BitFly: A Low-Bit Mixed-Precision Acceleration Framework for Edge RISC-V Vector Processors

Zixuan Zeng[1,2], Chen Zhang[3✉], Zhe Liu[3], and Peng Li[1,2✉]

[1] Institute of Software, Chinese Academy of Sciences, Beijing, China, 100190
[2] University of Chinese Academy of Sciences
[3] Shanghai Jiao Tong University, Shanghai, China, 200240
[✉]Corresponding author, chenzhang.sjtu@sjtu.edu.cn, lipeng@iscas.ac.cn

Abstract. Low-bit quantization is increasingly important for edge large language model (LLM) inference, but reduced model storage does not automatically yield proportional runtime speedup on processors without native support for low-bit mixed-precision general matrix multiplication (LBMP-GEMM). This limitation is especially acute on RISC-V Vector Extension (RVV) processors, where low-bit operands still traverse higher-precision vector datapaths and incur substantial packing, sequencing, and partial-sum overheads. We present BitFly, an RVV-compatible hardware-software co-design. BitFly integrates a lane-local bit-plane mixed-precision unit (BMPU) built around a 1-bit primitive, a context-banked accumulation store, and a lightweight instruction interface with offline-searched, shape-specific execution configurations, enabling efficient LBMP-GEMM computation. Cycle-accurate evaluation shows up to a 35.7× prefill speedup in llama2.c over RVV baselines across W1A8, W2A8, and W4A8 on models ranging from 15M to 3B parameters. Relative to the synthesized baseline accelerator, BitFly improves peak throughput by 27.3× with 28.2% additional area, while increasing area and energy efficiency by 21.4× and 14.3×, respectively.

Keywords: Edge LLM Inference · Low-Bit Mixed-Precision GEMM · RISC-V Vector Processors.

1 Introduction

Large Language Models (LLMs) are emerging as an important workload for edge intelligence, where inference must simultaneously satisfy stringent latency, privacy, and power constraints [17, 31]. This trend is driving deployment from cloud-centric infrastructures toward resource-constrained embedded platforms [23, 1]. On RVV embedded processors, inference kernels are carried primarily through software-visible vector paths rather than a dedicated programmable matrix engine. Under such constraints, quantization is indispensable for the feasibility of deployment. However, reductions in model size do not, by themselves, translate into proportional improvements in runtime performance.

A more fundamental source of this gap lies in the fact that quantized low-bit mixed-precision general matrix multiplication (LBMP-GEMM) does not naturally map onto the organization of baseline RVV processors. Low-bit operands

continue to traverse higher-precision vector datapaths, and the nominal arithmetic advantage of quantization is substantially offset by packing, conversion, sequencing, and partial-sum management overheads [26]. The limitation is therefore not explained adequately by arithmetic density alone. Rather, it reflects a structural mismatch between the computational requirements of LBMP-GEMM and the baseline RVV substrate, which does not directly provide the operand organization, accumulation locality, or result-materialization behavior required by low-bit mixed-precision computation.

This mismatch is particularly restrictive in edge-oriented RVV deployment. Inference in this setting is typically latency-sensitive and low-batch under tight area and power budgets, while matrix multiplication continues to dominate execution time. At the same time, INT8 activations and low-bit weights differ substantially in packing granularity, reuse behavior, and accumulation requirements [14, 15]. Consequently, efficient support for LBMP-GEMM cannot be obtained simply by attaching a denser arithmetic primitive to an otherwise unchanged vector pipeline. An effective design must instead satisfy three conditions simultaneously: it must consume heterogeneous operands efficiently under lane-local, banked vector register file organization; it must retain intermediate reduction state close to compute rather than repeatedly materializing partial sums through the vector register file; and it must preserve compatibility with the RVV control model without introducing a heavyweight detached accelerator.

To address this architectural gap, we propose BitFly, an RVV-compatible hardware-software co-design methodology for LBMP-GEMM, which supports W1A8, W2A8, and W4A8 mixed-precision computation, where WpA8 denotes p-bit weights and 8-bit activations. Rather than treating low-bit support as an isolated arithmetic extension, BitFly organizes it as a native low-bit execution flow spanning a VRF-backed operand scratchpad, a lane-local bit-plane compute fabric, a context-banked accumulation store, and a deferred result-export path under a lightweight instruction protocol with offline-selected mappings. On the baseline processor, cycle-accurate evaluation shows that BitFly achieves geometric-mean kernel speedups of 77.8×, 54.5×, and 29.0× for W1A8, W2A8, and W4A8, respectively, and up to 35.7× framework-level prefill speedup in llama2.c [2]. Post-synthesis analysis shows a 27.3× peak low-bit compute-throughput increase at 28.2% additional area, indicating that efficient low-bit mixed-precision computation in RVV depends jointly on operand residency, accumulation locality, and execution control.

Our key contributions are summarized as follows. **(1)** We identify and formalize three RVV constraints for efficient LBMP-GEMM execution: lane-local and VRF-banked operand organization, long-lived accumulation state, and strict control-path integration. These constraints explain why low-bit quantization does not automatically translate into proportional runtime speedup on RVV processors. **(2)** We propose BitFly as an RVV-compatible co-design for LBMP-GEMM. BitFly enables efficient low-bit execution through a VRF-backed operand scratchpad, a lane-local bit-plane compute fabric, a context-banked accumulation store, and a deferred result-export datapath with a minimal

instruction interface. **(3)** We implement and evaluate BitFly across kernel-level, layer-level, and framework-level workloads as well as synthesis analysis. The results show that BitFly can substantially improve low-bit computational throughput and hardware efficiency with moderate hardware overhead.

2 Background

Low-Bit Quantized LLMs. Low-bit quantization is increasingly attractive for edge LLM deployment because it reduces both parameter storage and memory traffic [15]. As precision drops below 4 bits, model quality becomes more sensitive to outliers and value imbalance, motivating mixed-precision formulations that preserve critical values using higher precision while quantizing the dominant weights more aggressively [24, 29, 16]. Recent BitNet-style models and edge-oriented runtimes suggest that such sub-4-bit regimes, including binary-dominant weight formats, are evolving from primarily algorithmic explorations into plausible deployment targets [27, 28]. Compression-quality trade-offs alone do not determine realized performance. Practical speedup still depends on whether the processor can execute low-bit mixed-precision operations without repeatedly incurring unpacking, conversion, and auxiliary control overheads [26].

Microarchitecture for Low-Bit Mixed-Precision Inference. These execution-side constraints make RVV a useful target for study. RVV preserves a general-purpose vector programming model while exposing customization opportunities at both the ISA and dataflow levels. Prior RVV-oriented designs have explored packed or asymmetric arithmetic support, hybrid scheduling, and remapping low-bit operations onto wider datapaths [3, 25, 9]. More broadly, open RISC-V platforms such as Gemmini, Snitch, and Spatz illustrate a wider design space of customized compute backends [12, 30, 6]. Nevertheless, support for low-bit mixed-precision execution under conventional vector-register-file (VRF) organization and operand-layout constraints remains limited, especially when execution must remain compatible with RVV control logic.

3 Motivation

On RVV processors, low-bit compression does not automatically translate into proportional throughput improvement. For LBMP-GEMM, lowering weight precision reduces storage costs, but the speedup still depends on whether packed low-bit weights and INT8 activations can traverse without incurring substantial reshuffling or serialization overhead. The RVV accelerator is not organized around the accumulation, data-layout, and control requirements of LBMP-GEMM. Consequently, expressing such execution through standard vector instructions increases the instruction count, synchronization costs, and data movement overhead.

Table 1 makes this realization gap explicit through measurements collected from llama.cpp. The observed pattern is informative: although quantization yields substantial compression in model storage, the corresponding throughput improvement remains smaller than the compression ratio. This discrepancy indicates that software implementations of low-bit computation preserve much of

Table 1: An example throughput comparison in the llama.cpp framework [13].

Model	Quantization Method	Size (MiB)	Compression Ratio	Phase	Throughput (token/s)	Speedup	Acceleration Density
Qwen3 0.6B	BF16	1136.64	–	Prefill	932.78	–	–
				Decode	43.83	–	–
	IQ1_S	199.02	5.71×	Prefill	655.35	0.70×	0.12
				Decode	78.73	1.80×	0.32
	IQ2_XXS	217.55	5.22×	Prefill	634.78	0.68×	0.13
				Decode	72.89	1.66×	0.32
	IQ3_XXS	263.34	4.32×	Prefill	570.66	0.61×	0.14
				Decode	64.41	1.47×	0.34

the memory-footprint reduction, while converting only a limited portion of that reduction into realized runtime acceleration, owing to the repeated unpacking, reformatting, and accumulation of low-bit operands within higher-precision computation. To quantify the extent to which compression is translated into realized speedup, the following metric is introduced: $\eta = \frac{\text{Observed Speedup}}{\text{Compression Ratio}}$.

An ideal implementation would yield $\eta \to 1$, indicating that compression is converted almost fully into runtime improvement. However, Table 1 shows that η remains well below 1 across representative quantized settings. This observation indicates that the bottleneck lies in the execution substrate rather than the quantization quality. Although lower-bit weights reduce the number of bytes moved, a substantial fraction of the potential speedup is still lost to unpacking, control, and synchronization overheads. This shortfall motivates BitFly.

4 BitFly Framework

4.1 Microarchitecture Design

BitFly targets LBMP-GEMM on RVV accelerators. Its goal is to embed native low-bit execution into the RVV lane organization. BitFly keeps the RVV frontend and vector substrate, attaches one tightly coupled bit-plane mixed-precision unit (BMPU) per lane, and derives higher weight precision by reusing a native 1-bit primitive across bit planes. This section introduces the RVV constraints, the lane microarchitecture, and the accumulation and export methods.

RVV Constraints Efficient support for LBMP-GEMM on an RVV accelerator is governed by three constraints. First, **operand organization** is lane-local and VRF-banked. Each lane stores only a slice of a vector register, so operands with different precisions cannot be rearranged freely at execution time without additional movement, replication, or bank conflicts. Second, **accumulation state** is long-lived. LBMP-GEMM does not produce a final output from a single vector operation; instead, intermediate sums can be preserved across reduction steps and bit-plane composition steps, making repeated VRF round-trips expensive. Third, **control integration** must remain in the RVV issue/retire datapaths. Low-bit support must fit within the existing decode, issue, and retire logic.

These constraints define BitFly's design target. An effective RVV solution must consume heterogeneous operands with a low lane-local access cost, keep intermediate sums near computation until reduction completes, and expose execution progress through the standard RVV control path. BitFly addresses this

target with a lane-local bit-plane compute fabric, a VRF-backed operand scratchpad for staged operand residency, a context-banked accumulation store for partial sums, and a deferred result-export path for finalized outputs. Fig. 1 shows the physical-slot organization rather than a fixed activation/weight role assignment. Sec. 4.2 explains software tiling, grouping, and instruction sequencing.

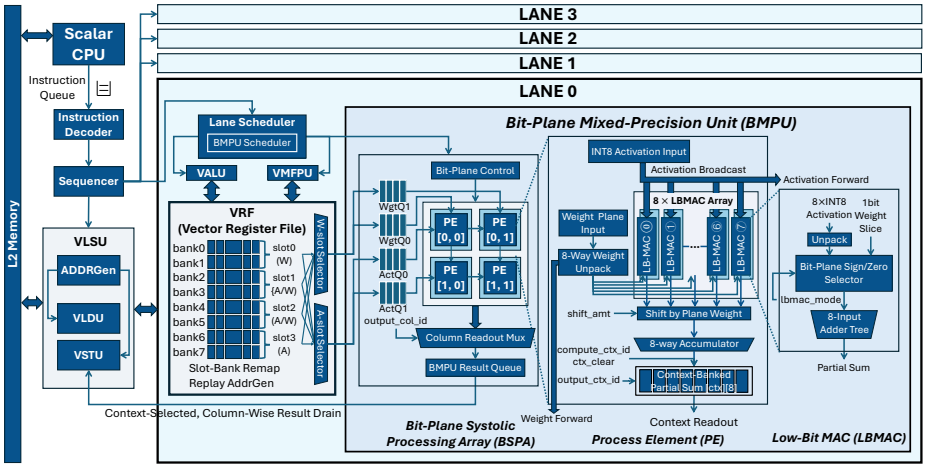

Fig. 1: BitFly extends the RVV lane backend with a lane-local BMPU, a VRF-backed resident-operand scratchpad, and a context-banked accumulation store; finalized outputs are drained through a deferred export path.

Lane Microarchitecture The lane-local BMPU is organized hierarchically from low-bit MAC units (LBMACs), to processing elements (PEs), to a bit-plane systolic processing array (BSPA). This hierarchy is described bottom-up.

LBMAC. LBMAC is the primitive compute unit. It consumes one 8-element activation group and one 8×1 weight slice and produces the dot product for one weight column of one bit plane. For each activation element, the weight bit selects pass, negate, or suppress under the active mode, after which an adder tree reduces the eight selected values. LBMAC therefore realizes the per-plane mixed-precision operation directly, without introducing a general low-bit multiplier.

PE. PE is the first aggregation level above the 1-bit primitive. Each PE instantiates eight LBMACs and processes one complete 8×8 weight tile per cycle. The activation group is broadcast to all LBMACs, while column-wise slicing delivers one 8×1 weight slice to each unit. As a result, one PE performs a tile-level GEMV between one 1×8 INT8 activation group and one 8×8 1-bit weight tile and produces an 8-element output fragment. A local plane scheduler derives the active LBMAC mode and shift alignment from the precision mode and wavefront position, enabling W1A8, W2A8, and W4A8 to share the same PE structure. The output fragment is accumulated into context-indexed local registers, making the PE both the first output-stationary stage and the first level at which multi-plane partial results are merged across the accumulation dimension.

BSPA. BSPA is the lane-level spatial fabric that composes PE-local updates into GEMM progress. Each lane has a 2×2 BSPA served by four resident

operand slots in the VRF-backed scratch and by two activation queues and two weight queues. These four slots form a shared pool rather than a fixed two-activation/two-weight partition: for each tile group, the sequencer assigns them dynamically between activation and weight windows according to the current (g_m, g_n) reuse pattern, tile shape, and precision mode. Physically, slot s maps to a fixed VRF bank pair $(2s, 2s+1)$; the dynamic assignment changes the payload carried by each slot rather than the underlying bank topology. Together, LBMACs, PEs, and the BSPA form one tightly coupled BMPU per lane. BitFly reuses the existing frontend and most lane infrastructure, including the VRF, lane pipeline, dispatcher/sequencer, and VLSU, while specializing only the lane backend for low-bit execution. BMP-specific modifications are therefore confined to operand delivery, local accumulation, and result export. W1A8, W2A8, and W4A8 are all derived by reusing the same lane hierarchy under different plane schedules, rather than by introducing separate arithmetic backends. At the processor level, activations are partitioned across lanes along the row dimension, while the same weight tile is broadcast to all lanes. Under the default four-lane configuration, this yields a processor-wide minimum block of 8×16, formed by four lane-local 2×16 blocks sharing one weight tile.

Context-Banked Accumulation and Deferred Result Export BitFly retains partial sums in context-banked local state rather than materializing them through the VRF after each update. Under an execution configuration, one software tile is decomposed into lane-local micro-blocks indexed by $(m_{\mathrm{blk}}, n_{\mathrm{blk}})$, and this micro-block index is mapped to a BSPA context id. The selected context id determines the PE-local accumulation bank updated by the compute wavefront, so the same bank is reused across all later iterations along the reduction dimension and across all required bit-plane composition steps for that micro-block. A bank is cleared only when a new reduction epoch begins. The context-banked accumulation store is an effective accumulation unit and is the key mechanism that allows BitFly to keep reduction state on chip.

Result export is handled by a separate context/column readout datapath. Once accumulation for a context completes, the BMPU supplies an output-context id and an output-column id to the BSPA. The BSPA then reads the selected column for that context while returning all BSPA rows in parallel. The returned values are packed into 64-bit lane words and pushed into the result queue for deferred store-out. This context-banked accumulation combined with context/column-selective drain is architecturally distinct from a conventional RVV realization: accumulation placement is governed by local reduction state, whereas result materialization follows a later deterministic drain order. This decoupling removes repeated VRF read-modify-write traffic for intermediate sums and keeps the lane backend compact despite wider accumulated outputs.

4.2 LBMP-GEMM Mapping and Execution

The execution of LBMP-GEMM on BitFly is determined in two stages. Offline, BitFly selects an efficient tile configuration and operand reuse group configuration subject to BMPU alignment, buffer-capacity, and bank-capacity constraints.

At runtime, the kernel uses the selected configuration to traverse tile groups, accumulate along the reduction dimension, and export finalized outputs.

BitFly maps LBMP-GEMM at three levels. First, the matrices are partitioned into software tiles of size (m_t, n_t, k_t), with the corresponding tile counts (m_b, n_b, k_b). Second, neighboring output tiles are grouped by (g_m, g_n): g_m reuses one weight tile across adjacent M-direction output tiles, whereas g_n reuses one activation tile across adjacent N-direction output tiles. Third, each software tile is executed as a sequence of processor-wide micro-block updates. For a full tile, (B_m, B_n, B_k) denote the update counts along the M, N, and K dimensions. Since only the boundary tile in the K direction may reduce the required reduction depth, the full-tile count B_k is distinguished from the boundary-tile count B_k^{eff}, which is determined by the aligned effective depth k_{eff} of the final K tile.

Offline search first determines whether a tile/group configuration is legal. Tile sizes must match PE granularity, so m_t is searched in steps of 8 and n_t in steps of 16. Once (g_m, g_n) is chosen, the grouped outputs retained on chip during grouped reuse must fit within the local accumulation budget:

$$m_t n_t g \cdot 16\,\text{bit} \le C_{buf}, \qquad g = g_m g_n. \tag{1}$$

The left-hand side is the 16-bit storage space required to hold one grouped $(g_m \times g_n)$ output region before the group is drained. Candidates that underfill this budget remain legal, they are ranked later by computational latency. In addition, k_t must satisfy the bank capacity constraints:

$$m_t k_t \cdot 8\,\text{bit} \le N_{row} \cdot W_{bank} \cdot D_{bank}, \qquad n_t k_t \cdot p_{weight} \le N_{col} \cdot W_{bank} \cdot D_{bank}. \tag{2}$$

Here, N_{row} and N_{col} denote the lane-local BSPA row and column counts, W_{bank} and D_{bank} denote the bank width and depth. Meanwhile, C_{buf} is the grouped-output buffer budget for one mapped tile group. All legal grouped mappings also satisfy $g = g_m g_n \le 8$. For full tiles in the definition, $k_{\text{exec}} = k_t$, only the boundary K tile replaces k_t with the aligned effective depth k_{eff} at runtime.

Legal candidates are ranked by an event-driven timing model that matches the computation method: one load channel, one compute channel, and a four-slot resident-operand schedule that overlaps grouped reuse with operand staging. For a full-tile candidate, the numbers of processor-wide updates per software tile are

$$B_m = \left\lceil \frac{m_t}{N_{\text{lane}} \cdot N_{row}} \right\rceil, \qquad B_n = \left\lceil \frac{n_t}{N_{col} \cdot 8} \right\rceil, \qquad B_k = \left\lceil \frac{k_t}{8} \right\rceil, \tag{3}$$

For one processor-wide update, the BSPA wavefront latency and the resulting compute latency of one software tile are given by

$$T_{BSPA} = (N_{row} - 1) + B_k\, p_{weight} + N_{col}, \qquad T_{comp} = B_m B_n\, T_{BSPA}, \tag{4}$$

where $p_{weight} \in \{1, 2, 4\}$ denotes the number of weight planes in the different mixed-precision modes. Memory latency is jointly determined by 8-bit activation transfer and p_{weight}-bit weight transfer under the precision mode. BitFly then evaluates how much of this transfer time can be hidden both within one grouped-reuse epoch and across consecutive groups, and selects configurations by latency.

Runtime execution is orchestrated by four instructions. They encode the selected tile/group configuration, stage operands, trigger computation, and drain

Table 2: Semantics of the BitFly custom instructions.

Instr.	Assembly form	Function
BMPCFG	`bmpcfg prec, k_exec, m_t, n_t, g_m, g_n`	Programs the selected execution configuration.
BMPLE	`bmple imm(rs1), a|w`	Loads one activation or weight tile into a resident slot.
BMPMM	`bmpmm`	Launches one resident activation-weight tile pair.
BMPSE	`bmpse imm(rs1)`	Drains one completed context/column pair.

results. Table 2 summarizes the instruction definitions. Given a configuration $(m_t, n_t, k_t, g_m, g_n, \text{prec})$, execution follows the three-stage schedule.

Stage I: Group Traversal and Execution Plan Generation. We first traverse the grouped software-tile coordinates implied by the tile counts (m_b, n_b, k_b). For each group origin, the runtime derives the boundary-clipped group extents (mg_len and ng_len). The plan module then specializes the configuration to this concrete group under the four-slot resident budget: it assigns resident slots to activation and weight windows, derives the corresponding window counts, selects the serpentine window walk (row_snake) when beneficial, and chooses the preferred reuse direction (reuse_a). Stage I therefore produces one group-local resident-window traversal strategy. Fig. 2.

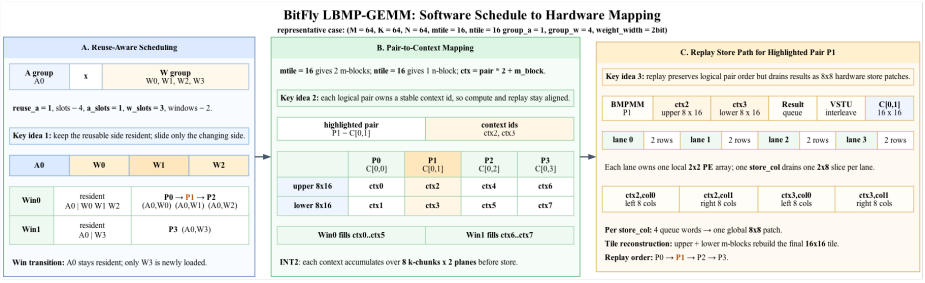

Fig. 2: Runtime execution schedule of LBMP-GEMM on BitFly.

Stage II: Accumulation Across Grouped Tiles. Stage II follows the Stage I schedule while advancing along the K dimension. For each K tile, the runtime forms the depth k_{exec}, equal to k_t for full tiles and to the aligned k_{eff} for the boundary tile. It then walks the activation and weight windows, emits `BMPCFG` for the current window, reloads only the operand whose window has changed via `BMPLE`, and issues `BMPMM` over resident activation-weight pairs in the plan-selected order. The plan is selected once for the current group and then reused for every K tile. This is the key effect of the plan in Fig. 2: it decides which operand remains resident across neighboring pair launches, thereby extending reuse in the VRF-backed operand scratchpad while partial sums stay live in the context-banked accumulation store. For all non-boundary tiles, the corresponding accumulation counts are (B_m, B_n, B_k); only the final K tile may be shorter, in which case its executed update count becomes $B_k^{\text{eff}} = \left\lceil \frac{k_{\text{eff}}}{8} \right\rceil$.

Let $g = (m_g, n_g)$ denote the origin of the current output-tile group, and let $o = (a_i, w_i)$ denote the offset of one output tile within that group. Thus, (g, o) identifies the output tile at $(m_g + a_i,\ n_g + w_i)$. The contribution from the k_i-th

tile along the K dimension to this output tile is

$$\Delta \mathbf{C}_{g,o}^{(k_i)} = \sum_{\mu=0}^{B_m-1} \sum_{\nu=0}^{B_n-1} \Delta \mathbf{C}_{g,o,\mu,\nu}^{(k_i)}, \qquad \mathbf{C}_{g,o}^{(k_i+1)} = \mathbf{C}_{g,o}^{(k_i)} + \Delta \mathbf{C}_{g,o}^{(k_i)}. \tag{5}$$

Here, k_i indexes the current tile along the K dimension, $\mathbf{C}_{g,o}$ denotes the accumulated state of output tile (g, o), and $\Delta \mathbf{C}_{g,o,\mu,\nu}^{(k_i)}$ denotes the processor-wide update generated at local micro-block coordinate (μ, ν) for that K tile.

For p_{weight}-bit weights, BitFly decomposes each weight tile into bit planes and reuses the same 1-bit datapath across planes. Let $\mathbf{W}^{(b)}$ denote the b-th bit plane, and let α_b denote the coefficient assigned to the plane b:

$$\mathbf{W} = \sum_{b=0}^{p_{weight}-1} \alpha_b \, \mathbf{W}^{(b)}, \qquad \Delta \mathbf{C}_{g,o,\mu,\nu}^{(k_i)} = \sum_{b=0}^{p_{weight}-1} \alpha_b \, \Delta \mathbf{C}_{g,o,\mu,\nu,b}^{(k_i)}. \tag{6}$$

Here, $\Delta \mathbf{C}_{g,o,\mu,\nu,b}^{(k_i)}$ is the 1-bit partial result from bit plane b for local micro-block (μ, ν) of output tile (g, o). For unsigned magnitude composition, $\alpha_b = 2^b$; for signed or mixed modes, the active LBMAC mode supplies the effective sign or zero behavior during PE-level shift and merge. These equations describe Stage II: hardware accumulates across (μ, ν, b) within one `BMPMM` stream, while software advances k_i and retains grouped partial sums until the tile group completes.

Stage III: Deterministic Export. After the last K tile has been accumulated, Stage III exports finalized outputs through `BMPSE`. It follows the same window sequence and pair order fixed in Stage I, reissues `BMPCFG` with the final aligned K setting, derives the destination address $cptr = \mathrm{addr}_C(\cdot)$ for each completed output pair, and replaces `BMPMM` with `BMPSE`. Stage III is therefore the export phase of the same group-local schedule: finalized outputs are drained in the Stage I order, and the store path preserves that issue order.

4.3 Roofline Analysis

We further project legal configurations under an architecture-aware roofline that fixes the precision-specific BMPU compute roof and varies only tile/group configurations. Each point in Fig. 3 corresponds to one legal candidate for the representative shape, evaluated under the buffer constraints above and the same event-driven load/compute/drain model used in offline search. In this space, (m_t, n_t) control how much computation is amortized over one staged activation/weight residency interval, while (g_m, g_n) control how far that staged residency can be reused across neighboring output tiles before drain. Varying (g_m, g_n) therefore changes reuse, resident-slot behavior, and exposed drain overhead.

Points farther to the right achieve higher reuse, whereas points higher on the plot convert more of the fixed compute headroom into sustained throughput. In particular, larger (m_t, n_t) increase the amount of work extracted from each staged operand fill, whereas larger (g_m, g_n) further improve operational intensity when the accumulation store can retain grouped partial sums long enough to suppress intermediate drains. For the shape $(128, 2048, 8192)$, the best configurations are $(m_t, n_t, k_t, g_m, g_n) = (8, 128, 64, 1, 1)$ for W1A8 and $(8, 16, 128, 2, 4)$

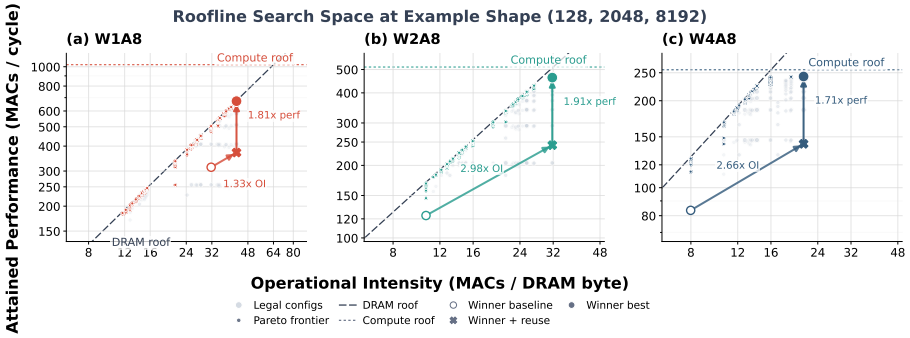

Fig. 3: Architecture-aware roofline. Each subplot shows the legal design space for the representative shape (128,2048,8192); filled points mark the Pareto frontier.

for both W2A8 and W4A8. The following evaluation uses the best legal configurations searched by this same execution model and constraint set.

5 Evaluation

We use Ara as the baseline platform, a 64-bit RISC-V vector accelerator compatible with RVV 1.0 [5, 21, 19]. All experiments use the default Ara configuration with N_{LANE}=4 and VLEN=4096. Cycle-level results are obtained from Verilator v5.012 and QuestaSim v2024.1, with identical memory-system assumptions for BitFly and the RVV baseline. Both implementations use the same quantized model format and arithmetic semantics. The RVV baseline executes low-bit linear layers by vectorized unpack followed by W8A8 GEMM, whereas BitFly executes the same layers directly on its native low-bit datapath.

We evaluate BitFly at three levels. The kernel-level suite covers 18 GEMM shapes sampled from square, small-M, and projection-dominated regimes. The layer-level suite covers the seven dominant projections in one transformer block, across ten edge-oriented quantized LLM workloads spanning the Gemma, SmolLM2, Qwen2.5, Replit-Code, TinyLlama, OPT, and StableLM families. We sample block 0 because these transformer blocks repeat the same projection shapes within each model. At the framework level, we integrate BitFly into llama2.c and evaluate W1A8, W2A8, and W4A8 prefill execution on TinyStories-15M, TinyStories-42M, TinyStories-110M, Llama-3.2-1B, and Llama-3.2-3B with sequence lengths $L \in \{32, 64, 128, 256\}$ [2]. For each quantized linear layer, BitFly uses a pre-searched offline configuration $(m_t, n_t, k_t, g_m, g_n)$. The search is performed once per layer shape offline and cached as a configuration table; runtime execution only performs configuration lookup. The search is therefore excluded from the reported runtime. We report geometric means for kernel speedups across heterogeneous shape instances and arithmetic means for the layer-level and framework-level suites, where each workload instance is treated as an equally weighted evaluation point.

5.1 Performance Evaluation

Kernel-Level. For all three precision modes, the RVV reference is the post-unpack W8A8 GEMM on the same (M, N, K) shape. This is because the eval-

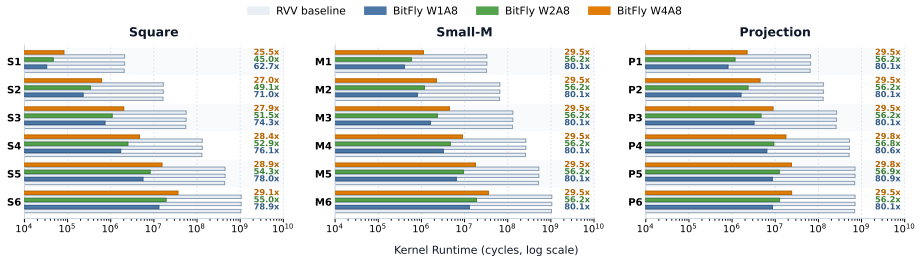

Fig. 4: Kernel-level speedup of BitFly over the RVV baseline on 18 representative GEMM shapes spanning square, small-M, and projection-dominated regimes. Speedup is reported as RVV kernel cycles over BitFly kernel cycles.

uated RVV baseline supports integer execution only down to 8-bit granularity, so W1A8, W2A8, and W4A8 must first be unpacked into 8-bit operands before GEMM execution. Kernel cycles include the compute body together with in-kernel load/store, address generation, and loop control. BitFly achieves geometric-mean speedups of 77.8×, 54.5×, and 29.0× for W1A8, W2A8, and W4A8, respectively. Square matrices are the least favorable cases, with speedups of 62.7×–78.9×, 45.0×–55.0×, and 25.5×–29.1×, while projection-dominated and small-M shapes are slightly higher on average.

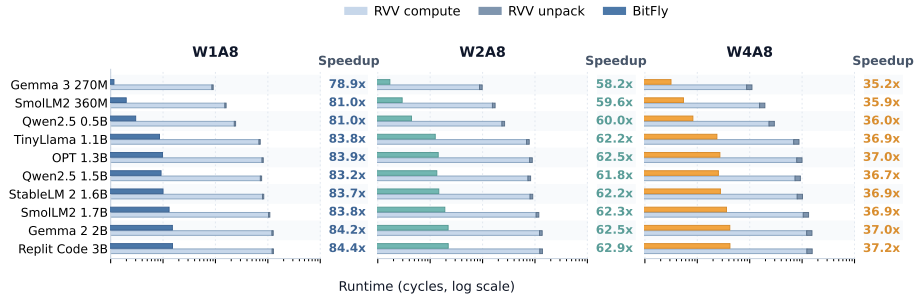

Fig. 5: Layer-level evaluation across ten LLM workloads. Each point aggregates the seven dominant linear layers from one transformer block for one model and one precision; the RVV baseline performs vectorized unpack followed by W8A8 GEMM, whereas BitFly executes the same layers directly in low precision.

Layer-Level. Unlike the kernel-level result, this comparison measures layer-level execution and includes the RVV unpack overhead that precedes the W8A8 GEMM. This difference in accounting explains why the gains at the layer-level can exceed the gains at the kernel-level. Using arithmetic means across the ten workloads, BitFly reduces the layer runtime by 82.8×, 61.4×, and 36.6× for W1A8, W2A8, and W4A8, respectively. The performance among different workloads remains narrow, at 78.8× – 84.4×, 58.1× – 62.8×, and 35.2× – 37.2×, consistent with a stable advantage across different LLM layer shapes.

Framework-Level. The RVV and BitFly variants execute the same computation graph and differ only in the backend used for quantized linear layers. In all configurations, the speedup remains ordered as W1A8 > W2A8 > W4A8. The maximum speedup reaches 35.7× on the 3B model at L=32, and the arithmetic

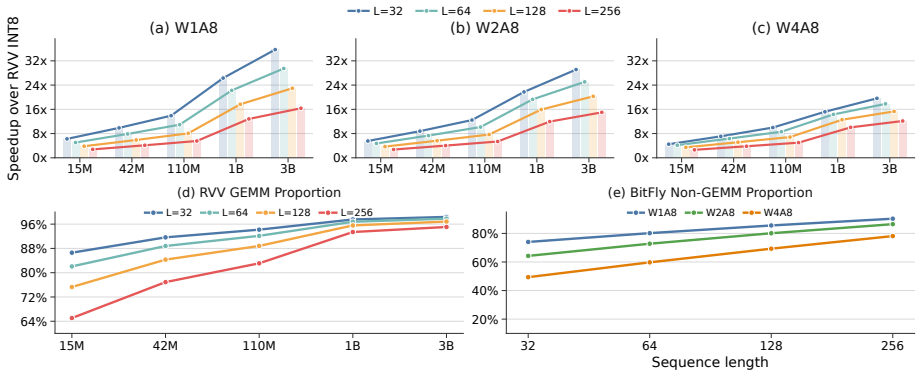

Fig. 6: Prefill speedup of llama2.c over the baseline [2]. The lower-left panel reports the GEMM proportion of total RVV prefill cycles. The lower-right panel reports the non-GEMM proportion of total BitFly prefill cycles.

means over all model-size and sequence-length pairs are 13.4×, 11.8×, and 9.2× for W1A8, W2A8, and W4A8, respectively. The lower panels explain the scaling trends: the GEMM proportion of RVV prefill cycles increases with model size, whereas the non-GEMM proportion of BitFly prefill cycles increases with sequence length. Together, these trends are consistent with BitFly delivering larger gains when GEMM dominates the workload and smaller prefill gains as non-GEMM operators absorb a larger fraction of the critical path.

5.2 Synthesis Results

Table 3: Comparison with prior RISC-V AI accelerators.

Metric	BitFly (Ours)	Ibex†	MixGEMM-v2	XPULPNN	Marcellus	SPEED	DORY	DARKSIDE	Dustin	Yun	Vega
		[8, 3]	[9]	[10]	[7]	[25]	[4]	[11]	[18]	[20]	[22]
Tech. (nm)	28 (22)	7 (22)	22	22	22	28 (22)	65 (22)	65 (22)	65 (22)	65 (22)	22
Arch. (-)	RV64+ Decoupled	RV32	RV64	8×RV32	RV32+ Decoupled	RV64+ Decoupled	8×RV32	8×RV32	16×RV32	RV64+ Decoupled	10×RV32
Freq. (MHz)	851 (1083)	14.75 (79.61)	1200	400	420	1050 (1337)	260 (767)	290 (855.5)	205 (604.75)	280 (826)	450
Area (mm^2)	1.13 (0.69)	0.038 (0.375)	0.03	0.086	0.46	1.20 (0.740)	-	12.00 (1.38)	10.00 (1.15)	6.00 (0.69)	12.00
Peak Perf. (GOPS)	**1742.84 (2217.99)**	0.85 (0.27)	29.0	47.9	569	737.9 (937.1)	4.2 (12.4)	65 (172.3)	58 (171.1)	22.9 (67.5)	15.6
Area Eff. (GOPS/mm^2)	**1542.33 (3214.47)**	22.37 (0.72)	1580	557	1236	614.6 (1266.4)	-	5.4 (124.5)	5.8 (149)	3.8 (97.5)	1.3
Energy Eff. (GOPS/W)	**3125.80 (3977.99)**	1470 (468.15)	1335	1100	5370	1383.4 (1757)	15.9 (47)	835 (2463.25)	1152 (3398)	100.5 (296.5)	614
Benchmark	LLM & GEMM	CNN	CNN	CNN	CNN	CNN	CNN	CNN	CNN	GEMM	CNN

Note: All metrics are reported at their original technology. Values in parentheses project non-22 nm designs to the 22 nm using standard scaling laws: area $\propto L^2$, frequency $\propto 1/L$, and power unchanged due to non-scaled supply voltage (V_{dd}).
† Ibex extended with specialized MAC FUs for mixed-precision workloads.

BitFly and Ara were synthesized in Synopsys Design Compiler (T-2022.03-SP2) using the TSMC 28 nm PDK [5, 21, 19]. Relative to Ara, BitFly increases peak throughput by 27.3× with 28.2% additional area. This translates into 21.4× higher area efficiency and 14.3× higher energy efficiency. As summarized in Table 3, under 22 nm normalization BitFly achieves 2.22 TOPS peak throughput, 3214.47 GOPS/mm^2 area efficiency, and 3977.99 GOPS/W energy efficiency. Among the other state-of-the-art accelerators, BitFly provides the highest peak

throughput and area efficiency while remaining competitive in energy efficiency, demonstrating its hardware efficiency for edge LLM inference.

6 Conclusion

BitFly shows that the central challenge of LBMP-GEMM on RVV is not merely how to execute fewer-bit arithmetic, but how to organize operand delivery, reduction state, and execution control so that low-bit computation remains native throughout the full execution flow. We summarize three takeaways.

Takeaway 1: Low-bit mixed-precision execution on RVV is constrained primarily by asymmetric operand delivery rather than by arithmetic density alone. Flexible on-chip tiling and grouped reuse are necessary to amortize operand staging overhead, reduce data latency, and mitigate pipeline stalls.

Takeaway 2: Partial-sum residency is an essential architectural requirement for LBMP-GEMM. By retaining intermediate results in a programmable context-banked accumulation store across software execution rounds, BitFly absorbs the long-lived and irregular reduction workload of low-bit mixed-precision computation and decouples accumulation from final result export.

Takeaway 3: Practical RVV integration should preserve the original RVV control pipeline while reorganizing the VRF into a VRF-backed operand scratchpad. Window-based scheduling and serpentine traversal then realize low-overhead, shape-adaptive operand reuse without introducing a detached execution path.

Statement Artificial intelligence tools, if used, were used only for language organization and figure/table polishing, and did not participate in the generation of the core ideas or technical content.

References

1. Abadade, Y., Temouden, A., Bamoumen, H., Benamar, N., Chtouki, Y., Hafid, A.S.: A comprehensive survey on tinyml. IEEE Access **11**, 96892–96922 (2023). https://doi.org/10.1109/ACCESS.2023.3294111
2. Andrej Karpathy: Karpathy/llama2.c: Inference Llama 2 in one file of pure C. GitHub repository (2025), `https://github.com/karpathy/llama2.c`
3. Armeniakos, G., Maras, A., Xydis, S., Soudris, D.: Mixed-precision Neural Networks on RISC-V Cores: ISA extensions for Multi-Pumped Soft SIMD Operations (Aug 2024). https://doi.org/10.1145/3676536.3676840
4. Burrello, A., Garofalo, A., Bruschi, N., Tagliavini, G., Rossi, D., Conti, F.: Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. IEEE Transactions on Computers **70**(8), 1253–1268 (2021). https://doi.org/10.1109/TC.2021.3066883
5. Cavalcante, M., Schuiki, F., Zaruba, F., Schaffner, M., Benini, L.: Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **28**(2), 530–543 (Feb 2020). https://doi.org/10.1109/TVLSI.2019.2950087
6. Cavalcante, M., Wüthrich, D., Perotti, M., Riedel, S., Benini, L.: Spatz: A Compact Vector Processing Unit for High-Performance and Energy-Efficient Shared-L1 Clusters. In: Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design. pp. 1–9 (Oct 2022). https://doi.org/10.1145/3508352.3549367

7. Conti, F., Paulin, G., Garofalo, A., Rossi, D., Di Mauro, A., Rutishauser, G., Ottavi, G., Eggiman, M., Okuhara, H., Benini, L.: Marsellus: A heterogeneous risc-v ai-iot end-node soc with 2–8 b dnn acceleration and 30%-boost adaptive body biasing. IEEE Journal of Solid-State Circuits **59**(1), 128–142 (2024). https://doi.org/10.1109/JSSC.2023.3318301
8. Davide Schiavone, P., Conti, F., Rossi, D., Gautschi, M., Pullini, A., Flamand, E., Benini, L.: Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In: 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS). pp. 1–8 (2017). https://doi.org/10.1109/PATMOS.2017.8106976
9. Fornt, J., Reggiani, E., Fontova-Musté, P., Rodas, N., Pappalardo, A., Unsal, O.S., Cristal Kestelman, A., Altet, J., Moll, F., Abella, J.: Mix-GEMM: Extending RISC-V CPUs for Energy-Efficient Mixed-Precision DNN Inference Using Binary Segmentation. IEEE Transactions on Computers **74**(2), 582–596 (Feb 2025). https://doi.org/10.1109/TC.2024.3500369
10. Garofalo, A., Tagliavini, G., Conti, F., Benini, L., Rossi, D.: Xpulpnn: Enabling energy efficient and flexible inference of quantized neural networks on risc-v based iot end nodes. IEEE Transactions on Emerging Topics in Computing **9**(3), 1489–1505 (2021). https://doi.org/10.1109/TETC.2021.3072337
11. Garofalo, A., Tortorella, Y., Perotti, M., Valente, L., Nadalini, A., Benini, L., Rossi, D., Conti, F.: DARKSIDE: A Heterogeneous RISC-V Compute Cluster for Extreme-Edge On-Chip DNN Inference and Training. IEEE Open Journal of the Solid-State Circuits Society **2**, 231–243 (2022). https://doi.org/10.1109/OJSSCS.2022.3210082
12. Genc, H., Kim, S., Amid, A., Haj-Ali, A., Iyer, V., Prakash, P., Zhao, J., Grubb, D., Liew, H., Mao, H., Ou, A., Schmidt, C., Steffl, S., Wright, J., Stoica, I., Ragan-Kelley, J., Asanovic, K., Nikolic, B., Shao, Y.S.: Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration (Jul 2021). https://doi.org/10.48550/arXiv.1911.09925
13. ggml-org: ggml-org/llama.cpp. GitHub repository (Jul 2025), `https://github.com/ggml-org/llama.cpp`
14. Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M.W., Keutzer, K.: A survey of quantization methods for efficient neural network inference (2021), `https://arxiv.org/abs/2103.13630`
15. Gong, R., Ding, Y., Wang, Z., Lv, C., Zheng, X., Du, J., Qin, H., Guo, J., Magno, M., Liu, X.: A Survey of Low-bit Large Language Models: Basics, Systems, and Algorithms (Sep 2024). https://doi.org/10.48550/arXiv.2409.16694
16. Ma, L., Sun, M., Shen, Z.: FBI-LLM: Scaling Up Fully Binarized LLMs from Scratch via Autoregressive Distillation (Jul 2024). https://doi.org/10.48550/arXiv.2407.07093
17. Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., Gao, J.: Large language models: A survey (2025), `https://arxiv.org/abs/2402.06196`
18. Ottavi, G., Garofalo, A., Tagliavini, G., Conti, F., Mauro, A.D., Benini, L., Rossi, D.: Dustin: A 16-Cores Parallel Ultra-Low-Power Cluster With 2b-to-32b Fully Flexible Bit-Precision and Vector Lockstep Execution Mode. IEEE Transactions on Circuits and Systems I: Regular Papers **70**(6), 2450–2463 (Jun 2023). https://doi.org/10.1109/TCSI.2023.3254810
19. Perotti, M., Cavalcante, M., Andri, R., Cavigelli, L., Benini, L.: Ara2: Exploring Single- and Multi-Core Vector Processing with an Efficient RVV 1.0 Compliant

Open-Source Processor. IEEE Transactions on Computers **73**(7), 1822–1836 (Jul 2024). https://doi.org/10.1109/TC.2024.3388896
20. Perotti, M., Cavalcante, M., Ottaviano, A., Liu, J., Benini, L.: Yun: An Open-Source, 64-Bit RISC-V-Based Vector Processor With Multi-Precision Integer and Floating-Point Support in 65-nm CMOS. IEEE Transactions on Circuits and Systems II: Express Briefs **70**(10), 3732–3736 (Oct 2023). https://doi.org/10.1109/TCSII.2023.3292579
21. Perotti, M., Cavalcante, M., Wistoff, N., Andri, R., Cavigelli, L., Benini, L.: A "new ara" for vector computing: An open source highly efficient risc-v v 1.0 vector processor design. In: 2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP). pp. 43–51 (2022). https://doi.org/10.1109/ASAP54787.2022.00017
22. Rossi, D., Conti, F., Eggiman, M., Mauro, A.D., Tagliavini, G., Mach, S., Guermandi, M., Pullini, A., Loi, I., Chen, J., Flamand, E., Benini, L.: Vega: A ten-core soc for iot endnodes with dnn acceleration and cognitive wake-up from mram-based state-retentive sleep mode. IEEE Journal of Solid-State Circuits **57**(1), 127–139 (2022). https://doi.org/10.1109/JSSC.2021.3114881
23. Sai, S., Prasad, M., Dashore, G., Chamola, V., Sikdar, B.: On-device generative ai: The need, architectures, and challenges. IEEE Consumer Electronics Magazine **14**(4), 21–32 (2025). https://doi.org/10.1109/MCE.2024.3518761
24. Shang, Y., Yuan, Z., Wu, Q., Dong, Z.: Pb-llm: Partially binarized large language models (2023), `https://arxiv.org/abs/2310.00034`
25. Wang, C., Fang, C., Wu, X., Wang, Z., Lin, J.: SPEED: A Scalable RISC-V Vector Processor Enabling Efficient Multi-Precision DNN Inference. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **33**(1), 207–220 (Jan 2025). https://doi.org/10.1109/TVLSI.2024.3466224
26. Wang, H., Ma, S., Dong, L., Huang, S., Wang, H., Ma, L., Yang, F., Wang, R., Wu, Y., Wei, F.: Bitnet: Scaling 1-bit transformers for large language models (2023), `https://arxiv.org/abs/2310.11453`
27. Wang, H., Ma, S., Wei, F.: BitNet v2: Native 4-bit Activations with Hadamard Transformation for 1-bit LLMs (Jun 2025). https://doi.org/10.48550/arXiv.2504.18415
28. Wang, J., Zhou, H., Song, T., Cao, S., Xia, Y., Cao, T., Wei, J., Ma, S., Wang, H., Wei, F.: Bitnet.cpp: Efficient Edge Inference for Ternary LLMs (Feb 2025). https://doi.org/10.48550/arXiv.2502.11880
29. Xu, Y., Han, X., Yang, Z., Wang, S., Zhu, Q., Liu, Z., Liu, W., Che, W.: OneBit: Towards Extremely Low-bit Large Language Models (May 2024). https://doi.org/10.48550/arXiv.2402.11295
30. Zaruba, F., Schuiki, F., Hoefler, T., Benini, L.: Snitch: A tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads. IEEE Transactions on Computers **70**(11), 1845–1860 (Nov 2021). https://doi.org/10.1109/TC.2020.3027900
31. Zheng, Y., Chen, Y., Qian, B., Shi, X., Shu, Y., Chen, J.: A review on edge large language models: Design, execution, and applications (2025), `https://arxiv.org/abs/2410.11845`