Implementing an Efficient Continuous 3D Median Filter

Ashwin Chetty

Introduction

The median filter is a foundational technique in the analysis of multi-dimensional raster data. It is particularly effective at removing speckled, salt-and-pepper noise from images, and is preferred to a Gaussian blur when the edges of an image should be preserved.

However, median filters are notoriously expensive to compute, as the number of pixels for which the median must be calculated increases cubically with respect the radius of the median window. A naïve implementation might iterate through each pixel, and, for each pixel, compute the median of pixels in the radius-R cube around it.

As an optimization, histogram-based approaches are generally used in image-processing applications. The median of a histogram can be computed quickly by summing up the bin values, and it is therefore possible to calculate the median of a histogram of data values in time linear to the number of elements contained within it, where the linear dependency only involves inserting the input values of the window into the histogram. Other optimizations are possible with histograms, and, in fact, Perreault et al. describes a median filtering algorithm with a time complexity of $O(1)$ with respect to window radius [1].

However, time complexity generally increases quickly with the number of bins. A median filter for an image can typically be computed with four passes of a 256-binned median filter, once for each color channel. But histogram approaches become infeasible when applied to data with greater bit depths (ie. 16-bit integers), such as those commonly used in scientific computing. Additional transformations might also be required when working with floating-point data, such as the ordinal transform described by Weiss [2].

Huang describes a histogram-based approach that is able to reduce the time-complexity a 2D median filter from $O(n^2)$ to $O(n)$ with a clever strategy for moving the median window. The window zig-zags across the image, and, for each pixel, $2R + 1$ pixels are subtracted from the histogram, and $2R + 1$ pixels are added [3].

Perreault et al. describe another histogram approach that boasts $O(1)$ time complexity in the window radius. Their algorithm is based on the principle that it is efficient to compute the median of a summation of multiple histograms. In a 2D input image of size $W \times W$, Perreault maintains a list of W columnar histograms, each of size $2R + 1$. At the start of the algorithm, each histogram is initialized with the top-most values in the input array. With each pixel transition (ie. with each movement of the window), only these columnar histograms are updated, and, throughout the course of the algorithm, they each

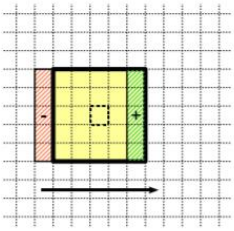


Fig. 1: In Huang's $O(n)$ algorithm, $2r + 1$ pixels must be added to and subtracted from the kernel's histogram when moving from one pixel to the next. In this figure, $r = 2$.

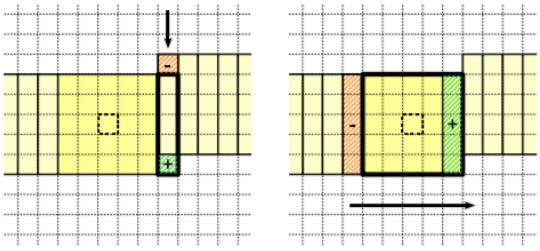


Fig. 2: The two steps of the proposed algorithm. (a) The column histogram to the right is moved down one row by adding one pixel and subtracting another. (b) The kernel histogram is updated by adding the modified column histogram and subtracting the leftmost one.

Images from the paper by Perreault et al. [1]

individually sink to the bottom of the image. The median of the window is then calculated as the median of the summation (alternatively, the set union) of these component columns. Each horizontal window movement requires only a single insertion into the histogram and a single deletion. The next pixel is added into the histogram and the top pixel is removed. At the edges of the image, the window moves vertically before wrapping back in the other direction. Each vertical window movement requires $2R + 1$ insertions and deletions—that is, a single insertion and deletion in each of $2R + 1$ histograms. Total, for an image of size $W \times W$, W^2 insertion and deletion operations are performed, one per pixel.

This paper discusses several approaches to attempt to optimize a median filter in three dimensions, borrowing some of the ideas of Huang and Perreault, while avoiding histograms, which do not generalize well to arbitrary data types. Three algorithms are introduced, and their relative efficiencies are discussed. Of these three algorithms, the one based on Perrault's algorithm is markedly slower than the naïve algorithm, although it can claim better asymptotic complexity. The other two are faster. Testing was done on a Microsoft Surface Pro 3 with 8GB RAM and a dual-core 1.7GHz Intel i7 processor, with 128KB L1 cache, 512KB L2 cache, and 4MB L3 cache. All algorithms were compiled with GNU g++ with the -O3 flag set.

Notation

We consider an input 3D image I, with dimensions $W \times L \times H$, indexed by x, y, z from fastest to slowest. We attempt to calculate a median filter with a radius R and window diameter $N \coloneqq 2R + 1$. The output is another image I', with the same dimension. The borders of the image, where the median filter cannot be calculated, are set to 0.

Algorithm 0: Naïve Median Filter

The Quickselect algorithm is a variant of Quicksort which allows for the calculation of the median in average $O(n)$ time and worst-case $O(n^2)$ time. Quickselect works by, at each iteration, arbitrarily choosing a pivot element and then rearranging all other elements so that the pivot is greater than all elements to its left and less than all elements to its right. While Quicksort must recursively sort each of these partitions, Quickselect need only recurse on the partition which contains the median element. Assuming that the array is partitioned in half with each iteration, approximately $N + \frac{N}{2} + \frac{N}{4} + \cdots = 2N$ copy operations are required. Since Quickselect modifies the input array in-place, the data must be copied into a temporary buffer before the median is found.

As a baseline, we consider a naïve median filter implementation. This implementation scans through the image in raster order. At each pixel, an $N \times N \times N$ region around the image is copied into a temporary buffer. Quickselect is run on this buffer, and the output pixel is set to this result. Surprisingly, this algorithm runs in subcubic time with respect to window radius. This is likely due to architecture-level parallelism and cache-related artefacts, as data cache misses contribute more to the time complexity than the number of arithmetic operations performed.

Algorithm 1: Perreault

The first algorithm is inspired by the traversal strategy described by Perreault et al. However, instead of using Histograms, we use instrumented Red-Black Binary Trees. Red-Black Trees promise log(n) insertion and worst-case replacement operations at the cost of an increased constant factor (involved in rebalancing the tree). It is likely that the cost of rebalancing for each pixel outweighs the practical cost of searching, and it would be prudent to test the algorithm without rebalancing the tree on each operation. Binary trees allow for quick in-order traversal, and allow for the calculation of the median in log(n) time.

To allow for a log(n) median calculation, we maintain a list of the number of children at each node, which is updated with each rebalancing operation.

In the 3D input image I, we maintain $W \times L$ columnar trees, analogous to the histograms described by Perreault. With each lateral window movement, N trees are updated, and one element is inserted into (and deleted from) each. With each vertical window movement downward, N^2 trees are updated. The topmost values are removed from each tree, and values at the immediately-lower vertical index are inserted. Like in Perreault's algorithm, the window zig-zags across the entire image, completing a slice on the $x - y$ plane before moving downward.

Perreault's algorithm was based on the fact that medians of the sums of histograms are easy to compute. This proposed algorithm is based on the fact that medians of sorted lists are easy to compute. The median of a single sorted list can be found in constant time. The median of n sorted lists, each of length m, can be computed in $n \cdot \log(m)$ time. The median of N sorted lists, each of length N, can be computed in $N \cdot \log N$ time.

Computation of the Median

There exists an $O(\log n)$ algorithm to compute the median of many sorted lists.

As notation, for ordered arrays x, y, let $x \oplus y$ be their concatenation. Further, let $med(x) \equiv \overline{x}$ be its median. Assume that each array has the same length, N, and that N is odd, with $N = 2R + 1$. Consider a group of arrays, $A = \{A_1 = (a_{11}, a_{12}, \dots, a_{1m}), \dots, A_n = (a_{n1}, a_{n2}, \dots, a_{nm})$. The median of this group of arrays lies between the minimum and maximum individual median. Denote the total median $M = \overline{A_1 \oplus A_2 \oplus \dots \oplus A_n}$, and the minimum and maximum medians $M_{min} = \min_{i \in \{1..n\}} \overline{A_i}$ and $M_{max} = \max_{i \in \{1..n\}} \overline{A_i}$.

Lemma 3: If $|A| > 2$, then $M_{min} < M < M_{max}$.

Lemma 4: If $|A| = 2$, then $M_{min} \leq M \leq M_{max}$.

We may use this fact to efficiently compute the median. To compute the total median M, we begin with a list of medians. At each stage of the algorithm, we find the minimum and maximum medians. We discard elements less than or equal to the minimum median, and elements greater than or equal to the maximum median, making sure to discard the same number of elements from each array. We repeat this procedure until we are left with a single element. At each iteration, at least one array is halved in size. Thus, the running time is $O(M \cdot \log N)$, if we begin with M arrays.

This algorithm is naturally extendable to ordered binary trees, and can be implemented with the same average time complexity. For a particular window of size $N \times N \times N$, it is necessary to compute the median of N^2 trees, each of length N. Thus, the median for a particular pixel can be computed in time $O(N^2 \log N)$.

Median calculation using this technique proved to be highly inefficient. A more thorough description of its implementation is provided as an appendix.

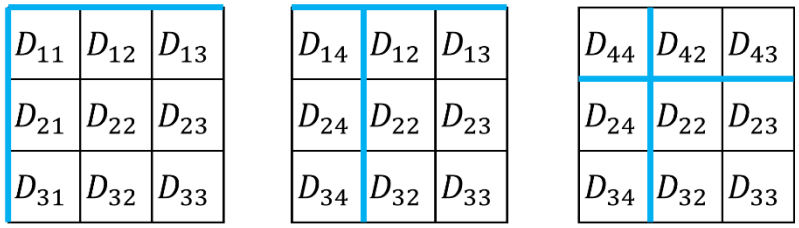


Figure 1 – The window is stored in an array and is updated with each window movement. A naïve implementation might shift each element over in memory to reflect its new position in the window. Instead, the position of the origin itself changes, and values wrap around. Left: initial position of the window. Middle: the window moves along the x-axis. Elements D_{11}, D_{21}, and D_{31}, which are now to the left of the window, are removed, and are replaced with D_{14}, D_{24}, and D_{34}, which have just entered the window to the right. Right: the window moves down the y-axis, the window origin moves downward, and the incoming data values are added immediately before the origin position.

Window Movement

The efficiency of Perreault's algorithm is found in its ability to reuse median calculations between adjacent voxels. To maximize the number of shared voxels per calculation, we move the window in a zig-zag path across the image.

Lateral Movement

We maintain an $N \times N$ array of Trees W, which represents the current window of which the median will be computed. To minimize the number of memory copy operations required, the origin position within the window shifts with the window. Figure 3 depicts the contents of the window across two window movements, where D_{ij} denotes the tree that resides in column (i, j). For each lateral window movement, N copy operations are performed, in relatively contiguous regions of memory.

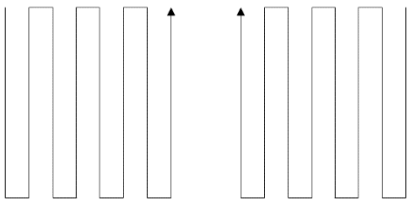


Figure 2 – The median window zig-zags across the image. After it completing a horizontal slice, the window moves down in the z-axis and traverses the next slice in the reverse direction.

The incoming trees might then be shifted downward to include the entering elements. This step, in general, requires exactly one tree-insertion operation per pixel.

Vertical Movement

When a tree must be shifted downward—either during lateral or vertical window movement—a similar sliding origin technique is used. All nodes are stored in contiguous memory. When a new node enters the window, exactly one element of the array is replaced, and the tree is rebalanced.

Efficiency

This algorithm boasted good asymptotic complexity, but turned out to be the least efficient of those tested on radii up to 30. This median-calculation algorithm turned out to be very slow, with the majority of time being spent on the in-order traversal of the binary trees. It might be possible to transform binary trees into sorted arrays before finding their median—however, it is much more efficient to simply ignore the tree structure and operate on the raw data using an algorithm like Quickselect. One potential improvement might involve the use of the strategy employed by Blum, Floyd, Pratt, Rivest and Tarjan[4], which proceeds by recursively

computing the median of an array's subarrays, and could make use of the fact that the median of each tree is initially quick to compute.

The use of Quickselect with Perreault's traversal strategy yielded faster results with radii up to 30, at the apparent cost of an increased asymptotic complexity. Indeed, this result is somewhat misplaced, as there is no reason to maintain all of Perreault's histograms if Quickselect is going to be used to compute the median.

Algorithm 2: Huang

We may consider a tree-based implementation of Huang's original algorithm. Here, instead of maintaining $W \times W$ trees of size N, we only maintain a single tree of size N^3. At each window movement, we remove the preceding N^2 pixels and add the values of the succeeding N^2 pixels into the window, and rebalance the tree at each step. We then compute the median of this tree in $\log N^3$ time, and set the output pixel to this value. To reduce the number of required memory operations, as before, a 3D sliding origin technique is used, so that each **replace** operation only modifies N^2 elements in memory. To improve cache efficiency, all tree nodes are stored in contiguous memory.

This algorithm actually provides us with the best asymptotic efficiency of any of the algorithms tested. However, the constant factor is large, and the algorithm may be better served without the overhead of Red-Black rebalancing.

Algorithm 3: Huang + Quickselect

Here, we used the same zig-zag patterning. At each pixel, we replace the preceding N^2 pixels with the succeeding N^2 pixels in an array, copy this array into a temporary buffer, and perform Quickselect on this buffer. We set the output pixel to this calculated median.

The use of Quickselect as opposed to the tree algorithm in fact reduces the time required by almost a factor of 3, and makes it highly competitive with the histogram algorithm used in the scientific visualization toolkit Teem. The asymptotic complexity was unsurprisingly the same as with the naïve algorithm.

Results

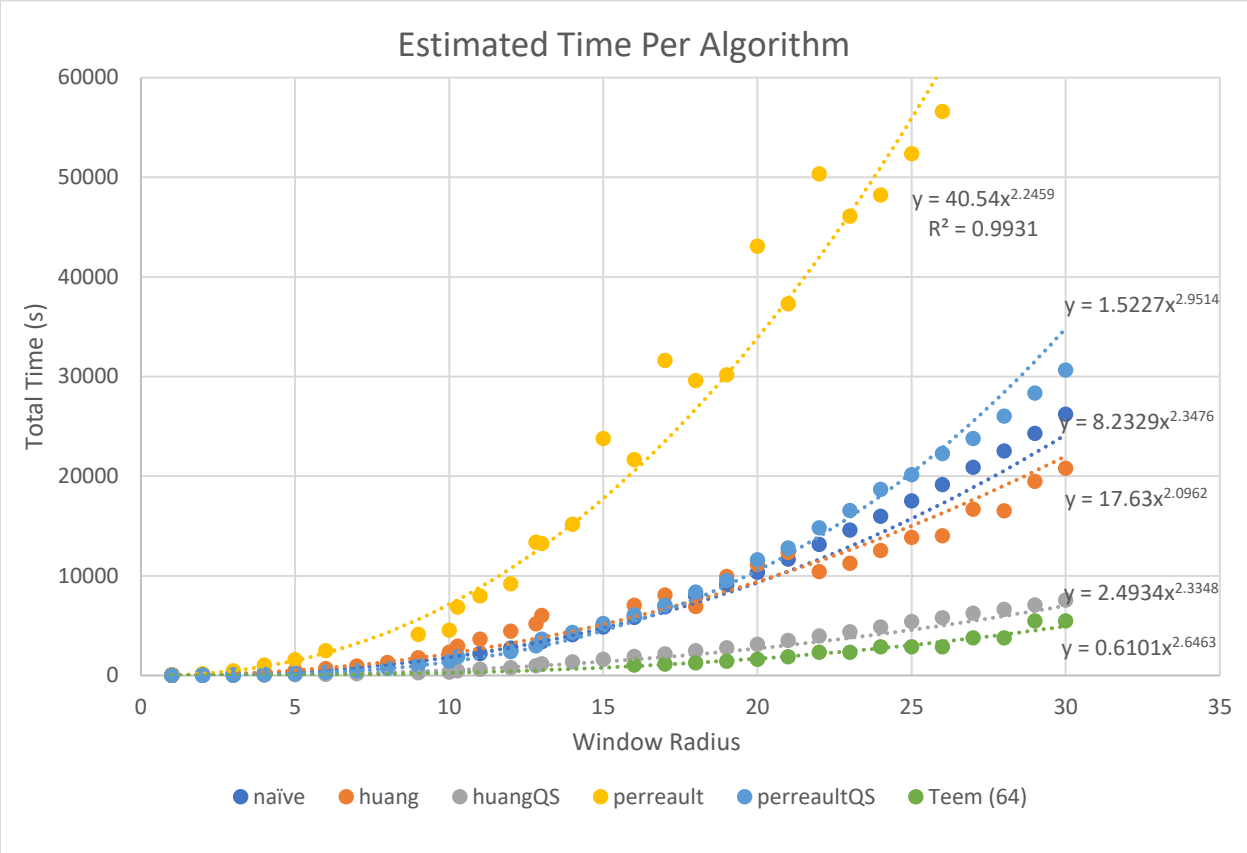
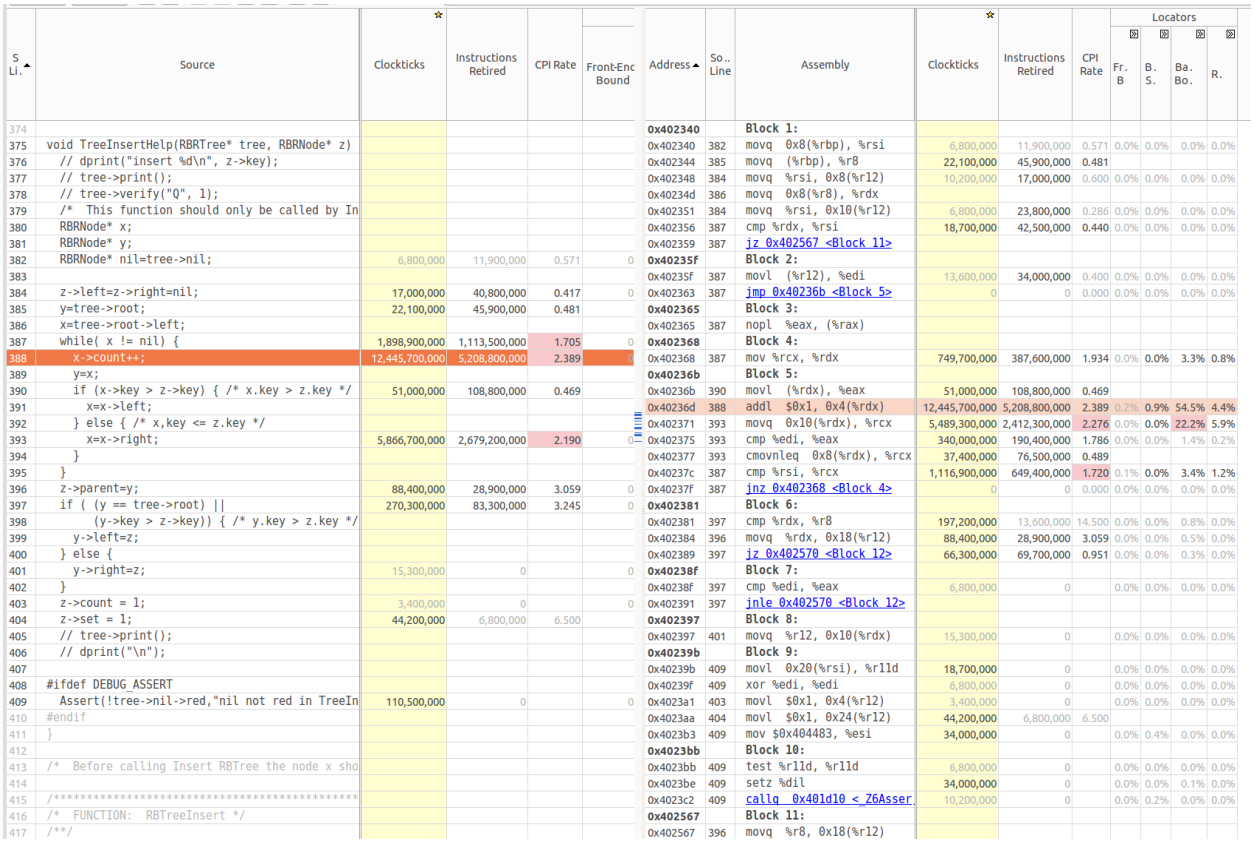


Figure 3: Depiction of the estimated time required to run each algorithm to completion, for window sizes between 1 and 30. Regression lines are shown with their equations. All R^2 values were greater than 0.97. In order from top to bottom, exponents are 2.24, 2.95, 2.34, 2.09, 2.33, and 2.64. The last several results of perreault *are omitted.*

Estimated time per algorithm was calculated for each radius by executing each algorithm for 60 seconds, counting the number of output pixels that had been written, and linearly extrapolating a total completion time. Results are shown for each of the algorithms discussed as well as the **unu cmedian** implementation found in the Teem computation toolkit. The time required to run **cmedian** seems to increase approximately linearly with bin size. A bin size of 64 closely matched the efficiency of the Huang-Quicksort implementation; however, a bin size of 64 is likely too low to be used in practice. The median filter was computed for an image of size $496 \times 258 \times 190$ with randomly populated data.

The adapted Perreault's algorithm was the slowest of all algorithms tested. In this algorithm, the most time was spent in the median computation, and, in particular, on the **node.successor()** function when performing an in-order tree traversal.

The adapted Huang's algorithm fared slightly better than the naïve algorithm, and had the best asymptotic efficiency among the algorithms tested. However, it suffered from a large number of cache misses which slowed it down considerably. It suffered from an average CPI of 1.5, compared to 0.525 for Huang+QS. With a window radius of 15, approximately 20% of the execution time was spent performing tree insertion, and 13% performing tree deletion (not including the cost of red-black balancing operations, which was actually rather negligible).



S Li.	Source	Clockticks	Instructions Retired	CPI Rate	Front-End Bound
374					
375	void TreeInsertHelp(RBRTree* tree, RBRNode* z)				
376	// dprint("insert %d\n", z->key);				
377	// tree->print();				
378	// tree->verify("Q", 1);				
379	/* This function should only be called by In				
380	RBRNode* x;				
381	RBRNode* y;				
382	RBRNode* nil=tree->nil;	6,800,000	11,900,000	0.571	0
383					
384	z->left=z->right=nil;	17,000,000	40,800,000	0.417	0
385	y=tree->root;	22,100,000	45,900,000	0.481	
386	x=tree->root->left;				
387	while(x != nil) {	1,898,900,000	1,113,500,000	1.705	0
388	x->count++;	12,445,700,000	5,208,800,000	2.389	0
389	y=x;				
390	if (x->key > z->key) { /* x.key > z.key */	51,000,000	108,800,000	0.469	
391	x=x->left;				
392	} else { /* x,key <= z.key */				
393	x=x->right;	5,866,700,000	2,679,200,000	2.190	0
394	}				
395	}				
396	z->parent=y;	88,400,000	28,900,000	3.059	0
397	if ((y == tree->root) ||	270,300,000	83,300,000	3.245	0
398	(y->key > z->key)) { /* y.key > z.key */				
399	y->left=z;				
400	} else {				
401	y->right=z;	15,300,000	0		0
402	}				
403	z->count = 1;	3,400,000	0		0
404	z->set = 1;	44,200,000	6,800,000	6.500	
405	// tree->print();				
406	// dprint("\n");				
407					
408	#ifdef DEBUG_ASSERT				
409	Assert(!tree->nil->red,"nil not red in TreeIn	110,500,000	0		0
410	#endif				
411	}				
412					
413	/* Before calling Insert RBTree the node x sho				
414					
415	/***				
416	/* FUNCTION: RBTreeInsert */				
417	/**/				

Address	So.. Line	Assembly	Clockticks	Instructions Retired	CPI Rate	Locators: Fr. B	B. S.	Ba. Bo.	R.
0x402340		**Block 1:**							
0x402340	382	movq 0x8(%rbp), %rsi	6,800,000	11,900,000	0.571	0.0%	0.0%	0.0%	0.0%
0x402344	385	movq (%rbp), %r8	22,100,000	45,900,000	0.481				
0x402348	384	movq %rsi, 0x8(%r12)	10,200,000	17,000,000	0.600	0.0%	0.0%	0.0%	0.0%
0x40234d	386	movq 0x8(%r8), %rdx							
0x402351	384	movq %rsi, 0x10(%r12)	6,800,000	23,800,000	0.286	0.0%	0.0%	0.0%	0.0%
0x402356	387	cmp %rdx, %rsi	18,700,000	42,500,000	0.440	0.0%	0.0%	0.0%	0.0%
0x402359	387	jz 0x402567 <Block 11>							
0x40235f		**Block 2:**							
0x40235f	387	movl (%r12), %edi	13,600,000	34,000,000	0.400	0.0%	0.0%	0.0%	0.0%
0x402363	387	jmp 0x40236b <Block 5>	0	0	0.000	0.0%	0.0%	0.0%	0.0%
0x402365		**Block 3:**							
0x402365	387	nopl %eax, (%rax)							
0x402368		**Block 4:**							
0x402368	387	mov %rcx, %rdx	749,700,000	387,600,000	1.934	0.0%	0.0%	3.3%	0.8%
0x40236b		**Block 5:**							
0x40236b	390	movl (%rdx), %eax	51,000,000	108,800,000	0.469				
0x40236d	388	addl $0x1, 0x4(%rdx)	12,445,700,000	5,208,800,000	2.389	0.2%	0.9%	54.5%	4.4%
0x402371	393	movq 0x10(%rdx), %rcx	5,489,300,000	2,412,300,000	2.276	0.0%	0.0%	22.2%	5.9%
0x402375	393	cmp %edi, %eax	340,000,000	190,400,000	1.786	0.0%	0.0%	1.4%	0.2%
0x402377	393	cmovnleq 0x8(%rdx), %rcx	37,400,000	76,500,000	0.489				
0x40237c	387	cmp %rsi, %rcx	1,116,900,000	649,400,000	1.720	0.1%	0.0%	3.4%	1.2%
0x40237f	387	jnz 0x402368 <Block 4>	0	0	0.000	0.0%	0.0%	0.0%	0.0%
0x402381		**Block 6:**							
0x402381	397	cmp %rdx, %r8	197,200,000	13,600,000	14.500	0.0%	0.0%	0.8%	0.0%
0x402384	396	movq %rdx, 0x18(%r12)	88,400,000	28,900,000	3.059	0.0%	0.0%	0.5%	0.0%
0x402389	397	jz 0x402570 <Block 12>	66,300,000	69,700,000	0.951	0.0%	0.0%	0.3%	0.0%
0x40238f		**Block 7:**							
0x40238f	397	cmp %edi, %eax	6,800,000	0		0.0%	0.0%	0.0%	0.0%
0x402391	397	jnle 0x402570 <Block 12>							
0x402397		**Block 8:**							
0x402397	401	movq %r12, 0x10(%rdx)	15,300,000	0		0.0%	0.0%	0.0%	0.0%
0x40239b		**Block 9:**							
0x40239b	409	movl 0x20(%rsi), %r11d	18,700,000	0		0.0%	0.0%	0.0%	0.0%
0x40239f	409	xor %edi, %edi	6,800,000	0		0.0%	0.0%	0.0%	0.0%
0x4023a1	403	movl $0x1, 0x4(%r12)	3,400,000	0		0.0%	0.0%	0.0%	0.0%
0x4023aa	404	movl $0x1, 0x24(%r12)	44,200,000	6,800,000	6.500				
0x4023b3	409	mov $0x404483, %esi	34,000,000	0		0.0%	0.4%	0.0%	0.0%
0x4023bb		**Block 10:**							
0x4023bb	409	test %r11d, %r11d	6,800,000	0		0.0%	0.0%	0.0%	0.0%
0x4023be	409	setz %dil	34,000,000	0		0.0%	0.0%	0.1%	0.0%
0x4023c2	409	callq 0x401d10 < Z6Asser	10,200,000	0		0.0%	0.2%	0.0%	0.0%
0x402567		**Block 11:**							
0x402567	396	movq %r8, 0x18(%r12)							

Figure 4 – Intel Amplifier profiling output for the Binary Tree Insertion procedure used with Huang's algorithm. We see that the CPI rates are most prominent when member variables of pointer objects are accessed. The line **`x->count++`** *, homologous to* **`addl $0x1, 0x4(%rdx)`**, *suffers a CPI rate of 2.389, and consumes 12,445,700,000 clockticks.*

The program admitted a very high number of cache misses through its execution. Pointer dereferences were the most expensive operations performed, and were the fundamental bottleneck. Tree data structures also unfortunately exhibit essentially random data access patterns which do not lend themselves well to memory access prediction strategies employed by the lower-level architecture. Changing the raster order (ie. traversing the y axis first before the x axis) also did not affect efficiency.

Another strategy might also involve reducing the size of a tree Node as much as possible. In the current implementation, a Node is 52 bytes (with a 4-byte payload). However, it is possible to reduce the size of a Node to only 20 bytes, by eschewing the red-black structure and defining the structure as follows:

```
struct Node{
    int payload;
    int count;
    int parent;  // using integral offsets instead of pointers,
    int left;    // and limiting window radius to 813 voxels
    int right;
};
```

Red-black operations do not accrue much of an overhead (using roughly 3% of the total time of the program). However, assuming random input data, a simple binary search tree will also allow for $\log n$ performance, and maintenance of the red-black property is likely unnecessary.

Unfortunately, it is infeasible to store the tree simply as a length-N^3 list of integers, as preserving the binary tree property would require at least linear time for each replacement operation, which is slower than Quickselect.

Conclusion

Ultimately, Huang's traversal strategy with Quickselect proved the fastest for radii less than 30. Perreault's algorithm was exceedingly slow when instrumented to be used with trees, with the bottleneck involving in-order tree traversal.

The tree-based approach provided a better asymptotic complexity than other methods, but at the cost of an increased constant factor due to its pathological memory-access patterns. If it is possible, however, to reduce the CPI of this algorithm by a factor of three, then it will be highly competitive with the Quickselect algorithm and even with existing histogram approaches.

References

[1] S. Perreault and P. H´ebert. Median filtering in constant time. TIP, pages 2389–2394, 2007.

[2] B. Weiss, “Fast Median and Bilateral Filtering,” ACM Transactions on Graphics (TOG), vol. 25, no. 3, pp. 519–526, 2006.

[3] T. S. Huang, “Two-Dimensional Signal Processing II: Transforms and Median Filters,” 1981, pp. 209–211, Berlin: Springer-Verlag.

[4] Blum, Manuel, et al. "Time bounds for selection." Journal of computer and system sciences 7.4 (1973): 448-461.

Appendix 1: The Slow Median-Calculation Algorithm

The bottleneck in this algorithm is the **Node.successor(*i*)** operation.

Our algorithm requires us to efficiently find the maximum and minimum of a dynamically changing list of medians. We find it useful to keep track of both the *upper median*[1] (ie. the element at index $\frac{n}{2}$), which is greater than or equal to the median as conventionally defined, as well as the *lower median*, ie. the element at index $\frac{n}{2} - 1$, which is always less than the median. At each iteration, we either remove elements less than or equal to the lower median or elements greater than or equal to the upper median. It is necessary to maintain separate lists for both upper and lower medians, while exposing a getMinimum operation for the lower medians and a getMaximum operation for the upper medians, and while still maintaining a linkage between the lower and upper medians for a particular array. When we reduce an array, we must efficiently update the position of its lower median as well as its upper median in these lists.

To do this, we make use of an instrumented data structure called the CorrelatedHeap, which encapsulates a linked minheap and maxheap. Each element in the minheap maintains a pointer to a corresponding element in the maxheap, which itself maintains a pointer to its corresponding minheap element. It exposes the operations:

insert(int *vmin*, int *vmax*)	Insert **vmin** into **minheap** Insert **vmax** into **maxheap** Link **vmin** with **vmax**
update(Node nmin, Node nmax, int vmin, int vmax, int cmin, int cmax)	Set **nmin.value = vmin** Set **nmax.value = vmax** Set **nmin.linked.value = cmin** Set **nmax.linked.value = cmax**
minmax	Return the pair of **minheap.min()** and **maxheap.max()**. These values are not necessarily linked (and in general are not).

The **insert** and **update** operations complete in $O(\log n)$ worst-case time. **minmax** completes in constant time.

The algorithm is written in pseudocode below. A full implementation can be found on GitHub.

```
int median(RBRTree *trees, int R, int N){
  CorrelatedHeap medians(N);

  Node *medmin, *medmax;

  alive_count = N;

  for t in trees:
    t.medianhigh = t.nth(n/2)
    t.medianlow  = t.nth(n/2 - 1)
    medians.insert(t.medianlow, t.medianhigh)

  while alive_count > 0:
```

[1] The idea of maintaining an upper and a lower median was originally inspired by a StackOverflow post by users Nemo and Himadri Choudhury. It is possible to not maintain the upper and lower medians. However, optimizing this code by removing the upper and lower medians would probably increase efficiency at most by a factor of 2, which is still slower than the naïve Quickselect algorithm. The CorrelatedHeap structure would still be required.

```
  Node medmin, medmax = medians.minmax()
  int cut = min(medmin.size/2, medmax.size/2)

  medmin.size = medmin.size - cut;
  medmax.size = medmax.size - cut;

  medmin.medianlow  = medmin.medianlow.successor(cut/2)
  medmin.medianhigh = medmin.medianlow.successor(1)

  medmax.medianlow  = medmax.medianlow.successor(cut/2)
  medmax.medianhigh = medmax.medianlow.successor(1)

  if medmin.size <= 0:
    medmin.alive = false
    alive_count -= 1
  if medmax.size <= 0:
    medmax.alive = false
    alive_count -= 1

  medians.update(medmin, medmax, medmin.medianlow, medmin.medianhigh)
return the median of the most recently killed array.
```