Fast Median Finding on Digital Images

Kevin Dinkel[1] and Andrew Zizzi[2]
University of Colorado at Boulder, Department of Aerospace Engineering Sciences

In digital image processing, a fundamental first step in feature extraction often involves finding the median pixel value of the entire image. However, on systems where computation time is constrained, finding the median of a large image can be costly. Modern CCD and CMOS chips produce arrays containing several million elements; however, this data is discrete and integer. *A priori* knowledge about the sensor's bit resolution, b, can bound the data values within the range $[0, 2^b - 1]$. These attributes enable the organization of the digitized values into a $\theta(2^b)$ space complexity histogram in only $\theta(N)$ time, where N represents the number of pixels in the image. Once the histogram has been constructed, finding the median takes at worst $O(2^b)$ time, and usually much less. The derived performance of this method outperforms other linear-time, deterministic algorithms, such as Median of Medians, which are too general to take advantage of these unique properties. For time constrained digital imaging systems, the histogram median finding method can be implemented and executed more efficiently.

Nomenclature

b = sensor bit resolution
C = multiplicative coefficient for an asymptotic runtime
k = refers to the index of the kth smallest entry in a list of length N ($k=N/2$ for the median)
N = number of pixels in the image
S = a set of elements
v = pivot value

I. Introduction

Finding the median pixel value of an image is often a useful step in image processing, due to its ability to robustly estimate the background, or surrounding, intensity of an image. The median can be defined as:[3]

1) The middle value in a sorted list of finite length N
2) The value such that there is an equal number of values greater and smaller than it in a list of finite length N

The power of the median is its ability to remain invariant to $N/2 - 1$ outliers in an array of length N. This property does not hold for other statistical measures, including the mean. In applications such as astronomical observation, ignoring outliers such as hot pixels or stars while estimating the background intensity is necessary before further analysis.

While finding the median is useful, it can be time consuming. This poses a problem for time constrained image analysis systems such as startrackers, which often have limited processing resources and a large number of total pixels. Using the first definition, the median can be directly determined by finding the value of the middle pixel in a sorted list. However, even the best sorting algorithms take $O(N \log N)$ time for an image of N pixels. In practice, computing the median of a set is typically handled by a class of algorithms referred to as selection algorithms[2], which utilize the second definition of the median. The Median of Medians[1] is the most frequently implemented algorithm in this class. While Median of Medians boasts an $O(N)$, linear time, solution to finding the median, it is too general to take advantage of the unique aspects of real world digital images. An alternative approach, the

[1] Student, University of Colorado at Boulder.
[2] Student, University of Colorado at Boulder.

histogram median method, uses *a priori* knowledge of these properties to produce medians, easier, quicker, and with less memory.

II. Median of Medians

Median of Medians is a divide-and-conquer, partitioning algorithm that performs in $O(N)$, linear, time.[4] The algorithm operates by dividing a set, S, into groups of 5 elements. Due to their small size, the median of each group can be calculated in constant time, $O(1)$, using any standard sorting algorithm. The median of these medians is then calculated by repeating this process recursively. Once the median of the medians is found, it is used as the pivot, v, to partition the original problem set, S, into two subsets: a list of values less than the median of medians, S_L, and a list of values greater than or equal to the median of medians, S_R. The true median must lie in either S_L or S_R. Since the median is defined as the middle value of a sorted set, its position, $k = N/2$, can be compared to the size of each partition to determine in which subset it must lie. The algorithm is then called recursively on the selected partition, S_L or S_R, until the true median is determined.[2]

The algorithm described above can be implemented using the following pseudocode. The median is calculated by calling method *select()* with inputs: *list* = S, *left* = 0, *right* = $N - 1$, and $k = N/2$. This pseudocode assumes the input set, S, is a one-dimensional list for algorithmic clarity. It is possible to alter these functions for use on two-dimensional arrays (ie. images) using index wrapping, but that exercise is left to the implementer. The arguments *left* and *right* refer to the bookend indexes of the active list, which change in value dynamically during recursive calls on S_L or S_R. Input k corresponds to the k^{th} smallest entry in an list, as specified in generic selection algorithms. Again, to find the median, k must point to the middle index of a sorted list, $N/2$.

```
// Swaps the values found in list[index1] and list[index2]
swap(list, index1, index2)
    temp := list[index1]
    list[index1] := list[index2]
    list[index2] := temp

// Divides the list into a left partition (values in list < pivotIndex)
// and a right partition (values in list ≥ pivotIndex) by swapping values.
// Returns the index at the beginning of the right partition
partition(list, left, right, pivotIndex)

    // Get the value of the pivot
    pivotValue := list[pivotIndex]

    // Move the pivot to the end for later use
    swap(list, pivotIndex, right)

    // Swap any values less than the pivotValue to the left side of the list
    // Any remaining values, those greater or equal to pivotValue, will stay on the right side of the list
    storeIndex := left
    for i from left to (right – 1)
        if list[i] < pivotValue
            swap(list, storeIndex, i)
            storeIndex := storeIndex + 1  // Increment the location of the partition

    // Move the pivot to its final place, at the beginning of the right partition
    swap(list, right, storeIndex)

    // Return index at the beginning of the right partition
    return storeIndex
```

```
// Sorts list using the basic bubble sort algorithm, which is sufficiently fast for a list of length 5
sort(list, left, right)

    // Calculate number of elements in the  list
    N := right – left + 1

    // Sort by swapping
    do
        swapped := false
        for i from 0 to (N – 2)
            if list[left + i] > list [left + i + 1]
                swap(list, left + i, left + i + 1)
                swapped := true
    while swapped = true

// Median of medians selection algorithm. This must be called initially with left set to the 0th entry of
// the list and right set to the length of the list minus 1. k must be set to the length of the list divided by 2.
// Returns the index in list that corresponds to the median value of the list.
select(list, left, right, k)

    // Calculate number of elements in list
    N := right – left + 1

    // If list is small, calculate the median in constant time
    if N ≤ 5
        sort(list, left, right)  // Sorting is usually Ω(N log N), but for small N time is constant, O(1)
        return left + ⌊N/2⌋  // Return middle index, the index of the median

    // Select to pivot based on the median of medians
    // Partition values into subsets of 5 elements each
    storeIndex := left
    for i from 0 to N/5  // Be careful to handle any elements in remainder (not shown here)!!
        // Find the median of each group of 5 elements
        medianIndex := select(list, left + i*5, left + i*5 + 4, 3)
        // Store the median's index at the beginning of the list
        swap(list, medianIndex, storeIndex)
        storeIndex := storeIndex + 1

    // Calculate the median of medians and use this as the pivot
    MoMIndex := select(list, left, storeIndex – 1, ⌊N/10⌋) // k is (N/5)/2, the middle index

    // Partition the list using median of medians as pivot value
    pivotIndex := partition(list, left, right, MoMIndex)

    // Calculate pivot index relative to beginning of active list
    pivotDistance := pivotIndex – left + 1

    // If k lies in the lower half, recurse on the lower half
    if k < pivotDistance
        return select(list, left, pivotIndex – 1, k)
    // If k lies in the upper half, recurse on the upper half
    else if k > pivotDistance
        return select(list, pivotIndex + 1, right, k – pivotDistance)
    // If k = pivotDistance, return the median index
    else
        return pivotIndex
```

Analysis of Median of Medians stems from the generalized selection problem. Selection algorithms can produce linear best-case run times, but exhibit quadratic run times with a "poor" choice of a randomized pivot, v. When v is consistently chosen as the minimum or maximum value of the set, the worst case scenario produces the recurrence relation[2]

$$\left.\begin{array}{c} T(n) \le T(n-1) + O(n) \\ T(n-1) \le T(n-2) + O(n) \\ T(n-2) \le T(n-3) + O(n) \\ \vdots \\ T(1) = 1 \end{array}\right\} T(n) \le n\theta(n) = \theta(n^2)$$

However, if the pivot is consistently chosen to be in the middle 50% of values in the set, then the subsets S_L and S_R have a maximum size $3/4$ that of S, which simplifies the recurrence relation to[2]

$$T(n) \le T(3n/4) + O(n) = O(n)$$

The Median of Medians algorithm is guaranteed to run in $O(N)$ since the median of medians, chosen as v, ensures that the pivot's location is always in the middle 50% of values in the set.[2]

The space complexity for Median of Medians is $\theta(N)$, since there is no reduction of the original set size. The included pseudocode is implemented "in place", eliminating the need to allocate memory for any extra data structures.[2] While, this aids in the memory performance of the algorithm, it should be noted that the input list order is altered during runtime by the internal *swap()* mechanism. If an unaltered list is desired along with the median value, the entire image must be copied prior to running *select()* – a time and memory intensive process.

III. Histogram Median Method

While selection algorithms are good candidates for finding the median in a general case, they fail to take advantage of the unique properties provided by digital sensors. CMOS and CCD detectors produce values that are discrete, integer, and within the range $[0, 2^b - 1]$, where b is the bit resolution of the sensor. This means there are a finite number of values, 2^b, that each pixel can assume. The array size of the image, N, is also known. Modern, multi-megapixel sensors usually have the property that $N \gg 2^b$, meaning that there are many more pixels in the image than possible pixel values. Taking advantage of this *a priori* knowledge, the histogram median finding method improves efficiency over Median of Medians.

Constructing the histogram of an image can be achieved in linear time with respect to the number of pixels in the sensor, N, over the known range $[0,2^b - 1]$ provided by the sensor. With the histogram generated, the median relates to the bin that splits the area under the curve into two equal sections, Figure 1. A simple method for finding this value is to add the counts from each bin, starting on one end and progressing towards the other. The median is found once the total count is greater than $N/2$. Pseudocode for this algorithm is provided below.

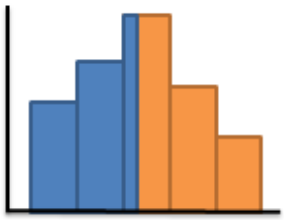

Figure 1: The bin that splits the histogram into equal areas contains the median value

```
// Returns the median value of list
histogram_median(list, b, N)

    // Histogram has 2^b number of bins
    histogram[2^b]

    // Initialize all bins to zero
    for i from 0 to (2^b-1)
        histogram[i] := 0

    // Populate the histogram
    for i from 0 to (N – 1)
        histogram[list[i]] := histogram[list[i]] + 1

    // Median is defined as the middle pixel value
    count := N / 2

    // Find the median
    for i from 0 to (2^b-1)
        count := count – histogram[i]
        if count < 0
            return i  // The bin that contains the median pixel value
```

The running time of the histogram median method can be broken down into the three fundamental steps: Initializing the bins of the histogram takes $\theta(2^b)$ time, populating the histogram takes $\theta(N)$ time, and finding the median is, at worst, $O(2^b)$ time. This yields an overall running time of $O(N + 2^b)$. Because modern digital sensors have the property $2^b \ll N$, the running time can be simplified to $\theta(N)$.

This linear runtime matches the best case performance of the fastest selection algorithms, including Median of Medians. However, the omitted multiplicative coefficients in front of these run times (ie. $C * \theta(N)$) vary greatly. For example, the histogram median method's leading constant is defined as:

$$C_{Hist} = 1 + \frac{2^b}{N}$$

A sensor with 16 bit depth resolution (b = 16) and a 3 megapixel spatial resolution (N = *3E6)* has a C_{Hist} which equates to ~ 1.02. This means that for most standard cases only $\sim N$ operations need to be performed on the image. It can be shown that the coefficient for the asymptotic runtime for Median of Medians, C_{MoM}, is far greater than that of the histogram median method. The derivation of this claim is not included, but the magnitude of C_{MoM} is apparent in the extra overhead associated with the *swap()*, *sort()*, and *partition()* operations found in the Median of Medians' pseudocode. Even though both runtimes vary linearly with the size of the input image, N, the histogram median method will outperform Median of Medians by a factor of C_{MoM}/C_{Hist}.

The space complexity of the histogram median method is $\theta(2^b)$ since calculating the median uses only the histogram data structure of length 2^b. If $2^b < N$, the histogram median method will outperform Median of Medians by a factor of $N/2^b$ (~45.8 times better for a sensor with b = 16 and N = *3E6*). Also, unlike Median of Medians, the histogram median method does not alter the order of the values found in the input list during runtime. This means that a duplicate copy of the list need not be made in order to retain its original integrity.

It should be noted that the memory footprint of the histogram median method is particularly low when finding medians in entire images produced by a modern CCD or CMOS sensor because $2^b \ll N$. For applications where $2^b \ll N$ no longer holds, such as implementing a median filter over a small subset of an image, the space and time complexity of the histogram median method balloons. Both metrics becomes more dependent on the depth of the bit resolution, b, rather than the number of pixels, N. However, in the case of a multi-megapixel image, even the range of values from a 16-bit sensor is on the order of a few percent of the total number of pixels. Thus, the histogram median method will perform in the desired $\theta(N)$ time with a leading coefficient $C_{Hist} \approx 1$ and $\theta(2^b)$ space complexity.

IV. Conclusion

Finding the median of an entire image quickly is vital to many real-time image processing systems. The histogram median method provides a guaranteed linear running time, $\theta(N)$, which matches the best case of the fastest selection algorithms, including Median of Medians. However, through examination it can be seen that the multiplicative coefficient in front of the histogram median method running time, C_{Hist}, is much less than that of Medians of Medians, C_{MoM}.

The histogram median method also improves on the space complexity of selection algorithms, decreasing the size of the data structure from N to 2^b, while preserving the integrity of the original image. In contrast, the Median of Medians algorithm presents a space complexity of $\theta(N)$, with no reduction in the size of the original data structure. Since Median of Medians is run "in place", the image is altered during runtime, requiring a duplicate copy to be made if the user wants both the median and the original image.

The true advantage in the histogram median method may lie in its simplicity. This algorithm has been formulated and utilized on a student designed, balloon-stationed startracker named DayStar from the University of Colorado at Boulder. As a vital step in calculating a robust estimation of the sky background brightness, this method has proven both effortless in implementation and efficient during runtime. The DayStar project does not claim rights to the invention of this method, just its effective use in a unique application.

Acknowledgments

The authors would like to thank Dr. Eliot Young of the Southwest Research Institute for his involvement and encouragement in the publication of this work.

References

[1]Blum, M., Floyd, V., Pratt, V., Rivest, R., and Tarjan, R., "Time bounds for selection," *J. Comput. System Sci. 7*, 1973.

[2]Dasgupta, S., Papadimitriou, C., and Vazirani, U., *Algorithms*, 2008, pp. 54, 74.

[3]Devillard, N., "Fast median search: an ANSI C implementation," 1998, pp. 1-14.

[4]"ICS 161: Design and Analysis of Algorithms; Lecture nodes for January 30, 1996," Dept. Information & Computer Science – UC Irvine, 2000. <http://www.ics.uci.edu/~eppstein/161/960130.html>