FASTGATR: AN OPTIMIZED EASY GEOMETRIC ALGEBRA TRANSFORMER

Phuc Vu*, **Hafsa Sheikh Mohamud Ahmed***, **Elitsa Popova***, **Omar Abdesslem***

Department of Computer Science
ETH Zürich, Switzerland

ABSTRACT

The Geometric Algebra Transformer (GATr) is a neural network architecture for 3D geometric data in which every feature is a 16-dimensional multivector of projective geometric algebra. Its layers are built from a small set of equivariant operations that the reference implementation evaluates as dense contractions against fixed structure tensors, even though these tensors are very sparse. We reimplement six core GATr functions in C++ and optimize them in three stages: exploiting the sparsity and constant structure of the tensors together with scalar optimizations such as independent accumulators and operator fusion, improving data layout and weight packing, and vectorizing with ARM NEON intrinsics. Together, these optimizations cut the operation count by up to two orders of magnitude and significantly speed up the runtime. On an Apple M1, the resulting vectorized code is up to $1500\times$ faster than a straightforward C++ baseline and 1.7–$4.5\times$ faster than the optimized PyTorch reference (EzGATr).

1. INTRODUCTION

Most neural networks treat geometric quantities like positions, orientations, and velocities as plain coordinate vectors, forcing the model to learn geometric symmetries from the data. The Geometric Algebra Transformer (GATr) [1] instead represents features as *multivectors* of the projective geometric algebra in 3D space, with layers that are equivariant to rigid motions by construction. This makes GATr a natural fit for physical simulation, robotics, molecular modeling, and computer vision, where geometric structure improves accuracy and data efficiency.

Rather than a single dense matrix product, a GATr forward pass is assembled from a small set of equivariant multivector operations (Figure 1), which are repeated throughout the network: the equivariant linear map (`EquiLinear`), equivariant RMS normalization (`EquiRMSNorm`), the geometric product, the equivariant join, geometric attention, and a scalar-gated GELU. What they share is that each is fixed by the algebraic structure tensors of the geometric algebra rather than by free parameters alone: the geometric product, the join, and the inner products inside attention are *bilinear* maps that contract two 16-component multivectors against a $16 \times 16 \times 16$ tensor, while `EquiLinear` is a linear map whose few learnable weights act through the same kind of fixed, sparse basis. Evaluated naively, as in the EzGATr PyTorch reference, each of these contractions is carried out densely against the full structure tensor and repeated for every multivector, channel, and token, so the forward pass quickly becomes slow as the number of tokens and channels increases.

Our starting point is that these structure tensors are extremely *sparse* and *constant*: only a small, fixed subset of the entries is non-zero, and the non-zero values are a handful of known constants. The dense reference therefore spends most of its work multiplying by zero. Exploiting this structure is what our optimization is built on.

Existing work focuses on the model rather than on its low-level performance. The original GATr paper [1] introduces the architecture and its operations, and EzGATr [2] provides a readable PyTorch reference that we use as our starting point and as a correctness oracle for validation. Neither targets the single-core performance of the underlying kernels, which is the gap we address.

Contribution. We reimplement six core GATr kernels in C++, turning the reference's dense structure-tensor contractions into sparse, specialized code. The key idea lies in exploiting the fixed, highly sparse structure of the operator tensors: replacing the dense contractions with their non-zero terms, folding `EquiLinear`'s constant basis into its weights, and reducing the attention scores to closed form together give an algorithmic $\approx 140\times$ reduction in operation count. We then layer standard single-core optimizations and ARM NEON vectorization on top, as detailed in Section 3.

*Equal contribution. This paper was supervised by Mohamed-Hicham Leghettas and Prof. Markus Püschel.

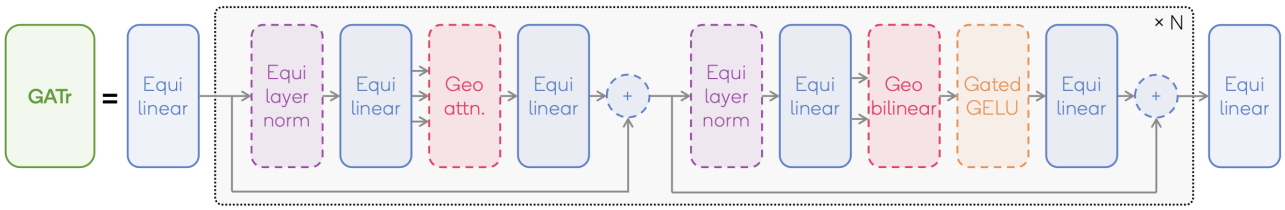

Fig. 1. Overview of the Geometric Algebra Transformer [1].

2. BACKGROUND ON THE ALGORITHM/APPLICATION

We briefly discuss GATr and the operations from which it is built, then we define the cost measure that we used throughout.

Geometric Algebra Transformer. GATr works in the projective geometric algebra of 3D space. A feature is a *multivector* $x \in \mathbb{R}^{16}$, which is a vector of coefficients with respect to the 16 basis blades. Such a multivector x encodes scalars, points, lines, planes, and the rigid transformations between them. In GATr, the geometric product, outer product, and inner products used in attention are *bilinear* maps that combine two multivectors through a fixed structure tensor $B \in \mathbb{R}^{16\times16\times16}$, where different operations correspond to different B. The equivariant join, for instance, is the outer product wrapped in the dual map, a fixed permutation-and-sign of the 16 coefficients. `EquiLinear` is instead a *linear* map but is also built from a fixed, sparse basis rather than a dense weight matrix. In every case the tensor is very sparse and constant, only a small subset of the $16^3 = 4096$ entries are non-zero.

The network. Figure 1 shows the full GATr forward pass: an `EquiLinear` layer embeds the input multivectors, then a stack of N blocks transforms them, and a final `EquiLinear` head reads them out. Each block consists of a geometric attention sub-layer and a geometric MLP (RMS norm, a bilinear sub-layer combining the geometric product and the join, a scalar-gated GELU, and further linear projections).

Cost measure. We count floating-point work as an operation count $W(n) = \sum_{\text{op}} c_{\text{op}}\, N_{\text{op}}(n)$, with weights that charge transcendental and divide-like operations more than additions and multiplications (add/mul 1, division 4, square root 8, exp and tanh 20). In practice these costly operations are under 0.04% of the total, so the weighting changes $W(n)$ by at most $\approx 0.5\%$ and it is effectively an unweighted flop count.

Cost analysis. The overall asymptotic cost is $O(LB(TC^2 + HT^2C))$, but the hidden constants matter just as much. For an input $x \in \mathbb{R}^{B\times T\times C_{\text{in}}\times 16}$, the equivariant linear map contracts each multivector against a fixed, sparse grade-projection basis $P \in \mathbb{R}^{9\times16\times16}$ and the learnable weights W.

Evaluated densely, the layer computes $16 \cdot 9 \cdot 16 = 2304$ terms for every batch element, token, input channel, and output channel. Each term costs two multiplies and an add, since the basis is not folded into W, giving

$$W^{\text{lin}}_{\text{dense}} \approx 3 \cdot 2304\; B\,T\,C_{\text{in}}\,C_{\text{out}} \tag{1}$$

flops for the linear layer. But the basis P is sparse: only about 24 of the 2304 interactions are non-zero, so the map needs only

$$W^{\text{lin}}_{\text{sparse}} \approx 2 \cdot 24\; B\,T\,C_{\text{in}}\,C_{\text{out}}, \tag{2}$$

a $\approx 140\times$ reduction in weighted operations ($96\times$ if counted as fused multiply-add terms, as in Table 1. The dense kernel does two multiplies per term, the optimized kernel one). Similarly for the bilinear operations, the dense form touches all 16^3 entries, but the sparse structure tensors have only 192 non-zero terms for the geometric product and 81 for the join. The remaining functions contribute few of these operations: RMS norm and the gated GELU are pointwise streaming passes with a handful of flops per element, while attention's cost lies in its $O(T^2)$ score and value-accumulation loops.

3. OPTIMIZATION PERFORMED

Overview. We first translated the EzGATr PyTorch reference [2] into C++. We then removed redundant arithmetic by exploiting tensor sparsity, exposed instruction-level parallelism, applied scalar replacement, and fused intermediate computations. Profiling the resulting scalar implementation identified `EquiLinear` as the remaining bottleneck at larger channel counts. We therefore improved its weight layout before vectorizing our kernels with ARM NEON.

Baseline implementation. The baseline evaluates each operator directly from its mathematical definition. The bilinear maps and `EquiLinear` load their coefficient tensors at runtime and evaluate them densely with one accumulator per output. Attention constructs intermediate multivector tensors, while the join reconstructs its coefficient tensor on every call. Although this implementation provides a simple

correctness reference, it evaluates many zero-valued terms and performs unnecessary memory accesses.

Exploiting tensor sparsity. The coefficient tensors are fixed and highly sparse. We therefore determine their nonzero entries in advance and generate only the required computations.

Kernel	Dense	Nonzero	Reduction
`EquiLinear`	2304	24	96×
`Geom. product`	4096	192	21×
`Equi. join`	4096	81	51×

Table 1. Number of tensor terms evaluated before and after exploiting sparsity. The reduction compares the dense and nonzero term counts.

For the geometric product and join, a Python code generator traverses the coefficient tensors and emits one operation for each nonzero term. Since the remaining coefficients are $+1$ or -1, their signs are implemented with addition or subtraction rather than a separate multiplication. As Table 1 shows, the geometric product retains 192 of 4096 terms, while the join retains 81.

The optimized `EquiLinear` kernel similarly keeps only its 24 nonzero $(w, s) \to d$ interactions. We also fold the fixed basis-normalization factors 1, $1/2$, $1/\sqrt{3}$, and $1/\sqrt{6}$ into the weights, leaving one multiply–add per interaction in the inner loop. `EquiRMSNorm` hardcodes the eight blades selected by its inner product, while the attention inner-product score retains only its seven contributing blades.

This specialization assumes the fixed 16-component projective geometric algebra used by GATr. Supporting another algebra or basis would require regenerating the kernels.

Algebraic simplification of attention. The directional (DAA) score in attention goes one step beyond dropping zeros: its bilinear map of two normalized trivectors is worked out in closed form, where the surviving terms cancel and collapse into a handful of reused subexpressions ($\|q\|^2$, $\|k\|^2$, and the 3D dot $\langle q, k\rangle$), leaving $\mathrm{daa} = 2\,q_3 k_3 \langle q, k\rangle - \|q\|^2 k_3^2 - q_3^2 \|k\|^2$. This reduces the operation count below the number of non-zero tensor terms because many terms cancel and several subexpressions can be reused. We also move trivector normalization outside the $O(T_q T_k)$score loop and normalize each query and key only once.

Instruction-level parallelism. The baseline equilinear, geometric-product, and join kernels use one accumulator for each serial reduction. As Figure 2 shows, each fused multiply–add depends on the result of the preceding instruction.

Fig. 2. A single accumulator creates a serial dependency chain in which every fused multiply–add depends on the previous result.

We replace the shared accumulator with independent per-blade accumulators and combine them only after the reduction. This allows the processor to issue operations from different dependency chains concurrently.

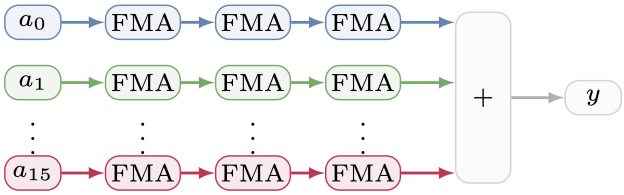

Fig. 3. Independent per-blade accumulators expose instruction-level parallelism by removing dependencies between different output blades.

Figure 3 illustrates the resulting independent accumulation chains. For the geometric product and join, the 192 and 81 nonzero terms are distributed across the corresponding output accumulators instead of being evaluated through one serial chain.

Scalar replacement. The generated geometric-product and join kernels reuse the same input blades across many operations. We therefore load each operand blade once into a scalar variable and reuse it for all dependent multiply–adds. In the geometric product, 16 input blades contribute to 192 terms, giving an average reuse factor of approximately 12×. Keeping these values in registers avoids repeated loads within the generated kernel.

Geometric fusion. The baseline geometric-attention implementation computes the IPA and DAA features, writes them to intermediate tensors, and reloads them in the pairwise scoring loop. We fuse feature generation with score computation so that each feature is produced and consumed in the same loop. This removes the intermediate feature tensors and avoids the associated stores and reloads.

Profiling. After applying the scalar optimizations, we profiled the C++ implementation on an Apple M1 using the CPU Counters instrument in Xcode Instruments [3]. Each configuration was executed 100 times, after which we exported the profiling log and corresponding `.trace` file. The measurements showed that `EquiLinear` remained the dominant kernel at larger channel counts and motivated the subsequent data-layout optimization.

Locality and data layout. The original `EquiLinear` weight layout did not match the order in which the optimized

inner loop consumed the coefficients. It therefore required repeated index calculations and accesses to separate regions of the weight tensor.

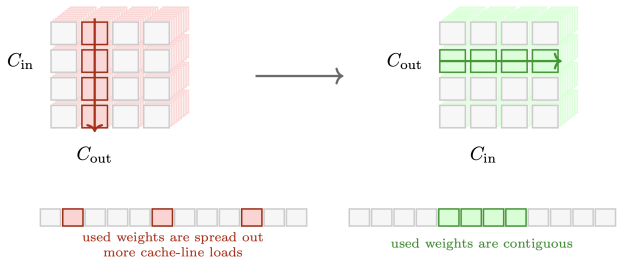

Fig. 4. Repacking of the `EquiLinear` weights into the order used by the optimized inner loop. The fixed basis-normalization factors are folded into the packed coefficients.

Figure 4 shows that the transformation changes. We repack the weights from $[C_{\text{out}}, C_{\text{in}}, 9]$ into $[C_{\text{in}}, C_{\text{out}}, 9]$. For each input-output channel pair, the packed layout stores the nine required coefficients contiguously and in their order of use. The packing step also folds the fixed basis-normalization factors into the weights.

Metric	Before	After
Instruction-processing bottleneck	12.29%	1.93%
Instruction-delivery bottleneck	41.54%	6.66%

Table 2. Xcode Instruments measurements before and after weight packing. Lower values indicate less time lost to the corresponding bottleneck.

Table 2 shows that both reported bottleneck metrics decrease significantly after packing. Since Instruments exposed only aggregate metrics on our setup, we do not attribute the improvement to a particular cache level. The measurements nevertheless indicate that matching the weight layout to the access order improves the efficiency of the inner loop.

SIMD vectorization. We hand-vectorize the hottest kernels using 128-bit ARM NEON intrinsics, which process four single-precision values per instruction. Because the interactions between the 16 multivector blades are irregular, we choose a different vectorization axis for each kernel.

For `EquiLinear` and `EquiRMSNorm`, we vectorize within one multivector. In `EquiLinear`, blades 1–4, 5–8, and 11–14 share the same packed coefficient within each group. Four scalar fused multiply–adds can therefore be replaced by one `vfmaq_n_f32` instruction. The remaining blades are processed with scalar instructions.

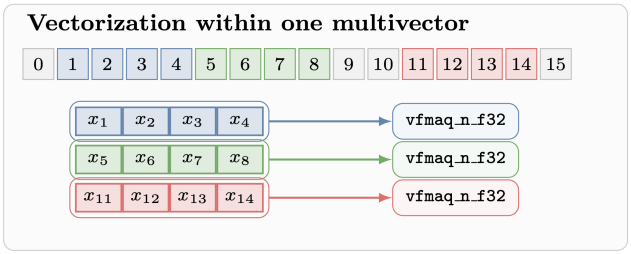

Fig. 5. Vectorization within a multivector for `EquiLinear` and `EquiRMSNorm`. Compatible groups of four blades are processed by one NEON instruction, while the remaining blades remain scalar.

Figure 5 illustrates this mixed scalar-vector mapping. The geometric product and join do not contain suitable groups of four blades within one multivector. We therefore vectorize them across the batch dimension, assigning one multivector to each SIMD lane. Every lane evaluates the same blade interaction for a different input, allowing the full 192-term kernel to execute without idle lanes.

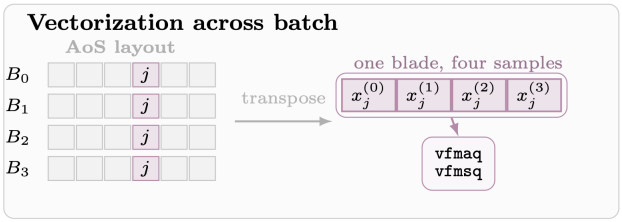

Fig. 6. Batch-wise vectorization of the geometric product. Each NEON lane evaluates the same blade interaction for a different multivector.

As Figure 6 shows, the fixed coefficients $+1$ and -1 map directly to NEON multiply–add and multiply–subtract instructions. A one-time `AoS→SoA` transpose prepares the inputs, and its cost is amortized over the 192 interactions. Our experiments use $B = 8$, so the batch dimension is divisible by the four NEON lanes.

Geometric attention is vectorized the same way along its channel dimension: the per-pair score is a 4-wide dot over the contributing blades, with channels padded to a multiple of four so that the spare lanes fall away.

4. EXPERIMENTAL RESULTS

4.1. Experimental setup

All experiments run on an Apple M1 (Firestorm performance core, ≈ 3.2 GHz; 128 KiB L1 data and 192 KiB L1 instruction cache per core, 12 MiB shared L2). Code is compiled with Apple Clang and different flag configurations. To isolate hand-written vectorization, the baseline and scalar builds

additionally pass -fno-tree-vectorize and -fno-slp-vectorize to disable compiler auto-vectorization, whereas the SIMD build uses NEON intrinsics with auto-vectorization left on. For testing, we scale batch, tokens, channels, and the number of layers together over seven sizes from $(B, T, C, L) = (8, 2, 1, 1)$ to $(8, 128, 64, 64)$. The internal width is fixed at 32 throughout, with 4 attention heads and $C_{qk} = C_v = 32$. The test cases run for several hours for the baseline c++ and much less time for the other implementations. Each configuration is timed after 10 warm-up runs and reported as the mean over 30 repeats. Every implementation is verified for correctness against the Python EzGATr reference, with end-to-end mean absolute percentage error below 0.1% across all sizes.

4.2. Results

We compare three optimized builds: *optimized C++* applies the sparsity exploitation together with the scalar optimizations (ILP, scalar replacement, and fusion), *optimized-for-cache* adds the `EquiLinear` weight-layout optimization on top, while *SIMD* further vectorizes the kernels with ARM NEON. We evaluate them along three axes: end-to-end runtime, sustained performance against the hardware peak, and a per-kernel breakdown. Then we finally end by placing the results on a cache-aware roofline.

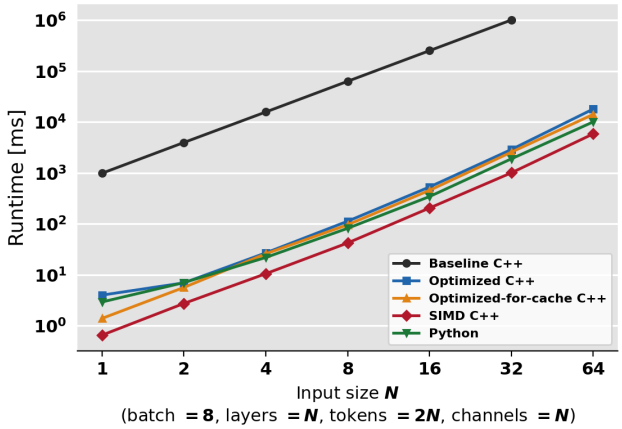

Fig. 7. End-to-end runtime comparison on a log scale.

End-to-end impact of the optimization stages. Figure 7 asks how much each optimization stage reduces the full model runtime compared to the original baseline. A log scale is needed because the baseline runs up to three orders of magnitude slower than the optimized versions. Exploiting the sparse algebraic structure is by far the largest single step: it removes a large amount of redundant arithmetic derived in Section 3. The first optimized versions, with and without cache-optimizations, are faster than the reference python code at very small sizes but slower at larger sizes. In contrast, the vectorized implementation outperforms the python code in terms of runtime across all sizes.

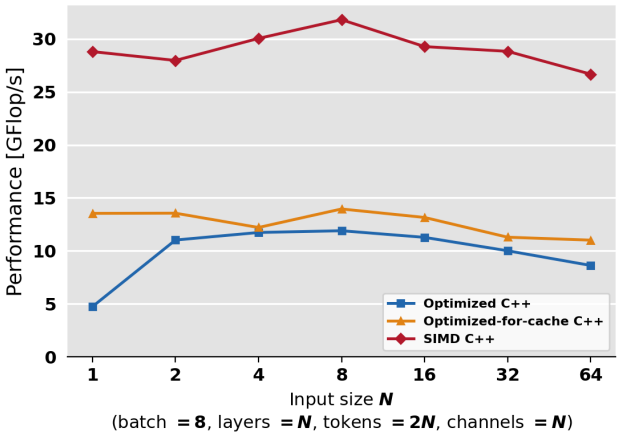

Fig. 8. End-to-end performance of the optimized, cache-optimized, and SIMD C++ builds, in GFlop/s.

Performance of the optimized implementations. Figure 8 compares the three builds with the same operation count. Python is excluded because it performs substantially more work. The scalar builds sustain $\approx$ 11–15 GFlop/s, while SIMD reaches $\approx$ 27–32 GFlop/s. Our fastest non-vectorized build achieves $\approx$ 14 GFlop/s with auto-vectorization disabled, whereas hand-written NEON reaches $\approx$ 32 GFlop/s, a $\approx 2.3\times$ gain and 36% of the measured 89.4 GFlop/s single-core peak. The remaining gap reflects memory-bound kernels, especially attention: the slight decline at the largest size is consistent with the memory-bound roofline regime (Fig. 13).

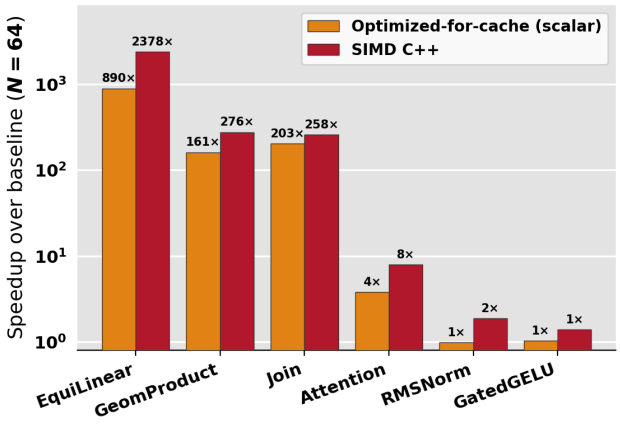

Fig. 9. Per-kernel speedup over the C++ baseline for the scalar (optimized-for-cache) and SIMD builds, at the largest size N=64.

Speedup per kernel Figure 9 breaks the gain down by ker-

nel, which the end-to-end numbers hide. At the largest size the dominant `EquiLinear` is about 890× faster scalar and 2378× faster vectorized than the baseline, and the two bilinear kernels (geometric product, join) gain 161 and 203× scalar and 276 and 258× with the vectorized version. These are the operations whose structure tensors are very sparse, so removing the zeros o removing the zeros is the largest part of the story. In contrast, geometric attention gains only ≈4× (scalar) / ≈8× (SIMD), because its runtime is set by the value accumulation and the $O(T^2)$ score and softmax passes, which are memory-bound streaming operations with low arithmetic intensity. Unlike the bilinear maps, removing the sparse-score zeros barely helps it. For `RMSNorm` and the gated GELU, the scalar build is no faster than the baseline, and NEON gives only ≈1-2× speedup, because these pointwise kernels make a single streaming pass with only a few flops per element, leaving little arithmetic to optimize. As they are a negligible share of total runtime (Fig. 10), we did not optimize them further. The end-to-end curve is therefore set by `EquiLinear` at scale and held back by attention and the pointwise kernels. The gap between the scalar and SIMD bars is the pure vectorization gain: while the working set stays cache-resident it reaches 4–5× (e.g. `geometric_product` 4.7×, `EquiLinear` 4.5× at T=16), and falls to 1.3–2.9× at the largest size (T=128) as the kernels become memory-bound and load bandwidth, which SIMD does not improve, starts to dominate.

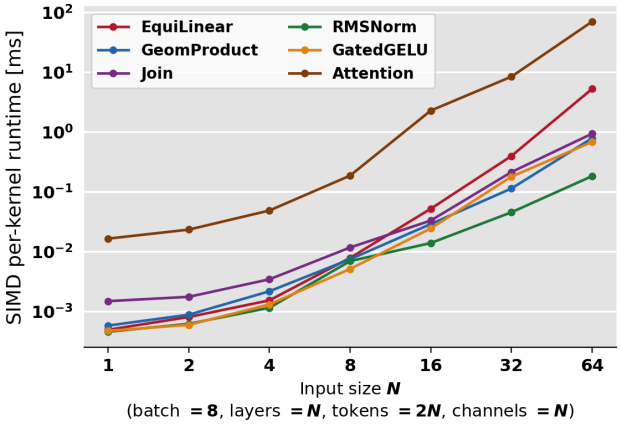

Fig. 10. Per-kernel runtime of the SIMD build vs. input size.

Where time is spent. Figure 10 shows the resulting per-kernel runtime of the SIMD build. `EquiLinear` and attention dominate at the larger sizes: because `EquiLinear` was sped up the most, the bottleneck shifts toward attention, whose softmax and value accumulation gain little from sparsity or vectorization, while the pointwise kernels (`RMSNorm`, gated GELU) remain negligible.

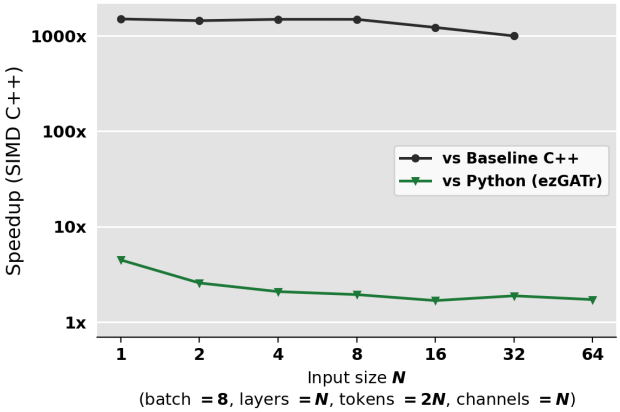

Fig. 11. End-to-end speedup of SIMD C++ over the scalar baseline and over the Python (EzGATr) reference. Baseline data stops at N=32.

Speedup over the baseline and the Python reference. Figure 11 reports the end-to-end speedup of the SIMD build over both references. Over the scalar baseline, it reaches ≈ 1500×. Most of this speedup is due to the operation-count reduction, with the layout, ILP, and NEON optimizations multiplying on top. The ratio flattens at the largest sizes because sparsification cuts the flop count but not the bytes moved (we still stream x, the weights, and the output), so the kernels turn memory-bound. Over the optimized PyTorch reference (EzGATr), the speedup is 1.7–4.5× and *shrinks* with size: EzGATr already runs its dense contractions through PyTorch's vectorized tensor kernels, so our advantage there comes from the lower operation count and reduced dispatch overhead.

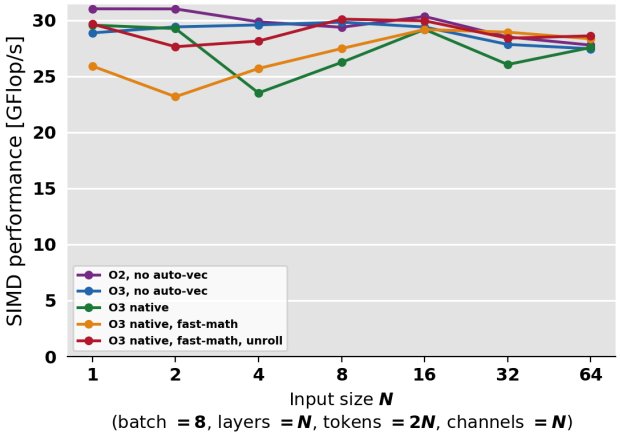

Fig. 12. Compiler-flag sweep for the SIMD C++ build.

Compiler-flag sensitivity. Figure 12 sweeps compiler flags for the SIMD build. The swept configurations are essentially indistinguishable: because the hot loops are

explicit NEON intrinsics, disabling auto-vectorization barely moves the curve, and `-ffast-math` does not help since the kernels are bound by FMA throughput, memory access, and register pressure rather than by expensive `libm` calls. The same holds for the scalar optimized build, where auto-vectorization can add register pressure without exposing useful parallelism. Overall, the gains come from algorithmic specialization and memory layout rather than from compiler flags.

Roofline analysis. We use the Cache-Aware Roofline Model (CARM) because most working sets fit in L2 cache, making a DRAM-only roofline misleading: the standard model classifies all implementations as compute-bound despite their different performance. CARM includes repeated loads from all levels of the memory hierarchy when computing data movement Q, which is important because cache re-reads dominate our memory traffic.

The x-axis reports operational intensity in FLOPs per loaded byte, and the y-axis reports analytical FLOPs per forward pass divided by measured runtime. We derive the work W and load count Q from the kernel source, as our Apple M1 setup does not expose the required cache-traffic counters.

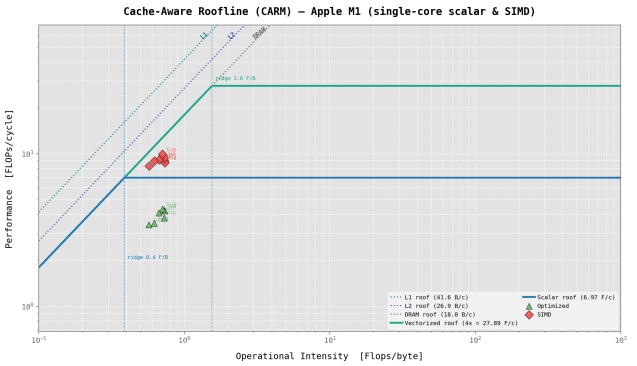

Fig. 13. Cache-Aware Roofline Model (CARM) on Apple M1.

Instead of relying on published specifications, we measured the roofline limits directly on the M1. DRAM bandwidth was measured using STREAM Triad and reached 57.6 GB/s (including writes). L1 and L2 read bandwidths were obtained with a simple NEON load microbenchmark and measured at 133 GB/s and 86 GB/s, respectively. Peak scalar and NEON FMA throughput were also measured, giving 22.3 GFLOP/s and 89.4 GFLOP/s.

Both the optimized and SIMD implementations are below the cache bandwidth roofs. The SIMD kernel is memory-bound, so further speed gains would mainly come from reducing memory traffic. The optimized scalar kernel has similar arithmetic intensity but remains below both the scalar compute roof and the bandwidth roofs, so it is not clearly limited by either.

Why further performance is limited. The speedups mainly came from removing work rather than approaching the arithmetic peak. Once the redundant computation is gone, the binding constraint is memory traffic: exploiting sparsity cut the operation count by a lot but left the data movement unchanged. We still stream the inputs, the packed weights, and the outputs, so the arithmetic intensity is low, and the cache-aware roofline places every implementation in the memory-bound region (Fig. 13). The kernels split into two groups (Fig. 9): `EquiLinear` and the bilinear maps are arithmetic-heavy and benefit most from vectorization while the working set is cache-resident, but drift toward memory-bound at the largest sizes, whereas attention is memory-bound throughout, dominated by streaming the $O(T^2)$ score and value-accumulation passes, so it gains little from SIMD. Because the forward pass is a chain of fixed-size (16-wide) multivector operations rather than one large dense matrix multiplication, data reuse is inherently low. Beyond the operator fusion we already apply, further gains would have to come from blocking the working set across operators to cut the streamed traffic, rather than from more vectorization.

5. CONCLUSIONS

We implemented a single-core optimized forward pass of the Geometric Algebra Transformer. The final implementation is up to $1500\times$ faster than a straightforward C++ baseline and 1.7–$4.5\times$ faster than the optimized PyTorch reference, EzGATr.

The largest gain comes from exploiting the fixed sparsity of the algebraic structure tensors. Evaluating only their nonzero terms removes up to two orders of magnitude of arithmetic. Weight packing, independent accumulators, operator fusion, and ARM NEON vectorization improve the specialized kernels further.

After these optimizations, memory traffic becomes the main limitation of the forward pass. Attention and the pointwise kernels benefit less because their softmax, value accumulation, and streaming accesses remain largely unchanged. Further improvements should therefore reduce data movement through additional fusion, blocking, and reuse across operators.

Overall, much of GATr's cost comes from applying fixed sparse operators through general dense computations; once this structure is made explicit, standard low-level optimizations become much more effective.

6. REFERENCES

[1] Johann Brehmer, Pim de Haan, Sönke Behrends, and Taco Cohen, "Geometric algebra transformer," 2023.

[2] Guest400123064, "EzGATr: Easy geometric algebra transformer," `https://github.com/Guest400123064/ezgatr`, 2024, GitHub repository, accessed June 2026.

[3] Apple Inc., "Addressing cpu bottlenecks," `https://developer.apple.com/documentation/xcode/addressing-cpu-bottlenecks`, Apple Developer Documentation, accessed June 17, 2026.

7. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

Omar. Implemented baseline C++ version of equi_geometric attention, equi_join, outer_product, and geometric_product together with Hafsa. Hardcoded non zeroes for equi_rms_norm function. Focused on cache optimization, in particular aligning weight tensors for contiguous memory access and SIMD Vectorization for the equilinear function. Experimented with different compiler flags. Used Instruments XCode profiling tool in collaboration with Hafsa to identify memory bottlenecks.

Hafsa. Implemented baseline C++ version of equi_geometric attention, equi_join, outer_product, and geometric_product together with Omar. Focused on scalar optimization, such as generated kernels, hardcoding zeroes, ILP, and scalar replacement for geometric and outer product. Did SIMD vectorization for geometric_product. Worked on the flop counters for the initial baseline. Used XCode to profile and identify bottlenecks in collaboration with Omar.

Elitsa. Implemented a baseline C++ version of rms_norm and gated_gelu. Applied scalar optimizations to equi_linear by hardcoding sparse basis computations, added accumulators, removed inner-loop loads (see code in section 9.1). Fused and optimized geometric_attention as described above, and vectorized it. Created roofline plots and did cache bandwidth benchmarking. Worked on the FLOP counter, data movement counter.

Phuc. Implemented the baseline C++ version of the EquiLinear module and assembled the full GATr forward pass (`asl_mv_only_gatr`) from the individual kernels, debugging the forward pass so the network produced valid outputs. Analyzed the coefficient tensors of the bilinear maps to identify and exploit their sparsity. Wrote the Python code generator that produces the hardcoded, sparsity-exploiting scalar kernels for the equivariant join and the geometric product. Implemented the SIMD vectorization of the equivariant join.

8. CODE

8.1. most performance-critical part of fastest scalar code

```
void equi_linear_packed_kernel(float* out, const float* x, const float* packed_weight, const float* bias,
    int B, int T, int channels_in, int channels_out) {
    for (int b = 0; b < B; b++) {
        for (int t = 0; t < T; t++) {
            const int batch_token = b * T + t;
            const float* x_row = x + batch_token * channels_in * 16;
            float* out_row = out + batch_token * channels_out * 16;
            for (int o = 0; o < channels_out; o++) {
                float scalar_blade = 0.f;
                float blade_1_diag = 0.f, blade_1_off = 0.f;
                float blade_2 = 0.f, blade_3 = 0.f, blade_4 = 0.f;
                float blade_5_diag = 0.f, blade_5_off = 0.f;
                float blade_6_diag = 0.f, blade_6_off = 0.f;
                float blade_7_diag = 0.f, blade_7_off = 0.f;
                float blade_8 = 0.f, blade_9 = 0.f, blade_10 = 0.f;
                float blade_11_diag = 0.f, blade_11_off = 0.f;
                float blade_12_diag = 0.f, blade_12_off = 0.f;
                float blade_13_diag = 0.f, blade_13_off = 0.f;
                float blade_14 = 0.f;
                float blade_15_diag = 0.f, blade_15_off = 0.f;
                for (int i = 0; i < channels_in; i++) {
                    const float* x_channel = x_row + i * 16;
                    const float* packed_io = packed_weight + (i * channels_out + o) * 9;
                    scalar_blade += packed_io[0] * x_channel[0];
                    blade_1_diag += packed_io[1] * x_channel[1];
                    blade_2      += packed_io[1] * x_channel[2];
                    blade_3      += packed_io[1] * x_channel[3];
                    blade_4      += packed_io[1] * x_channel[4];
                    blade_5_diag += packed_io[2] * x_channel[5];
                    blade_6_diag += packed_io[2] * x_channel[6];
                    blade_7_diag += packed_io[2] * x_channel[7];
                    blade_8      += packed_io[2] * x_channel[8];
                    blade_9      += packed_io[2] * x_channel[9];
                    blade_10     += packed_io[2] * x_channel[10];
                    blade_11_diag += packed_io[3] * x_channel[11];
                    blade_12_diag += packed_io[3] * x_channel[12];
                    blade_13_diag += packed_io[3] * x_channel[13];
                    blade_14      += packed_io[3] * x_channel[14];
                    blade_15_diag += packed_io[4] * x_channel[15];
                    blade_1_off  += packed_io[5] * x_channel[0];
                    blade_5_off  += packed_io[6] * x_channel[2];
                    blade_6_off  += packed_io[6] * x_channel[3];
                    blade_7_off  += packed_io[6] * x_channel[4];
                    blade_11_off += packed_io[7] * x_channel[8];
                    blade_12_off += packed_io[7] * x_channel[9];
                    blade_13_off += packed_io[7] * x_channel[10];
                    blade_15_off += packed_io[8] * x_channel[14];
                }
                write_equi_linear_row(
                    out_row + o * 16,
                    scalar_blade,
                    blade_1_diag, blade_2, blade_3, blade_4,
                    blade_5_diag, blade_6_diag, blade_7_diag,
                    blade_8, blade_9, blade_10,
                    blade_11_diag, blade_12_diag, blade_13_diag,
                    blade_14, blade_15_diag,
                    blade_1_off, blade_5_off, blade_6_off, blade_7_off,
                    blade_11_off, blade_12_off, blade_13_off, blade_15_off,
                    bias ? bias[o] : 0.0f);
                    }}}}
```

8.2. most performance-critical part of fastest SIMD code

```
void equi_linear_packed_kernel(float* __restrict out, const float* __restrict x,
    const float* __restrict packed_weight, const float* __restrict bias,
    int B, int T, int channels_in, int channels_out) {

    for (int b = 0; b < B; b++) {
        for (int t = 0; t < T; t++) {
            const int batch_token = b * T + t;
            const float* __restrict x_row = x + batch_token * channels_in * 16;
            float* __restrict out_row = out + batch_token * channels_out * 16;

            for (int o = 0; o < channels_out; ++o) {
                float scalar_blade = 0.f;
                float blade_1_off = 0.f;
                float blade_5_off = 0.f, blade_6_off = 0.f, blade_7_off = 0.f;
                float blade_9 = 0.f, blade_10 = 0.f;
                float blade_11_off = 0.f, blade_12_off = 0.f, blade_13_off = 0.f;
                float blade_15_diag = 0.f, blade_15_off = 0.f;
                float32x4_t blades_1_4 = vdupq_n_f32(0.0f);
                float32x4_t blades_5_8 = vdupq_n_f32(0.0f);
                float32x4_t blades_11_14 = vdupq_n_f32(0.0f);

                for (int i = 0; i < channels_in; i++) {
                    const float* __restrict x_channel = x_row + i * 16;
                    const float* __restrict packed_io = packed_weight + (i * channels_out + o) * 9;
                    scalar_blade += packed_io[0] * x_channel[0];
                    blades_1_4 = vfmaq_n_f32(blades_1_4, vld1q_f32(x_channel + 1), packed_io[1]);
                    blades_5_8 = vfmaq_n_f32(blades_5_8, vld1q_f32(x_channel + 5), packed_io[2]);
                    blade_9 += packed_io[2] * x_channel[9];
                    blade_10 += packed_io[2] * x_channel[10];
                    blades_11_14 = vfmaq_n_f32(blades_11_14, vld1q_f32(x_channel + 11), packed_io[3]);
                    blade_15_diag += packed_io[4] * x_channel[15];
                    blade_1_off += packed_io[5] * x_channel[0];
                    blade_5_off += packed_io[6] * x_channel[2];
                    blade_6_off += packed_io[6] * x_channel[3];
                    blade_7_off += packed_io[6] * x_channel[4];
                    blade_11_off += packed_io[7] * x_channel[8];
                    blade_12_off += packed_io[7] * x_channel[9];
                    blade_13_off += packed_io[7] * x_channel[10];
                    blade_15_off += packed_io[8] * x_channel[14];
                }
                write_equi_linear_row(out_row + o * 16, scalar_blade, blades_1_4, blades_5_8,
                    blade_9, blade_10, blades_11_14, blade_15_diag, blade_1_off,
                    blade_5_off, blade_6_off, blade_7_off, blade_11_off, blade_12_off,
                    blade_13_off, blade_15_off, bias ? bias[o] : 0.0f);
            }
        }
    }
}
```