LAYER-WISE ACCELERATION OF CLIFFORD NEURAL NETWORKS

Anirudhh Ramesh, Konstantinos Chasiotis, Mikkeline Elleby, Simon Huber

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Clifford Neural Networks have been shown to be useful, but because of their structure, performance suffers. To enable optimized forward passes on three layers, we first translate them into C. In a second step, we optimize each layer for x86_64 CPUs. For all three layers, significant speedups were achieved.

1. INTRODUCTION

As data grew larger and chips grew faster, deep learning has emerged as one of the dominant machine learning practices for image classification, control, simulation, and scientific modeling tasks. Neural networks are built on conventional linear algebra, operating on data as vectors or matrices of numbers. Such models, use operations like matrix multiplication and dot products to carry out their tasks. However, these models do not encode geometric meaning like orientation, distance, or shape into how the data is represented or transformed. For instance, a 3D point (x, y, z) should behave differently when rotated versus translated. So, they learn these geometric relationships, which effectively requires a lot of data and training. This shortcoming becomes pronounced in domains where geometry is not a side effect, but a central feature. To address this, Geometric Deep Learning has emerged which makes use of symmetry groups and geometric priors [1].

Motivation. Although geometric deep learning models are designed for specific domains or symmetry groups, they often operate on limited geometric primitives. To address these limitations, Clifford Algebra has been used to characterize relationships in high dimensions of geometry. Clifford Algebra is a mathematical framework that generalizes scalars, vectors, and higher-grade geometric objects into a unified system known as multivectors.

Built from this framework, Clifford Neural Networks (Clifford NNs) are a class of models that use this algebra to represent data, weights, and transformations. By doing so, they incorporate geometric structure and model spatial transformations, physical interactions, and structural invariants within the data [2, 3].

Despite their expressiveness, Clifford NNs are computationally expensive. Each multivector consists of 2^n components (where n is the dimension of the vector space), and the geometric product introduces intricate dependencies across components. As a result, most existing implementations are based on high-level frameworks (e.g., PyTorch) and are not optimized for low-level performance—making inference slow for real-time or large-scale applications.

Contribution. To address the performance limitations of existing Clifford NN implementations, we present a high-performance inference framework. Building on the open-source *CliffordLayers* reference implementation[1], which realizes every layer in Python, we develop a set of optimized `C99` layers targeting inference. We focus on accelerating the three layers in Clifford NNs: (i) Clifford linear projection, (ii) multivector sigmoid-linear unit (SLU) and (iii) Clifford Group Normalisation [3].

Each layer is written from scratch and combines cache blocking, AVX2 SIMD vectorization, fast and sparse MVM to maximize performance on modern x86_64 CPUs. To ensure compatibility with existing model code, the optimized functions are exposed through a minimal `ctypes` wrapper that leaves the high-level Python API unchanged.

2. MATHEMATICAL FOUNDATIONS OF CLIFFORD LAYERS

Clifford NNs use multivector-valued features using Clifford algebra. The core operation of Clifford algebra is the geometric product, which combines the dot and wedge products into a single associative operation:

$$uv = u \cdot v + u \wedge v, \text{ where } u, v \in \mathbb{R}^n$$

The geometric product encodes parallel and orthogonal interactions between vectors, enabling the representation of complex geometric objects within a unified framework. To make use of such an operation, our data and weights are not scalars or vectors, but mutlivectors.

A multivector is a linear combination of basis elements called blades. These blades span different grades, where each grade corresponds to a different geometric primitive such as a scalar (grade-0), vector (grade-1), bivector (grade-2), all the way to a pseudoscalar (grade-n). Multivectors can

[1] `https://microsoft.github.io/cliffordlayers/`

be expressed as a linear combination of basis blades:

$$x = \sum_{I \subseteq \{1,\ldots,n\}} x_I e_I, \text{ where } x_I \in \mathbb{R},$$

and each basis blade e_I is defined as the geometric product of an ordered subset of orthonormal basis vectors $\{e_1, \ldots, e_n\}$. A result of this definition is that the total number of such blades is 2^n; all the possible subsets of the n basis vectors. This exponential growth means that the dimension of the Clifford algebra grows rapidly with n.

To perform the multiplication between these basis blades, the algebra makes use of a set of antisymmetric rules that define the algebra's structure. These are:

$$e_i e_j = \begin{cases} +1 & \text{if } i = j \text{ and } e_i^2 = +1, \\ -1 & \text{if } i = j \text{ and } e_i^2 = -1, \\ -e_j e_i & \text{if } i \neq j. \end{cases}$$

These rules determine how products of basis vectors combine with each other into higher-grade blades and whether signs (+/-) need to be introduced when reordering the basis. Because of these, we can produce sums of blades of different grades in a non-commutative fashion (e.g., scalars + bivectors), known as multigrade outputs.

Clifford NNs use the above formulations to construct layers that operate directly on multivector-valued feature channels. In the reference architecture proposed by Brandstetter et al. [4], three layers are discussed: **(i)** Clifford Linear Projection, **(ii)** Multivector Sigmoid–Linear Unit (SLU), **(iii)** Clifford Group Normalization.

Due to the dense coupling in the geometric product and the exponential number of basis elements, the computational cost of Clifford layers is significant. Taking that in consideration, we proceeded with optimizing the given code.

3. IMPLEMENTATION FRAMEWORK

Starting off, it was necessary to lay out the infrastructure that we would be building our work upon. This involved choosing the appropriate architecture and michroarchitecure, converting the PyTorch code to `C99`, and developing a clear execution pipeline that would allow us to benchmark and validate our results.

Architecture & Microarchitecture. For the architecture and microarchitecture, we used a system equipped with an Intel Core i5-1035G4 (Ice Lake) processor, featuring a base frequency of 1.10 GHz and a maximum turbo frequency of 3.70 GHz.

We choose a device with such specifications due to the amount of resources available from the course on the Ice Lake microarchitecture. This assisted us verifying in a theoretical manner the results we recorded.

C integration and Build. The next steps involved translating the reference PyTorch code into `C99`. We transalted the three performance-critical layers, and exposed them through a minimal `ctypes` wrapper. This allowed the original training and inference scripts to remain unchanged, effectively providing a compatible interface which is integrated with the existing code base.

Functional Verification. To validate our C implementations, we employ a `pytest` based test harness that generates 10^3 random input tensors per layer. These cover grades $d \in \{2, 3\}$, channel counts $4 \leq C \leq 64$, and batch sizes $1 \leq B \leq 32$. For each case, we compare the output of the C and Python implementations using the infinity norm: $\|y_{\mathrm{C}} - y_{\mathrm{Py}}\|_\infty < 10^{-6}$. All tests run automatically in the continuous integration workflow.

Benchmarking Pipeline. To evaluate the performance of our optimizations, we developed a benchmarking pipeline for each of layer. This pipeline mimics the functionality of *CodeExpert*, which we used during the course. Each optimization we develop is wrapped in an interface and registered dynamically through a central registry. The pipeline counts how many cycles it takes for each optimization we register to finish, and measures the relative speedup compared to the baseline. In addition, it tests whether the outputs of each optimization are the same as those of the baseline.

The pipeline supports two modes: a fixed input mode for cycle-level performance analysis. Here, we keep the batch size fixed ($B = 2^4$) allowing us to measure how an optimization compares to others. The other mode is the scaling mode. Here, we sweep over different input batch sizes ($B = 2^4 ... 2^{19}$), resulting in the plotting of our performance plots.

For more details on how the pipeline works and is executed, we refer to the `README.md` of our repository

4. LAYER 1: CLIFFORD LINEAR PROJECTION

4.1. Background and Notation

The Clifford Linear Layer is the affine transformation component in Clifford NN. It extends a fully connected layer by making use of multivector-valued inputs and the geometric product. Each input and output feature channel encodes a multivector with 2^d components (blades), where d is the geometric dimension of the Clifford algebra $\mathrm{Cl}(p, q)$.

We denote the key dimensions as follows: B is the batch size, C_{in} and C_{out} are the number of input and output channels, respectively, and $I = 2^d$ is the number of blades in the algebra. Thus, the input tensor is $x \in \mathbb{R}^{B \times C_{\text{in}} \times I}$ and the output is $y \in \mathbb{R}^{B \times C_{\text{out}} \times I}$.

The layer computes output multivectors as a sum of geometric products between learned multivector weights and input multivectors. For input multivectors x_i and learned

weight multivectors W_{ji}, the output is: $y_j = \sum_i W_{ji} x_i$, where both W_{ji} and x_i are multivectors, and the product is the geometric product.

In practice, this is implemented as a matrix–vector multiplication over flattened blade coefficients. The full projection matrix $K \in \mathbb{R}^{(C_{\text{out}} \cdot I) \times (C_{\text{in}} \cdot I)}$ is constructed by combining learnable weights w, metric coefficients g, and the Clifford multiplication table. This effectively applies a batched affine map to multivector inputs.

At runtime, the input is flattened to $x_{\text{flat}} \in \mathbb{R}^{B \times (C_{\text{in}} \cdot I)}$, and the output is computed via:$y_{\text{flat}} = x_{\text{flat}} \cdot K^\top$, then reshaped back to $\mathbb{R}^{B \times C_{\text{out}} \times I}$. This allows the layer to leverage efficient matrix-multiplication routines while preserving the geometric structure.

Cost Model. The core operation is a dense fp32 matrix multiplication between the flattened input tensor and a sparse weight matrix shaped $(C_{\text{out}} I) \times (C_{\text{in}} I)$, producing $C_{\text{out}} \cdot I$ output features per sample.

Each output feature is computed via a dot product of length $C_{\text{in}} \cdot I$, with each multiply–add contributing two FLOPs. An additional output feature incurs an extra addition. Thus, the total number of FLOPs for the matrix multiplication across the batch including the bias, is:

$$F_{\text{gemm}} = 2\, B\, C_{\text{in}}\, C_{\text{out}}\, I^2 = 2\, B\, C_{\text{in}}\, C_{\text{out}}\, 4^d + B\, C_{\text{out}}\, 2^d$$

This exponential growth in d highlights the necessity of optimized implementations for tractable training and inference.

Memory Footprint. Considering we are using 32-bit floats (4 bytes per value), the total memory used at inference time is: $M_{\text{total}} = 4 \cdot \left[2BC_{\text{in}}I + 2BC_{\text{out}}I + C_{\text{in}}C_{\text{out}}I^2 + C_{\text{out}}I\right]$ where the first term within the bracket comes from the input and its flattened copy, the second term from the output and its flattened form, the third from the kernel and the forth from the bias vector.

4.2. Optimizations performed

We split our optimizations into a 'kernel-based' approach (following the baseline construction) and a 'no-kernel' approach. We do not count the cycle times of the kernel construction, given it is dependent on the weights, which is given beforehand.

Kernel-based approach. The baseline is directly adapted from the Python version.

Optimization 1: Kernel Caching and Memory Alignment. The baseline recomputes the kernel matrix K on each forward pass, which is wasteful. Thus, we cache K and only recompute it when the required size changes. Furthermore, we use `__builtin_assume_aligned` to ensure K is aligned to 64 bytes, which allows more efficient SIMD instructions.

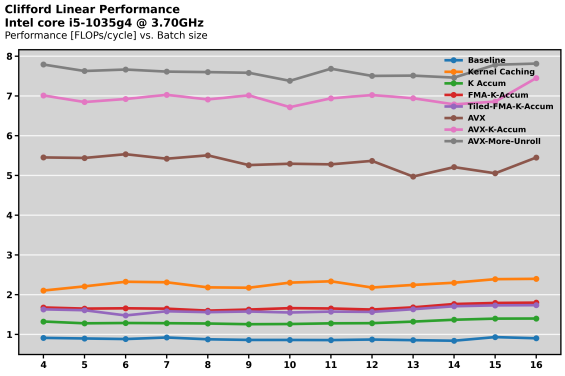

Fig. 1. Kernel-based optimizations performance plot

Optimization 2: Accumulators and Fused Multiply-Add (FMA). We break dependency stalls by using k- accumulators, thereby increasing instruction-level parallelism (ILP). We also use FMA instructions to the dot-products, increasing performance.

Optimization 3: AVX Vectorization with Multiple Accumulators. We process 8 floats per cycle by applying AVX2 FMA ops directly, since the scalar memory accesses are already contiguous. We then use 2 vector accumulators, which leads to significant speed-up by breaking data dependencies. We extend this to 4 accumulators which adds a smaller further speed-up.

No-kernel approach. In this approach, we compute the output without a kernel, thereby removing the kernel overhead if construction time should be considered.

To do this, each computation directly indexes on the weights and multiplies against the input blades. We then accumulate this to get the output blade value.

Furthermore, we are able to factor out the g-coefficients from the multiplication (over the input blades), which saves us cycles.

Optimization 1: Flattened weights, factored g coefficients, FMA. We flatten the weights to efficiently index in a cache-optimal way. Furthermore, we accumulate the outputs of the input channels into a per-blade temp vector. This allows us to factor out the g coefficients. Additionally, we use FMA operations which further speeds up the optimization.

Optimization 2: Inverse computation. As Clifford kernel is symmetric, we compute both the top row and the bottom row in the same iteration of the input channel. This allows better re-use of input, leading to a significant speed-up.

4.3. Experimental results

The Performance Plot in 1 shows the most significant kernel-based optimizations. 2 shows the no-kernel optimizations

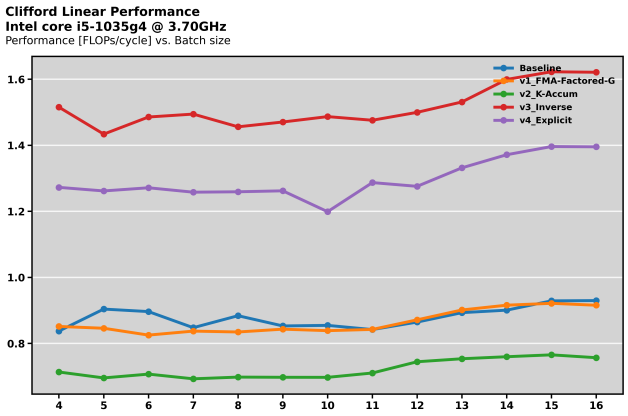

Fig. 2. No-kernel optimizations performance plot

Kernel-based Analysis. Kernel Caching By caching the kernel, we reduce redundant computation, especially for models run on repeated input shapes. This gains a 2x improvement. **FMA-K-Accum** Using fma and k-accumulators performs poorly on the chosen architecture, possibly due to hardware-related limitations. **AVX** Vectorization allows us to perform significantly more flops per cycle using the 256-bit registers, leading to 5.5x speed-up over baseline. **AVX-More-Unroll** Our best optimization uses 4-vector accum, which further breaks the dot-product accumulation.

No-Kernel Analysis. v1 The no-kernel approach leads to small speed-up over baseline, due to no overhead from kernel computation or input flattening. **v3** Using the inverse computation leads to a significant speed-up since we re-use the input channels. **v4** Explicitly specifying the computation using switch-case performs worse, likely bad for compiler

Roofline Analysis and Bottlenecks. 3 shows close to peak scalar, with vector much lower (due to algorithmic constraints) & overall heavily compute-bound.

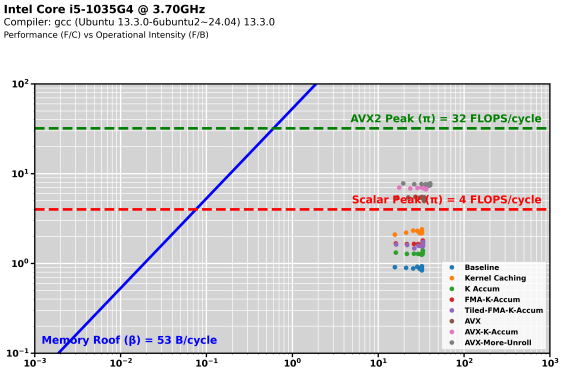

Fig. 3. Roofline plot for the Linear Layer

5. LAYER 2: MULTIVECTOR SIGMOID LINEAR UNIT

5.1. Background on the Algorithm

The sigmoid linear unit (SLU) introduces nonlinearity into Clifford NNs by using a learned sigmoid gating mechanism. Multivector features are stored in a tensor $X \in \mathbb{R}^{B\times C\times L\times I}$ [2]. For every $(b, c, p) \in [0, B) \times [0, C) \times [0, L)$:

1. Embedding. Each input blade coefficient $x_{b,c,i}$ is mapped into a full-algebra embedding buffer $v \in \mathbb{R}^{B\times C\times I_{\text{full}}}$ at position β_i: $v_{b,c,\beta_i} = x_{b,c,i}$ [3]

2. Depth-wise Convolution and Gating. Next, we compute a channel-specific gating factor by applying a depth-wise convolution. For each pair (b, c), a pre-activation scalar $a_{b,c}$ is computed as: $a_{b,c} = b_c + \sum_{k=0}^{K-1} w_{c,k}\, v_{b,c,\kappa_k}$ [4]

A sigmoid activation transforms this scalar into a gating factor: $g_{b,c} = 1/1 + e^{-a_{b,c}}$

3. Gathering and Blade-wise Scaling. Finally, each original input blade coefficient is scaled uniformly by the gating factor and gathered back to form the output multivector: $y_{b,c,i} = g_{b,c}\, v_{b,c,\beta_i} = g_{b,c}\, x_{b,c,i}$

Cost model. Counting one flop per multiply, add, or reciprocal, 12 flops per exponential, plus one multiply and one write per blade in embed/get, it gives us:

$$\underbrace{2K}_{\text{conv}} + \underbrace{15}_{\text{exp (12) + recip + 2 adds (bias accum.)}} + \underbrace{2I}_{\text{embed + gather}} + \underbrace{I}_{\text{final scale}}$$
$$= 2K + 3I + 15 \quad \text{flops per } (b, c, p)\,.$$

Memory Footprint. The algorithm requires the following buffers during runtime: the temporary buffer $v \in \mathbb{R}^{B\times C\times I_{\text{full}}}$ for full blade embedding, the gating array $\text{gate} \in \mathbb{R}^{B\times C}$ storing sigmoid values, and the input/output arrays of shape $B \times C \times L \times I$ that hold the multivector feature maps.

5.2. Optimization performed

We applied a variety of performance improvements to this layer; below are the most impactful ones.

Optimization 1: Compact Index Inversion to Leverage Blade Sparsity and Pre-filtering Active Blades. In our baseline activation, each invocation allocated and zeroed a large $B \times C \times I_{\text{full}}$ buffer even though only $I \ll I_{\text{full}}$ blades held data, writing and reading every index and generating $\mathcal{O}(BCI_{\text{full}})$ traffic that overwhelmed caches and dominated runtime. We eliminate this overhead by replacing the full buffer with a compact inverse map, a one-time $\mathcal{O}(I_{\text{full}})$ array `inv` where `inv[b]` gives the slot for blade b or -1 if unused. The activation kernel's inner loop now consults

[2] batch B, channel C, postions L, blades I, dim of full algebra I_{full}

[3] Indices $i = 0, \ldots, I-1$ represent blade positions within the multivector and $\beta_i = \texttt{input\_blades[i]}, \quad \beta_i \in [0, I_{\text{full}})$

[4] where b_c is a learnable bias, $w_{c,k}$ are learnable weights, K is the kernel size (number of blades in conv) , $\kappa_k = \texttt{kernel\_blades}[k]$

`inv` for the K non-zero weights, cutting per-(batch,channel) work from $\mathcal{O}(I_{\text{full}})$ to $\mathcal{O}(K)$. As a result, memory traffic collapses to $\mathcal{O}(I_{\text{full}})+\mathcal{O}(BCK)$, exposing true sparsity and boosting throughput. Because active blades and weights never change, we scan each channel once, drop zero entries, and avoid needless multiplies. Each batch's dot product therefore runs in $\mathcal{O}(K_n)$ time[5].

Optimization 2: Sigmoid Acceleration via Lookup. In earlier kernels the sigmoid, `gate = 1.0f / (1.0f + expf(-acc));` dominated runtime resulting in millions of `expf()` calls per batch stall the pipeline for hundreds of cycles each.

We replace it with a 512-entry lookup table spanning $[-6, 6]$ and tested/tuned to still get the required 1×10^{-5} accuracy. At initialization, we sample the sigmoid function at evenly spaced intervals and store these samples in an array. During activation, each accumulated value a is clamped into $[-6, 6]$ and mapped into a floating-point index: $f = (a + 6)\frac{511}{12}$

This index is then split into integer and fractional parts, used to fetch two adjacent table entries, and combined through linear interpolation to approximate the sigmoid result. The transcendental and divide collapse to two array loads and three ALU ops, removing the hotspot.

Optimization 3: Tiling for Sparse Locality. As batch size grew, performance plunged; profiling showed cache thrashing as larger working sets evicted L1/L2 lines. We tiled the loop space (batch = 64, channel = 4) so the inner-loop footprint—inputs, outputs, weights, lookup tables—fits in the i5's 64 KB L1 cache (32 KB data + 32 KB instructions), restoring locality and throughput. Tuning confirmed this choice od parameters gives the best balance of locality and setup overhead.

Optimization 4: Branchless Mask-Based Scaling. Profiling revealed that the final gating step (deciding for each element whether to zero, copy, or multiply by the sigmoid gate) introduced costly conditional branches into our inner loop. To eliminate these, we compute a single floating-point mask s that encodes all three behaviors in one arithmetic expression[6].

We introduce a small tolerance $\varepsilon = 1 \times 10^{-6}$ so that any gate value $\leq \varepsilon$ yields zero and any value $\geq 1 - \varepsilon$ yields one and then every output is then computed as output $=$ input $\times\, s$.. Through testing, we confirmed that this threshold both avoids numerical edge cases and allows us to remove the if statements without misclassifying values.

5.3. Experimental results

Figure 4 summarizes every step: Across all batch sizes, the decisive gains stem from eliminating redundant memory traffic, removing the `expf`, and confining each tile to L1 and L2.

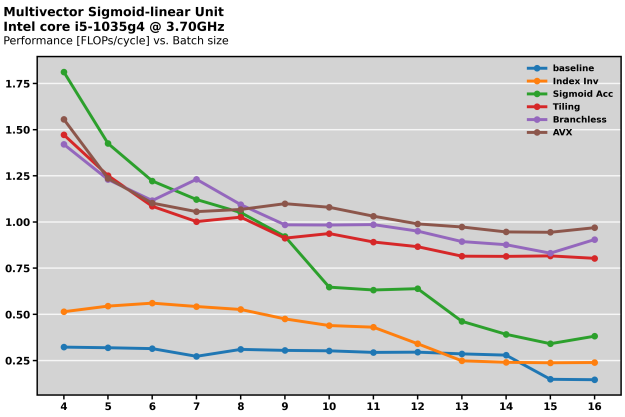

Fig. 4. Performance plot for the SLU

Inverse Map & Filtering. Replacing the baseline's scatter–gather buffer with a blade $\rightarrow$ slot inverse map and pruning zero weights cuts memory traffic in half and lifts throughput from ≈ 0.30 to ≈ 0.55 F/C for small batches.

LUT Sigmoid. Replacing `expf` with a 512-entry lookup table makes the sigmoid essentially free and pushes the peak to ≈ 1.8 F/C for the tiniest batch. However, scattered inputs and the LUT now contend for the caches so the performance decreases rapidly.

Tiling. Processing data in 64×4 (batch $\times$ channel) tiles keeps each slice's inputs, weights and LUT resident in L1. The tiling overhead lowers the tiny-batch peak because of the overhead for computing the tiling, yet the curve stays flat around ≈ 0.9 F/C for all larger batches instead of collapsing.

Branchless Mask. Turning three conditionals into one arithmetic scale factor removes the last unpredictable branches, letting the tiled kernel reach ≈ 0.9 F/C even for small batches.

AVX and Bottleneck. The plot barely rises: As our dot-product is extremely sparse, every weight points to a blade value sitting at an unpredictable offset in memory. Thus, the CPU executes the inner loop as a stream of gather instructions, each chasing eight independent pointers. Those random gathers flood the (only) two load ports and stall for data fetching. Having more accumulators just multiplies the number of outstanding gathers, so the kernel stays waits due to memory latency. So here the true bottleneck is the scatter-style, sparse dot-product itself.

Roofline analysis. See figure 5, all dots share one vertical line (batch size changes intensity only a bit) so none cross the red scalar-peak bar. Until we replace costly random gathers (maybe by pre-packing inputs), further speed-ups are capped.

[5] K_n is the number of non-zero weights

[6] one or two comparisons followed by a few multiplies and adds

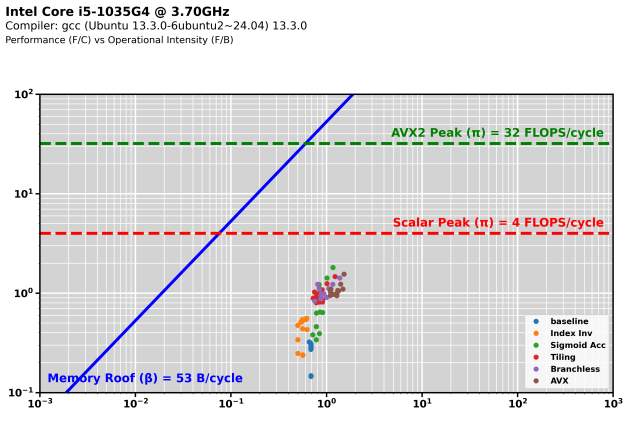

Fig. 5. Roofline plot for the SLU

6. LAYER 3: CLIFFORD GROUP NORMALISATION

6.1. Background on the algorithm

The group normalization (GN) acts on the input data of the shape $X \in \mathbb{R}^{B\times C\times D\times I}$ (batch B, channels C, data D, blades I). The channels are split into groups. Normalization is applied within each group, operating on multivector chunks of (group size, D). The algorithm proceeds in six steps:

1. Centering. To center the data around 0, the empirical mean is calculated within each group and subtracted.

2. Covariance Calculation. From each group (dimensions: (group size, D, I)) we calculate the $I \times I$ covariance matrix:

$$\Sigma = \frac{1}{D * \text{ group size}} X^T X,$$

3. Finding Maximums. Within each covariance matrix the maximum entry scaled by ε is added along the diagonal. This is done to increase the numerical stability of step four.

4. Decomposing Covariance. Each covariance matrix is then decomposed with Cholesky decomposition: $\Sigma = LL^T$

5. Back substitution. Back substitution is used to calculate (L is triangular): $\hat{X} = L^{T^{-1}} X$

6. Scaling. Optionally a user can provide an $I \times I$ weight matrix and a bias for each group to scale the normalized multivectors.

Cost model. The FLOPS are counted with a macro, where addition, multiply and division are counted as one FLOP and a square root is counted as 30 FLOPS.

Memory footprint. The computation needs the following memory (simplified):

1. input and output: (B, C, D, I)
2. covariance (temp): (B, number of groups, I, I)
3. weights: (number of groups, I, I)
4. temporary memory: (B, number of groups, I, I) and (B, C, D, I) (different uses across versions)

6.2. Optimization performed

Baseline Implementation. The baseline is directly adapted from the Python version [4]. The Cholesky decomposition code was taken from [5].

Profiling and Analysis.

Each of the 6 computation steps discussed above has been optimized; in addition, some cache optimizations have been performed.

Optimization 1: Centering. For the 3D case ($I = 8$) we use AVX 256 vectors holding 8 floats, this removes the inner most loop from the baseline. To increase the ILP we introduce 8 accumulators. The 2D case is handled analogs by just using AVX 128 vectors holding 4 floats each. The 1D case actually uses AVX 256 vectors with 2 accumulators, where one vector now spans 4 rows of the matrix (requiring shuffling in the end). The subtraction of the mean has also been fused into the mean calculation loop.

Optimization 2: Covariance Calculation. The baseline does a transpose into a temporary array, this is avoided in the optimized version. Given the input X, we can calculate the covariance row by row, where each row in the result requires iterating all groupsize $*$ D rows of X. In the 3D case we use AVX 256 vectors which conveniently hold one row of X. To increase ILP we use 8 accumulators. The 2D case is identical but uses AVX 128 vectors. In the 1D version we use 2 scalar accumulators for each of the two columns.

Optimization 3: Finding Maximums. For the 3D case, we use AVX 256 vectors to accumulate the maximum for each column, with a reduction. For the 2D and 1D case we simply use the baseline version, as the matrices are quite small.

Optimization 4: Decomposing Covariance. For the Covariance (and Back substitution) we were presented with problems where it is difficult to optimize just one computation (ie. one decomposition), because there are inherent data dependencies throttling throughput. Our way to overcome this challenge is to permute the data, which enables us to do multiple Cholesky decompositions / Back substitutions at once.

In the case of the Cholesky Decomposition, we look at the covariance data in chunks of 8 covariance matrices. We permute each chunk in such a way that all elements with the same matrix indices are contiguous in memory. This allows us to use AVX 256 vectors to trivially do 8 decompositions at once.

Optimization 5: Back substitution. In the Back substitution we can use the same general method as used for the Cholesky Decomposition. We specifically do 32 Back

substitutions at once with 4 AVX 256 vectors. The decision to use 4 AVX vectors at once was taken to increase the ILP.

Optimization 6: Scaling. In the baseline we used to replicate the (I_0, I_1, number of groups) weights across a (B, number of groups, I_0, I_1) array, this is quite inefficient. In the optimization we avoid this and just access the weights without replicating along the batch dimension.

In the 3D case we keep the size of the weight array but bring it to the shape (number of groups, I_1, I_0), this allows us to go column by column in the matrix multiplication using AVX 256 vectors. We also use 4 accumulators to increase the ILP. The 2D case is handled very similar using AVX 128 vectors and 2 accumulators to increase the ILP. The 1D case uses a fully unrolled scalar version.

Optimization 7: Cache Optimizations. The computation often has an outer loop iterating over the batch dimension and accessing a somewhat small and contiguous data chunk in the loop body. This is already pretty efficient. But, when the batch size becomes large enough, we start to evict data form the cache across these outer loops. To improve this problem, we have fused many of these outer loops.

6.3. Experimental results

The Performance Plot in Figure 6 shows the 4 most significant optimizations. They build on each other in the order presented, ie. the last mention includes all other optimizations.

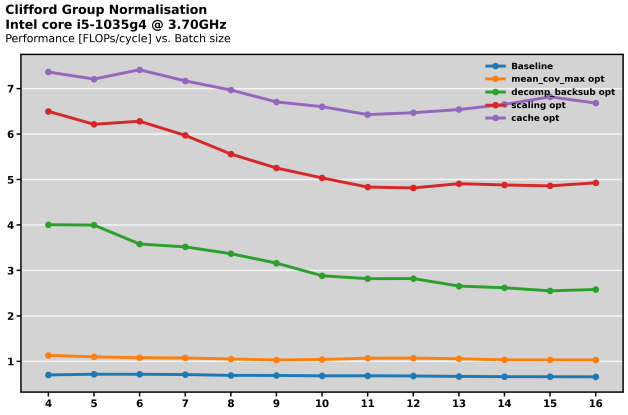

Fig. 6. Performance plot for the Group Normalization

Mean_cov_max opt. introduces Optimizations 1-3. By just optimizing this part and can push beyond 1 FLOP/cycle even when the other parts are not optimized.

Decomp_backsub opt. applies Optimizations 4 and 5 on top. By reducing the effect of the inter iteration data dependencies we can achieve up to 4 FLOPS/cycle with this version.

Scaling opt. removes the last unoptimized part by introducing Optimization 6. Reducing redundant memory replication and optimizing the $I \times I$ matrix multiplication allows a performance up to 6 FLOPS/cycle.

Version `cache opt`. applies the changes outlined in Optimization 7 across the entire computation. By reducing cache misses we can gain a little more and reach 7 FLOPS/cycle.

Roofline Analysis and Bottlenecks. The Roofline plot shown in Figure 7, indicates that while the computation is compute bound, we do not reach the roof. Some of the computations have inherent data dependencies (Cholesky Decomposition and Back substitution). Increasing throughput in these parts of the computation is only possible by permuting the data, which introduces parts that are likely memory bound, even if the entire computation is not memory bound.

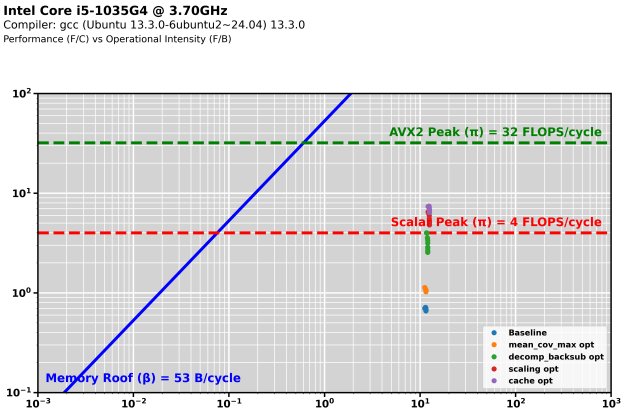

Fig. 7. Roofline plot for the Group Normalization

7. CONCLUSIONS

Our evaluation shows that for inference, the Clifford Linear Projection responds best to precomputed kernel caching, AVX2 FMAs and k-accumulators and is in a clearly compute-bound regime at roughly 7 FLOP/cycle. We can additionally remove the kernel construction for speed-up if construction should be accounted for. Whereas the Multivector Sigmoid Linear Unit, even after dropping its zero-filled buffer, swapping the exponential for a 512-entry LUT, and tiling the sparse gather kernel into L1, the performance stays low due to irregular gathers that still saturate the load ports; by contrast, Clifford Group Normalization gains most from batching multiple Cholesky decompositions / back substitutions and fully vectorizing the surrounding small-matrix code, reaching about 7 FLOP / cycle, yet the serial dependencies of the triangular solves and the extra permutation traffic leave a modest gap to the architectural roof.

8. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

Anirudhh. Produced detailed roofline plots to guide performance analysis. Implemented all linear layer optimizations (besides kernel-caching), including the extensive no-kernel re-design and implementation.

Konstantinos. Developed the end-to-end benchmarking pipeline, established the linear baseline, and diagnosed and fixed bugs in the linear layer optimizations. Worked alongside Mikkeline for the AVX optimizations of her layer.

Mikkeline. Wrote the C wrapper interface, implemented the Sigmoid baseline and drove further Sigmoid kernel optimizations to enhance vectorized throughput, applied AVX-accelerated Sigmoid optimizations.

Simon. Built the group normalization baseline and engineered a suite of group normalization optimizations to accelerate convergence and runtime.

9. REFERENCES

[1] Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković, "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges," *arXiv preprint arXiv:2104.13478*, 2021.

[2] Alexander Ruhe, Johannes Brandstetter, and Max Welling, "Geometric clifford algebra networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[3] Johannes Brandstetter, Max Welling, and Matthias Witek, "Clifford neural layers for pde modeling," in *International Conference on Learning Representations (ICLR)*, 2023.

[4] Johannes Brandstetter, Rianne van den Berg, Max Welling, and Jayesh K. Gupta, "Clifford neural layers for PDE modeling," *arXiv preprint*, vol. arXiv:2209.04934, 2022.

[5] Jonathan Irwin, "In-place cholesky decomposition for l in a = ll^t," 2014.