Fast Median Filters using Separable Sorting Networks

ANDREW ADAMS, Adobe Research, USA

Fig. 1. The median filter is a staple of photographic image processing. Here a 29×29 median filter was applied (with hand-painted matte) in Adobe Photoshop to detexture the background and remove stray hairs (center top), de-emphasize moles and blemishes (center middle), and soften wrinkles (center bottom). In practice, this processing would be followed by copying the highest frequencies back from the original to reintroduce grain. This image is 16-bit, as is now standard. The technique described in this paper computes median filters 8× faster than the state of the art at this size and bit-depth, and is up to 100× faster at other sizes and bit-depths. See supplemental material for additional typical uses of a median filter in photo editing.

Median filters are a widely-used tool in graphics, imaging, machine learning, visual effects, and even audio processing. Currently, very-small-support median filters are performed using sorting networks, and large-support median filters are handled by $O(1)$ histogram-based methods. However, the constant factor on these $O(1)$ algorithms is large, and they scale poorly to data types above 8-bit integers. On the other hand, good sorting networks have not been described above the 7×7 case, leaving us with no fast way to compute integer median filters of modest size, and no fast way to compute floating point median filters for any size above 7×7.

This paper describes new sorting networks that efficiently compute median filters of arbitrary size. The key idea is that these networks can be factored to exploit the separability of the sorting problem – they share common work across scanlines, and within small tiles of output. We also describe new ways to run sorting networks efficiently, using a sorting-specific instruction set, compiler, and interpreter.

The speed-up over prior work is more than an order of magnitude for a wide range of data types and filter sizes. For 8-bit integers, we describe the fastest median filters for all sizes up to 25×25 on CPU, and up to 33×33 on GPU. For higher-precision types, we describe the fastest median filters at all sizes tested on both CPU and GPU.

Author's address: Andrew Adams, Adobe Research, 345 Park Avenue, San Jose, CA, 95110, USA.

0730-0301/2021/8-ART70 $15.00
https://doi.org/10.1145/3450626.3459773

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

Additional Key Words and Phrases: Median filters, sorting networks

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

1 INTRODUCTION

The venerable median filter [Tukey 1974], which replaces each pixel with the median of its neighbors, has many useful properties. It is robust to outliers, making it useful for removing hot pixels or salt-and-pepper noise. It is invariant to monotonic transformations, avoiding nagging questions about gamma correction or linear vs log luminance. Unlike a bilateral filter, it has no intensity-related parameter, which neatly avoids issues that arise in images with regions of very different contrast. Finally, its derivative is trivial to compute – simply backpropagate all error to the input that was selected as median.

For these reasons, the median filter is widely used. It has been a Photoshop staple for decades, where it is popular for denoising, skin smoothing (Fig. 1), and removing dust and scratches. In the technical literature, it is often an important but unglamorous component of a more complex algorithm. To pick a few examples spanning multiple decades, Freeman [1988] uses it to suppress color artifacts that arise in demosaicking. For a long time it was a standard tool in optical flow, with Sun et al. [2010] delving into why it is so helpful. It is still popular in machine learning as a network layer. For example, Žbontar and LeCun [2016] use it in stereo matching. It is a key part

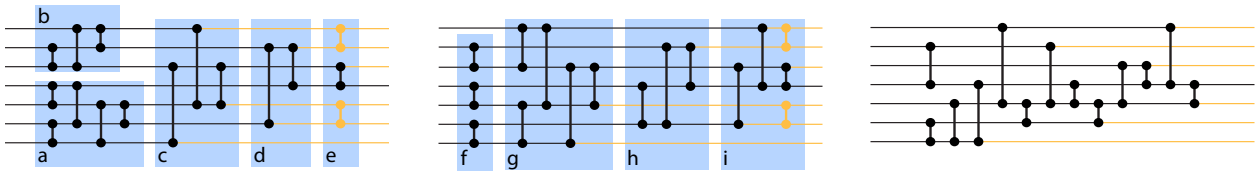


Fig. 2. The three primitive types of sorting network used in this paper, demonstrated here computing a median of 7 values. A sorting network is a sorting algorithm consisting of a fixed sequence of min and max instructions. These are often depicted graphically as a circuit. The values to sort travel along the horizontal lines from left to right. Each vertical link swaps two values if they are not in the correct order using a min/max instruction pair. *Left:* Batcher's odd-even mergesort first sorts each half of the input separately (a, b). It then merges the odd-numbered values (c), and then the even-numbered values (d). A final series of pairwise swaps completes the sort (e). The swaps and values in orange are unnecessary if we only want the median output, and can be pruned, leaving 14 swap operations in total. These pruned sorting networks are known as "selection" networks. The odd-even mergesort is our network of choice for merging two already-sorted lists, as we can simply skip steps a and b. *Middle:* Parberry's pairwise sort first sorts the values in pairs (f), then sorts the odd-numbered values (g), and then the even-numbered values (h). A slightly more complex merge step completes the sort (i). For a full sort, the pairwise sorting network always uses the same number of swaps as the odd-even mergesort. However, in the case where there are many inputs and only some outputs are desired, the pairwise sort can be more aggressively pruned. It is also useful when the input is already sorted in pairs (as in the right of Fig. 8), as we can skip step (f). *Right:* Neither of these networks performs the minimum number of swaps to find the median of 7 values. For small numbers of inputs, we can use branch-and-bound search to find the optimal sorting network for a task.

of some visual effects pipelines [Vavilala 2019]. The 2D median filter is even used in audio processing – Fitzgerald [2010] applies it to spectrograms to separate the harmonic from the percussive.

1.1 The problem

Compared to the vast literature on linear filters, not much is known about how to run median filters efficiently. There was a flurry of interest around 15 years ago which culminated in several histogram-based algorithms that do $O(1)$ work per pixel (e.g. [Perreault and Hébert 2007]). However, the constant factor is large — around a hundred cycles per pixel at 8-bit, and a thousand cycles per pixel at 16-bit. Worse, these algorithms are nigh-impossible to adapt to increasingly-standard floating point formats[1]. So the usefulness of these histogram-based methods has decreased over time as commonly-used pixel data types have increased in precision.

The other basic approach to computing median filters is sorting. Sorting *networks* [Batcher 1968] are a popular approach to this, because they sort using a fixed sequence of min and max instructions, with no data-dependent control flow, which means that SIMD instructions can fully employed. A 16-bit 3×3 median filter computed in this way uses a mere two cycles per pixel on a current x86 processor. Unlike the histogram-based approaches, the cost of a sorting network scales linearly with the bit-width, so a floating point 3×3 median filter clocks in at just under four cycles per pixel.

However, sorting networks do not scale gracefully with the filter size in two ways. First, the size of the sorting network grows superlinearly with the support of the filter, which means that large filters are slow to run. Second, to realize the benefits described above, a separate piece of code must be compiled per size, and the size and hence compile-time of this code also grows with the size of the filter.

Finally, the difficulty of *finding* a good sorting network also grows quickly with filter size. Optimal networks are known for small sizes, but even a 5×5 median filter is well outside this regime. As we will see in Section 2, prior work often uses inefficient networks even at the small sizes supported.

For these reasons, popular libraries use sorting networks for 3×3, 5×5, and occasionally 7×7 filters, and use histograms above that (or simply do not support larger sizes).

1.2 The contribution

This paper tackles these three problems. To address the size of the sorting network, Section 3 describes new ways to factor the work performed when computing a median filter into a composition of sorting networks so that most work can be shared between nearby pixels. These are analogous to the separability tricks used when computing linear filters. For an $d \times d$ filter, the two types of separability we exploit are:

(1) Per scanline of output, pre-computing every sorted column of d pixels, so that the per-pixel task can start from d sorted arrays of size d rather than an unsorted array of size d^2.
(2) Computing output in small tiles of a size that grows with the filter size, with the maximum amount of work done at the intersection of the footprints of all pixels in the tile, less work done in the parts of the footprint that are common to only a few outputs, and minimal work done in parts of the footprint unique to a single output.

We also reduce the size of the component sorting networks themselves by first judiciously selecting the best ones for each task, and then applying techniques from superoptimization to further specialize them to that task by fusing operations, removing operations, or just searching for an optimal network from scratch (as we did in Fig. 2).

To address the second problem (needing to compile a separate piece of code per filter size), Section 4 describes a new family of sorting networks with coarser-grained primitives than single min/max instruction pairs (which we call a "swap" in this paper). These networks perform moderately more swaps than existing networks, but many fewer loads and stores, so they are around three times faster to

[1]Though Weiss cleverly used an ordinal transform for this [Weiss 2006].

dynamically execute than unstructured networks. We then describe a specialized low-overhead interpreter for running these sorting networks. In Section 5 we benchmark this new algorithm and show its performance relative to other ways of computing a median filter.

2 PRIOR WORK

2.1 Histogram-based median filters

The basic insight underlying fast histogram-based methods for computing median filters is that a histogram over a square region can be computed by spatially filtering per-pixel histograms. Various approaches exist for computing arbitrarily-sized box filters in $O(1)$ time, and each can be extended to filtering histograms. A popular algorithm in this class is the Constant-Time Median Filter (CTMF) of Perreault et al. [2007], which applies the filter separably using sliding windows. Porikli [2005] used integral images. Weighted or approximate median filters can also be computed using IIR filters [Kass and Solomon 2010; Yang et al. 2015].

Sánchez et al. [2013] describe an $O(1)$ cumulative-density-function-based algorithm with a fast GPU implementation. Faster still is Green's GPU adaptation of the CTMF [2017], which we will use as a baseline for histogram-based methods on GPU.

Histogram-based methods are and will likely continue to be the fastest methods for very-large-support median filters for low-bit-width integer types. This paper advances the state of the sorting network approach, demonstrating that they are superior for small to medium filters or high-bit-width types.

2.2 Sorting networks

Sorting networks were first introduced by Batcher [1968]. See Figure 2 for an explanation of how they work. Readers may be most familiar with his bitonic network, which is useful for sorting large arrays on massively-parallel architectures. We instead parallelize across independent sorting tasks, so Batcher's odd-even mergesort is superior as it uses fewer swap operations. We use Knuth's generalization of Batcher's merge procedure [Knuth 1998].

Parberry [1992] later invented the pairwise sorting network, which can be pruned more aggressively than the odd-even network when only some of the outputs are required [Codish and Zazon-Ivry 2010]. When pruned, these are known as pairwise *selection* networks.

2.3 Sorting-network-based median filters

Readers in graphics may be familiar with small-support median filter shaders implemented entirely with min and max instructions [McGuire 2008]. These are sorting networks. McGuire's 3×3 filter uses 20 swaps to compute a median of 9, where 19 is possible using the procedure in Figure 4. He also proposes a 5×5 filter that uses 94 swaps, but unfortunately it is incorrect[2]. OpenCV [2000] uses a sorting network with 19 swaps in the 3×3 case, identical to the one first proposed by Mahmoodi (as described by Waltz [1989]). In the 5×5 case, OpenCV uses 113 swaps. The procedure in Figure 4 uses 99, and a pairwise selection network uses 103.

[2]An input patch where it fails, provided by a SAT solver, is: 

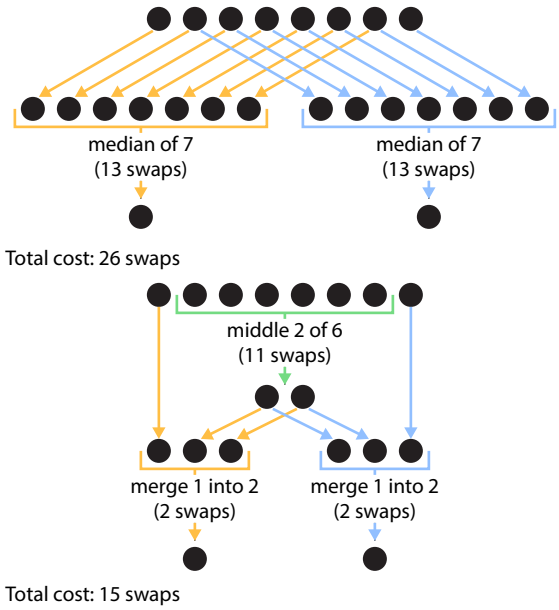


Fig. 3. When finding the median of two overlapping windows, much of the work can be shared. *Top:* The optimal network for finding the median of 7 values uses 13 swaps, so without sharing work, computing two adjacent outputs uses 26 swaps. *Bottom:* A median of 7 is greater than 3 other values and less than 3 other values. If we first sort the shared 6 values at the intersection of the footprints, then only the middlemost 2 of those could possibly be a median of 7, as the first 2 values are less than 4 others, and the last 2 values are greater than 4 others. This illustrates an important principle used in this paper: To find a median of n values, with k values held out (here $k = 1$), we only need the middlemost $k + 1$ values of what remains. Finding the middlemost 2 of 6 in sorted order can be done in 11 swaps. Two more swaps per output complete the task. By exploiting this kind of separability we can substantially reduce the total work.

Waltz et al. [1998] later extended Mahmoodi's method to reuse 6 of the comparisons done at the previous output pixel to reduce the steady-state work to a mere 13 swaps. This is equivalent to the first kind of separability described in Section 1. Kim at al. [2015] describe a generalization of this strategy to 5×5 filters and also provide a more SIMD-friendly implementation strategy. Kim's method is the transpose of the one illustrated in Figure 4 up to step d, at which point it diverges.

Intel does not reveal what methods are used by their IPP [2004] package for median filters, but reading the assembly reveals chains of SIMD min and max operations for the 3×3 and 5×5 cases, with some reuse of previous intermediate values in the manner of Waltz et al.

Perrot [2014] describes sorting network-based median filters up to 7×7 based on an $O(d^4)$ "forgetful selection" network similar to McGuire's 3×3 network. Notably, Perrot also shows how the bulk of this network can be shared when computing two adjacent outputs. This is the second type of separability described in Section 1. Salvador [2018] extends these ideas to 2×2 tiles of outputs, and compares the method to a "diagonal" sort which is equivalent to

Figure 4 at these small sizes. The maximum filter size is still 7×7 in this work.

Our paper can be viewed as a generalization of Salvador's method to arbitrary filter and tile sizes, though we dispense with the forgetful selection network, as it does not scale well (see Fig. 6). Unlike Salvador, we also incorporate the kind of separability described by Waltz et al. [1998], though this has a diminishing effect on total runtime as filter sizes grow (Fig. 6).

Larger selection networks show up in the SAT-solving literature, where they are used to encode cardinality constraints. Codish and Zazon-Ivry [2010] argue that a pairwise selection network is best for this task, a result which we rely on in this paper.

3 SEPARABLE SORTING NETWORKS

In this section we describe our algorithm for computing a median filter. It will be a composition of various sorting networks. The main idea is to craft this composition so that as much work as possible can be shared between multiple outputs. In Section 3.1 we describe the primitive networks used. Section 3.2 describes our sorting network for computing a median filter at a single output pixel. We call this "diagonal sort", as it is a generalization of the network of the same name used by Salvador et al. [2018]. In Section 3.3 we generalize this median-filtering algorithm to computing small tiles of output pixels at once, by first applying the diagonal sort to the intersection of the footprints of the output pixels, and then using merging networks to integrate the remaining inputs.

3.1 Sorting and merging

3.1.1 Sorting. To sort a list, we use a pairwise selection network, pruned to the output range of interest in the manner suggested by Codish and Zazon-Ivry [2010]. For full sorts of sizes up to 16 we use Knuth's list of the best known sorting networks [Knuth 1998]. For other tasks (e.g. median-finding) at sizes less than 13 we found we could improve runtime by about 1% using our own table of superoptimized sorting networks, computed as described in supplemental material.

3.1.2 Merging. To merge two sorted lists, we use the merge step from Batcher's odd-even merge network [Batcher 1968], as generalized by Knuth to handle two lists of arbitrary size [Knuth 1998]. To merge n lists, we can either merge them pairwise in a binary tree, or if they are all the same size we can also generate a sorting network of size n, and interpret each swap (i, j) in it as a directive to merge lists i and j in place. We use this second method in Figure 7.

There is an important principle we can exploit when merging two lists of different size when we only need a small number of the outputs (for example, just the median). Say we have two sorted lists of size $n - k$ and k, and we only want outputs at indices α through β inclusive (counting from index 0). The largest output of interest is greater than exactly β other values from the union of both lists. Anything at an index greater than β in either list is therefore already greater than too many other values to possibly be of interest to us, and can be ignored. So we can trim each list down to at most size $\beta + 1$ before we even start the merge. A similar argument can be made using α to trim the start of each list.

In the special case of median-finding, the principle simplifies to this: If you are finding a median of n things, and you have only seen $n - k$ of them so far, then only the middlemost $k + 1$ of those could be the median, and the rest can be discarded. This is the principle behind the forgetful selection sort used by McGuire [2008], Perrot [2014], and Salvador [2018]. It is illustrated for the case of $n = 7, k = 1$ in Figure 3.

3.2 Diagonal sort

We now describe a new sorting network for selection tasks (e.g. median-finding), based on a generalization of the ones used for small sizes in prior work [Kim et al. 2015; Perrot et al. 2014; Salvador et al. 2018; Waltz 1989]. It is best explained visually, and is depicted in Figure 4. The network first sorts a rectangular footprint down the columns, then along the rows, then diagonally up and to the right. Each step preserves the orderings induced by the previous steps (see supplemental material for a proof). After each sort we eliminate each value that could not possibly be in the range of interest by counting the number of other values known to be greater or less than it[3]. The correct outputs are extracted from what remains using a pairwise selection network. For the fully general description of the algorithm, see the appendix or the included source code.

To use this diagonal sorting network as a median filter of diameter d on an image of width w, the entire algorithm is: For each scanline (in parallel), first compute a $w \times d$ array of the size-d sorted columns centered on each pixel (using a pairwise sorting network), and then for each overlapping $d \times d$ window, copy the window into a scratch buffer and apply the algorithm as depicted in Figure 4.

3.3 Tiling to share overlapping work

The algorithm described above computes the output at each pixel largely independently, sharing only the presorted columns. If we produce small tiles of output instead of single pixels, we can share much of the remaining work between the pixels in that tile. To do this, we first apply the algorithm above to the intersection of the footprints of the pixels in the tile, and then separately handle the values outside that intersection. We use the middlemost $k + 1$ principle to set α and β at each step of this procedure.

Again, this is best explained visually. See Figure 5 for a 7×7 filter computed in 2×2 tiles, and Figure 7 for an 11×11 filter computed in 4×4 tiles. The 4×4 case readily generalizes to larger tile sizes. For a fully general description, see the appendix or the included source code.

This tiling is profitable at all sizes, with the ideal tile size growing with the filter size (Fig. 6). For compiled filters, we use 1×2 tiles for a 3×3 filter, 2×2 tiles for filters up to diameter 23, and 4×4 tiles above that. For interpreted filters, we use 2×2 tiles for small filters, switching to 4×4 at diameter 15, and then 8×8 at diameter 61. These thresholds were set empirically.

4 IMPLEMENTATION

The previous section described how to combine several sorting networks to compute a median filter in as few swaps per pixel as possible. In this section we describe how to implement these

[3] This is equivalent to Kim's lemma 1 [2015]

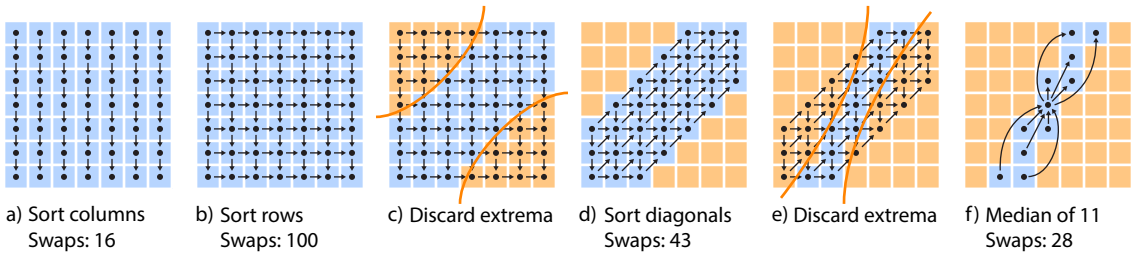


Fig. 4. (a) To find the median of a square footprint, we first sort the columns. The arrows indicate the ordering relationships that result. This footprint overlaps with other nearby pixels in the same scanline, so we precompute these sorted columns per scanline of output, resulting in an amortized cost of sorting one additional column. This is a form of separability. (b) We then sort each row. The columns remain sorted (see the supplement for a proof). (c) With these orderings established, each value is greater than everything above and to the left, and less than everything below and to the right. We can therefore exclude a hyperbolic slice at the top left as being less than too many other values to possibly be the median, and similarly exclude a hyperbolic slice at the bottom right as being greater than too many other values to possibly be the median. Rewinding for a moment to step b, notice that the stated cost (100) is not a multiple of the number of rows. By anticipating the exclusion that occurs in step c, we can save some work. In the top row, for example, we can use a sorting network that gives the top 3 of 7, rather than one that sorts the entire row. (d) We then sort up and to the right along each diagonal. Surprisingly, the rows and columns remain sorted (again see the supplement for a proof). (e) The new diagonal ordering relationships mean that each value is greater than a wide fan of values to the right, and less than a similar fan of values to the left. This gives us two new hyperbolic slices to exclude, resulting in a narrow slanted hourglass of remaining candidates for the median. (f) Finally, we use a pairwise selection network to find the median of the values that remain.

networks on CPU and GPU in such a way as to perform each swap in as few *cycles* as possible.

4.1 Direct compilation for a fixed filter size

One approach to running the sorting networks described so far is to generate the network at compile-time for each kernel size of interest, and then emit specialized code that performs the swaps for that size. This produces code similar to that of popular software packages that use sorting networks for small median filters. This approach scales reasonably up to about 29×29 filters. For an AVX2 implementation on x86 written in Halide [2012] and running on a Skylake X core, this approach requires around 2 cycles per vector swap operation, depending on the filter size.

Unfortunately, the code for this single filter size takes around 10 minutes to compile and produces 519 kilobytes of compiled code per data type. Clearly we cannot continue along this path indefinitely. We want to be able to run larger sorting networks than this, and for floating point median filters (where histogram-based methods do not exist) we want to run sorting networks of arbitrary size.

4.2 Interpreting sorting networks

For sizes above 29×29 we instead compute the sorting network at runtime (cached per size, outside the loop over pixels), and then run it using a sorting network interpreter. A straightforward approach to running a sorting network provided at runtime is to iterate over the swaps in it one by one (Fig. 8, bottom left). This has fairly poor performance (10.4 cycles per vector swap operation on x86 for a 29×29 filter), as the runtime is dominated by loading and storing the values to be swapped.

Instead, our interpreter operates using coarser-grained primitives. We treat this as a compiler problem, and define an abstract sorting machine along with an instruction set for it, and then describe how to lower this instruction set to an efficiently-executed bytecode.

4.2.1 The sorting machine. Our abstract sorting machine operates in an unbounded linear memory space. The input data required by a tile of output pixels has been preloaded into this space starting at index zero.

The instruction set is:

(1) `Sort(start, size, α, β)`
Sorts `size` values in-place starting at address `start`, where only indices α through β in the output need be correct (the others are undefined).
(2) `Merge(start1, size1, start2, size2, α, β)`
Merges two already-sorted non-overlapping sequences, overwriting the first sequence with the smaller `size1` values, and overwriting the second sequence with the larger values. Only indices α through β in the merged sequence need be correct. The others are undefined.
(3) `Copy(src, step, size, dst)`
Copies `size` values starting at `src` with stride `step` to a dense sequence starting at `dst`. This is useful for making copies of values that will be consumed by multiple in-place operations, and also for compacting strided sequences so that they can be sorted or merged. The `step` parameter may be positive or negative.

4.2.2 Limiting maximum instruction size. To lower a list of instructions to bytecode for a high-performance interpreter, it is necessary to concretize the sorting network used for each `Sort` and `Merge` operation (or we have begged the question, and still need a loop over swaps). Our interpreter can only support some finite list of sizes for the `Sort` and `Merge` operations, so we need a way to decompose operations into smaller ones, without going all the way to individual swaps and losing any benefit from using coarser-grained instructions.

See Figure 8 for the construction we use. Large sort and merge operations can be constructed out of a combination of smaller ones,

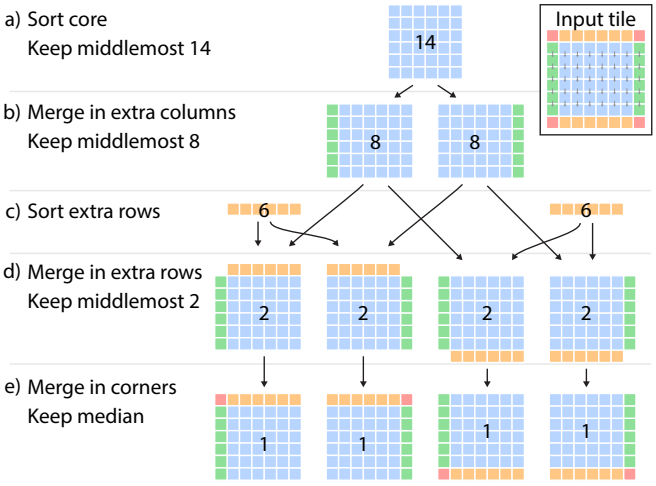


Fig. 5. Computing a 2×2 tile of output for a 7×7 median filter. The total footprint on the input is the union of the footprints of each pixel — an 8×8 square. The inputs needed by all four outputs are in blue. We refer to this as the "core". The extra rows, each used by only two outputs, are in orange, and the extra columns are in green. These columns were already sorted in the same pre-pass that sorted the columns of the core. Finally, there is a single corner value unique to each output in red. Each intermediate set of values is a fully sorted list of the size given by the label, and covering the subset of the input indicated by its picture. Note that as the region covered grows, the number of values in the list shrinks, as we only need to maintain the middlemost $k + 1$, where k is the number of inputs not yet included (See Fig. 3). (a) First we sort the intersection of the footprints of all pixels in the output tile using the procedure in Figure 4. While that figure illustrates the case where we only need the median, here we need the middlemost 14, as each output pixel requires 13 values not yet included in this intersection. (b) Next, we fork this into two to serve the leftmost two outputs and the rightmost two outputs. Into each we merge the appropriate already-sorted extra column and keep the middlemost 8. (c) We then sort the extra rows. (d) These sorted rows are merged into copies of the outputs to step (b), keeping the middlemost two. (e) Finally, we merge in the corner pieces. The total number of swaps performed per output pixel produced is 87.25. If we include the amortized cost of presorting the columns that grows to 93.25. For comparison, using the procedure in Figure 4 without tiling uses 203 swaps, and a pairwise selection network that computes a median of 49 uses 282 swaps. Exploiting separability in these two ways (presorting the columns and sharing work within a small tile) reduces the total work performed by a factor of three.

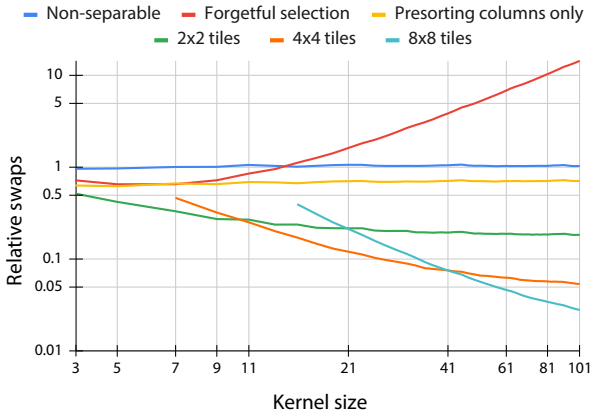


Fig. 6. Swaps performed by various methods, normalized to the swaps done by a pairwise selection network. Lower is better. *Non-separable:* Using the diagonal sorting procedure in Figure 4 without first presorting the columns provides no benefit over a pairwise selection network. *Presorting columns only:* Sorting the columns ahead of time per scanline of output reduces the number of swaps required by roughly 30%. 2×2 *tiles,* 4×4 *tiles,* 8×8 *tiles*: Sharing more work by sorting in tiles (Figs. 5,7) is profitable at any size, with the optimal tile size growing with the footprint of the filter. *Forgetful Selection:* The sorting network used by Salvador et al. [2018] performs well for the sizes used in that work, but scales quadratically with the number of pixels under the kernel footprint.

where the combination is guided by a "template" sorting network. We refer to this as *inflating* the template network.

Our size limit is set such that the values loaded fit into registers on the target platform. For this reason the maximum size for the sort operation is double the maximum size of the merge operation, as the sort operation only needs to load one sequence.

To decompose a `Sort` operation of size n to meet a size limit k, we first break the n inputs into $\lceil n/k \rceil$ leaf sequences of size at most k, and sort each pair of leaves together using sort instructions of size at most $2k$. We then generate a pairwise selection network of size $\lceil n/k \rceil$, omitting the initial pairwise swaps. Each swap (i, j) in this pairwise network becomes a merge operation between two sequences of size k at locations ik and jk. To handle the case where n is not a multiple of k, we implicitly pad the sequence with $+\infty$ and trim any of the operations that would touch these infinities down to smaller ones.

To decompose a `Merge` operation of sequences of size n, m to meet a size limit k, we implicitly pad the first sequence at the beginning with $-\infty$ to make its size a multiple of k, and similarly pad the second sequence with $+\infty$ at the end. We then divide the sequences into leaves of size k, and generate an odd-even merge network for two sequences of size $\lceil n/k \rceil$ and $\lceil m/k \rceil$ to act as a template. The template is then inflated in the same way as in the `Sort` operation.

This procedure can expose opportunities to skip work where part of a sequence to be sorted has already been sorted by an earlier step (this occurs between steps e and f in Figure 4). As we generate the size-limited instruction sequence, we track which sequences are known to be sorted, and avoid emitting instructions that sort any already-sorted sequence.

4.2.3 The bytecode. To lower a size-limited instruction stream to a bytecode, we enumerate all instructions up to the size limit to make an instruction table, and then map each instruction to the entry in the table most capable of executing it (ignoring the `start` parameters, which are passed at runtime). See Figure 8 bottom right for a sketch of the interpreter loop.

4.3 Implementation details

To keep the compiled and interpreted implementations as similar as possible, in both cases we generate the instruction sequence as described in this section. In the compiled case we generate it at

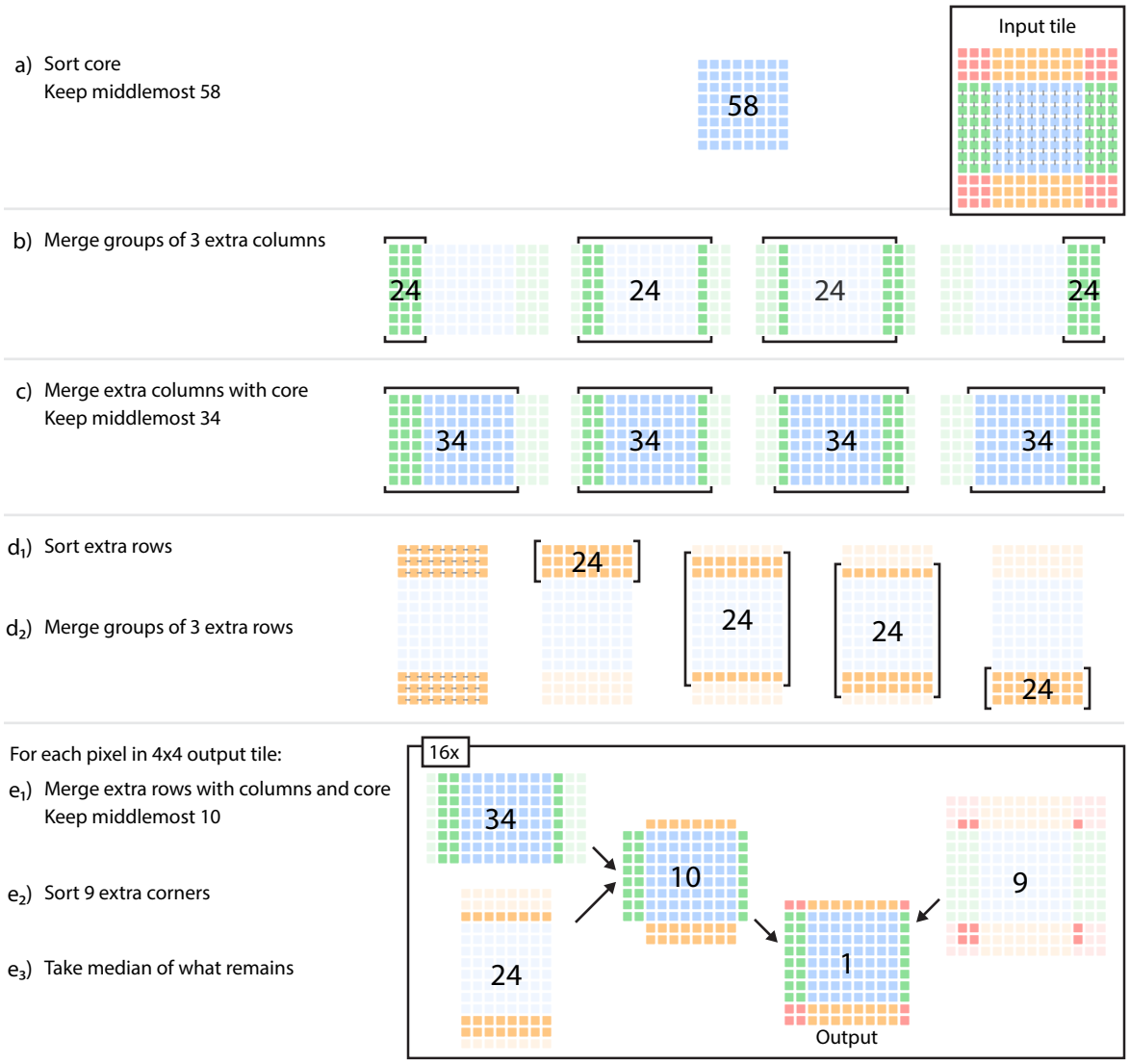
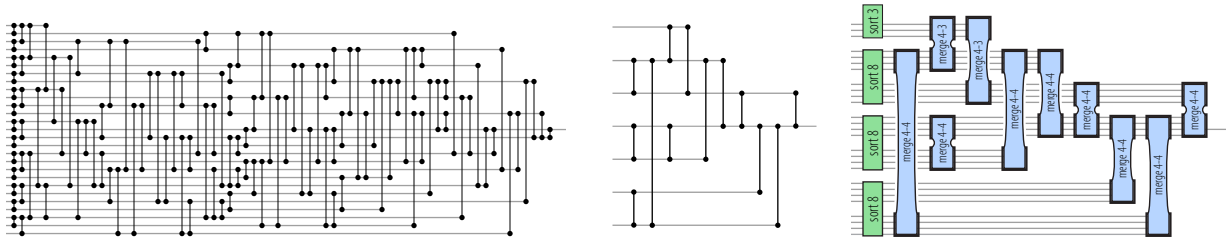


Fig. 7. Computing a 4×4 tile of output for an 11×11 median filter. This figure shows how to generalize the procedure in Figure 5 to arbitrary tile sizes. a) The core (the intersection of the footprints) is an 8×8 square. Each output will be constructed from that core (in blue), three extra columns (in green), three extra rows (in orange), and nine corner values (in red). The number of extra rows and columns is always the tile size minus one. a) First we sort the core as in Figure 5, keeping the middlemost 58, as each output requires 57 values from outside the core. b) We then merge all groups of three adjacent columns. There are four such groups, corresponding to the four columns of output. c) Next we merge these with the sorted core, keeping the middlemost 34 values. d) We then handle the extra rows, first sorting each individually (d_1) and then merging all groups of three adjacent extra rows (d_2). e) With these ingredients prepared, we can construct each of the 16 outputs by first merging each row group from step d_2 with each column-and-core group from step c. From each of these we only need the middlemost 10 values. Then we sort the 9 corners not yet considered and take the median of this with the 10 remaining candidates from the rest of the footprint. Including the cost of presorting the columns, this approach results in an amortized cost of just under 252 swaps per output pixel. For comparison, a pairwise median-finding network on 11×11 inputs uses 1001 swaps. The benefit grows with the filter footprint (Fig. 6).

compile time, and do not limit the maximum instruction size or encode to bytecode, instead just lowering each operation to swaps directly.

Both implementations are metaprogrammed. The compiled implementation is parameterized by the filter size, and the interpreter is parameterized by the maximum instruction size and hence the instruction table. Both are also parameterized by the tile size (Section 3.3). We therefore write both in Halide [Ragan-Kelley et al. 2012][4]. When compiling to the GPU we schedule the interpreter loop on the CPU, and dispatch each instruction as a separate kernel launch. This achieves better occupancy for the small instructions

[4]We use Halide master as of Jan 25 2021



Median-of-27 network constructed from pairwise sorting network
Swaps: **111** Loads/Stores: 222

Median-of-27 network constructed by inflating median-of-7 network
Swaps: 148 Loads/Stores: **105**

```
for (auto pair : network) {
  // 2 loads, 1 swap, 2 stores
  x = data[pair.first];
  y = data[pair.second];
  sort2(x, y);
  data[pair.first] = x;
  data[pair.second] = y;
}
```

```
for (auto op : network) {
  switch (op.code) {
  case Sort_3:
    // 3 loads, 3 swaps, 3 stores
    x = data[op.first];
    y = data[op.first + 1];
    z = data[op.first + 2];
    sort2(x, y);
    sort2(y, z);
    sort2(x, y);
    data[op.first] = x;
    data[op.first + 1] = y;
    data[op.first + 2] = z;
    break;
    ...
  case Sort_8:
    // 8 loads, 19 swaps, 8 stores
    ...
  case Merge_4_3:
    // 7 loads, 8 swaps, 7 stores
    ...
  case Merge_4_4:
    // 8 loads, 9 swaps, 8 stores
    ...
}
```

Fig. 8. Inflating sorting networks for better arithmetic intensity. To compute the median of 27 values, a good network to use (in terms of the number of swaps) is a pairwise selection network (top left). Any specific network like this one can be compiled efficiently to a series of min and max instructions that operate mostly on values in registers (assuming the register allocator does a good job). However if we want to run median filters of arbitrary size using the techniques in this paper, then the specific sorting networks are known at compile time. A straightforward approach to running a dynamically-constructed sorting network is to represent it as a list of pairs, and then iterate over this list applying each swap (bottom left). Implemented in this way, however, for every swap operation we must perform two loads and two stores. On the right we present an alternative way to construct large sorting networks. Starting from a smaller "template" median-of-7 network, we inflate it by a factor of four by replacing each value with a group of four sorted values. The swaps in the template become operations that merge two sorted lists of size four (the operations in blue). At the start of the network we must first sort each group of four inputs, or that sort can be fused into the first merge operation that touches those inputs (the sort-8 links). Unused input capacity (we wanted 27 inputs, not 28) can be trimmed by treating values at the top as ∞, and simplifying the operations that touch those values. Each operation in this network loads at most 8 values into registers, performs some swap operations, and then stores them again. To execute such a network, we run an interpreter loop over a bytecode which encodes the operations to perform (bottom right). For the median-of-27 pictured, this approach uses one third more swaps than the pairwise network on the top left, but executes fewer than half as many memory operations. See Figure 9 for the effect of this trade-off on runtime for a large median filter.

than including the entire interpreter in a single kernel. However this approach uses a very large working set, so we process large images serially in tiles sized so that the working space does not exceed two gigabytes. The CPU interpreter uses a maximum op size of 7 (so that at most 14 values are loaded, avoiding spilling), and the GPU interpreter uses a maximum op size of 12. Larger is slightly better for performance (See Fig. 9), but the instruction table grows quadratically in the limit used. At a limit of 12 there are 238 instructions in the table.

5 EVALUATION

In this section we measure the runtime of the proposed algorithm for a wide range of filter sizes. All benchmarks are run on the 6 megapixel image in Figure 1. Rather than applying a boundary condition, the input is padded on each side by at least the radius of the filter.

CPU Benchmarks are run on an Intel i9-9960X locked to 3.1GHz with hyperthreading disabled. Halide's thread pool was used to parallelize all algorithms using 16 threads. GPU benchmarks are run on an NVIDIA GeForce 2060 RTX. Time taken to transfer data to or from the GPU is not included. Benchmarking is done with Halide's adaptive benchmarking routine with the accuracy parameter set to 0.0001. All benchmarks were run on under Ubuntu 20.04.

For each platform we compare to the best algorithm from the literature for which code is available, vendor libraries from Intel or NVIDIA, and some popular open source packages. See Figure 10 for the results. We also compare to our own implementation of Kim et al. [2015], and provide an estimate of the performance of Salvador et al. [2018].

On CPU, the methods we compare to are:

(1) *Intel performance primitives (IPP)* [2004] has the fastest existing small support median filters on CPU. By inspecting the hot loops in a profiler, we infer that it uses a sorting network for 3×3 and 5×5 filters, and then switches to either histogram or sorting-based methods depending on the data type. We

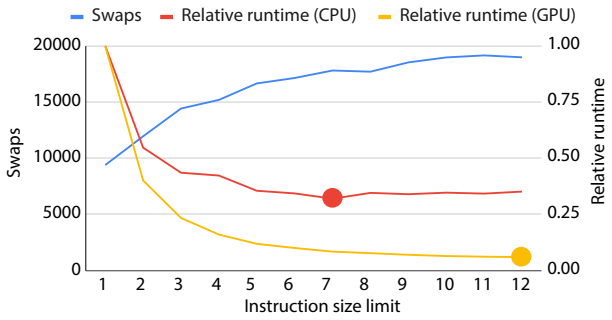


Fig. 9. Performance of an interpreted 101×101 median filter using 8×8 tiles in terms of swaps, CPU runtime, and GPU runtime, as a function of the instruction size limit. Lower is better. Runtimes are normalized to the worst-case performance, which occurs when the interpreter performs one swap at a time (Fig. 8, bottom left). As the maximum instruction size of our interpreter grows, runtime decreases even as swaps increase, because the larger instructions accomplish more useful work per value loaded and stored. Our implementations use the limits indicated by the filled circles. On CPU (using AVX2), the minimum runtime occurs just before values start spilling to the stack. On GPU, the minimum occurs above the range tested. An interpreter that supports all possible sorting and merging operations up to size 12 already requires 238 instructions and takes 10 minutes to compile, so we stop here.

parallelize IPP by slicing up the image into horizontal strips. We use IPP version 2020.4.304.

(2) *CTMF* is the constant-time median filter of Perreault et al. [2007]. For 16-bit integers we use the code as generalized by EBImage [Pau et al. 2010]. We further modify it to use AVX2 instructions and parallelize over vertical strips. This modified implementation is included in supplemental material.

(3) *OpenCV* [Bradski 2000] uses sorting networks for 3×3 and 5×5 filters, and CTMF for 8-bit types at larger sizes. It does not support median filters of larger sizes for other types. We parallelize OpenCV's implementation in the same way as IPP. We use OpenCV master as of Jan 3 2021.

(4) *Kim* et al. [2015] uses a circular buffer of presorted rows and scans down the image in vertical strips. It does not otherwise reuse work within tiles. We compare to our own implementation in Halide, tuned to maximize performance on our benchmarking machine. The code is included in supplemental material.

For 8-bit integers, our method is the fastest up to size 25, at which point O(1) histogram-based methods take over. For higher-precision types, our method is the fastest across all sizes tested. For floating point types at sizes 7 and above, we are more than an order of magnitude faster than the best alternative. Interpreter overhead is below 1.5× for all types starting at 17×17 filters.

On GPU we test against:

(1) *NVIDIA Performance Primitives (NPP)* [2020] is NVIDIA's answer to IPP. It supports median filters for all types and sizes, however it produces incorrect output above 15×15 for 8-bit integers, so we exclude those data points. The methods used include radixselect, quickselect, and sorting networks. We use NPP version 10.1.243-3.

(2) *OpenCV* also contains a CUDA implementation of a median filter as a "contrib" module, using Green's $O(1)$ algorithm [Green 2017]. It supports 8-bit integers only.

(3) *ArrayFire* [Yalamanchili et al. 2015] contains CUDA median filters for sizes up to 15×15 based on forgetful selection networks. As expected, these work well at small sizes, but scale poorly. We use ArrayFire master as of Dec 17 2020.

(4) *Salvador* et al. [2018] do not provide code for their sorting-network-based algorithm, however they report results for 3×3, 5×5, and 7×7 filters on a GTX 1060, giving the same performance for all types. We multiply their reported kernel-only throughputs by 1.5 (the ratio of the number of cuda cores in our GPU vs theirs) to provide an estimate of performance on our RTX 2060.

Our method is the fastest median filter on the GPU up to size 33 for 8-bit, and at all sizes for higher-precision types. For floating point types and kernel sizes above 7×7, we are at least 50× faster than the next best alternative. The interpreter overhead for the GPU implementation is somewhat higher. At 29×29 it is $1 - 4\times$ slower than our compiled version, depending on the type.

Note that the absolute performance on the GPU is worse than the CPU at larger sizes. Partially this is simply because we're using a high-end CPU and a mid-range GPU, but the larger factor is that once the working set no longer fits into the register file, no method comes close to achieving peak compute performance on the GPU, as all methods are bottlenecked on the memory subsystem. The working set for a median filter is large relative to the amount of work to perform.

6 CONCLUSION

This work demonstrates how to use sorting networks to compute median filters quickly at arbitrary sizes. The key idea was to factor the sorting task so that most of the work can be shared between pixels in the same row or tile. This generalizes prior approaches along these lines beyond small-support filters. We describe how to generate high-performance implementations of these filters at arbitrary sizes using a novel sorting-specific instruction set and interpreter. We found this problem-specific-compiler approach very productive – reifying the sequence of operations to perform made it easier to debug, display, and analyze them. We recommend this approach for other problems.

6.1 Limitations

There are two main limitations of this work to keep in mind. First, compared to the simplicity of the constant-time median filter [Perreault and Hébert 2007], our method is complex to describe and to implement. To use the best-performing method in Figure 10 in all cases requires tens of minutes of compile time and 134 megabytes of compiled code (8.4 megabytes if only CPU implementations are needed). In practical deployment, a trade-off will need to be made between performance and code size.

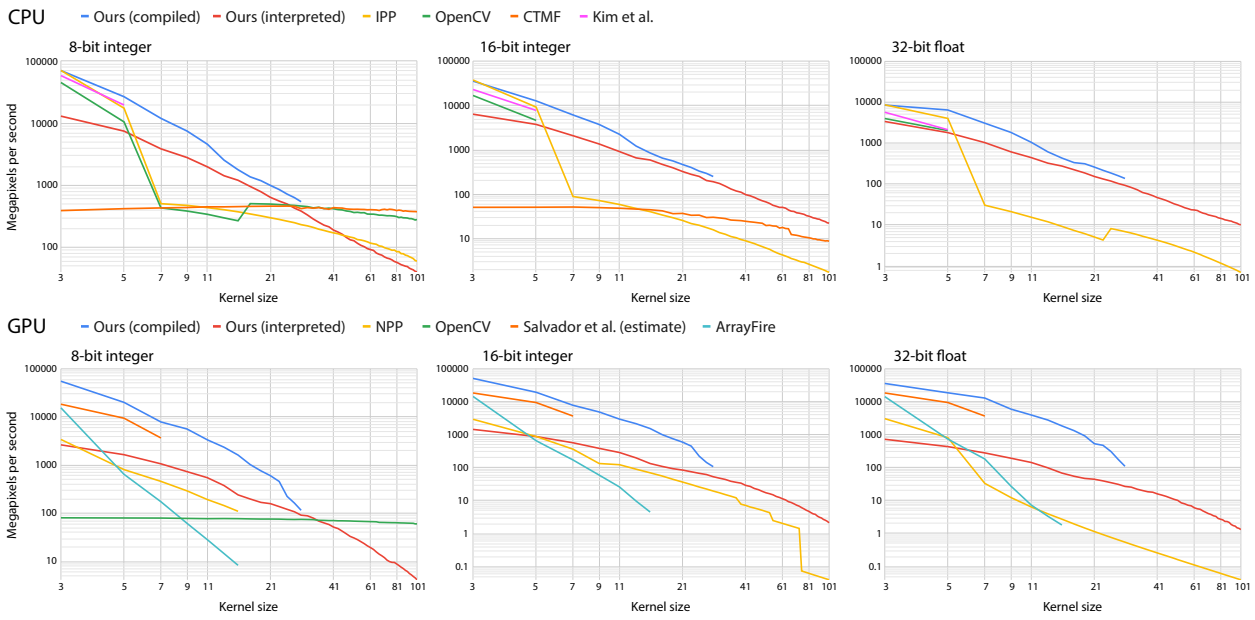


Fig. 10. Performance of the methods proposed in this paper in terms of megapixels per second. Higher is better. We compare to vendor libraries, popular open source packages, and the best-performing related work. Our method is often more than an order of magnitude faster than the best alternative.

Second, we focused on square footprints. In the supplement we provide a generalization to arbitrary footprints, albeit at a lesser gain due to its reduced separability.

6.2 Future work

There are various ways the algorithm as described could be further improved. First, in some stages of the algorithm there is more work that could be shared to further reduce swap count. For example, in Figure 7 step b, adjacent column groups merge overlapping subsets of additional columns.

Second, it is possible to improve our primitive sorting networks, by scaling superoptimization to larger sizes, and *above* that regime by using newer work by Karpiński and Piotrów [2019], who describe smaller selection networks than the pairwise one recommended by Codish and Zazon-Ivry [2010].

Finally, we have described compiling for a range of specific filter sizes ahead of time, and using a high-performance interpreter for other sizes. A possible third approach is light-weight JIT-compilation of a network (for example using Enoki [Jakob 2019]). This may ameliorate some of the code size issues with the GPU implementation.

The median filter is widely used, yet research on speeding it up has slowed to a trickle, perhaps because histogram-based methods seemed to have solved the problem. However, as graphics finishes its migration away from 8-bit pixel data types, we once again have no fast way to run median filters of non-trivial size. In Photoshop, for example, all filters based on the median filter are simply disabled for floating-point images. We argue in this paper that generalizing sorting-network-based approaches to larger sizes is an effective way to attack this problem. We hope that this inspires a renewed interest in fast ordinal filters. We also hope that the runtime improvement we bring to median filters applied to *all* data-types makes it easier for more applications to exploit the many useful properties of the median.

ACKNOWLEDGMENTS

Thanks to Florian Kainz, Shoaib Kamil, Elena Adams, Jiawen Chen, Kevin Wampler, and Sylvain Paris for their assistance, advice, and feedback. Thanks also to the anonymous reviewers for their helpful comments and suggestions.

REFERENCES

Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 307–314.
G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).
Michael Codish and Moshe Zazon-Ivry. 2010. Pairwise cardinality networks. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 154–172.
Derry Fitzgerald. 2010. Harmonic/percussive separation using median filtering. In *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, Vol. 13.
William T Freeman. 1988. Median filter for reconstructing missing color samples. US Patent 4,724,395.
Oded Green. 2017. Efficient scalable median filtering using histogram-based operations. *IEEE Transactions on Image Processing* 27, 5 (2017), 2217–2228.
Wenzel Jakob. 2019. Enoki: structured vectorization and differentiation on modern processor architectures. https://github.com/mitsuba-renderer/enoki.
Michał Karpiński and Marek Piotrów. 2019. Encoding cardinality constraints using multiway merge selection networks. *Constraints* 24, 3-4 (2019), 234–251.
Michael Kass and Justin Solomon. 2010. Smoothed local histogram filters. In *ACM SIGGRAPH 2010 papers*. 1–10.
Minsik Kim, Deokho Kim, Minyong Sung, and Won Woo Ro. 2015. An accelerated separable median filter with sorting networks. In *2015 IEEE International Conference on Image Processing (ICIP)*. IEEE, 803–807.
Donald E. Knuth. 1998. *The Art of Computer Programming* (3rd ed.). Fundamental Algorithms, Vol. 1. Addison Wesley Longman Publishing Co., Inc. (book).
Morgan McGuire. 2008. A Fast, Small-Radius GPU Median Filter. In *Published in ShaderX6*. https://casual-effects.com/research/McGuire2008Median/index.html ShaderX6.

NVIDIA. 2020. NVIDIA Performance Primitives. https://developer.nvidia.com/NPP.
Ian Parberry. 1992. The pairwise sorting network. *Parallel Processing Letters* 2, 02n03 (1992), 205–211.
Grégoire Pau, Florian Fuchs, Oleg Sklyar, Michael Boutros, and Wolfgang Huber. 2010. EBImage—an R package for image processing with applications to cellular phenotypes. *Bioinformatics* 26, 7 (03 2010), 979–981. https://doi.org/10.1093/bioinformatics/btq046 arXiv:https://academic.oup.com/bioinformatics/article-pdf/26/7/979/559050/btq046.pdf
Simon Perreault and Patrick Hébert. 2007. Median filtering in constant time. *IEEE transactions on image processing* 16, 9 (2007), 2389–2394.
Gilles Perrot, Stéphane Domas, and Raphaël Couturier. 2014. Fine-tuned High-speed Implementation of a GPU-based Median Filter. *Journal of Signal Processing Systems* 75, 3 (2014), 185–190.
Fatih Porikli. 2005. Integral histogram: A fast way to extract histograms in cartesian spaces. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, Vol. 1. IEEE, 829–836.
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.
Gabriel Salvador, Juan M Chau, Jorge Quesada, and Cesar Carranza. 2018. Efficient GPU-based implementation of the median filter based on a multi-pixel-per-thread framework. In *2018 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI)*. IEEE, 121–124.
Ricardo M Sánchez and Paul A Rodríguez. 2013. Highly parallelable bidimensional median filter for modern parallel programming models. *Journal of Signal Processing Systems* 71, 3 (2013), 221–235.
E. Stewart. 2004. *Intel Integrated Performance Primitives: How to Optimize Software Applications Using Intel IPP*. Intel Press.
Deqing Sun, Stefan Roth, and Michael J Black. 2010. Secrets of optical flow estimation and their principles. In *2010 IEEE computer society conference on computer vision and pattern recognition*. IEEE, 2432–2439.
JW Tukey. 1974. Nonlinear (nonsuperposable) methods for smoothing data. *Proc. Cong. Rec. EASCOM'74* (1974), 673–681.
Vaibhav Vavilala. 2019. Light pruning on Toy Story 4. In *ACM SIGGRAPH 2019 Talks*. 1–2.
Frederick M Waltz. 1989. Fast implementation of ranked filters on general-purpose image processing hardware. In *Automated Inspection and High-Speed Vision Architectures II*, Vol. 1004. International Society for Optics and Photonics, 25–32.
Frederick M Waltz, Ralf Hack, and Bruce G Batchelor. 1998. Fast efficient algorithms for 3x3 ranked filters using finite-state machines. In *Machine Vision Systems for Inspection and Metrology VII*, Vol. 3521. International Society for Optics and Photonics, 278–287.
Ben Weiss. 2006. Fast median and bilateral filtering. In *ACM SIGGRAPH 2006 Papers*. 519–526.
Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm, and John Melonakos. 2015. ArrayFire - A high performance software library for parallel computing with an easy-to-use API. https://github.com/arrayfire/arrayfire
Qingxiong Yang, Narendra Ahuja, and Kar-Han Tan. 2015. Constant time median and bilateral filtering. *International Journal of Computer Vision* 112, 3 (2015), 307–318.
Jure Žbontar and Yann LeCun. 2016. Stereo matching by training a convolutional neural network to compare image patches. *The journal of machine learning research* 17, 1 (2016), 2287–2318.

A THE ENTIRE ALGORITHM

The algorithm proposed by this paper is best explained visually using the figures above. For completeness we also include a text description here. For a filter of diameter d, computing the output in $T_w \times T_h$ tiles, the entire algorithm is as follows:

(1) Let $C_w = (d - T_w + 1)$ and $C_h = (d - T_h + 1)$. This is the size of the *core*, which is the region of the input shared by all outputs within one tile. It is the region in blue in Figure 7.
(2) For each strip of T_h scanlines of the output, gather columns of C_h input pixels centered on the T_h pixels in the scanline, and sort each.
(3) From these sorted columns, gather overlapping windows of size $C_w \times C_h$. Each of these forms a core shared by one tile of output. We want to extract the middlemost $k + 1$ values, where $k = d^2 - C_w C_h$. Counting from zero, the first index of interest is $\alpha = C_w C_h - \frac{d^2+1}{2}$ and the last is $\beta = \frac{d^2-1}{2}$.
(4) Sort the core along the rows from left to right. The columns remain sorted (see the supplement for a proof).
(5) At this point, a value at position i, j in the core (counting from zero) is greater than or equal to the $g = (i + 1)(j + 1)$ values above and to the left (including itself), and less than or equal to $l = (C_w - j)(C_h - i)$ values below and to the right. Exclude this value from further consideration if $g > \beta$ or $l > \beta$. This excludes two hyperbolic slices at the top left and bottom right (Fig. 4c). During the previous step, anticipate this exclusion to reduce the sorting networks to cheaper selection networks, as we don't need the outputs that will be excluded at this step.
(6) Sort the values that remain along diagonals up and to the right. The rows and columns both remain sorted (again, see the supplement for a proof). For diagonals that contain a mix of live and already-excluded values, care must be taken to treat already-excluded values as $\pm\infty$ in this sort, shuffling them to the top right or bottom left of the diagonal. If $C_w > C_h$, we must transpose before this step, or equivalently sort down and to the left and adjust the rest of the algorithm accordingly.
(7) At this point, each value in the core is greater than or equal to every other value in the shaded directions ◐ and less than all values in the shaded directions ◑. Count these values and compare the counts to β, as before, thus excluding two new hyperbolic slices. In the previous step, anticipate this exclusion to reduce the sorting networks to cheaper selection networks. Don't bother sorting diagonals in which no values were excluded.
(8) Apply a pairwise selection network to extract the middlemost $k + 1$ values of what remains. We refer to this as the *sorted core*.
(9) For each column of the tile, gather the $T_w - 1$ nearest sorted input columns of height C_h that were not already included in the core and merge them into a single sorted list. These are in green in Figure 7. Into each of these merge the sorted core, keeping the middlemost $dC_h + 1$ values.
(10) Gather $(T_h - 1)$ rows of width C_w from above and below the core, and sort each. These are in orange in Figure 7.
(11) For each row of the tile, gather the $T_h - 1$ sorted rows from the previous step that lie within the intersection of the footprints across this row, but were not already included in the sorted core. Merge them.
(12) For each output pixel within the tile, merge the appropriate bundle of sorted rows from step 11 with the sorted columns and core from step 9, keeping the middlemost $(T_w - 1)(T_h - 1) + 1$. Then gather and sort the $(T_w - 1)(T_h - 1)$ values covered by the footprint of this output not already included in the previous steps. These are the values in red in Figure 7. Merge these two lists, keeping only the median. The bulk of the runtime is in this step, because it is the only piece of work not shared between multiple output pixels.