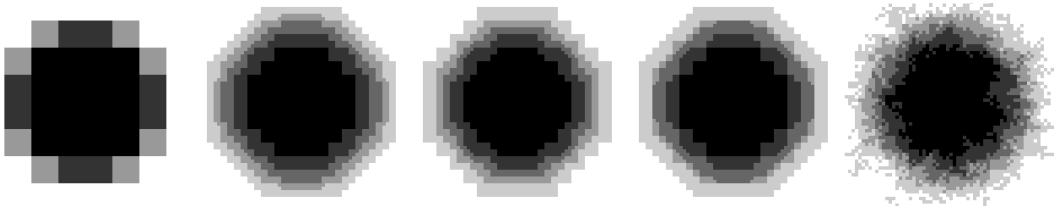
Better Fixed-Point Filtering with Averaging Trees

ANDREW ADAMS, Adobe Research
DILLON SHARLET, Google



Method:	Averaging tree	Round ties up	Round ties to even	Dithering
x86 Gpix/sec:	16.9	10.2	10.1	9.7
ARM Gpix/sec:	30.5	19.3	10.8	12.6

Fig. 1. Using trees of averaging instructions for fixed-point filtering is faster than existing methods while avoiding some of their drawbacks. Here we bilinearly upsample the small antialised circle on the left by a factor of two, three times, which involves repeated use of a fixed-point [1 3 3 9] kernel. This computational pattern shows up in pyramid-based algorithms. The circle is five intensity levels darker than the background, with the output rescaled for display. From left to right in order of cost: *Input*, upsampled using nearest neighbor; *Our method*, which chains together sequences of ubiquitous but seldom-used averaging instructions to produce an unbiased result; *Round ties up*, which is common practice when performance is critical, but can cause color artifacts in shadows as it adds a uniform bias to all color channels; *Round ties to even*, which is unbiased but accentuates banding artifacts because the even-valued bands are wider than the odd-valued bands; *Dithering*, which is excellent if done once at the end of an imaging pipeline, but is a poor choice for intermediate stages, as it introduces false structure. The throughput of each method was measured on a single core of an Intel i9-9960X and an Apple M1 Max. For this filter, our method is 60-70% faster than the next-fastest alternative.

Production imaging pipelines commonly operate using fixed-point arithmetic, and within these pipelines a core primitive is convolution by small filters – taking convex combinations of fixed-point values in order to resample, interpolate, or denoise. We describe a new way to compute unbiased convex combinations of fixed-point values using sequences of averaging instructions, which exist on all popular CPU and DSP architectures but are seldom used. For a variety of popular kernels, our averaging trees have higher performance and higher quality than existing standard practice.

CCS Concepts: • **Computing methodologies → Image processing**.

Additional Key Words and Phrases: fixed-point arithmetic, image filtering

Authors' addresses: Andrew Adams, Adobe Research, andrew.b.adams@gmail.com; Dillon Sharlet, Google, dsharlet@gmail.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2022 Copyright held by the owner/author(s).
2577-6193/2022/7-ART
https://doi.org/10.1145/3543869

ACM Reference Format:
Andrew Adams and Dillon Sharlet. 2022. Better Fixed-Point Filtering with Averaging Trees. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 3 (July 2022), 8 pages. https://doi.org/10.1145/3543869

1 FIXED-POINT FILTERING

Most photographs and videos are produced by imaging pipelines that operate using fixed-point arithmetic. Indeed, in a raw camera pipeline, nearly everything from demosaicking the Bayer sensor data to compressing the final JPEG is done using fixed-point math (e.g. [Hasinoff et al. 2016] [Adobe 2021]). Within these pipelines, perhaps the most common operation is filtering — taking a weighted combination of pixel values.

When implementing such a filtering operation with fixed-point coefficients, the standard approach is to use integer multiply-add instructions, which grows the range of the intermediate result by the sum of the coefficients. This requires either using larger data types for the computations that follow to accommodate the added range, or scaling (or normalizing) the data back down to the original range. Coefficients are often chosen to sum to a power of 2, so that this normalization can be performed using a right shift instruction instead of division.[1] This scaling operation requires some form of rounding, which can introduce bias (the image may become brighter or darker on average) and error (the result of the operation is not exact).[2] The optimal bias is zero, and the smallest peak error we can expect from any operation rounding to integers is 1/2.

The simplest rounding operation is truncating the remainder, which has a consistent bias towards zero, and an error approaching one. A widely-used superior alternative is rounding to the nearest integer with ties rounding up. This operation is commonly supported by instruction sets intended for signal processing, such as the rshrn ARM NEON instruction. This has less bias than simple truncation, but it still has a bias of +1/2 in the range prior to normalization, because ties are always rounded up. Rounding ties to nearest even avoids the bias of the simple rounding, but produces an uneven distribution of values, and requires more instructions to implement on most architectures. Dithering is a more complex scheme where noise with a mean of 1/2 in the post-normalization range is added to the data prior to truncation. Generating suitable blue noise for dithering is non-trivial (see [Ulichney 1988]), requires more instructions to implement on most architectures, and occupies some portion of the cache hierarchy with a noise table.

2 AVERAGING TREES

All currently-popular CPU and DSP instruction sets include integer averaging instructions. These come in a rounding-down variant that computes $\lfloor \frac{a+b}{2} \rfloor$, and a rounding-up variant that computes $\lceil \frac{a+b}{2} \rceil$. Both variants exist on ARM[3], Hexagon[4], and most DSP architectures. X86 includes the rounding-up variant only[5], though the rounding-down variant can be emulated with four instructions [Dietz 2021].

For any kernel with coefficients that sum to a power of two, it is straightforward to construct an averaging tree that implements normalized convolution by this kernel. For a kernel that sums to 2^n, construct a balanced binary tree of averaging operations of depth n, and assign each input to a number of leaf nodes corresponding to its coefficient. After collapsing all inner nodes that average a value with itself and deduplicating identical sub-trees, these trees will typically use fewer

[1] A widely-used exemplar of this approach can be found in libjpeg-turbo[2021].
[2] Formally, we define bias as the mean difference between the rounded result and the unrounded real-valued result over all possible inputs, and peak error as the maximum *absolute* difference.
[3] `hadd`, `rhadd` instructions
[4] `vavg`, `vavg:rnd` instructions [Qualcomm 2018]
[5] `pavgb`, `pavgw` instructions

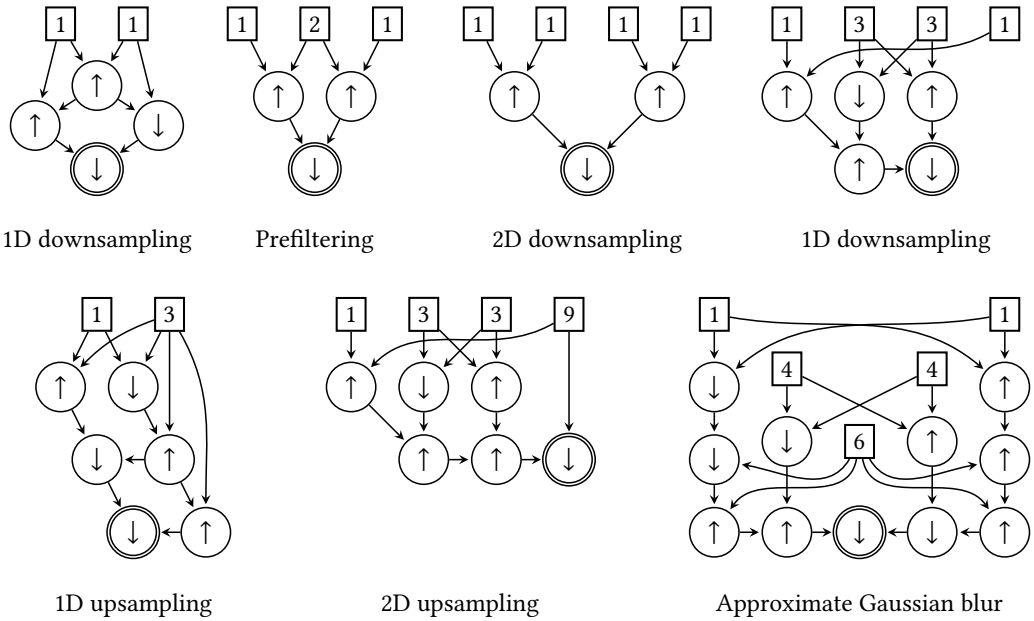


Fig. 2. Some averaging trees that implement common filters used for various tasks in imaging. The trees have been simplified into their minimal graph form (Fig. 4). Each rectangle corresponds to an input tap with the given kernel coefficient. Circles represent fixed-point averaging operations, with the rounding direction given by the arrow within. On most platforms each of these is a single machine instruction. The doubled circle is the output. Using these sequences of averaging operations for the tasks listed is typically superior to existing standard practice in terms of speed, bias, and error (see Table 1). All trees pictured have zero bias and minimum error.

instructions than other rounding modes (Table 1). Unfortunately, there is a combinatorially-large number of such trees, and most of them have worse bias and error than other rounding modes.

Surprisingly, for many frequently-used kernels (Figure 2), trees exist that have the minimum peak absolute error of $1/2$ and zero bias. All such trees we have found are simultaneously the fastest and the highest-quality way to perform these filters, and should be used in every case.

For other kernels, we cannot provide averaging trees with minimum peak error, but we can still provide kernels with zero bias, which is more important in some applications. Bias often manifests as low frequency color artifacts (Figure 3), while a little extra error is usually not visible.

2.1 Averaging Tree Generation

We generate the averaging trees shown in this paper in two ways. These algorithms are run once, offline.

Algorithm 1. To construct an averaging tree for a given kernel, place each input at the leaves of a balanced binary tree a number of times corresponding to its coefficient. The inner nodes of the tree represent averaging operations. There are a combinatorially-large number of ways to choose which leaves correspond to which inputs, and what rounding direction to use for the averaging operations. For each of these, exhaustively measure bias and error over all possible n-bit inputs for maximum tree depth n. If no optimal tree is found, double all coefficients in the kernel, which

Fig. 3. Using averaging trees improves performance and reduces color artifacts caused by bias. On the top left we show a photograph from Google's HDR+ camera pipeline [Hasinoff et al. 2016] used in Pixel phones. On the bottom left is the pipeline's output cropped to the indicated region and brightened. Chrominance noise is biased towards purple, and low-frequency purplish blobs are present on these ideally-gray granite boulders. Bias in filtering operations translates into chrominance error because it moves the effective black level of the image, and operations that scale color channels (e.g. lens shading correction or white balance) are sensitive to the precise black level. On the bottom right is the output after replacing 59 different filtering operations with the averaging trees in this paper. On the top right is the difference between the two. Using averaging trees, chroma artifacts are reduced, and the pipeline stages affected run 11% faster.

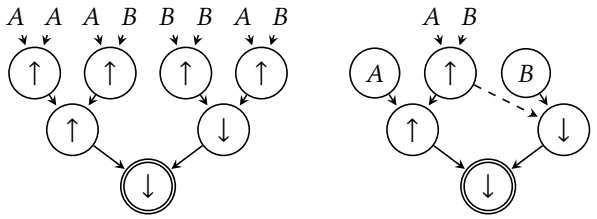


Fig. 4. The output of Algorithm 1 applied to the [1 1] kernel, with inputs [A B]. Two doublings occurred to arrive at the operation $(4A + 4B)/8$. On the right, we simplify the tree into a graph by collapsing nodes that average a value with itself (the circled letters), and deduplicating common subtrees (the dashed line). On most architectures, this graph computes an unbiased average in four instructions.

Kernel	Method	Error	Bias	ARM ops	ARM cycles	x86 ops	x86 cycles
1 1	Averaging tree	1/2	0	4	1.2	10	5.1
	Round up	1/2	1/4	1*	1.2	1*	2.7
	Round to even	1/2	0	10	2.5	18	8.2
	Dither	1/2	0	8	3.0	18	11.2
1 2 1	Averaging tree	1/2	0	3	1.5	6	4.6
	Round up	1/2	1/8	8	2.1	24	10.3
	Round to even	1/2	0	18	4.6	26	13.2
	Dither	3/4	0	12	3.7	24	13.2
1 1 1 1	Averaging tree	1/2	0	3	1.7	6	4.8
	Round up	1/2	1/8	8	2.2	26	12.1
	Round to even	1/2	0	18	4.6	28	12.2
	Dither	3/4	0	12	3.8	26	15.2
1 3 3 1	Averaging tree	1/2	0	5	1.9	12	5.8
	Round up	1/2	1/16	10	2.6	28	13.8
	Round to even	1/2	0	18	4.6	30	14.6
	Dither	7/8	0	14	4.1	28	15.2
1 3	Averaging tree	1/2	0	6	1.6	15	6.5
	Round up	1/2	1/8	6	1.7	20	10.5
	Round to even	1/2	0	16	4.0	22	11.5
	Dither	3/4	0	10	3.3	20	11.3
1 3 3 9	Averaging tree	1/2	0	6	1.7	12	5.6
	Round up	1/2	1/32	10	2.6	30	15.8
	Round to even	1/2	0	18	4.6	32	16.4
	Dither	15/16	0	14	4.1	30	16.1
1 4 6 4 1	Averaging tree	1/2	0	11	2.9	26	10.2
	Round up	1/2	1/32	12	3.6	36	17.6
	Round to even	1/2	0	22	5.2	38	16.6
	Dither	15/16	0	18	4.7	36	17.6

Table 1. Averaging trees have minimum bias, minimum peak error, and use the fewest instructions on ARM and x86 (using AVX2) for these popular kernels. Instruction counts are for computing a single native-width SIMD vector of output, and include only SIMD arithmetic instructions. Cycles are also per-SIMD vector and are measured by repeatedly filtering an array of 2^{17} 16-bit integers on a single core of an Apple M1 Max at 3.2 GHz and an Intel i9-9960X at 3 GHz for ARM and x86 respectively. The fastest method in each category is highlighted in blue. There is no average-rounding-down instruction on x86 (it must be emulated with a four-instruction sequence [Dietz 2021]), but x86 also lacks widening multiply-adds and rounding shifts, so it requires more instructions than ARM in every case. Note, however, that a SIMD vector on x86 with AVX2 is twice as wide as on ARM, so the performance difference between the platforms is not as large as it would appear. *In this case the compiler (Halide [2012]) knew to use the averaging-round-up instruction already, making this a degenerate averaging tree.

deepens the tree by one layer, and start again. Finally, simplify the generated tree into a directed graph by deduplicating identical nodes and collapsing nodes that average a value with itself. The algorithm terminates at the first optimal tree found, so we order the search so that round-ties-up averages come before round-ties-down averages, as rounding up is cheaper on x86.

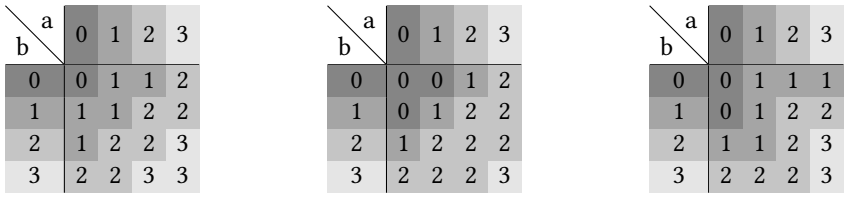


Fig. 5. All possible outputs for a [1 1] kernel, which simply averages two values a and b. See Figure 2 top left for the averaging tree used for our method. The inputs have mean 1.5, so the mean of the average should also be 1.5. Round-up has a mean output of 1.75, representing a positive bias of 0.25. Round-to-even has the correct mean output, but 0 and 2 are over-represented. Our method alternates rounding direction along the off-diagonals. While it is no longer commutative, it has the correct mean and a balanced distribution.

To give a worked example, consider a [1 2 1] filter applied to inputs A, B, C. The kernel sums to four, so we first construct a balanced binary tree with four leaves, and assign two of those leaves to B, and the other two to A and C. The three inner nodes of the tree represent averaging operations. Averaging operations are commutative, so there are only two meaningfully-different ways to assign inputs to leaf nodes in this example - either the two instances of B are together, as in $((B, B), (A, C))$ or they are not, as in $((A, B), (B, C))$. For each of these choices, we consider all combinations of rounding directions for the three inner nodes. Each node can round in two possible directions, so combined with the two choices for assignment of inputs to leaves, there are 16 trees to evaluate. For each of these we evaluate the trees on all possible two-bit values for A, B, and C, which amounts to 64 test cases per tree. The test cases are evaluated in parallel using Halide [2012], which helps scalability for larger trees. The [1 2 1] tree shown in Figure 2 is optimal, so the search terminates when it is found. If we had not found an optimal tree, we would have constructed a balanced binary tree with eight leaves, assigned four of them to B, and two each to A and C, effectively treating our [1 2 1] kernel as a [2 4 2] kernel instead. This doubling occurs twice for the [1 1] filter, as illustrated in Figure 4.

For a tree of depth n, there are $O(2^n)$ inner nodes, and $O(2^{2^n})$ possible combinations of rounding directions to consider for them. Thus, increasing n by one squares the amount of work done by our exhaustive search. At $n = 4$ (i.e. kernels that sum to 16) finding a solution may take several hours. At $n = 5$ (kernels that sum to 32) the method becomes intractable. Unfortunately, the minimum tree depth for which the common [1 4 6 4 1] filter has an unbiased form is 5. To find our solution we instead enumerated pairs of biased depth-4 [1 4 6 4 1] trees, and tested the average of each pair for bias and error. We were fortunate enough to find an optimal tree within this small subspace of all depth-5 trees. This created the symmetric structure seen in the bottom right of Figure 2.

Algorithm 2. In addition to a method for finding trees for a specific kernel, we would like a list of *which* kernels can be implemented with an averaging tree with zero bias and minimum error. This is useful in situations where there is some flexibility in which kernel is to be used. Our second, simpler method for generating averaging trees is to enumerate all averaging DAGs up to a fixed size. For each, we recursively compute which kernel it corresponds to. The kernel for an input node is a delta function, and the kernel for an averaging node is the average of the kernels of its children. We then exhaustively compute bias and peak error to find the ones with zero bias and minimum error.

For both algorithms, we compute bias and error by enumerating all possible inputs, as described above. Bias is computed as the mean difference between the real-valued unrounded result and the output of the averaging tree over this set, and error is computed as the maximum absolute difference between the unrounded result and the output.

Code for both of these methods and the list of averaging trees produced can be found in supplemental material.

3 EVALUATION

We evaluate the performance of our averaging trees on the task of computing a dense 1D filter of an array of 2^{17} 16-bit unsigned integers. This is sized to fit in L2 cache. In each case we manually count the number of arithmetic instructions in the inner loop, and also measure the number of cycles required to compute a single native-width SIMD vector of output. We compare to three conventional methods for computing these filters: rounding up, rounding to nearest even, and dithering. Results are in Table 1. Performance was measured on a single core of an Intel i9-9960X pinned to 3 GHz with hyperthreading disabled, and the performance core of an Apple M1 Max at 3.2 GHz.

The three baseline methods all use substantially more cycles and instructions than our averaging trees. The main cause is that these methods must start by widening the inputs to 32-bit unsigned integers to avoid overflow in the multiply-adds that follow. On a fixed-bit-width SIMD machine, using a wider intermediate type doubles the number of instructions required, as each instruction only handles half as many values.

We also evaluate our averaging trees with the iterated upsampling test in Figure 1. One way to bilinearly upsample an image by a factor of two is to apply all rotations of a square [1 3; 3 9] filter to each overlapping 2x2 tile of the input, to produce non-overlapping 2x2 tiles of output. Our method is the fastest, and avoids the artifacts caused by the other unbiased methods.

4 RELATED WORK

Most work in graphics pays little attention to which specific instructions are used. A few exceptions are recent fixed-point camera pipelines [Hasinoff et al. 2016], quantized simulators [Hu et al. 2021], quantized neural networks [Thomas et al. 2020], and median filters [Adams 2021], all of which demonstrate performance benefits by exploiting low-bit-width data types and instructions.

Our method performs an exhaustive search for unbiased averaging trees, making it an instance of enumerative program synthesis. Program synthesis has been used recently in graphics to translate legacy fixed-point imaging code to Halide [Ahmad et al. 2019]. The most popular approaches to program synthesis reduce the problem to a series of SAT or SMT queries and use a general purpose solver to find a program that produces the correct output. While it is straight-forward to map the problem of finding an averaging tree to a SAT query, we found that this scaled worse than our bespoke enumerative search.

When averaging trees are used in fast imaging code in the wild, they are treated as inexact relative to some ground truth rounding mode [Abel et al. 1999], or are modified with bit tricks to make them exactly match rounding ties up, intentionally introducing bias (see libwebp [2011] and libvpx [2017]). In contrast, libjpeg-turbo [2021] uses a conventional series of widening integer multiply adds to implement similar filters, rounding ties up on even x coordinates and rounding them down on odd x coordinates in a form of simple dithering. These three heavily-used libraries would be faster and produce higher quality output if they used one of our averaging trees, which do exist for the filters used.

5 CONCLUSION

Using averaging trees for fixed-point filtering is faster, less biased, has optimal worst case error, and maintains a more uniform distribution of output values than the existing standard practice. Making this seemingly minor change has meaningful effects on performance and quality. We took Google's HDR+ camera pipeline [Hasinoff et al. 2016], used in Pixel phones, and converted 59 different filtering operations to the averaging trees in this paper. This resulted in an 11% speed-up in local tone-mapping and sharpening, and visibly reduced color artifacts in shadows (Figure 3).

A challenge in using our method is that an averaging tree does not always exist for a given kernel. For now, the set of known kernels is limited to what could be found with brute-force enumeration, so trees for large kernels may be intractably difficult to find, if they even exist. Our method also requires the coefficients to sum to a power of two, and is inapplicable to kernels with negative coefficients, which are commonly used but no longer represent a *convex* combination of the inputs. Overcoming these limitations is an interesting area of future work.

Another possible issue is that our method can produce trees that produce different results when inputs corresponding to the same coefficient are permuted. For example, our unbiased averaging tree for a [1 1] kernel is not commutative, as shown in Figure 5. If this is undesirable it can be ameliorated by flipping the kernel at even/odd pixels at the cost of a few extra instructions.

Despite these limitations, and somewhat to our surprise, we find that the most commonly-used kernels do admit optimal averaging trees, and we claim that our averaging trees should be used whenever these kernels arise.

REFERENCES

James Abel, Kumar Balasubramanian, Mike Bargeron, Tom Craver, and Mike Phlipot. 1999. Applications Tuning for Streaming SIMD Extensions. https://www.intel.com/content/dam/www/public/us/en/documents/research/1999-vol03-iss-2-intel-technology-journal.pdf

Andrew Adams. 2021. Fast Median Filters Using Separable Sorting Networks. *ACM Trans. Graph.* 40, 4, Article 70 (jul 2021), 11 pages. https://doi.org/10.1145/3450626.3459773

Adobe. 2021. Adobe Camera Raw. https://www.adobe.io/camera-raw

Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically Translating Image Processing Libraries to Halide. *ACM Trans. Graph.* 38, 6, Article 204 (nov 2019), 13 pages. https://doi.org/10.1145/3355089.3356549

Somnath Bannerjee. 2011. libwebp. https://chromium.googlesource.com/webm/libwebp/+/refs/heads/main/src/dsp/upsampling_sse41.c#27

Henry Gordon Dietz. 2021. *The Aggregate Magic Algorithms.* Technical Report. University of Kentucky. http://aggregate.org/MAGIC/

Samuel W. Hasinoff, Dillon Sharlet, Ryan Geiss, Andrew Adams, Jonathan T. Barron, Florian Kainz, Jiawen Chen, and Marc Levoy. 2016. Burst photography for high dynamic range and low-light imaging on mobile cameras. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)* 35, 6 (2016).

Yuanming Hu, Jiafeng Liu, Xuanda Yang, Mingkuan Xu, Ye Kuang, Weiwei Xu, Qiang Dai, William T. Freeman, and Frédo Durand. 2021. QuanTaichi: A Compiler for Quantized Simulations. *ACM Transactions on Graphics (TOG)* 40, 4 (2021).

Scott LaVarnway. 2017. libvpx. https://chromium.googlesource.com/webm/libvpx/+/refs/heads/main/vpx_dsp/x86/highbd_intrapred_intrin_ssse3.c#29

libjpeg turbo. 2021. libjpeg-turbo. https://github.com/libjpeg-turbo/libjpeg-turbo/blob/5446ff88d617b2d2768456d9be1a8c47c4606c92/simd/arm/jdsample-neon.c#L187-L189

Qualcomm. 2018. Qualcomm hexagon V66 HVX programmer's reference manual. https://developer.qualcomm.com/downloads/qualcomm-hexagon-v66-hvx-programmer-s-reference-manual

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 1–12.

Manu Mathew Thomas, Karthik Vaidyanathan, Gabor Liktor, and Angus G. Forbes. 2020. A Reduced-Precision Network for Image Reconstruction. *ACM Trans. Graph.* 39, 6, Article 231 (nov 2020), 12 pages. https://doi.org/10.1145/3414685.3417786

R.A. Ulichney. 1988. Dithering with blue noise. *Proc. IEEE* 76, 1 (1988), 56–79. https://doi.org/10.1109/5.3288