] Jason K Eshraghian, Xinxin Wang, and Wei D Lu. Memristor-based binarized spiking neural networks: Challenges and applications. *IEEE Nanotechnology Magazine*, 16(2):14–23, 2022.

[10] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. Bitnet: Scaling 1-bit transformers for large language models. *arXiv preprint arXiv:2310.11453*, 2023.

[11] 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. *arXiv preprint arXiv:2402.17764*, 2024.

[12] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[13] 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. *arXiv preprint arXiv:1406.1078*, 2014.

[14] Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jing Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. Binarybert: Pushing the limit of bert quantization. *arXiv preprint arXiv:2012.15701*, 2020.

[15] Haotong Qin, Yifu Ding, Mingyuan Zhang, Qinghua Yan, Aishan Liu, Qingqing Dang, Ziwei Liu, and Xianglong Liu. Bibert: Accurate fully binarized bert. *arXiv preprint arXiv:2203.06390*, 2022.

[16] Zechun Liu, Barlas Oguz, Aasish Pappu, Lin Xiao, Scott Yih, Meng Li, Raghuraman Krishnamoorthi, and Yashar Mehdad. Bit: Robustly binarized multi-distilled transformer. *Advances in neural information processing systems*, 35:14303–14316, 2022.

[17] Dayou Du, Yijia Zhang, Shijie Cao, Jiaqi Guo, Ting Cao, Xiaowen Chu, and Ningyi Xu. Bitdistiller: Unleashing the potential of sub-4-bit llms via self-distillation. *arXiv preprint arXiv:2402.10631*, 2024.

[18] Zhaokun Zhou, Yuesheng Zhu, Chao He, Yaowei Wang, Shuicheng Yan, Yonghong Tian, and Li Yuan. Spikformer: When spiking neural network meets transformer. *arXiv preprint arXiv:2209.15425*, 2022.

[19] Zhaokun Zhou, Kaiwei Che, Wei Fang, Keyu Tian, Yuesheng Zhu, Shuicheng Yan, Yonghong Tian, and Li Yuan. Spikformer v2: Join the high accuracy club on imagenet with an snn ticket. *arXiv preprint arXiv:2401.02020*, 2024.

[20] Man Yao, Jiakui Hu, Zhaokun Zhou, Li Yuan, Yonghong Tian, Bo Xu, and Guoqi Li. Spike-driven transformer. *Advances in Neural Information Processing Systems*, 36, 2024.

[21] Man Yao, Jiakui Hu, Tianxiang Hu, Yifan Xu, Zhaokun Zhou, Yonghong Tian, Bo Xu, and Guoqi Li. Spike-driven transformer v2: Meta spiking neural network architecture inspiring the design of next-generation neuromorphic chips. *arXiv preprint arXiv:2404.03663*, 2024.

[22] Malyaban Bal and Abhronil Sengupta. Spikingbert: Distilling bert to train spiking language models using implicit differentiation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 10998–11006, 2024.

[23] Changze Lv, Tianlong Li, Jianhan Xu, Chenxi Gu, Zixuan Ling, Cenyuan Zhang, Xiaoqing Zheng, and Xuanjing Huang. Spikebert: A language spikformer learned from bert with knowledge distillation. 2023.

[24] Phuoc-Hoan Charles Le and Xinlin Li. Binaryvit: pushing binary vision transformers towards convolutional models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4664–4673, 2023.

[25] Yefei He, Zhenyu Lou, Luoming Zhang, Jing Liu, Weijia Wu, Hong Zhou, and Bohan Zhuang. Bivit: Extremely compressed binary vision transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5651–5663, 2023.

[26] Atli Kosson and Martin Jaggi. Multiplication-free transformer training via piecewise affine operations. *Advances in Neural Information Processing Systems*, 36, 2024.

[27] 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*, pages 10819–10829, 2022.

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

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

[30] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International conference on machine learning*, pages 933–941. PMLR, 2017.

[31] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[32] Jiayi Pan, Chengcan Wang, Kaifu Zheng, Yangguang Li, Zhenyu Wang, and Bin Feng. Smoothquant+: Accurate and efficient 4-bit post-training weightquantization for llm. *arXiv preprint arXiv:2312.03788*, 2023.

[33] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[34] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[35] Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. In *International Conference on Machine Learning*, pages 26670–26698. PMLR, 2023.

[36] Soham De, Samuel L Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, et al. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*, 2024.

[37] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

[38] Zhen Qin, Songlin Yang, and Yiran Zhong. Hierarchically gated recurrent neural network for sequence modeling. *Advances in Neural Information Processing Systems*, 36, 2024.

[39] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[40] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[41] AI@Meta. Llama 3 model card. 2024.

[42] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[43] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, abs/1308.3432, 2013.

[44] Yichi Zhang, Ankush Garg, Yuan Cao, Lukasz Lew, Behrooz Ghorbani, Zhiru Zhang, and Orhan Firat. Binarized neural machine translation. *Advances in Neural Information Processing Systems*, 36, 2024.

[45] Zechun Liu, Barlas Oguz, Aasish Pappu, Yangyang Shi, and Raghuraman Krishnamoorthi. Binary and ternary natural language generation. *arXiv preprint arXiv:2306.01841*, 2023.

[46] Zhen Qin, Dong Li, Weigao Sun, Weixuan Sun, Xuyang Shen, Xiaodong Han, Yunshen Wei, Baohong Lv, Fei Yuan, Xiao Luo, et al. Scaling transnormer to 175 billion parameters. *arXiv preprint arXiv:2307.14995*, 2023.

[47] Daria Soboleva, Faisal Al-Khateeb, Robert Myers, Jacob R Steeves, Joel Hestness, and Nolan Dey. SlimPajama: A 627B token cleaned and deduplicated version of RedPajama. https://www.cerebras.net/blog/slimpajama-a-627b-token-cleaned-and-deduplicated-version-of-redpajama, 2023.

[48] Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024.

[49] 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. *arXiv preprint arXiv:2001.08361*, 2020.

[50] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[51] Aiyuan Yang, Bin Xiao, Bingning Wang, Borong Zhang, Ce Bian, Chao Yin, Chenxu Lv, Da Pan, Dian Wang, Dong Yan, et al. Baichuan 2: Open large-scale language models. *arXiv preprint arXiv:2309.10305*, 2023.

[52] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[53] Vikas Yadav, Steven Bethard, and Mihai Surdeanu. Quick and (not so) dirty: Unsupervised selection of justification sentences for multi-hop question answering. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *EMNLP-IJCNLP*, 2019.

[54] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: can a machine really finish your sentence? In *Proceedings of the 57th Conference of the Association for Computational Linguistics*, pages 4791–4800, 2019.

[55] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. WinoGrande: an adversarial winograd schema challenge at scale. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence*, pages 8732–8740, 2020.

[56] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: reasoning about physical commonsense in natural language. *CoRR*, abs/1911.11641, 2019.

[57] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? A new dataset for open book question answering. *CoRR*, abs/1809.02789, 2018.

[58] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 12 2023.

[59] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*, 2024.

[60] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, Yuqing Yang, and Mao Yang. Ladder: Enabling efficient low-precision deep learning computing through hardware-aware tensor transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[61] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

[62] Shuangfei Zhai, Walter Talbott, Nitish Srivastava, Chen Huang, Hanlin Goh, Ruixiang Zhang, and Josh Susskind. An attention free transformer. *arXiv preprint arXiv:2105.14103*, 2021.

APPENDIX

A Quantization for MatMul-free Dense Layers

During training, we first quantized the weights to $\{-1, 0, 1\}$ by using an *absmean* quantization function, which scales the weight matrix by its average absolute value and rounds each element to the nearest ternary integer among $\{-1, 0, +1\}$:

$$\widetilde{\mathbf{W}} \in \mathbb{R}^{n \times m} = \text{RoundClip}(\frac{W}{\gamma + \epsilon}, -1, 1),$$

$$\text{RoundClip}(x, a, b) = \max(a, \min(b, \text{round}(x))),$$

$$\gamma = \frac{1}{nm} \sum_{ij} |W_{ij}|,$$

where n and m are the number of rows and columns of W. After weight quantization, activations are also quantized to 8-bit precision, as is done with BitNet. We use *absmax* quantization, which scales activations into the range $[-Q_b, Q_b]$, given that b is the number of bits and $Q_b = 2^{b-1}$:

$$\widetilde{x} = \text{Quant}(x) = \text{Clip}(x \times \frac{Q_b}{\gamma}, -Q_b + \epsilon, Q_b - \epsilon), \tag{6}$$

$$\text{Clip}(x, a, b) = \max(a, \min(b, x)), \quad \gamma = ||x||_\infty. \tag{7}$$

where ϵ is a small number that prevents overflow during clipping.

With these quantization equations, the MatMul can be written as:

$$y = \widetilde{x} \circledast \widetilde{\mathbf{W}}$$

To preserve variance and maintain numerical stability after quantization, we use RMSNorm [61] before activation quantization, which is also used in BitNet:

$$y = \widetilde{x} \circledast \widetilde{\mathbf{W}} = \text{Quant}(\text{RMSNorm}(x)) \circledast \widetilde{\mathbf{W}} \times \frac{\beta\gamma}{Q_b}, \tag{8}$$

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\text{E}(x^2) + \epsilon}}, \quad \beta = \frac{1}{nm}||W||_1, \quad \gamma = ||x||_\infty.$$

where Q_b is the max value for activation, and β is the mean of the weight matrix.

B RWKV-4 as a MatMul-free Token Mixer

RWKV-4 can also function as a token mixer which utilizes recurrence to mix temporal information and a 1-D hidden states that is updated using element-wise Hadamard products which avoids MatMul operations. This approach offers several advantages over conventional transformers, including computational efficiency, effective propagation of information across time steps, simplified model architecture, and reduced memory usage. Given the good performance of RWKV-4 in capturing dependencies and relationships between tokens across long-ranges of time steps, we additionally tested a ternary version of RWKV-4, though it underperformed compared to what we proposed in the

main manuscript. In the interest of saving the research community compute-hours, we explain the process and report our 'negative' results here. The RWKV-4 token mixer can be expressed as follows:

$$
\begin{aligned}
\mathbf{r}_t &= (\mu_r \mathbf{x}_t + (1-\mu_r)\mathbf{x}_{t-1}) \circledast \mathbf{W}_r \in \mathbb{R}^{1\times d}, \\
\mathbf{k}_t &= (\mu_k \mathbf{x}_t + (1-\mu_k)\mathbf{x}_{t-1}) \circledast \mathbf{W}_k \in \mathbb{R}^{1\times d}, \\
\mathbf{v}_t &= (\mu_v \mathbf{x}_t + (1-\mu_v)\mathbf{x}_{t-1}) \circledast \mathbf{W}_v \in \mathbb{R}^{1\times d}, \\
\mathbf{h}_t &= \frac{\mathbf{a}_{t-1} + e^{\mathbf{m}+\mathbf{k}_t} \odot \mathbf{v}_t}{\mathbf{b}_{t-1} + e^{\mathbf{m}+\mathbf{k}_t}} \in \mathbb{R}^{1\times d}, \\
\mathbf{a}_t &= e^{-\mathbf{w}} \odot \mathbf{a}_{t-1} + e^{\mathbf{k}_t} \odot \mathbf{v}_t, \\
\mathbf{b}_t &= e^{-\mathbf{w}} \odot \mathbf{b}_{t-1} + e^{\mathbf{k}_t} \in \mathbb{R}^{1\times d}, \\
\mathbf{o}_t &= (\sigma(\mathbf{r}_t) \odot \mathbf{h}_t) \circledast \mathbf{W}_o \in \mathbb{R}^{1\times d}, \\
\mathbf{a}_0 &= 0 \in \mathbb{R}^{1\times d}, \\
\mathbf{b}_0 &= 0 \in \mathbb{R}^{1\times d},
\end{aligned}
$$

where $\mathbf{W}_r, \mathbf{W}_k, \mathbf{W}_v, \mathbf{W}_o \in \mathbb{R}^{d\times d}$ are the ternary weights for the block, $a_t, b_t \in \mathbb{R}^{1\times d}$ are the hidden states at timestep t, $\circledast$ represents the ternary accumulation operation, and $odot$ represents the element-wise product. The variables r_t, k_t, v_t are the receptance, key, and value at timestep t, respectively. The decay factors $e^m, e^{-w} \in \mathbb{R}^{1\times d}$ are used to decay the hidden state and input, while μ_r, μ_k, μ_v are time mixing factors that allow 2-gram information flow between tokens, which is also used in RWKV-4. σ denotes the sigmoid function, used for gating.

RWKV-4 retains the softmax-like structure in calculating hidden state h_t, which is adopted from the Attention Free Transformer [62]. This approach has been shown to significantly improve model performance compared to other activation functions. However, the use of softmax introduces two challenges that may hinder the hardware implementation of MatMul-free models. First, the exponential operation, applied to $e^m + k$ in RWKV-4, is a transcendental funcntion and often requires approximations in resource-constrained hardware to compute arbitrarily, or look-up tables which increases memory usage. Second, the division between two dynamic vectors further increases computation cost. Additionally, the division operation expands the hidden state, resulting in a $2 \times d$ hidden state (a_t and b_t). Furthermore, during the training process of RWKV, numerical stability issues can easily arise without proper numerical handling. To avoid these issues, certain measures must be taken for efficient hardware deployment, such as performing computations in log-space.

C Introduction to Benchmark Datasets

- ARC-Easy and ARC-Challenge [53]: Question answering datasets that require models to demonstrate reasoning and knowledge acquisition abilities. ARC-Easy contains questions that are straightforward to answer, while ARC-Challenge includes more difficult questions.
- Hellaswag [54]: A commonsense inference dataset that tests a model's ability to choose the most plausible continuation of a given context. The dataset is constructed from a large corpus of movie scripts and requires models to have a deep understanding of everyday situations.
- Winogrande [55]: A benchmark for measuring a model's ability to perform commonsense reasoning and coreference resolution. The dataset consists of carefully constructed minimal pairs that require models to use commonsense knowledge to resolve ambiguities.
- PIQA [56]: A benchmark for physical commonsense reasoning that tests a model's understanding of physical properties, processes, and interactions. The dataset contains multiple-choice questions that require models to reason about physical scenarios.
- OpenbookQA [57]: A question answering dataset that measures a model's ability to combine scientific facts with commonsense reasoning. The dataset is constructed from a set of science questions and a collection of scientific facts, requiring models to use the provided facts to answer the questions.