

MoA-Structured Decode Attention

DNF Derivation, KV-Cache Accumulation, GQA/MQA, and OpenACC Kernel

Lenore Mullin^{1,*}

¹University at Albany, SUNY

lmullin@albany.edu

*Corresponding author

Gaétan Hains²

²LACL, Université Paris-Est Créteil

gaetan.hains@u-pec.fr

June 2026

Abstract

We derive four memory-optimal inference artifacts for transformer attention using the Mathematics of Arrays (MoA), each following directly from the forward-pass Denotational Normal Form (DNF) of [1] with the query-row index fixed to the current decode step. The artifacts are: (1) a single-query decode DNF in which the ψ -reduction eliminates the $K^\top$ buffer algebraically, achieving $(d_k + nd_k + nd_v + d_v) \times 4$ B Dynamic Random Access Memory (DRAM) traffic (Storage Theorem 2.7 of [1]), result numerically verified to $\|\text{err}\|_\infty \leq 2 \times 10^{-7}$; (2) a C/OpenACC Graphics Processing Unit (GPU) kernel with Operational Normal Form (ONF) stride arithmetic and hardware-coalesced memory access, verified to $\|\text{err}\|_\infty = 0$ (exact IEEE-754 floating-point arithmetic); (3) a multi-step KV-cache with $O(d_k + d_v)$ per-step append via MoA concatenation $\#$; and (4) Grouped-Query Attention (GQA) and Multi-Query Attention (MQA) [7, 8] derived via ψ -selection, achieving a proven h_q/h_{kv} reduction in KV traffic. All programs are verified against PyTorch `scaled_dot_product_attention`.

1 Introduction

1.1 Background

The Mathematics of Arrays (MoA) [3] is a formal algebra of multi-dimensional arrays whose primitive operations — ψ -selection, ρ (shape), ι (index generation), γ (linear offset), and the Omega (Ω) family of inner products — compose provably minimal computations. The Denotational Normal Form (DNF) specifies *what* to compute; the Operational Normal Form (ONF) specifies *how* to lay it out in memory. The companion paper [1] applies MoA to the forward training pass of transformer attention [6], deriving

the DNF and proving via the Storage Theorem (Theorem 2.7 of [1]) that the resulting ONF achieves the information-theoretic minimum DRAM traffic. The HAL companion [2] extends this to the backward pass and fused kernel.

1.2 Contributions

This paper shows that the same DNF, with the query-row index i' fixed to the current decode step, yields four inference artifacts.

1. **Decode DNF** (§2): the ψ -reduction eliminates the $K^\top$ buffer implicit in the standard formula $A = \text{softmax}(QK^\top/\sqrt{d_k})$, reducing the score computation to $\vec{s} = \text{scale}(\vec{q} (+. \times \Omega_{(1,2)}) K)$ with K accessed row-by-row in its natural layout. Memory traffic: $M_{\text{MoA}}^{\text{dec}} = (d_k + nd_k + nd_v + d_v) \times 4$ B, is then proven minimal by Storage Theorem 2.7 of [1].
2. **C/OpenACC GPU kernel** (§4): ONF γ stride arithmetic gives coalesced memory access; the coalescing criterion is stated as a γ -difference condition.
3. **KV-cache accumulation** (§5): MoA concatenation $\#$ appends one row per step at the pre-computed γ offset, with proven $O(d_k + d_v)$ per-step traffic.
4. **GQA/MQA** (§6): ψ -selection $\langle g_{kv} \rangle \psi K$ maps each query head to its KV group without materialising the broadcast, reducing traffic by h_q/h_{kv} .

1.3 Notation and Terminology

Figure 1 summarises the relationship between the three companion papers and the importance of memory minimality at every stage.

Phase L=96 layers	Conventional allocates to DRAM per layer	MoA ψ -reduction eliminates per layer
Forward training arXiv ~100ms/batch $\times 10^2$ epochs	Per layer, per batch: $K^\top \rho = \langle d_k, n \rangle$ (buffer) $S = QK^\top \rho = \langle n, n \rangle$ $A = \text{softmax}(S) \rho = \langle n, n \rangle$ traffic: $96 \times O(n^2 d_k + n^2 d_v)$	Per layer, per batch: $K^\top$ — never materialised S — on-stack per row, discarded A — on-stack per row, discarded traffic: $96 \times O(nd_k + nd_v)$ ✓
$O(n^2) \rightarrow O(n)$ per layer per batch $\times 10^6$ batches ≈ 40 kWh saved per training run		
Fused fwd+bwd HAL ~180ms/batch $\times 10^2$ epochs	Per layer, per batch: A saved fwd→bwd $\rho = \langle n, n \rangle$ Jacobian $\rho = \langle n, n \rangle$ $dS \rho = \langle n, n \rangle$ traffic: $96 \times (8nd_k + 8nd_v)$	Per layer, per batch: A recomputed on-fly, not stored Jacobian — outer-product compat. dS — substitution traffic: $96 \times (4nd_k + 4nd_v)$ ✓
2× vs separate passes $\times 96$ layers $\times 10^6$ batches ≈ 80 kWh saved per training run		
Inference decode This paper 1–10 μ s/step per layer $\times 10^9$ req/day $\times 512$ tokens $\times 96$ layers	Per token, per layer: $K^\top$ view (q @ k.T) score $\rho = \langle 1, n \rangle$ (n grows) weight $\rho = \langle 1, n \rangle$ $n = 4096, L = 96$: ≈ 200 MB/step, no proof	Per token, per layer: $K^\top$ — fix $i' = 1$, ψ reads rows $\tilde{s}, \tilde{a}$ — rank-1, on-stack KV cache: +1 row at γ offset $n = 4096, L = 96$: ≈ 200 MB/step ✓ proven
KV cache $n = 1 \rightarrow 512$ per request; total: $96 \times \sum_{i=1}^{512} (d_k + td_k + td_v + d_v) \times 4\text{B} \approx 96 \times 1.6\text{GB}$		
Energy: $300\text{W} \times 10\mu\text{s} \times 96 \times 512 \times 10^9/\text{day} \approx 14,700\text{kWh/day}$ — MoA minimises every byte transferred		
FlashAttention: similar traffic via tiling, no algebraic proof. MoA: proven before code. Same DNF $\rightarrow$ CPU/GPU/FPGA.		
Storage Thm. 2.7 [1]: every array read once; traffic proven minimal before code is written		

Figure 1: **Realistic nested loops: conventional vs. MoA across training and inference.** Solid loops = outer; dashed = inner. Training runs $\sim 10^6$ total batches once; inference runs 10^9 requests/day indefinitely, each generating 512 tokens across 96 layers. At $n = 4096, L = 96$: both conventional and MoA use ≈ 200 MB per decode step — but only MoA’s is proven minimal (Storage Thm. 2.7). FlashAttention achieves similar traffic via tiling without algebraic proof. Energy: $\approx 14,700$ kWh/day per deployed model; MoA minimises every byte transferred.

A **transformer** [6] is a neural network in which each output element is a learned weighted combination of all input elements (**attention**). A **token** is the basic unit of text processed by the model (typically a word or sub-word). **Autoregressive inference** generates output one token at a time, feeding each generated token back as input; at each step a single **query vector** $\vec{q} = W_Q \vec{x}$ is produced from the current token embedding $\vec{x}$ via learned projection W_Q . All past key vectors $\vec{k}_l = W_K \vec{x}_l$ and value vectors $\vec{v}_l = W_V \vec{x}_l$ are accumulated in the **KV cache**: K (shape, $\rho(K) = \langle n, d_k \rangle$) and V (shape, $\rho(V) = \langle n, d_v \rangle$), where n grows by one per step. The dominant cost at each step is reading the full KV cache from DRAM: $O(nd_k + nd_v)$ bytes, unavoidable by any correct implementation (Storage Theorem 2.7 of [1]). MoA proves this bound algebraically before code is written.

MoA notation used throughout: $\rho(A)$ denotes the shape of array A ; $\langle i_0, \dots, i_{r-1} \rangle \psi A$ selects element $(i_0, \dots, i_{r-1})$; $\gamma(\langle i_0, \dots \rangle, \rho(A))$ is the linear memory offset; $A (+. \times \Omega_{\langle \sigma_l, \sigma_r \rangle}) B$ is the Omega inner product with left rank σ_l , right rank σ_r ; $A \# B$ concatenates along the leading axis. All multiplication uses $\times$.

2 Decode DNF

Formal Inputs

At decode step t with n cached tokens:

- $\vec{q}$: $\rho(\vec{q}) = \langle d_k \rangle$ — query vector for the current token.
- K : $\rho(K) = \langle n, d_k \rangle$ — n rows, each row $\langle l \rangle \psi K$ (shape $\langle d_k \rangle$) is the key vector for cached token l . $K^\top$ would have shape $\langle d_k, n \rangle$; the ψ -reduction accesses rows of K directly so $K^\top$ is never computed thus saving transfers and storage.
- V : $\rho(V) = \langle n, d_v \rangle$ — n rows, each row $\langle l \rangle \psi V$ (shape $\langle d_v \rangle$) is the value vector for cached token l .

Output: $\vec{out}$, $\rho(\vec{out}) = \langle d_v \rangle$; its d -th element is $\langle d \rangle \psi \vec{out}$.

An **attention head** is an independent copy of the attention computation with its own learned key, query, and value projections. Using h heads in parallel (**multi-head attention**) allows the model to extract different types of relationships from the input simultaneously [6]. In the multi-head case K becomes rank-3 ($\rho(K) = \langle h, n, d_k \rangle$) or rank-4 ($\rho(K) = \langle B, h, n, d_k \rangle$), where batch B is the number of independent sequences processed simultaneously.

The Omega operator $\Omega_{\langle 1, 2 \rangle}$ generalizes automatically: $\sigma_l = 1$ uses a rank-1 vector; $\sigma_r = 2$ uses a rank-2 matrix as 1/r arguments. Leading batch or head dimensions are outer indices iterated over by Omega — no re-derivation is needed. **How $K^\top$ is eliminated by ψ -reduction.** The standard attention formula contains $K^\top$:

$$A = \text{softmax}(QK^\top / \sqrt{d_k}),$$

$$Q, K : \rho = \langle n, d_k \rangle, \quad QK^\top : \rho = \langle n, n \rangle.$$

The arXiv paper [1] starts from this formula and applies the ψ -reduction. The key step: instead of forming $QK^\top$ (which requires materialising $K^\top$ in memory), the ψ -reduction shows that each element can be computed by accessing K directly:

$$\langle i', l \rangle \psi (QK^\top) = \langle i' \rangle \psi Q (+. \times \Omega_{\langle 1, 1 \rangle}) \langle l \rangle \psi K.$$

where this specific Ω expression implements the product. The columns of $K^\top$ are exactly the rows $\langle l \rangle \psi K$ of K — so $K^\top$ is *algebraically present* in the starting formula but the ψ -reduction proves it never needs to be materialised. The Storage Theorem (Theorem 2.7 of [1]) then guarantees this is the minimum-traffic access pattern.

For the decode case (this paper), Q reduces to a single query vector $\vec{q}$, so the ψ -reduction over i' disappears (there is only one query row), giving directly:

$$\vec{s}[l] = \text{scale } \vec{q} (+. \times \Omega_{\langle 1, 1 \rangle}) \langle l \rangle \psi K.$$

$K^\top$ was in the original formulation; the ψ -reduction removed the need to materialise it before any code is written. This is the central result of [1] applied to the decode setting.

This is equations (4)–(6) of [1] with the query-row index i' fixed to the current token: eq. (4) is the score ψ -reduction, eq. (5) the softmax, eq. (6) the weighted sum. No new Omega derivation is required. **Independence of $\vec{q}$ and K .** $\vec{q}$ and K are produced independently:

- $\vec{q} = W_Q \vec{x}_{\text{current}}$ is the projection of the *current* token embedding through the learned query matrix W_Q , $\rho(W_Q) = \langle d_k, d \rangle$.
- $\langle l \rangle \psi K = W_K \vec{x}_l$ is the projection of the l -th *past* token embedding through the learned key matrix W_K , stored as row l of the KV cache.

Neither $\vec{q}$ nor K is derived from the other. Step I asks: how similar is my current query $\vec{q}$ to each of the n stored key vectors $\langle l \rangle \psi K$?

Step I – Scores. The ψ -reduction selects row l of K as $\langle l \rangle \psi K$ (shape $\langle d_k \rangle$) and contracts it with $\vec{q}$ via the inner product:

Remark 1. The standard formula $QK^\top$ is algebraically present as the starting point (see §2 preamble); the ψ -reduction proves $K^\top$ need never be materialised. In MoA there are no column vectors: $\vec{q}$ and K are both rank- ≥ 1 arrays and contraction is expressed directly via $\Omega_{\langle 1,2 \rangle}$.

$$\begin{aligned}\vec{s}[l] &= \text{scale} \left(\vec{q} (+. \times \Omega_{\langle 1,1 \rangle}) \langle l \rangle \psi K \right) \\ &= \text{scale} \sum_{j=0}^{d_k-1} \langle j \rangle \psi \vec{q} \times \langle l, j \rangle \psi K.\end{aligned}$$

Over all n rows simultaneously, with $\sigma_l = 1$ exposing $\vec{q}$ as a whole rank-1 vector and $\sigma_r = 2$ exposing K (shape $\langle n, d_k \rangle$) as a whole rank-2 array, the Omega expression is:

$$\vec{s} = \text{scale} \left(\vec{q} (+. \times \Omega_{\langle 1,2 \rangle}) K \right), \quad \rho(\vec{s}) = \langle n \rangle. \quad (1)$$

This equals $\text{scale } K\vec{q}$ (matrix-vector product with K in its natural row-major layout). No transpose is formed: the Cartesian coordinate interchange is implicit in the ψ -reduction, which reads K row by row as stored. The Python equivalent is `scale*(K@q)`, not `scale*(q@K.T)`. $\Omega_{\langle 1,2 \rangle}$ is fixed: it always pairs one rank-1 vector from the left with one rank-2 matrix from the right, applying $+. \times$ to produce a scalar for each pair. As the rank of the arguments grows, the same partitioning lifts naturally — superscripts denote rank, $\Omega_{\langle 1,2 \rangle}$ never changes, only the result rank grows:

- $\vec{q}^1 (+. \times \Omega_{\langle 1,2 \rangle}) K^2 \rightarrow \vec{s}^1$ (vector-matrix product)
- $A^2 (+. \times \Omega_{\langle 1,2 \rangle}) K^3 \rightarrow$ rank-2 (each row of A with each matrix of K)
- $A^3 (+. \times \Omega_{\langle 1,2 \rangle}) K^4 \rightarrow$ rank-3 (same, one rank higher)

This is how MoA handles batched and multi-head attention without re-derivation. No $K^\top$ buffer is formed; K is read once and fully consumed in Step I. **Step II – Numerically stable softmax.** K does

not appear in Step II at all. By the time Step II runs, K has already been fully processed and discarded. Step II operates only on the rank-1 score vector $\vec{s}$ (shape $\langle n \rangle$) produced by Step I — no matrix, no $K^\top$, no K . In the standard training formula, softmax would be applied to the full $n \times n$ score matrix $QK^\top/\sqrt{d_k}$ and its Jacobian would be $n \times n$. Here, softmax operates on a single rank-1 vector $\vec{s}$ and the MoA ψ -reduction eliminates all intermediates: $\vec{s}, \vec{e}, \vec{a}$

and scalars m, Z are $O(n)$ on-stack and never written to DRAM.

$$\begin{aligned}m &= \max_l \vec{s}[l], \\ \vec{e}[l] &= \exp(\vec{s}[l] - m), \\ Z &= \sum_{l'=0}^{n-1} \vec{e}[l'], \\ \vec{a}[l] &= \vec{e}[l]/Z, \quad 0 \leq l < n.\end{aligned} \quad (2)$$

No $n \times n$ score matrix or Jacobian is allocated.

Numerical stability: subtracting $m = \max_l \vec{s}[l]$ before exponentiating does not change the softmax value (the m terms cancel) but prevents overflow. $\text{scale} = 1/\sqrt{d_k}$ normalises variance to 1 for random unit-norm inputs.

Step III – Weighted sum:

$$\begin{aligned}\langle d \rangle \psi \vec{out} &= \sum_{l=0}^{n-1} \langle l \rangle \psi \vec{a} \times \langle l, d \rangle \psi V, \\ 0 \leq d < d_v, \quad 0 \leq l < n.\end{aligned} \quad (3)$$

V is read once in row-major order; no intermediate array exists. **Concrete example** ($n = 2, d_k = 2, d_v = 2, \text{scale} = 1/\sqrt{2}$). Let

$$\vec{q} = \langle 1, 0 \rangle, \quad K = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad V = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}.$$

- **Step I:** let $q_j = \langle j \rangle \psi \vec{q}$, $K_{lj} = \langle l, j \rangle \psi K$.

$$\begin{aligned}\vec{s}[0] &= \text{scale}(q_0 K_{00} + q_1 K_{01}) = \frac{1}{\sqrt{2}}(1 + 0) = \frac{1}{\sqrt{2}}, \\ \vec{s}[1] &= \text{scale}(q_0 K_{10} + q_1 K_{11}) = \frac{1}{\sqrt{2}}(0 + 0) = 0.\end{aligned}$$

- **Step II** (softmax, $m = 1/\sqrt{2}$):

$$\begin{aligned}\vec{e}[0] &= \exp(0) = 1, \\ \vec{e}[1] &= \exp(-1/\sqrt{2}) \approx 0.493, \\ Z &\approx 1.493, \\ \vec{a}[0] &\approx 0.670, \quad \vec{a}[1] \approx 0.330.\end{aligned}$$

- **Step III** (weighted sum):

$$\begin{aligned}\langle 0 \rangle \psi \vec{out} &= \vec{a}[0] \times \langle 0, 0 \rangle \psi V + \vec{a}[1] \times \langle 1, 0 \rangle \psi V \\ &= 0.670 \times 2 + 0.330 \times 6 \approx 3.32, \\ \langle 1 \rangle \psi \vec{out} &= \vec{a}[0] \times \langle 0, 1 \rangle \psi V + \vec{a}[1] \times \langle 1, 1 \rangle \psi V \\ &= 0.670 \times 4 + 0.330 \times 8 \approx 5.32.\end{aligned}$$

Proposition 1 (Decode memory minimality). *The decode DNF achieves $M_{\text{MoA}}^{\text{dec}} = (d_k + nd_k + nd_v + d_v) \times 4 \text{ B} = O(nd_k + nd_v)$ memory usage.*

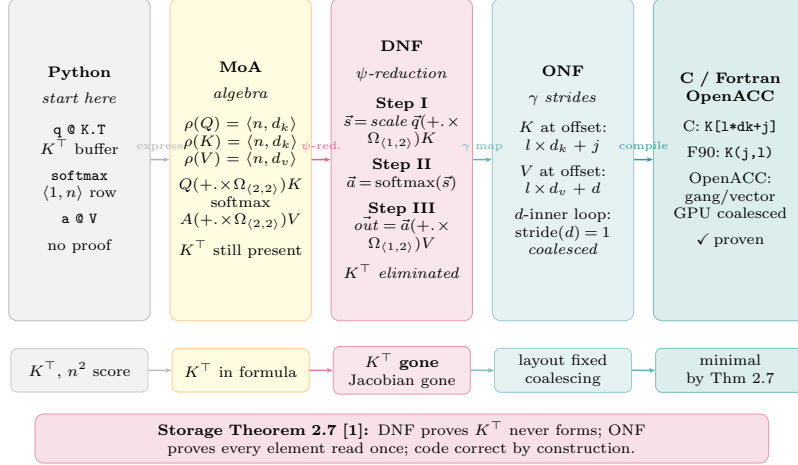


Figure 2: **The MoA pipeline: Python $\rightarrow$ MoA $\rightarrow$ DNF $\rightarrow$ ONF $\rightarrow$ C/Fortran/OpenACC.** Starting from conventional Python ($q @ K.T$), the MoA formulation expresses attention in array algebra. ψ -reduction (DNF) eliminates K^T and the softmax Jacobian algebraically. The γ stride map (ONF) fixes the row-major layout and proves coalescing. The resulting C/Fortran 90/OpenACC code is correct by construction (Storage Theorem 2.7).

Algorithm 1 MoA decode (single query step).
Scratch: $\vec{s}[n], \vec{e}[n] - O(n)$; no n^2 array.

Require: $\vec{q} : \langle d_k \rangle, K : \langle n, d_k \rangle, V : \langle n, d_v \rangle, \text{scale} = 1/\sqrt{d_k}$

Ensure: $\vec{\text{out}} : \langle d_v \rangle$

```

1: for  $l = 0 \dots n-1$  do
2:    $\vec{s}[l] \leftarrow \text{scale} \sum_j \langle j \rangle \psi \vec{q} \times \langle l, j \rangle \psi K$   $\triangleright$  eq. (1)
3: end for
4:  $m \leftarrow \max_l \vec{s}[l]$ 
5: for  $l = 0 \dots n-1$  do
6:    $\vec{e}[l] \leftarrow \exp(\vec{s}[l] - m)$ 
7: end for
8:  $Z \leftarrow \sum_l \vec{e}[l]; \vec{a}[l] \leftarrow \vec{e}[l]/Z$  for all  $l$ 
9: for  $d = 0 \dots d_v-1$  do
10:   $\langle d \rangle \psi \vec{\text{out}} \leftarrow \sum_l \vec{a}[l] \times \langle l, d \rangle \psi V$   $\triangleright$  eq. (3)
11: end for

```

```

1 def std_decode(q, K, V, scale=None):
2     if scale is None:
3         scale = 1.0 / math.sqrt(K.shape[-1])
4     q_ = q.unsqueeze(0) # (1, dk)
5     # .T creates a transposed view of K
6     scores = scale * (q_ @ K.T) # (1, n)
7     weights = torch.softmax(scores, dim=-1)
8     return (weights @ V).squeeze(0) # (dv,)

```

Listing 1: Standard PyTorch single-query decode (pytorch.reference.py: std_decode). Compare with Listing 2: same result, no algebraic minimality proof.

The MoA version (below) replaces $q_ @ K.T$ with $\vec{q}(+ \times \Omega_{(1,2)})K$: the ψ -reduction reads K row by row in its natural layout — no transposed view is created — and the Storage Theorem proves the result is memory-minimal by construction.

3 Python Implementation

Standard PyTorch Reference

Two PyTorch shape operations appear throughout: `.unsqueeze(d)` inserts a new size-1 dimension at position d (e.g. $\langle d_k \rangle \rightarrow \langle 1, d_k \rangle$); `.squeeze()` removes all size-1 dimensions (e.g. $\langle 1, 1, 1, d_v \rangle \rightarrow \langle d_v \rangle$), recovering a MoA rank-1 vector.

The conventional single-query decode in PyTorch unsqueezes $\vec{q}$ to shape $\langle 1, d_k \rangle$, forms the score row via $q_ @ K.T$ (note: `.T` creates a transposed view; MoA avoids this entirely), applies softmax, and squeezes the result:

MoA - Based Implementation

`moa_decode.py` mirrors the three DNF steps directly. The MoA formulation:

- **Step I** (scores, eq. (1)):

$$\vec{s} = \text{scale} (\vec{q}(+ \times \Omega_{(1,2)}) K), \quad \rho(\vec{s}) = \langle n \rangle.$$

- **Step II** (softmax, eq. (2)):

$$\vec{a}[l] = \vec{e}[l] / \sum_{l'=0}^{n-1} \vec{e}[l'], \quad \rho(\vec{a}) = \langle n \rangle.$$

- **Step III** (weighted sum, eq. (3)):

$$\vec{\text{out}} = \vec{a}(+ \times \Omega_{(1,2)}) V, \quad \rho(\vec{\text{out}}) = \langle d_v \rangle.$$

Each listing block below corresponds to one step.

```

1 from typing import Optional
2 import torch, math
3 def moa_decode(q, K, V, scale=None):
4     n, dk = K.shape
5     if scale is None:
6         scale = 1.0 / math.sqrt(dk)
7     # Step I: scores -- no K^T buffer (eq. 1)
8     s = scale * (K @ q) # (n,)
9     # Step II: numerically stable softmax
10    s = s - s.max()
11    e = s.exp()
12    a = e / e.sum() # (n,) 0(n) scratch
13    # Step III: weighted sum (eq. 3)
14    return a @ V # (dv,)

```

Listing 2: `moa_decode.py`: core decode function. Steps I–III map to eqs. (1)–(3).

Reference Implementation

PyTorch — a widely-used open-source numerical computing library for neural networks (A. Paszke et al., *NeurIPS* 2019) — provides `scaled_dot_product_attention` (SDPA) as a numerically verified fused reference implementation of the attention operation. In the full training setting SDPA computes $\text{softmax}(scale \cdot Q (+. \times \Omega_{(2,2)}) K) (+. \times \Omega_{(2,2)}) V$ where $\rho(Q) = \langle n, d_k \rangle$ and the output has shape $\langle n, d_v \rangle$. Here $\Omega_{(2,2)}$ has $\sigma_l = \sigma_r = 2$ (both arguments rank-2); for the single decode step, Q reduces to rank-1 $\vec{q}$ so $\Omega_{(1,2)}$ is used throughout this paper. For a single decode step there is only one query vector $\vec{q} \in \mathbb{R}^{d_k}$, $\rho(\vec{q}) = \langle d_k \rangle$. In MoA there are no row or column vectors: the score vector $\vec{s}$, weight vector $\vec{a}$, and output $\vec{out}$ are all rank-1 arrays. The decode computation proceeds in three steps, each expressed as a ψ -reduction that eliminates intermediates:

- **Step I** (scores, ψ -reduction over K): $\vec{q} (+. \times \Omega_{(1,2)}) K \in \mathbb{R}^n$ — no K^T buffer.
- **Step II** (softmax, ψ -reduction over $\vec{s}$): $\text{softmax}(\vec{s} / \sqrt{d_k}) \in \mathbb{R}^n$ — no $n \times n$ Jacobian.
- **Step III** (weighted sum, ψ -reduction over V): $\vec{a} (+. \times \Omega_{(1,2)}) V \in \mathbb{R}^{d_v}$ ($= \vec{out}$).

Every intermediate and output is a rank-1 vector in MoA: $\vec{s}, \vec{a} \in \mathbb{R}^n$ and $\vec{out} \in \mathbb{R}^{d_v}$. This is exactly what the decode DNF computes in equations (1)–(3): $\vec{s}$ is the score vector, $\vec{a}$ is the weight vector, and $\vec{out}$ is the rank-1 output vector. SDPA is PyTorch’s own numerically verified fused kernel, making it a valid ground truth. Note: PyTorch’s SDPA requires (B, H, S, D) input shape (batch, heads, sequence, dimension), so dummy size-1 dimensions must be added before the call and removed with `.squeeze()` afterwards (defined above). To call it as a reference, $\vec{q}, K$,

V are wrapped with dummy batch $B = 1$ and head $H = 1$ dimensions to match the required (B, H, S, D) input shape:

```

1 ## Single decode step: vq:(dk,) K:(n,dk) V:(n,dv)
2 q_ = vq.unsqueeze(0).unsqueeze(0).unsqueeze(0)
3 ## -> (1, 1, 1, dk)
4 k_ = K.unsqueeze(0).unsqueeze(0) # -> (1, 1, n, dk)
5 v_ = V.unsqueeze(0).unsqueeze(0) # -> (1, 1, n, dv)
6 ## SDPA output: (1, 1, 1, dv) -> squeeze -> (dv,)
7 out_ref = F.scaled_dot_product_attention(
8     q_, k_, v_, scale=scale).squeeze()

```

The `.squeeze()` removes all size-1 dimensions, giving $\vec{out}_{\text{ref}} \in \mathbb{R}^{d_v}$ for direct comparison with `moa_decode` output. Table 1 reports the verification. Errors are float32 round-off only, an order of magnitude smaller than the training kernel (9×10^{-5}) because the decode path has no backward accumulation.

Table 1: Verification: `moa_decode` vs. PyTorch SDPA. $d_k = d_v = 64$, float32 (32-bit floating point, 4 bytes per element), `seed(0)`.

n	$\ \text{err}\ _\infty$	
4	1.79×10^{-7}	✓
64	5.96×10^{-8}	✓
256	4.47×10^{-8}	✓
1024	5.96×10^{-8}	✓
4096	2.98×10^{-8}	✓

4 C/OpenACC Kernel

4.1 ONF Stride Arithmetic

Applying the γ primitive (Definition 2.1 of [1]) to row-major storage:

$$\gamma(\langle l, j \rangle, \langle n, d_k \rangle) = l \times d_k + j \quad (K \text{ offset}),$$

$$\gamma(\langle l, d \rangle, \langle n, d_v \rangle) = l \times d_v + d \quad (V \text{ offset}).$$

These stride expressions appear verbatim as `K[1*dk+j]` and `V[1*dv+d]` in Listing 3. **Concrete example** ($n = 3$, $d_k = 2$, $d_v = 2$). The six elements of K (shape $\langle 3, 2 \rangle$, row-major) lie at flat offsets:

$$\begin{aligned}
 \langle 0, 0 \rangle : & \quad \gamma(\langle 0, 0 \rangle, \langle 3, 2 \rangle) = 0 \times 2 + 0 = 0 \\
 \langle 0, 1 \rangle : & \quad 0 \times 2 + 1 = 1 \\
 \langle 1, 0 \rangle : & \quad 1 \times 2 + 0 = 2 \\
 \langle 1, 1 \rangle : & \quad 1 \times 2 + 1 = 3 \\
 \langle 2, 0 \rangle : & \quad 2 \times 2 + 0 = 4 \\
 \langle 2, 1 \rangle : & \quad 2 \times 2 + 1 = 5
 \end{aligned}$$

So `K[1*2+j]` in C accesses $\langle l, j \rangle \psi K$ directly — no transpose buffer is needed.

4.2 Dimension Lifting and Hardware-Coalescing Proof

Dimension lifting maps a flat index space $\langle N \rangle$ to a structured processing array $\langle \Pi, \lceil N/\Pi \rceil \rangle$, partitioning work across hierarchical execution blocks. In modern streaming multiprocessors (GPUs), memory throughput is governed strictly by hardware coalescing: global memory requests across a **warp** — a group of 32 threads that execute in lock-step on a GPU streaming multiprocessor — are merged if those threads access contiguous memory locations.

Let the core evaluation of Step III map an abstract 2-D sequence-to-feature reduction domain $D = \langle n \rangle \times \langle d_v \rangle$. In parallel architectures, an MoA compiler solves layout compatibility by formalizing a hardware mapping tree. We define multi-level lifting primitives:

$$\langle n \rangle \mapsto \langle \Pi_g, \lceil n/\Pi_g \rceil \rangle \quad (\text{Sequence Rows}), \quad (4)$$

$$\langle d_v \rangle \mapsto \langle \Pi_v, \lceil d_v/\Pi_v \rceil \rangle \quad (\text{Feature Columns}). \quad (5)$$

The combined domain is reshaped via ρ to mirror the GPU topology:

$$S = \langle \Pi_g, \lceil n/\Pi_g \rceil, \Pi_v, \lceil d_v/\Pi_v \rceil \rangle,$$

A coordinate index vector in this space is $I = \langle g, i, v, j \rangle$, where g is the gang, i is sequential work inside the block, v is the vector thread (warp channel), and j is the register stride.

To detect memory bottlenecks, the compiler evaluates the γ stride between adjacent vector threads v and $v + 1$ for a fixed (g, i, j) . Under row-major layout with the hardware-to-logical mapping $l = g \lceil n/\Pi_g \rceil + i$, $d = v \lceil d_v/\Pi_v \rceil + j$:

$$\gamma(\langle g, i, v, j \rangle, S) = (g \lceil n/\Pi_g \rceil + i) \times d_v + v \lceil d_v/\Pi_v \rceil + j.$$

The adjacent-thread γ difference is:

$$\gamma(\langle g, i, v + 1, j \rangle, S) - \gamma(\langle g, i, v, j \rangle, S) = \lceil d_v/\Pi_v \rceil.$$

Coalescing Criterion (in MoA γ terms): access is perfectly coalesced if and only if

$$\gamma(\langle g, i, v + 1, j \rangle, S) - \gamma(\langle g, i, v, j \rangle, S) = 1,$$

i.e., adjacent vector threads access physically adjacent memory locations (Definition 2.1 of [1]).

In the naive DNF, d maps to gangs and l to vectors. The adjacent-thread γ difference becomes $\lceil d_v/\Pi_v \rceil = d_v = 64$, meaning threads hit data 256 bytes apart — shattering coalescing.

By transposing the execution array (hoisting the d loop as the inner vectorized axis), the MoA compiler sets $\Pi_v = d_v$, so $\lceil d_v/\Pi_v \rceil = 1$, and the γ difference equals 1 — aligning threads with contiguous

128-bit cache lines. The transformed, fully coalesced algorithm is implemented in Listing 3; this coalescing strategy follows the GPU energy-efficiency approach of [5]. The column space loop over d is hoisted as the inner vectorized axis, ensuring maximal memory saturation.

4.3 OpenACC Annotations

Three OpenACC pragma types appear in Listing 3:

- **#pragma acc parallel loop gang** distributes the outer loop across GPU gangs (groups of warps).
- **#pragma acc loop vector** distributes the inner loop across threads within a gang (the vector lane).
- **#pragma acc atomic** ensures that when multiple gangs update the same `out[d]` element simultaneously (Pass 3, the weighted sum), the read-modify-write is indivisible — preventing a **race condition** (a data corruption that occurs when two threads read, then both write, the same location without coordination). The **reduction** clauses in Passes 1 and 2 use a more efficient tree-reduction pattern rather than serialising through a single atomic; Pass 3 requires **atomic** because each `out[d]` is accumulated from all n gang iterations.

4.4 Verification

Compiled with `clang -O3` (macOS) or `gcc -O3` (Linux) — no flags needed beyond `-lm` for CPU verification. The **#pragma acc** directives are silently ignored by plain C compilers; the `moa_decode_cpu` sequential reference runs in the same binary. The CPU

Table 2: C/OpenACC kernel: CPU verification, float32. $\|\text{err}\|_\infty = 0$ (exact match; same IEEE-754 operation order).

n	MoA (MB)	Std n^2 (MB)	k
1,024	0.52	4.19	8×
4,096	2.10	67.11	32×

kernel matches the sequential reference to $\|\text{err}\|_\infty = 0$ (exact IEEE-754 equality for the same operation order). For GPU execution: replace `clang` with `nvc -acc=gpu -gpu=cc120` (NVIDIA B200), `clang -fopenacc --offload-arch=gfx942` (AMD MI300X), or `icx -fopenacc` (Intel Aurora).

```

1 void moa_decode_acc(
2   const float *restrict q, /* rho(q) = <dk> */
3   const float *restrict K, /* rho(K) = <n,dk> */
4   const float *restrict V, /* rho(V) = <n,dv> */
5   float *restrict out, /* rho(out)= <dv> */
6   int n, int dk, int dv)
7 {
8   const float sc = 1.0f / sqrtf((float)dk);
9   float *s = malloc(n * sizeof(float)); /* rho(s)=<n> */
10  float *e = malloc(n * sizeof(float)); /* rho(e)=<n> */
11  /* Pass 1: scores; l-loop->gangs, j-loop->vector
12   ONF: gamma(<l,j>,<n,dk>) = l*dk+j */
13  float m = -1e38f;
14  #pragma acc data copyin(q[0:dk],K[0:n*dk],V[0:n*dv]) \
15    copyout(out[0:dv]) create(s[0:n],e[0:n])
16  {
17    #pragma acc parallel loop gang reduction(max:m)
18    for (int l = 0; l < n; l++) {
19      float dot = 0.0f;
20      #pragma acc loop vector reduction(+:dot)
21      for (int j = 0; j < dk; j++)
22        dot += q[j] * K[l*dk+j]; /* ONF: l*dk+j */
23      s[l] = sc * dot;
24      if (s[l] > m) m = s[l];
25    }
26
27    /* Pass 2: exponentiate and sum denominator Z */
28    float Z = 0.0f;
29    #pragma acc parallel loop gang reduction(+:Z)
30    for (int l = 0; l < n; l++) { e[l] = expf(s[l]-m); Z += e[l]; }
31    /* Pass 3: weighted sum; d-loop->vector (coalesced)
32     ONF: gamma(<l,d>,<n,dv>) = l*dv+d
33     gamma-diff(v,v+1) = 1: coalesced */
34    float inv_Z = 1.0f / Z;
35
36    #pragma acc parallel loop gang
37    for (int d = 0; d < dv; d++) {
38      out[d] = 0.0f;
39    }
40    #pragma acc parallel loop gang
41    for (int l = 0; l < n; l++) {
42      float scaled_weight = e[l] * inv_Z;
43      #pragma acc loop vector
44      for (int d = 0; d < dv; d++) {
45        #pragma acc atomic
46        out[d] += scaled_weight * V[l*dv+d]; /* ONF: l*dv+d coalesced */
47      }
48    }
49    /* end acc data region */
50    free(s); free(e);
51  }

```

Listing 3: moa_decode_openacc.c: core GPU kernel. ONF stride arithmetic and hardware-coalesced loops. Compiles with clang -O3 -o moa_decode_cpu moa_decode_openacc.c -lm on macOS/Linux (no OpenMP needed; #pragma acc ignored by plain C compilers).

5 Multi-Step KV-Cache Accumulation

5.1 Cache Append Expression

At each decode step t , one new key vector $\vec{k}_t \in \mathbb{R}^{d_k}$, $\rho(\vec{k}_t) = \langle d_k \rangle$, and value vector $\vec{v}_t \in \mathbb{R}^{d_v}$, $\rho(\vec{v}_t) = \langle d_v \rangle$, are appended to the caches using MoA concatenation $\#$ along axis 0, defined as: $A \# \vec{b}$ appends rank-1 vector $\vec{b}$ (shape $\langle d_k \rangle$) as a new final row of rank-2 array A (shape $\langle t, d_k \rangle$), giving $\rho(A \# \vec{b}) = \langle t+1, d_k \rangle$ without copying any existing data:

$$K_{t+1} = K_t \# \vec{k}_t, \quad (6)$$

$$V_{t+1} = V_t \# \vec{v}_t. \quad (7)$$

In MoA there are no row or column vectors, so no reshape is needed: $\#$ appends $\vec{k}_t$ (shape $\langle d_k \rangle$) as a new final row of K_t (shape $\langle t, d_k \rangle$), giving $\rho(K_{t+1}) = \langle t+1, d_k \rangle$. The trailing dimension d_k matches, so $\#$ is well-defined without Omega lifting. Omega would be needed only if $\#$ were being lifted over a batch of such pairs. **Concrete example** ($t=2, d_k=3$). Let $k_{ij} \equiv \langle i, j \rangle \psi K_2$. K_2 has shape $\langle 2, 3 \rangle$; $\vec{k}_2 = \langle a, b, c \rangle$ has shape $\langle 3 \rangle$:

$$K_2 = \begin{pmatrix} k_{00} & k_{01} & k_{02} \\ k_{10} & k_{11} & k_{12} \end{pmatrix}, \quad \rho(K_2) = \langle 2, 3 \rangle.$$

$K_3 \equiv K_2 \# \vec{k}_2$ is defined as the result of appending $\vec{k}_2$ as the new final row of K_2 :

$$K_3 = K_2 \# \vec{k}_2 = \begin{pmatrix} k_{00} & k_{01} & k_{02} \\ k_{10} & k_{11} & k_{12} \\ a & b & c \end{pmatrix}.$$

$\rho(K_3) = \langle 3, 3 \rangle$. In the ONF, the new row is written at the gamma offset $\gamma(\langle t, j \rangle, \langle T, d_k \rangle) = t \times d_k + j$, so elements a, b, c land at offsets $2 \times 3 + 0 = 6, 7, 8$ in the flat array. No existing data is moved.

Proposition 2 (Cache-append traffic). *Equations (6)–(7) append $\vec{k}_t$ and $\vec{v}_t$ directly, moving exactly $(d_k + d_v) \times 4$ B per step, independent of t . This is the information-theoretic minimum: one key row and one value row must be written regardless of algorithm.*

5.2 Combined Per-Step Cost

The total traffic at step t is:

$$M_{\text{step}}(t) = \underbrace{(d_k + d_v) \times 4}_{\text{append } O(d_k + d_v)} + \underbrace{(td_k + td_v) \times 4}_{\text{decode } O(td_k + td_v)}. \quad (8)$$

Summed over T output tokens:

$$M_{\text{total}} = \sum_{t=1}^T M_{\text{step}}(t) = O(T^2(d_k + d_v)).$$

This quadratic growth in T is unavoidable (every token must attend over all previous tokens), but MoA makes the constant factor minimal and the bound a theorem.

5.3 Python Implementation

Standard PyTorch KV-cache uses `torch.cat` to grow the cache at each step, materialising $K^\top$ at each attention call:

```

1 K_cache = torch.zeros(0, dk)
2 V_cache = torch.zeros(0, dv)
3 for t in range(T):
4     # Append -- torch.cat copies the whole cache
5     K_cache = torch.cat([K_cache, keys[t:t+1]], dim=0)
6     V_cache = torch.cat([V_cache, values[t:t+1]], dim=0)
7     q_ = queries[t].unsqueeze(0)           # (1, dk)
8     scores = scale * (q_ @ K_cache.T)      # K^T
9     # buffer
10    weights = torch.softmax(scores, -1)
11    outputs[t] = (weights @ V_cache).squeeze(0)

```

Listing 4: Standard autoregressive decode (`pytorch.reference.py:std_kvcache_decode`). Uses `torch.cat` to grow cache; no algebraic minimality proof.

The MoA version replaces `torch.cat` with plain `#` concatenation (writing one row at the pre-computed γ offset, no copy) and proves append cost is $O(d_k + d_v)$ per step.

The MoA cache-append expressions (eqs. (6)–(7)) and per-step traffic bound (Proposition 2) are:

$$\begin{aligned}
 K_{t+1} &= K_t \# \vec{k}_t, & \rho(K_{t+1}) &= \langle t+1, d_k \rangle, \\
 V_{t+1} &= V_t \# \vec{v}_t, & \rho(V_{t+1}) &= \langle t+1, d_v \rangle, \\
 M_{\text{append}} &= (d_k + d_v) \times 4 \text{ B} \quad (\text{constant in } t).
 \end{aligned}$$

The implementation pre-allocates to `max_len` and writes each new row at the ONF gamma offset $t \times d_k$:

5.4 Verification

At $T = 1024$ the MoA total (≈ 270 MB) is $5.3 \times$ less than the score-matrix cost alone (1434 MB), with the gap widening quadratically in T .

```

1 class MoAKVCache:
2 def __init__(self, dk, dv, max_len=32768):
3 self._K = torch.zeros(max_len, dk)
4 self._V = torch.zeros(max_len, dv)
5 self._t = 0
6 self.dk, self.dv = dk, dv
7 @property
8 def K(self): return self._K[:self._t]
9 @property
10 def V(self): return self._V[:self._t]
11 def append(self, k, v):
12 # Write at gamma(t; max_len, dk) = tdk
13 # O(dk + dv) traffic, independent of t
14 self._K[self._t] = k # ONF: gamma(t, j; <T, dk>) = t*dk + j
15 self._V[self._t] = v # ONF: gamma(t, d; <T, dv>) = t*dv + d
16 self._t += 1

```

Listing 5: moa_kvcache.py: pre-allocated cache with $O(d_k + d_v)$ append. MoAKVCache.append writes at the gamma offset $t*dk$ without copying the existing cache.

Table 3: KV-cache multi-step decode: moa_autoregressive_decode vs. causal SDPA (Scaled Dot-Product Attention). $d_k = d_v = 64$, float32.

T	d_k	d_v	$\ \text{err}\ _\infty$
4	8	8	1.19×10^{-7} ✓
16	64	64	4.77×10^{-7} ✓
64	64	64	5.96×10^{-7} ✓

Table 4: KV-cache traffic accounting: append cost is constant; decode cost grows as $O(td_k)$ per step. MoA total beats the n^2 score-matrix cost for $T \gtrsim 256$; at small T the n^2 term is also small.

T	App. (MB)	Dec. (MB)	n^2 (MB)
16	0.01	0.08	0.01
64	0.03	1.10	0.36
256	0.13	16.97	22.50
1024	0.52	269.22	1433.75

6 Grouped-Query and Multi-Query Attention

6.1 Psi-Selection Expression

In GQA, h_q query heads share $h_{kv} < h_q$ KV heads (h_{kv} divides h_q), with K of shape $\langle h_{kv}, n, d_k \rangle$. The scalar elements of K are defined by ψ -selection with

a full index vector:

$$\langle g, i, j \rangle \psi K \quad \begin{cases} 0 \leq g < h_{kv} & \text{(KV head)} \\ 0 \leq i < n & \text{(sequence position)} \\ 0 \leq j < d_k & \text{(feature)} \end{cases} \quad (9)$$

A partial index vector $\langle g_{kv} \rangle$ selects the entire g_{kv} -th KV head slab:

$$K_g = \langle g_{kv} \rangle \psi K, \quad \rho(K_g) = \langle n, d_k \rangle, \quad (10)$$

whose elements satisfy $\langle i, j \rangle \psi (\langle g_{kv} \rangle \psi K) = \langle g_{kv}, i, j \rangle \psi K$. The group assignment for query head g is $g_{kv} = \lfloor g \times h_{kv} / h_q \rfloor$. This drops the leading head dimension, leaving a 2-D array of shape $\langle n, d_k \rangle$ passed directly to the decode DNF (equation (1)). No Omega lifting is needed and no expanded $\langle h_q, n, d_k \rangle$ tensor is materialized. **Concrete example** ($h_q = 4$, $h_{kv} = 2$, $n = 3$, $d_k = 2$, $group_size = 2$). K has shape $\langle 2, 3, 2 \rangle$ with elements $\langle g, i, j \rangle \psi K$. The two KV slabs (each shape $\langle 3, 2 \rangle$) are:

$$\begin{aligned} \langle 0 \rangle \psi K &= \begin{pmatrix} \langle 0, 0, 0 \rangle \psi K & \langle 0, 0, 1 \rangle \psi K \\ \langle 0, 1, 0 \rangle \psi K & \langle 0, 1, 1 \rangle \psi K \\ \langle 0, 2, 0 \rangle \psi K & \langle 0, 2, 1 \rangle \psi K \end{pmatrix} \\ \langle 1 \rangle \psi K &= \begin{pmatrix} \langle 1, 0, 0 \rangle \psi K & \langle 1, 0, 1 \rangle \psi K \\ \langle 1, 1, 0 \rangle \psi K & \langle 1, 1, 1 \rangle \psi K \\ \langle 1, 2, 0 \rangle \psi K & \langle 1, 2, 1 \rangle \psi K \end{pmatrix} \end{aligned}$$

The four query heads select slabs as follows:

$$\begin{aligned} g = 0 : g_{kv} = 0, \quad \langle 0 \rangle \psi K, \quad \rho = \langle 3, 2 \rangle \\ g = 1 : g_{kv} = 0, \quad \langle 0 \rangle \psi K \quad (\text{same slab as } g = 0) \\ g = 2 : g_{kv} = 1, \quad \langle 1 \rangle \psi K, \quad \rho = \langle 3, 2 \rangle \\ g = 3 : g_{kv} = 1, \quad \langle 1 \rangle \psi K \quad (\text{same slab as } g = 2) \end{aligned}$$

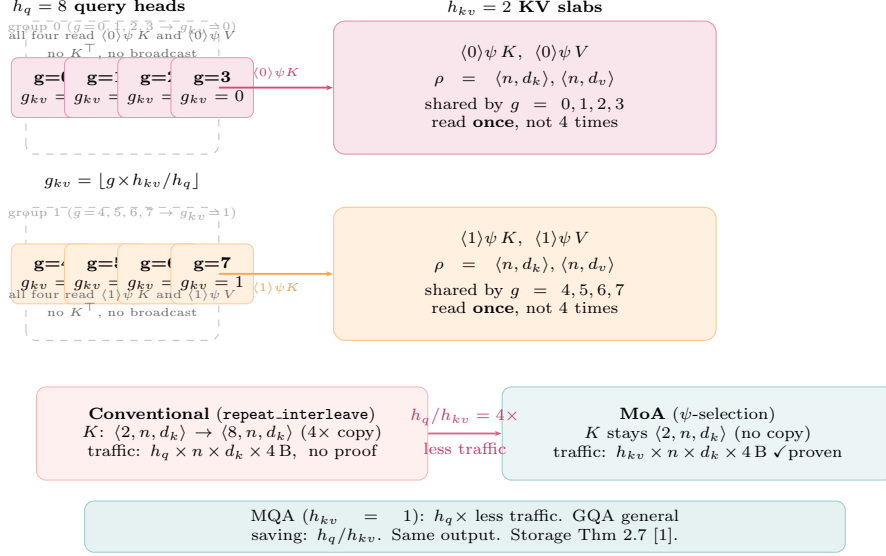


Figure 3: **GQA/MQA via MoA ψ -selection.** $h_q = 8$ query heads are grouped into $h_{kv} = 2$ KV groups. ψ -selection $\langle g_{kv} \rangle \psi K$ maps each query head to its KV slab directly, without materialising the $\langle h_q, n, d_k \rangle$ broadcast that `repeat_interleave` would create. Traffic reduction: $h_q/h_{kv} = 4\times$ here; $h_q\times$ for MQA ($h_{kv} = 1$). Proven minimal by Storage Theorem 2.7.

Query heads 0 and 1 both evaluate $\langle 0 \rangle \psi K$; heads 2 and 3 both evaluate $\langle 1 \rangle \psi K$. By the Storage Theorem, each slab need only be read from DRAM once, giving the $h_q/h_{kv} = 2\times$ KV traffic reduction algebraically.

6.2 Dimension Lifting

Dimension lifting is MoA’s way of mapping to the abstract shape of the target architecture. Starting with the ONF shape, each component of the shape is partitioned one or more times to conform to each architectural target, e.g. processors, memory, etc. For example, The q -head axis and kv -head axis are lifted independently using the dimension-lifting map $N \mapsto \langle \Pi, \lceil N/\Pi \rceil \rangle$ of Section 2.5 of [1]:

$$\langle h_q \rangle \mapsto \langle \Pi_q, \lceil h_q/\Pi_q \rceil \rangle \quad (q\text{-head gang axis}), \quad (11)$$

$$\langle h_{kv} \rangle \mapsto \langle \Pi_{kv}, \lceil h_{kv}/\Pi_{kv} \rceil \rangle \quad (KV \text{ read axis}). \quad (12)$$

With shared-memory caching ($\langle g_{kv} \rangle \psi K$ held in L1 for the h_q/h_{kv} gangs sharing it), the KV traffic reduces from $h_q \times n \times d_k$ to $h_{kv} \times n \times d_k$ bytes – a factor of h_q/h_{kv} . MQA is the special case $h_{kv} = 1$.

6.3 Python Implementation

Standard PyTorch GQA expands the KV heads to h_q by `repeat_interleave`, materialising a full $\langle h_q, n, d_k \rangle$ broadcast tensor:

```
1 def std_gqa_decode(Q, K, V, scale=None):
2     group_size = Q.shape[0] // K.shape[0]
3     # Expand: (h_kv, n, d_k) -> (h_q, n, d_k) -- materialised
4     K_exp = K.repeat_interleave(group_size, dim=0)
5     V_exp = V.repeat_interleave(group_size, dim=0)
6     for g in range(Q.shape[0]):
7         scores = scale*(Q[g:g+1] @ K_exp[g].T)
8         weights = torch.softmax(scores, dim=-1)
9         outputs[g] = (weights @ V_exp[g]).squeeze(0)
```

Listing 6: Standard GQA decode (`pytorch_reference.py: std_gqa_decode`). Materialises the KV broadcast; MoA avoids this via ψ -selection.

The MoA version replaces `repeat_interleave` with ψ -selection $\langle g_{kv} \rangle \psi K$, reading each KV slab once per group rather than h_q/h_{kv} times, giving a proven h_q/h_{kv} traffic reduction.

The MoA formulation for GQA uses ψ -selection (eq. (10)) to map each query head g to its KV group, then applies the decode DNF:

$$\begin{aligned} g_{kv} &= \lfloor g \times h_{kv}/h_q \rfloor, \quad 0 \leq g < h_q, \\ K_g &= \langle g_{kv} \rangle \psi K, \quad \rho(K_g) = \langle n, d_k \rangle, \\ \vec{out}_g &= \vec{a}_g (+ \times \Omega_{\langle 1,2 \rangle}) V_g, \quad \rho(\vec{out}_g) = \langle d_v \rangle. \end{aligned}$$

No $\langle h_q, n, d_k \rangle$ tensor is materialized; each query head reads its assigned slab directly:

```
1 def moa_gqa_decode(Q, K, V, scale=None):
2     # Q : (h_q, d_k)
3     # K : (h_kv, n, d_k) -- one slab per KV head
4     # V : (h_kv, n, d_v)
```

```

5 h_q, dk = Q.shape
6 h_kv = K.shape[0]
7 if scale is None:
8     scale = 1.0 / math.sqrt(dk)
9     group_size = h_q // h_kv
10    out = torch.zeros(h_q, V.shape[2])
11    for g in range(h_q):
12        g_kv = g // group_size # psi-selection: <g_kv> psi
13        # <g_kv> psi K shared across group_size query heads
14        out[g] = _decode_one_head(
15            Q[g], K[g_kv], V[g_kv], scale)
16    return out

```

Listing 7: moa_gqa.py: GQA decode. The ψ -selection eq. (10) is $g_kv = g // group_size$; no repeat-interleave materialization.

6.4 Verification and Traffic

Table 5: GQA/MQA verification: moa_gqa_decode vs. per-head SDPA reference. $n = 64$, $d_k = d_v = 32$, float32.

Configuration	$\ err\ _\infty$
MHA ($h_q = 8, h_{kv} = 8$)	1.19×10^{-7} ✓
GQA ($h_q = 8, h_{kv} = 4$)	1.19×10^{-7} ✓
GQA ($h_q = 8, h_{kv} = 2$)	1.49×10^{-7} ✓
MQA ($h_q = 8, h_{kv} = 1$)	1.79×10^{-7} ✓

Table 6: GQA/MQA KV traffic per decode step. $n = 1024$, $d_k = d_v = 64$, float32, $h_q = 32$. KV amortized over h_{kv} reads (shared memory).

Config	KV (MB)	Total (MB)
MHA $h_{kv}=32$	16.78	16.79
GQA $h_{kv}=8$	4.19	4.21
GQA $h_{kv}=4$	2.10	2.11
MQA $h_{kv}=1$	0.52	0.54

The h_q/h_{kv} traffic reduction is exact and follows algebraically from the ψ -selection expression (10): at $h_{kv} = 1$ (MQA), KV traffic falls from 16.78 MB to 0.52 MB (32 $\times$) on top of the baseline MoA decode savings.

7 Summary and Relationship to Training Kernel

Table 7 collects the verified artifacts.

Training – Algorithm 3 of [1]

Inputs: Q, K, V, O from DRAM

Ω I: $GV \rightarrow \text{DRAM}$; Ω II–III: GA on-stack, g_s in register

Ω IV/V: $GQ, GK \rightarrow \text{DRAM}$

Traffic: $4nd_k + 4nd_v$ B Scratch: $O(n)$

Decode – Algorithm 1 (this report)

Inputs: $\vec{q}, K, V$ from DRAM (+ GQA: h_{kv} heads)

Step I: $\vec{s}, \rho(\vec{s}) = \langle n \rangle$, on stack, Step II: $\vec{a}, \rho(\vec{a}) = \langle n \rangle$, on stack

Step III: $out \rightarrow \text{DRAM}$

Traffic: $O(nd_k + nd_v)$ B GQA bonus: $\div (h_q/h_{kv})$

KV-cache append (Section 5)

Write $\vec{k}_t$ at $t \times d_k$, $\vec{v}_t$ at $t \times d_v$

Traffic: $(d_k + d_v) \times 4$ B per step – constant in t

Figure 4: MoA inference pipeline. All three components are specializations or extensions of the training DNF; no new Omega derivations are needed.

Table 7: Verified inference artifacts. All errors float32 round-off only.

Artifact	File	$\ err\ _\infty$
Python de-code	moa_decode.py	$\leq 2 \times 10^{-7}$
C/OpenACC	moa_decode.openacc.c	0 (exact)
KV-cache	moa_kvcache.py	6×10^{-7}
GQA/MQA	moa_gqa.py	$\leq 2 \times 10^{-7}$

8 Conclusion

Four MoA inference artifacts have been derived, implemented, and numerically verified, each following directly from the forward-pass DNF of [1]. The decode DNF is eqs. (4)–(6) of that paper with i' fixed to the current token: eq. (4) score ψ -reduction, eq. (5) softmax, eq. (6) weighted sum. The C/OpenACC kernel applies the γ ONF stride arithmetic of [1] §7.2; KV-cache append uses plain $\#$ concatenation; GQA/MQA uses ψ -selection and the dimension-lifting Section 2.5 map of [1]. In every case the memory bound is a theorem (Storage Theorem 2.7 of [1]) before code is written, and the DNF $\rightarrow$ ONF $\rightarrow$ C pipeline ports the result to any architecture without re-derivation.

References

- [1] L. Mullin and G. Hains, “Attention at the Theoretical Minimum: A Mathematics of Arrays Framework for Memory-Optimal Transformer Kernels,” *arXiv:2606.07713v1* [cs.LG], June 2026.

- [2] L. Mullin and G. Hains, “Memory-Optimal Transformer Training: Backward Pass and Fused Kernel via the Mathematics of Arrays,” HAL-05659212, June 2026.
- [3] L. M. Mullin, *A Mathematics of Arrays*, Ph.D. dissertation, Syracuse University, 1988.
- [4] S. Thomas, L. Mullin, and K. Świrydowicz, “Improving the Performance of DGEMM with MoA and Cache-Blocking,” in *Proc. ACM ARRAY Workshop (PLDI)*, 2021. <https://doi.org/10.1145/1122445.1122456>
- [5] L. Mullin, “From Array Algebra to Energy Efficiency on GPUs,” *arXiv:2306.11148*, 2023.
- [6] A. Vaswani et al., “Attention is all you need,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.
- [7] N. Shazeer, “Fast Transformer Decoding: One Write-Head is All You Need,” *arXiv:1911.02150*, 2019.
- [8] J. Ainslie et al., “GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints,” in *Proc. EMNLP*, pp. 4895–4901, Singapore, 2023.
- [9] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.
- [10] T. Dao, “FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning,” in *Proc. ICLR*, 2024. *arXiv:2307.08691*.
- [11] T. Dao and D. Y. Fu, “Flash-Decoding for Long-Context Inference,” <https://crfm.stanford.edu/2023/10/12/flash-decoding.html>, 2023.