Fast and Memory-Efficient Wavelet Convolutions via I/O-Aware Reformulation

Amit Aflalo, Shahaf E. Finder, Roy Amoyal, Eran Treister, and Oren Freifeld
The Department of Computer Science, Ben-Gurion University of the Negev, Israel

Abstract

Wavelet convolution (WTConv) has emerged as an increasingly popular drop-in replacement for standard convolutions, expanding a network's receptive field exponentially with the number of decomposition levels while keeping the parameter count linear. However, its reference implementation is severely memory-bound due to excessive data movement through high-bandwidth memory (HBM). We develop an I/O model of WTConv to characterize this bottleneck and use it to guide three algebraic reformulations: (1) recomputing the inexpensive Haar analysis butterfly on chip, (2) collapsing the multi-level synthesis cascade into a single closed-form pass indexed by output-coordinate bits, and (3) folding learned per-channel scales into the convolution weights. Together, these reformulations enable an I/O-aware fused implementation that substantially reduces HBM traffic. We evaluate the WTConvNeXt configuration across decomposition levels and a broad range of tensor shapes. Despite performing comparable arithmetic, the reference WTConv is substantially slower than the depthwise convolution it replaces. Our reformulation reduces modeled HBM traffic by approximately 2.55×, yielding up to a 4.35× training speedup over the reference while roughly halving peak memory usage. Thus, our reformulation preserves the benefits of WTConv while substantially reducing its execution time and memory footprint, removing the systems overhead that previously limited its practical efficiency. Source code is available in the official WTConv repository under fast_wtconv.

1 Introduction

Large receptive fields are important for modern convolutional networks, but obtaining them with conventional convolutions is expensive. Stacking small kernels expands the theoretical receptive field only linearly with depth, while directly increasing kernel size incurs parameter and arithmetic costs proportional to the kernel area (Luo et al., 2016; Ding et al., 2022; Liu et al., 2023). Fortunately, the widely-used WTConv (Finder et al., 2024) offers an appealing alternative: it applies small depthwise convolutions across the progressively downsampled levels of a wavelet decomposition, so the receptive field grows exponentially with the number of levels while the parameter count grows only linearly. This makes WTConv a drop-in replacement for large depthwise convolutions. WTConv has also produced accuracy and robustness gains in architectures including ConvNeXt and MobileNetV2 (Liu et al., 2022; Sandler et al., 2018).

Yet this advantage does not translate into execution speed. In the original WTConv paper, WTConvNeXt uses WTConv with a 5×5 kernel (k=5) as a replacement for the 7×7 depthwise convolution in a ConvNeXt block. In this setting, the reference implementation is *slower than the convolution it replaces*: over a full training step, it trails the depthwise 7×7 baseline by 2.46–3.42× in `fp32` and 1.53–2.20× in `fp16`; at inference, the corresponding gaps are 3.79–5.28× and 2.05–2.70×. These gaps cannot be explained by arithmetic alone. For an input containing $N = B \cdot C \cdot H \cdot W$ elements, WTConv with k=5 performs $58N$–$69N$ multiply–accumulates as the number of decomposition levels increases from L=1 to L=5, compared with $49N$ for the 7×7 depthwise baseline. This increase in arithmetic is far smaller than the observed latency gap. Thus, despite its favorable parameter scaling, the reference operator incurs a substantial wall-clock penalty. Closing this performance gap is the primary objective of this work.

We show that the discrepancy arises because FLOPs are the wrong cost model for this operator. The reference WTConv implementation has an arithmetic intensity of only ≈ 1.63 FLOP/byte in `fp32`, placing it deep in the memory-bound regime on modern GPUs. Its execution is therefore dominated not by the cost of the Haar transform or the depthwise convolutions themselves, but by repeatedly materializing intermediate wavelet coefficients and reconstructions in high-bandwidth memory (HBM). Under a tensor-materialization I/O model, a forward evaluation incurs $17.75N$–$21.32N$ element reads and writes to global memory for an input containing N elements.

This observation suggests a different optimization target: rather than reducing arithmetic, we reformulate WTConv so that mathematical intermediates need not become memory-resident intermediates. We exploit three properties of the operator. First, Haar analysis uses only signed additions and fixed power-of-two scaling and can be recomputed on chip inside the depthwise convolution. Second, linearity of Haar synthesis allows the entire L-level reconstruction recursion to be written as a single closed-form sum whose signs and coefficient addresses are determined by bits of the output coordinate. Third, the learned per-channel scales can be folded into the convolution weights. Together, these transformations preserve the same mathematical operator while eliminating the large intermediate tensors responsible for the majority of its data movement. The resulting formulation reduces modeled HBM traffic by 2.54–2.56× for $L = 1, \ldots, 5$.

We implement this formulation in CUDA and evaluate both inference and full training steps over a broad sweep of tensor shapes, decomposition levels, and precisions. For a full training step, the implementation is 3.71–4.35× faster than the reference in `fp32` and 2.68–3.09× faster in `fp16`, while reducing peak memory by a factor of 1.83–2.31. More importantly, the systems reformulation reverses the practical comparison that motivates the work: over a training step the optimized WTConv at k=5 is 1.27–1.50× faster in `fp32` and 1.40–1.76× faster in `fp16` than the depthwise 7×7 convolution it is proposed to replace, at every decomposition level tested. Training is the regime in which the comparison matters most, since it is where the materialized intermediates are both written and re-traversed; at inference the reversal is complete in `fp16` but not in `fp32`, where the fused layer remains within 3–17% of the depthwise baseline rather than ahead of it (§ 5.2).

Contributions.

- **An I/O cost model for WTConv.** We derive an element-level accounting of HBM traffic for the reference implementation, $Q_{\text{ref}} = 7N + \frac{43}{3}N(1 - 4^{-L})$, which is independent of the kernel size, and use a roofline analysis to show that WTConv's arithmetic intensity is a factor of roughly 31 below the compute-saturation ridge point in `fp32` at k=5. This identifies data movement, rather than arithmetic, as its dominant cost.
- **An algebraically exact, I/O-aware reformulation.** We combine register-resident Haar analysis, a closed-form bit-indexed synthesis over all decomposition levels, and scale folding to eliminate unnecessary HBM-resident intermediates. The resulting formulation reduces predicted forward-pass traffic by 2.54–2.56× across $L = 1, \ldots, 5$ while preserving the WTConv operator up to floating-point evaluation order.
- **End-to-end empirical validation.** Across all measured configurations, our CUDA implementation substantially reduces training and inference latency and peak memory relative to the reference WTConv implementation, and over a training step outperforms the depthwise 7×7 convolution WTConv is proposed to replace. We additionally verify numerical agreement for the forward output and all parameter gradients.

2 Background and Related Work

2.1 The WTConv operator

Let $\mathsf{X} \in \mathbb{R}^{B \times C \times H \times W}$ denote the input, and let $N = B \cdot C \cdot H \cdot W$ be its element count, the unit in which we quote every I/O cost. Let $*_{\text{dw}}$ denote depthwise convolution (one $k \times k$ kernel per channel, zero-padded to preserve resolution) and let $\odot$ denote per-channel scaling.

Let $\mathcal{A}$ denote one level of Haar analysis, which splits each channel into four half-resolution subbands $(\mathrm{LL}, \mathrm{LH}, \mathrm{HL}, \mathrm{HH})$, and let $\mathcal{A}^\top$ denote its synthesis, which inverts it exactly (Proposition 1). For a tensor carrying a subband axis, let $[\,\cdot\,]_{\mathrm{LL}}$ denote the low-pass band, $[\,\cdot\,]_{\mathrm{H}}$ the three high-pass bands, and $\|$ their concatenation. WTConv preserves the channel count. It learns a $k \times k$ kernel and a per-channel scale at each level, $(\mathbf{W}^{(\ell)}, \mathbf{s}^{(\ell)})$ for $\ell = 1, \ldots, L$, together with a full-resolution base convolution $(\mathbf{W}^{(0)}, b, \mathbf{s}^{(0)})$, giving $\mathcal{O}(Lk^2C)$ parameters in total. Algorithm 1 states the operator as the reference implements it.

Algorithm 1 WTConv, reference implementation (Finder et al., 2024)

Require: input X, levels L, parameters $\{\mathbf{W}^{(\ell)}, \mathbf{s}^{(\ell)}\}_{\ell=0}^{L}$, bias b

$\mathsf{X}^{(0)} \leftarrow \mathsf{X}$ ▷ decomposition carrier
for $\ell = 1, \ldots, L$ **do** ▷ downward: decompose and filter
  $\mathsf{Y}^{(\ell)} \leftarrow \mathcal{A}\,\mathsf{X}^{(\ell-1)}$ ▷ Haar analysis
  $\mathsf{Z}^{(\ell)} \leftarrow \mathbf{s}^{(\ell)} \odot \left(\mathsf{Y}^{(\ell)} *_{\mathrm{dw}} \mathbf{W}^{(\ell)}\right)$ ▷ filter, then scale
  $\mathsf{X}^{(\ell)} \leftarrow \left[\mathsf{Y}^{(\ell)}\right]_{\mathrm{LL}}$ ▷ carry the *raw*, unfiltered low-pass band
end for
$R^{(L+1)} \leftarrow 0$
for $\ell = L, \ldots, 1$ **do** ▷ upward: reconstruct and accumulate
  $R^{(\ell)} \leftarrow \mathcal{A}^\top\left(\left(\left[\mathsf{Z}^{(\ell)}\right]_{\mathrm{LL}} + R^{(\ell+1)}\right) \,\middle\|\, \left[\mathsf{Z}^{(\ell)}\right]_{\mathrm{H}}\right)$
end for
return $\mathbf{s}^{(0)} \odot \left(\mathsf{X} *_{\mathrm{dw}} \mathbf{W}^{(0)} + b\right) + R^{(1)}$ ▷ base path + reconstruction

2.2 Haar transform differentiation

For each non-overlapping 2×2 block, the normalized Haar transform maps four input samples to four subband coefficients using the following matrix.

Proposition 1 (Symmetry and self-inversion)**.** $\mathbf{H} = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$ *satisfies* $\mathbf{H}^\top = \mathbf{H}$ *and* $\mathbf{H}^2 = \mathbf{I}$. *Therefore,* $\mathbf{H}^{-1} = \mathbf{H}^\top = \mathbf{H}$.

The full analysis operator $\mathcal{A}$ applies $\mathbf{H}$ independently to every block, and synthesis applies $\mathcal{A}^\top$. For $\mathbf{y} = \mathbf{H}\mathbf{x}$, the chain rule gives

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \mathbf{H}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{y}} = \mathbf{H}\frac{\partial \mathcal{L}}{\partial \mathbf{y}}. \tag{1}$$

Thus, backward through analysis applies synthesis, and backward through synthesis applies analysis. The Haar step needs no separate derivative; convolution and scale gradients are computed as usual.

2.3 The roofline

The roofline model (Williams et al., 2009) bounds attainable throughput by $\min(\pi, \beta I)$, where π is peak compute, β peak bandwidth, and I the arithmetic intensity in FLOP/byte. The ridge point $I^\star = \pi/\beta$ separates memory-bound from compute-bound operation. For the RTX A6000 used throughout, $\beta = 768$ GB/s and $\pi \approx 38.7$ TFLOP/s in `fp32`, so $I^\star \approx 50$ FLOP/byte. Any operator with $I \ll I^\star$ is bandwidth-limited: its runtime is governed by the bytes it moves rather than by how the arithmetic is scheduled. The bound is an upper bound on attainable throughput and not an equality, so this identifies traffic as the quantity to minimize without implying that runtime is proportional to it. The practical content of this paper is that WTConv is deeply in that regime, so minimizing traffic is not merely one optimization among many; it is the dominant optimization target.

2.4 Kernel fusion and memory-bound operators.

It is well established that data movement, rather than arithmetic, is often the dominant cost in deep learning workloads (Ivanov et al., 2021; Wahib & Maruyama, 2014). FlashAttention (Dao et al., 2022; Dao, 2024) provides the canonical example: by exactly reformulating the computation so that intermediate results remain

on chip, it substantially reduces the practical cost of attention. Our work applies the same principle to a different operator, but exploits additional algebraic structure: the fused transform uses only signed additions and fixed power-of-two scaling, and is symmetric and self-inverse. These properties make recomputation inexpensive and allow analysis and synthesis to exchange roles during backpropagation. General-purpose compilers (Ragan-Kelley et al., 2013; Chen et al., 2018; Sabne, 2020; Ansel et al., 2024) can automate fusion across elementwise and reduction chains, but the reformulation developed in § 4.2 (which collapses an L-step recursion into a closed-form, bit-indexed sum) depends on specific properties of the Haar basis that are not available to a general-purpose compiler.

3 WTConv is memory-bound

3.1 I/O cost of the reference implementation

The reference implementation expresses each level as a chain of framework primitives, every one of which reads its input from HBM and writes its output back (Figure 1a). We count elements crossing the HBM boundary: Each stage reads its input tensor once and writes its output tensor once. Weight traffic is $\mathcal{O}(Ck^2)$ and therefore negligible against $\mathcal{O}(N)$. Writing $N_\ell = N/4^{\ell-1}$ for the element count entering level ℓ:

- *Analysis*, per level: the Haar transform is executed as a grouped stride-2 `conv2d` against a constant $\pm\frac{1}{2}$ filter ($2N_\ell$); the depthwise convolution over the $4C$-channel coefficient tensor ($2N_\ell$); the scale multiply ($2N_\ell$). Total $6N_\ell$.
- *Synthesis*, per level: the cross-level low-pass add ($\frac{3}{4}N_\ell$); the concatenation of the summed low-pass band with the three high bands ($2N_\ell$); the `conv_transpose2d` ($2N_\ell$). Total $\frac{19}{4}N_\ell$.
- *Base path*: convolution ($2N$), scale multiply ($2N$), final add ($3N$). Total $7N$.

Using $\sum_{\ell=1}^{L} 4^{-(\ell-1)} = \frac{4}{3}(1-4^{-L})$,

$$Q_{\text{ref}} \;=\; \underbrace{7N}_{\text{base path}} \;+\; \underbrace{\left(6+\tfrac{19}{4}\right)}_{\text{per-level cost}} \cdot \underbrace{\tfrac{4}{3}N\left(1-4^{-L}\right)}_{\sum_{\ell=1}^{L} N_\ell} \;=\; 7N + \tfrac{43}{3}N\left(1-4^{-L}\right), \tag{2}$$

rising from $17.75N$ at $L=1$ to $21.33N$ as $L\to\infty$; that is, evaluating the layer moves between 18 and 21 times the input tensor's worth of data through HBM. Note that k does not appear: the kernel size sets how much arithmetic each resident element receives, not how many elements cross the boundary. For a representative B=8, C=64, H=W=256 input, N = 33.6M elements (128 MiB in `fp32`) and the reference therefore moves roughly 2.9 GB per forward pass.

3.2 Arithmetic intensity of the reference implementation

The base convolution performs k^2 multiply–accumulates per element. At level ℓ, the depthwise convolution over the coefficient tensor costs a further k^2 per element of N_ℓ, and the analysis and synthesis each cost 4, since the reference executes both as 2×2 convolutions rather than as additions. Summing over levels with the same geometric factor as before,

$$\text{MAC} \;=\; N\left[k^2 + (k^2+8)\cdot\tfrac{4}{3}\left(1-4^{-L}\right)\right]. \tag{3}$$

For $k=5$ and $L=5$ this is $69.0N$ multiply–accumulates, or $137.9N$ FLOPs, against $4Q_{\text{ref}} = 85.3N$ bytes in `fp32`:

$$I_{\text{ref}} \;=\; \frac{137.9N}{85.3N} \;\approx\; 1.63 \text{ FLOP/byte}. \tag{4}$$

Against the ridge point $I^\star \approx 50$ (§ 2.3), WTConv's arithmetic intensity is lower by a factor of roughly 31 in `fp32`; equivalently, it is approximately 3.2% of the ridge-point intensity. Enlarging the kernel does not change this conclusion, only its margin: increasing k increases the arithmetic work while leaving Eq. 2 untouched, so even at k=5 the operator remains more than an order of magnitude below the ridge point. Arithmetic is not the bottleneck: accelerating the transform itself would have little effect, whereas reducing memory traffic directly targets the dominant cost.

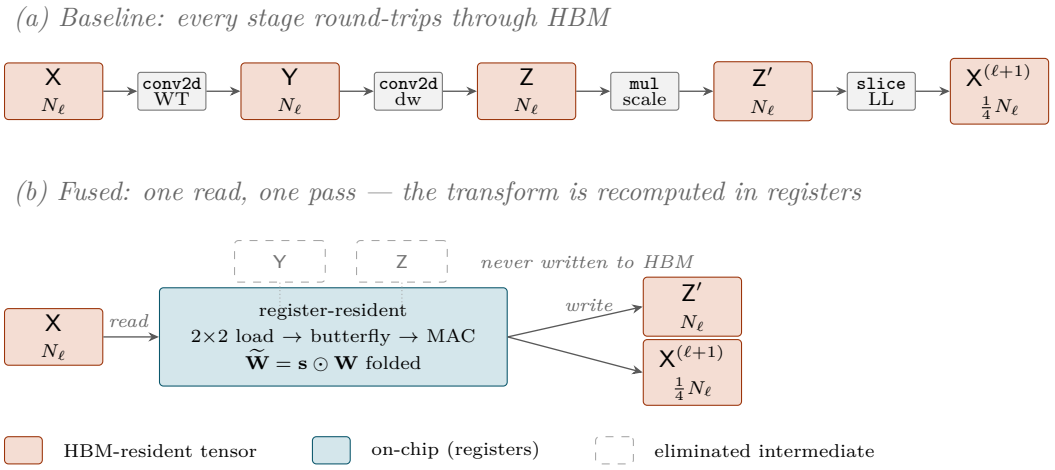

Figure 1: HBM dataflow for one WTConv decomposition level ℓ, with $N_\ell = N/4^{\ell-1}$ the number of elements entering that level. (a) The reference implementation expresses the level as a chain of framework primitives; every stage materializes a full-size tensor in HBM. (b) The fused formulation reads the input once, evaluates the Haar butterfly and the depthwise convolution on chip against weights that already carry the learned scale, and writes only the results. Y and Z are never written.

4 An I/O-Aware formulation

To eliminate the severe memory bottleneck of the reference WTConv, we derive an I/O-aware formulation based on three exact algebraic reformulations. These reformulations leave the underlying mathematical operator unchanged, differing from the reference implementation solely in floating-point evaluation order. Together, they eliminate the unnecessary Haar-analysis and reconstruction intermediates from high-bandwidth memory (HBM) by performing multi-stage computations directly in GPU registers and shared memory across all decomposition levels.

4.1 Fusing Haar analysis with depthwise convolution

For a 2×2 spatial patch $\left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]$, the normalized 2D Haar analysis transform maps four input pixels to subband coefficients $(\mathrm{LL}, \mathrm{LH}, \mathrm{HL}, \mathrm{HH})$ as follows:

$$\begin{aligned} \mathrm{LL} &= \tfrac{1}{2}(a+b+c+d), & \mathrm{LH} &= \tfrac{1}{2}(a+b-c-d), \\ \mathrm{HL} &= \tfrac{1}{2}(a-b+c-d), & \mathrm{HH} &= \tfrac{1}{2}(a-b-c+d). \end{aligned} \tag{5}$$

Evaluating Eq. 5 requires only signed additions and fixed scaling by $\frac{1}{2}$. In contrast, the reference implementation materializes the entire coefficient tensor $\mathsf{Y}^{(\ell)}$ in HBM via grouped convolutions with $\pm\frac{1}{2}$ filters. Each level runs Haar analysis, depthwise convolution, and learned scaling as separate passes. Each pass reads and writes an N_ℓ-element tensor, moving $6N_\ell$ elements in total. Our fused approach computes the Haar coefficients inside the depthwise convolution and folds the scale into its weights (§ 4.3). Because WTConv reaches only 1.63 FLOP/byte (§ 3), recomputing the coefficients is much cheaper than materializing them in HBM.

Naive fusion would recompute the transform for every convolution tap and load each input pixel k^2 times. Instead, each thread block transforms the 2×2 blocks for one output tile, including a coefficient-space halo of radius $R = (k-1)/2$, and stages the four subbands in shared memory. All convolution taps reuse the transformed tile, so each 2×2 input block is transformed only once per output tile. Except at the deepest level, the fused approach reads N_ℓ input elements, writes N_ℓ filtered coefficients for synthesis, and writes $\frac{1}{4}N_\ell$ unfiltered low-pass coefficients for the next level. It therefore moves $\frac{9}{4}N_\ell$ elements, versus $6N_\ell$ for the reference. The deepest level has no successor and moves $2N_L$.

4.2 Collapsing the synthesis cascade

The reference implementation executes reconstruction as an L-step sequential loop, where each level reads the lower-level reconstruction from HBM, adds it to its low-pass subband, and applies transposed convolution. This imposes L strict sequential dependencies and incurs an I/O cost of $\frac{19}{4}N_\ell$ per level.

Because Haar synthesis is linear, the low-pass carrier acts as a linear accumulator across levels. As a result, the multi-level synthesis cascade can be collapsed into a single closed-form pass across all L levels. Haar synthesis at level ℓ expands each subband coefficient over a $2^\ell \times 2^\ell$ pixel block using a fixed sign pattern determined by the spatial location of the output pixel within that block.

Proposition 2 (Bit-indexed single-pass synthesis)**.** *Let (y, x) denote an output pixel coordinate, and let level $\ell \in \{1, \dots, L\}$ have subband coefficients $(c_{\mathrm{LL}}^{(\ell)}, c_{\mathrm{LH}}^{(\ell)}, c_{\mathrm{HL}}^{(\ell)}, c_{\mathrm{HH}}^{(\ell)})$ at coarse grid location $(\lfloor y/2^\ell \rfloor, \lfloor x/2^\ell \rfloor)$. Define the coordinate parity signs at level ℓ as follows:*

$$s_y^{(\ell)} = (-1)^{\lfloor y/2^{\ell-1} \rfloor \bmod 2}, \qquad s_x^{(\ell)} = (-1)^{\lfloor x/2^{\ell-1} \rfloor \bmod 2}. \tag{6}$$

The reconstructed pixel value $R_{y,x}$ across all L levels is given by:

$$R_{y,x} = \sum_{\ell=1}^{L} 2^{-\ell} \Big(c_{\mathrm{LL}}^{(\ell)} + s_y^{(\ell)} c_{\mathrm{LH}}^{(\ell)} + s_x^{(\ell)} c_{\mathrm{HL}}^{(\ell)} + s_y^{(\ell)} s_x^{(\ell)} c_{\mathrm{HH}}^{(\ell)} \Big). \tag{7}$$

A derivation of the coordinate signs and normalization in Eq. 7 is given in Appendix A.

By evaluating Eq. 7 in a single pass, a GPU thread computing pixel (y, x) iterates from level L down to 1, addressing coarse coefficients via coordinate bit-shifts and obtaining signs through bitwise parity checks. This eliminates all intermediate low-pass tensors and sequential kernel dependencies. Furthermore, the base-path convolution output is folded directly into the final output store, eliminating a separate full-resolution tensor addition.

4.3 Folding the learned scale

The reference implementation applies learned per-channel scaling s_c after convolution ($\mathsf{Z}_c = s_c(\mathsf{X} *_{\mathrm{dw}} \mathbf{W})_c$) as a standalone elementwise pass. At an arithmetic intensity of $\frac{1}{8}$ FLOP/byte in `fp32` (one multiply per element read and written), this standalone pass is strongly memory-bound and adds an avoidable read–write traversal of the tensor.

By the linearity of convolution, the scale factor is folded into the depthwise weights and, on the base path, into its bias:

$$\widetilde{\mathbf{W}}_c = s_c \odot \mathbf{W}_c, \qquad \widetilde{b}_c = s_c\, b_c, \tag{8}$$

so that $\mathsf{Z} = \mathsf{X} *_{\mathrm{dw}} \widetilde{\mathbf{W}} + \widetilde{b}$ is computed in a single pass with zero extra runtime cost.

During backpropagation, exact parameter gradients with respect to the unscaled weight $\mathbf{W}_c$, bias b_c, and scale s_c are efficiently recovered as follows:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_c} = s_c \frac{\partial \mathcal{L}}{\partial \widetilde{\mathbf{W}}_c}, \qquad \frac{\partial \mathcal{L}}{\partial b_c} = s_c \frac{\partial \mathcal{L}}{\partial \widetilde{b}_c}, \qquad \frac{\partial \mathcal{L}}{\partial s_c} = \left\langle \frac{\partial \mathcal{L}}{\partial \widetilde{\mathbf{W}}_c}, \mathbf{W}_c \right\rangle + \frac{\partial \mathcal{L}}{\partial \widetilde{b}_c} b_c. \tag{9}$$

s_c enters both folds, so its gradient collects a term from each.

4.4 The resulting I/O cost

Collecting the reformulations across all levels $1, \dots, L$, with $N_\ell = N/4^{\ell-1}$:

- *Base path*: the folded vendor convolution reads N and writes N ($2N$).
- *Analysis & Depthwise pass*, levels $1 \le \ell \le L$: reads N_ℓ, writes N_ℓ convolved coefficients and $\frac{1}{4}N_\ell$ raw low-pass coefficients ($\frac{9}{4}N_\ell$ each, less the $\frac{1}{4}N_L$ the deepest level does not write).

- *Single-pass Synthesis*: reads all subband coefficients $\sum_{\ell=1}^{L} N_\ell$ and base-path output N, writing the final output N ($\sum_{\ell=1}^{L} N_\ell + 2N$).

Using $\sum_{\ell=1}^{L} N_\ell = \frac{4}{3}N\big(1 - 4^{-L}\big)$, the total memory traffic is:

$$Q_{\text{fused}} \;=\; 4N \;+\; \tfrac{13}{3}\,N\big(1 - 4^{-L}\big) \;-\; N\,4^{-L}. \tag{10}$$

Comparing this to the reference memory traffic Q_{ref} (Eq. 2), the theoretical reduction ratio in HBM data movement is:

$$\frac{Q_{\text{ref}}}{Q_{\text{fused}}} = \frac{7 + \frac{43}{3}\big(1 - 4^{-L}\big)}{4 + \frac{13}{3}\big(1 - 4^{-L}\big) - 4^{-L}} \xrightarrow[L\to\infty]{} \frac{64}{25} = 2.56. \tag{11}$$

As $L \to \infty$, the fused formulation achieves up to a 2.56× reduction in memory traffic over the reference implementation. Table 1 summarizes the exact element counts and traffic reduction ratios for decomposition levels L=1, . . . , 5. Like Eq. 2, neither count depends on k, so the traffic argument of this section is stated once and holds at every kernel size; § 5.4 verifies this empirically at k=3.

We test both predictions using hardware performance counters in Appendix B. Under the modeled access pattern, the fused and reference traffic agree with their predictions within 1% and 4%, respectively. Across tensor shapes, fused traffic remains within 5% of the prediction, whereas the reference can exceed it by up to 39% when cuDNN selects kernels with additional DRAM accesses. Thus, Q_{ref} should be interpreted as algorithmic traffic and, under such dispatches, as a lower bound.

Table 1: Elements crossing the HBM boundary per forward evaluation, in units of $N = B \cdot C \cdot H \cdot W$, from Eq. 2, Eq. 10, and Eq. 11. Both counts are independent of the kernel size k. The last row is a ratio of *traffic*, not of runtime; see the discussion below.

	Decomposition levels L				
	1	2	3	4	5
Reference, Q_{ref}	$17.75N$	$20.44N$	$21.11N$	$21.28N$	$21.32N$
Fused, Q_{fused}	$7.00N$	$8.00N$	$8.25N$	$8.31N$	$8.33N$
Traffic reduction factor	2.54×	2.55×	2.56×	2.56×	2.56×

5 Results

5.1 Setup

All measurements are single-layer microbenchmarks on an RTX A6000, except the end-to-end networks in § 5.5 and the cross-hardware validation in § 5.7, which repeats the layer sweep on a second, architecturally different GPU. We sweep $C \in \{32, 64, 128\}$, $H = W \in \{128, 256, 512\}$ at B=8, and $L \in \{1, \ldots, 5\}$ in `fp32` and `fp16`. Timings are averaged over 50 iterations following 20 warm-up iterations.

Baselines. WTConv (reference and fused) runs at k=5 throughout § 5.2 and § 5.3; § 5.4 repeats the sweep at k=3, and § 5.7 repeats it at k=5 on a second GPU. We evaluate plain depthwise convolutions under two protocols: *matched* (k=5) to isolate wavelet overhead, and *drop-in* (k=7) representing the standard ConvNeXt replacement (Liu et al., 2022).

Reporting. Table values are the geometric mean over the (C, H) sweep of the per-configuration ratio between the WTConv reference and the evaluated method. The reference is always 1.00×. Latency is reported as speedup (reference/method; higher is better) and memory as footprint fraction (method/reference; lower is better). Appendix C verifies numerical agreement.

5.2 Layer latency

Training step. Table 2 reports full training-step (forward and backward) latency. The reference operator is slower than the convolution it replaces: the depthwise 7×7 convolution of a ConvNeXt block completes a training step 2.46–3.42× faster in `fp32` and 1.53–2.20× faster in `fp16`; at matched kernel size the margin widens to 4.89–6.79× (`fp32`, depthwise 5×5).

Table 2: Latency of a full training step (forward and backward) relative to the reference WTConv (1.00×; higher is faster).

Method	Precision	Decomposition levels L				
		1	2	3	4	5
Depthwise conv, k=5	`fp32`	4.89×	6.21×	6.60×	6.74×	6.79×
	`fp16`	6.28×	8.12×	8.67×	8.90×	9.07×
Depthwise conv, k=7	`fp32`	2.46×	3.13×	3.33×	3.40×	3.42×
	`fp16`	1.53×	1.97×	2.11×	2.16×	2.20×
WTConv, reference	`fp32`	1.00×	1.00×	1.00×	1.00×	1.00×
	`fp16`	1.00×	1.00×	1.00×	1.00×	1.00×
WTConv, **fused**	`fp32`	3.71×	4.23×	4.34×	4.35×	4.33×
	`fp16`	2.68×	3.02×	3.08×	3.09×	3.08×

Table 3: Forward latency of every method relative to the reference WTConv (1.00×; higher is faster).

Method	Precision	Decomposition levels L				
		1	2	3	4	5
Depthwise conv, k=5	`fp32`	7.35×	9.28×	9.87×	10.10×	10.24×
	`fp16`	9.79×	11.95×	12.54×	12.75×	12.89×
Depthwise conv, k=7	`fp32`	3.79×	4.79×	5.09×	5.21×	5.28×
	`fp16`	2.05×	2.50×	2.62×	2.66×	2.70×
WTConv, reference	`fp32`	1.00×	1.00×	1.00×	1.00×	1.00×
	`fp16`	1.00×	1.00×	1.00×	1.00×	1.00×
WTConv, **fused**	`fp32`	3.67×	4.20×	4.32×	4.35×	4.37×
	`fp16`	3.38×	3.60×	3.60×	3.57×	3.53×

Fusion reverses this. The fused layer is 3.71–4.35× faster than the reference in `fp32` and 2.68–3.09× in `fp16`. The ratio grows with L and then flattens, because deeper decompositions give the reference more intermediates to materialize while Eq. 10 is nearly flat in L; the change from L=4 to L=5 is below 1% in both precisions, as Eq. 11 predicts. At every level and in both precisions this suffices to overturn the comparison above: against the depthwise 7×7 convolution it replaces, the fused layer trains 1.27–1.50× faster in `fp32` and 1.40–1.76× faster in `fp16`. It does not win against a depthwise convolution at its *own* kernel size, which remains 1.32–1.57× faster in `fp32` and 2.34–2.94× in `fp16`; that kernel moves $2N$ elements against the fused layer's $7N$–$8.33N$ and carries no decomposition at all, so the direction is expected. The speedup over the reference is *smaller* in `fp16` than in `fp32` because half precision halves the bytes moved by both implementations, and the reference, which moves more of them, benefits more.

Inference. Table 3 reports forward-only latency. Against the reference the picture is unchanged: the fused layer is 3.67–4.37× faster in `fp32` and 3.38–3.60× in `fp16`, and the reference remains slower than both plain convolutions in the table, including the depthwise 7×7 it is meant to replace, by 3.79–5.28× and 2.05–2.70×. The comparison against the baselines, however, does not reverse as cleanly as over a training step. Against the depthwise 7×7 convolution the fused layer wins in `fp16` (1.31–1.65×) but not in `fp32`, where it sits at 0.83–0.97×.

5.3 Peak memory

Table 4: Peak allocated memory over a training step, as a fraction of what the reference WTConv allocates (lower is better)

Method	Precision	Decomposition levels L				
		1	2	3	4	5
Depthwise conv, k=5	`fp32`	0.63×	0.44×	0.44×	0.44×	0.44×
	`fp16`	0.73×	0.51×	0.51×	0.51×	0.51×
Depthwise conv, k=7	`fp32`	0.63×	0.44×	0.44×	0.44×	0.44×
	`fp16`	0.73×	0.51×	0.51×	0.51×	0.51×
WTConv, reference	`fp32`	1.00×	1.00×	1.00×	1.00×	1.00×
	`fp16`	1.00×	1.00×	1.00×	1.00×	1.00×
WTConv, **fused**	`fp32`	0.55×	0.43×	0.44×	0.45×	0.45×
	`fp16`	0.55×	0.44×	0.45×	0.45×	0.45×

Training step. Table 4 reports peak allocated memory over a training step as a fraction of what the reference WTConv allocates. The fused layer needs 0.43–0.55× the memory of the reference (a reduction by a factor of 1.83–2.31), and the reduction is essentially flat in L beyond L=2, because the dominant saved allocation is the level-1 coefficient tensor: it holds N elements, three times as many as all deeper levels combined ($\sum_{\ell\geq 2} N_\ell \to N/3$). This behavior follows from the mechanism described in § 4: every per-level subband tensor the reference writes to HBM is also retained by autograd until the backward pass, whereas the fused layer recomputes the Haar coefficients on chip. Against the plain convolution, the fused layer's training-step footprint is 0.87–1.02× that of the depthwise 7×7 convolution in `fp32`, and lower still in `fp16` (0.76–0.89×). The fused L-level layer therefore has a training footprint within a few percent of the memory budget of a single plain convolution and falls below it in half precision.

Table 5: Peak allocated memory for inference, as a fraction of what the reference WTConv allocates (lower is better)

Method	Precision	Decomposition levels L				
		1	2	3	4	5
Depthwise conv, k=5	`fp32`	0.42×	0.29×	0.29×	0.30×	0.30×
	`fp16`	0.65×	0.45×	0.46×	0.47×	0.47×
Depthwise conv, k=7	`fp32`	0.56×	0.39×	0.40×	0.40×	0.40×
	`fp16`	0.66×	0.46×	0.47×	0.47×	0.47×
WTConv, reference	`fp32`	1.00×	1.00×	1.00×	1.00×	1.00×
	`fp16`	1.00×	1.00×	1.00×	1.00×	1.00×
WTConv, **fused**	`fp32`	0.65×	0.50×	0.52×	0.53×	0.53×
	`fp16`	0.65×	0.51×	0.53×	0.54×	0.54×

Inference. Table 5 is the forward-only counterpart, with no autograd tape. The fused/reference footprint ratio is 0.50–0.65× during inference, versus 0.43–0.55× during training. The smaller reduction at inference is expected because a large part of what fusion removes is tape-retained subband tensors that inference never allocates in the first place; what remains is the traffic-side reduction of Eq. 10. The plain convolutions are correspondingly harder to match, since without a tape their footprint is close to the compulsory input plus output: the fused layer sits at 1.15–1.34× the depthwise 7×7 convolution's footprint in `fp32` and 0.99–1.15× in `fp16`, the residual being the filtered subband coefficients it must materialize between the analysis and synthesis passes.

5.4 Generality across kernel size

Table 6: The fused layer compared with the reference WTConv, both at k=3. Latency is reported as speedup relative to the reference (1.00×; higher is faster), and peak allocated memory as a fraction of the reference (1.00×; lower is better).

Quantity	Precision	Decomposition levels L				
		1	2	3	4	5
Training-step speedup	`fp32`	3.86×	4.50×	4.63×	4.67×	4.68×
	`fp16`	3.31×	3.82×	3.93×	3.94×	3.94×
Inference speedup	`fp32`	4.48×	5.06×	5.24×	5.28×	5.32×
	`fp16`	4.40×	4.81×	4.84×	4.82×	4.77×
Peak memory, training	`fp32`	0.54×	0.43×	0.44×	0.44×	0.44×
	`fp16`	0.54×	0.43×	0.44×	0.44×	0.44×
Peak memory, inference	`fp32`	0.64×	0.49×	0.52×	0.52×	0.52×
	`fp16`	0.64×	0.49×	0.52×	0.52×	0.52×

Every measurement above runs at k=5, the kernel size WTConvNeXt deploys. The modeled tensor-materialization traffic, however, is independent of k: Eq. 2 and Eq. 10 are both independent of k, which sets how much arithmetic each resident element receives rather than how many elements cross the HBM boundary. Table 6 tests that claim by repeating the sweep of § 5.1 at k=3.

The speedup over the reference is larger at k=3 than at k=5 in every cell of the table. A training step is 3.86–4.68× faster than the reference in `fp32` and 3.31–3.94× in `fp16`, against 3.71–4.35× and 2.68–3.09× at k=5; inference is 4.48–5.32× and 4.40–4.84×, against 3.67–4.37× and 3.38–3.60×. § 3 predicts this direction. Shrinking the kernel removes arithmetic while leaving Eq. 2 untouched, so the reference falls further into the memory-bound regime.

5.5 End-to-end networks

Table 7: End-to-end network throughput and peak allocated GPU memory, for a forward pass and for a full training step (forward and backward).

		ConvNeXt-T	WTConvNeXt-T		Fused /
		(depthwise)	reference	fused	reference
Inference	Throughput (img/s) ↑	1394.9	420.7 (0.30×)	988.5 (0.71×)	2.35×
	Peak memory (MiB) ↓	596.8	887.4 (1.49×)	750.8 (1.26×)	0.85×
Training step	Throughput (img/s) ↑	247.7	166.3 (0.67×)	263.0 (1.06×)	1.58×
	Peak memory (MiB) ↓	6933.2	9685.6 (1.40×)	7618.4 (1.10×)	0.79×

To evaluate the operator at architectural scale, we benchmark ConvNeXt-T and WTConvNeXt-T (Finder et al., 2024), which replaces ConvNeXt-T's depthwise convolutions with WTConv (k=5, $L \in \{5, 4, 3, 2\}$). We use a batch size of 64, 224 × 224 inputs, and `fp32`; Table 7 reports the results. Because the reformulation preserves the mathematical operator up to floating-point evaluation order, it does not alter the model architecture or learned parameters.

The reference WTConv imposes a severe bottleneck, restricting the network to 30% of ConvNeXt-T's inference throughput and 67% of its training throughput. Our fused implementation effectively mitigates this, achieving 2.35× and 1.58× the reference's inference and training throughputs, respectively. Crucially, WTConvNeXt-T equipped with the fused layer trains 1.06× faster than the baseline ConvNeXt-T, although inference throughput reaches only 0.71× of the baseline.

The fused formulation similarly reduces peak memory, cutting the reference WTConvNeXt-T's training and inference footprints to 0.79× and 0.85×. This limits the network's peak memory to 1.10× that of ConvNeXt-T during training and 1.26× during inference.

5.6 Ablation of the reformulations

We evaluate the three reformulations cumulatively, in their order in § 4. The first variant fuses Haar analysis, depthwise convolution, and scaling at each level. The second also replaces the sequential synthesis loop with the bit-indexed pass of Proposition 2. The full method further folds the base-path scale and fuses its addition into the final store. Each variant otherwise follows the reference `WTConv2d` implementation and is verified against its outputs and parameter gradients (Appendix C). We measure full training steps at k=5 in `fp32`, using the (C, H) sweep of § 5.1. The latency and peak-memory results are reported in Table 8 and Table 9, respectively.

Table 8: Training-step speedup for the cumulative ablation at k=5, relative to the reference WTConv (1.00×; higher is better).

Method	Precision	Decomposition levels L				
		1	2	3	4	5
WTConv, reference	`fp32`	1.00×	1.00×	1.00×	1.00×	1.00×
+ Fused Haar analysis (§ 4.1)	`fp32`	1.35×	1.44×	1.46×	1.45×	1.44×
+ Collapsed synthesis (§ 4.2)	`fp32`	2.54×	2.95×	3.02×	3.04×	3.05×
+ Scale folding (§ 4.3) = Full (**Ours**)	`fp32`	3.71×	4.23×	4.34×	4.35×	4.33×

Table 9: Peak training-step memory for the cumulative ablation at k=5, relative to the reference WTConv (1.00×; lower is better).

Method	Precision	Decomposition levels L				
		1	2	3	4	5
WTConv, reference	`fp32`	1.00×	1.00×	1.00×	1.00×	1.00×
+ Fused Haar analysis (§ 4.1)	`fp32`	0.79×	0.60×	0.59×	0.59×	0.59×
+ Collapsed synthesis (§ 4.2)	`fp32`	0.65×	0.49×	0.50×	0.50×	0.50×
+ Scale folding (§ 4.3) = Full (**Ours**)	`fp32`	0.55×	0.43×	0.44×	0.45×	0.45×

5.7 Generality across hardware

All preceding layer measurements use an RTX A6000 (Ampere, sm_86). To assess device dependence, Table 10 repeats the k=5 sweep on an NVIDIA RTX PRO 6000 Blackwell Max-Q Workstation Edition (sm_120).

The fused implementation remains faster for all L and both precisions. Relative to the reference, training is 2.46–2.79× faster in `fp32` and 1.95–2.42× faster in `fp16`; inference is 2.87–3.05× and 2.92–3.10× faster, respectively. The realized speedup is device-dependent even though both implementations remain well below the device's roofline ridge point. Differences in achieved bandwidth, kernel dispatch, launch overhead, cache behavior, and kernel utilization are not captured by the tensor-traffic model. Peak-memory reductions are stable across devices, since they are governed mainly by tensor materialization rather than GPU throughput, matching the A6000 trends at 0.40–0.55× for training and 0.50–0.65× for inference.

Table 10: The fused layer compared with the reference WTConv, both measured on the second GPU (Section 5.7), k=5. Latency is reported as speedup relative to the reference (1.00×; higher is faster), and peak allocated memory as a fraction of the reference (1.00×; lower is better).

Quantity	Precision	Decomposition levels L				
		1	2	3	4	5
Training-step speedup	`fp32`	2.49×	2.53×	2.79×	2.46×	2.48×
	`fp16`	1.95×	2.07×	2.07×	2.41×	2.42×
Inference speedup	`fp32`	2.87×	3.01×	3.03×	3.02×	3.05×
	`fp16`	2.92×	3.07×	3.10×	3.10×	3.07×
Peak memory, training	`fp32`	0.55×	0.43×	0.44×	0.45×	0.45×
	`fp16`	0.49×	0.40×	0.41×	0.41×	0.41×
Peak memory, inference	`fp32`	0.65×	0.50×	0.52×	0.53×	0.53×
	`fp16`	0.65×	0.51×	0.53×	0.54×	0.54×

6 Limitations

Our reformulation relies on properties specific to the Haar wavelet, including its low-arithmetic-cost, symmetric, and self-inverse transform structure. In particular, the fused analysis and bit-indexed closed-form synthesis do not directly generalize to wavelet families with longer filters or more complex reconstruction rules. Extending the approach beyond Haar would therefore require new transform-specific formulations.

7 Conclusion

WTConv's performance bottleneck is not arithmetic but data movement. Although the operator performs only moderately more computation than the depthwise convolution it replaces, its reference implementation repeatedly materializes intermediate wavelet coefficients and reconstructions in HBM, leaving it deeply memory-bound. By making this I/O cost explicit, we derived an algebraically equivalent formulation that keeps Haar analysis on chip, collapses the multi-level synthesis recursion into a single bit-indexed pass, and folds learned scales into the convolution weights.

These reformulations reduce modeled HBM traffic by approximately 2.55× and translate directly into substantial practical gains. Across the evaluated configurations, the fused implementation is 3.71–4.35× faster than the reference in `fp32` and 2.68–3.09× faster in `fp16` over a full training step, while reducing peak memory by a factor of 1.83–2.31. More importantly, the optimization changes the practical trade-off that motivates WTConv: the fused layer trains 1.27–1.50× faster in `fp32` and 1.40–1.76× faster in `fp16` than the depthwise 7×7 convolution it is intended to replace.

The broader lesson is that favorable FLOP counts and parameter scaling do not by themselves imply an efficient operator. For structured, multi-stage layers built from inexpensive transforms, intermediate tensor materialization can dominate execution cost. In such settings, I/O-aware algebraic reformulation is not merely an implementation optimization; it can determine whether the theoretical advantages of an operator translate into practical gains.

Broader Impact Statement

This work reduces the time and memory required to evaluate an existing operator without changing its function; the primary effect is lower computational cost for practitioners already using WTConv. We do not identify application-specific ethical risks beyond the general risks associated with making computer-vision systems more computationally efficient.

References

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, pp. 929–947, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366. 4

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pp. 578–594, 2018. 4

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024. 3

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. 3

Xiaohan Ding, Xiangyu Zhang, Jungong Han, and Guiguang Ding. Scaling up your kernels to 31× 31: Revisiting large kernel design in cnns. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11953–11965. IEEE, 2022. 1

Shahaf E Finder, Roy Amoyal, Eran Treister, and Oren Freifeld. Wavelet convolutions for large receptive fields. In *European conference on computer vision*, pp. 363–380. Springer, 2024. 1, 3, 10

Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021. 3

Shiwei Liu, Tianlong Chen, Xiaohan Chen, Xuxi Chen, Qiao Xiao, Boqian Wu, Tommi Kärkkäinen, Mykola Pechenizkiy, Decebal Constantin Mocanu, and Zhangyang Wang. More convnets in the 2020s: Scaling up kernels beyond 51x51 using sparsity. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=bXNl-myZkJl. 1

Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *2022 IEEE/CVF conference on computer vision and pattern recognition (CVPR)*, pp. 11966–11976. IEEE, 2022. 1, 7

Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. Understanding the effective receptive field in deep convolutional neural networks. *Advances in neural information processing systems*, 29, 2016. 1

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013. 4

Amit Sabne. Xla : Compiling machine learning for peak performance, 2020. 4

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF conference on computer vision and pattern recognition*, pp. 4510–4520. Ieee, 2018. 1

Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound gpu applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 191–202. IEEE, 2014. 3

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. 3

A Why the bit-indexed Haar synthesis formula is correct

We first recall what a single Haar synthesis step does. Given the four coefficients $(c_{\mathrm{LL}}, c_{\mathrm{LH}}, c_{\mathrm{HL}}, c_{\mathrm{HH}})$ of one 2×2 block, the value reconstructed at within-block position $(r_y, r_x) \in \{0,1\}^2$ is

$$\frac{1}{2}\left(c_{\mathrm{LL}} + (-1)^{r_y} c_{\mathrm{LH}} + (-1)^{r_x} c_{\mathrm{HL}} + (-1)^{r_y + r_x} c_{\mathrm{HH}}\right). \tag{12}$$

This is simply the corresponding row of the Haar synthesis matrix $\mathbf{H}^\top = \mathbf{H}$. The sign pattern is easier to see explicitly:

(r_y, r_x)	position	c_{LH}	c_{HL}	c_{HH}
$(0,0)$	top left	$+$	$+$	$+$
$(0,1)$	top right	$+$	$-$	$-$
$(1,0)$	bottom left	$-$	$+$	$-$
$(1,1)$	bottom right	$-$	$-$	$+$

(13)

We now apply this observation to level ℓ. A coefficient at this level reconstructs a $2^\ell \times 2^\ell$ block of the final image. The coefficient tuple that affects output pixel (y, x) is therefore the one at coarse-grid location

$$\left(\left\lfloor \frac{y}{2^\ell} \right\rfloor, \left\lfloor \frac{x}{2^\ell} \right\rfloor\right). \tag{14}$$

Within that block, the bit

$$b_y^{(\ell)} = \left\lfloor \frac{y}{2^{\ell-1}} \right\rfloor \bmod 2 \tag{15}$$

tells whether y lies in the top half ($b_y^{(\ell)} = 0$) or bottom half ($b_y^{(\ell)} = 1$). Similarly, $b_x^{(\ell)} = \lfloor x/2^{\ell-1} \rfloor \bmod 2$ selects the left or right half. Hence the signs in Eq. 12 are exactly

$$(-1)^{b_y^{(\ell)}} = s_y^{(\ell)}, \qquad (-1)^{b_x^{(\ell)}} = s_x^{(\ell)}, \qquad (-1)^{b_y^{(\ell)} + b_x^{(\ell)}} = s_y^{(\ell)} s_x^{(\ell)}. \tag{16}$$

In other words, dividing by $2^{\ell-1}$ discards the lower coordinate bits, and reducing modulo two extracts precisely the bit that selects the quadrant in Eq. 13.

It remains to explain the factor $2^{-\ell}$. The four level-ℓ bands first undergo the synthesis step in Eq. 12, which contributes a factor $1/2$. Their result then travels through the LL input of the remaining $\ell - 1$ finer synthesis steps. In an LL-only step, all four children receive the parent value with positive sign and another factor $1/2$. There are therefore ℓ factors of $1/2$ in total:

$$\underbrace{\frac{1}{2} \cdots \frac{1}{2}}_{\ell \text{ times}} = 2^{-\ell}. \tag{17}$$

For example, a level-1 contribution uses the lowest bits ($y \bmod 2, x \bmod 2$) and is weighted by $1/2$. A level-2 contribution uses the next bits ($\lfloor y/2 \rfloor \bmod 2, \lfloor x/2 \rfloor \bmod 2$) and is weighted by $1/4$: its bands are combined once, and the result passes through one additional LL-only synthesis step.

Finally, Haar synthesis and the recursive LL additions are linear. We may therefore compute the contribution of each level independently and add the results. Substituting the address in Eq. 14, the signs in Eq. 16, and the normalization in Eq. 17, then summing over $\ell = 1, \ldots, L$, gives Eq. 7.

B Empirical validation of the I/O model

Equations Eq. 2 and Eq. 10 predict HBM traffic by assuming that each implementation stage reads its input once and writes its output once. We evaluate this assumption using hardware performance counters. The fused implementation closely follows the model, whereas the reference matches it only when the library-selected kernels follow the assumed access pattern.

Method. We profile one forward pass with Nsight Compute on an RTX A6000. We sum `dram__bytes_read.sum` and `dram__bytes_write.sum` over all kernels and divide by the element size and N, yielding the normalized traffic reported in Table 1. We warm up each layer before profiling to stabilize cuDNN algorithm selection and exclude allocator and extension-initialization overheads. Unless noted otherwise, we use `fp32`, B=8, and k=5.

B.1 Agreement under the modeled access pattern

For C=128 and H=W=128 (N = 16.8M), the selected kernels follow the access pattern assumed in § 3. The measured and predicted traffic are compared in Table 11.

Table 11: Predicted and measured HBM traffic, normalized by $N = B \cdot C \cdot H \cdot W$, for B=8, C=128, H=W=128, k=5, and `fp32`.

		Decomposition levels L				
		1	2	3	4	5
Reference, Q_{ref}	Eq. 2	$17.75N$	$20.44N$	$21.11N$	$21.28N$	$21.32N$
	measured	$17.52N$	$20.83N$	$21.79N$	$21.97N$	$22.13N$
	measured/predicted	0.99×	1.02×	1.03×	1.03×	1.04×
Fused, Q_{fused}	Eq. 10	$7.00N$	$8.00N$	$8.25N$	$8.31N$	$8.33N$
	measured	$6.96N$	$7.94N$	$8.18N$	$8.24N$	$8.25N$
	measured/predicted	0.99×	0.99×	0.99×	0.99×	0.99×
Traffic reduction	Eq. 11	2.54×	2.55×	2.56×	2.56×	2.56×
	measured	2.52×	2.62×	2.66×	2.67×	2.68×

Across all levels, the fused measurements are within 1% of Eq. 10, and the reference measurements are within 4% of Eq. 2. Consequently, the measured traffic reduction of 2.52–2.68× closely agrees with the predicted 2.54–2.56×.

B.2 Effect of library kernel selection

The reference implementation depends on library kernel selection. Changing only the shape to C=64 and H=W=256 causes its measured traffic to exceed Eq. 2 by up to 39% (Table 12), while the fused implementation remains within 1% of Eq. 10.

Table 12: Measured HBM traffic and its ratio to the prediction for B=8, C=64, H=W=256, k=5, and `fp32`.

		Decomposition levels L				
		1	2	3	4	5
Reference	measured	$17.98N$	$27.97N$	$28.80N$	$28.82N$	$29.69N$
	vs. Eq. 2	1.01×	1.37×	1.36×	1.35×	1.39×
Fused	measured	$6.98N$	$7.95N$	$8.19N$	$8.22N$	$8.23N$
	vs. Eq. 10	1.00×	0.99×	0.99×	0.99×	0.99×

The excess originates from the grouped `conv_transpose2d` used for Haar synthesis. From L=2 onward, cuDNN selects `dgrad2d_alg1_1` for the shallow levels. At L=2, this kernel reads $0.54N$ elements but writes $6.80N$, because it repeatedly accumulates into an output tensor that exceeds the L2 capacity. The discontinuity between L=1 and L=2, and its dependence on tensor shape, identify kernel dispatch rather than the WTConv algorithm as the source of the additional traffic.

Thus, Eq. 2 should be interpreted as the algorithmic traffic of the reference implementation, and as a lower bound when library kernels perform additional accesses. For this shape, the measured reduction is 2.58–3.61×, compared with the predicted 2.54–2.56×. The model therefore gives a conservative estimate of the practical reduction. In contrast, the fused implementation uses kernels with explicit access patterns and remains close to the prediction across the evaluated configurations.

B.3 Dependence on kernel size and tensor shape

Both models are independent of kernel size k and, after normalization by N, of tensor shape. Table 13 evaluates these predictions.

Table 13: Measured HBM traffic at L=3 in `fp32` across tensor shapes (left) and kernel sizes (right). The predicted reference and fused traffic are $21.11N$ and $8.25N$, respectively.

	Tensor shape, k=5			Kernel size, C=64, H=W=256		
	C=32 H=512	C=64 H=256	C=128 H=128	k=3	k=5	k=7
Reference	$26.97N$	$29.88N$	$21.78N$	$29.61N$	$29.88N$	$28.59N$
Fused	$8.60N$	$8.17N$	$8.18N$	$8.19N$	$8.17N$	$8.19N$

Traffic shows no systematic dependence on k: over $k \in \{3, 5, 7\}$, the fused measurements vary by $0.02N$ and the reference measurements by approximately $1N$. Across shapes whose element counts differ by 4×, fused traffic remains within 5% of the prediction. The larger variation in the reference measurements ($21.78N$–$29.88N$) is consistent with the dispatch effect described in § B.2.

C Supplementary measurements

C.1 Absolute layer timings

Table 14 and Table 15 report the absolute latencies underlying the ratios in Table 2, Table 3, and Table 6. Each entry is the geometric mean over the nine (C, H) configurations in § 5.1 of the mean latency from 50 timed iterations. Consequently, before the displayed values are rounded, dividing a reference entry by another entry reproduces the corresponding geometric-mean speedup in the main paper. Plain convolutions do not depend on L, so their measured latency is repeated across the five columns.

Table 14: Absolute latency in milliseconds for the k=5 sweep. WTConv uses k=5 throughout. The depthwise k=5 row is the matched-kernel baseline, and the depthwise k=7 row is the ConvNeXt drop-in baseline.

Method	Precision	Decomposition levels L				
		1	2	3	4	5
Forward pass						
Depthwise conv, k=5	`fp32`	1.154	1.154	1.154	1.154	1.154
	`fp16`	0.479	0.479	0.479	0.479	0.479
Depthwise conv, k=7	`fp32`	2.237	2.237	2.237	2.237	2.237
	`fp16`	2.291	2.291	2.291	2.291	2.291
WTConv, reference	`fp32`	8.487	10.710	11.387	11.652	11.815
	`fp16`	4.689	5.723	6.004	6.104	6.174
WTConv, **fused**	`fp32`	2.312	2.552	2.634	2.681	2.703
	`fp16`	1.389	1.592	1.668	1.710	1.748
Training step (forward and backward)						
Depthwise conv, k=5	`fp32`	5.913	5.913	5.913	5.913	5.913
	`fp16`	2.289	2.289	2.289	2.289	2.289
Depthwise conv, k=7	`fp32`	11.737	11.737	11.737	11.737	11.737
	`fp16`	9.417	9.417	9.417	9.417	9.417
WTConv, reference	`fp32`	28.901	36.738	39.036	39.854	40.163
	`fp16`	14.362	18.592	19.842	20.366	20.758
WTConv, **fused**	`fp32`	7.800	8.693	8.995	9.154	9.277
	`fp16`	5.365	6.148	6.442	6.600	6.733

Table 16 quantifies the numerical agreement referenced in § 5.1.

Table 15: Absolute latency in milliseconds for the reference and fused WTConv implementations in the k=3 sweep.

Method	Precision	Decomposition levels L				
		1	2	3	4	5
Forward pass						
WTConv, reference	`fp32`	7.474	9.506	10.198	10.449	10.651
	`fp16`	4.529	5.552	5.834	5.940	5.999
WTConv, **fused**	`fp32`	1.667	1.880	1.945	1.979	2.001
	`fp16`	1.029	1.155	1.205	1.234	1.258
Training step (forward and backward)						
WTConv, reference	`fp32`	23.462	31.003	33.036	33.845	34.315
	`fp16`	12.955	17.093	18.310	18.821	19.186
WTConv, **fused**	`fp32`	6.081	6.891	7.135	7.248	7.332
	`fp16`	3.908	4.470	4.665	4.773	4.870

Table 16: Numerical agreement with the reference implementation. Entries are maximum absolute deviation over the whole tensor, at L decomposition levels, for the forward output and layer gradient. The fused kernels compute the same multilinear map with a different association order, so the residual is floating-point reassociation error alone.

Backend	dtype	L	forward	gradients
Fused (CUDA)	`fp32`	1	2.4e-07	2.4e-07
Fused (CUDA)	`fp32`	2	2.4e-07	2.4e-07
Fused (CUDA)	`fp32`	3	2.4e-07	2.4e-07
Fused (CUDA)	`fp32`	4	2.4e-07	2.4e-07
Fused (CUDA)	`fp32`	5	2.4e-07	2.4e-07
Fused (CUDA)	`fp16`	1	2.0e-03	9.8e-04
Fused (CUDA)	`fp16`	2	2.0e-03	2.0e-03
Fused (CUDA)	`fp16`	3	2.0e-03	2.0e-03
Fused (CUDA)	`fp16`	4	2.0e-03	2.0e-03
Fused (CUDA)	`fp16`	5	2.0e-03	2.0e-03