Optimizing Llama.cpp and GGML for RISC-V Vector (RVV)

Taimur Ahmad[1], Rehan Qasim[1], David Baker[2], Ahsan Jalil[1], Saad Bin Nasir[1]
[1]10xEngineers Inc. [2]Akeana Inc.

RISC-V Summit Europe 2026 | Bologna, Italy

Outline

Background & Motivation

- Llama.cpp and RISC-V
- Current gaps in RVV support

Our Approach - 3 Components

- Kernel coverage
  - Floating-Point (FP) Kernels
  - Quantization Kernels
- VLEN-aware repacking
- Testing framework

Results & Benchmarks

- Collaboration with **RISE RP-014**
- All benchmarking performed on **Banana Pi BPI-F3 (VLEN=256)** for **TinyLlama 1.1B**
- All artifacts discussed and presented are **upstreamed or being upstreamed** to mainline Llama.cpp **and/or open-source in RISE repos**

Background & Motivation

Llama.cpp is where LLMs run on CPUs

- Most popular open-source CPU inference framework
- Supports all major models and quantization formats
- First-class support for x86 and ARM

RVV Adoption in AI Hardware is Growing

- Hardware shipping with RVV: BPI-F3, MILK-V Jupiter
- VLENs from 128-bit to 1024-bit and beyond

But RISC-V Lacks Support in Llama.cpp

- Many floating-point kernels not vectorized for RVV
- Quantized vec-dot: only legacy + K-Quants, VLEN ≤256
- **No VLEN-aware** GEMM/GEMV optimizations

Architecture support in Llama.cpp

	x86	ARM	RISC-V
FP kernels	✓	✓	✗
All quant types	✓	✓	✗
Specialized GEMM/GEMV	✓	✓	✗
Wide VLEN support	N/A	N/A	✗

Goal: Make RISC-V a first-class citizen in Llama.cpp

Project Scope

Three components to bring RISC-V on par with x86 and ARM in Llama.cpp

1 Kernel Coverage

Find missing RVV implementations

- Floating-point kernels
(dot, mad, scale, silu, convert)
- Specialized kernels for Matmul and Flash Attention
- Quantized vec-dot
(I-Quants, Ternary, MXFP4)

VLENs 128 - 1024

→

2 VLEN-Aware Repacking

Improve existing matmul path

- Repack weights and activations
- Outer-product GEMM/GEMV and repacking kernels
- Tile sizes templated per VLEN

FP + legacy + K-Quants

→

3 Testing Framework

Verify correctness across VLENs

- test-float, test-repack
- perf-float, perf-repack
- QEMU: VLENs 128 - 1024
- BPI-F3: performance (256)

Open-source repo

All work supported by RISE RP-014 | Being upstreamed to mainline Llama.cpp

Floating-point Kernel Coverage

*FP kernels are used by several key operations in Llama.cpp such as matrix-multiplication, flash attention and softmax. Many had **missing RVV** implementations.*

Kernel	Category	GGML Operation	Data Types
vec_dot	**Dot product**	**OP_MUL_MAT**	**FP16**, FP32, **BF16**
vec_dot_unroll	**Dot product**	**OP_MUL_MAT**	**FP16**
vec_mad	**Multiply-add**	**OP_FLASH_ATTN**	**FP16**, **FP32**
vec_scale	**Scaling**	**OP_FLASH_ATTN**	**FP16**, **FP32**
vec_silu	**Activation**	**OP_SILU**	**FP32**
vec_swiglu	Activation	OP_SWIGLU	FP32
vec_soft_max	**Activation**	**OP_SOFT_MAX**	**FP32**
vec_expf	Activation	OP_SOFT_MAX	FP32
cpu_fp_convert	**Conversion**	**various**	**FP16, FP32, BF16**
vec_add/sub/mul/div	Utility	various	FP16, FP32

We worked with **several** floating-point kernels used by core GGML operations.
6 were pushed upstream by 10xE. All kernels support VLENs **128-bit to 1024-bit**.

***Bold** indicates kernels that were covered in this project*

FP Optimization & Results

Sample: vec_dot_f16 (vectorized)

```
void ggml_vec_dot_f16(...) {
  int vl = __riscv_vsetvlmax_e32m8();
  vfloat32m8_t sum0 = vfmv(0.0f, vl);
  vfloat32m8_t sum1 = vfmv(0.0f, vl);
  ...
  ...
  for (int i = 0; i < n; i += step) {
    // Load x and y as fp16
    ax0 = __riscv_vle16_v_f16m4(x+i, epr);
    ay0 = __riscv_vle16_v_f16m4(y+i, epr);
    // Widening FMA: fp16 x fp16 -> fp32
    sum0 = __riscv_vfwmacc_vv(sum0,ax0,ay0, epr);
    asm volatile ("" ::: "memory")

    ax1 = __riscv_vle16_v_f16m4(x+i+epr, epr);
    ay1 = __riscv_vle16_v_f16m4(y+i+epr, epr);
    sum1 = __riscv_vfwmacc_vv(sum1,ax1,ay1, epr);
  }

  // Reduce to scalar
  ...
}
```

LMUL Sweep

Clobbering

Loop Unrolling

Optimization factors explored:

- LMUL (1, 2, 4, 8)
- Loop unrolling (1, 2, 4, 8)
- Memory clobbering

Best configurations (cache hot & cold), Row Size = 2048, Threads = 1

Kernel	LMUL	Unroll	Clobber	Hot (ns)	Cold (ns)	Speedup
vec_dot_f16	2	2	Yes	291	1917	**9.0×**
vec_scale_f16	4	2	Yes	291	1500	**3.9×**
vec_mad_f16	4	2	Yes	375	2250	**3.7×**
cpu_fp16_to_fp32	2	2	No	416	2125	**4.1×**
vec_silu_f32	2	1	No	6083	7041	**11.2×**

Best config selected per kernel via benchmarking

- 1000 iterations per config at various input sizes (numbers shown above for **2048 vector size**)
- Best time selected by cache hot first and then cache cold
- Tested on 64-byte alignment
- Threads 1 to 8

Benchmarked on Banana Pi BPI-F3 (VLEN=256) with Spacemit X1 CPU

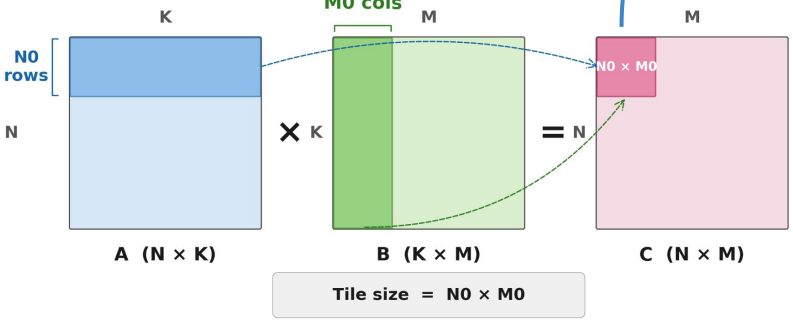
Llamafile SGEMM for RVV

What is Llamafile SGEMM?

- Tiled matrix-matrix multiplication kernels
- Used for Prefill / Prompt Processing
- Exists for x86, ARM, PowerPC
- Missing for RVV
- Implemented for FP16, FP32 and BF16

GGML_OP_MUL_MAT Kernel Selection

- Matrix x Matrix (Prefill): → **Llamafile SGEMM**
- Matrix x Vector (Decode): → **Vector Dot**



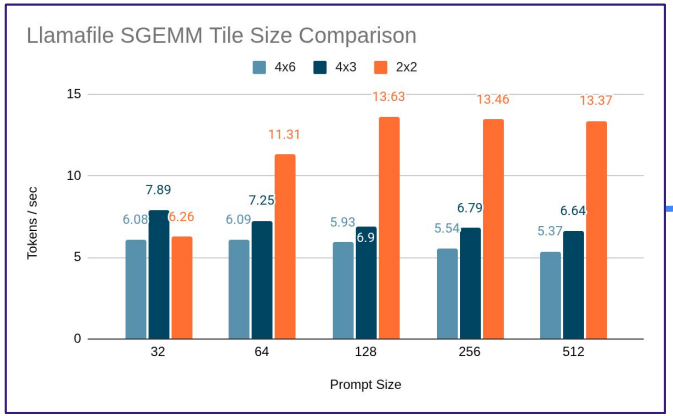

```
template <int M0, int N0>
   inline void gemm_bloc(int64_t ii, int64_t jj) {
       D Cv[N0][M0] = {};  // Accumulators based on tile size

       // Iterate over full rows
       for (int64_t l = 0; l < k; l += K) {
               // Load all rows of B
              V Bv[N0];
              for (int64_t j = 0; j < N0; ++j) {
                  Bv[j] = load<V>(B + ldb * (jj + j) + l);
              }

              // Load 1 row of A and accumulate
              for (int64_t i = 0; i < M0; ++i) {
                  V Av = load<V>(A + lda * (ii + i) + l);
                  for (int64_t j = 0; j < N0; ++j) {
                      Cv[j][i] = madd(Av, Bv[j], Cv[j][i]);
                  }
              }
          }
       }

       // Reduce
       for (int64_t j = 0; j < N0; ++j)
           for (int64_t i = 0; i < M0; ++i)
               C[ldc * (jj + j) + (ii + i)] = reduce(Cv[j][i]);
   }
```

Underlying M0 x N0 GEMM Kernel

Llamafile SGEMM - Results

Tile Size Selection for SGEMM for (FP16 x FP16 -> FP32)

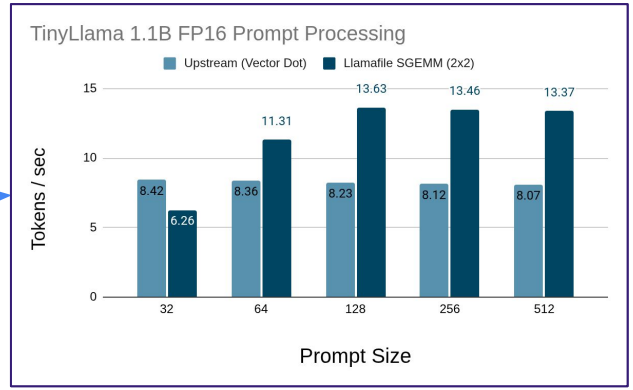
Tile	LMUL	Acc Regs	LHS Regs	RHS Regs	Total Regs
4×6	1	4 x 6 x m1 = **24**	1 x mf2 = **1**	6 x mf2 = **6**	31
4×3	2	4 x 6 x m2 = **24**	3 x m1 = **6**	1 x m1 = **1**	31
2×2	4	2 x 2 x m4 = **16**	2 x m2 = **4**	2 x m2 = **4**	24



End-to-end: TinyLlama 1.1B FP16 Prompt Processing

Selected 2x2 (LMUL=4) for VLEN=256

End-to-end: TinyLlama 1.1B FP16 Prompt Processing



Key findings

- **SGEMM 2×2 -> 1.5×** over vec-dot for prompt ≥64
- Small prompts (32): Vector Dot performs better

Benchmarked on Banana Pi BPI-F3 (VLEN=256) with Spacemit X1 CPU (8 threads)

Flash Attention - GEMM

Flash Attention in Llama.cpp

- Fuses $Q \cdot K^T \rightarrow$ softmax $\rightarrow \times V$ into a single pass
- Default path in Llama.cpp

Three execution paths:

- Split-KV (decode): n_seq==1, n_kv≥512
- Non-tiled (fallback): everything else
- Tiled (prefill): n_seq≥64, uses GEMM

Tiled path uses SIMD GEMM which was missing for RVV

- **Flash Attention** takes only 5-6% of total compute
- **9x improvement** in operator only results in around **5-8% improvement** in end-to-end

Op-level Performance (GFLOPs)

@TinyLlama-1B parameters hsk=64, hsv=64, nh=4, nr23=[8,1], mask=1, prec=f32, type_KV=f16

Tokens	Upstream	Vectorized	Speedup
128	2.69	**22.75**	**8.5×**
256	2.49	**23.70**	**9.5×**
512	2.69	**24.07**	**9.0×**
1024	2.60	**24.43**	**9.4×**
2048	2.64	**23.48**	**8.9×**
4096	2.72	**23.16**	**8.5×**

End-to-End Model-level: TinyLlama 1B FP32

Prompt	Upstream	Vectorized	Speedup
128	6.28	**6.58**	**1.05×**
256	6.05	**6.46**	**1.07×**
512	6.03	**6.51**	**1.08×**

Benchmarked on Banana Pi BPI-F3 (VLEN=256) with Spacemit X1 CPU (8 threads)

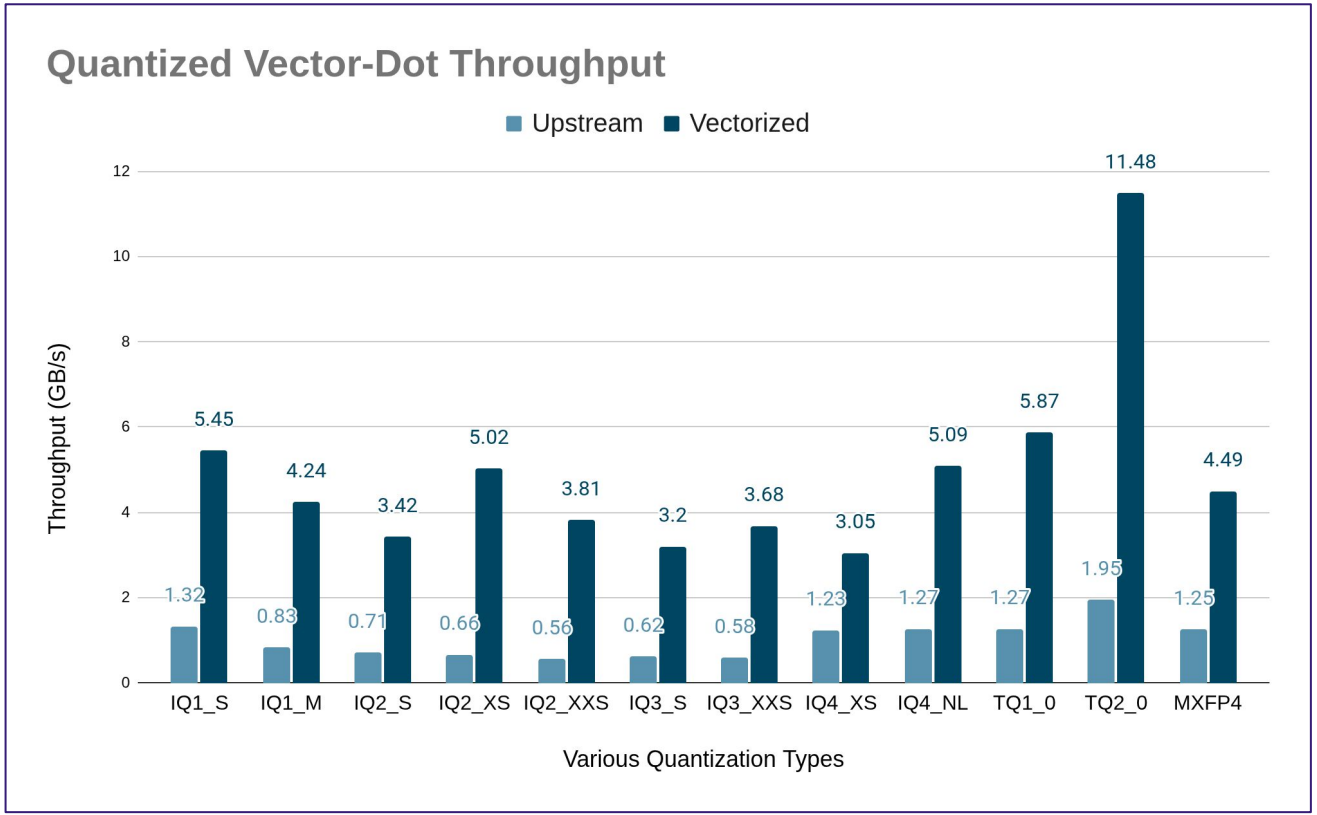
Quantization Kernels Coverage

Extended vec-dot support beyond legacy and K-Quants to cover all modern quantization types. ***Quantization types*** *are widely used by* ***Llama.cpp users****, hence requiring* ***proper vectorized support****.*

Quant Type	Kernels	VLENs	Status
Legacy	Q4_0, Q5_0, Q4_1, Q5_1, Q8_0	128–256+	Already upstream
K-Quants	Q2_K, Q4_K, Q5_K	128–256+	Already upstream
	Q3_K, Q6_K	128, 256, **512, 1024**	**Our work**
I-Quants	**IQ1_S, IQ1_M, IQ2_S, IQ2_XS, IQ2_XXS, IQ3_S, IQ3_XXS, IQ4_XS, IQ4_NL**	**128–1024**	**Our work**
Ternary	**TQ1_0, TQ2_0**	**128–256+**	**Our work**
MXFP4	**MXFP4**	**128–256+**	**Our work**

We added **12 new quantization vec-dot kernels** (9 I-Quants + 2 Ternary + MXFP4) with VLEN support from **128 to 1024-bit**.

Quantization Vector Dot - Results



Up to **3-7x** improvement across quantization types

Benchmarked on Banana Pi BPI-F3 (VLEN=256) with Spacemit X1 CPU

*Llama.cpp's official **test-quantize-perf** tool used with single thread*

End of component 1: Kernel Coverage

Why VLEN-Aware Repacking?

Matrix multiplication dominates LLM inference

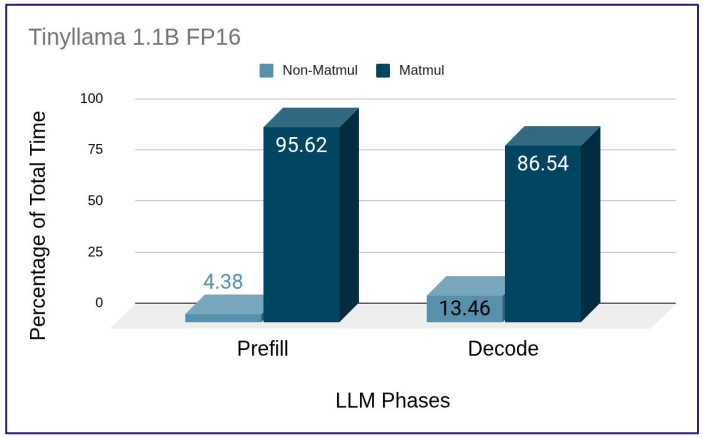
- >90% of compute time is spent in matmul operations
- Both prefill (GEMM) and decode (GEMV) are matmul-bound

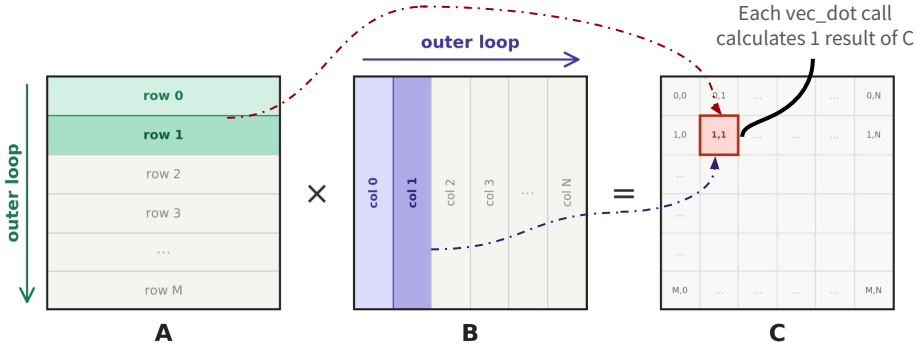
Current vec-dot approach has poor data reuse

- Matmul and matvec through multiple vec-dot calls
- Same weight data loaded repeatedly from memory
- Poor data reuse

Llama.cpp provides a repacking interface

- Already used by x86 and ARM for improved performance
- Weights repacked during model loading (one-time cost)
- **Missing for RISC-V - we added it**





Two outer loops (rows of A × columns of B) → each vec_dot call fills one cell of C

No data reuse: each row of A and column of B is reloaded on every call

Repacking - Implementation

What we implemented:

Repacking functions + **GEMM/GEMV kernels** for:

- **Floating-point**: FP16, FP32
- **Legacy quants**: Q4_0, Q8_0
- **K-Quants**: Q2_K, Q3_K, Q4_K, Q5_K, Q6_K

All templated with VLEN (128–1024)

Design decisions

- Tile sizes **maximize register usage** while avoiding spills
- **Outer-product** approach: accumulate into tile-sized output
- Builds on **IREE MMT4D** for RVV
- **Weights repacked ahead of time** during Llama.cpp initialization
- **Activations** repacked **before** each matmul

Tile sizes (selected via benchmarking + register pressure)

Floating-point

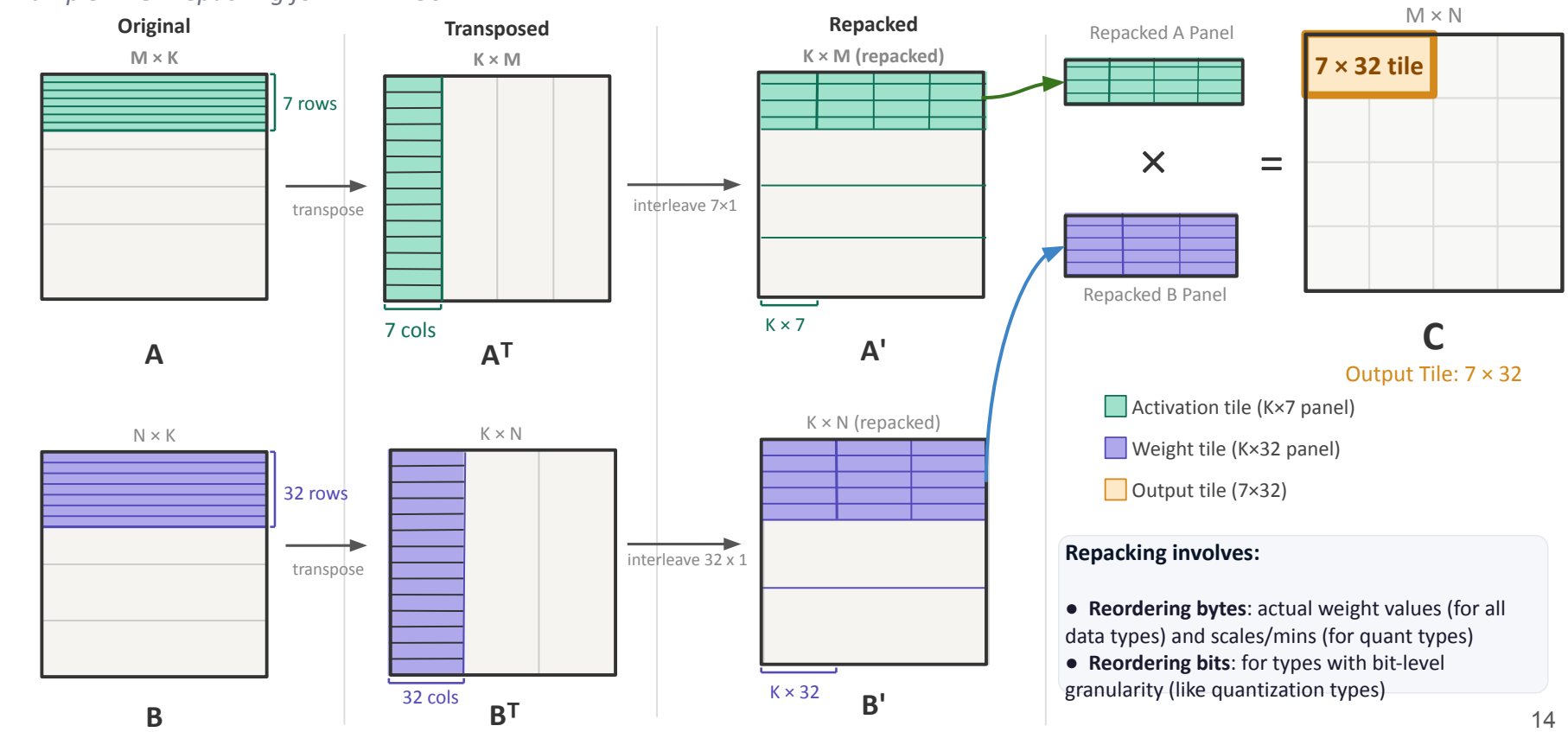
Phase	Activations	Weights	Output
Prefill	7 x 1	VLEN / 8 x 1	7 x VLEN / 8
Decode	Not Packed	VLEN / 8 x 1	1 x VLEN / 8

Quantization (Legacy and K-Quants)

Phase	Activations	Weights	Output
Prefill	4 x 1	VLEN/16 x 1	4 × VLEN/16
Decode	Not Packed	VLEN/16 x 1	1 × VLEN/16

Repacking - Floating-Point Example

Example: FP32 repacking for VLEN=256

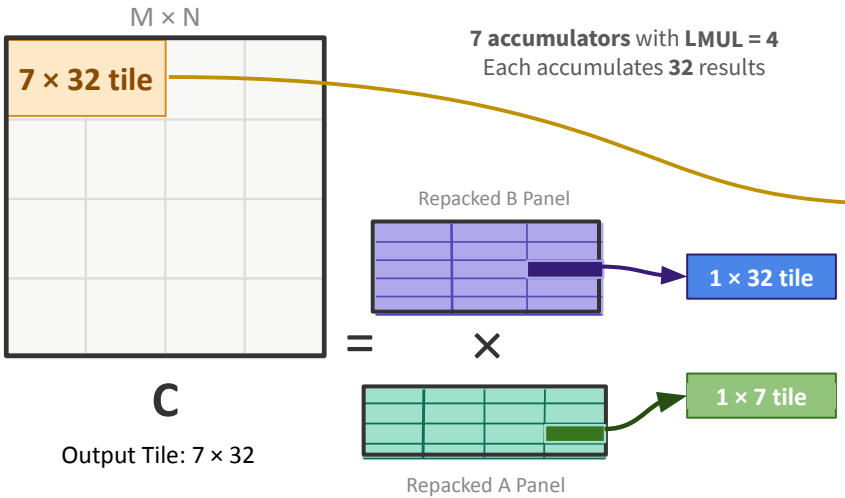


Repacking involves:

- **Reordering bytes**: actual weight values (for all data types) and scales/mins (for quant types)
- **Reordering bits**: for types with bit-level granularity (like quantization types)

Repacking - Floating-Point Example

Example: FP32 repacked GEMM for VLEN=256



Each **inner loop broadcast multiplies** each individual element of the **1x7 A Tile with Tile B (1x32)**

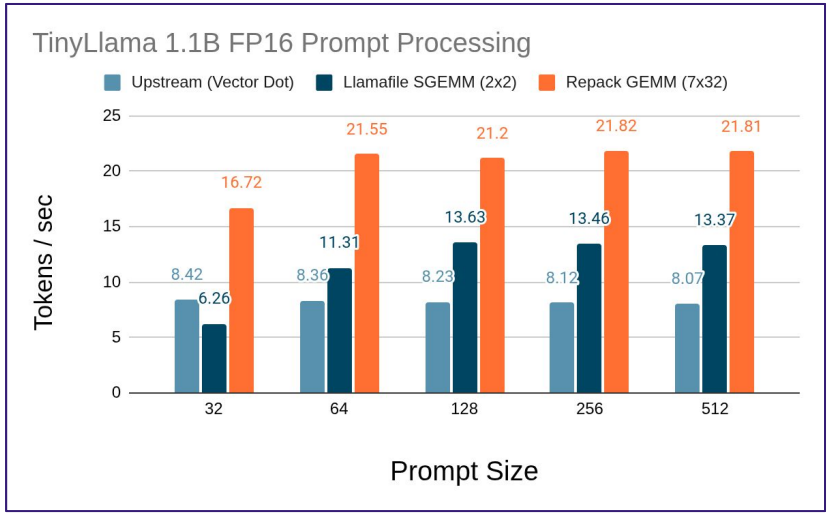
- **Repack GEMM** follows the same for quantization types with the difference being additional logic for unpacking scales/mins and re-ordering bits

```
void gemm_f32_7x32(int n, float *s, float *a, float *b) {
    const block_f16_7x1  * a_ptr = ...  // Activations
    const block_f16_32x1 * b_ptr = ...  // Weights
    // Accumulators
    vfloat32m4_t sumf_0 = __riscv_vfmv_v_f_f32m4(0.0f, 32);
    vfloat32m4_t sumf_1 = __riscv_vfmv_v_f_f32m4(0.0f, 32);
    vfloat32m4_t sumf_2 = __riscv_vfmv_v_f_f32m4(0.0f, 32);
    vfloat32m4_t sumf_3 = __riscv_vfmv_v_f_f32m4(0.0f, 32);
    vfloat32m4_t sumf_4 = __riscv_vfmv_v_f_f32m4(0.0f, 32);
    vfloat32m4_t sumf_5 = __riscv_vfmv_v_f_f32m4(0.0f, 32);
    vfloat32m4_t sumf_6 = __riscv_vfmv_v_f_f32m4(0.0f, 32);
    for (int l = 0; l < nb; l++) {
        vfloat32m4_t b_0 = __riscv_vle32_v_f32m4(&b_ptr[l].d[0], 32);
        sumf_0 = __riscv_vfmacc_vf_f32m4( sumf_0, a_ptr[l].d[0] , b_0, 32);
        sumf_1 = __riscv_vfmacc_vf_f32m4( sumf_1, a_ptr[l].d[1] , b_0, 32);
        sumf_2 = __riscv_vfmacc_vf_f32m4( sumf_2, a_ptr[l].d[2] , b_0, 32);
        sumf_3 = __riscv_vfmacc_vf_f32m4( sumf_3, a_ptr[l].d[3] , b_0, 32);
        sumf_4 = __riscv_vfmacc_vf_f32m4( sumf_4, a_ptr[l].d[4] , b_0, 32);
        sumf_5 = __riscv_vfmacc_vf_f32m4( sumf_5, a_ptr[l].d[5] , b_0, 32);
        sumf_6 = __riscv_vfmacc_vf_f32m4( sumf_6, a_ptr[l].d[6] , b_0, 32);
    }
    __riscv_vse32_v_f32m4(&s[...],  sumf_0, 32);
    __riscv_vse32_v_f32m4(&s[...],  sumf_1, 32);
    __riscv_vse32_v_f32m4(&s[...],  sumf_2, 32);
    __riscv_vse32_v_f32m4(&s[...],  sumf_3, 32);
    __riscv_vse32_v_f32m4(&s[...],  sumf_4, 32);
    __riscv_vse32_v_f32m4(&s[...],  sumf_5, 32);
    __riscv_vse32_v_f32m4(&s[...],  sumf_6, 32);
```

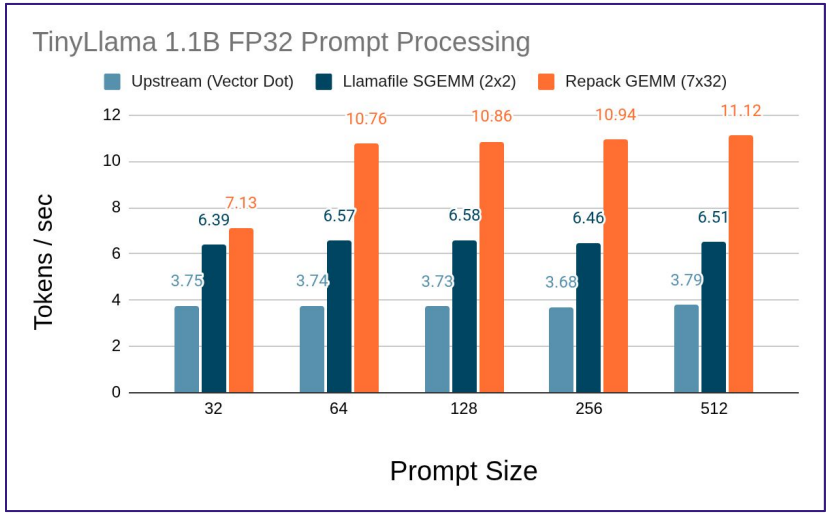
Underlying 7x32 Repacked GEMM Kernel

Repacked GEMM - Floating-point Results

End-to-end: TinyLlama 1.1B FP16 Prompt Processing



End-to-end: TinyLlama 1.1B FP32 Prompt Processing



Prompt Processing
~2.5x over vector dot
~**1.5x** over SGEMM

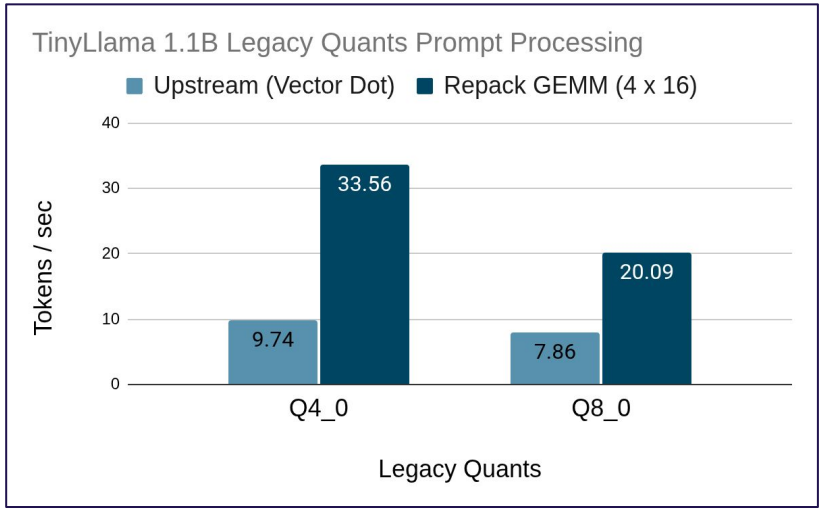
Decode
Maintains performance with upstream vector dot

Benchmarked on Banana Pi BPI-F3 (VLEN=256) with Spacemit X1 CPU on 8 threads

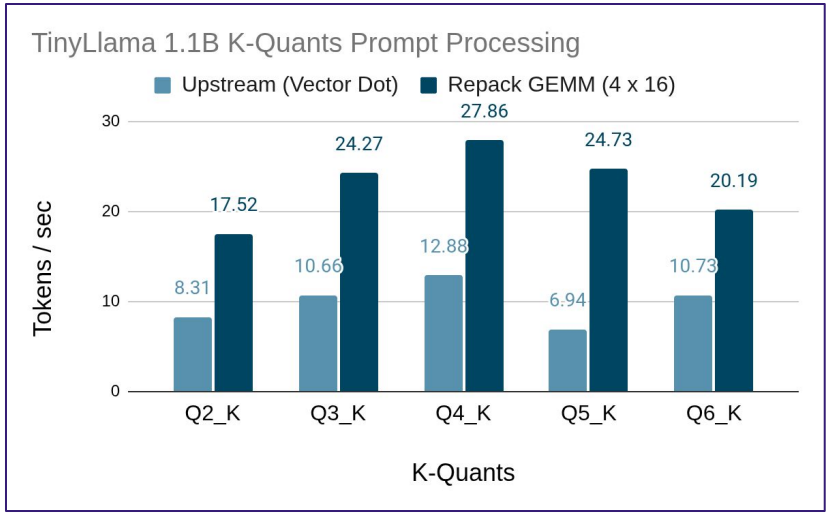
*Llama.cpp's official **llama-bench** tool*

Repacked GEMM - Quantization Results

End-to-end: TinyLlama 1.1B Legacy Quants Prompt Processing



End-to-end: TinyLlama 1.1B K-Quants Prompt Processing



End of component 2: VLEN-Aware Repacking

Prompt Processing
2-3x over upstream vector dot

Decode
Maintains performance with upstream vector dot, with degradation in **Q3_K and Q6_K**

Benchmarked on Banana Pi BPI-F3 (VLEN=256) with Spacemit X1 CPU on 8 threads

*Llama.cpp's official **llama-bench** tool*

Testing Framework

Extend upstream Llama.cpp testing environment with additional functional correctness and benchmarking files.
https://github.com/riseproject-dev/llama.cpp-validation

Category	Type	Testing Files	Status
Kernel	**Floating-Point Kernels (vec dot + activation functions and utility function kernels)**	**test-float**	**Our work**
		perf-float	**Our work**
	Quantization Vector Dot	**test-quantize-fns**	Already upstream (**Modified by us to enable I-Quants**)
		test-quantize-perf	Already upstream
	Repack GEMM/GEMV (Floating-Point + Quantization)	**test-repack**	**Our work**
		perf-repack	**Our work**
Operator	Operator level testing (Flash-attention and llamafile-sgemm)	test-backend-ops	Already upstream
Model	Model-level Functional Correctness	llama-perplexity	Already upstream
	Model-level Benchmarking	llama-bench	Already upstream

- **QEMU**: functional correctness for VLENs 128-1024
- **BPI-F3**: performance benchmarking (VLEN=256)
- **Automated scripts** for reproducibility
- **CI/CD** enabled on the repo with native RISC-V hardware by Ludovic Henry (@luhenry)

***Bold** indicates tests that were worked on in this project*

End of component 3: Testing Framework

References & Acknowledgments

[1] llama.cpp: LLM Inference in C/C++
https://github.com/ggml-org/llama.cpp
[2] RISE RP-014: Optimizing Llama.cpp and GGML for RVV
https://lf-rise.atlassian.net/wiki/spaces/HOME/pages/628817924/Project+RP014+Optimizing+Llama.cpp+and+GGML+for+RVV
[3] Llamafile SGEMM
https://github.com/ggml-org/llama.cpp/blob/master/ggml/src/ggml-cpu/llamafile/sgemm.cpp
[4] IREE MMT4D for RISCV64
https://github.com/iree-org/iree/pull/20263
https://iree.dev/community/blog/2021-10-13-matrix-multiplication-with-mmt4d/
[5] Repos
https://github.com/riseproject-dev/llama.cpp
https://github.com/riseproject-dev/llama.cpp-validation

Acknowledgements

This work is supported by RISE (RISC-V Software Ecosystem) under Project RP-014.
We thank the Llama.cpp maintainers and the RISC-V community for their support.

Summary

Kernel Coverage
Finalized missing RVV support: FP ops + 12 quant types, VLEN 128 to 1024

VLEN-Aware Repacking
2-3× prompt processing improvement for floating-point and quantized workloads

Testing Framework
Extended verification across VLENs with custom test infrastructure

All artifacts discussed and presented are upstreamed or being upstreamed to mainline Llama.cpp and/or open-source in RISE repos

Supported by RISE RP-014

Thank You!

Questions?

https://10xengineers.ai/contact-us/

Also visit our other poster presentations at this summit!