

# High-Performance GEMV on AMD Zen 3 via Architecture-Specific AVX2/FMA Optimization

Rafał Lenart  
0009-0007-8389-7835  
Maria Curie-Skłodowska University  
Lublin, Poland  
Email: rafa.lenart12@gmail.com

Beata Bylina  
0000-0002-1327-9747  
Maria Curie-Skłodowska University  
Lublin, Poland  
Email: beata.bylina@umcs.pl

Jarosław Bylina  
0000-0002-0319-2525  
Maria Curie-Skłodowska University  
Lublin, Poland  
Email: jaroslaw.bylina@umcs.pl

**Abstract**—The General Matrix-Vector Multiply (GEMV) is a fundamental Level-2 BLAS operation appearing throughout scientific computing and numerical linear algebra. GEMV is inherently memory-bound, making effective cache use the dominant optimization factor.

We present a hand-optimized GEMV kernel for AMD Zen 3, implemented using AVX2 (Advanced Vector Extensions 2) and FMA (fused multiply-add) intrinsics. Our single-threaded kernel (ZenGEMV) processes 4 rows simultaneously with dual accumulators per row, creating 8 independent FMA chains that fully saturate the Zen 3 pipeline. Benchmarked against OpenBLAS 0.3.31 and AOCL-BLAS 5.2.0 on an AMD Ryzen 5 5600, ZenGEMV achieves up to  $1.37\times$  speedup over OpenBLAS and  $1.20\times$  over AOCL-BLAS. Our parallel variant (ZenGEMV-P) achieves up to  $3.22\times$  speedup over OpenBLAS on small matrices.

We analyse why dispatch overhead, suboptimal accumulator strategies, and generic design choices in general-purpose BLAS libraries leave performance gains available in architecture-specific workloads.

**Index Terms**—GEMV, BLAS, AVX2, FMA, AMD Zen 3, SIMD, OpenMP

## I. INTRODUCTION

THE Basic Linear Algebra Subprograms (BLAS)[1] are a widely adopted set of standardized interfaces for vector (Level-1), matrix-vector (Level-2), and matrix-matrix (Level-3) operations. Among these, the Level-2 BLAS operation GEMV (General Matrix-Vector Multiplication) computes  $\mathbf{y} \leftarrow \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$ . In computational science, GEMV serves as a core subprogram in a broad range of algorithms, including:

- 1) Dense matrix factorizations such as Householder QR and Hessenberg reduction[2], where each step of the unblocked algorithm applies a Householder reflector via a GEMV on the trailing submatrix.
- 2) Token generation in Large Language Models (LLMs), where each decoding step reduces to a sequence of GEMV operations.[3]
- 3) Recurrent neural network (RNN/LSTM) inference[4], where each time step performs dense matrix-vector multiplications with the weight matrices.

GEMV is fundamentally memory-bound. For an  $M \times N$  matrix, the operation performs  $2MN$  floating-point operations while transferring  $O(MN)$  bytes of matrix data, yielding an operational intensity of approximately 0.25 FLOP/byte—well below the ridge point of modern processors (we derive

this value formally in Section II). This memory-bound nature means that the primary optimization target is not compute throughput but rather effective use of the memory hierarchy and minimization of pipeline stalls.

Production BLAS libraries such as OpenBLAS[5] and AOCL-BLAS[6] are designed for generality: they handle arbitrary strides, all combinations of transpose and conjugate operations, and dynamically select microkernels based on detected hardware. This generality introduces a roughly constant dispatch overhead per call. While this overhead is amortized away for large matrices in both GEMM and GEMV, it becomes significant for tiny and small matrices where the compute-to-overhead ratio is low—a regime in which GEMV is frequently invoked, for example during per-token decoding in LLM inference.

This paper investigates whether a hand-optimized kernel, tailored to a specific microarchitecture and constrained use case, can outperform these general-purpose libraries. Our contributions are:

- 1) An optimized single-threaded GEMV kernel (ZenGEMV) using AVX2 (Advanced Vector Extensions 2)[7] and FMA (fused multiply-add)[8] intrinsics with dual accumulators that fully saturate the Zen 3[9] FMA pipeline.
- 2) A parallel variant (ZenGEMV-P) with row-parallel decomposition using manual thread work division aligned to SIMD boundaries.
- 3) A source-level analysis identifying the concrete cause of lost GEMV performance on Zen 3—most notably OpenBLAS’s reuse of a Haswell microkernel that supplies only 4 of the 8 independent FMA chains the dual-FMA pipeline requires—together with a roofline account of where each library’s throughput is bound.

## II. BACKGROUND AND RELATED WORK

The GEMV operation computes:

$$\mathbf{y} \leftarrow \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y} \quad (1)$$

where  $\mathbf{A} \in \mathbb{R}^{M \times N}$  is a matrix with  $M$  rows and  $N$  columns,  $\mathbf{x} \in \mathbb{R}^N$  is the input vector,  $\mathbf{y} \in \mathbb{R}^M$  is the output vector, and  $\alpha, \beta$  are scalars. The dimensions  $M$  (rows) and  $N$  (columns)

determine both the computational cost and the memory footprint of the operation.

The operation requires  $2MN$  floating-point operations—one multiplication and one addition per element of  $A$ . The total data movement is  $(MN + N + 2M) \times 8$  bytes for double-precision (the matrix  $A$ , input vector  $\mathbf{x}$ , and the output vector  $\mathbf{y}$  read and written), giving an operational intensity[10] of:

$$OI = \frac{2MN}{8(MN + N + 2M)} \approx \frac{1}{4} \text{ FLOPs/byte} \quad (2)$$

for large  $M$  and  $N$ , where the vector terms become negligible relative to the matrix. This makes GEMV memory-bound on all modern architectures.

Our target platform is the AMD Ryzen 5 5600, based on the Zen 3 microarchitecture. The relevant parameters are sourced from the AMD Software Optimization Guide[11] and Agner Fog’s instruction tables[12] and microarchitecture guide[13]:

- **Cache hierarchy:** 32 KiB L1d (8-way, latency 4 clocks) per core, 512 KiB L2 (8-way, latency 14 clocks) per core, 32 MiB shared L3 (16-way, latency 47 clocks).
- **Execution units:** Two 256-bit FMA units capable of executing `vfmadd` instructions.
- **FMA pipeline:** Each `vfmadd231pd` instruction has a 4-cycle latency but the two FMA units can accept a new instruction every cycle (2-per-cycle throughput). Because the pipeline is 4 stages deep, an FMA result is not available for 4 cycles after issue. To avoid stalls, the code must maintain independent operations that can fill the pipeline while earlier results are computed. With 2 units each having 4-cycle latency, the minimum number of independent FMA operations needed for full utilization is  $4 \times 2 = 8$ .
- **Peak throughput:**  $2 \text{ FMA} \times 4 \text{ doubles} \times 2 \text{ ops} \times 4.4 \text{ GHz} = 70.4 \text{ GFLOPS/core}$  at maximum boost clock. This value is the theoretical peak.
- **Memory bandwidth:** The Ryzen 5 5600 uses a DDR4 memory controller. Our test system is equipped with DDR4 3200 MT/s CL16 modules, providing a theoretical peak of 51.2 GB/s in dual-channel mode. This value was calculated using an online tool.[14]

The Roofline model[10] provides an upper bound on attainable performance as a function of a kernel’s operational intensity (OI, the ratio of floating-point operations to bytes transferred) and the system’s memory bandwidth (BW, in GB/s):

$$P_{\max} = \min(P_{\text{peak}}, \text{BW} \times \text{OI}) \quad (3)$$

For GEMV with  $\text{OI} \approx 0.25 \text{ FLOP/byte}$ , the theoretical maximum depends on which level of the memory hierarchy serves the data. The practical bandwidth at each cache level is derived from the sustained transfer width per cycle at the sustained core frequency of 4.4 GHz:

- **L1d:** The L1d cache supports up to two 256-bit (32-byte) loads and one 128-bit (16-byte) load simultaneously which combined with the 4.4 GHz boost clock gives

TABLE I  
BANDWIDTH CEILINGS AND GEMV PERFORMANCE LIMITS PER MEMORY LEVEL (OI = 0.25 FLOP/BYTE).

| Memory Level | Practical BW | GEMV Ceiling | Size   |
|--------------|--------------|--------------|--------|
| L1d          | 267 GB/s     | 66.8 GFLOPS  | 32 KB  |
| L2           | 132 GB/s     | 33.0 GFLOPS  | 512 KB |
| L3           | 106 GB/s     | 26.5 GFLOPS  | 32 MB  |
| DRAM         | 51.2 GB/s    | 12.8 GFLOPS  | —      |

$2 \times 32 \times 4.4 + 1 \times 16 \times 4.4 = 352 \text{ GB/s}$  theoretical throughput; practical throughput is likely less than  $\sim 281.6 \text{ GB/s}$  due to at least one port being usually dedicated to store operations.[11]

- **L2:** The L2 to L1 data path delivers 32 bytes/cycle to L1, giving  $32 \times 4.4 = 140.8 \text{ GB/s}$ .[11]
- **L3:** The per-core L3 read bandwidth was measured empirically using the *bandwidth* tool by Zack Smith [15] on sequential 256-bit reads. Buffer sizes in the L3-resident range (1 MB–16 MB) yielded  $\sim 106 \text{ GB/s}$  sustained read bandwidth. The raw measurement output is shown in Fig. 1.
- **DRAM:** DDR4-3200 dual-channel provides  $3200 \times 8 \times 2 = 51.2 \text{ GB/s}$  theoretical. The *bandwidth* microbenchmark reports  $\sim 42 \text{ GB/s}$  for sequential 256-bit reads from a single core (Fig. 1), but GEMV’s highly regular streaming access pattern enables more effective hardware prefetching; our measured GEMV throughput on DRAM-resident data (Section V) approaches the full 51.2 GB/s ceiling, which we therefore adopt as an approximate practical DRAM bandwidth.

The GEMV ceiling at each level is then computed from Equation (3) as  $\text{BW} \times \text{OI}$  (e.g.,  $267 \times 0.25 = 66.75 \text{ GFLOPS}$  for L1d). Table I summarizes these values.

When the working set fits in cache, the effective bandwidth increases dramatically, enabling much higher throughput for small to medium matrices than the DRAM-bound prediction would suggest. Fig. 2 shows the roofline diagram with our measured ZenGEMV performance for each benchmark preset, demonstrating how cache residency determines the reachable performance level.

#### A. Related Work

**OpenBLAS**[5] descends from GotoBLAS[16] and provides architecture-specific assembly kernels. Its GEMV implementation uses column blocking with `NBMAX=2048`. All ZEN generations share the microkernel `dgemv_t_microk_haswell-4.c`, which processes 4 rows simultaneously with a single accumulator per row.

**AOCL-BLAS** is AMD’s optimized fork of the BLIS[17] framework, decomposes GEMV operations into fused Level-1 microkernels with a configurable fuse factor (detailed in Section IV), and includes a Zen 3 tuned configuration in version 5.2.0.

The ATLAS project[18] pioneered empirical auto-tuning for BLAS, searching blocking parameters and code transforma-

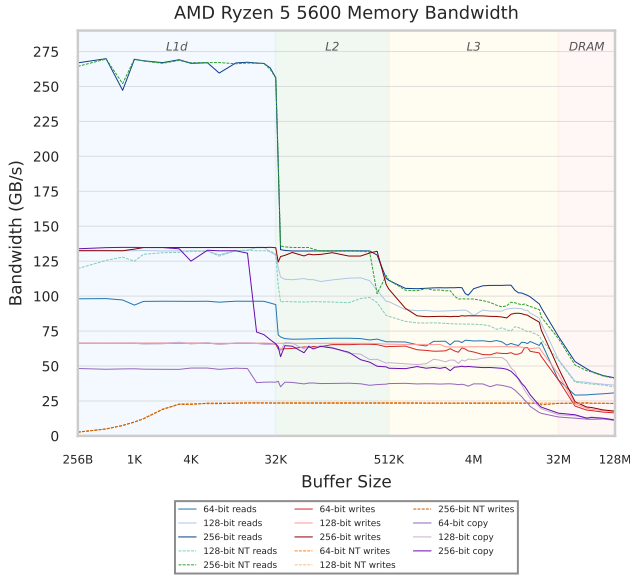


Fig. 1. Memory bandwidth measured with the *bandwidth* tool [15] on the AMD Ryzen 5 5600 for sequential reads, writes, nontemporal (NT) operations, and copies at 64/128/256-bit widths. The shaded regions mark cache levels: L1d (~267 GB/s peak read), L2 (~132 GB/s), L3 (~106 GB/s), and DRAM (~42 GB/s).

tions to maximise cache efficiency. Hassan[19] demonstrates that hand-written AVX SGEMV kernels can outperform Intel MKL on multicore x86 processors; Fog[12], [13] documents detailed instruction latencies and throughput for x86 microarchitectures, which informed our kernel design.

On the GPU side, Abdelfattah et al.[20] present KBLAS, an optimized library for dense matrix–vector multiplication on NVIDIA GPUs. Their double-buffering technique overlaps data movement with computation, and collaborative thread-block processing with atomic operations yields up to 60% speedup over cuBLAS. A subset of the KBLAS kernels have been integrated into NVIDIA’s cuBLAS.

Liang and Zhang[21] specifically target Intel AVX processors, combining prefetching strategies, AVX intrinsics with loop unrolling, and OpenMP parallelization to achieve 5–10% improvements over Intel MKL, GotoBLAS, and ATLAS. Their work is directly comparable to ours, but targets an Intel microarchitecture rather than AMD Zen 3.

Singh and Basso[22] take a complementary, portable approach: their cache-blocked GEMV algorithms exploit L1/L2 blocking and FMA instructions without hand-written assembly, relying on compiler optimizations to remain competitive with MKL, BLIS, and OpenBLAS across multiple x86 platforms. In contrast, our work deliberately specializes the microkernel for AMD Zen 3, trading portability for higher utilization of the specific pipeline resources. Cache- and vectorization-aware optimization of dense linear algebra has also been studied at FedCSIS, with attention to energy consumption: Bylina et al. examine how loop transformations and storage order affect both the performance and energy of dense

matrix factorization on multicore x86[23], while Jakobs and R nger measure the performance and energy cost of the AVX load/store variants that dominate memory-bound kernels[24].

### III. IMPLEMENTATION

The ZenGEMV kernel processes 4 rows of the matrix simultaneously in the inner loop. This design is motivated by two key constraints of the Zen 3 FMA pipeline:

- 1) **FMA latency hiding:** Each `vfmadd231pd` instruction has 4-cycle latency[12]. With 2 FMA units executing per cycle, we need  $4 \times 2 = 8$  independent FMA operations to keep both units fully occupied.
- 2) **reusing vector x:** Loading the `x` vector once and applying it to 4 rows reduces memory traffic.

For each of the 4 rows, we maintain two independent accumulator registers (`sum_a` and `sum_b`).

The main loop processes 16 elements per iteration (4 AVX2 loads of 4 doubles each), alternating between accumulators `a` and `b`:

After the main loop, the dual accumulators are combined, and cleanup loops handle remaining elements in groups of 8, 4, and finally scalar elements. A horizontal sum reduction (`hsum_pd`) converts each 256-bit accumulator to a scalar result using `vextractf128` and `vhaddpd`[7].

Compiler hints including GCC’s `always_inline` and `hot` attributes are used to ensure the kernel is inlined at the call site and marked as performance-critical for the compiler’s optimization heuristics. The `__restrict` qualifier on all pointer parameters enables the compiler to assume no aliasing.

The compilation flags `-O3 -march=znver3 -mfma -mavx2 -flto`[25] enable architecture-specific code generation and link-time optimization. We deliberately omit `-ffast-math`: the kernel is hand-vectorized with intrinsics, so the compiler has no scope to apply unsafe reassociations, and we measured no observable performance change from enabling it.

The ZenGEMV-P variant adds multi-threaded execution using OpenMP[26] with a row-parallel decomposition strategy. Rather than using `#pragma omp parallel for`, which incurs per-iteration scheduling overhead, we use a single `#pragma omp parallel` region with manual work division:

Key design decisions:

- **4-row alignment:** Thread boundaries are aligned to 4-row groups, ensuring each thread uses the optimized 4-row kernel without partial processing at boundaries.
- **No false sharing:** Each thread writes to a disjoint region of `y`, eliminating cache-line contention.
- **Adaptive thresholds:** Parallelization is only applied when the matrix has at least 128 rows and between 64K and 8M elements. Below the lower threshold, thread creation overhead dominates. Above the upper threshold, DRAM bandwidth saturation makes additional threads counterproductive—measurements confirm that serial ZenGEMV outperforms the parallel variant in this

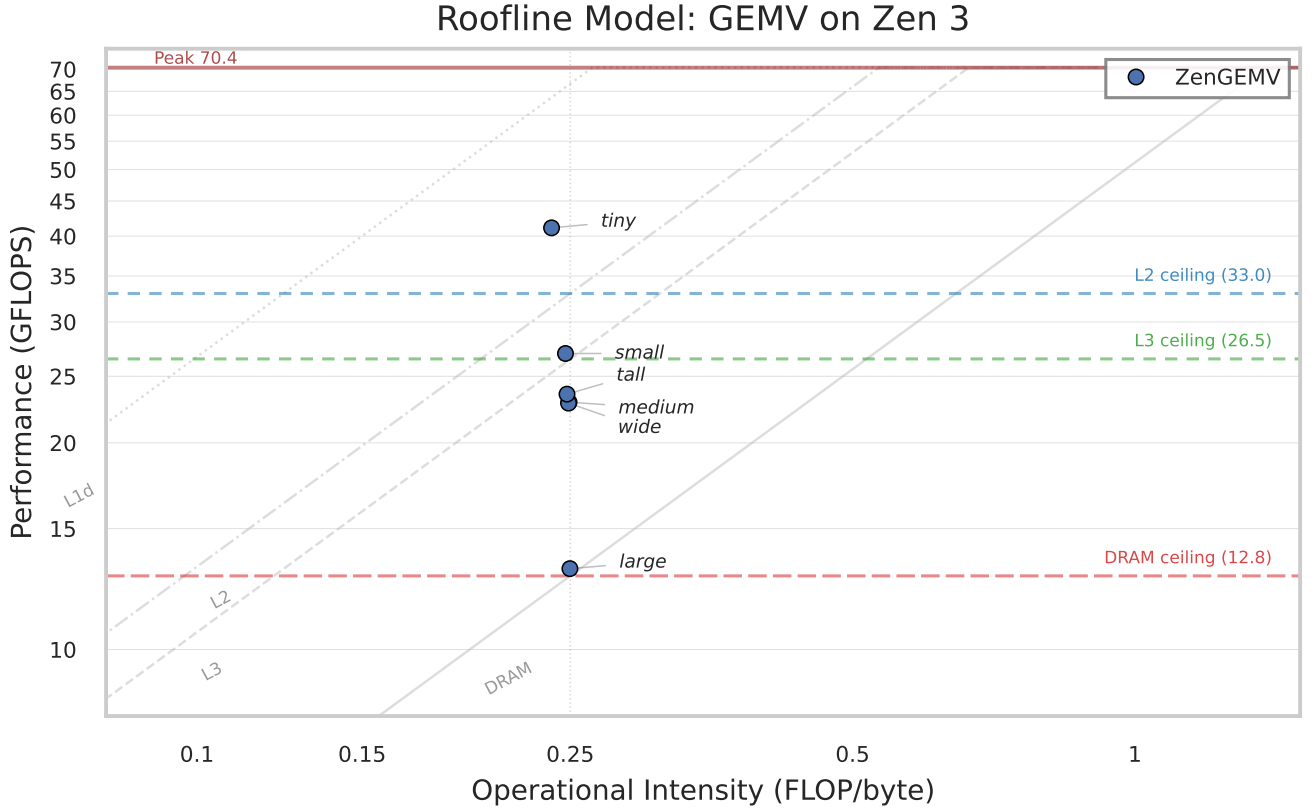


Fig. 2. Roofline model for Zen 3. Diagonal lines show memory bandwidth ceilings from DRAM through L1d; horizontal colored lines mark the GEMV-specific performance ceilings at each cache level ( $BW \times 0.25$ ). Points show measured ZenGEMV performance at each preset's operational intensity, illustrating how cache residency determines the attainable performance level.

```

1 for (; j + 15 < cols; j += 16) {
2   __m256d x0 = __mm256_load_pd(&x[j]);
3   __m256d x1 = __mm256_load_pd(&x[j + 4]);
4   __m256d x2 = __mm256_load_pd(&x[j + 8]);
5   __m256d x3 = __mm256_load_pd(&x[j + 12]);
6
7   sum0a = __mm256_fmadd_pd(
8     __mm256_load_pd(&a0[j]),      x0, sum0a);
9   sum0b = __mm256_fmadd_pd(
10    __mm256_load_pd(&a0[j + 4]),   x1, sum0b);
11   sum0a = __mm256_fmadd_pd(
12    __mm256_load_pd(&a0[j + 8]),   x2, sum0a);
13   sum0b = __mm256_fmadd_pd(
14    __mm256_load_pd(&a0[j + 12]),  x3, sum0b);
15   // ... rows 1-3 follow the same pattern
16 }

```

Fig. 3. Inner loop of ZenGEMV (abbreviated, showing 1 of 4 rows).

regime, consistent with DRAM bandwidth saturation predicted by the roofline model (Fig. 2).

#### IV. ANALYSIS OF LIBRARY OVERHEAD

OpenBLAS GEMV processing involves several layers of overhead before reaching the kernel:

- 1) **CBLAS dispatch:** `cblas_dgemv` validates the memory layout (`CblasRowMajor/CblasColMajor`) and

```

1 #pragma omp parallel
2 {
3   int nt = omp_get_num_threads();
4   int tid = omp_get_thread_num();
5   int total_chunks = (rows + 3) / 4;
6   int chunks_per = total_chunks / nt;
7   int extra = total_chunks % nt;
8   int start = tid * chunks_per + min(tid, extra);
9   int end = start + chunks_per + (tid < extra);
10  process_row_range(start * 4, min(end * 4, rows),
11                    cols, alpha, A, x, y);
12 }

```

Fig. 4. Thread work division in ZenGEMV-P.

transpose flag (`NoTrans/Trans/ConjTrans`), swapping  $M/N$  and the transpose sense before forwarding to the Fortran-style `dgemv` entry point.

- 2) **Column blocking:** The implementation splits columns into blocks of  $NBMAX=2048$ , performing a horizontal reduction after each block. For matrices with  $N > 2048$ , this introduces additional reduction overhead and breaks the accumulation across blocks.
- 3) **Accumulator strategy:** OpenBLAS compiles all ZEN generations (ZEN through ZEN4) using the same shared microkernel `dgemv_t_microk_haswell-4.c`,

as determined by the preprocessor block in `kernel/x86_64/dgemv_t_4.c`:

```
#if defined(HASWELL) || defined(ZEN) || ...
#include "dgemv_t_microk_haswell-4.c"
#endif
```

This kernel processes 4 rows simultaneously with a single accumulator per row, providing only 4 independent FMA chains. On Zen 3, 8 chains are needed for full utilization—this kernel leaves half the pipeline capacity unused.

AOCL-BLAS uses an object-based design that provides clean abstractions at a performance cost:

- 1) **Object dispatch:** `bli_dgemv` constructs `obj_t` structures (AOCL-BLAS’s internal matrix/vector descriptors carrying dimensions, strides, and data pointers) and queries the hardware context (`cntx_t`—a structure storing function pointers to architecture-specific micro-kernels along with cache and register blocking parameters for the target platform).
- 2) **Kernel abstraction:** AOCL-BLAS decomposes the GEMV operation into fused Level-1 primitives rather than implementing it monolithically. For the non-transpose case (`bli_gemv_unf_var2`), it uses `AXPYF`—a fused `axpy` that performs  $f$  scaled vector additions in a single pass, with fuse factor `b_fuse=BLIS_AF = 5` on Zen 3. For the transpose case (`bli_gemv_unf_var1`), it uses `DOTXF`—a fused dot product that simultaneously computes the dot product of  $x$  against  $f$  matrix rows, with `b_fuse=BLIS_DF = 8` on Zen 3. While this modularity enables code reuse across different BLAS operations, it prevents optimizations that span the entirety of the GEMV operation.
- 3) **Per-call initialization:** Every entry through the BLAS compatibility layer invokes `bli_init_auto()`—a library initialization guard—followed by parameter mapping (`bli_param_map_netlib_to_blis_trans`), dimension extraction, and stride setup before any computation begins. The kernel itself is invoked through a function pointer retrieved from `cntx` at runtime, adding an indirect call on every GEMV invocation.

Despite these overheads, general-purpose libraries have advantages in certain scenarios:

- **AOCL-BLAS on medium (1T):** AOCL-BLAS achieves 23.72 GFLOPS vs. our 22.92 GFLOPS on  $1024 \times 1024$ . The AOCL-BLAS Zen 3 kernel configuration is well-tuned for L3-resident data, and its grouping strategies may provide better cache utilization at this specific size.
- **Multi-threaded medium/tall presets:** OpenBLAS and AOCL-BLAS achieve competitive or slightly better multi-threaded performance on medium and tall presets, where the larger amount of work per call amortizes their thread-dispatching overhead.

TABLE II

BENCHMARK PRESETS WITH WORKING SET SIZES AND CACHE RESIDENCY. WORKING SET COMPUTED AS  $(MN + N + 2M) \times 8$  BYTES.

| Preset | $M$  | $N$  | Working Set | Cache Level |
|--------|------|------|-------------|-------------|
| Tiny   | 64   | 64   | 33.5 KB     | L1d         |
| Small  | 256  | 256  | 518.0 KB    | L2          |
| Medium | 1024 | 1024 | 8.02 MB     | L3          |
| Large  | 4096 | 4096 | 128.1 MB    | DRAM        |
| Wide   | 256  | 8192 | 16.1 MB     | L3/DRAM     |
| Tall   | 8192 | 256  | 16.1 MB     | L3/DRAM     |

TABLE III

SINGLE-THREADED GEMV PERFORMANCE (MEDIAN GFLOPS, 5 RUNS). SPEEDUP COLUMNS SHOW ZENGEMV DIVIDED BY THE RESPECTIVE LIBRARY’S THROUGHPUT.

| Preset | Zen-GEMV | Open-BLAS | AOCL-BLAS | Speedup |        |
|--------|----------|-----------|-----------|---------|--------|
|        |          |           |           | vs. OB  | vs. AB |
| Tiny   | 41.13    | 29.96     | 34.37     | 1.37×   | 1.20×  |
| Small  | 26.99    | 23.65     | 23.12     | 1.14×   | 1.17×  |
| Medium | 22.92    | 22.79     | 23.72     | 1.01×   | 0.97×  |
| Large  | 13.11    | 11.71     | 12.32     | 1.12×   | 1.06×  |
| Wide   | 22.85    | 23.69     | 17.38     | 0.96×   | 1.31×  |
| Tall   | 23.55    | 22.30     | 22.78     | 1.06×   | 1.03×  |

- **Generality:** Libraries handle arbitrary strides, non-contiguous memory layouts, complex alpha/beta combinations, and all transpose variants—features our kernel does not support.

## V. EXPERIMENTAL EVALUATION

All experiments were conducted on an AMD Ryzen 5 5600 (6 cores, 12 threads, Zen 3, base clock 3.5 GHz, boost up to 4.47 GHz) running Arch Linux with kernel 6.18.7-zen1-1-zen, with the following configuration:

- **Clock source:** `CLOCK_MONOTONIC_RAW` to avoid NTP adjustments.
- **Measurement:** 5 runs per configuration, reporting median GFLOPS with standard deviation, minimum, and maximum.
- **Core pinning:** Single-threaded tests pinned to core 0 via `taskset -c 0`; multi-threaded tests pinned to physical cores 0–5 via `taskset -c 0–5`.
- **Compiler:** GCC 15.2.1 with `-O3 -march=znver3 -mfma -mavx2 -flto`.
- **Libraries:** OpenBLAS 0.3.31 and AOCL-BLAS 5.2.0, both built from source with the same compiler and the same optimization flags as our kernel (AOCL-BLAS additionally configured with `-t openmp` and the `zen3` target).

Six benchmark presets cover different cache residency scenarios, summarized in Table II.

Fig. 5 shows single-threaded performance across all presets. Table III provides detailed numerical results. The speedup columns show the ratio of ZenGEMV performance to each library.

The results reveal a clear pattern tied to cache residency:

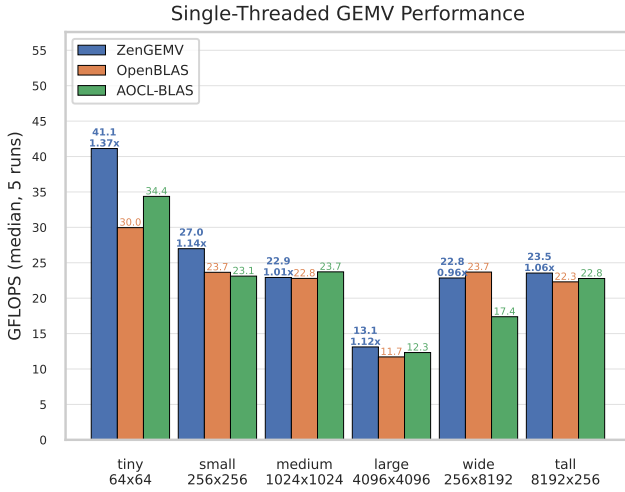


Fig. 5. Single-threaded GEMV performance comparison. Annotations above ZenGEMV bars show throughput (GFLOPS) and speedup relative to OpenBLAS.

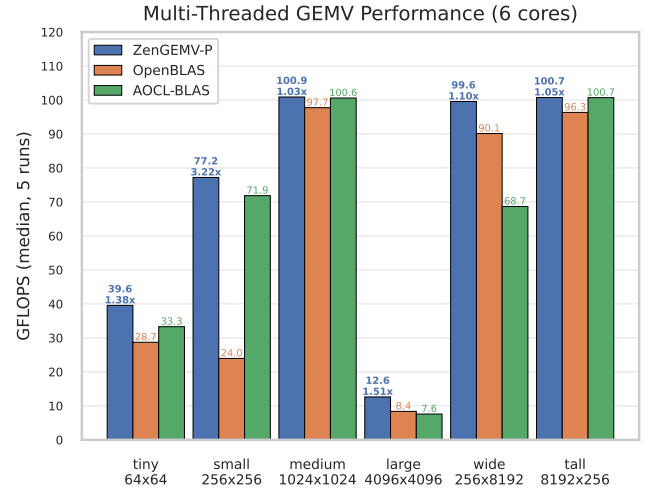


Fig. 6. Multi-threaded GEMV performance (6 physical cores). Annotations above ZenGEMV-P bars show throughput (GFLOPS) and speedup relative to OpenBLAS.

TABLE IV  
MULTI-THREADED GEMV PERFORMANCE (MEDIAN GFLOPS, 6 CORES, 5 RUNS). SPEEDUP COLUMNS SHOW ZENGEMV-P DIVIDED BY EACH LIBRARY.

| Preset | Zen-GEMV-P | Open-BLAS | AOCL-BLAS | Speedup |        |
|--------|------------|-----------|-----------|---------|--------|
|        |            |           |           | vs. OB  | vs. AB |
| Tiny   | 39.57      | 28.71     | 33.30     | 1.38×   | 1.19×  |
| Small  | 77.20      | 23.96     | 71.85     | 3.22×   | 1.07×  |
| Medium | 100.88     | 97.74     | 100.58    | 1.03×   | 1.00×  |
| Large  | 12.62      | 8.38      | 7.59      | 1.51×   | 1.66×  |
| Wide   | 99.56      | 90.13     | 68.66     | 1.10×   | 1.45×  |
| Tall   | 100.73     | 96.35     | 100.69    | 1.05×   | 1.00×  |

**Tiny (L1-resident):** ZenGEMV achieves 41.13 GFLOPS, a 1.37 $\times$  speedup over OpenBLAS. At this size, the matrix, vector, and result all fit in L1d cache, making dispatch overhead significant relative to computation time. ZenGEMV’s direct function call with inlined kernel eliminates this overhead entirely.

**Small (L2-resident):** ZenGEMV achieves 26.99 GFLOPS (1.14 $\times$  over OpenBLAS, 1.17 $\times$  over AOCL-BLAS). At this size, dispatch overhead is a smaller fraction of total execution time, bringing results from all implementations closer together.

**Medium (L3-resident):** AOCL-BLAS achieves the highest throughput at 23.72 vs. 22.92 GFLOPS for ZenGEMV. The AOCL-BLAS Zen 3 configuration appears better suited to this cache level, outperforming both OpenBLAS and ZenGEMV.

**Large (DRAM-bound):** All implementations converge near 12–13 GFLOPS, consistent with the DRAM roofline prediction of  $\sim 12.8$  GFLOPS (Table I). At this point, memory bandwidth is the bottleneck and kernel optimizations yield diminishing returns.

Fig. 6 shows multi-threaded performance with 6 physical cores. Table IV provides numerical details.

Key observations:

**Tiny and small presets:** OpenBLAS deliberately runs single-threaded for  $MN < 115,200 \times T$  (where  $T = 4$  is its compile-time multithread threshold, giving a cutoff of 460,800). Both the tiny ( $MN = 4,096$ ) and small ( $MN = 65,536$ ) presets fall well below this limit. The rationale is that for small matrices it is more efficient to issue multiple independent GEMV calls concurrently at the application level than to parallelize within a single call. ZenGEMV-P and AOCL-BLAS both choose to parallelize within the call: on the small preset ZenGEMV-P reaches 77.20 GFLOPS (2.86 $\times$  over its single-threaded result) versus AOCL-BLAS at 71.85 GFLOPS (3.11 $\times$  over its 23.12 GFLOPS), a 1.07 $\times$  speedup for ZenGEMV-P.

**Large preset (serial fallback):** ZenGEMV-P falls back to serial execution for matrices exceeding 8M elements, achieving 12.62 GFLOPS. Both OpenBLAS (8.38) and AOCL-BLAS (7.59) attempt parallelization but suffer from memory controller contention, performing significantly worse than the serial case.

**Medium/Wide/Tall:** These L3-resident sizes show competitive performance across all implementations. ZenGEMV-P leads at medium (1.03 $\times$  over OpenBLAS, 1.00 $\times$  over AOCL-BLAS), wide (1.10 $\times$ , 1.45 $\times$ ), and tall (1.05 $\times$ , 1.00 $\times$ ).

Table V presents a bandwidth utilization analysis across all three implementations. For each preset, the achieved memory bandwidth is derived from the roofline identity  $BW_{\text{achieved}} = P_{\text{measured}}/OI$ , where  $P_{\text{measured}}$  is the observed throughput in GFLOPS and OI is the operational intensity from Equation (2). The utilization percentage is the ratio of achieved bandwidth to the practical ceiling at the dominant cache level.

**Bandwidth efficiency across cache levels.** ZenGEMV’s bandwidth utilization increases with cache level depth. At L1d (tiny), only 64.5% of the 267 GB/s ceiling is reached. The gap is attributable to a couple factors. L1d bandwidth is shared

TABLE V  
ACHIEVED MEMORY BANDWIDTH AND UTILIZATION EFFICIENCY FOR SINGLE-THREADED GEMV. ACHIEVED BW DERIVED AS  $P_{\text{MEASURED}} / \text{OI}$ ; UTILIZATION IS THE FRACTION OF THE PRACTICAL BANDWIDTH CEILING AT THE DOMINANT CACHE LEVEL.

| Preset | Level | Ceil.<br>(GB/s) | ZenGEMV |       | OpenBLAS |      | AOCL-BLAS |      |
|--------|-------|-----------------|---------|-------|----------|------|-----------|------|
|        |       |                 | GB/s    | %     | GB/s     | %    | GB/s      | %    |
| Tiny   | L1d   | 267             | 172.2   | 64.5  | 125.5    | 47.0 | 143.9     | 53.9 |
| Small  | L2    | 132             | 109.2   | 82.7  | 95.7     | 72.5 | 93.5      | 70.9 |
| Medium | L3    | 106             | 91.9    | 86.7  | 91.4     | 86.2 | 95.1      | 89.7 |
| Large  | DRAM  | 51.2            | 52.5    | 102.5 | 46.9     | 91.6 | 49.3      | 96.3 |
| Wide   | L3    | 106             | 91.8    | 86.6  | 95.1     | 89.7 | 69.8      | 65.8 |
| Tall   | L3    | 106             | 94.9    | 89.5  | 89.9     | 84.8 | 91.8      | 86.6 |

between loads (matrix rows and vector  $\mathbf{x}$ ) and stores (output  $\mathbf{y}$ ), so the two load ports cannot sustain full read bandwidth simultaneously with writes. The `vhaddpd/vextractf128` sequence that collapses each 256-bit accumulator to a scalar result is executed once per 4-row group, and at  $64 \times 64$ , the inner loop body is only 16 iterations per row group, making loop control and cleanup a non-negligible fraction of total execution time. At L2 (small), utilization rises to 82.7% as the inner loop count grows to 64 iterations, reducing relative overhead. The L3-resident presets (medium, wide, tall) achieve 86.6–89.5% utilization—consistent across three different aspect ratios, confirming that the kernel’s sequential row-major streaming pattern is well-matched to the L3 subsystem regardless of matrix shape. The large (DRAM) preset measures 102.5% of the 51.2 GB/s ceiling. Since this marginally exceeds the theoretical figure, the result should be read as the kernel saturating the DDR4-3200 memory controller to within measurement tolerance, with residual L3 reuse across the 5 timed iterations (the  $4096 \times 4096$  matrix at 128 MB exceeds the 32 MB L3, yet a fraction remains resident between iterations) a plausible source of the small excess.

**Cross-implementation comparison.** At L1d, ZenGEMV’s 64.5% substantially exceeds OpenBLAS (47.0%) and AOCL-BLAS (53.9%), confirming that dispatch overhead consumes a disproportionate fraction of the available bandwidth window at small problem sizes. At the L3 level, all three implementations converge to 84–90%, with AOCL-BLAS slightly leading on the medium preset (89.7% vs. 86.7%). AOCL-BLAS’s low 65.8% on the wide preset ( $256 \times 8192$ ) stems from its transpose (DOTXF) code path (`bli_gemv_unf_var1`), which fuses `BLIS_DF = 8` dot products and so traverses 8 columns concurrently while reusing  $\mathbf{x}$  (of length  $N$ ). At  $N = 8192$ ,  $\mathbf{x}$  (64 KB) overflows L1—the only L3-resident preset for which it does—and the eight 64 KB-strided streams are served less efficiently than ZenGEMV’s four-stream, row-major pattern, which sustains 86.6% at the same shape. At DRAM, the performance difference becomes 13.1 vs. 11.7 vs. 12.3 GFLOPS, consistent with all implementations approaching the same physical bandwidth limit—kernel optimizations yield lesser returns once DRAM bandwidth is the binding constraint.

**Roofline model validation.** The bandwidth utilization data confirms that GEMV is memory-bound across all preset sizes, as predicted by its operational intensity of

$\sim 0.25$  FLOP/byte—well below the ridge point of the Zen 3 roofline at  $\sim 1.37$  FLOP/byte (L3) to  $\sim 2.67$  FLOP/byte (L1d). The roofline model correctly predicts the performance hierarchy across cache levels and provides a useful upper-bound estimate: predictions are most accurate for L3/DRAM-resident data (86–102% of ceiling) and least accurate for L1-resident data (47–65%), where problems not captured by the simple model—port contention, reduction overhead, loop control—become significant.

## VI. DISCUSSION AND FUTURE WORK

Our results demonstrate that architecture-specific kernel optimization yields measurable performance improvements over general-purpose BLAS libraries for Level-2 operations where dispatch overhead is non-negligible relative to compute time. Several avenues for further improvement remain:

**Custom kernels at small sizes:** Production BLAS libraries avoid intra-call parallelism for small matrices, assuming the application will issue multiple calls concurrently. When this assumption does not hold, a hand-optimized kernel is the natural solution: it eliminates dispatch overhead and can exploit all available threads without requiring application-level batching. Our results on the tiny and small presets confirm that dispatch overhead savings alone yield  $1.40\times$  and  $1.08\times$  speedups over OpenBLAS in the single-threaded case.

**Assembly implementation:** While intrinsics provide portability, the compiler still makes register allocation and scheduling decisions that may be suboptimal. A hand-written assembly kernel could precisely control instruction ordering and register usage, potentially closing the gap to the roofline on L1-resident data.

**Shape-adaptive dispatch:** Our kernel uses the same strategy for all matrix shapes. A runtime dispatch based on the aspect ratio (wide vs. tall vs. square) could select between row-parallel and column-parallel decomposition strategies, potentially improving cache utilization for extreme aspect ratios.

**Application-level impact:** The present evaluation isolates GEMV at the kernel level; measuring the kernel within a complete application where GEMV dominates runtime would quantify how kernel-level gains translate into overall speedups.

It should be noted that the current implementation imposes constraints not present in general-purpose libraries: row-major layout, 32-byte aligned memory, and unit alpha/beta scalars. Extending support to arbitrary strides and memory layouts

would require additional code paths that may reduce the observed performance advantage.

## VII. CONCLUSION

We have presented ZenGEMV and ZenGEMV-P, optimized GEMV kernels for the AMD Zen 3 microarchitecture that outperform OpenBLAS and AOCL-BLAS across most benchmark configurations. The key insight is that Zen 3's dual FMA units with 4-cycle latency require 8 independent operations in flight, achieved through our 4-row  $\times$  2-accumulator design. This provides strictly higher pipeline utilization than OpenBLAS's 4-chain Haswell microkernel.

In single-threaded mode, ZenGEMV reaches up to  $1.37\times$  over OpenBLAS and  $1.20\times$  over AOCL-BLAS, with the largest advantage on L1-resident data where dispatch overhead dominates. In multi-threaded mode, ZenGEMV-P scales efficiently across all 6 cores and reaches a  $3.22\times$  advantage over OpenBLAS on the small preset where OpenBLAS deliberately serializes.

These results indicate that for fixed-platform deployments in high-performance and computational science contexts where GEMV constitutes a performance-critical operation, architecture-specific hand optimization remains a viable and productive strategy, even alongside mature general-purpose libraries. Future work includes a hand-written assembly kernel to close the remaining gap to the roofline ceiling, shape-adaptive dispatch for non-square matrices, extended support for arbitrary memory layouts and scaling parameters, and an evaluation of the kernel within a complete application where GEMV dominates runtime.

## REFERENCES

- [1] L. S. Blackford, A. Petitet, R. Pozo, et al., "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, 2002. <https://doi.org/10.1145/567806.567807>
- [2] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. Baltimore: Johns Hopkins University Press, 2013.
- [3] C. He, Y. Huang, P. Mu, Z. Miao, J. Xue, L. Ma, et al., "WaferLLM: Large language model inference at wafer scale," *arXiv:2502.04563 [cs.LG]*, 2025. <https://doi.org/10.48550/arXiv.2502.04563>
- [4] C. Holmes, D. Mawhirter, Y. He, F. Yan, and B. Wu, "GRNN: Low-latency and scalable RNN inference on GPUs," in *Proc. 14th EuroSys Conf. (EuroSys'19)*, New York: ACM, 2019, pp. 41:1–41:16. <https://doi.org/10.1145/3302424.3303949>
- [5] OpenBLAS: An optimized BLAS library. <https://www.openblas.net>, <https://github.com/OpenMathLib/OpenBLAS>
- [6] AMD, "AOCL-BLAS (BLIS fork with AMD optimizations)," version 5.2.0. <https://github.com/amd/blis>
- [7] P. Gepner, "Using AVX2 instruction set to increase performance of high performance computing code," *Computing and Informatics*, vol. 36, no. 5, pp. 1001–1018, 2017. [https://doi.org/10.4149/cai\\_2017\\_5\\_1001](https://doi.org/10.4149/cai_2017_5_1001)
- [8] H. Zhang, D. Chen, and S. B. Ko, "Efficient multiple-precision floating-point fused multiply-add with mixed-precision support," *IEEE Trans. Comput.*, vol. 68, no. 7, pp. 1035–1048, 2019. <https://doi.org/10.1109/TC.2019.2895031>
- [9] M. Evers, L. Barnes, and M. Clark, "The AMD next-generation 'Zen 3' core," *IEEE Micro*, vol. 42, no. 3, pp. 7–12, 2022. <https://doi.org/10.1109/MM.2022.3152788>
- [10] S. Williams, A. Waterman, and D. A. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, 2009. <https://doi.org/10.1145/1498765.1498785>
- [11] Advanced Micro Devices, "Software optimization guide for AMD EPYC 7003 series processors (family 19h models 00h–0Fh)," rev. 3.00. [https://kib.kiev.ua/x86docs/AMD/Optimization/56665\\_3.00\\_Software%20Optimization%20Guide%20for%20AMD%20Family%2019h%20Processors%20%28PUB%29.pdf](https://kib.kiev.ua/x86docs/AMD/Optimization/56665_3.00_Software%20Optimization%20Guide%20for%20AMD%20Family%2019h%20Processors%20%28PUB%29.pdf)
- [12] A. Fog, "Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs," 2011, last updated Sep. 20, 2025. [https://www.agner.org/optimize/instruction\\_tables.pdf](https://www.agner.org/optimize/instruction_tables.pdf)
- [13] A. Fog, "The microarchitecture of Intel, AMD, and VIA CPUs: an optimization guide for assembly programmers and compiler makers," 2017, last updated Dec. 15, 2025. <https://www.agner.org/optimize/microarchitecture.pdf>
- [14] FinlayDaG33k, "RAM bandwidth calculator." <https://edu.finlaydag33k.nl/calculating%20ram%20bandwidth/>, <https://gitlab.com/FinlayDaG33k/ram-calculator>
- [15] Z. Smith, "bandwidth: a memory bandwidth benchmark." <https://zs3.me/bandwidth>
- [16] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, p. 12, 2008. <https://doi.org/10.1145/1356052.1356053>
- [17] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, p. 14, 2015. <https://doi.org/10.1145/2764454>
- [18] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proc. 1998 ACM/IEEE Conf. Supercomputing (SC'98)*. IEEE, 1998, p. 38. <https://doi.org/10.1109/SC.1998.10004>
- [19] S. A. Hassan, M. M. Mahmoud, A. M. Hemeida, and M. A. Saber, "Effective implementation of matrix–vector multiplication on Intel's AVX multicore processor," *Comput. Lang. Syst. Struct.*, vol. 51, pp. 158–175, 2017. <https://doi.org/10.1016/j.cl.2017.06.003>
- [20] A. Abdelfattah, D. Keyes, and H. Ltaief, "KBLAS: An optimized library for dense matrix-vector multiplication on GPU accelerators," *ACM Trans. Math. Softw.*, vol. 42, no. 3, p. 18, 2016. <https://doi.org/10.1145/2818311>
- [21] J. Liang and Y. Zhang, "Optimization of GEMV on Intel AVX processor," *Int. J. Database Theory Appl.*, vol. 9, no. 2, pp. 47–60, 2016. <https://doi.org/10.14257/ijda.2016.9.2.06>
- [22] A. Singh and C. Basso, "High-Performance Level-1 and Level-2 BLAS," *arXiv:2108.02025 [cs.MS]*, 2021. <https://doi.org/10.48550/arXiv.2108.02025>
- [23] B. Bylina, J. Bylina, and M. Piekarczyk, "Influence of loop transformations on performance and energy consumption of the multithreaded WZ factorization," in *Proc. 17th Conf. Computer Science and Intelligence Systems (FedCSIS), Ann. Comput. Sci. Inf. Syst.*, vol. 30, pp. 479–488, 2022. <https://doi.org/10.15439/2022F251>
- [24] T. Jakobs and G. Rünger, "On the energy consumption of Load/Store AVX instructions," in *Proc. 2018 Federated Conf. Computer Science and Information Systems (FedCSIS), Ann. Comput. Sci. Inf. Syst.*, vol. 15, pp. 319–327, 2018. <https://doi.org/10.15439/2018F28>
- [25] Free Software Foundation, "Using the GNU Compiler Collection (GCC)," version 15.2.0, 2025. <https://gcc.gnu.org/onlinedocs/gcc-15.2.0/gcc/>
- [26] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.