HYPERHADAMARD: LAYOUT-AWARE HADAMARD TRANSFORMS ON TENSOR CORES

Anonymous authors
Paper under double-blind review

ABSTRACT

Low-bit quantization is an appealing way to cut the memory and bandwidth cost of large language models. But activation outliers make few-bit quantization unstable, and rotation-based methods address them by mixing channels before quantization. The Hadamard transform is the rotation of choice because it admits a fast algorithm. Tensor-core FWHT kernels made these rotations cheap enough to deploy, but at large rotation dimensions existing kernels sustain only 42–77% of achievable memory bandwidth, even though the transform is memory-bound. We identify intermediate-layout materialization as a source of excess instructions and dependency stalls. We propose HyperHadamard, which treats butterfly stages as rotations along the binary axes of a hypercube and assigns those axes to tensor-core register layouts at compile time, so intermediate transposes become metadata rather than data movement. At rotation dimensions large enough for these overheads to dominate, HyperHadamard is up to 1.59× faster than HadaCore, the tensor-core FWHT used in vLLM, and stays within 1.7% of achievable memory bandwidth. Together with a gather remap and a fused sign load, a true-FP32 instantiation of the same schedule speeds up a billion-dimensional randomized gradient projection used for training-data selection by 1.75× while remaining bit-exact. The same design supports non-power-of-two transforms built from available catalog Hadamard matrices, whereas prior fast kernels expose only power-of-two and selected fixed-factor sizes. Integrated into vLLM, a Llama-3.1-8B QuIP W4A16 model produced by llm-compressor runs 14.7× faster end to end than vLLM's dense GEMM path and 1.9× faster than the same factorization assembled from Dao-AILab kernels and cuBLAS, with no measurable change in perplexity or zero-shot accuracy.

1 INTRODUCTION

Rotation-based quantization has become a standard way to run large language models at low precision. Methods such as QuaRot (Ashkboos et al., 2024), QuIP# (Tseng et al., 2024), and SpinQuant (Liu et al., 2025) multiply weights and activations by an orthogonal matrix before quantization, which spreads the energy of outlier channels across all coordinates so that a coarse grid represents them with less error. Hadamard matrices are the usual choice because the fast Walsh–Hadamard transform (FWHT) applies them in $O(N \log N)$ operations rather than the $O(N^2)$ of a dense matrix product. The same transform is a standard sketching operator in randomized numerical linear algebra (Tropp, 2011). Some rotations can be merged into adjacent weights, but others must be applied to activations during inference, and their cost adds to every forward pass. A rotation implemented as a dense half-precision matrix multiplication can cancel much of the speedup that low-bit arithmetic is meant to deliver (Agarwal et al., 2024).

The FWHT is memory-bound: an ideal kernel runs at the speed of reading its input once and writing its output once. For short transforms, the state of the art achieves this. On an H100, HadaCore (Agarwal et al., 2024), the tensor-core kernel that vLLM uses for these rotations, reaches the achievable memory bandwidth for transforms up to length 4096. For longer transforms its throughput declines, and at length 32768 it leaves more than a third of the achievable bandwidth unused. Across the available CUDA and Triton kernels, throughput at this length ranges from 42% to 77% of the achievable bandwidth (Figure 2).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Long and irregular transforms are not a corner case. A rotation mixes information only within the block it spans. Several quantization methods therefore rotate an entire hidden or MLP dimension: QuaRot applies an online Hadamard transform across the full MLP dimension, and QuIP# applies randomized Hadamard transforms across full weight dimensions (Ashkboos et al., 2024; Tseng et al., 2024). In randomized numerical linear algebra, a sketch of a long vector requires a transform as long as the vector; the gradient projector in Influence Distillation applies transforms of length 32768 to a vector with a billion entries (Nikdan et al., 2026). Many model dimensions are also not powers of two, such as the 14336-wide MLP of Llama-3.1-8B (Grattafiori et al., 2024). Quantization toolkits handle these dimensions with Kronecker products of catalog Hadamard matrices (Tseng et al., 2024; AI & vLLM Project, 2024). Existing fast kernels cover only power-of-two lengths and a few fixed factors, not catalog factors such as H_{224} and H_{256}, so inference engines such as vLLM fall back to dense matrix multiplications (Kwon et al., 2023). The llm-compressor documentation reports that dense rotations raise the latency of a quantized 1B model from 0.45 to 1.29 s relative to HadaCore (AI & vLLM Project, 2024).

The difficulty lies in how GPUs execute the transform. An FWHT of length 2^d consists of d butterfly stages, and each stage combines elements that lie farther apart than in the stage before. The Dao-AILab kernel executes these stages on ordinary arithmetic units and exchanges data between threads through warp shuffles and shared memory (Dao-AILab, 2026). Tensor cores can apply several stages in one matrix multiplication, but only on small fixed-size tiles, and each multiplication returns its result distributed across threads in a different arrangement from its input. HadaCore uses tensor cores while following the stage order of the classical recursive algorithm (Agarwal et al., 2024), so it must rearrange data between groups of stages before it can continue. The number of rearrangements grows with the transform length.

We introduce HyperHadamard, a GPU kernel that computes exact Hadamard transforms from length 256 to 32768 and at non-power-of-two model dimensions such as 14336, at nearly the speed of memory. Its design rests on one observation: the stages of a Hadamard transform commute, so they can be applied in any order without changing the result. HyperHadamard uses this freedom to choose, at every step, the stages that match the arrangement in which the hardware left the data. Rearranging data then reduces to bookkeeping at compile time, and even the longest transforms exchange data through shared memory only once. A Kronecker factorization extends the same design to non-power-of-two orders in FP16 and BF16, and a true-FP32 variant covers power-of-two lengths. Our contributions are:

- **A new GPU algorithm for the Hadamard transform** (Section 3). We formulate the FWHT as commuting rotations along the binary axes of a hypercube and schedule them to match the register layouts that tensor-core instructions produce. A typed register fragment records the physical position and completion state of every axis, carries each output layout into the next instruction, and rejects schedules that leave an axis unrotated.
- **State-of-the-art performance against existing kernels** (Section 4.1). On an H100, HyperHadamard stays within 1.7% of the empirical memory roofline from $N = 256$ to $N = 32768$. At $N = 32768$ it is 1.52–1.59× faster than HadaCore and 1.29–2.34× faster than the Dao-AILab, FlagGems, and open-source Triton kernels. At $N = 32768$, the FP32 kernel is 1.46× faster than the Dao-AILab FP32 kernel and bit-exact with it. Hardware profiles tie the low-precision gain to HadaCore's 3.5–3.6× larger instruction count and its shared-memory queue stalls, and materializing even one intermediate layout slows our kernel by 1.8–5.9%.
- **Integration into vLLM inference** (Section 4.2). With catalog Hadamard factors, HyperHadamard runs the released Llama-3.1-8B QuIP W4A16 checkpoint in vLLM 14.70× faster than vLLM's dense GEMM path, with unchanged perplexity and zero-shot accuracy. It is also 1.92× faster than a structured baseline that we assemble from Dao-AILab kernels and cuBLAS for the same factorization. Replacing vLLM's embedded HadaCore in a power-of-two checkpoint improves prefill throughput by up to 1.05×.
- **Randomized projection** (Section 4.3). Built on the FP32 kernel, with an exact gather remap and random signs fused into the kernel's load, our projector for Influence Distillation, which transforms a 2^{30}-element vector, is 1.75× faster than the released implementation with bit-exact output.
- **Rotation size under aggressive quantization** (Section 4.4). For Mistral-7B with INT4 activations, full-dimension rotations lower WikiText-2 perplexity relative to H_{128} blocks

by 6.4% with INT3 weights, monotonically in the rotation size, and by 39.6% with INT2 weights. With BF16 or INT8 activations, rotation size brings no gain.

2 Related work

Hadamard rotations in quantized models. QuaRot and QuIP# rotate weights and activations with Hadamard transforms so that outlier channels do not dominate the quantization scale, and SpinQuant shows that the choice of rotation affects quantized accuracy (Ashkboos et al., 2024; Tseng et al., 2024; Liu et al., 2025). Later work replaces the fixed Hadamard with learned butterfly rotations under 2-bit quantization (Xu et al., 2025) and applies rotations to aggressive KV-cache quantization (Su et al., 2025; Zandieh et al., 2026; Saxena & Roy, 2025). For microscaling FP4 formats, rotations are often confined to small blocks that match the scaling groups (NVIDIA Corporation, 2026b; Tseng et al., 2025). These methods specify which rotation to apply; HyperHadamard is concerned with applying exact Hadamard rotations quickly, including the full-dimension and non-power-of-two rotations that some of these methods and their released checkpoints use (AI & vLLM Project, 2024).

Randomized Hadamard transforms. The subsampled randomized Hadamard transform is a standard sketching operator with subspace-embedding guarantees, used for low-rank approximation, approximate matrix products, Nyström methods, and structured orthogonal random features (Tropp, 2011; Boutsidis & Gittens, 2013; Balabanov et al., 2023; Yu et al., 2016). Influence Distillation uses a randomized Hadamard projector to compress billion-dimensional gradients for data selection (Nikdan et al., 2026). These applications apply transforms whose length is set by the data, which is the setting where existing kernels leave bandwidth unused.

GPU FWHT kernels. The Dao-AILab kernel implements the FWHT in CUDA for FP32, FP16, and BF16 up to length 32768, exchanging data between butterfly stages with warp shuffles and shared memory (Dao-AILab, 2026). It pads other lengths to the next power of two, and separate variants handle lengths with a small fixed non-power-of-two factor. HadaCore maps 16-point Hadamard blocks onto tensor-core matrix multiplications for FP16 and BF16 and is the transform backend in vLLM (Agarwal et al., 2024). FlagGems provides a Triton implementation, and an open-source Triton kernel also uses tensor cores (contributors, 2024; Feeney, 2025). These kernels follow the stage order of the recursive algorithm and rearrange data between groups of stages. None implements the catalog Hadamard factors used by released QuIP checkpoints.

Layout abstractions and IO-aware kernels. CuTe represents tensor layouts as composable functions from logical coordinates to threads, registers, and memory (Thakkar et al., 2023), and Triton's linear layouts express the same mappings as linear maps over the bits of the register, lane, and warp indices (Zhou et al., 2026). ThunderKittens provides tile abstractions at the warp and block level for writing fast kernels (Spector et al., 2024). These systems describe how a tensor is distributed and convert between the layouts that a fixed computation requires. In the IO-aware line of work, FlashAttention and FlashAttention-2 gain speed by reorganizing an exact computation to reduce data movement (Dao et al., 2022; Dao, 2024).

3 HyperHadamard

Let X be a batch of rows of length $N = 2^d$, where $d = \log_2 N$ is the number of binary axes. For each row x, HyperHadamard computes the normalized transform $H_N x$ in one CUDA kernel. An FWHT performs $O(N \log N)$ additions, and any implementation must read and write all N values. For the large batched workloads we target, tensor cores make the low-precision arithmetic inexpensive enough that the standalone operation is memory-bound. The true-FP32 path uses CUDA-core butterflies but is likewise memory-bound in the large batched regime. Our goal is therefore to read each input once, perform all butterfly stages on chip, and write each output once.

HyperHadamard expresses the FWHT as rotations along independent binary axes, maps those axes to tensor-core register layouts at compile time, and carries each hardware-produced layout directly into the next rotation (Figure 1). Transforms that exceed the register budget use two register phases separated by one shared-memory exchange.

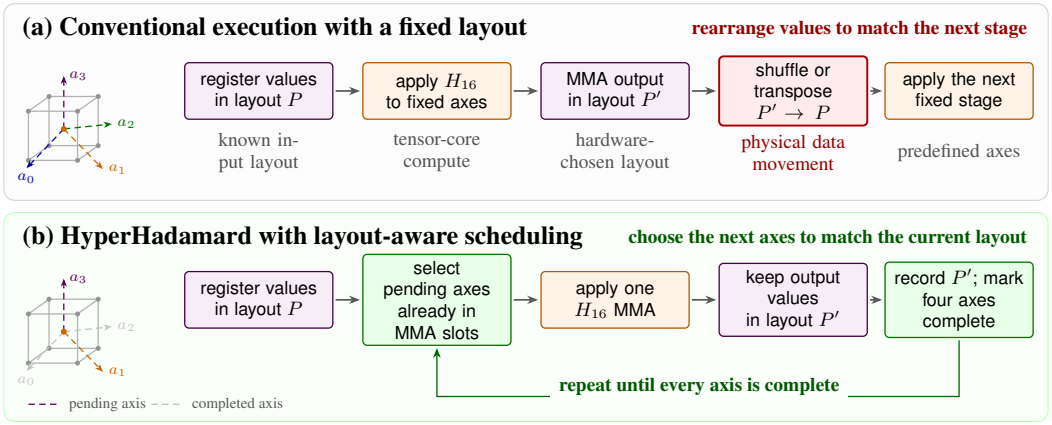

Figure 1: HyperHadamard schedules the transform around the hardware layout. (a) Conventional execution applies stages in a fixed axis order, so it must shuffle or transpose each MMA output back to the layout the next stage expects. (b) Because FWHT axes commute, HyperHadamard applies the next MMA to pending axes that already sit in its input slots; only the compile-time layout and completed-axis set change.

3.1 FWHT AS COMMUTING HYPERCUBE-AXIS ROTATIONS

Let the transform length be $N = 2^d$ and index an element by $b = (b_0, \ldots, b_{d-1}) \in \{0,1\}^d$. For $s \in \{0,1\}$, let $b^{(k \leftarrow s)}$ denote b with bit k replaced by s. A normalized butterfly along axis k is the operator

$$(R_k x)_b = \frac{x_{b^{(k\leftarrow 0)}} + (-1)^{b_k} x_{b^{(k\leftarrow 1)}}}{\sqrt{2}}. \tag{1}$$

It applies a sum/difference to every pair of hypercube vertices that differs only in bit k. Equivalently,

$$R_k = I_2^{\otimes(d-1-k)} \otimes H_2 \otimes I_2^{\otimes k}. \tag{2}$$

Lemma 1 (Axis-order invariance)**.** *For any permutation π of $\{0, \ldots, d-1\}$,*

$$H_N = R_{\pi(0)} R_{\pi(1)} \cdots R_{\pi(d-1)}. \tag{3}$$

Thus a schedule computes the FWHT if it applies every logical-axis rotation exactly once, irrespective of their order.

Proof. For $i \neq j$, R_i and R_j act on distinct factors of the tensor product, so $R_i R_j = R_j R_i$. The conventional factorization is $H_N = H_2^{\otimes d} = R_0 R_1 \cdots R_{d-1}$. Repeatedly commuting adjacent operators produces any ordering π without changing the product. □

Lemma 1 gives the implementation its scheduling freedom: at each step we rotate axes that already sit where the tensor-core instruction expects them, so the conventional order $0, 1, \ldots, d-1$ is never recovered.

3.2 d-DIMENSIONAL REGISTER FRAGMENTS

A tensor-core instruction assigns matrix coordinates to lane and register coordinates according to a fixed thread–value layout, and its output layout differs from its input layout. Restoring a canonical layout after every instruction would require additional register shuffles or shared-memory transposes. We therefore introduce a warp-level abstraction called a *Fragment*. A d-dimensional Fragment represents a logical $2 \times \cdots \times 2$ tensor distributed across the registers of one warp, so each value has two physical coordinates: the lane that owns it and its position among that lane's registers.

HyperHadamard instead represents the logical $2 \times \cdots \times 2$ tensor and its physical placement together. Physical positions $p = 0, \ldots, 4$ denote the five bits of the warp-lane ID, and $p \geq 5$ denote bits of the thread-local register index. We use the convention $P[p] =$

the logical hypercube axis occupying physical position p. The register fragment is the typed object $\mathcal{F} = (v, P, S)$, where v contains the values held by the warp. The compile-time permutation P records where each logical axis resides, while S records whether each axis is pending or complete. Only v is stored at runtime. Appendix A gives the corresponding C++ type. The type serves two purposes. First, it carries the layout produced by one instruction directly into the next instruction. Changing P relabels the resident values without moving them. Second, S lets the compiler reject a schedule that leaves an axis pending, and a completed axis becomes a batch axis, so no axis is rotated twice. The same metadata generates memory addresses: given the stride of each logical axis, P determines the address owned by each lane and register. We check coalescing and shared-memory bank conflicts at compile time. A physical register reordering is emitted only when two thread-local axes must exchange positions; it does not require a warp shuffle or shared memory.

3.3 Four axis rotations with one MMA

Tensor cores apply four axis rotations in one instruction, but only on a fixed tile and with a fixed output layout. The Fragment abstraction describes such an instruction by the logical axes it transforms and the physical layout it produces. Four axis rotations form the matrix $H_{16} = H_2^{\otimes 4}$. We reshape 128 resident values as $B \in \mathbb{R}^{16\times 8}$ and compute $D = H_{16}B$. The eight columns of B are independent length-16 vectors, so one `mma.m16n8k16` instruction applies the same four FWHT stages to all 128 values. The H_{16} operand is fixed and generated directly in the register layout required by the instruction; its entries include the normalization factor $1/4$.

The remaining question is where the input and output values reside. Write a row of H_{16} using four bits $(m_0, \ldots, m_3)$, its contracted column using $(k_0, \ldots, k_3)$, and a column of B using three bits $(n_0, \ldots, n_2)$. Thus $B[k, n]$ is the input and $D[m, n]$ is the output. NVIDIA specifies the per-lane registers for both operands (NVIDIA Corporation, 2026c); in our notation they are

physical position	ℓ_0	ℓ_1	ℓ_2	ℓ_3	ℓ_4	r_0	r_1
data operand B	k_1	k_2	n_0	n_1	n_2	k_0	k_3
output D	n_1	n_2	m_0	m_1	m_2	n_0	m_3

Here ℓ_i is a lane-ID bit and r_i selects one of the values held by that lane. For a Hadamard multiply, output axis m_i is the transformed version of input axis k_i and therefore retains the same logical-axis identity. In the convention above, the two rows are $P_B = (1, 2, 4, 5, 6, 0, 3)$, $P_D = (5, 6, 0, 1, 2, 4, 3)$. Expressing each output position in input physical coordinates gives $R_{\text{mma}} = P_B^{-1} \circ P_D = (3, 4, 5, 0, 1, 2, 6)$, which Appendix A derives entry by entry. For a fragment with current layout P, our primitive has the compile-time semantics

$$\text{rotate4} : \mathcal{F}(P, S) \longrightarrow \mathcal{F}(P \circ R_{\text{mma}}, S'), \qquad (P \circ R)[p] \equiv P[R[p]], \tag{4}$$

where S' marks $P[0]$, $P[1]$, $P[5]$, and $P[6]$ complete. Dimensions beyond the seven axes of one MMA tile are treated as batch axes. A compile-time assertion rejects a schedule whose final fragment contains a pending axis. Appendix A gives the implementation's encoding of these layouts and walks through the complete H_{256} schedule. The same register path supports H_{512}, H_{1024}, and H_{2048} by first applying packed add/subtract butterflies to additional thread-local axes and then using the eight-axis schedule. Normalization is folded into these butterflies and the fixed MMA operand, so no separate scaling pass is needed.

3.4 Scaling to large transforms

The register-only schedule above becomes impractical as the transform grows: each thread must hold more values, and at $N \geq 4096$ the complete transform no longer fits within the register budget. We therefore partition the logical axes into two sets, A_1 and A_2. By Lemma 1, we may complete the axes in A_1, exchange the partial results, and then complete the axes in A_2.

Warp count. The FP16/BF16 kernel uses a single warp (32 threads) per thread block for every supported power-of-two length, $N = 256, \ldots, 32768$. Because the FWHT has low arithmetic intensity, we do not add warps to raise intra-block concurrency. A warp-local block needs no inter-warp synchronization such as `__syncthreads()`, and a Fragment stays distributed across one warp's registers. For short transforms the warp processes multiple rows; for large transforms it owns one row and its shared-memory phase buffer.

Two-phase computation. For $N \geq 4096$, HyperHadamard splits the logical axes into two register-compute phases separated by one shared-memory exchange: global memory $\rightarrow$ shared memory $\rightarrow \mathcal{F}_1 \rightarrow$ shared memory $\rightarrow \mathcal{F}_2 \rightarrow$ global memory. Phase 1 loads contiguous fragments, rotates the axes in A_1, and stores the partial transform in shared memory. After one warp synchronization, phase 2 reads the same buffer through a strided view that places the axes in A_2 directly into the register positions used by `rotate4`. It then rotates the remaining axes and writes the completed transform to global memory.

Pipelined input staging. The global input is read in contiguous chunks. To hide the latency of these loads, the main large-transform path maintains a depth-seven software pipeline using NVIDIA's asynchronous global-to-shared copy instruction (`cp.async`). While phase 1 transforms a ready chunk, copies of later chunks remain in flight.

Eliminating explicit data rearrangement. Section 3.2 removes layout restoration between MMA calls. For large transforms, the same abstraction also removes HadaCore's final output realignment, which uses warp shuffles followed by a shared-memory store and reload (Agarwal et al., 2024). We place logical axes $0, \ldots, 5$, which span 64 contiguous FP16/BF16 values, in the phase-2 Fragment. Its strided view loads each value directly into the lane and register needed by the MMA, whose output can then be written contiguously to global memory. The large-transform path therefore adds no explicit shuffle, transpose, or final shared-memory realignment.

True FP32 on CUDA cores. Tensor-core MMA rounds FP32 operands to TF32, so using it would change the numerical operation. We instead instantiate the same hypercube-axis schedule with a true-FP32 Fragment. Thread-local axes use FP32 add–subtract butterflies, lane axes use `__shfl_xor_sync`, and the compile-time permutation records where each logical axis resides. Large transforms retain the two register phases and one shared-memory boundary. At $N = 32768$, the 128-KiB phase buffer permits only one thread block per SM, so the FP32 kernel uses 16 cooperating warps to expose latency-hiding parallelism while keeping the live fragment small. This path uses neither TF32 nor tensor-core instructions. For transforms that fit in registers, A_2 is empty and the shared-memory exchange is skipped. Algorithm 1 summarizes the resulting dataflow.

3.5 Transforms beyond powers of two

Model dimensions are not always powers of two. When $N = q2^k$ and a Hadamard matrix of order q is available, we use the Kronecker factorization, $H_N = H_q \otimes H_{2^k}$. Viewing the input as a $q \times 2^k$ array, this applies a power-of-two FWHT along the power-of-two dimension and then mixes the q resulting groups with H_q. For dimensions 14336 and 28672, we use $q = 28$. The H_{28} stage is a small dense tensor-core multiply padded to order 32; only the 28 valid outputs are stored. Both factors execute in the same kernel and reuse the same shared-memory buffer, so the intermediate array is not written to global memory. Section 4 measures these kernels against existing implementations and in three applications.

4 Evaluation

4.1 Operator-level performance

Setup. We evaluate an NVIDIA H100 80GB HBM3 GPU (Choquette, 2023) (driver 590.48.01) with PyTorch 2.9.1+cu128, CUDA 12.9.1, and GCC 12.2.0. Each point processes 2^{30} FP16 or BF16 elements. Figure 2 reports five fresh paired processes per baseline, with 20 timed calls per method and randomized method order in each process. Compilation and JIT time are excluded. All methods compute the same normalized full transform through their public interfaces. HadaCore and the open-source Triton kernel use tensor cores; the Dao-AILab and FlagGems kernels do not. We define effective bandwidth as one read and one write of the input tensor divided by kernel time. As references, we report the 3.35 TB/s HBM3 datasheet peak and an empirical roofline of 3.03 TB/s, measured with an elementwise read–modify–write kernel on the same H100 model. Appendix C gives the measurement provenance and confidence intervals.

FP16 and BF16. HyperHadamard stays within 1.7% of the empirical roofline at every length, sustaining 2.98–3.03 TB/s (Figure 2). For $N \leq 4096$, HadaCore also reaches the roofline, and the two

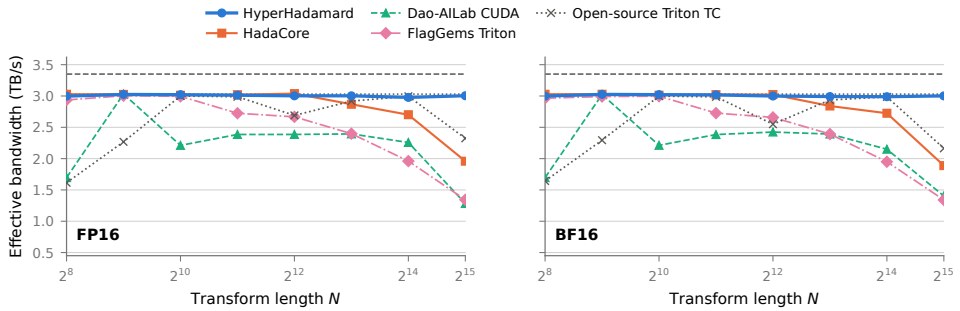

Figure 2: Effective bandwidth on H100, counting one input read and one output write. The dashed line is the 3.35 TB/s HBM3 datasheet peak; the dotted line is the 3.03 TB/s empirical read–modify–write roofline actually achievable on this GPU. HyperHadamard stays near the achievable ceiling across the sweep. HadaCore is similarly efficient at small N but loses bandwidth as the transform grows; the other baselines fall further below the ceiling.

kernels are at parity (0.988–1.003×). The gap opens as N grows. At $N = 32768$, HadaCore falls to 62–65% of the roofline, and HyperHadamard is 1.524–1.535× faster in FP16 and 1.589–1.591× faster in BF16 across five repetitions. At the same length, the FP16/BF16 speedups are 2.34/2.14× over Dao-AILab, 2.24/2.24× over FlagGems, and 1.29/1.39× over the open-source Triton kernel. The advantage also requires enough independent rows. In a separate ten-process sweep at fixed $N = 32768$, HyperHadamard overtakes HadaCore once the tensor holds 2^{23} elements and reaches 1.548/1.601× at 2^{30} elements. Before timing, each external baseline matches HyperHadamard's output within 0.01 in FP16 and 0.0625 in BF16 (largest observed differences 0.00391 and 0.04688). Appendix D repeats the sweep on an A100 with the same pattern: within 3.5% of HadaCore up to $N = 16384$ and a 1.34–1.37× speedup over HadaCore at $N = 32768$.

FP32. The FP32 kernel performs the transform in FP32 arithmetic and is bit-exact with the Dao-AILab FP32 kernel from $N = 256$ to $N = 32768$. At $N = 32768$ and 2^{28} total elements, it reduces latency from 1.0762 to 0.7367 ms, a 1.461× speedup, and sustains 2.915 TB/s, 96% of the empirical roofline. At $N = 4096$ and $N = 16384$ the two kernels are at parity, because the Dao-AILab kernel already reaches the memory roofline at those lengths. Appendix E profiles this kernel: at $N = 32768$ it uses 80 registers per thread against Dao-AILab's 254, raising achieved occupancy from 12.4% to 23.9%.

Source of the FP16/BF16 speedup. At $N = 32768$, HyperHadamard executes 3.47× fewer warp instructions than HadaCore in FP16 and 3.60× fewer in BF16 under Nsight Compute (Figure 3). It executes fewer instructions at every profiled length, but the saving speeds the kernel up only once HadaCore falls below the roofline: 1.05× at $N = 8192$, 1.10× at $N = 16384$, and 1.52–1.59× at $N = 32768$ (Figure 2). It issues about half as many load/store-unit wavefronts (memory transactions issued by shared-memory and global-memory instructions), and the fraction of stall samples caused by a full memory input/output queue, which serves shared-memory instructions, falls from 5.70–5.80% to 0.07–0.11%. To bound the remaining headroom, we build an arithmetic-free kernel with the same launch geometry and data movement. The production kernel is only 1.055× slower than this kernel in FP16 and 1.058× slower in BF16, so at large N it runs close to the floor set by its own data movement.

A within-kernel ablation isolates the role of the layout. Forcing the kernel to materialize its phase-2 views in memory slows $N = 32768$ by 1.75% in FP16 and 5.89% in BF16, adds 3.6–7.4% more instructions, and reduces physical DRAM throughput by 5.6–14.7%. Moving one logical axis across the phase boundary leaves latency unchanged or reduces it by up to 1.5%. Avoiding physical movement of intermediate views therefore contributes to the speedup, whereas the phase boundary can shift by one axis without cost.

4.2 End-to-end inference in vLLM

Catalog Hadamard rotations. The released `nm-testing/Llama-3.1-8B-Instruct-quip-w4a16` checkpoint, produced with llm-compressor's QuIP trans-

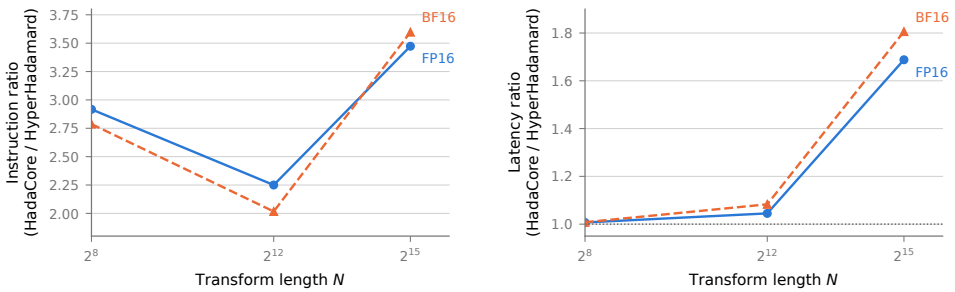

Figure 3: Matched H100 Nsight Compute profile against HadaCore at $N = 256$, 4096, and 32768. Each point is one profiled launch. HyperHadamard executes 2.0–3.6× fewer dynamic warp instructions (left), but the latency ratio (right), measured at Nsight Compute's locked clocks, is large only at $N = 32768$ and exceeds the wall-clock speedups in Figure 2.

forms (AI & vLLM Project, 2024), rotates activations with Hadamard matrices of orders 14336, 4096, and 1024 built from the catalog matrices H_{224} and H_{256}. Existing FWHT kernels implement only the Sylvester construction, so stock vLLM 0.14.0 applies these rotations as FP64 dense matrix multiplications. HyperHadamard computes each rotation with its native factorization (Section 3.5) followed by a small BF16 basis correction, which reproduces the checkpoint's matrix exactly in exact arithmetic (Appendix F).

On 64 prompts with 2,048 input tokens and one output token, a prefill-dominated workload, HyperHadamard is 14.70× faster than stock vLLM end to end (95% CI 14.69–14.72× over six order-balanced process pairs), raising throughput from 1,330 to 19,550 tokens/s. The dense FP64 transforms account for 95% of stock GPU kernel time, and this dense path is the reference against which llm-compressor reports its HadaCore speedups (AI & vLLM Project, 2024).

To separate the kernel from the factorization, we also run the same factorization assembled from existing components: pinned Dao-AILab kernels for the power-of-two factor and BF16 cuBLAS (NVIDIA Corporation, 2026a) for the catalog factor. Against this structured baseline, HyperHadamard reduces end-to-end time from 12.87 to 6.72 s, a 1.915× speedup (95% CI 1.913–1.917×). Model quality matches stock vLLM: full-corpus WikiText-2 perplexity moves from 7.94542 to 7.94563 (+0.0026%), and combined zero-shot ARC-Easy and ARC-Challenge accuracy is 2499/3548 on both (Merity et al., 2016; Clark et al., 2018).

Power-of-two rotations. The released Llama-3.2-1B QuIP W4A16 checkpoint uses power-of-two rotations, which vLLM runs with its embedded HadaCore kernel. Replacing only the CUDA implementation behind `_C::hadacore_transform`, with the checkpoint, quantized GEMMs, scheduler, and engine unchanged, speeds up the same 64-prompt workload by 1.022–1.052× across scheduler token budgets from 2,048 to 16,384; every 95% interval from six order-balanced process pairs excludes parity (Appendix F). A timed trace attributes the gain to the transforms, which fall from 126.2 to 85.6 ms of an 819.3 ms interval and so predict the measured 1.052×. In a batch-8 decode workload, HyperHadamard reaches 0.978× the throughput of HadaCore, consistent with the small-batch boundary of the operator sweep.

4.3 Influence Distillation

Influence Distillation selects training data by compressing approximately one billion FP32 parameter-gradient coordinates with a randomized Hadamard projector (Nikdan et al., 2026). Its released `Had1GProjector` pads the vector to 2^{30} elements, views it as a 32768×32768 matrix, applies two H_{32768} transforms separated by physical transposes, and gathers 4096 coordinates. Table 1 reports the projector with the true-FP32 kernel of Section 3.

Substituting the kernel alone gives 1.088×, because the two physical transposes dominate the projector. We therefore optimize the projector further. Remapping the gather indices removes the second transpose and benefits both kernels; with it, HyperHadamard is 1.123× faster than the Dao-AILab kernel (21.896 versus 24.581 ms). Because we control the kernel's load stage, we also fold the Rademacher signs into it, which removes a separate pass over the 4 GiB vector. The resulting

Table 1: The released Influence Distillation projector and bit-exact optimizations on H100. Every row includes padding the one-billion-coordinate FP32 input to 2^{30} and gathering 4096 outputs.

Implementation	Latency (ms)	Speedup
Released Dao-AILab + two transposes	33.215	1.000×
HyperHadamard + two transposes	30.526	1.088×
Dao-AILab + transpose-2 gather remap	24.581	1.351×
HyperHadamard + gather remap	21.896	1.517×
HyperHadamard + remap + fused signs	**19.012**	**1.747**×

Table 2: Mistral-7B W3A4 WikiText-2 perplexity as a function of rotation domain (H_k blocks or the full dimension). Rotations run as FP64 hooks, so the table reports quality only.

Rotation	H_{128}	H_{256}	H_{512}	H_{1024}	Full
Perplexity	6.98561	6.78784	6.71592	6.61284	**6.54047**
Improvement vs. H_{128}	–	2.83%	3.86%	5.34%	**6.37%**

projector is 1.747× faster than the released implementation, with bit-exact output. Through the released `pick_first_order` API, five order-balanced pairs retain a 1.744× projector speedup and improve a bounded Jacobian–vector-product and selection run by 1.015× with byte-identical selected examples and weights. The public repository omits the full 200k-candidate experiment, so we use this run only to validate the integration.

4.4 Rotation size under aggressive quantization

The preceding sections show that HyperHadamard makes full-dimension rotations inexpensive. We now ask whether such rotations are useful when quantization error is large. We quantize Mistral-7B-Instruct-v0.3 (Jiang et al., 2023) with GPTQ-calibrated signed INT3 weights (group size 128) (Frantar et al., 2022) and dynamic per-token signed INT4 activations, and vary only the Hadamard rotation domain; calibration data, test tokens, and quantizer are fixed. As Table 2 shows, WikiText-2 perplexity decreases monotonically as the rotation domain grows, and full-dimension rotations improve perplexity by 6.37% relative to H_{128} blocks.

The benefit depends on activation precision. With BF16 or INT8 activations, full-dimension rotations change perplexity by $+0.2$ to $+0.5\%$ relative to H_{128} blocks for both INT3 and INT2 weights. With INT4 activations, they improve it by 6.37% for INT3 weights and by 39.6% for INT2 weights, from 57.15 to 34.50 (Appendix G). Appendix H separates the two full-width rotations: H_{4096} helps on its own, whereas H_{14336} helps only together with H_{4096}. These results cover one model and one integer quantization recipe.

5 Conclusion

We introduced HyperHadamard, a GPU algorithm for the Hadamard transform in quantized inference and randomized projection. It interprets butterfly stages as rotations of independent binary-hypercube axes and tracks their placement in tensor-core register fragments at compile time, so each instruction starts from the layout the previous one produced. On an H100, HyperHadamard is up to 1.59× faster than HadaCore and stays near the memory roofline. In vLLM, it runs the catalog rotations of a released Llama-3.1-8B checkpoint 14.70× faster than the dense GEMM path and 1.915× faster than an operation-matched structured baseline. Its true-FP32 variant accelerates the released Influence Distillation projector by 1.747× with bit-exact output, and on Mistral-7B with INT4 activations, full-dimension rotations improve perplexity over H_{128} blocks by up to 39.6%.

The quantization study covers one model and recipe, and the Influence Distillation study covers the released projector, since the full 200k-candidate experiment is not public. Next, we plan to evaluate HyperHadamard on online serving traces and other GPU architectures, and to fuse it into the quantization and projection kernels around it.

REFERENCES

Krish Agarwal, Rishi Astra, Adnan Hoque, Mudhakar Srivatsa, Raghu Ganti, Less Wright, and Sijia Chen. Hadacore: Tensor core accelerated hadamard transform kernel. *arXiv preprint arXiv:2412.08832*, 2024.

Red Hat AI and vLLM Project. LLM Compressor, 8 2024. URL https://github.com/vllm-project/llm-compressor.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. *Advances in Neural Information Processing Systems*, 37:100213–100240, 2024.

Oleg Balabanov, Matthias Beaupère, Laura Grigori, and Victor Lederer. Block subsampled randomized hadamard transform for Nyström approximation on distributed architectures. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 1564–1576. PMLR, 2023. URL https://proceedings.mlr.press/v202/balabanov23a.html.

Christos Boutsidis and Alex Gittens. Improved matrix algorithms via the subsampled randomized hadamard transform. *SIAM Journal on Matrix Analysis and Applications*, 34(3):1301–1340, 2013.

Jack Choquette. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro*, 43(3):9–17, 2023.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

The FlagOS contributors. Flagos/flaggems: An operator library for large language models implemented in the triton language., 2024. URL https://github.com/flagos-ai/FlagGems.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, volume 2024, pp. 35549–35562, 2024.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Dao-AILab. Fast hadamard transform in CUDA. https://github.com/Dao-AILab/fast-hadamard-transform, 2026. Software, commit e7706faf8d1c3b9f241e36860640ad1dac644ede.

Arthur Feeney. FWHT: A fast walsh–hadamard transform in Triton. https://github.com/arthurfeeney/fwht, 2025. Software, commit 9fcb9fe8dfac000bbebed001809b02b49a55dcdc.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant: Llm quantization with learned rotations. In *International Conference on Learning Representations*, volume 2025, pp. 92009–92032, 2025.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Mahdi Nikdan, Vincent Cohen-Addad, Dan Alistarh, and Vahab Mirrokni. Efficient data selection at scale via influence distillation. *Advances in Neural Information Processing Systems*, 38:143229–143265, 2026.

NVIDIA Corporation. *cuBLAS Library Documentation*, 2026a. URL `https://docs.nvidia.com/cuda/cublas/index.html`. Accessed: September 26, 2026.

NVIDIA Corporation. NVFP4: Transformer engine user guide. `https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/features/low_precision_training/nvfp4/nvfp4.html`, 2026b. Accessed 2026-08-23.

NVIDIA Corporation. Parallel thread execution ISA. `https://docs.nvidia.com/cuda/parallel-thread-execution/`, 2026c. CUDA documentation, accessed 2026-09-26.

Utkarsh Saxena and Kaushik Roy. Kvlinc: Kv cache quantization with hadamard rotation and linear correction. *arXiv preprint arXiv:2510.05373*, 2025.

Benjamin F Spector, Simran Arora, Aaryan Singhal, Daniel Y Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels. *arXiv preprint arXiv:2410.20399*, 2024.

Zunhai Su, Zhe Chen, Wang Shen, Hanyu Wei, Linge Li, Huangqi Yu, and Kehong Yuan. Rotatekv: Accurate and robust 2-bit kv cache quantization for llms via outlier-aware adaptive rotations. *arXiv preprint arXiv:2501.16383*, 2025.

Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Aditya Atluri, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023. URL `https://github.com/NVIDIA/cutlass`.

Joel A Tropp. Improved analysis of the subsampled randomized hadamard transform. *Advances in Adaptive Data Analysis*, 3(01n02):115–126, 2011.

Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. Quip#: Even better llm quantization with hadamard incoherence and lattice codebooks. *Proceedings of machine learning research*, 235:48630, 2024.

Albert Tseng, Tao Yu, and Youngsuk Park. Training llms with mxfp4. *arXiv preprint arXiv:2502.20586*, 2025.

Bingxin Xu, Zhen Dong, Oussama Elachqar, and Yuzhang Shang. Butterflyquant: Ultra-low-bit llm quantization through learnable orthogonal butterfly transforms. *arXiv preprint arXiv:2509.09679*, 2025.

Felix Xinnan X Yu, Ananda Theertha Suresh, Krzysztof M Choromanski, Daniel N Holtmann-Rice, and Sanjiv Kumar. Orthogonal random features. *Advances in neural information processing systems*, 29, 2016.

Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. Turboquant: Online vector quantization with near-optimal distortion rate. In *International Conference on Learning Representations*, volume 2026, pp. 56418–56439, 2026.

Keren Zhou, Mario Lezcano-Casado, Adam P Goucher, Akhmed Rakhmati, Jeff Niu, Justin Lebar, Pawel Szczerbuk, Peter Bell, Phil Tillet, Thomas Raoux, et al. Linear layouts: robust code generation of efficient tensor computation using f_2. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 132–146, 2026.

AI USE STATEMENT

We used large language models to aid and polish the writing and to assist with research execution, specifically running jobs on our compute cluster. The research ideas and experiment design are our own. We reviewed all text and code, validated every reported result against saved measurements, and take full responsibility for the content of this paper.

REPRODUCIBILITY STATEMENT

We will release our code and benchmarks upon publication.

ETHICS STATEMENT

This work studies GPU kernels for Hadamard transforms used in quantized inference and randomized projection. We do not identify direct human-subject, privacy, or safety risks beyond the energy and hardware cost of machine-learning research.

A MMA LAYOUT DERIVATION AND THE H_{256} SCHEDULE

A.1 FROM PER-LANE REGISTERS TO THE MMA LAYOUT UPDATE

Section 3.3 lists the per-lane ownership of the data operand B and output D for `mma.sync.aligned.m16n8k16.row.col`. We read both rows from the register-fragment figures of the PTX ISA (NVIDIA Corporation, 2026c). The B fragment packs $B[k, n]$ with k_1, k_2 on lane bits ℓ_0, ℓ_1 and k_0, k_3 on the two within-lane value bits. The output fragment packs $D[m, n]$ with n_1, n_2 on ℓ_0, ℓ_1 and n_0, m_3 on the value bits. Both permutations map a physical bit position to a logical-axis position. Their inverse has the opposite interpretation: $P_B^{-1}[a]$ tells us which physical bit held logical axis a in the input. We can therefore express each output position in input physical coordinates as $R_{\text{mma}}[p] = P_B^{-1}[P_D[p]]$. For example, output physical position zero holds n_1, logical-axis position five. That axis occupied input physical position three, so $R_{\text{mma}}[0] = P_B^{-1}[5] = 3$. Repeating this lookup for all seven positions gives $R_{\text{mma}} = (3, 4, 5, 0, 1, 2, 6)$. The implementation encodes the two layouts once and derives the update from them:

```
PermB = {1,2,4,5,6,0,3};  // P_B, MMA data input
PermC = {5,6,0,1,2,4,3};  // P_D, MMA output
R_mma = inverse(PermB) compose PermC;
```

R_{mma} is therefore the instruction's fixed output register layout, expressed in input coordinates. The generic fragment adapter presents any current layout P as P_B, issues the MMA, and returns the result typed as $P \circ R_{\text{mma}}$, so no data moves between instructions. At the type level, a Fragment and `rotate4` have the shortened signatures

```
template <DType T, int D, AxisState S[D], Permutation P>
struct Fragment {
  half_bits value[1 << (D - 5)];  // registers owned by one lane
};

rotate4(Fragment<T,D,S,P> x)
  -> Fragment<T,D,mark_done(S,P[0],P[1],P[5],P[6]),
              compose(P, R_mma)>;
```

A.2 WORKED H_{256} SCHEDULE

For one length-256 vector, the warp's five lane bits and each thread's three register-index bits give eight physical axes. A coalesced load places eight contiguous values in each lane and produces

$$P_0 = (3, 4, 5, 6, 7, 0, 1, 2). \tag{5}$$

Thus lane bits $(\ell_0, \ldots, \ell_4)$ represent $(b_3, \ldots, b_7)$ and local bits (r_0, r_1, r_2) represent (b_0, b_1, b_2). The first `rotate4` call rotates

$$\{P_0[0], P_0[1], P_0[5], P_0[6]\} = \{3, 4, 0, 1\}. \tag{6}$$

Physical position 7 (register bit r_2) lies outside the seven MMA axes and acts as a batch axis, so this call lowers to two MMA instructions. Extending R_{mma} by identity on that axis gives

$$P_1 = P_0 \circ R_{\mathrm{mma}} = (6, 7, 0, 3, 4, 5, 1, 2). \tag{7}$$

The kernel keeps this hardware-produced layout and applies a local reindex, a register swap of physical positions 6 and 7 within each thread:

$$P_2 = (6, 7, 0, 3, 4, 5, 2, 1), \tag{8}$$

so the next call rotates $\{P_2[0], P_2[1], P_2[5], P_2[6]\} = \{6, 7, 5, 2\}$, exactly the four axes that remain. The type-level trace is

Step	Layout	Completed logical axes
Coalesced load	$P_0 = (3, 4, 5, 6, 7, 0, 1, 2)$	none
First `rotate4`	$P_1 = (6, 7, 0, 3, 4, 5, 1, 2)$	$\{0, 1, 3, 4\}$
Local reindex	$P_2 = (6, 7, 0, 3, 4, 5, 2, 1)$	$\{0, 1, 3, 4\}$
Second `rotate4`	$P_3 = (3, 4, 5, 6, 7, 0, 2, 1)$	$\{0, \ldots, 7\}$
Local reindex	$P_0 = (3, 4, 5, 6, 7, 0, 1, 2)$	$\{0, \ldots, 7\}$

The second call also lowers to two MMA instructions, and a final local reindex restores P_0, so each lane writes eight consecutive outputs. Overall, the warp executes four MMA instructions and no shared-memory transpose or warp shuffle. The order $\{3, 4, 0, 1\}$ then $\{6, 7, 5, 2\}$ computes H_{256} by Lemma 1.

B DATAFLOW OF THE POWER-OF-TWO KERNEL

Algorithm 1 summarizes the power-of-two kernel of Section 3. The compile-time schedule fixes the two axis sets, the register layout used by each phase, and whether an axis is handled by a local butterfly or by `rotate4`.

Algorithm 1 HyperHadamard power-of-two transform

Require: Input row $x \in \mathbb{R}^{2^d}$
Ensure: $y = H_{2^d} x$
Fix axis sets (A_1, A_2) and their fragment layouts at compile time
if $A_2 = \emptyset$ **then** ▷ register-resident transform
    Load x into register fragments
    Rotate every axis in A_1 with local butterflies or `rotate4`
    Write the fragments contiguously to y
else ▷ large transform
    Begin a depth-seven asynchronous copy pipeline from x to shared memory M
    for each ready contiguous register fragment $\mathcal{F}$ of M **do**
        Prefetch a later chunk while computing the current chunk
        Rotate every axis in A_1 and write $\mathcal{F}$ back to M
    end for
    Synchronize the warp
    for each strided register fragment $\mathcal{F}$ of M **do**
        Rotate every axis in A_2 and write $\mathcal{F}$ contiguously to y
    end for
end if
return y

C H100 MEASUREMENT PROVENANCE

The FP16/BF16 operator sweep in Figure 2 was collected on an H100 80GB HBM3 GPU. We measured the empirical bandwidth reference separately on the same GPU model. Five fresh processes

each used three warmup calls and 20 randomized timed calls at 2^{30} elements. The elementwise read–modify–write kernel sustained 3.030 TB/s in FP16 (95% CI 3.028–3.031) and 3.028 TB/s in BF16 (3.026–3.029); device copy sustained 3.037 TB/s for both dtypes. The raw measurements and five-process summary accompany the artifact.

D OPERATOR-LEVEL RESULTS ON A100

We repeat the FP16/BF16 comparison with HadaCore on an NVIDIA A100-SXM4-80GB (HBM2e), rebuilding both kernels natively for SM80. We follow the protocol of Section 4.1. Each point processes 2^{30} elements, and we report five fresh processes with randomized paired method order and the median of 20 timed calls after warmup. Both kernels pass the correctness check at every length, with maximum absolute errors of 0.00391 in FP16 and 0.03125 in BF16.

Table 3: Speedup of HyperHadamard over HadaCore on A100 (95% confidence intervals across five processes).

N	FP16	BF16
256	0.990× (0.989–0.991)	0.992× (0.991–0.993)
512	0.992× (0.991–0.993)	0.992× (0.991–0.992)
1024	1.010× (1.009–1.011)	1.013× (1.012–1.013)
2048	0.999× (0.999–1.000)	1.000× (0.999–1.001)
4096	1.025× (1.024–1.026)	1.006× (1.005–1.007)
8192	1.029× (1.027–1.030)	1.023× (1.022–1.024)
16384	1.019× (1.018–1.020)	1.034× (1.033–1.035)
32768	**1.335**× (1.335–1.336)	**1.373**× (1.372–1.373)

Table 3 reports the speedups, which follow the H100 pattern of Figure 2. HyperHadamard sustains at least 97.8% of the empirical read–modify–write roofline at every length, and the two kernels are within 3.5% of each other up to $N = 16384$. At $N = 32768$, HadaCore falls to 74–75% of the roofline, and HyperHadamard is 1.335× faster in FP16 and 1.373× faster in BF16. A matched Nsight Compute profile at this length shows the same mechanism as on H100: HadaCore executes 3.59× (FP16) and 4.01× (BF16) as many dynamic warp instructions per element, and HyperHadamard sustains 1.45–1.55× higher physical DRAM throughput.

E FP32 PROFILE

The FP32 speedup has a different immediate mechanism from the tensor-core path. At $N = 32768$, Dao-AILab uses 254 registers per thread and sustains only 8 resident warps per SM (12.4% achieved occupancy). HyperHadamard uses 80 registers per thread and sustains 15.3 resident warps (23.9% occupancy). Consequently, cycles with an eligible warp rise from 26.9% to 42.9%, long-scoreboard stall samples fall from 43.3% to 30.0%, and physical DRAM throughput under Nsight Compute rises from 1.76 to 2.45 TB/s. Both kernels have effectively zero tensor-pipe activity, and HyperHadamard executes 15% more dynamic warp instructions. It issues 15% fewer shared-memory wavefronts, but its barrier and MIO-throttle stall fractions are higher, ruling those out as the dominant cause. The gain therefore comes from controlling register growth and hiding memory latency.

F VLLM INTEGRATION DETAILS

Catalog basis correction. Each rotation in the Llama-3.1-8B checkpoint has the form $C_q \otimes H_{2^k}$, applied after the checkpoint's stored random signs, where C_q is a catalog matrix of order $q = 224$ or 256. HyperHadamard first applies a transform of the same order whose base factor B_q it computes natively: the $H_{28} \otimes H_{512}$ kernel of Section 3.5 for 14336, so that $B_{224} = H_{28} \otimes H_8$, and the power-of-two FWHT for 4096 and 1024. A fixed $q \times q$ BF16 matrix $C_q B_q^\top / q$, applied along the base axis, then converts the result to the checkpoint's matrix. Because $B_q^\top B_q = qI$, the composition equals $C_q \otimes H_{2^k}$ exactly in exact arithmetic.

HadaCore replacement. Sweeping `max_num_batched_tokens` over 2,048, 4,096, 8,192, and 16,384 gives end-to-end speedups of 1.0521, 1.0276, 1.0220, and 1.0234×, each over six order-balanced process pairs. At the 2,048-token budget, throughput rises from 163,662 to 172,184 tokens/s, and H_{8192} accounts for 73.3% of the transform-time saving.

G Rotation size across weight and activation precision

Table 4 reports all twelve cells of the Mistral-7B ablation. Every cell uses the same GPTQ calibration data, test tokens, and quantizer; activations use dynamic per-token signed integer quantization, and BF16 means activations are not quantized.

Table 4: Mistral-7B WikiText-2 perplexity for H_{128} block rotations and full-dimension rotations. The last row of each block gives the relative change from H_{128} to full.

Weights	Rotation	BF16 act.	INT8 act.	INT4 act.
INT3	H_{128} blocks	6.1511	6.1424	6.9856
	Full dimension	6.1645	6.1673	6.5405
	Change	$+0.22\%$	$+0.41\%$	$\mathbf{-6.37}\%$
INT2	H_{128} blocks	17.981	18.112	57.150
	Full dimension	18.031	18.206	34.497
	Change	$+0.27\%$	$+0.52\%$	$\mathbf{-39.64}\%$

H Decomposing the full-width rotations

Starting from H_{1024} blocks at 6.61284 perplexity, enabling only H_{4096} lowers perplexity to 6.57811, enabling only H_{14336} raises it to 6.66787, and enabling both gives 6.54047. H_{4096} therefore helps on its own, whereas H_{14336} helps only in combination with H_{4096}.