Fast Isotropic Median Filtering— Supplemental Material

BEN WEISS, Google Research, USA

ACM Reference Format:
Ben Weiss. 2025. Fast Isotropic Median Filtering— Supplemental Material. 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, 3 pages. https://doi.org/10.1145/3721238.3730763

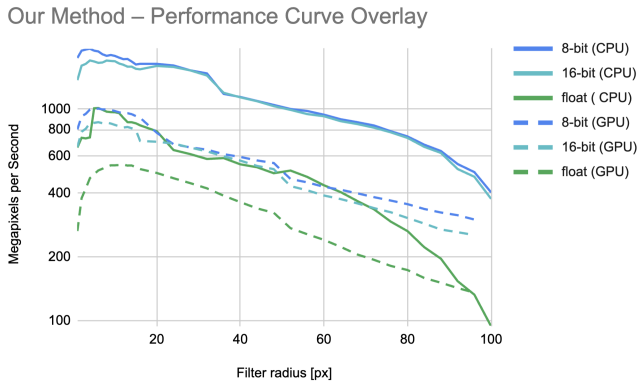

Fig. 1. Our method's performance curves for various platforms and data types, superimposed.

1 RUNTIME CHARACTERISTICS

As shown in Fig. 1, our method's throughput with respect to filter radius does not follow a simple characteristic curve like O(1) or O(r). Instead, at first it becomes faster as the radius increases, then displays a nearly flat runtime between radius 8 and radius ~32 (particularly for integer types on CPU), then tapers off roughly linearly for large radii, dipping more sharply toward radius 100. As the algorithm is nominally $O(r)$, it displays unexpectedly strong performance in the intermediate range of filter sizes from roughly radius 8 to 60. This unusual performance curve deserves more explanation.

On CPU, the flat part of the curve is partly attributable to the ability of wide SIMD architectures to absorb much of the early filter size increase "for free". For instance, our inner loop scans the omnigram in 64-element sections; SIMD parallelization and pipelining make this nearly as fast as processing a much smaller number of elements. We also find that the minimum practical output tile size is 32x32; smaller tile sizes make it difficult to saturate the

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

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

ACM ISBN 979-8-4007-1540-2/2025/08
https://doi.org/10.1145/3721238.3730763

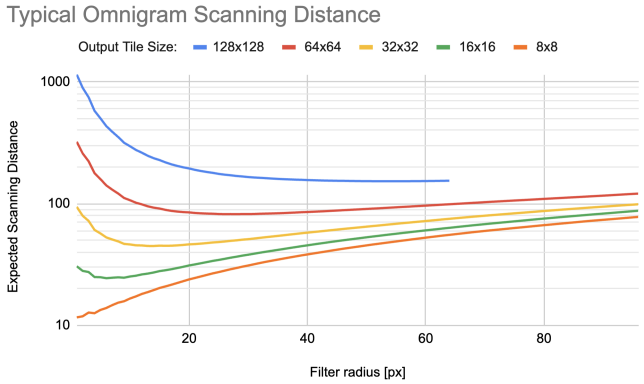

Fig. 2. Average omnigram scanning distance, by filter radius and output tile size, for the 8-megapixel photo from the paper's leading figure. The sharp increase for small radii, combined with reduced vectorizability of smaller tiles, limits our method's performance in the small-kernel range.

concurrency of the SIMD architecture, such as AVX2 which has a native width of 32 8-bit elements.

A second important factor is omnigram scanning distance. The pivots-and-counts technique relies on the heuristic that adjacent median-filtered values are usually similar. We find that for a given output tile size, this heuristic also performs best for an intermediate range of filter sizes. Fig. 2 illustrates the relationship between output tile size, filter radius, and average scanning distance.

For very large kernels, we reach a "cliff" at an input tile size of 256x256, where exceeding that limit would require doubling the omnigram size to 32 bits per element. For this reason, our current implementation constrains the input tile size to 256x256. This leads to an $O((128 - r)^{-2})$ performance curve for large radii (where the bottleneck becomes the ordinal transform), and this explains the dip in performance as the filter radius approaches 100. But this dip is not fundamental to the algorithm; in principle, switching to a 32-bit omnigram would yield an $O(r)$ performance curve as the radius scales to 100 and beyond.

On GPU, the primary considerations are fast memory capacity and thread occupancy. For our algorithm, the 32-threads-per-warp CUDA architecture works most naturally when the output tile size is a multiple of 32. For a 64x64 output tile size, we reach the 16-bit omnigram limit at radius 96. It would be possible to implement up to radius 112 by using a 32x32 output tile size, but performance would drop substantially for that minimal increase in filter radius. However, as GPU L1 cache sizes increase in the future (already some datacenter GPU's have ~256k of fast memory per SM), we expect larger kernels to become more viable by switching to a 32-bit omnigram.

2 ALGORITHMS (PSEUDOCODE)

For the benefit of those who would like to implement our algorithm on their own, we provide a pseudocode outline of our CPU and GPU methods. Note that for a more complete reference, our working code can be found at https://github.com/google/fast-isotropic-median-filter.

2.1 CPU Algorithm

Algorithm 1. CPU Implementation (pseudocode)

for all columns of cardinal output tiles O_c **do**
 for all tiles in column **do**
  generate data structures: $I_c \mapsto \{I, \Omega_I, C_I\}$ ▷ Eq. 5
  if top tile in column **then**
   solve $m(0,0)$, storing aligned pivot p and count c
   shear/transpose top rows of input pixels ▷ Fig. 7
   for all output pixel columns col in top row **do**
    slide window right, updating $c[col]$
    solve $m(col, 0)$ by scanning Ω_I from $p[col]$
    select nearby 64-element-aligned $p[col]$ and $c[col]$
    write to output: $O_c(col, 0) = C_I[m(col, 0)]$
   end for
  else
   Incorporate forwarded solutions from previous tile
   for all pixels (x, 0) in first row **do**
    Obtain median from ordinal image.
    []If I is quantized, scan Ω_I to refine exact median.]
 $m(x, 0)$ = I(last_offset[x] + (x + rad, rad))
   end for
  end if
  for all subsequent rows in output tile **do**
   for all leading/trailing pixel offsets (dx, dy) **do**
    for all output columns **do** ▷ Fig. 8
     $c[col]$ += $Compare(I[col + dx, row + dy], p[col])$
    end for
   end for
   for all output columns **do**
    scan Ω_I from $p[col]$ to find exact $m(col, row)$
    select nearby 64-element-aligned $p[col]$ and $c[col]$
    write to output: $O_c(col, row) = C_I[m(col, row)]$
   end for
   Forwards last-row solutions to next tile
   if last row in tile **then**
    for all pixels (x, y) in last row **do**
     Store offset of m(x, y) from window center
     last_offset[x] = $\Omega_I[m(x, y)] - (x + rad, y + rad)$
    end for
   end if
  end for
 end for
end for

2.2 GPU Algorithm

Algorithm 2. GPU Implementation (pseudocode)

for all output tiles O_c **do**
 pack input pixels + coords (Fig. 10) into key-value arrays
end for

Using e.g. cub::SegmentedDeviceRadixSort():
key-value sort arrays to construct Ω_I and C_I

for all output tiles O_c **do**
 for all seed pixel windows $W_{s_{xy}}$ in parallel **do**
  Construct coarse seed-window histogram $H_{s_{xy}}$
  for all indices k in Ω_I **do**
   if $\Omega_I[k] \in W_{s_{xy}}$ **then** $H_{s_{xy}}[k >> 6]$++
  end for
  Note: the approximate median defines the pivot $p[s_{xy}]$.
  scan $H_{s_{xy}}$ to find approximate median for $W_{s_{xy}}$
  refine to exact median $m(s_{xy})$ by scanning Ω_I
  write to output: $O_c(s_{xy}) = C_I[m(s_{xy})]$
 end for
 Construct ordinal image into shared memory
 for all indices k in Ω_I **do**
  $I[\Omega_I[k]]$ = k » 6
 end for
 Slide horizontally from seed windows to solve seed rows
 for all seed windows $W(s_{xy})$ in parallel **do**
  for all remaining columns until seed row is solved **do**
   for all leading/trailing pixel offsets (dx, dy) **do**
    $c[xy]$ += $Compare(I[col + dx, row + dy], p[xy])$
   end for
   scan Ω_I from $p[xy]$ to find exact median m(xy)
   set p[xy] to a quantized value near m(xy)
   write to output: $O_c(xy) = C_I[m(xy)]$
  end for
 end for
 Slide vertically from seed rows to solve rest of tile
 for all seed row windows W_{xy} in parallel **do**
  for all remaining rows until tile is solved **do**
   for all leading/trailing pixel offsets (dx, dy) **do**
    $c[xy]$ += $Compare(I[col + dx, row + dy], p[xy])$
   end for
   scan Ω_I from $p[xy]$ to find exact median
   write to output: $O_c(xy) = C_I[m(xy)]$
  end for
 end for
end for

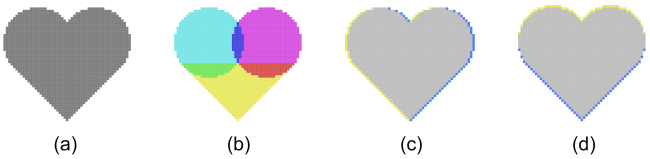

Fig. 3. (a) A heart shape can be represented by (b) two intersecting circles and a triangle. During the window-sliding phase, leading and trailing pixels are shown for (c) horizontal and (d) vertical sliding.

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 convex (or mostly-convex) kernel shapes. Simple shapes such as polygons (including squares) can be achieved by correspondingly modifying the analytical test in the paper's Eq. 3, remaining fully vectorizable on CPU. For instance, a heart shape (Fig. 3) could be modeled as the union of two intersecting circles $\{C_1, C_2\}$ and a triangle T. In this specific case, Eq. 3 would be replaced with:

$$H_{\heartsuit}[v] = \begin{cases} 1 & \text{if } (\Omega_I[v] \in C_1) \vee (\Omega_I[v] \in C_2) \vee (\Omega_I[v] \in T) \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

and the sliding-window phase would use leading and trailing offsets as shown in Fig. 3.

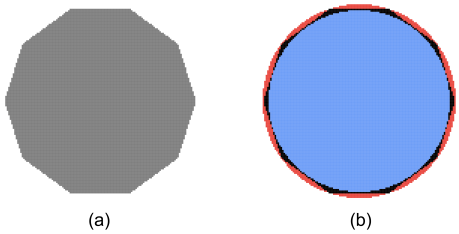

Fig. 4. (a) A decagon could be implemented with a more complex analytical formula, or with a "trimap" approach (b). Points inside the blue circle would be included, points outside the red circle excluded, and intermediate points solved by bitmap lookup.

More complex shapes could be achieved by implementing the paper's Eq. 2 with a bitmap lookup, which would remain efficient on GPU, but might sacrifice pure vectorizability on CPU, even if most queries could still be solved analytically by testing against e.g. a bounding circle. Shapes of intermediate complexity could be implemented with a "trimap" approach (Fig. 4): testing all omnigram elements against inscribed and circumscribed shapes, and reserving bitmap lookups only for the infrequent "midliers".

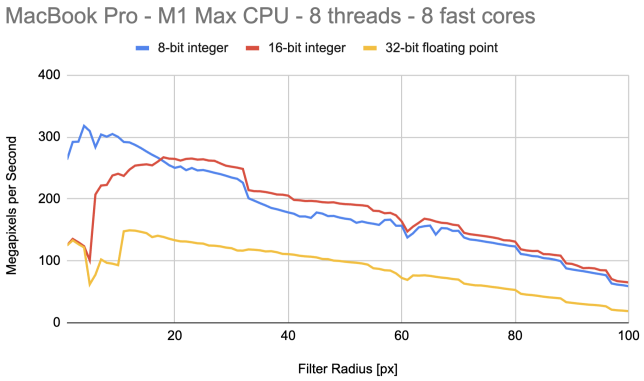

Fig. 5. Our method's performance on arm64 CPU.

4 ADDITIONAL ARCHITECTURES

In addition to x86_64 and CUDA, we have implemented an optimized version of our algorithm for the arm64 architecture, which runs on both desktop and mobile devices. As shown in Fig. 5, performance on an M1 MacBook Pro CPU is very strong; on a per-core basis it consistently outperforms the AMD 5995WX Threadripper by about 20 to 60 percent.

For our particular algorithm, one specific advantage we find on both x86_64 and CUDA architectures is the presence of a "find-nth-set-bit" hardware instruction, as "pdep" on x86 and "__fns()" on CUDA. The presence of an x86 instruction to rotate just the low bits of an integer register is also helpful for our custom radix sort.

On NVIDIA GPU's, we find that our method's performance scales almost exactly linearly with CUDA cores; e.g. on RTX 5080 hardware (with 3.5x the core count) we've measured that the per-core throughput is ~99% that of the RTX 4060. For other GPU architectures such as AMD, we find that the limiting factor is the scarcity of pre-existing libraries for e.g. parallel key-value-sorting or merging. As more such foundational libraries become available, we expect that algorithms like ours will become that much easier to develop and distribute across a wider variety of platforms.