Platinum: Path-Adaptable LUT-Based Accelerator Tailored for Low-Bit Weight Matrix Multiplication

Haoxuan Shan, Cong Guo, *Member, IEEE,* Chiyue Wei, Feng Cheng, Junyao Zhang,
Hai (Helen) Li, *Fellow, IEEE,* Yiran Chen, *Fellow, IEEE*
Department of Electrical and Computer Engineering, Duke University
{*haoxuan.shan, cong.guo, chiyue.wei, feng.cheng, junyuao.zhang, hai.li, yiran.chen*}*@duke.edu*

***Abstract*—The rapid scaling of large language models demands more efficient hardware. Quantization offers a promising trade-off between efficiency and performance. With ultra-low-bit quantization, there are abundant opportunities for results reuse, and thus it can be boosted with lookup tables (LUTs) based acceleration. However, existing LUT-based methods suffer from computation and hardware overheads for LUT construction, and rely solely on bit-serial computation, which is suboptimal for ternary-weight networks. We propose *Platinum*, a lightweight ASIC accelerator for integer weight mixed-precision matrix multiplication (mpGEMM) using LUTs. *Platinum* reduces LUT construction overhead via offline-generated construction paths and supports both general bit-serial and optimized ternary-weight execution through adaptive path switching. On BitNet b1.58-3B, *Platinum* achieves up to 73.6×, 4.09×, and 2.15× speedups over SpikingEyeriss, Prosperity, and 16-thread T-MAC (CPU), respectively, along with energy reductions of 32.4×, 3.23×, and 20.9×, all within a 0.96mm^2 chip area. This demonstrates the potential of LUT-based ASICs as efficient, scalable solutions for ultra-low-bit neural networks on edge platforms.**

***Index Terms*—Ultra-low-bit quantization, Ternary weights matrix multiplication, Lookup Table-based accelerator, LLM**

I. Introduction

The size of deep neural networks (DNNs), particularly large language models (LLMs), continues to grow rapidly [1]–[3], leading to increased energy consumption and computational latency. Among core operations in LLMs, general matrix multiplication (GEMM) dominates both fully connected and attention layers. As model sizes scale, GEMM's computational burden grows proportionally, creating major deployment challenges.

Quantization has emerged as a promising solution by reducing computation and memory overhead with minimal accuracy loss. Numerous studies show that aggressive low-bit quantization yields substantial efficiency gains [4]–[9]. Instead of uniformly quantizing weights and activations, applying low-bit quantization to weights alone has shown promise [10]–[13], motivating the need for accelerating mixed-precision GEMM (mpGEMM). For example, BitNet-b1.58 [13] uses ternary weights (-1, 0, 1) to balance efficiency and accuracy.

Interestingly, ultra-low-bit quantization enables LUT-based acceleration strategies, which store precomputed results for reuse. This approach is well-suited for LLMs, where large hidden dimensions allow reusing between thousands of weight multiplications per input. Combined with bit-serial approaches [14]–[17], LUT methods can efficiently support integer-based quantization mpGEMM. As a result, LUT-based acceleration [14], [18]–[23] has recently gained attention for efficient LLM inference.

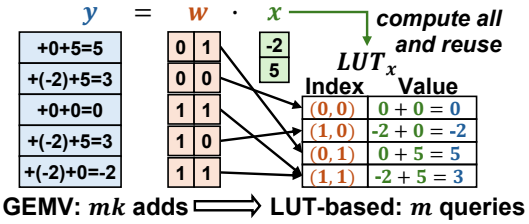

Fig. 1: Comparison between original binary matrix vector multiplication (GEMV) and LUT-based optimization using a binary weight matrix $\mathbf{w}$ of shape (m, k) and input vector $\mathbf{x}$ of shape (k, n), where $m = 5, n = 1, k = 2$. LUT-based optimization reduces computation by a factor of k.

Precomputing all potential LUTs and loading them at runtime is prohibitively costly. Most existing methods build LUTs at runtime for incoming activations and query with weights. Minimizing LUT construction overhead is thus critical. Prosperity [24] addresses this by using "shortcuts" that reuse intermediate entries during construction. However, its dynamic scheduling incurs high area and energy overhead—unnecessary for models like BitNet with uniformly distributed weights that utilize most LUT entries.

Moreover, existing ASICs typically use bit-serial execution for general weight precision, introducing redundancy for bitwidth like ternary weights. Bit-serial encoding for ternary numbers uses 2 bits, exceeding the optimal 1.58 bits ($\log_2 3$), and incurs overhead from merging partial sums. In contrast, ternary LUTs directly represent final results, eliminating this cost. Experiments show over 1.3× improvement with ternary LUTs over binary LUTs for ternary weights, highlighting optimization potential for specific bitwidths.

To address these gaps, we propose ***Platinum***, a Path-adaptable LUT-based Accelerator Tailored for Low-bit weights mpGEMM. Our key contributions include:

- We designed *Platinum*, a novel LUT-based accelerator for mpGEMM with integer weights, leveraging a disaggregated path-based LUT construction framework to reduce LUT generation cost and minimize hardware overhead.

This work was supported in part by NSF grants 2328805 and 2112562. The authors thank the anonymous reviewers for their constructive feedback. Cong Guo is the corresponding author of this paper.

- The accelerator supports integer-weight mpGEMM via bit-serial execution. By switching the construction path, it can be reconfigured for optimized execution at specific weight precision such as ternary.
- The architecture is optimized for parallelism and dataflow efficiency, and is lightweight and modular for edge deployment, with a chip area of just $0.96\,\text{mm}^2$.
- *Platinum* achieves up to $73.6\times$ speedup and $32.4\times$ energy savings over state-of-the-art baselines on BitNet b1.58 3B, one of the most representative ternary weight models.

II. Background

Bit-serial LUT-based mpGEMM. Bit-serial execution decomposes a b-bit integer weight into b binary matrices $w^{(0)}, ..., w^{(b-1)}$, and computes $y = wx$ by summing $2^i w^{(i)} x$. For a given input x, all $w^{(i)}$ share the same LUT, increasing reuse. In Figure 1, each output element in the product between a binary matrix and input vector has at most 2^k unique values, where k is the input vector length. These values can be precomputed and then queried, replacing mk additions with m LUT accesses and $(k-1)2^k$ additions for LUT construction. When m is large and LUT access cost is lower than k additions, this method reduces runtime computation. LLMs naturally satisfy the first condition due to large hidden dimensions, and the second is met by choosing a large k while ensuring LUTs still reside in cache or on-chip buffers.

Tradeoffs in LUT construction. Precomputing all LUTs offline is impractical. For example, buffering all LUTs for 8-bit activations with $k = 2$ already require 4GB of memory, which is a high cost. As a result, most works build LUTs at runtime for each input vector. However, constructing binary LUTs requires $(k-1)2^k$ additions if each entry is computed independently, which grows exponentially. To mitigate the cost, prior works [18], [24] introduce shortcut-based construction. Prosperity [24] dynamically detects shortcuts between LUT entries to reduce cost, which is effective when only a few entries are accessed. However, dynamic construction incurs significant overhead—its runtime shortcut scheduling modules

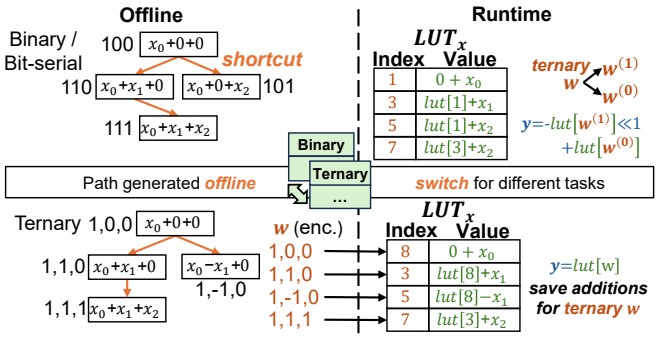

Fig. 2: *Platinum* leveraging programmable construction path to support both bit-serial LUT-based and ternary LUT-based mpGEMM. The path generation is disaggregated to offline to reduce runtime overhead. Refer to Section III-C for more details of ternary LUT benefits for ternary weights.

account for 24% of chip area and 32.3% of total power. For models like BitNet-b1.58, whose uniformly distributed ternary weights lead to high LUT utilization, runtime path scheduling is thus unnecessary. For instance, only 1.16% of LUT entries remain unused when tiling the M-dimension with size 1080.

In contrast, our accelerator, *Platinum*, eliminates runtime overhead by disaggregating LUT construction into offline path generation and lightweight online path-based construction (Figure 2). This removes the need for dynamic scheduling hardware, reducing area and energy costs while enabling more processing elements (PEs) for higher throughput. Moreover, the reprogrammable path design allows *Platinum* to support both bit-serial and ternary LUTs. This enables optimized execution for ternary-weight networks, while maintaining compatibility with general weight precisions.

III. Methodologies

A. Platinum Overview

Platinum (Figure 3) comprises L Platinum Processing Elements (PPEs), aggregators, high-bandwidth on-chip buffers, and Special Functional Units (SFUs). In each single computation round, it first constructs L LUTs based on input with shape $(Lc, 1)$ and then sequentially queries (m, Lc) weights and aggregates the results into the output buffer. The complete algorithm is shown in Algorithm 1. c represents the chunk size (vector length k) for constructing LUT. The chunk size is set to 5 for ternary weights to align with the weight encoding (Section III-C), which requires LUTs with 128 entries.

PPEs are responsible for LUT construction and query. Each includes a controller, adders, and a dedicated LUT buffer. Aggregators share the adders in PPEs and work alongside additional adders to form a pipelined adder tree for efficient aggregation and accumulation during the Reduction stage. The rationale for including these extra adders is discussed in Section IV-B. LUT buffers are implemented using on-chip SRAM. Each entry is 8-bit to align with Bitnet's 8-bit activation. Given that the LUT occupies a relatively small portion of the overall chip area, using higher precision (i.e., more bits per entry) is also feasible if needed. Each LUT buffer includes a read-write port and an additional read-only port to prevent pipeline stalls. Both ports are used simultaneously during the query phase to maximize throughput. The remaining buffers include storage for weights, inputs, outputs, and build paths. They are divided into banks to maximize throughput. These buffers are accessed sequentially during execution, benefiting from prefetching. SFUs are included to support operations beyond mpGEMM, such as vector multiplications and activation functions. Since this work focuses on GEMM, it only serves as a hardware overhead for fair comparison with baselines.

Platinum employs a disaggregated path-based LUT construction scheme (Algorithm 2). The build path, generated offline, represents shortcut paths as illustrated in Figure 2 and is sequentially loaded during the LUT construct stage of runtime until a "Finish" token is reached. Each shortcut path triggers the computation of a new LUT entry by combining an existing entry with an input element. A dedicated four-stage

Algorithm 1 Single Computation Round

Input: Weights (m, Lc), Inputs $(Lc, 1)$, Outputs $(m, 1)$
Output: Outputs $(m, 1)$
Hardware: Platinum Processing Elements (PPE), Aggregator (AGG)
function FLIP(v,s) **return** if s then $-v$ else v
function PPE.QUERY($index$)
    return FLIP($LUT[index[6:0]]$, $index[7]$)
end function
function AGG.REDUCE($vals$, $output$)
    $sum \leftarrow$ AGG.AGGREGATE(vals)
    $output \leftarrow output + sum$
end function
function ROUND($inputs$, $weights$, $outputs$)
    PPE.CONSTRUCT($inputs$) (Algorithm 2)
    for $i = 0$ to $m - 1$ **do**
        $vals \leftarrow$ PPE.QUERY($weights[i]$)
        AGG.REDUCE($vals$, $outputs[i]$)
    end for
end function

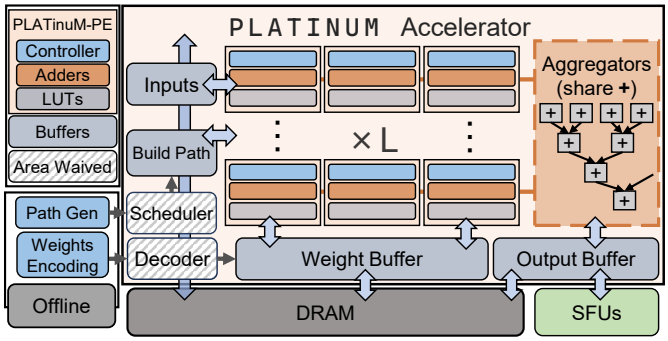

Fig. 3: Architecture of *Platinum* Processor

construction pipeline (Figure 4) is implemented to support runtime construction and minimize construction stalls. At each cycle, the pipeline fetches a path entry from the build path buffer and sends to PPEs. The controller in PPEs extracts the source and destination addresses, input index, and sign bit. In the second stage, the required values are read from the LUT and input buffers. These values are combined—either added or subtracted—in the third stage, and the result is written back in the final stage.

B. Offline Path Generation with MST

Previous designs construct each LUT entry in parallel by directly computing from the inputs. While this strategy is well-suited for GPUs, which benefit from abundant parallelism and rich computational resources, it introduces significant overhead in compact accelerators with limited on-chip resources. This overhead is tolerable for small chunk sizes but a more efficient construction method is desirable for larger chunk sizes and ternary weights. BIQGEMM [18] proposed a dynamic programming approach for binary weights, obtaining each LUT entry with just a left shift and one addition. Prosperity [24] leveraged "shortcut" to maximize the reuse of entries already computed. Inspired by these ideas, we propose a

Algorithm 2 Path-Based LUT Construction

Parameters: Chunk size c, Table size S
Input: Input activations $a_1, \ldots, a_c$; build_path[S]
Output: Look-up Table LUT[S]
function LUT_CONSTRUCTION($a_1, \ldots, a_c$)
    Initialize LUT[0] $\leftarrow$ 0, PC $\leftarrow$ 0
    while build_path[PC] is not "Finish" **do**
        path $\leftarrow$ build_path[PC]
        dst, src, j, sign $\leftarrow$ path
        LUT[dst] $\leftarrow$ LUT[src] +Flip($a_j, sign$)
        PC $\leftarrow$ PC +1
    end while
    return LUT
end function

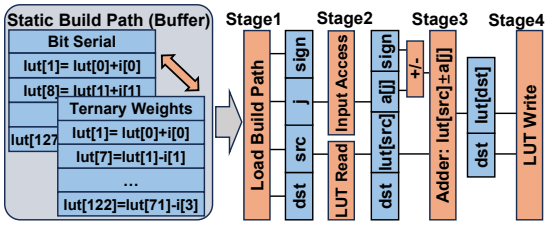

Fig. 4: Four-stage construction pipeline with build path.

new method from a graph-theoretic perspective with minimal construction cost.

The LUT construction can be formalized as a directed hypergraph. Each node represents a LUT entry, and each hyperedge represents a computation. For example, a hyperedge $e = (\{n_1, n_2\}, n_3, \texttt{add\_cost})$ denotes an addition operation $n_3 = n_1 + n_2$. Given a set of source nodes $\{s_j\}$, a valid build path is a subset of hyperedges that connects all other nodes to the source set. The optimal build path minimizes the total cost and corresponds to a minimum spanning tree (MST) in the hypergraph. In our setting, the source node is lut_0, representing zero, and the operations are restricted to additions/subtractions of input entry, which are reversible. This allows us to convert the hypergraph into a standard undirected graph and apply classical MST algorithms such as Prim's algorithm [25] for efficient construction. The resulting MST yields a build path with operations of the form $(\texttt{dst}, \texttt{src}, j, \texttt{flip})$, representing $lut[\texttt{dst}] = lut[\texttt{src}] \pm a_j$. The build order is naturally topologically sorted from the MST, ensuring correct data dependencies. Notably, for $c = 5$, the shortest Read-After-Write (RAW) dependency distance exceeds the number of pipeline stages, eliminating the need for additional hardware hazard handling. Combined with LUT size reduction via symmetry, our MST-based approach reduces the number of additions by $\sim 10\times$ at $c = 5$, compared to naive LUT construction.

The generated build path is stored in on-chip buffer and retrieved by the four-stage construction pipeline at runtime. By generating separate build paths for bit-serial and ternary LUT, and dynamically switching paths during execution (Figure 4), our accelerator efficiently supports multiple weight precisions, including GEMM in attention layers.

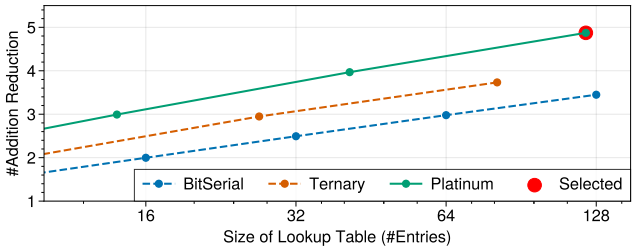

Fig. 5: #Addition reduction for ternary weights mpGEMM over LUT sizes. Assume $M = 1080$.

C. LUT Method Tailored for Ternary Weights mpGEMM

In this section, we present a ternary LUT-based mpGEMM approach and demonstrate why it outperforms bit-serial LUT methods when handling ternary weights. For ternary mpGEMM with weights of size $M \times K$ and input of size $K \times N$, naive computation requires MKN additions. Subtractions are counted as additions due to negligible sign-flip cost. The bit-serial LUT method reduces this cost by approximately $c/2$ when M is large, as shown in Equation (1). For each c-element input chunk, $c2^c$ additions are required to construct the LUT, followed by $2M$ LUT queries and M additions to merge results. This process repeats $\lceil K/c \rceil N$ times. An additional $M(\lceil K/c \rceil - 1)N$ additions are needed to accumulate partial results.

$$\#\text{add}_{\text{bs}} = \left[\left\lceil \frac{K}{c} \right\rceil c2^c + M \left\lceil \frac{K}{c} \right\rceil + M \left(\left\lceil \frac{K}{c} \right\rceil - 1 \right) \right] N \quad (1)$$

In contrast, the ternary LUT-based method avoids the $M\lceil K/c \rceil N$ term and has lower total cost with proper choice of c, as shown in Equation (2). Here, LUT construction requires $c3^c$ additions per chunk, and result accumulation is the same as in the bit-serial case.

$$\#\text{add}_{\text{ter}} = \left[\left\lceil \frac{K}{c} \right\rceil c3^c + M \left(\left\lceil \frac{K}{c} \right\rceil - 1 \right) \right] N \quad (2)$$

We further reduce cost by exploiting symmetry (mirror consolidation) in ternary weights, a technique used in prior works [14], [19], [21]. For example, weight vectors $[-1, 1]$ and $[1, -1]$ are symmetric; when multiplied with the same input vector, the output of one can be obtained by negating the other. We store only LUT entries where the leftmost non-zero value is $+1$ (e.g., [0, 1, -1]) and infer the rest by negation, reducing LUT size to $\lceil 3^c/2 \rceil$. Combined with our offline path-based LUT construction (Section III-B), this approach reduces LUT construction cost from $c3^c$ to $\lceil 3^c/2 \rceil$, yielding a $\sim 2c\times$ reduction compared to naive ternary LUT construction (Equation (3)). As shown in Figure 5, our method achieves the lowest addition count across varying chunk sizes for BitNet b1.58 3B.

$$\#\text{add}_{\text{platinum}} = \left[\left\lceil \frac{K}{c} \right\rceil \left\lceil \frac{3^c}{2} \right\rceil + M \left(\left\lceil \frac{K}{c} \right\rceil - 1 \right) \right] N \quad (3)$$

Compact weight encoding is also critical. Storing each ternary weight as a byte is highly redundant. A common solution, e.g., T-MAC [14], uses 2-bit encoding, which still far exceeds the optimal 1.58 bits. Our work adopts a strategy

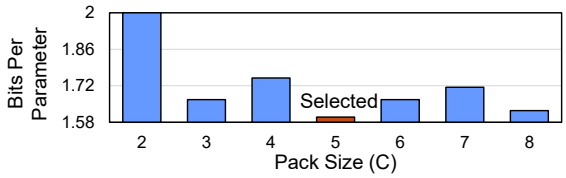

Fig. 6: Average bits per weight w.r.t the pack size c.

similar to that used in Bitnet.cpp [21]. Every c ternary weights are packed into a base-3 integer, requiring only $\lceil \log_2 3^c \rceil$ bits. The number is further split into a sign bit and $\lceil \log_2 3^c \rceil - 1$ index bits to preserve symmetry without decoding. As shown in Figure 6, encoding overhead is minimized at $c = 5$, achieving 1.6 bits per weight. This fits neatly into a byte and aligns well with typical memory systems. Additionally, we reorder indices based on the construction path to ensure LUT entries are accessed sequentially and no hazard detection is needed for the construction pipeline, forming the final encoded weight stream. Since the encoding is performed offline, it reduces hardware complexity while avoiding runtime overhead.

IV. System-Level Design

A. Parallelism and Scaling

There are two primary strategies for improving accelerator throughput: (1) enhancing the processing capacity of each PE, and (2) increasing the number of PEs. Our LUT-based GEMM approach exploits the fact that each input vector is reused across many weight vectors, and vice versa-each weight vector serves as a query across multiple input chunks, as also observed in [20]. Rather than computing a standard $M \times c$ weight matrix with a $c \times 1$ input, we extend processing to multiple input columns simultaneously, exploiting parallelism in N-dimension. Given $c \times n_{\text{cols}}$ input blocks, we construct a LUT with block size equal to n_{cols}, allowing each query to return a block of n_{cols} partial sums. This technique reduces query overhead and overall area consumption compared to a single-column LUT design. Theoretically, increasing n_{cols} boosts throughput. However, Cacti 7.0 [26] analysis shows diminishing area efficiency beyond $n_{\text{cols}} > 8$. Additionally, for small N, large n_{cols} values cause resource under-utilization. Based on this, we set $n_{\text{cols}} = 8$ in our final design.

The second scaling axis involves increasing the number of PEs to enable parallel processing along the K and N dimensions. With L PEs, we process $L \cdot c \times n_{\text{cols}}$ inputs in parallel, and stream the resulting partial sums directly to the accumulators, reducing output buffer pressure. However, the choice of L is constrained by memory bandwidth and tiling granularity. Excessive L can result in resource underutilization. We empirically set $L = 52$ to balance throughput and resource efficiency, also facilitating tiling for BitNet-b1.58 models.

B. Utilization Improvements

We observed an imbalance between computation and memory access requirements during the LUT construction, query, and reduction stages. In the construction stage, a stall-free design requires roughly one addition, one LUT read, and one LUT write per cycle, implying one adder per two LUT ports.

In contrast, during querying, both LUT ports are used for queries, yielding two partial sums per cycle. This necessitates two adders for reduction to maximize the throughput. This discrepancy results in idle resources: either adders or LUT ports. Since the query and reduction stages dominate execution time, it is more beneficial to maximize resource utilization during these stages. To this end, we provision extra adders to fully support the reduction stage's demands. It ensures theoretically near 100% utilization of both LUT ports and an average adder utilization of 90.5%, resulting in high overall efficiency.

C. Tiling and Stationarity

Tiling is critical for GEMM acceleration due to the high energy and latency cost of DRAM accesses. Large buffers are allocated for weights and outputs to maintain them on-chip while provisioning only minimal input buffering, since it is only accessed during LUT construction. To determine optimal tiling configurations, we conducted design space exploration on the prefill stages of three BitNet-b1.58 models. Alongside tile sizes, we evaluated various data stationary strategies. The evaluations over configurations across three models are shown in Figure 7. To strike a balance between latency, energy, and area, we picked m_tiled=1080, k_tiled=520, n_tiled=32 with mnk-stationary, which is marked with red in the figure. We integrate 272KB on-chip SRAM for buffers, together with 52KB LUT, resulting only a total of 324KB.

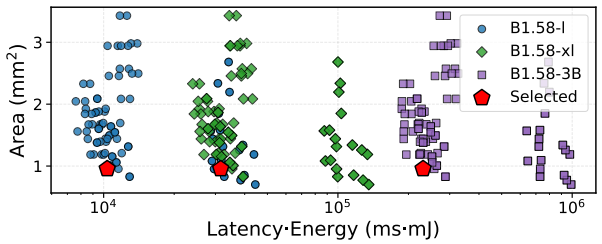

Fig. 7: Design Space Exploration for Tilling Sizes

V. Evaluation

A. Experimental Setup

We evaluate our accelerator using a comprehensive methodology that combines RTL synthesis, memory modeling, DRAM power estimation, and cycle-accurate simulation. Both bit-serial based execution (*Platinum*-bs) and execution optimized for ternary weights are evaluated (*Platinum*). *Platinum*-bs adapts binary LUT and $c = 7$ to align with the LUT size. For benchmarking, we compare our design against SpikingEyeriss [27], Prosperity [24], and T-MAC [14] under equivalent workloads. Since SpikingEyeriss [24], [27], [28], and Prosperity [24] are designed for spiking neural networks, they are evaluated with a bit-serial execution. Specifically, for a ternary-weight matrix W, each ternary-weight mpGEMM is computed with two passes, handling '1' and '-1' in weights separately. The final result is then obtained by subtracting the results from two passes. T-MAC [14] provides a CPU-based implementation. We evaluate it on an Apple M2 Pro laptop with 16 threads, which serves as a strong baseline.

Model and Kernel Extraction. We extract evaluation kernels from the BitNet-b1.58 model suite [13], which includes b1.58-l, b1.58-xl, and b1.58-3B models with 700M, 1.3B, and 3B parameters, respectively. These models utilize BitLinear layers as their primary compute blocks. We extract the input (K) and output (M) feature dimensions from these layers, and vary the product of batch size and sequence length (N) to evaluate both the prefill and decode stages of LLM inference. Specifically, we test with $N = 1024$ for the prefill stage and $N = 8$ for the decode stage.

Hardware Modeling. The RTL of our *Platinum* processing elements is implemented in SystemVerilog. We use Synopsys Design Compiler with ARM's standard cell library targeting a commercial 28nm technology node and 500 MHz frequency to obtain post-synthesis estimates for area and dynamic/static power. On-chip SRAM buffer characteristics are modeled using CACTI 7.0 [26], configured for the same process. Off-chip DRAM is modeled using DRAMsim3 [29] to estimate memory system energy consumption. We followed the settings in [24] for fair comparison, which employs 64GB DDR4 2133R with 64GB/s as maximum bandwidth. We extend the open-source Prosperity [24] simulator to support BitNet-b1.58 kernels and to develop a cycle-accurate simulator that captures computation cycles, memory accesses, and PE activities. Using trace outputs and synthesized area/power data, we compute total cycles and energy per kernel.

B. Area and Power Breakdown

The chip has an overall size of 0.96 mm^2. On-chip buffers dominate the overall chip area. Specifically, the buffers for storing weights and activations occupy approximately 65% of the total area. When including the memory allocated for LUTs, this figure rises to 83.3%, underscoring the memory-intensive nature of a LUT-based accelerator. The aggregator and PPEs, which carry out the core computations, account for just 15% of the area. The compact footprint of the PPE highlights the area efficiency of compute resources with LUT-based method. This benefit becomes even more pronounced when operator area costs increase, such as FP16 adders used for FP16 activations.

When running the prefill workloads of the b1.58-3B model, our accelerator consumes a power of 3.2W. DRAM access and weight buffer access are the most power-intensive operations, contributing 53.5% and 31.6% of total power, respectively.

TABLE I: Accelerator specifications.

	Eyeriss [27]	Prosperity [24]	T-MAC [14]	***Platinum*** **Ours**
Type	ASIC	ASIC	CPU†	ASIC
Freq. (MHz)	500	500	3490	500
Tech. (nm)	28	28	5	28
# of PEs	168	256	—	416
Area (mm^2)	1.07	1.06*	289	0.955
Throughput‡ (GOP/s)	20.8	375	715	1534

* Prosperity is scaled for fair comparison.
† T-MAC: Benchmarked on Apple M2 Pro Laptop.
‡ The number of operations is based on the additions/subtractions for naively compute the b1.58-3B model with N=1024.

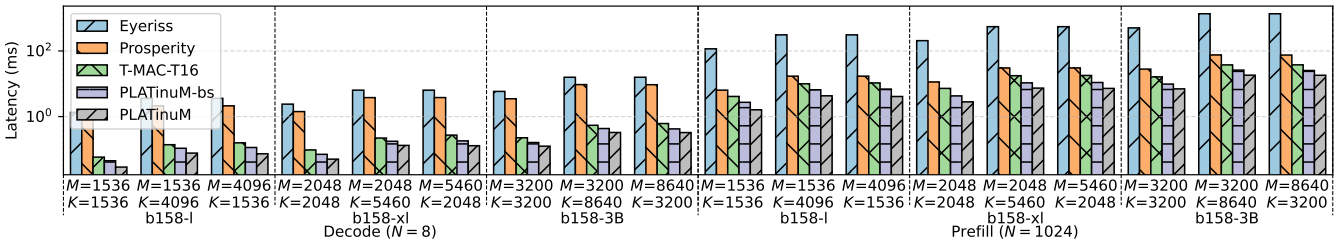

Fig. 8: Comparison of kernel latency across *Platinum*, CPU (T-MAC), SpikingEyeriss, and Prosperity.

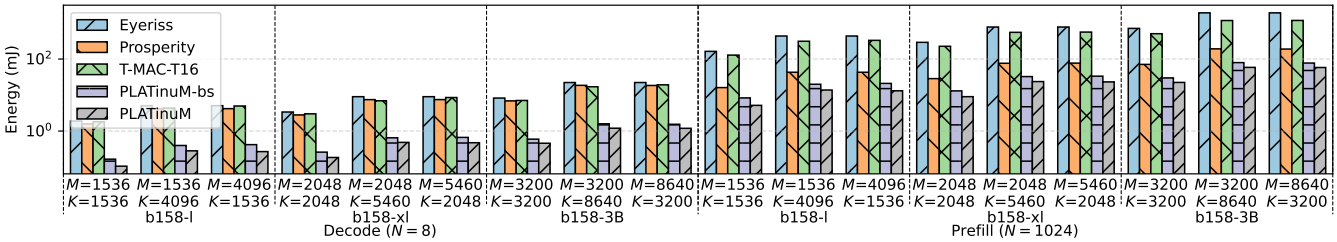

Fig. 9: Comparison of kernel energy across *Platinum*, CPU (T-MAC), SpikingEyeriss, and Prosperity.

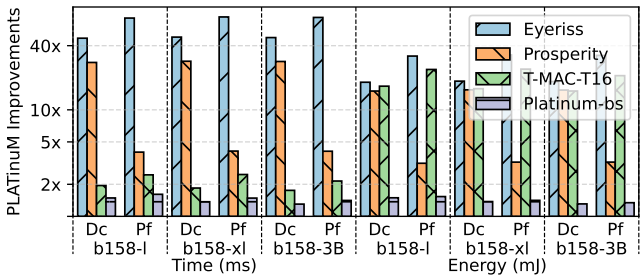

Fig. 10: *Platinum* improvements of performance and energy efficiency in Model-level.

This highlights the importance of minimizing off-chip memory access and accesses to weight buffers, showing that using compact weight representations is critical to improve energy efficiency. Additionally, the LUT buffer exhibits lower power usage compared to the weight buffer and DRAM, indicating that LUT introduces low power overhead in LUT-based computation paradigm.

C. Kernel-Level and model-level evaluations

We evaluate *Platinum* and baseline accelerators using representative kernel workloads extracted from BitNet-b1.58, covering both the prefill and decode stages. The results (Figure 8, Figure 9) demonstrate that *Platinum* outperforms state-of-the-art ASIC accelerators and a CPU-based design in terms of both raw performance and energy efficiency.

Using kernels from the b1.58-3B model configuration (Figure 10), *Platinum* achieves average speedups of 73.6×, 4.09×, and 2.15× for the prefill stage, and 47.6×, 28.4×, and 1.75× for the decode stage, over SpikingEyeriss, Prosperity, and 16-thread T-MAC, respectively. Three key factors contribute to this high performance over the baselines. First, by disaggregating "shortcut" path scheduling to offline, it not only benefits from low LUT construction cost, but also minimizes the hardware overhead of runtime scheduling, and thus more area is available for additional PEs. Second, *Platinum* achieves high PE utilization by duplicating adders to fully exploit LUT ports and by choosing $ncols = 8$ per PPE to guarantee utilization under low-N workloads. In contrast, baseline accelerators like Prosperity suffer from significant underutilization of PEs for decode workloads. Finally, *Platinum* incorporates optimizations specifically for ternary weights, delivering 1.3× and 1.4× speedups over its own bit-serial mode in decode and prefill stages, respectively.

Platinum also delivers strong energy efficiency, consuming 18.4×, 15.3×, 15.0×, and 1.31× less energy in the decode stage, and 32.4×, 3.23×, 20.9×, and 1.34× less in the prefill stage, compared to SpikingEyeriss, Prosperity, T-MAC, and *Platinum*-bs, respectively. While Prosperity demonstrates high energy efficiency in prefill, *Platinum* outperforms it by eliminating runtime scheduler overhead and using compact weight formats to reduce memory access and footprint. Additionally, high parallelism along the K dimension lowers DRAM access frequency of output data, further boosting energy efficiency.

These findings highlight that bit-serial execution on our accelerator already achieves strong performance and efficiency across general weight precisions. Furthermore, optimizations tailored for ternary weights provide an additional performance boost, underscoring the effectiveness of our accelerator for specific weight precision.

VI. Conclusion

In this work, we present *Platinum*, a lightweight and energy-efficient ASIC accelerator optimized for low-bit weight matrix multiplication in neural networks. *Platinum* employs LUT-based computation with adaptable, offline-generated construction paths to achieve high performance and energy efficiency within a compact hardware footprint. By switching paths and compressing weights offline, it achieves even better efficiency for specific weight precision, such as the ternary weights in BitNet-b1.58 models. Given its efficiency and versatility, *Platinum* holds promise for broader adoption in deploying LLM workloads on edge devices.

REFERENCES

[1] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," CoRR, vol. abs/2302.13971, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.13971

[2] H. Touvron and et al., "Llama 2: Open foundation and fine-tuned chat models," CoRR, vol. abs/2307.09288, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09288

[3] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," CoRR, vol. abs/1909.08053, 2019. [Online]. Available: http://arxiv.org/abs/1909.08053

[4] C. Guo, C. Zhang, J. Leng, Z. Liu, F. Yang, Y. Liu, M. Guo, and Y. Zhu, "ANT: exploiting adaptive numerical data type for low-bit deep neural network quantization," in 55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022. IEEE, 2022, pp. 1414–1433. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00095

[5] C. Guo, J. Tang, W. Hu, J. Leng, C. Zhang, F. Yang, Y. Liu, M. Guo, and Y. Zhu, "Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization," in Proceedings of the 50th Annual International Symposium on Computer Architecture, 2023, pp. 1–15.

[6] H. Peng, K. Wu, Y. Wei, G. Zhao, Y. Yang, Z. Liu, Y. Xiong, Z. Yang, B. Ni, J. Hu, R. Li, M. Zhang, C. Li, J. Ning, R. Wang, Z. Zhang, S. Liu, J. Chau, H. Hu, and P. Cheng, "FP8-LM: training FP8 large language models," CoRR, vol. abs/2310.18313, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.18313

[7] F. Cheng, C. Guo, C. Wei, J. Zhang, C. Zhou, E. Hanson, J. Zhang, X. Liu, H. Li, and Y. Chen, "Ecco: Improving memory bandwidth and capacity for llms via entropy-aware cache compression," in Proceedings of the 52nd Annual International Symposium on Computer Architecture, 2025, pp. 793–807.

[8] C. Guo, F. Cheng, Z. Du, J. Kiessling, J. Ku, S. Li, Z. Li, M. Ma, T. Molom-Ochir, B. Morris et al., "A survey: Collaborative hardware and software design in the era of large language models," IEEE Circuits and Systems Magazine, vol. 25, no. 1, pp. 35–57, 2025.

[9] S. Haoxuan, W. Chiyue, Y. Xiaoxuan, G. Cong, L. Hai, C. Yiran, Z. Mengyuan et al., "Neuromorphic computing in the era of large models," Artificial Intelligence Science and Engineering, vol. 1, no. 1, pp. 17–30, 2025.

[10] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, "AWQ: activation-aware weight quantization for LLM compression and acceleration," CoRR, vol. abs/2306.00978, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.00978

[11] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: accurate post-training quantization for generative pre-trained transformers," CoRR, vol. abs/2210.17323, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2210.17323

[12] H. Wang, S. Ma, L. Dong, S. Huang, H. Wang, L. Ma, F. Yang, R. Wang, Y. Wu, and F. Wei, "Bitnet: Scaling 1-bit transformers for large language models," CoRR, vol. abs/2310.11453, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.11453

[13] S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei, "The era of 1-bit llms: All large language models are in 1.58 bits," CoRR, vol. abs/2402.17764, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.17764

[14] J. Wei, S. Cao, T. Cao, L. Ma, L. Wang, Y. Zhang, and M. Yang, "T-MAC: CPU renaissance via table lookup for low-bit LLM deployment on edge," CoRR, vol. abs/2407.00088, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2407.00088

[15] Y. Chen, A. F. AbouElhamayed, X. Dai, Y. Wang, M. Andronic, G. A. Constantinides, and M. S. Abdelfattah, "Bitmod: Bit-serial mixture-of-datatype LLM acceleration," in IEEE International Symposium on High Performance Computer Architecture, HPCA 2025, Las Vegas, NV, USA, March 1-5, 2025. IEEE, 2025, pp. 1082–1097. [Online]. Available: https://doi.org/10.1109/HPCA61900.2025.00084

[16] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016, pp. 1–12.

[17] C. Guo, C. Wei, J. Tang, B. Duan, S. Han, H. Li, and Y. Chen, "Transitive array: An efficient gemm accelerator with result reuse," in Proceedings of the 52nd Annual International Symposium on Computer Architecture, 2025, pp. 990–1004.

[18] Y. Jeon, B. Park, S. J. Kwon, B. Kim, J. Yun, and D. Lee, "Biqgemm: matrix multiplication with lookup table for binary-coding-based quantized dnns," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020, C. Cuicchi, I. Qualters, and W. T. Kramer, Eds. IEEE/ACM, 2020, p. 95. [Online]. Available: https://doi.org/10.1109/SC41405.2020.00099

[19] Z. Mo, L. Wang, J. Wei, Z. Zeng, S. Cao, L. Ma, N. Jing, T. Cao, J. Xue, F. Yang, and M. Yang, "LUT tensor core: Lookup table enables efficient low-bit LLM inference acceleration," CoRR, vol. abs/2408.06003, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.06003

[20] G. Park, B. Park, M. Kim, S. Lee, J. Kim, B. Kwon, S. J. Kwon, B. Kim, Y. Lee, and D. Lee, "LUT-GEMM: quantized matrix multiplication based on luts for efficient inference in large-scale generative language models," in The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=gLARhFLE0F

[21] J. Wang, H. Zhou, T. Song, S. Cao, Y. Xia, T. Cao, J. Wei, S. Ma, H. Wang, and F. Wei, "Bitnet.cpp: Efficient edge inference for ternary llms," CoRR, vol. abs/2502.11880, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2502.11880

[22] S. Kim, J. Lee, and H.-J. Yoo, "Slim-llama: A 4.69mw large-language-model processor with binary/ternary weights for billion-parameter llama model," in 2025 IEEE International Solid-State Circuits Conference (ISSCC), vol. 68, 2025, pp. 421–423.

[23] C. Wei, B. Duan, C. Guo, J. Zhang, Q. Song, H. Li, and Y. Chen, "Phi: Leveraging pattern-based hierarchical sparsity for high-efficiency spiking neural networks," in Proceedings of the 52nd Annual International Symposium on Computer Architecture, 2025, pp. 930–943.

[24] C. Wei, C. Guo, F. Cheng, S. Li, H. F. Yang, H. H. Li, and Y. Chen, "Prosperity: Accelerating spiking neural networks via product sparsity," in 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2025, pp. 806–820.

[25] R. C. Prim, "Shortest connection networks and some generalizations," Bell System Technical Journal, vol. 36, no. 6, pp. 1389–1401, 1957.

[26] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," ACM Trans. Archit. Code Optim., vol. 14, no. 2, pp. 14:1–14:25, 2017. [Online]. Available: https://doi.org/10.1145/3085572

[27] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in 2016 IEEE International Solid-State Circuits Conference, ISSCC 2016, San Francisco, CA, USA, January 31 - February 4, 2016. IEEE, 2016, pp. 262–263. [Online]. Available: https://doi.org/10.1109/ISSCC.2016.7418007

[28] S. Narayanan, K. Taht, R. Balasubramonian, E. Giacomin, and P.-E. Gaillardon, "Spinalflow: an architecture and dataflow tailored for spiking neural networks," in Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture, ser. ISCA '20. IEEE Press, 2020, p. 349–362. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00038

[29] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. L. Jacob, "Dramsim3: A cycle-accurate, thermal-capable DRAM simulator," IEEE Comput. Archit. Lett., vol. 19, no. 2, pp. 110–113, 2020. [Online]. Available: https://doi.org/10.1109/LCA.2020.2973991