

1. Architecture: Evolution from XNNPACK to YNNPACK

XNNPACK: REGISTER-FUSED MONOLITHIC KERNELS

Monolithic Hand-Written Kernel

[Quantize × GEMM × Dequantize × Activation]
(~40 of 120+ datatype combinations implemented)

• **Combinatorial Explosion:** Every new fusion, quantization bit-width, or packing layout requires a new kernel.

• **Off the Fast Path:** Unsupported op or datatype combinations fall back to slow reference code.

YNNPACK: COMPOSABLE PRIMITIVES

Quantize / Convert
(fp32, fp16, bf16, int8)

Generalized Dot or Reduce
(int8×int8, int8×int4,
fp32, fp16, bf16)

Dequantize / Elementwise
(fp32, fp16, bf16, int8)

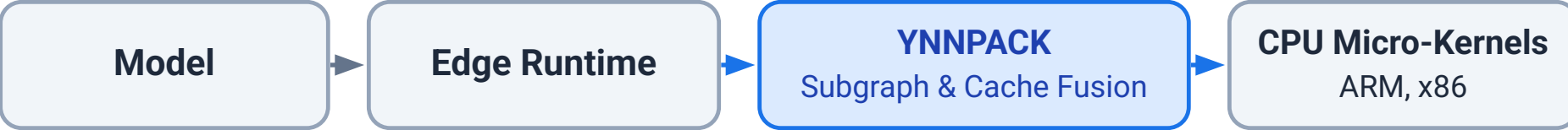
• **Slinky Runtime:** Provides a lightweight runtime to semi-automatically optimize data flow pipelines for locality—fusing loops in L1/L2 cache with zero custom fused kernels.

THE "MORE RISC" APPROACH: DECOMPOSING OPERATORS INTO PRIMITIVES

XNNPACK Operator (Monolithic)	YNNPACK Subgraph Decomposition
Batch Matrix Multiply / Fully Connected	Pack + Dot + Quantization
Convolution / Deconvolution	Stencil + Dot
Static Reduce	Reduce (sum, min, max over arbitrary axes)
Max Pooling / Average Pooling (any pooling)	Stencil + Reduce

• **80% Focus on Dot & Reduce:** Keeping micro-kernels simple and handling packing, quantization, and tiling in the subgraph means optimizing Dot and Reduce accelerates all workloads.

WHERE YNNPACK FITS IN THE ON-DEVICE CPU STACK



2. Case Study: Attention (SDPA) on Edge CPUs

$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}((\mathbf{Q} \cdot \mathbf{K}^T) / \sqrt{d}) \cdot \mathbf{V}$ [Q: [T, d], K, V: [S, d], Scores: [T, S]]

THE $O(T \times S)$ INTERMEDIATE MATERIALIZATION BOTTLENECK

1. Raw Attention Scores (in RAM):

$S[T, S] = Q[T, d] \times K^T[d, S]$

2. Softmax Probabilities (in RAM):

$P[T, S] = \text{softmax}(S[T, S])$

• **Multiple $O(T \times S)$ Materializations:** Naive evaluation materializes full [T, S] matrices in RAM twice—for raw scores ($Q \cdot K^T$) and Softmax probabilities—causing quadratic memory growth and heavy DRAM traffic.

Splitting Output (T) Is Independent

Out[0, :] relies only on Q[0, :], K, V
Out[1, :] relies only on Q[1, :], K, V

• **Independent Query Rows:** Splitting the output across queries (T) is straightforward: each query row computes its attention output independently of all other rows.

Splitting Sequence (S) Breaks Softmax

Denominator = $\sum \exp(\text{Score}_i)$ ($i = 0..S$)
Cannot normalize a partial S chunk alone!

• **The Softmax Reduction Barrier:** Splitting along the sequence dimension (S) is non-trivial: normalizing any block requires the sum of exponentials across the entire row of length S.

THE DECLARATIVE GRAPH CHALLENGE (STATELESS EXECUTION)

• **Flash Attention Uses Stateful Loops:** Hand-written C++ kernels tile S by maintaining loop-carried mutable state (running max and sum) across sequence blocks.

• **Stateless Graph Hurdle:** Declarative graphs express stateless tensor ops and cannot represent loop-carried state without a monolithic custom C++ attention kernel.

3. Optimizing SDPA in YNNPACK: Tiling and Two-Pass Reduction

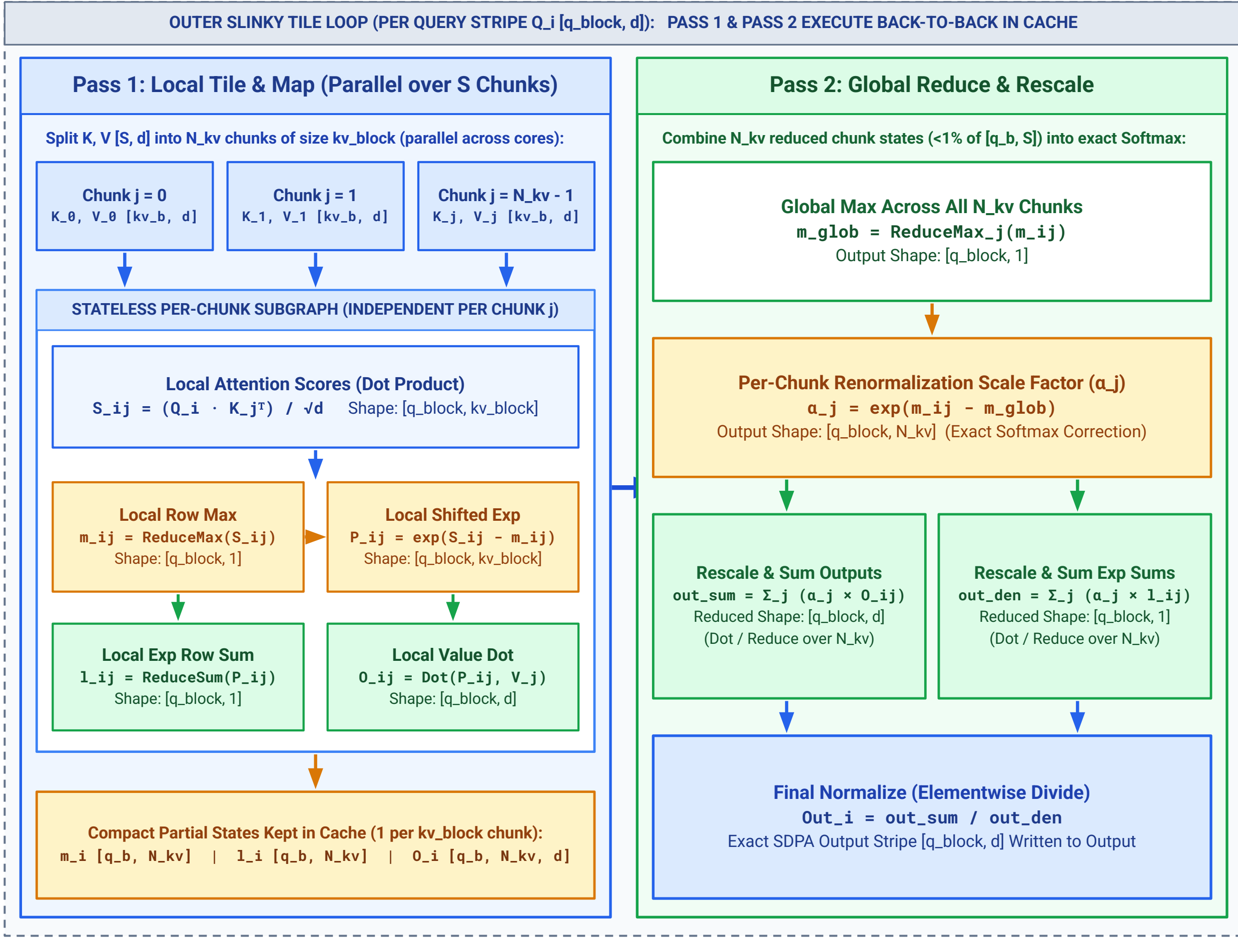
STEP 1: TILING THE OUTPUT DIMENSION (T)

• **Independent Query Stripes:** Because each query row is independent, Slinky automatically tiles the outer loop over T into q_block stripes—shrinking intermediate attention matrices from [T, S] to [q_block, S].

• **Unfused Softmax via Composable Primitives:** YNNPACK has no monolithic Softmax operator; Softmax is decomposed into primitive ops (ReduceMax, Sub, Exp, ReduceSum, Div) that Slinky fuses on the same q_block stripe in cache.

STEP 2: TILING THE SEQUENCE DIMENSION (S) VIA TWO-PASS REDUCTION

• **Associative Two-Pass Reduction over S:** To also tile sequence dimension S without loop-carried state, YNNPACK uses reduction associativity to split Softmax + V-accumulation across S into two stateless passes fused inside each q_block tile:



KEY BENEFITS OF THE TWO-PASS DECLARATIVE ALGORITHM

1. Lock-Free S-Parallelism

Pass 1 computes kv_block chunks independently across CPU cores without locks—enabling S-parallelism when T is small or $T = 1$.

2. Tiny Cache Footprint

Pass 1 reduces [q_block, S] into [q_block, N_kv]. Pass 2 rescales <1% of the data inside the same q_block tile—staying in CPU cache.

3. Micro-Kernel Reuse

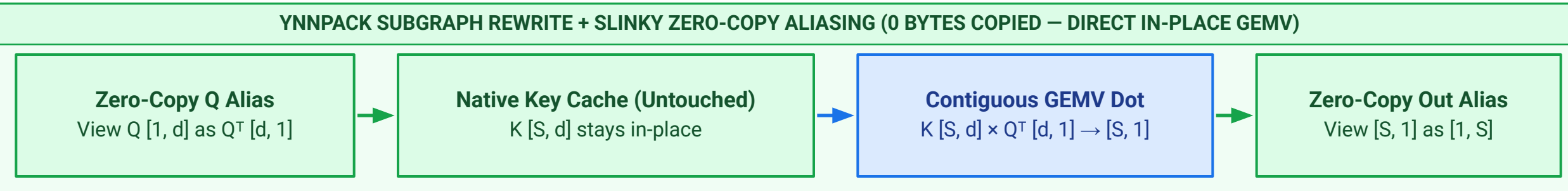
Requires zero custom monolithic attention kernels. Directly composes YNNPACK's existing dot, reduce, and elementwise micro-kernels.

DECODE PHASE ($T = 1$): SUBGRAPH TRANSPOSE REWRITE + ZERO-COPY ALIASING

• **The Decode K-Cache Bottleneck:** For $T = 1$, $Q[1, d]$ is a single row while Key cache K is [S, d]. Physically transposing K to [d, S] before $Q \cdot K^T$ on every step consumes >50% of decode time.

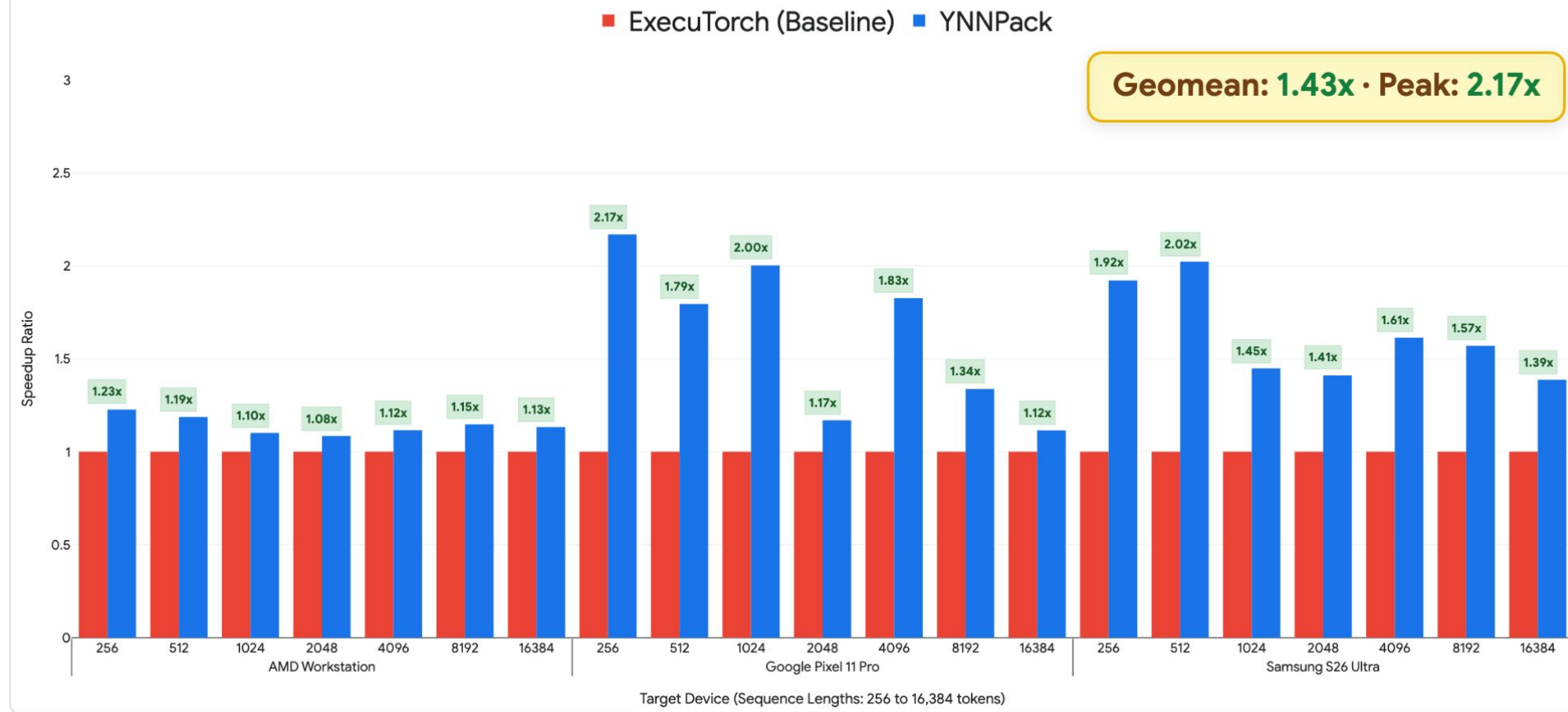
• **Algebraic Transpose Reordering:** YNNPACK rewrites $Q[1, d] \cdot K[S, d]^T$ as $(K[S, d] \cdot Q[1, d]^T)^T$ —keeping [K[S, d], Q[1, d]] in-place as a contiguous GEMV and only transposing 1D vectors.

• **Zero-Copy Buffer Aliasing:** Slinky's buffer aliasing turns those 1D transposes ([1, d] $\rightarrow$ [d, 1] and [S, 1] $\rightarrow$ [1, S]) into zero-cost metadata views, eliminating physical copies completely.

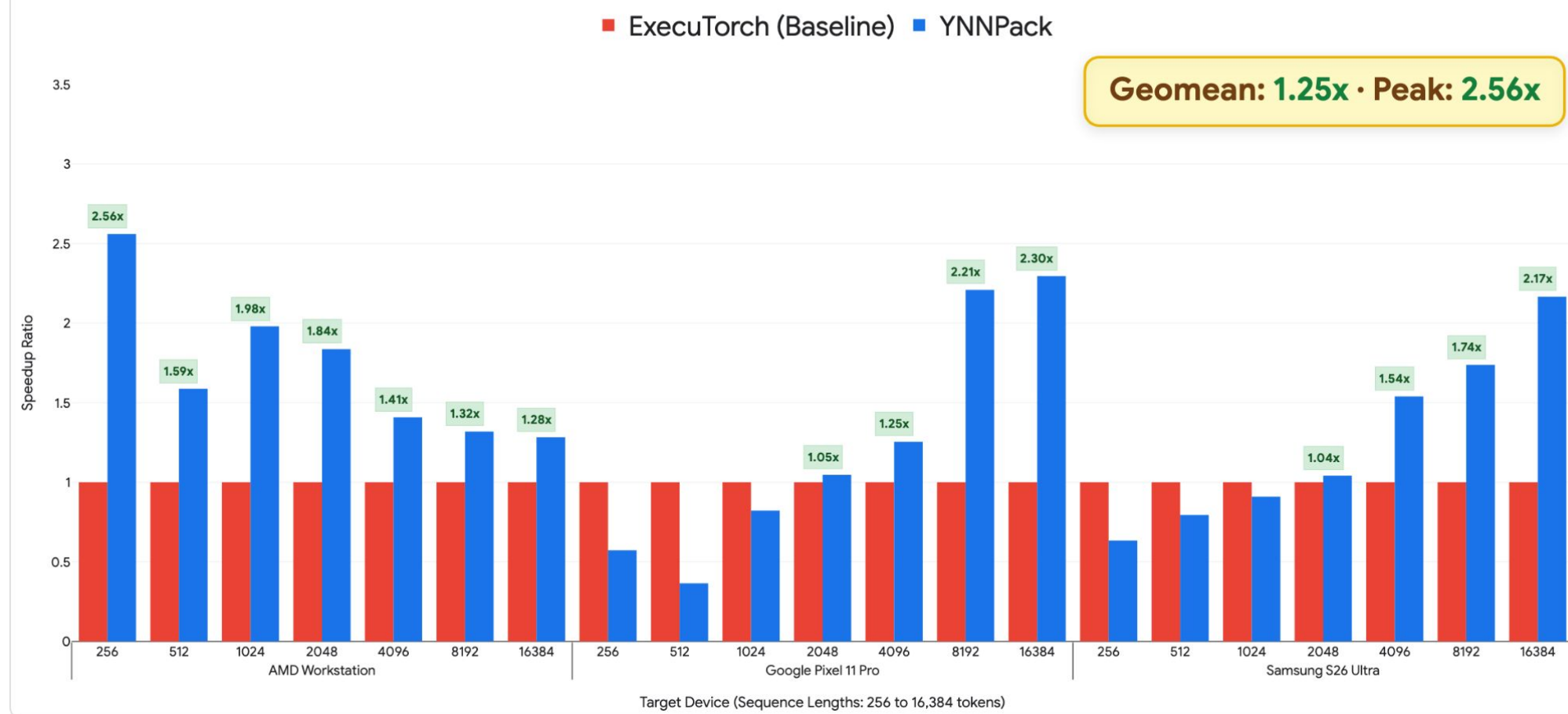


4. Performance Evaluation

Prefill Speedup (vs ExecuTorch)



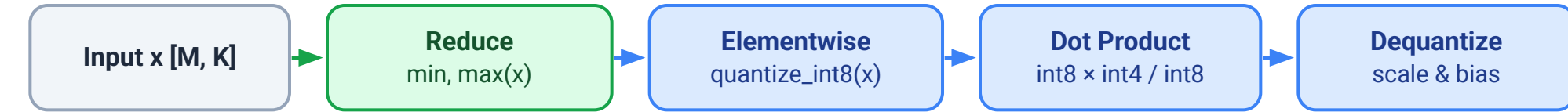
Decode Speedup (vs ExecuTorch)



5. Beyond SDPA: A General Blueprint for Composite Ops

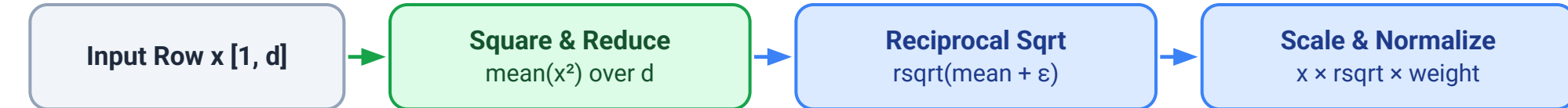
• **Zero-Kernel Composite Ops:** In monolithic engines, composite ops (normalizations, RoPE) either suffer extra DRAM round-trips or need hand-written kernels per datatype. YNNPACK automatically tiles and fuses them so they stay in CPU cache.

EXAMPLE 1: DYNAMICALLY QUANTIZED FULLY CONNECTED LAYER



• **Modular Dynamic Quantization:** Decomposes dynamic quantized linear layers into $\text{reduce_min_max} \rightarrow \text{quantize_int8} \rightarrow \text{dot} \rightarrow \text{dequantize}$. Splitting monolithic GEMMs costs only one extra L1/L2 cache trip while supporting 2/4/8-bit weights.

EXAMPLE 2: RMSNORM / LAYERNORM



• **Fused Row Normalizations:** Built from primitive multiply, reduce, and rsqrt ops. Slinky fuses both row passes so x stays hot in cache across reduction and normalization, avoiding a second DRAM round-trip.

