

Fast Isotropic Median Filtering

BEN WEISS, Google Research, USA

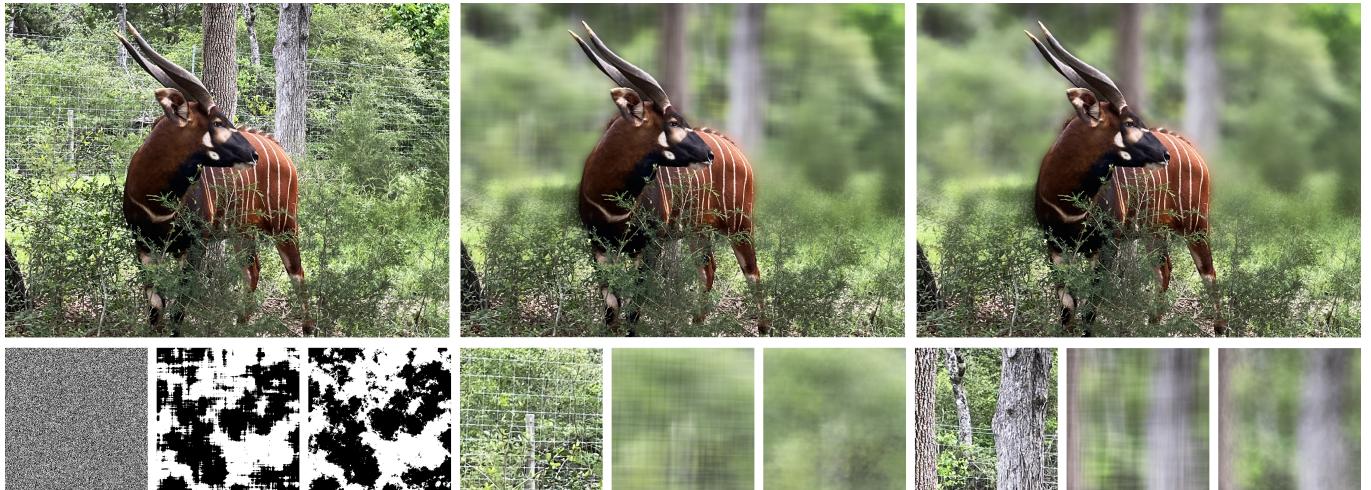


Fig. 1. Top row: Original photo, median-filtered with a radius-48 square kernel (with hand-applied matte), and with an equivalent-area circular kernel. Bottom row: square vs. circular filtered image quality for binary noise, as well as for two swatches from the image, unsharp-masked 200% to emphasize the high-frequency artifacts. Across a wide range of parameters, our circular median filter is dramatically faster and higher quality than the state of the art.

Median filtering is a cornerstone of computational image processing. It provides an effective means of image smoothing, with minimal blurring or softening of edges, invariance to monotonic transformations such as gamma adjustment, and robustness to noise and outliers. However, known algorithms have all suffered from practical limitations: the bit depth of the image data, the size of the filter kernel, or the kernel shape itself. Square-kernel implementations tend to produce streaky cross-hatching artifacts, and nearly all known efficient algorithms are in practice limited to square kernels. We present for the first time a method that overcomes all of these limitations. Our method operates efficiently on arbitrary bit-depth data, arbitrary kernel sizes, and arbitrary convex kernel shapes, including circular shapes.

CCS Concepts: • Computing methodologies → Image processing.

Additional Key Words and Phrases: computer graphics, image processing, median filtering

ACM Reference Format:

Ben Weiss. 2025. Fast Isotropic Median Filtering. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers (SIGGRAPH Conference Papers '25)*, August 10–14, 2025, Vancouver, BC, Canada. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3721238.3730763>

Authors' address: Ben Weiss, Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA, 94043, USA.

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).

SIGGRAPH Conference Papers '25, August 10–14, 2025, Vancouver, BC, Canada

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1540-2/2025/08

<https://doi.org/10.1145/3721238.3730763>

1 INTRODUCTION

The median filter [Tukey 1974] is a foundational element of classical signal processing. Extended to two dimensions [Huang et al. 1979], it allows removal of image noise and smoothing of edges, without the softening that is characteristic of e.g., Gaussian blur. Historically it has been fairly inefficient and artifact-prone; the nonlinear nature of the filter has resisted optimization attempts through vectorization techniques, and the ubiquitous square-kernel implementation produces distinct cross-hatching visual artifacts that reduce the filter's usefulness for quality-sensitive applications.

In this paper, we present an algorithm that performs efficient median filtering using a circular kernel, substantially improving image quality while attaining state-of-the-art performance for medium and large kernel sizes. Our primary contributions are as follows:

- Extension of the 1D compound histogram technique from Weiss [2006] to a more general and powerful 2D "omnigram" data structure, from which histogram elements of any ordinal image region of any shape can be obtained in constant time.
- A rank-order-filtering algorithm capable of processing circular kernel shapes, eliminating the visual artifacts characteristic of square-kernel filtering, while achieving state-of-the-art performance, even compared to the best existing square-kernel implementations.

After reviewing related work in section 2, we give an overview of our method in section 3. We then describe our implementation in detail in section 4 and provide results and comparative studies in section 5. Code and data for this paper are at:

<https://github.com/google/fast-isotropic-median-filter>.

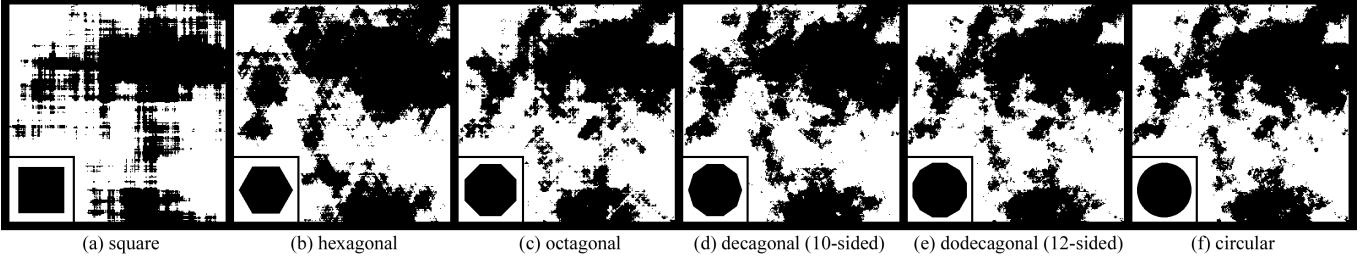


Fig. 2. Median-filtering binary noise with increasingly isotropic kernel shapes. By observing the cross-hatching artifacts introduced by filtering noise with low-order polygonal kernels, the importance of using near-isotropic kernels (ideally 12-sided or circular) to obtain optimal image quality is strikingly evident.

2 RELATED WORK

Approach	Runtime	Implementations
Sorting Networks	$\sim O(r^2)$	McGuire [2008] Adams [2021]
Sliding-Window Histograms	$O(r)$, $O(\log^k(r))$, $O(1)$, $\sim O(r)$	Huang et al. [1979] Weiss [2006] Perreault and Hébert [2007] Our method
Integral Queries	$O(1)$	Moroto and Umetani [2022]
Isotropic Approximations	$O(1)$	Kass and Solomon [2010] Yang et al. [2015]

2.1 Sorting Networks

Sorting networks were introduced by Batcher [1968], where inputs are sorted directly using a fixed sequence of min-max swaps to obtain the median. McGuire [2008] pioneered this approach for 3x3 and 5x5 kernels, and Adams [2021] extended it to larger kernels using separable sorting networks. For square kernels of fairly small size, this is the most efficient known technique. However, it becomes impractically slow for large kernels.

2.2 Sliding-Window Histograms

Other methods involve explicit histograms that are maintained and updated using sliding windows. Originated by Huang et al. [1979], this provides a canonical baseline $O(r)$ algorithm for 8-bit images. In our previous work ([Weiss 2006]) we had improved on the sliding-window technique for 8-bit images by constructing hierarchical trees of histograms, improving the runtime from $O(r)$ to $O(\log(r))$, and also extended the method to higher bit-depths in $O(\log^2(r))$ time. Perreault and Hébert [2007] employed a sliding-window technique over single-column histograms, yielding a constant-time median filtering algorithm for 8-bit data.

2.3 Integral Queries

Porikli [2005] first recognized that small (e.g. 8-bit) histograms could be spatially integrated, yielding summed-area-table-like structures from which histogram components of any rectangular region could be extracted in constant time. More recently, Moroto and Umetani [2022] extended the 1D wavelet matrix technique of Claude et al. [2015] to process arbitrary-depth 2D images, allowing efficient and arbitrary 2D rectangular rank-order queries and median filtering in constant time.

2.4 Isotropic Approximations

The characteristic cross-hatching artifacts [Fig. 2(a)] of square-kernel median filters have long been recognized, but the algorithmic costs of avoiding these artifacts have been difficult to overcome, particularly for exact filters. Thus, most current implementations (e.g. OpenCV, Adobe Photoshop) support only square-kernel median filtering. Mishra et al. [2020] described the qualitative advantages of using an approximately circular kernel to median-filter medical MRI images, although they were limited to very small kernels.

Perreault and Hébert [2007] and Moroto and Umetani [2022] described extensions to their methods that support higher-order polygonal kernels, but at a linearly increasing performance cost (and working set overhead) in the number of sides.

Kass and Solomon [2010] developed a constant-time approximate method for isotropic median filtering, by low-pass-filtering histogram data into a continuous representation, then applying a spatial Gaussian filter to achieve an isotropic weighted kernel shape. Yang et al. [2015] proposed a similar method involving coarse-granularity cost volumes, interpolated to find approximate medians. Both methods were targeted at 8-bit images; it is unclear how well they might practically scale to e.g. HDR domains.

Finally, in our earlier paper [Weiss 2006], we mentioned as a topic of future research that our 1D "compound histogram" technique might be extended to accommodate non-rectangular kernel shapes, such as circles, but did not offer a practical implementation. In this paper we show that this conceptual approach can indeed work, and that it can be implemented efficiently and powerfully, with state-of-the-art performance and quality and a small memory footprint, on both GPU and CPU architectures.

3 METHOD OVERVIEW

3.1 Core concepts: Histograms, Pivots, and Counts

In a histogram, the median value is found at the smallest index for which the cumulative histogram up to that point reaches the midpoint rank; e.g. for a 7x7 median filter, the 25th of 49 values. This can be found by scanning the histogram from its endpoints, but when the median must be extracted from two similar histograms (as for overlapping image windows), as observed by Huang et al., it is more efficient to choose a "pivot" index near the median value of the first region, for which the "count" (cumulative number of values strictly below the pivot) is known, then track how the "count" changes as the histogram is updated for the second region. The pivot

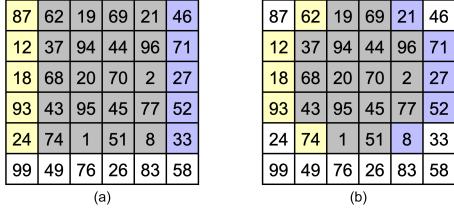


Fig. 3. Sliding-window method of Huang et al. [1979]. (a) As the top left 5x5 window slides one pixel to the right, blue-shaded pixels are added to a running histogram, and yellow-shaded pixels removed. (b) The method as adapted to a 21-tap circular kernel.

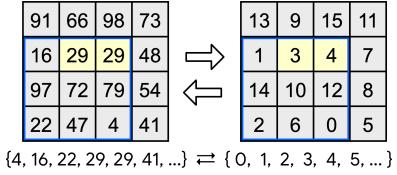


Fig. 4. Ordinal transform of Weiss [2006]. Cardinal brightness values (left) map to their respective ordinal ranks (right), creating an "ordinal image." Duplicate cardinal values (yellow) map to consecutive ordinal values. Note that the median values in e.g. the blue squares (29 and 4, respectively) map to each other, illustrating the invariance of the filter under this transform. The lower-left array is the "reverse map" that maps ordinal results back to cardinal output values.

from the first region is then used as the starting index to scan the updated histogram to find the median value for the second region.

Huang's method (Fig. 3) is linear-time in the filter radius, and works by sliding a square window across the image, following this pivot/count principle. It has the advantage of simplicity, and can be straightforwardly adapted to process non-square kernel shapes, such as circular shapes¹. Its main drawback is that its primary bottleneck, the histogram-updating step, is fundamentally scalar and memory-bound. A second drawback is that its memory requirements grow exponentially with bit-depth, and also linearly with concurrency; each thread must maintain its own separate mutable histogram.

3.2 The Ordinal Transform and Compound Histogram

In [Weiss 2006] we described an invertible "ordinal transform" between brightness-valued ("cardinal") and rank-ordered ("ordinal") images, as shown in Fig. 4, replacing duplicate cardinal values with consecutive ordinal values, to yield an "ordinal image" I , that has two particularly useful properties. First, each value in I is unique, ensuring that histograms of any subregion of it can be represented with just a single bit per element. Second, rank-order operations such as median-filtering are invariant to this transform.

We leveraged these properties to develop a "compound histogram" array, where the value at each index contains the column of that index within the ordinal image, if it is inside the rectangular input strip corresponding to the output scanline currently being processed, or a sentinel value if it is not. From this compound histogram, any binary histogram element of any rectangular subregion within that strip

¹We are not sure whether Huang et al. were aware of this.

	A	B	C	D	E	F
0	10	30	29	16	22	6
1	20	33	25	8	1	26
2	19	7	15	21	9	35
3	32	28	3	0	17	5
4	18	27	11	31	34	13
5	14	4	12	24	2	23

	Ordinal value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
Compound Histogram	d	e	-	c	-	f	f	b	d	e	a	c	-	f	-	c	...	
Omnigram (ours)	D3	E1	E5	C3	B5	F3	F0	B2	D1	E2	A0	C4	C5	F4	A5	C2	...	

Fig. 5. Comparison of the compound histogram (Weiss [2006]) and the omnigram (our method). The compound histogram (red) covers a subset of input rows, encoding the column index of each ordinal value in those rows, or a sentinel '-' for values outside those rows. With the compound histogram, membership of a given ordinal value in an arbitrary full-height rectangle within the red strip can be determined in constant time. With the omnigram (blue), the full 2D location of each ordinal value is encoded, encompassing the entire input image. With this encoding, membership in any arbitrary image region (e.g. the blue circle) can be determined in constant time.

(including every square window) can be obtained in constant time. Once the median value is found for every pixel of the corresponding output row, the compound histogram is slid down one row, with leading values added and trailing values removed, and the next row is processed.

Paired with that paper's hierarchical $O(\log(r))$ method for computing low-precision median filters, the compound histogram approach yields an $O(\log^2(r))$ method for an arbitrary-precision median filter. But it is restricted to square (or rectangular) kernel shapes, and like Huang's method, is limited by the fact that its working set grows linearly with concurrency, because the compound histogram must be updated as the windows are slid vertically. This impairs its adaptability to massively-concurrent GPU architectures, where limited fast memory is at a premium.

3.3 The Omnigram

Our method extends this "compound histogram" concept to two dimensions. Instead of each array value encoding just a column [or sentinel] value, we encode both the row and column in each element. This doubles the size of the structure relative to the compound histogram, but it has two key benefits. First, it makes the array immutable; any area of the image can be processed with it, not just a single scanline of output, and it never needs to be modified after construction. Second, it allows histogram elements of any image subregion of any shape (not just rectangular shapes) to be extracted in constant time. With these properties, it functions as a universal histogram. We refer to this as our omnigram, Ω_I .

While processing our image, for any image region R and any image value v , we want to be able to efficiently obtain the histogram entry:

$$H_R[v] = \# \text{ of occurrences of value } v \text{ in region } R \quad (1)$$

Since our ordinal image I is unique-valued, these histogram elements can only ever be 1 or 0, and our omnigram Ω_I specifies where each value is located within I . Thus, this reduces to:

$$H_R[v] = \begin{cases} 1 & \text{if } \Omega_I[v] \in R \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

And for our specific use case of a circular region centered at point P , with radius r , this becomes:

$$H_{C(P,r)}[v] = \begin{cases} 1 & \text{if } \|\Omega_I[v] - P\| \leq r \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

With these foundations, we can distill our method to its essence: we apply a circular modification of Huang's method to the ordinal-transformed image, obtaining histogram values from the immutable omnigram in lieu of maintaining and updating explicit histograms. And instead of sliding a single window at a time, we vertically slide adjacent windows for many columns of output at once. With this approach, each of the algorithm's bottlenecks becomes vectorizable and parallelizable. This enables high performance on both CPU and GPU architectures, while providing the substantial quality improvement associated with isotropic kernels.

3.4 Method Summary

At a high level, our method works as follows:

- (1) Process the image in small single-plane tiles, with input tiles padded by the filter radius on all sides with respect to the output tiles. (Output tiles may overlap vertically by one row.)
- (2) For each input tile, perform an ordinal transform [replacing each pixel with its ordinal rank in the tile], also constructing the ordinal->cardinal reverse map and the omnigram Ω_I .
- (3) For a tile at the top of the image, determine the median ordinal value m_{00} for the top left circular window W_{00} , and store a "pivot" value p_0 near m_{00} , and a "count" c_0 of values smaller than p_0 in this window.
- (4) Determine the rank of p_0 within the next horizontally adjacent window W_{01} , by comparing leading/trailing values to p_0 as the window slides one pixel to the right. Then scan Ω_I from p_0 to determine the exact median value m_{01} for the new window. Select a new pivot p_1 near m_{01} , and continue sliding horizontally this way to solve the top output row, storing the respective pivots p_k and counts c_k for all of the top-row windows.
- (5) Slide all of the top-row windows down vertically one pixel, in parallel, adjusting each c_k based on leading/trailing pixels, and using p_k as the starting point to scan Ω_I for window W_{1k} . Solve the medians for the new row, and update p_k and c_k to remain close to the median m_{1k} for window W_{1k} .
- (6) Continue this way through all the output rows in the tile. Solutions for the last row can be "forwarded" to the next tile below, since the spatial offset of a solution median pixel from its window center is invariant (for a given global output pixel) between overlapping tiles, and is directly obtainable from Ω_I . This allows the relatively expensive steps 3 and 4 to be skipped for subsequent tiles.
- (7) Invert the ordinal transform using the reverse map to yield the cardinal median-filtered result.

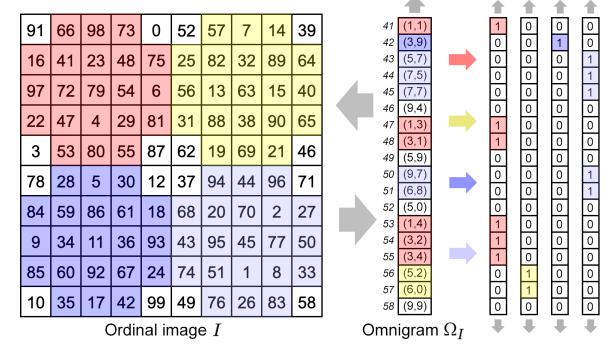


Fig. 6. Ordinal image and omnigram. Binary histogram elements (right) of any ordinal image region (tinted circles, left) can be extracted in constant time by querying the omnigram (center).

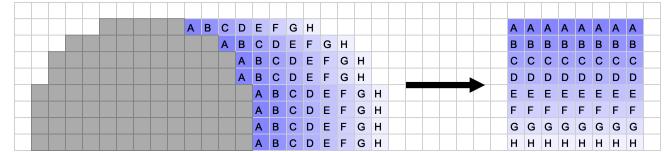


Fig. 7. Processing the first row. For sliding the circular kernel window horizontally, it is helpful to shear/transpose the first $2 * \text{radius} + 1$ rows of input pixel data to facilitate SIMD processing.

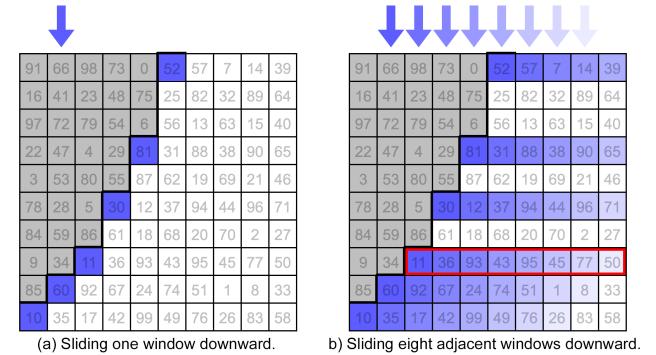


Fig. 8. Sliding vertically. (a) When a single non-rectangular window is slid downward, the leading [and trailing] pixels (blue) are scattered in memory, precluding SIMD optimization on a per-window basis. (b) When several adjacent windows are slid downward together, the leading/trailing pixels form coherent groups (e.g. outlined in red), enabling SIMD optimization.

4 IMPLEMENTATION

We process the image in fairly small tiles, limiting the input tile size to 256x256 pixels. This allows the omnigram to be encoded with 16 bits per element; 8 each for x and y.

Constructing Ω_I and the cardinal-to-ordinal reverse-map for the cardinal input tile I_c is equivalent to key-value sorting the tuples {brightness, location} for each input pixel. The sorted list of cardinal brightnesses C forms the reverse map; the corresponding list of locations comprises Ω_I . If solutions are forwarded between tiles,

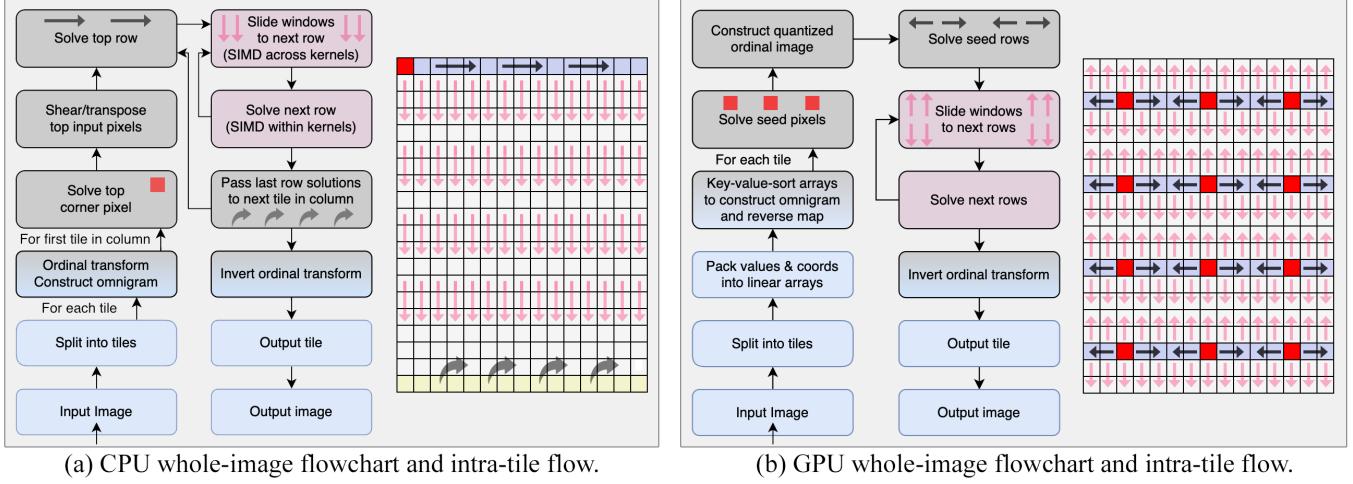


Fig. 9. CPU vs. GPU implementations. (a) On CPU, for the top tile in a column, we begin by solving the top left pixel (red), then step across the top row (blue), sweep to the bottom of the tile, then forward the last-row solutions to the next tile. (b) On GPU each tile is processed independently. We solve a sparse grid of "seed" pixels (red), then step left-right to complete sparse rows (blue), then sweep up-down to complete the tile.

the key-value sort must be stable to preserve relative coordinate order; otherwise, stability isn't required. Once Ω_I is constructed, the ordinal input image I is straightforwardly generated as:

$$I(\Omega_I[v]) \leftarrow v, \quad \forall v \in \{0, 1, \dots, |\Omega_I| - 1\} \quad (4)$$

The vertical window-sliding phase exploits the spatial coherence between adjacent windows shown in Fig. 8 to enable efficient SIMD parallelism. Each vector can load several adjacent values from the ordinal image at once, and compare them against the pivots for the corresponding windows, updating the corresponding counts accordingly.

Note that when sliding from window A to window B, Huang's method uses the median value m of window A as the "pivot" against which to compare entering and exiting values. We observe that the value used for this pivot does not need to be exactly m ; it may be any nearby value for which the exact "count" (number of pixels in the window strictly less than the pivot) is known. To facilitate vector alignment while scanning the omnigram, and also because quantized values can be stored in fewer bits, we choose our pivot to be the nearest multiple of 64 to the median value of window A.

After sliding our windows, the subsequent refinement phase scans Ω_I from the pivot index to find the exact solution for a given pixel. To do this, each 16-bit omnigram element must be reduced to a 1-bit histogram element for the relevant window, as per Eq. 3 and Fig. 6. We perform this reduction in 64-element segments, converting 128 contiguous bytes of Ω_I to a single 64-bit integer bitmask. The popcount (number of '1' bits) in this bitmask gives us a "coarse" histogram element for the circular region, spanning all 64 indices. If the median value is in this "bin", we scan the bitmask to find the exact ordinal solution m , writing the cardinal solution $C[m]$ to the output image. If not, we continue scanning Ω_I until the exact solution is found.

As our pivots are always multiples of 64, we observe that if our input tiles are no larger than 128x128, the pivot can be encoded with just 8 bits. We further observe that the low 6 bits of the ordinal image do not affect the result of comparisons with these quantized pivots, so those low bits can also be discarded; we can store just the high 8 bits of the ordinal image as well. For larger input tiles (necessary for filtering kernels above radius ~40), we still use 16 bits per ordinal pixel, to preserve the ability to scan Ω_I in 64-element increments.

On CPU, we implement the ordinal transform differently based on the type of the input data. For 8- or 16-bit integer data we use a single-pass bucket sort. For 16-bit inputs, our process for the ordinal transform is as follows:

- Construct a 65536-element 16-bit histogram H of the tile I_c .
- Exclusive prefix-sum the histogram: $H'[k] = \sum_{n=0}^{k-1} H[n]$.
- Re-scan the tile in row-major order. For each cardinal pixel value c at $I_c((x, y))$, select the ordinal value $v = H'[c]$ and increment $H'[c]$ by 1. Concurrently, construct the cardinal reverse map C , omnigram Ω_I , and ordinal image I :

$$\begin{aligned} C[v] &\leftarrow p, \\ \Omega_I[v] &\leftarrow (x, y), \\ I(x, y) &\leftarrow v \quad // \text{ or } v \gg 6, \text{ in the quantized case.} \end{aligned} \quad (5)$$

For floating-point images, we implement a modified radix sort to construct the ordinal map. We XOR the exponent and mantissa bits by the sign bit (and flip the sign bit) to allow comparisons of the bit-patterns as unsigned integer, then bucket-sort the top 16 bits in the first pass, followed by bucket-sorting 8 bits at a time to complete the radix sort. In the latter two stages, quicksort is used if only a small number of elements need to be sorted.

4.1 GPU

Our GPU CUDA implementation mirrors the CPU version in several aspects, but with a few key differences. The first of these involves the construction of the omnigram and reverse map. Whereas on CPU we use a custom bucket-sort or radix-sort, on GPU we rely on the highly optimized `cub::DeviceSegmentedRadixSort()` [CCCL Development Team 2023] to sort the entire set of input tiles in parallel. As fast memory is at a premium, we exclude corner pixels that cannot be part of any circular window; each logical input tile is thus a rounded rectangle (as shown in Fig. 10, rather than a square. The excluded corner pixels do not participate in the ordinal transform.

CUDA's 32-way warp parallelism informs our choice of tile size; we use output tile sizes of 32x32 for small radii, and 64x64 for larger radii, allocating 256 threads per tile for the former and either 512 or 1024 threads per tile for the latter. As the top-down scanline-by-scanline approach we use on CPU would be insufficient to saturate the GPU's massive concurrency, we instead solve from the middle out, starting from 32 or 64 "seeds" per tile, as illustrated in Fig. 9. The median values for the seed windows are solved directly by constructing low-precision histograms from the omnigram (which remains in global memory), scanning these histograms to find approximate median solutions, then refining to exact solutions using the omnigram.

Next, the ordinal image is constructed into shared memory. It is quantized by a 6-bit rightshift for $\text{radii} \leq 48$, or a 7-bit rightshift for $\text{radii} \leq 96$, enabling it to fit in 8 bits per element. A small amount of additional shared memory is used to store quantized pivots and counts for the seed rows, which are solved by sliding the kernel windows horizontally from the seed windows both left and right; each thread is responsible for one window and one direction. (Some threads are idle in this phase.) When scanning the omnigram to find exact solutions for these windows, the 32 threads in each warp collaborate to scan 64-element blocks of the omnigram for unsolved windows belonging to that warp, until all of the warp's windows have been solved.

Once the seed rows are solved, the seed row windows and new vertical directions (up/down) are reassigned to all threads; these windows are then slid vertically up and down to solve the rest of the tile. With each of the hundreds of threads sharing the same small single immutable ordinal image and omnigram, memory locality is optimal. Finally, as solutions are found, the ordinal->cardinal reverse map is applied to yield the final output image.

5 RESULTS

We ran our filter's benchmark on a 37-megapixel grayscale image on a 64-core AMD 5995WX CPU with hyperthreading disabled, and on an NVIDIA RTX 4060 GPU. On the same hardware, we ran Moroto's published benchmark code for their 2D Wavelet Matrix filter (2DWM), and also Adams' published benchmark code for their implementation of both their Separable Sorting Networks algorithm and of Perrault's Constant Time Median Filter (CTMF).

Across all bit-depths and architectures, our method overtakes the fastest SOTA square-kernel implementations between roughly

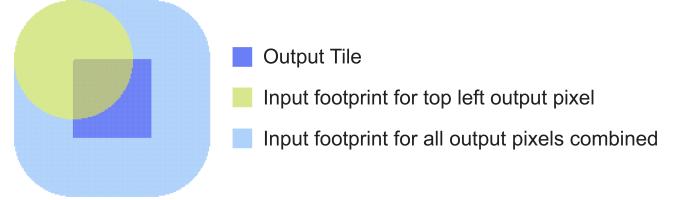


Fig. 10. Tile geometry on GPU. For the radius-48 case, each output tile is 64x64 pixels (dark blue). The corresponding input tile footprint (the union of circular kernel windows for every output pixel) is a rounded rectangle (light blue) comprising 23584 pixels, of which 16192 could potentially be the median of any given circular radius-48 window (e.g. yellow). Quantizing pivots to multiples of 64 allows us to cover this range using just 8 bits. (For larger radii, we quantize to multiples of 128.)

radius 8 and 12, and strongly outperforms SOTA octagonal or 12-sided methods² at all radii, often by over an order of magnitude. On 8-bit images, our method is slightly faster than CTMF's square-kernel implementation on CPU, and outperforms the 2D Wavelet Matrix square-kernel GPU implementation up to radius 84.

We have also implemented our method for the arm64 architecture, enabling it to run on a wider variety of desktop and mobile devices. And we have benchmarked it on higher-end graphics cards, such as the NVIDIA RTX 5080, confirming that our implementation's performance scales near-linearly in the number of cores. These results are discussed in supplemental material.

Qualitatively, our method provides substantially higher image quality than SOTA square-kernel implementations, as seen in Fig. 1 and supplemental images. Fig. 13 analyzes this effect quantitatively, showing that the filter's anisotropic artifacts are reduced by well over an order of magnitude with our method, to visually imperceptible levels. Our method may therefore even be preferred over SOTA sorting-network approaches for small kernels ($\text{radius} < 8$), where the latter retains a nominal performance advantage, at the cost of image quality.

6 LIMITATIONS AND FUTURE WORK

Our algorithm's runtime is data-dependent, and thus is theoretically vulnerable to "adversarial" inputs that could result in slow runtimes. (An example would be an input tile that is a high-frequency black-and-white checkerboard, with a thick solid gray perimeter.) We have not observed this with real-world images, but a mechanism to identify and more efficiently process such images might be an area to explore.

A second limitation is that the filter kernel (while isotropic) remains uniformly weighted throughout. Adapting the algorithm to be spatially weighted by e.g. a Gaussian kernel, perhaps along the lines of Kass and Solomon [2010], might be a useful line for future exploration. Making an "anti-aliased" kernel with smooth falloff at the edges might be another useful improvement.

²Based on Perrault's estimate that their octagonal method is 5x slower than square, and Moroto's estimate that their octagonal and 12-sided methods are 3x and 5x slower than square, respectively.

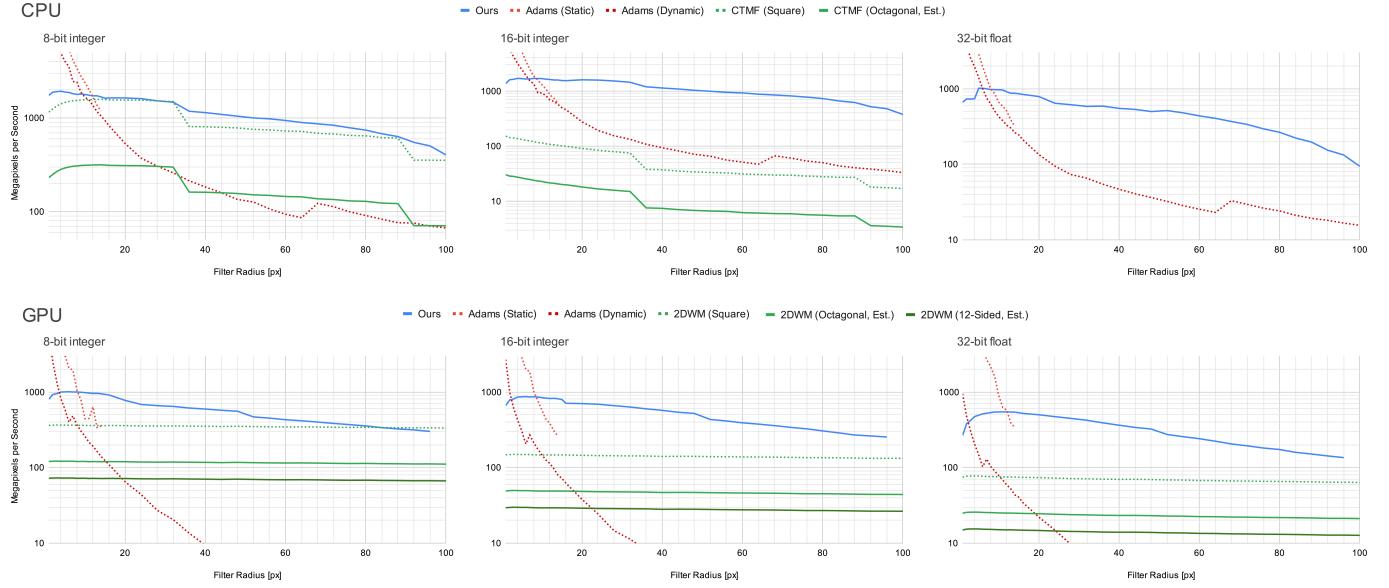


Fig. 11. Performance results on CPU and GPU. As our focus is on isotropic filtering, square-kernel implementations are shown with dashed lines.

6.1 Future Work

6.1.1 Coherence. A key feature of our method is that it relies on the coherence of median-filtered output; the result for each pixel is usually (but not always) similar to the result for the preceding pixel. A systematic statistical analysis of this coherence across a wide variety of images and image types would be useful, to determine optimal parameters for our method, as well as to identify contexts where the image data is more likely to be "adversarial" as described above.

6.1.2 Small Kernels. Our main optimization focus has been on medium to large kernels, but we believe there is room for improvement by fine-tuning our method for smaller tiles and smaller kernels. Additionally, we hope this work serves as inspiration for the development of sorting network techniques capable of processing small circular kernels with even higher efficiency.

6.1.3 Arbitrary Convex Kernel Shapes. While we anticipate that circular-kernel filtering will be the primary use case for our algorithm, it can be modified to handle a variety of kernel shapes. We discuss this further in supplemental material.

6.1.4 Ordinal Transform Redundancy. When the filter radius is a significant fraction of the tile size, there are opportunities to reduce redundancy when ordinal-transforming adjacent overlapping input tiles. E.g., when processing a radius-32 median filter using overlapping 128x128 input tiles (with 64x64 output tiles), each input pixel participates in four ordinal transforms, and thus is sorted four times. Partially sorting the overlapping regions in advance could result in significant speedups for larger kernels, particularly for floating-point images.

6.1.5 Multi-Parameter Filtering. Our implementation processes a single radius and percentile value at a time, but it could be straightforwardly modified to compute multiple such values at once, amortizing the cost of the ordinal transform and significantly increasing marginal efficiency. E.g. the filter could return three images with 25th, 50th, and 75th percentile values, for variance calculations. Similarly, it could be modified to process multiple kernel sizes (e.g. radius 8, 16, 32) within a single ordinal transform, mirroring the "runtime-only" efficiency of Moroto et al. [2022].

7 CONCLUSION

We have introduced a novel fast algorithm for median and general percentile filtering with circular kernels that provides significantly higher image quality and performance than the state of the art, across much of the useful parameter space. By overcoming the limitations of prior methods, we hope our results spur continued research into rank-order-filtering algorithms and their practical and creative applications.

ACKNOWLEDGMENTS

We are indebted to Google for providing the resources and environment necessary to pursue this research. Thanks to Yael Pritch, Adi Zicher, Neal Wadhwa, Chloe LeGendre, James Vecore, David Jacobs, Anna Lieb, Frank Barchard, and Artem Belevich for their support and assistance. Thanks to Michael Herf for his timely hardware wizardry. We also thank the anonymous reviewers for their time and helpful comments. Finally, special thanks to Jenna McGrath for her unwavering support, and for the perfect photo.

REFERENCES

- Andrew Adams. 2021. Fast Median Filters using Separable Sorting Networks. *ACM Trans. Graph.* 40, 4 (aug 2021), 1–11. <https://doi.org/10.1145/3450626.3459773>
- Kenneth E. Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. ACM, ACM, New York, NY, USA, 307–314. <https://doi.org/10.1145/1468075.1468121>
- CCCL Development Team. 2023. CCCL: CUDA C++ Core Libraries. NVIDIA. <https://github.com/NVIDIA/cccl>
- Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez. 2015. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems* 47 (2015), 15–32. <https://doi.org/10.1016/j.is.2014.06.002>
- T.S. Huang, G.J. Yang, and G.Y. Tang. 1979. A Fast Two-Dimensional Median Filtering Algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing ASSP-27*, 1 (jan 1979), 13–18. <https://doi.org/10.1109/TASSP.1979.1163188>
- Michael Kass and Justin Solomon. 2010. Smoothed local histogram filters. *ACM Trans. Graph.* 29, 4, Article 100 (jul 2010), 10 pages. <https://doi.org/10.1145/1778765.1778837>
- Morgan McGuire. 2008. A fast, small-radius GPU median filter. In *High Performance Graphics 2008*. Eurographics Association, Eurographics Association, Aire-la-Ville, Switzerland, 59–66. <https://doi.org/10.2312/HPG/HPG08/059-066>
- Deepa Mishra, Sangeeta Sharma, and V.K. Shandilya. 2020. A Circular Adaptive Median Filter for Salt and Pepper Noise Suppression from MRI Images. *International Journal of Biomedical Engineering and Technology* 36, 4 (2020), 364–382.
- Yuji Moroto and Nobuyuki Umetani. 2022. Fast median filters using 2D wavelet matrix. *ACM Transactions on Graphics (TOG)* 41, 6 (2022), 1–13. <https://doi.org/10.1145/3550454.3555512>
- Simon Perreault and Patrick Hébert. 2007. Median Filtering in Constant Time. *IEEE Transactions on Image Processing* 16, 9 (sep 2007), 2389–2394. <https://doi.org/10.1109/TIP.2007.902329>
- Fatih Porikli. 2005. Integral histogram: a fast way to extract histograms in Cartesian spaces. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Proc. of IEEE Conference On Computer Vision and Pattern Recognition* 1, 829–836 vol. 1. <https://doi.org/10.1109/CVPR.2005.188>
- John W. Tukey. 1974. Nonlinear (nonsuperposable) methods for smoothing data. In *In 1974 EASCON convention record*. IEEE, IEEE, New York, New York, NY, 673.
- Ben Weiss. 2006. Fast Median and Bilateral Filtering. *ACM Trans. Graph.* 25, 3 (jul 2006), 519–526. <https://doi.org/10.1145/1141911.1141918>
- Qingxiong Yang, Narendra Ahuja, and Kar-Han Tan. 2015. Constant Time Median and Bilateral Filtering. *Int. J. Comput. Vision* 112, 3 (May 2015), 307–318. <https://doi.org/10.1007/s11263-014-0764-y>



Fig. 12. Rank-order "bracketing", for a range of percentiles. Our circular-kernel implementation is shown across the top; an equivalent square-kernel percentile filter across the bottom. The square artifacts are particularly evident toward the endpoints of the percentile range.



Fig. 13. Rotational invariance. The 16-bit input (left) is rotated 22.5° in either direction with bicubic resampling, filtered with equivalent-area square and circular median filters, then rotated back. Our circular-filtered results match each other ~35x more precisely under rotation than standard square-filtered results. (Standard deviation 0.06 levels versus 2.12 levels.) The filtered swatches are shown with 200% unsharp masking to emphasize high frequencies.



Fig. 14. Varying percentiles reveals different image characteristics. Here the input, and 10th, 30th, 50th, 70th, and 90th percentiles, with our radius-48 filter.

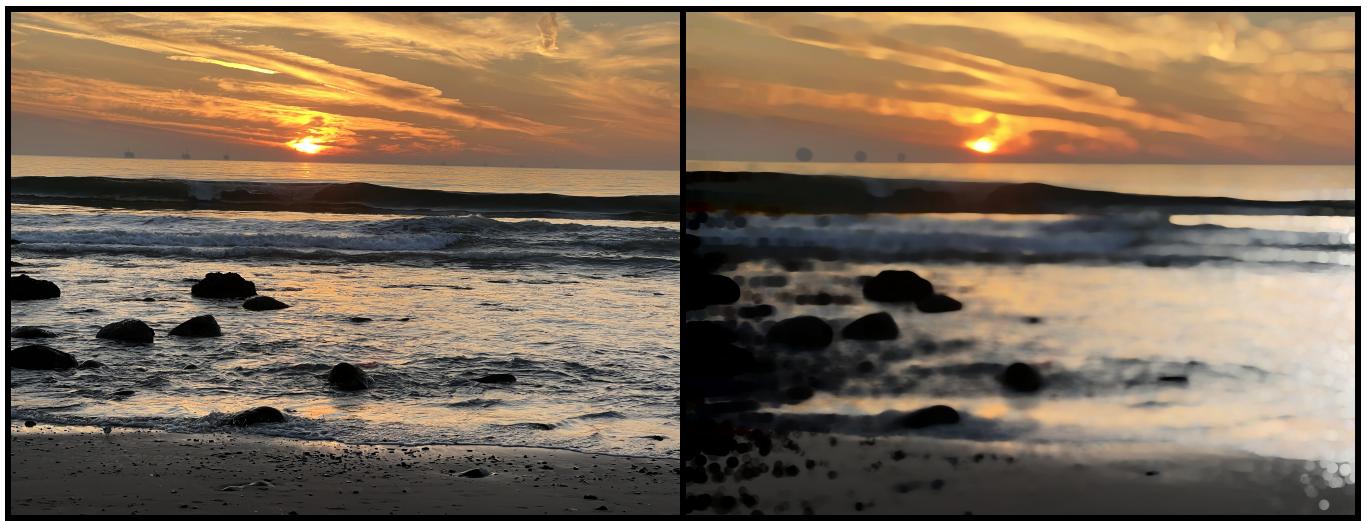


Fig. 15. The requested percentile does not need to be constant. Input (left), output (right). Our method can be modified to allow the percentile to vary continuously across the image, shown here varying 0% to 100% left-to-right, using our circular radius-32 filter.