Project RP014: Optimizing Llama.cpp and GGML for RVV

Implementation Plan

- **Phase 1: VLEN-agnostic RVV Kernels**
  - **Floating-Point Kernels**
    - **SIMD Mappings**
    - **High-priority Kernels**
    - **Activation Functions and other Utilities**
    - Llamafile SGEMM
    - Repacking GEMM and GEMV
  - Quantization Kernels
    - Quantization/Dequantization
    - Vec Dot
    - Repack GEMM and GEMV
    - Llamafile SGEMM Quantization
- Phase 2: Dynamic Dispatching
  - Integration
- Phase 3: Benchmarking and Testing Software
  - Development
  - Integration with Mainline Llama.cpp

Floating-Point Kernels

RVV SIMD mappings for **GGML FP16/FP32 functions** in ***simd_mappings.h***

- Upstream [**PR #15057**] → added fixed **LMUL=2** mappings
- **Explicit per-kernel fallback needed → not portable in RVV** (C/C++ restrictions) or **different LMUL needs**

Extend PR #15057 to add support for remaining floating-point kernels:

- ggml_vec_dot_bf16
- ggml_vec_scale_f16/f32
- ggml_vec_mad_f16
- Conversion Kernels (FP32 to FP16, etc)
- Activation functions (**ggml_silu)** and other utilities (**vec_mul, vec_div)**, etc

Llamafile SGEMM: Provides **tiled matmul** for **prompt processing**.

- Explore tiling combinations under varying LMUL configurations → pick **most effective** based on **benchmarking** results.

SIMD Mappings

RVV SIMD mappings for FP16/FP32 in ***simd_mappings.h*** *(incompatible with RVV)*

- Architecture-agnostic **GGML C Macros**
- **Covers:**
  - LOAD, STORE
  - FMA, ADD, MUL, REDUCE,
  - ZERO, SET1, STEP, EPR

Status

- Upstream: **PR #15057** added fixed **LMUL=1** mappings for **FP32**

```
GGML_F32_VEC sum[GGML_F32_ARR];
GGML_F32_VEC ax[GGML_F32_ARR];
GGML_F32_VEC ay[GGML_F32_ARR];

for (...) {
        for (...) {
          ax[j] = GGML_F32_VEC_LOAD(...);
          ay[j] = GGML_F32_VEC_LOAD(...);
          sum[j] = GGML_F32_VEC_FMA(...);
        }
}
```

Issues with RVV

1. Some macros (e.g., ggml_vec_dot_f16) use array-like accumulators → **not portable in RVV** (C/C++ restrictions)

2. **LMUL fixed at compile-time** → can't vary across kernels. **Explicit per-kernel fallback needed** if LMUL changes.

SIMD Mappings

- **SIMD Mappings** are utilized by various architectures to implement **Floating-Point Kernels**

Kernels

List of kernels that use these SIMD mappings (for other architectures):

- `ggml_vec_dot_f16`
- `ggml_vec_dot_f32`
- `ggml_vec_dot_f16_unroll`
- `ggml_vec_mad_f16`
- `ggml_vec_mad_f32`
- `ggml_vec_mad_f32_unroll`
- `ggml_vec_mad1_f32`
- `ggml_vec_scale_f16`
- `ggml_vec_scale_f32`

Operators

List of operators that use these SIMD mappings (for other architectures):

- **`ggml_compute_forward_conv_2d_dw_cwhn`**
- `ggml_compute_forward_ssm_scan_f32`
- `ggml_compute_forward_rwkv_wkv7_f32`

These operators are **not used by TinyLlama or BERT**

We will not be looking into these

Floating-Point Kernels

In addition to the SIMD kernels, the following kernels are also vectorized for other architectures.

We will also be looking into these.

Vector Dot Kernels

- `ggml_vec_dot_bf16`

Activation Functions

- `ggml_vec_silu_f32`
- `ggml_vec_swiglu_f32`
- `ggml_vec_soft_max_f32`
- `ggml_v_expf`
- `ggml_v_silu`

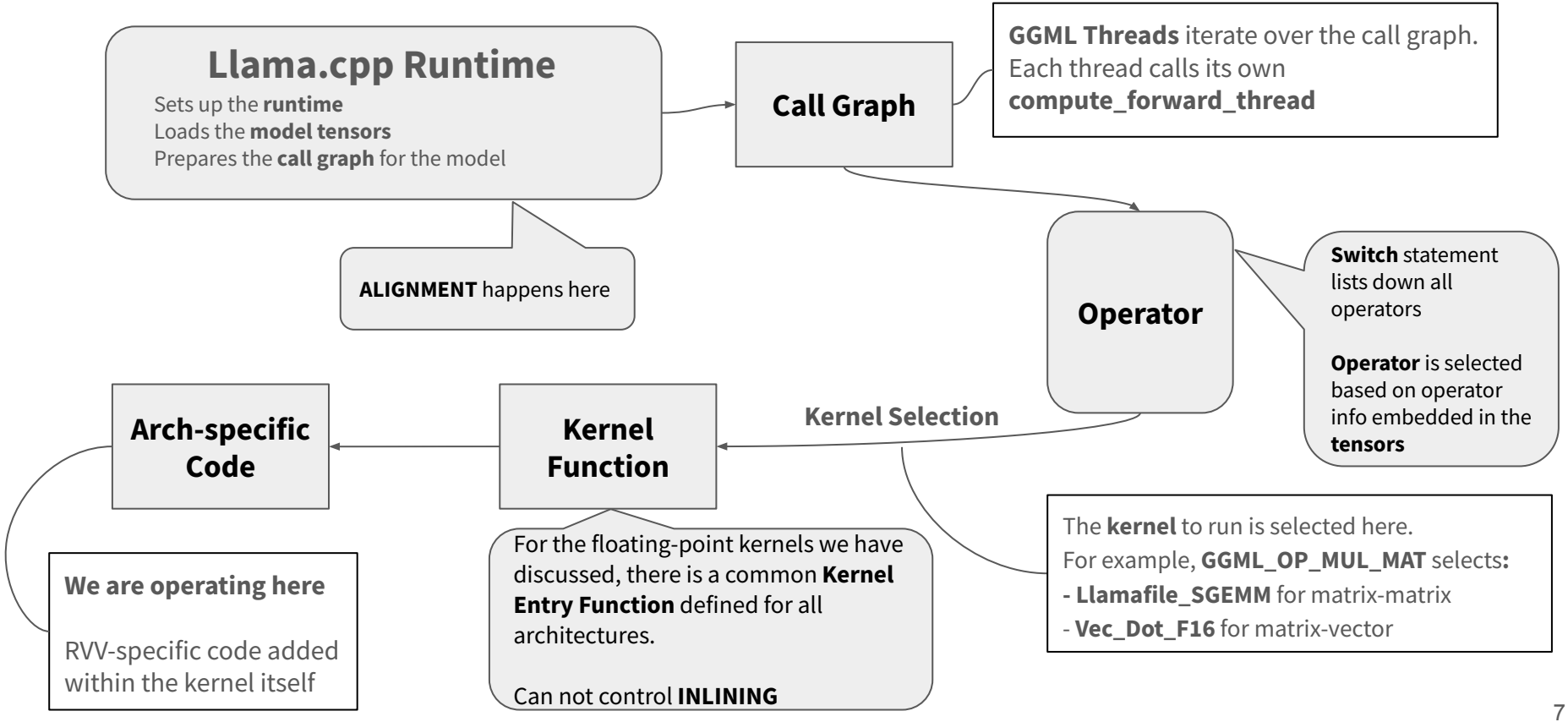
Floating-Point Conversion

- `ggml_cpu_fp32_to_fp16`
- `ggml_cpu_fp16_to_fp32`
- `ggml_cpu_bf16_to_fp32`

Utilities

- `ggml_vec_add_f16/f32`
- `ggml_vec_sub_f16/f32`
- `ggml_vec_mul_f16/f32`
- `ggml_vec_div_f16/f32`

Llama.cpp Execution Flow



Floating-Point Kernels Format

- Each floating-point kernel implements a **SIMD implementation** (usually through a SIMD mappings) and a **scalar fallback**
- An **exception** is added for any **architecture**, where SIMD mappings can not be used

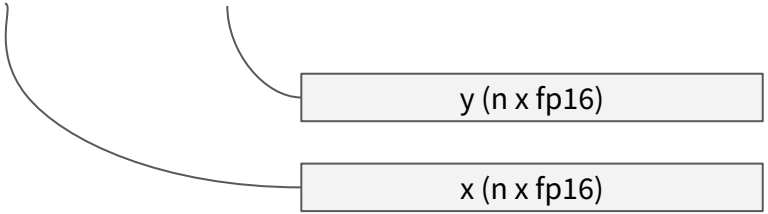
```
void ggml_vec_dot_f16(int n, float *s, ggml_fp16_t *x, ggml_fp16_t *y) {

#if defined(GGML_SIMD)
        // SIMD/Vector Route

        #if defined(__ARM_FEATURE_SVE)
        // ARM SVE Implementation
      ...
        #elif defined(__riscv_v_intrinsic)
        // RVV Implementation
        ...
        #else
        // SIMD Mappings
        ...
#else
        // Scalar Route
        …
}
```

Common entry point for all archs

We will add exception for RISC-V, similar to ARM SVE

Kernel Structure

```
void ggml_vec_dot_f16(int n, float *s, ggml_fp16_t *x, ggml_fp16_t *y) {
#if defined(GGML_SIMD)
        // SIMD/Vector Route

        #if defined(__ARM_FEATURE_SVE)
        // ARM SVE Implementation
       ...
        #elif defined(__riscv_v_intrinsic)
        // RVV Implementation
        ...
        #else
        // SIMD Mappings
        ...
#else
        // Scalar Route
        …
}
```



Kernels operate on individual rows of data

No room for TILING here

Templating

- Templating structure not used by other architectures here
- Kernels such as **Llamafile SGEMM** implement **tiling** for **prompt processing**
  - We will add **tiling** and **templating** in these kernels

Vec Dot BF16 (Scalar)

```
void ggml_vec_dot_bf16(int n, float *s, ggml_bf16_t *x, ggml_bf16_t *y) {
    ggml_float sumf = 0;

    for (; i < n; ++i) {
        sumf += (ggml_float)(GGML_BF16_TO_FP32(x[i]) *
                             GGML_BF16_TO_FP32(y[i]));
    }

    *s = sumf;
}
```

y (n x bf16)

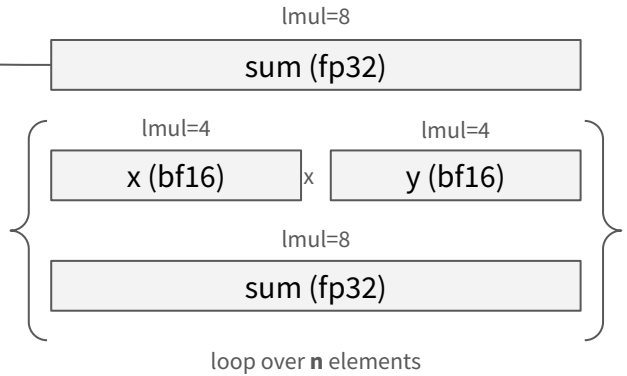
x (n x bf16)

Accumulated dot product of **x** and **y** returned in **s**

Vec Dot BF16 (Vector)

```
void ggml_vec_dot_bf16(int n, float *s, ggml_bf16_t *x, ggml_bf16_t *y) {
    ggml_float sumf = 0;

    // Initialize `sum` to all zeroes.
    int vl = __riscv_vsetvlmax_e32m8();
    vfloat32m8_t sum = __riscv_vfmv_v_f_f32m8(0.0f, vl);

    // Multiply-accumulate over `n` elements.
    for (int i = 0; i < n; i += vl) {
        vl = __riscv_vsetvl_e16m4(n - i);
        vbfloat16m4_t ax = __riscv_vle16_v_bf16m4((const __bf16 *)&x[i], vl);
        vbfloat16m4_t ay = __riscv_vle16_v_bf16m4((const __bf16 *)&y[i], vl);
        sum = __riscv_vfwmaccbf16_vv_f32m8(sum, ax, ay, vl);
    }

    // Reduce and accumulate in `sumf`.
    vl = __riscv_vsetvlmax_e32m8();
    vfloat32m1_t redsum = __riscv_vfredusum_vs_f32m8_f32m1(sum, __riscv_vfmv_v_f_f32m1(0.0f, 1), vl);
    sumf += __riscv_vfmv_f_s_f32m1_f32(redsum);

    *s = sumf;
}
```

Processing maximum possible elements at lmul=8



Reduction sum accumulated in **fp32**

Vec Mad F16 (Scalar)

```
inline static void ggml_vec_mad_f16(const int n, ggml_fp16_t *y, const ggml_fp16_t *x, const float v) {
      for (int i = 0; i < n; ++i) {
          y[i] = GGML_CPU_FP32_TO_FP16(
              GGML_CPU_FP16_TO_FP32(y[i]) +
              GGML_CPU_FP16_TO_FP32(x[i])*v);
      }
}
```

x (n x fp16)

y (n x fp16)

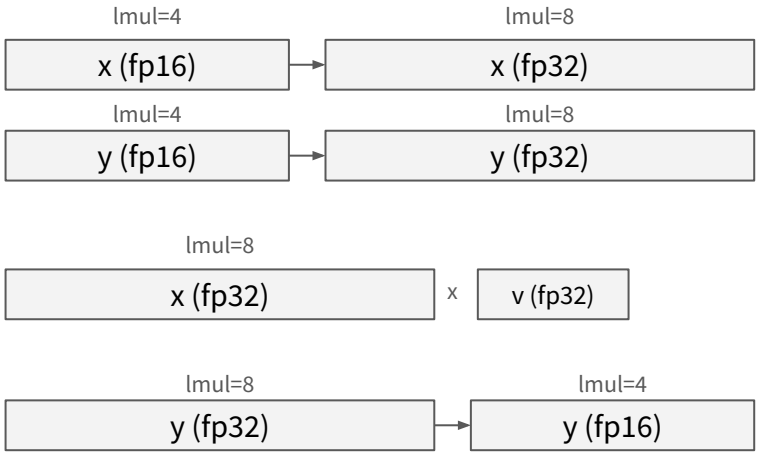
Constant f32 scale **v**

Result stored back in **y**

Vec Mad F16 (Vector)

```
inline static void ggml_vec_mad_f16(const int n, ggml_fp16_t *y, const ggml_fp16_t *x, const float v) {
    int vl  = __riscv_vsetvlmax_e16m4();
    vfloat16m4_t ax16, ay16;
    vfloat32m8_t ax, ay;
    for (int i = 0; i < n; i += vl) {
        vl = __riscv_vsetvl_e16m4(n - i);

        // Widen `x` and `y` to `fp32`.
        ax16 = __riscv_vle16_v_f16m4((const _Float16*)x + i, vl);
        ax = __riscv_vfwcvt_f_f_v_f32m8(ax16, vl);
        ay16 = __riscv_vle16_v_f16m4((_Float16*)y + i, vl);
        ay = __riscv_vfwcvt_f_f_v_f32m8(ay16, vl);

        // Multiply and narrow down back to `fp16`.
        ay = __riscv_vfmacc_vf_f32m8(ay, v, ax, vl);
        ay16 = __riscv_vfncvt_f_f_w_f16m4(ay, vl);
        __riscv_vse16_v_f16m4((_Float16*)y + i, ay16, vl);
    }
}
```

Narrow down to **fp16** and store in **y**



Vec Scale F16 (Scalar)

```
inline static void ggml_vec_scale_f16(const int n, ggml_fp16_t *y, const float v) {
    for (int i = 0; i < n; ++i) {
        y[i] = GGML_CPU_FP32_TO_FP16(
            GGML_CPU_FP16_TO_FP32(y[i])*v
        );
    }
```

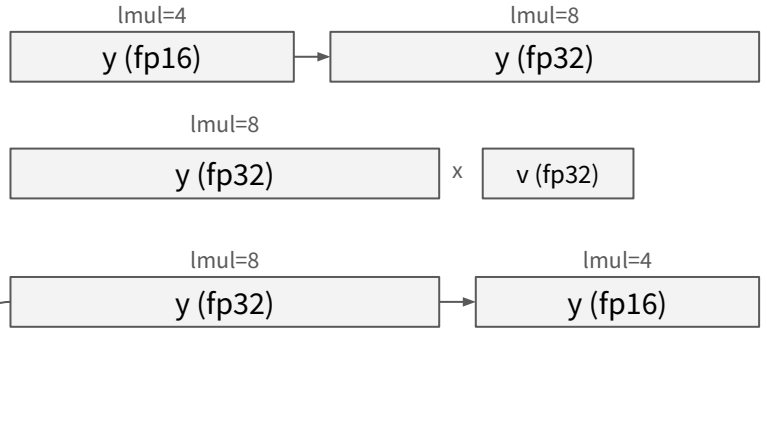
y (n x bf16)

Constant f32 scale **v**

Result stored back in **y**

Vec Scale F16 (Vector)

```
inline static void ggml_vec_scale_f16(const int n, ggml_fp16_t *y, const float v) {
    int vl;
    vfloat16m4_t ay;
    vfloat32m8_t tmp;
    for (int i = 0; i < n; i += vl) {
        vl = __riscv_vsetvl_e16m4(n - i);

        // Widen to `fp32`.
        ay = __riscv_vle16_v_f16m4((_Float16*)y + i, vl);
        tmp = __riscv_vfwcvt_f_f_v_f32m8(ay, vl);

        // Multiply with the scale.
        tmp = __riscv_vfmul_vf_f32m8(tmp, v, vl);

        // Narrow back to `fp16` and store.
        ay = __riscv_vfncvt_f_f_w_f16m4(tmp, vl);
        __riscv_vse16_v_f16m4((_Float16*)y + i, ay, vl);
    }
}
```



Vec Dot F16 Unroll (Scalar)

```
#define GGML_VEC_DOT_UNROLL  2

inline static void ggml_vec_dot_f16_unroll(const int n, const int xs, float *s, void *xv, ggml_fp16_t *y) {
    ggml_float sumf[GGML_VEC_DOT_UNROLL] = { 0.0 };

    ggml_fp16_t *x[GGML_VEC_DOT_UNROLL];

    for (int i = 0; i < GGML_VEC_DOT_UNROLL; ++i) {
        x[i] = (ggml_fp16_t *) ((char *) xv + i*xs);
    }

    // Scalar
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < GGML_VEC_DOT_UNROLL; ++j) {
            sumf[j] += (ggml_float)(GGML_CPU_FP16_TO_FP32(x[j][i])*GGML_CPU_FP16_TO_FP32(y[i]));
        }
    }

    for (int i = 0; i < GGML_VEC_DOT_UNROLL; ++i) {
        s[i] = (float)sumf[i];
    }
}
```

y (n x bf16)

x (n x bf16)

x (n x bf16)

Two accumulators

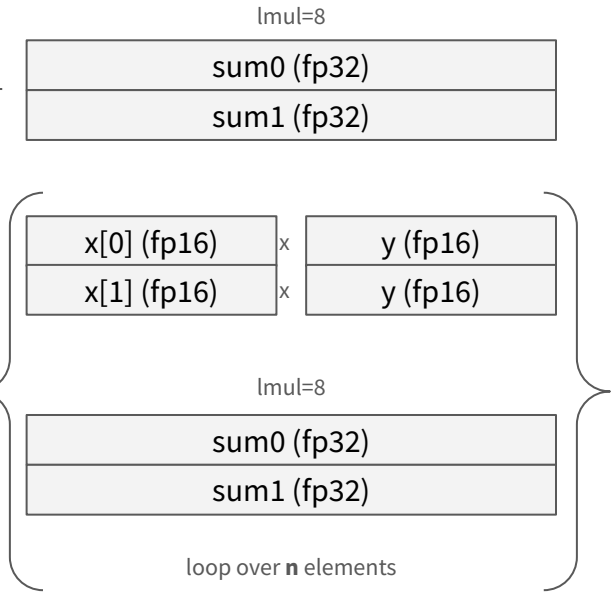
Vec Dot F16 Unroll (Vector)

```
inline static void ggml_vec_dot_f16_unroll (const int n, const int xs, float *s, void *xv, ggml_fp16_t *y) {
   ...

   int vl = __riscv_vsetvlmax_e32m8 ();
   // Initialize `sum0` and `sum1` to all zeroes.
   vfloat32m8_t sum0 = __riscv_vfmv_v_f_f32m8 (0.0f, vl);
   vfloat32m8_t sum1 = __riscv_vfmv_v_f_f32m8 (0.0f, vl);

   // Multiply-accumulate over `n` elements.
   for (int i = 0; i < n; i += vl) {
       vl = __riscv_vsetvl_e16m4 (n - i);
       vfloat16m4_t ay = __riscv_vle16_v_f16m4 ((const _Float16 *)y + i, vl);
       vfloat16m4_t ax0 = __riscv_vle16_v_f16m4 ((const _Float16 *)(x[0] + i), vl);
       vfloat16m4_t ax1 = __riscv_vle16_v_f16m4 ((const _Float16 *)(x[1] + i), vl);
       sum0 = __riscv_vfwmacc_vv_f32m8 (sum0, ax0, ay, vl);
       sum1 = __riscv_vfwmacc_vv_f32m8 (sum1, ax1, ay, vl);
   }

   // Reduce and store in sumf.
   vl = __riscv_vsetvlmax_e32m8 ();
   vfloat32m1_t redsum0 = __riscv_vfredusum_vs_f32m8_f32m1 (sum0, __riscv_vfmv_v_f_f32m1 (0.0f, 1), vl);
   sumf[0] = __riscv_vfmv_f_s_f32m1_f32 (redsum0);
   vfloat32m1_t redsum1 = __riscv_vfredusum_vs_f32m8_f32m1 (sum1, __riscv_vfmv_v_f_f32m1 (0.0f, 1), vl);
   sumf[1] = __riscv_vfmv_f_s_f32m1_f32 (redsum1);

   for (int i = 0; i < GGML_VEC_DOT_UNROLL ; ++i) {
       s[i] = (float)sumf[i];
   }
}
```

Processing maximum possible elements at lmul=8

loop over **n** elements

Two reduction sums



GGML BF16 TO FP32 (Scalar)

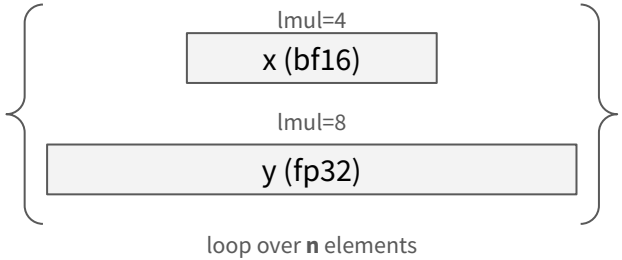
```
void ggml_cpu_bf16_to_fp32(const ggml_bf16_t * x, float * y, int64_t n) {
    for (; i < n; i++) {
        y[i] = GGML_BF16_TO_FP32(x[i]);
    }
}
```

x (n x bf16)

```
static inline float
ggml_compute_bf16_to_fp32(ggml_bf16_t h) {
    union {
        float f;
        uint32_t i;
    } u;
    u.i = (uint32_t)h.bits << 16;
    return u.f;
}
```

GGML BF16 TO FP32 (Vector)

```
void ggml_cpu_bf16_to_fp32(const ggml_bf16_t * x, float * y, int64_t n) {
    size_t vl;
    for (; i < n; i += vl) {
        vl = __riscv_vsetvl_e16m4(n - i);
        vbfloat16m4_t x_vec = __riscv_vle16_v_bf16m4(
                  (const __bf16*)x + i, vl);
        vfloat32m8_t y_vec = __riscv_vfwcvtbf16_f_f_v_f32m8(
                       x_vec, vl);
        __riscv_vse32_v_f32m8(y + i, y_vec, vl);
    }
}
```



GGML FP16 TO FP32 (Scalar)

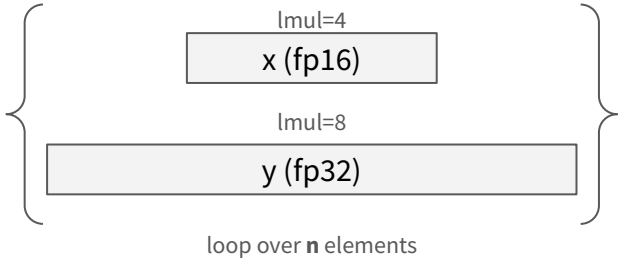
```
void ggml_cpu_fp16_to_fp32(const ggml_fp16_t * x, float * y, int64_t n) {
    for (; i < n; ++i) {
        y[i] = GGML_CPU_FP16_TO_FP32(x[i]);
    }
}
```

x (n x bf16)

```
static inline float
riscv_compute_fp16_to_fp32(ggml_fp16_t h) {
        _Float16 hf;
        memcpy(&hf, &h, sizeof(ggml_fp16_t));
        return hf;
}
```

GGML FP16 TO FP32 (Vector)

```
void ggml_cpu_fp16_to_fp32(const ggml_fp16_t * x, float * y, int64_t n) {
    size_t vl;
    for (; i < n; i += vl) {
        vl = __riscv_vsetvl_e16m4(n - i);
        vfloat16m4_t x_vec = __riscv_vle16_v_f16m4(
                        (const _Float16*)x + i, vl);
        vfloat32m8_t y_vec = __riscv_vfwcvt_f_f_v_f32m8(
                        x_vec, vl);
        __riscv_vse32_v_f32m8(y + i, y_vec, vl);
    }
}
```



Vec SiLU F32 (Scalar)

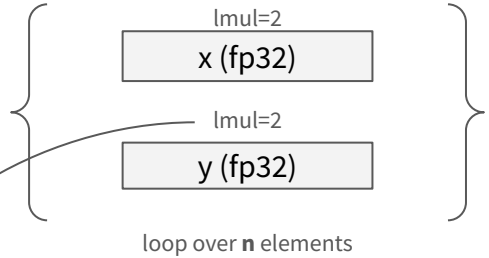
```
void ggml_vec_silu_f32(const int n, float * y, const float * x) {
    for (; i < n; ++i) {
        y[i] = ggml_silu_f32(x[i]);
    }
}
```

x (n x fp32)

```
// Sigmoid Linear Unit (SiLU) function
inline static float ggml_silu_f32(float x) {
    return x/(1.0f + expf(-x));
}
```

Vec SiLU F32 (Vector)

```
void ggml_vec_silu_f32(const int n, float * y, const float * x) {
    size_t vl;
    for (; i < n; i += vl) {
        vl = __riscv_vsetvl_e32m2(n - i);
        __riscv_vse32_v_f32m2(y + i, ggml_v_silu_m2 (
            __riscv_vle32_v_f32m2(x + i, vl), vl), vl);
    }
}
```



Upstream **ggml_v_silu** and **ggml_v_exp** use **LMUL=2**

Scaling Kernels (Removed Widening)

```
inline static void ggml_vec_scale_f16(const int n,
ggml_fp16_t *y, const float v) {
    int vl;
    vfloat16m4_t ay;
    vfloat32m8_t tmp;
    for (int i = 0; i < n; i += vl) {
        vl = __riscv_vsetvl_e16m4(n - i);

        // Widen to `fp32`.
        ay = __riscv_vle16_v_f16m4((_Float16*)y + i, vl);
        tmp = __riscv_vfwcvt_f_f_v_f32m8(ay, vl);

        // Multiply with the scale.
        tmp = __riscv_vfmul_vf_f32m8(tmp, v, vl);

        // Narrow back to `fp16` and store.
        ay = __riscv_vfncvt_f_f_w_f16m4(tmp, vl);
        __riscv_vse16_v_f16m4((_Float16*)y + i, ay, vl);
    }
}
```

```
inline static void ggml_vec_scale_f16 (const int n,
ggml_fp16_t *y, const float v) {
int vl;
    vfloat16m8_t ay;
    const ggml_fp16_t s = GGML_CPU_FP32_TO_FP16 (v);
    const _Float16 scale = *(const _Float16*)(&s);
    for (int i = 0; i < n; i += vl) {
        vl = __riscv_vsetvl_e16m8 (n - i);
        ay = __riscv_vle16_v_f16m8 ((_Float16*)y + i,
vl);
        // Multiply with the scale and store
        // back in `y`.
        ay = __riscv_vfmul_vf_f16m8 (ay, scale, vl);
        __riscv_vse16_v_f16m4 ((_Float16*)y + i, ay,
vl);
    }
}
```

Functional Testing Strategy

Methodology:

- Every optimized kernel is tested against the llama.cpp generic implementations.
- The test fails if the output difference exceeds a predefined error threshold (1e-3f).

Vector Sizes:

- Tested across a range of sizes [0, 1, 7, 16, 31, 32, 1024, 1025] to cover edge cases, unaligned, and SIMD-aligned scenarios.

Data Patterns:

- Inputs include typical values (cosine generated between 2.1 to -1.9),
- All zeros
- Special floating-point numbers (INF, -INF, NaN).

Edge-Case Scalars:

Operations using scalars are tested with values like 0.0, 1.0, -1.0, INF, and NaN.

Padding: 0, 5, 8, 16

Functional Testing CLI (test-float-fns)

```
(llama.cpp) taimur-ahmad@bpf3-16G-3:~/10x/llama.cpp-10x$ ./release/bin/test-float-fns
Executing Kernel Tests.

--- Testing ggml_vec_scale_f16---
Tests Passed 144/144
--- Testing ggml_vec_mad_f16 ---
Tests Passed 192/192
--- Testing ggml_vec_dot_f16 ---
Tests Passed 40/40
--- Testing ggml_vec_dot_bf16 ---
Tests Passed 32/32
--- Testing ggml_vec_dot_f16_unroll ---
Tests Passed 64/64
-------------------------------------------------------
```

Kernel Functional Testing

Kernel Function	Test Parameters	Total Tests	Status
ggml_vec_scale_f16	Vector Size, Data Scenarios, Scalar Value	144	**Passed**
ggml_vec_mad_f16	Vector Size, Data Scenarios, Scalar Value	192	**Passed**
ggml_vec_dot_bf16	Vector Size, Data Scenarios	32	**Passed**
ggml_vec_dot_f16_unroll	Vector Size, Data Scenarios, Memory Layout (Stride)	64	**Passed**
ggml_cpu_bf16_to_fp32	Vector Size, Data Scenarios	32	**Passed**
ggml_cpu_fp16_to_fp32	Vector Size, Data Scenarios	32	**Passed**
ggml_vec_silu_f32	Vector Size, Data Scenarios	32	**Passed**

Benchmarking Strategy

Goal: Compare optimized vs. upstream implementation performance

Benchmarking Parameters:

- Vector Sizes: (512, 1024, 2048).
- 64 byte alignment

Methodology:

- Perform a warm-up phase (10 runs) before measurement to make the caches hot.
- Run a high number of iterations (1000) per test to get average.

Performance matrix:

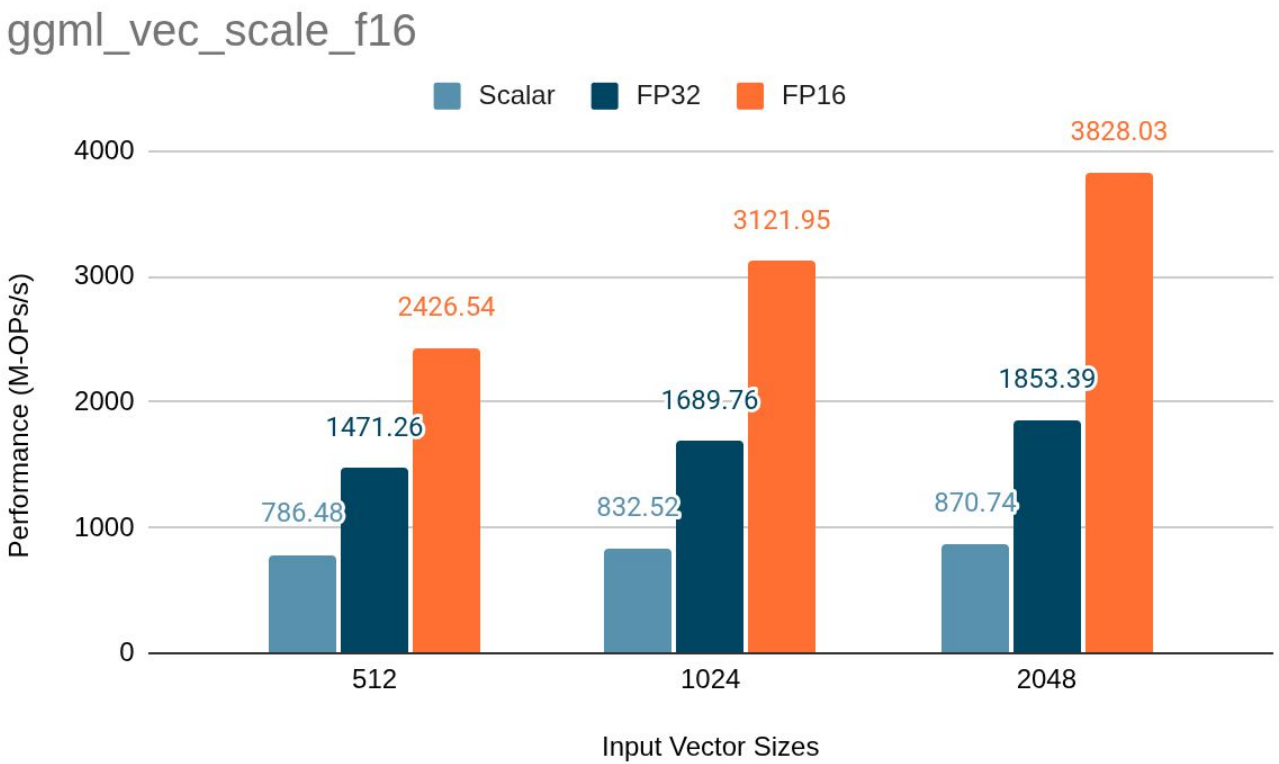
- Throughput (M-Ops/s): Measures the overall processing speed. Higher is better.
  Throughput (M-Ops/s) = Theoretical ops/average time per call

Kernel	operations
`vec_mad`	2 × N
`vec_scale`	N
`vec_dot_unroll`	2 × N × Unroll-factor
`bf16_to_fp32`	N
`fp16_to_fp32`	N
`silu_f32`	33 x N

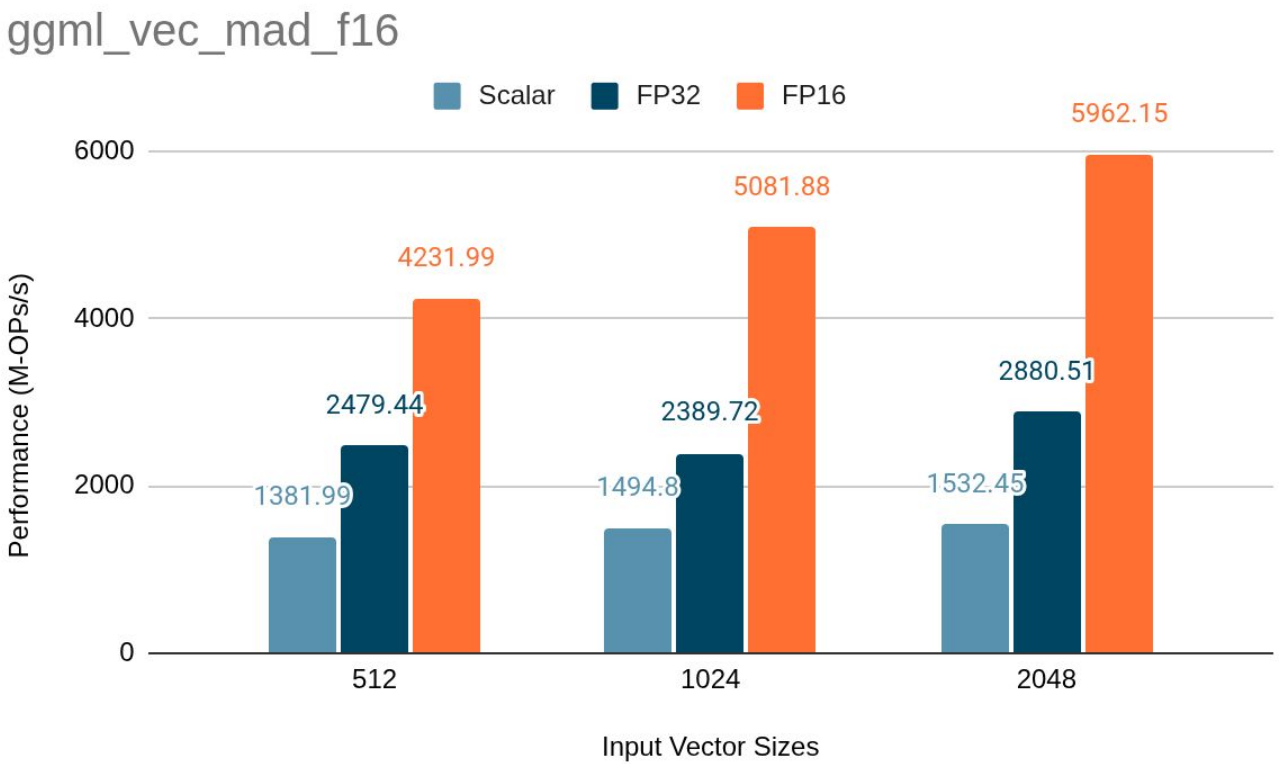
Benchmarking CLI (test-float-perf)

```
(llama.cpp) taimur-ahmad@bpf3-16G-3:~/10x/llama.cpp-10x$ ./release/bin/test-float-perf

Benchmarking Floating Point Kernels (Iterations: 1000, Warmup: 10)
================================================================

--- Kernel: ggml_vec_dot_f16_unroll (unroll=2) ---
  Implementation       |     Size |      M-Ops/s
  ---------------------------------------------------------------
  Reference            |      512 |     670.5959
  Optimized            |      512 |    6095.2381
  Reference            |     1024 |     692.2427
  Optimized            |     1024 |    7613.3829
  Reference            |     2048 |     700.5901
  Optimized            |     2048 |    8687.1686

--- Kernel: ggml_vec_scale_f16 ---
  Implementation       |     Size |      M-Ops/s
  ---------------------------------------------------------------
  Reference            |      512 |     786.4823
  Optimized            |      512 |    1406.5934
  Reference            |     1024 |     844.8845
  Optimized            |     1024 |    1686.9852
  Reference            |     2048 |     848.0331
  Optimized            |     2048 |    1853.3937

--- Kernel: ggml_vec_mad_f16 ---
  Implementation       |     Size |      M-Ops/s
  ---------------------------------------------------------------
  Reference            |      512 |    1402.7397
  Optimized            |      512 |    2415.0943
  Reference            |     1024 |    1476.5681
  Optimized            |     1024 |    2673.6292
  Reference            |     2048 |    1516.4754
  Optimized            |     2048 |    2868.3473
```

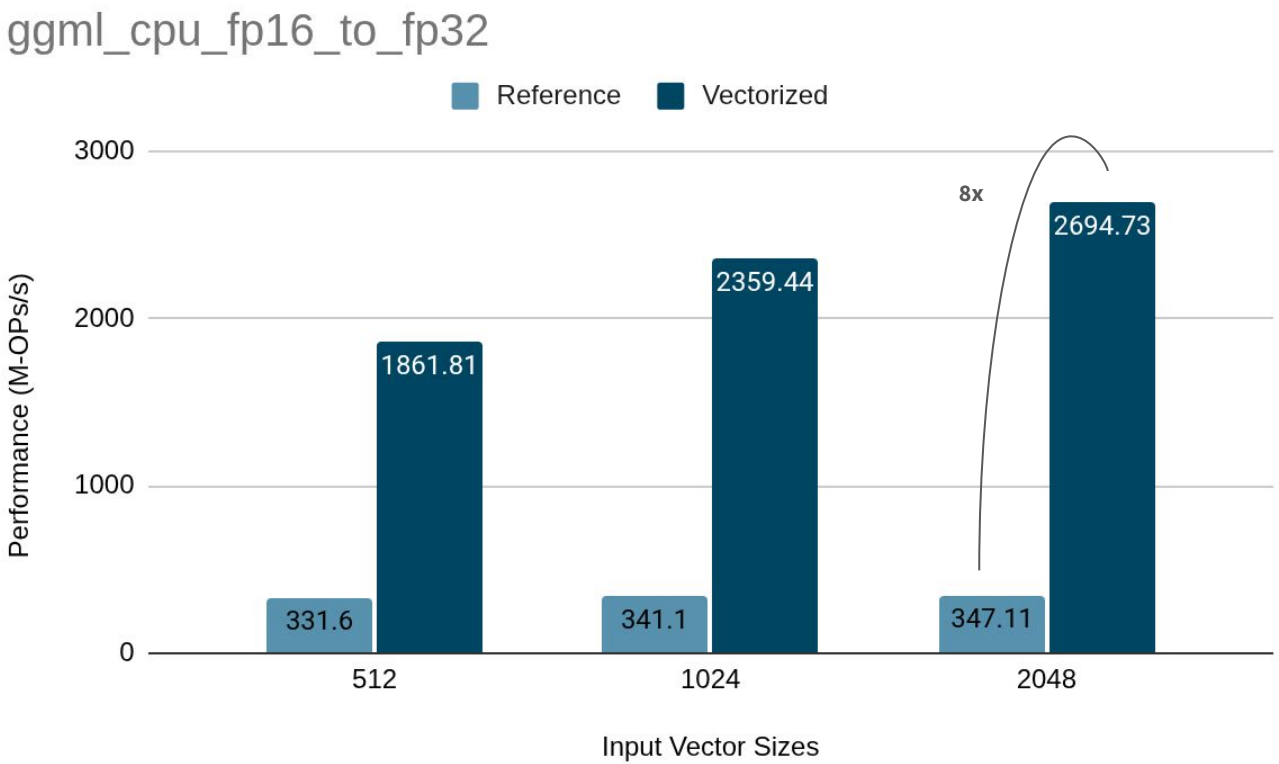
Scaling Kernels (FP16 vs FP32)



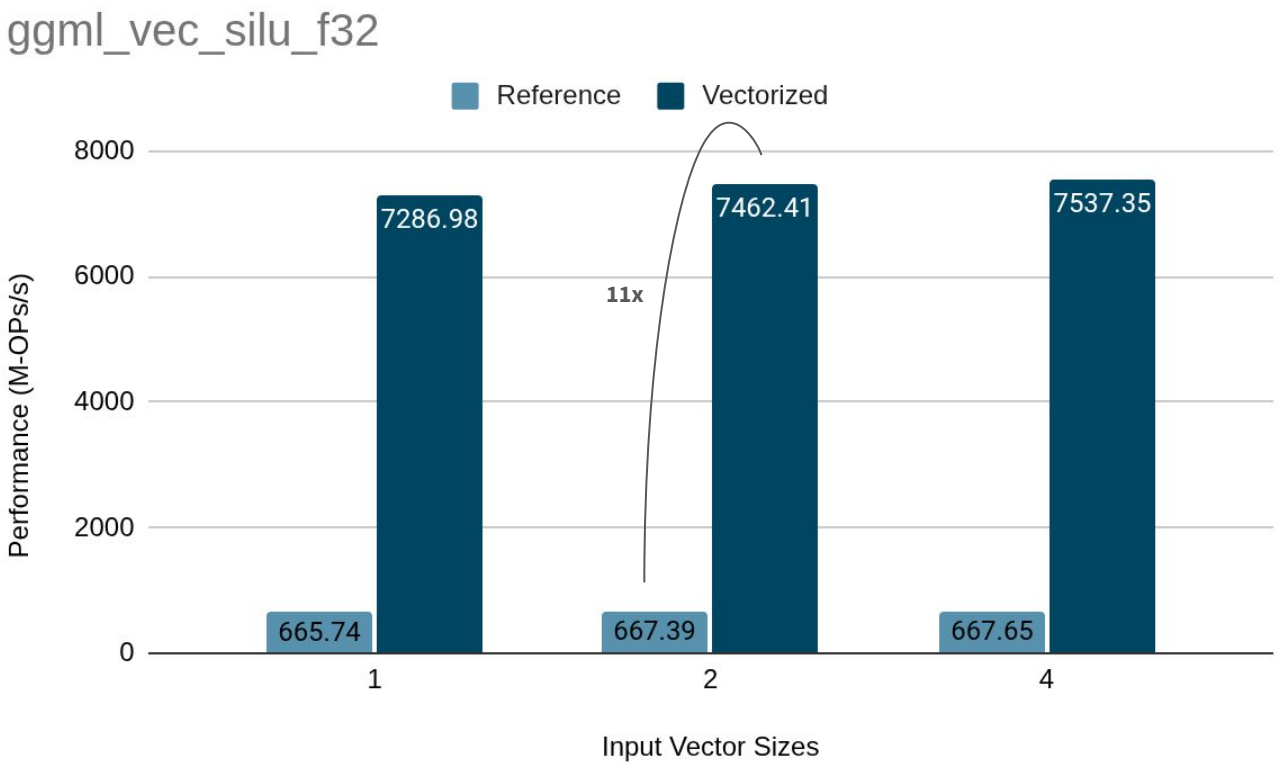
Scaling Kernels (FP16 vs FP32)



Kernel Performance Benchmarking Results



Kernel Performance Benchmarking Results



Scaling Kernels (Tolerance)

Kernel	Accumulation	Avg. Error (against Scalar)
`ggml_mad_f16`	fp16	~0.004f
`ggml_mad_f16`	fp32	<0.001f
`ggml_scale_f16`	fp16	~0.004f
`ggml_scale_f16`	fp32	<0.001f

Kernel Optimizations

- We are primarily focused improving kernel optimizations (preferring BPIF-3) without modifying Llama.cpp's execution flow
- What we can do
  - **Varying LMUL**
  - **Unrolling**
  - **Software Pipelining**
  - **Prefetching**
- Other things we are considering
  - **GCC** v Clang
  - **Compiler optimization:** O2 v **O3**

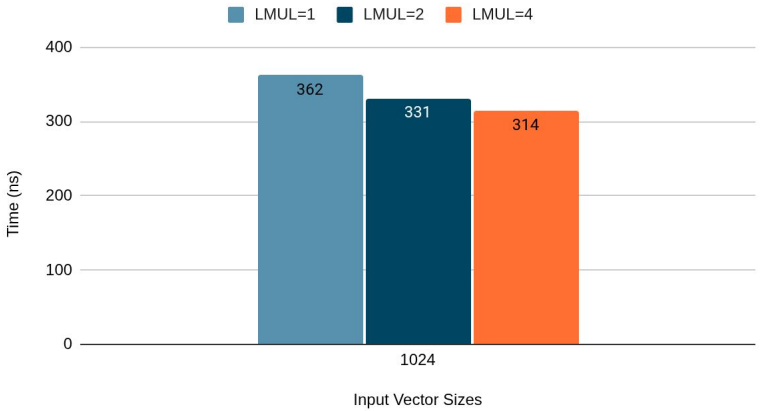
Kernel Performance Statistics (LMUL)

ggml_vec_dot_f16

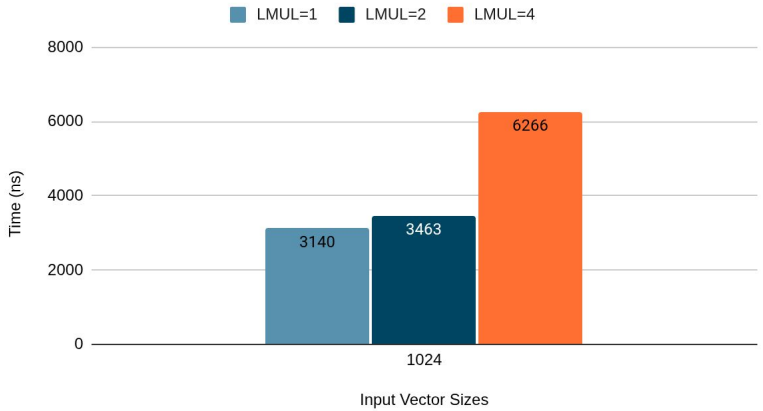
We varied LMUL from 1 to 4

Can not exceed LMUL=4, since this is a widening multiply with accumulation in FP32

Cache Hot

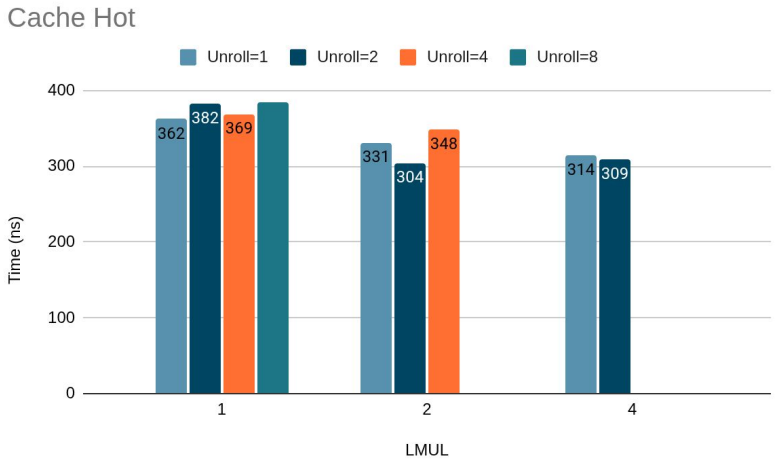


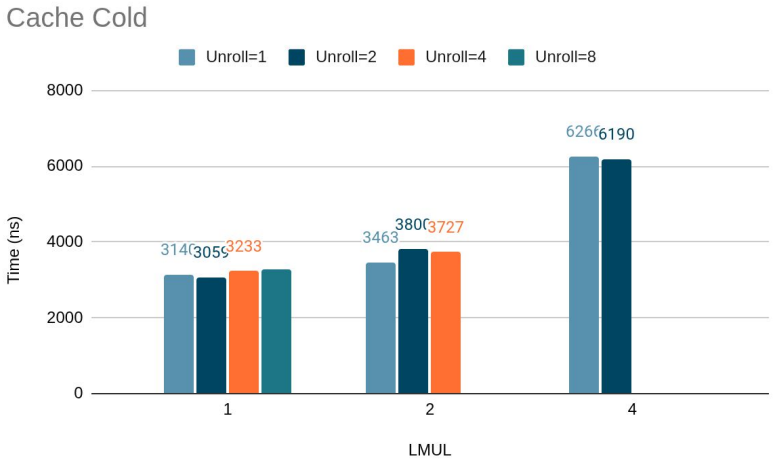
Cache Cold



Kernel Performance Statistics (Unrolling)

ggml_vec_dot_f16





Register Spilling

Unrolling 4 not tested on LMUL=2 and LMUL=4

Unrolling 8 not tested on LMUL=4

Kernel Performance Statistics (Unrolling)

Cache hot (Multiplication Bottleneck)

- LMUL=2, Unroll=2, Reordering=Yes
- **LMUL=4, Unroll=1**
- ***LMUL=4, Unroll=2***

Cache cold (Loading Bottleneck)

- ***LMUL=1, Unroll=2, Reordering=No <- We will choose this***
- LMUL=1, Unroll=1

Kernel Performance Statistics (Software Pipelining)

ggml_vec_dot_f16

- At **O3, GCC** processes all load instructions first and then the macc instructions
- It increases the structural hazards between instructions in the issue window.
- **asm volatile ("" ::: "memory")** acts as a compiler barrier to avoid reorder of memory instructions across it.

```
for (int i = 0; i < n; i += step) {
    vfloat16m1_t ax0 = __riscv_vle16_v_f16m1(&x[i], epr);
    vfloat16m1_t ay0 = __riscv_vle16_v_f16m1(&y[i], epr);
    vsum0 = __riscv_vfwmacc_vv_f32m2(vsum0, ax0, ay0, epr);
    asm volatile ("" ::: "memory");

    vfloat16m1_t ax1 = __riscv_vle16_v_f16m1(&x[i + epr], epr);
    vfloat16m1_t ay1 = __riscv_vle16_v_f16m1(&y[i + epr], epr);
    vsum1 = __riscv_vfwmacc_vv_f32m2(vsum1, ax1, ay1, epr);
    __asm__ __volatile__ ("" ::: "memory");

    vfloat16m1_t ax2 = __riscv_vle16_v_f16m1(&x[i + 2 * epr], epr);
    vfloat16m1_t ay2 = __riscv_vle16_v_f16m1(&y[i + 2 * epr], epr);
    vsum2 = __riscv_vfwmacc_vv_f32m2(vsum2, ax2, ay2, epr);
    __asm__ __volatile__ ("" ::: "memory");

    vfloat16m1_t ax3 = __riscv_vle16_v_f16m1(&x[i + 3 * epr], epr);
    vfloat16m1_t ay3 = __riscv_vle16_v_f16m1(&y[i + 3 * epr], epr);
    vsum3 = __riscv_vfwmacc_vv_f32m2(vsum3, ax3, ay3, epr);
    __asm__ __volatile__ ("" ::: "memory");
}
```

Kernel Performance Statistics (Software Pipelining)

ggml_vec_dot_f16

```
vec_dot_f16_m1_unroll1_memory0:
.LFB56:
    .cfi_startproc
    vsetvli a6,zero,e32,m2,ta,ma
    vmv.v.i v2,0
    ble a1,zero,.L2
    sext.w  t1,a6
    vsetvli zero,zero,e16,m1,ta,ma
    li  a4,0
.L3:
    slli    a5,a4,1
    add a7,a2,a5
    add a5,a3,a5
    vle16.v v4,0(a7)
    vle16.v v1,0(a5)
    addw    a4,a4,t1
    vfwmacc.vv  v2,v4,v1
    bgt a1,a4,.L3
.L2:
    vsetvli a5,zero,e32,m1,ta,ma
    fmv.s.x fa4,zero
    vfadd.vv    v2,v3,v2
    vmv.v.i v1,0
    vfredusum.vs    v2,v2,v1
    vfmv.f.s    fa5,v2
    fadd.s  fa5,fa5,fa4
    fsw fa5,0(a0)
    ret
    .cfi_endproc
```

vec_dot_f16_m1

4460ns

```
.L19:
    add a4,t1,t6
    add a7,a4,a5
    slli    a7,a7,1
    add t4,a2,a7
    add a6,t1,a5
    vle16.v v16,0(t4)
    slli    a4,a4,1
    slli    t3,t1,1
    slli    a6,a6,1
    add a7,a3,a7
    add t4,a2,a4
    vle16.v v15,0(a7)
    add a4,a3,a4
    vle16.v v14,0(t4)
    vle16.v v13,0(a4)
    add a7,a2,a6
    add a4,a2,t3
    vle16.v v12,0(a7)
    add a6,a3,a6
    vle16.v v10,0(a4)
    add t3,a3,t3
    vle16.v v11,0(a6)
    vle16.v v1,0(t3)
    addw    t1,t1,t5
    vfwmacc.vv  v6,v16,v15
    vfwmacc.vv  v4,v14,v13
    vfwmacc.vv  v8,v12,v11
    vfwmacc.vv  v2,v10,v1
    bgt a1,t1,.L19
    vsetvli zero,zero,e32,m2,ta,ma
```

vec_dot_f16_m1_unroll4_memory0

3233ns

```
.L24:
    slli    a4,a6,1
    add a7,a2,a4
    add a4,a3,a4
    vle16.v v10,0(a7)
    vle16.v v1,0(a4)
    vfwmacc.vv  v2,v10,v1
    add a4,a6,a5
    slli    a4,a4,1
    add a7,a2,a4
    add a4,a3,a4
    vle16.v v10,0(a7)
    vle16.v v1,0(a4)
    vfwmacc.vv  v8,v10,v1
    add a4,a6,t4
    slli    a7,a4,1
    add t1,a2,a7
    add a7,a3,a7
    vle16.v v10,0(t1)
    vle16.v v1,0(a7)
    vfwmacc.vv  v4,v10,v1
    add a4,a4,a5
    slli    a4,a4,1
    add a7,a2,a4
    add a4,a3,a4
    vle16.v v10,0(a7)
    vle16.v v1,0(a4)
    vfwmacc.vv  v6,v10,v1
    addw    a6,a6,t3
    bgt a1,a6,.L24
    vsetvli zero,zero,e32,m2,ta,ma
```

vec_dot_f16_m1_unroll4_memory1

Kernel Performance Statistics (Prefetching)

Key Observations:

- The vec-dot kernel accesses data with a fixed stride pattern.
- No measurable performance improvement was observed in our experiments when adding manual/software prefetching.

Next Steps:

- Investigate performance impact of prefetching for
    - Quantized kernels
    - Matrix multiplication (matmul) kernels

Benchmarking Strategy

Goal: Find the most performant kernel

Benchmarking Parameters:

- LMUL Sweeps (LMUL=1 to 8)
- Unrolling factor (1, 2, 4, 8)
- 64 byte alignment
- Cache cold vs hot performance
- Software Pipelining

Methodology:

- Perform a warm-up phase (10 runs) before measurement
- Run a high number of iterations (1000) per kernel
- Find the best time from the iterations
- Repeat for cache cold and cache hot

Performance matrix:

- Best time (ns)

Benchmarking Results (vec_dot_f16)

Kernel	LMUL	Unrolling	Clobbering	Cache Cold (ns)	Cache Hot(ns)
vec_dot_f16_scalar	Auto-Vectorized			3687	2625
vec_dot_f16	1	2	No	1791	333
vec_dot_f16	1	2	Yes	1833	333
vec_dot_f16	1	4	Yes	1999	375
vec_dot_f16	1	8	Yes	1791	375
vec_dot_f16	2	1	No	1791	375
vec_dot_f16	2	2	No	2375	291
vec_dot_f16	**2**	**2**	**Yes**	**1917**	**291**
vec_dot_f16	2	4	Yes	1791	291
vec_dot_f16	4	1	No	3458	291
vec_dot_f16	4	2	Yes	3333	333

Benchmarking Results (vec_mad_f16)

Kernel	LMUL	Unrolling	Clobbering	Cache Cold (ns)	Cache Hot(ns)
vec_mad_f16_scalar	Auto-Vectorized			2583	1416
vec_mad_f16	1	2	No	2124	541
vec_mad_f16	2	1	No	1916	458
vec_mad_f16	2	2	No	2000	416
vec_mad_f16	2	4	No	1833	458
vec_mad_f16	2	4	Yes	3125	458
vec_mad_f16	**4**	**2**	**Yes**	**2250**	**375**
vec_mad_f16	4	4	No	2375	375
vec_mad_f16	4	4	Yes	2417	375
vec_mad_f16	8	1	No	2258	375
vec_mad_f16	8	2	Yes	2458	375
vec_mad_f16	8	2	No	2667	375

Benchmarking Results (vec_scale_f16)

Kernel	LMUL	Unrolling	Clobbering	Cache Cold (ns)	Cache Hot(ns)
`vec_scale_f16_scalar`	Auto-Vectorized			1766	416
`vec_scale_f16`	1	2	Yes	1750	416
`vec_scale_f16`	1	4	Yes	1833	416
`vec_scale_f16`	1	8	Yes	1375	416
`vec_scale_f16`	2	1	No	1584	333
`vec_scale_f16`	2	2	No	1708	334
`vec_scale_f16`	2	2	Yes	1666	333
`vec_scale_f16`	4	2	No	1625	291
`vec_scale_f16`	**4**	**2**	**Yes**	**1500**	**291**
`vec_scale_f16`	4	4	No	1708	291
`vec_scale_f16`	4	4	Yes	1666	291
`vec_scale_f16`	8	2	No	1666	291

Benchmarking Results (vec_dot_f16_unroll)

Kernel	LMUL	Unrolling	Clobbering	Cache Cold (ns)	Cache Hot(ns)
vec_dot_f16_unroll_scalar	Auto-Vectorized			5083	3333
vec_dot_f16_unroll	1	1	No	2833	625
vec_dot_f16_unroll	1	2	No	2958	500
vec_dot_f16_unroll	1	2	Yes	4124	500
vec_dot_f16_unroll	1	4	No	3833	541
vec_dot_f16_unroll	1	4	Yes	4375	500
vec_dot_f16_unroll	2	1	No	2750	500
vec_dot_f16_unroll	**2**	**2**	**No**	**3875**	**458**
vec_dot_f16_unroll	2	2	Yes	4208	458
vec_dot_f16_unroll	4	1	No	3583	500
vec_dot_f16_unroll	4	2	No	5666	1416
vec_dot_f16_unroll	4	2	Yes	6332	1416

Benchmarking Results (cpu_fp16_to_fp32)

Kernel	LMUL	Unrolling	Clobbering	Cache Cold (ns)	Cache Hot(ns)
cpu_fp16_to_fp32_scalar	Auto-Vectorized			2499	708
cpu_fp16_to_fp32	1	2	Yes	2250	541
cpu_fp16_to_fp32	1	4	Yes	2500	541
cpu_fp16_to_fp32	2	1	No	2500	416
cpu_fp16_to_fp32	**2**	**2**	**No**	**2125**	**416**
cpu_fp16_to_fp32	2	2	Yes	2042	458
cpu_fp16_to_fp32	2	4	No	2250	458
cpu_fp16_to_fp32	2	4	Yes	2333	458
cpu_fp16_to_fp32	4	1	No	2416	458
cpu_fp16_to_fp32	4	2	No	2000	458
cpu_fp16_to_fp32	4	4	Yes	2625	458

Benchmarking Results (ggml_vec_silu_f32)

Kernel	LMUL	Unrolling	Clobbering	Cache Cold (ns)	Cache Hot(ns)
`vec_silu_f32_scalar`	Not Auto-Vectorized			55959	54459
`vec_silu_f32`	1	1	N/A	8625	7666
`vec_silu_f32`	**2**	**1**	**N/A**	**7041**	**6083**
`vec_silu_f32`	4	1	N/A	8125	6875