VERTICAL DATA STRUCTURES AND COMPUTATION OF SLIDING WINDOW AVERAGES IN TWO-DIMENSIONAL DATA

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Adam Paul Helsene

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

May 2020

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Vertical Data Structures and
Computation of Sliding Window Averages in Two-Dimensional Data

By

Adam Paul Helsene

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Anne Denton
Chair

Dr. Gursimran Walia

Dr. David Franzen

Approved:

5/15/2020	Dr. Kendall Nygard
Date	Department Chair

ABSTRACT

A vertical-style data structure and operations on data in that structure are explored and tested in the domain of sliding window average algorithms for geographical information systems (GIS) data. The approach allows working with data of arbitrary precision, which is centrally important for very large GIS data sets.

The novel data structure can be constructed from existing multi-channel image data, and data in the structure can be converted back to image data. While in the new structure, operations such as addition, division, and bit-level shifting can be performed in a parallelized manner. It is shown that the computation of averages for sliding windows on this data structure can be performed faster than using traditional computation techniques, and the approach scales to larger sliding window sizes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Improvements in the capture of geospatial image data has driven an explosion of raw data that is to be processed in geographic information systems. Processing this data in its raw format can be time intensive due to its volume. Often, data is preprocessed in some manner before ingestion into the algorithm of interest in order to smooth out the data or to reduce the data set size in a way that has minimal impact on the accuracy of the end result. Given the size of typical data sets, even relatively simple processing steps can require significant processing time and have extensive storage requirements.

This paper explores bitwise-vertical data structures, and processing using that structure, with the goal of improving computation efficiency. It introduces a new technique for processing bitwise-vertical data that reduces the overall computation time required for sliding window calculations. A major benefit of these data structures and techniques is that they can handle arbitrary precision integer data.

By leveraging and combing techniques and enhancements from different areas of study, a novel framework for applying operations and algorithms simultaneously across multiple data points emerges and is further developed and explored.

CONCEPTS

Geographic Information Systems

A geographic information system (GIS) is a computerized system for importing, exporting, managing, and analyzing spatial data as it relates to geographic regions or maps (Longley, Goodchild, Maguire, & Rhind, 2011, pp. 11-13). These systems, and their data, are commonly used to answer questions in various fields of study, as they relate to spatial data, specifically those of a geographic nature. Civil infrastructure and planning, transportation, land-use planning, logistics, and farming are all examples of fields where GIS plays an important role. As more specific examples, terrain slope data can be used to help predict dangerous areas for landslides (Longley, Goodchild, Maguire, & Rhind, 2011, pp. 14-17), and images of fields can be used to help predict harvest yields (Pradhan, 2001).

Remote sensing

GIS data include many types and formats. One common source of data for GIS systems is remotely sensed image data, often captured by satellites, airplanes, or unmanned air vehicles (UAVs). Satellites have evolved significantly over time, with early Landsat satellites capturing images with a nominal resolution of 79 meters per pixel. Newer satellites capture at 30, 15, or even 5 meters per pixel (Jensen, 2005, p. 13), and some UAV-based methods can capture at resolutions as high as centimeters per pixel via digital photography.

Independent of resolution, different remote sensing methods and imaging platforms can capture different sets of wavelengths in image data. Early Landsat models, which used the Landsat Multispectral Scanner (MSS) were capable of sensing and recording four different bandwidths of image data: red, green, and two near-infrared bands. Newer iterations used new sensing equipment which enabled seven or eight different channels of data, spanning from lower-wavelength blue through higher-wavelength thermal infrared data (Jensen, 2005, pp. 51-57). Other systems such as Airborne Visible Infrared Imaging Spectrometer (AVIRIS), which collects from an air-based vehicle, can collect up to 224 bands of data (Jensen, 2005, pp. 13, 435, 443).

Remote sensing of data has been a driver of the creation and advancement of algorithms to analyze or process that data. Improvements to remote sensing equipment have increased the need to summarize and process data of increasingly high resolution, to better analyze this data, and make decisions at the desired scope of reference. Moving from 30-meter resolution in a two-dimensional image to a 5-meter resolution also increased the amount of data to be stored by a factor of 36. Moving to 1-meter resolution would increase storage by a factor of 900. The process of averaging over a window also serves as a proof-of-concept for a large number of other window-based computations.

Image formats

Image data takes either of two forms, raster or vector. Raster images represent a field of data as individual cells, or pixels. Each individual pixel's data describes the intensity of a measurement of that cell. For common visual image data, each pixel is described by three or four values, with each value corresponding to an intensity of an individual band, or channel, of color. This is called "band interleaved by pixel" format, and is the format for traditional bitmap files, among other popular image files (Jensen, 2005, pp. 101-103, 119). Satellite image data are also stored in a raster format, but the bands are normally stored separately.

Vector images describe an image as a set of vertices and lines, curves, or polygons. This format can enable calculations and processes that are not practical for raster data, due to its precise representation, storage attributes, and availability of general algorithms that can be applied to vertex, line, curve, and polygon objects (Longley, Goodchild, Maguire, & Rhind, 2011, p. 214). Vector data can very accurately and clearly define coordinates and shapes of objects. It is also useful for certain classes of calculations and algorithms that apply to line-based data and curves.

In memory, spatial raster image data in computer systems is generally stored in an array format. A one-dimensional array is indexed such that the *n*-th item of the array is addressed as array[n]. For spatial data, such as geographic images, the data takes the form of a two-dimensional array of pixels. For these arrays, there are two indexing values used to address a specific pixel, and an array is typically addressed as $array[m, n]$ where m is the row of the intended value, and n is the column index. Even though there are two index values, the data is still stored in memory in the same manner as a single-dimension array of size $m * n$ (Null & Lobur, 2010, p. 740). This allows for a two-dimensional array to be addressed in the same manner as a single dimension array, using $array[m * w + n]$, where *w* is the width of a single row of the data.

There are many ways to describe and store bitmapped raster data, such as bitmap (BMP), graphics interchange format (GIF), and tagged image file format (TIFF). These formats store data in the form of a file header which describes the image, and the image data array. A common format for an image data array itself is RGB format, where red, green, and blue color information is stored for each pixel in pixel-primary order. For geospatial data, these different bands can be stored separately. Pixels in image data are ordered in row-first order, from left to right. The values of each pixel of the band takes the form of positive integer data, stored as binary. Band information can have varying levels of detail for different images, often dictated by limits of the imaging hardware that is capturing the image data. A pixel format of 8 bits per band per pixel is common, but images and processing systems can be of 4 bits, 5 bits, 12 bits or other sizes depending on usage. (Jensen, 2005, pp. 154-157).

Raster data can consume a large amount of storage space, and is often stored in a compressed format, which may be compressed via many different methods. Examples of techniques that are employed for this compression include run-length encoding, block encoding, and wavelet transforms. Some of these techniques, such as wavelet transforms, involve an asymmetric function being applied to the image. Due to the asymmetric nature, the raw data cannot be recovered, and the compression is described as *lossy compression*. Other techniques, such as run-length encoding, can be reversed to result in the original data set and are described as *lossless compression* (Longley, Goodchild, Maguire, & Rhind, 2011, pp. 211-214).

Compressed data can create challenges or barriers in the ability to process the raw data, however, and often these compressions are not used during processing, but instead only for persistent storage of the data. In this paper, uncompressed bitmap data is used.

Sliding window techniques

These are a class of algorithms and techniques that describe processes by which an aggregating calculation is applied to data of an image using overlapping segments, or windows. These windows are processed using the chosen aggregation method, such as averaging, and one value is stored for each window. All subwindows of a given size are considered, and the process of applying the windows is said to be sliding, describing the apparent motion of the window as each next data point is calculated. Certain kinds of aggregations are common in GIS for many reasons, such as smoothing data, enhancing contrast, and preprocessing high resolution data for later work at different scales (Denton, Ahsan, Franzen, & Nowatzki, 2016, p. 1).

One basic aggregation applied to a sliding window is an average of the points of the window. For example, for two-dimensional data a window with size of height and width 2 may be used to calculate averages, to smooth data. For each cell of the data set, the resulting value of the output at that cell at location (m, n) is the the sum of the values at (m, n), $(m + 1, n)$, $(m, n + 1)$, and $(m + 1, n + 1)$ divided by 4, the number of cells. This generates an average at each point, comprised of data of nearby points. It performs preprocessing of the data and reduces the impact of outliers and variations at the levels of highest resolution.

Vertical data structures

In typical data access and analysis, a data set is comprised of multiple data records, and each record can store data representing multiple attributes of the object the record represents. When only specific attributes are desired, or data is scanned to aggregate values of a single attribute of each record of the data set, this can lead to inefficiencies. Because of this, techniques have been developed to create and handle vertical data structures for purposes of data mining and aggregation. In standard data sets, each record represents a single object, which then has multiple attributes. In vertical structures, data is pivoted such that each attribute becomes a record, and the values of each record's attributes is collected as the new record's data. This data is stored such that a vertically-oriented record represents the values of a specific attribute of all horizontal records (Han & Kamber, 2006, pp. 245-248).

These techniques need not only apply at a record and attribute level as considered by a database. The above vertical data structures consider data at the granularity level of a full attribute that is part of a data record. However, the attribute level is not the only granularity of data that can be made vertical. Even data of an attribute, such as an integer, can itself be considered as a record whose attributes are the bits that comprise the binary representation of that data.

Some programming languages have built-in constructs to create vertical versions of standard arrays, however, these are still concerned with data at the scope of individual records. For sliding window averages, the calculation process still requires the same kind of random access of items in the array, and bit depth that doesn't align with the processor's languages and preferences may still have inefficiencies with data storage and processing.

PROPOSED APPROACH

Data structure overview

In data mining, vertical data structuring is a technique used to transform data into a structure that's beneficial to various mining algorithms. This is typically done at the level of a data record, such as an integer (Han & Kamber, 2006). In a manner reminiscent of this, arrays of integer data can also be structured vertically at the binary level by performing a slice operation on the bits of the data set. In this manner, each bit of the data is sliced and converted to a bit vector. This is done such that the resulting data set contains *n* bit vectors, where *n* is the bit depth of the data being stored. Each bit vector is referenced by the index of the bit as it appeared in the original data record.

It is possible to restructure a dataset such that the new data structure provides certain desirable computational properties for certain classes of computations. A dataset can be transformed into a vertical-style data structure, which can have the necessary arithmetic operations applied to compute a sliding window average for each point in the data set.

Some benefits of a vertical-based approach include an increased ability to parallelize mass computation, flexibility with data precision, and increased speed of certain calculations compared to naïve implementations across the same data set.

In order to perform the aggregation function calculation, however, operations such as addition, multiplication, and division need to be implemented in a way to leverage the novel data structure in order to apply to the vertically structured data en masse. For this, we draw insight from hardware-based implementations and find that implementations of these operations at a low level, such as that done inside a general-purpose CPU, also can apply to the vertical data. The bit-vector based nature of the structure allows us to perform these hardware-style calculations in parallel, often with 32 or 64 values at a time.

Algorithm overview

Sliding window calculations, such as calculating an average, involve generating a sum of *n* or more numbers that are at different, but predictable positions in the data set. We use properties of vertical data structures to align these different numbers and then process the calculations in parallel.

The overall flow of the algorithm to compute a sliding window average using vertical data processing techniques involves pivoting the data to a vertical form, aligning the operands of the arithmetic operations, and performing the addition and division required to calculate an average, and finally reversing the pivot operation.

The initial pivot provides the structure for parallelization of the algorithm. It also gives the data and algorithm other useful characteristics, one example being that the operations performed can be against data of arbitrary bit depth.

The alignment step corresponds to choosing different positions in the sliding window to perform the calculations on. There are many ways to choose which positions in a window to operate on, the simplest being choosing the four corners of the window.

Pivoting the data

For computing systems, an array of integers is ultimately stored in binary, with items stored in sequence such that each number requires storage between 8 and 64 bits. These sizes and boundaries are convenient to match internal hardware of common processors. As an example, 4-bit integers stored in an array representing the set $\{1, 4, 6\}$ would be stored as 0001 0100 0110 in standard 4-bit horizontal format.

For a data set of *n* unsigned integers, we introduce a vertical data structuring of the numbers that allows us to transform a set of *n* integers of bit-depth *d* into a set of *d* bit vectors of cardinality *n*.

A bit plane *B*B_i is defined as a data structure that contains an index value *i* and a bit vector. The bit vector component contains the data of one bit of each integer of the original data set, while the index value represents the index of the bit from which the vector is originally derived.

We can represent this as $B_i = [i, \{b_{i0}, b_{i1}, b_{i2} \ldots b_{in}\}]$ where n is the number of integers in the original data set and $B_{in}$is the *i*-th bit of the *n*-th integer of the original data set. The full vertical data structure for all bits is represented as $B = \{B_0, B_1, B_2, \ldots, B_m\}$ where *m* is the bit depth of the integer data. In common usage, *m* is likely to be a power of 2.

```
function PivotData(originalData, colorBitDepth)
   let planes = new BitPlane[];
   let dataItemCount = originalData.Size;

   for(int i = 0; i < colorBitDepth; i++)

       let plane = new BitPlane(index: i);
       let planeData = new BitArray(length: dataItemCount);
       for(int j = 0; j < dataItemCount; j++)
           // Extract only the bit for this specific position
           let mask = 1 << i;
           planeData[j] = (originalData[j] AND mask)
       end for
       plane.Data = platData;
       colorPlanes.Add(plane);

   end for

   return planes;

end function
```

Figure 1: Pivoting an array of integers to the vertical bit plane structure.

As an example of the pivot operation, the integer set $\{4, 5, 6, 7\}$ of bit-depth 4 is used. These four numbers are normally represented in binary as 0100, 0101, 0110, and 0111, respectively.

Table 1: Integers as horizontal data shown as typically represented in computer memory.

	B_3	**B_2**	**B_1**	**B_0**
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1

To create the vertical bit vectors, we take the least significant bit from each row and construct a vector with $i = 0$. The next, more significant, bit is used for $i = 1$, and so on. The values of the bit vector are derived from the integer data This creates a set of bit planes that represent a bit index and the vertically structured data for that index:

$B = [0, \{0,1,0,1\}], [1, \{0,0,1,1\}], [2, \{1,1,1,1\}], [3, \{0,0,0,0\}]$.

Table 2: Pivoted data in bit vector format.

	4	**5**	**6**	**7**
B_0	0	1	0	1
B_1	0	0	1	1
B_2	1	1	1	1
B_3	0	0	0	0

The result of this operation is a set of vertically oriented data, with a data item previously at index *i* now composed of one bit from each vector at index *i*, for each vector. While this may not be necessarily an intuitive representation of the components of each integer, it enables applications of arithmetic operations to be applied in a SIMD (single instruction, multiple data) manner on a single processor.

Note that this structure is also not limited or constrained to have a particular number of bit vectors. In this example, four-bit vectors are created as the data is 4-bit integer data. There is no limit to the number of bits that comprise the integer data, other than practical limits of memory and storage. In this way, the data structure can represent integer data of arbitrary precision.

<u>Arithmetic operations on vertical data</u>

With this vertical data structure, it is possible to implement the different operations that comprise the calculation of sliding window averages. Sliding window averages use the common averaging formula over a set of numbers *S*:

$$\mathrm{Avg}_S = \frac{\sum_{i=0}^{n} s_i}{|s|}$$

Calculating these averages requires addition and division operations, and to aid in the calculation, a left shift operation is used to align operand vectors in the addition operation for computational convenience.

<u>Addition operation</u>

Addition on vertical data that is structured in the bit plane format closely resembles addition performed at the hardware and logic layer of a processor. Because we consider individual bits of data, much like a hardware logic circuit, we can draw inspiration for implementation of operations from the corresponding hardware logic implementations. As an example, a ripple-carry adder implementation can be applied to the bit vectors (Null & Lobur, pp. 139-140).

```
Function AddVerticalVectors(B, Op)
   let dataVectors = B // Vertically structured data
   let operandVectors = Op // Vertically structured operand data

   foreach (integer i from 0 to dataVectors.Length - 1)
      let dataBitVector = d[i]
      let operandBitVector = operandVector[i]

      // A temporary array used to carry overflow from bit i to i+1
      let carryVector = new BitVector(length: dataBitVector.length)
      d[i] = AddOperandVectorToDataVector(dataVector, operandVector, carryVector)
   end foreach
```

Figure 2: Algorithm for applying addition operation to each data vector.

```
Function AddOperandVectorToDataVector (dataVector, operandVector, carryVector)
   let initialVector = dataVector.copy()
   let intermediateVector = dataVector = dataVector XOR operandVector
   dataVector = dataVector XOR carryVector
   carryVector = (carryVector AND intermediateVector) AND operandVector
```

Figure 3: Ripple-carry adder implementation for vectors.

To add a number to each record of the entire data set while it's in bit plane format, the operand of the addition can be transformed into a set of vertical bit planes, with each bit of the constant operand transformed into constant vectors and applied to each vertical data bit vector.

For example, we can add the value 3 to every item in a set of 4-bit numbers in the following manner. The number 3, represented as 0101 in binary as a 4-bit integer, can be added to every element of a set of *n* numbers by constructing the vertical constant-value bit vectors corresponding to each bit of the 4-bit representation of the number,

$Op = \{[0, C(1,n)\}], [1, C(1,n)], [2, C(0,n)], [3, C(0,n)]\}$ where $C(m,n)$ represents a constant bit vector with value $m \in (0,1)$ and cardinality *n*.

Each of these operand bit vectors can then be applied to the data bit vectors via several *XOR* operations across vectors starting with the last significant bit. A carry vector is used to help us keep track of when we need to carry values of *1* at various indexes to the next operation.

To perform addition, we also need to detect and handle overflow from the vector with index n as a part of the calculation of the vector with index $n + 1$ to implement addition. The algorithm to perform overflow and carry operations across the digits of the vectors are analogous to the implementation of a standard hardware adder. Where a hardware adder may operate on a single record at a time, the vertical implementation can operate easily across 32 or 64 records as a group, one bit at a time, depending on the optimal architecture of the hardware.

In this way, hardware algorithms that typically operate directly against data in a binary format, such as a ripple-carry adder, can now be applied to the vectors as a parallel operation. The previous example used a constant set of vectors for the integer 3 to add that value to each value in the data vector. Non-constant vectors can be used in the same manner, to add different values to different items at different indexes in the data set.

<u>Multiplication operation</u>

We can extend hardware-style implementation algorithms to operations such as multiplication and division. In processor hardware, integer multiplication and division by powers of 2 can be implemented as a bit shift operation on a single value at a time. Each bit shift corresponds to a single multiplication or division by 2 (Null & Lobur, 2010, p. 71).

The data, after being structured vertically, is now in a format where there are *n* vectors with each vector corresponding to a bit. This multiplication bit shifting operation is now as simple as re-assigning index values to the different bit planes that contain the data vectors.

As an example, to multiply by eight, we can multiply by 2 three times. A set of numbers with bit depth = 16, represented as 16 separate vertical bit planes, can all be multiplied by 8 by adding 3 to each index value and constructing new zero-value constant bit vectors with plane indexes of 0, 1, and 2.

The hardware analog for this operation is a left shift without carry. A right shift without carry can perform division with rounding as a floor function.

```
Function Multiply (dataBitPlanes, operand)
   // First determine how many shifts are required
   let numberOfShifts = Math.LogBase2(operand)
   let planeSize = dataBitPlanes[0].Length

   // Perform the shifts
   foreach (dataBitPlane in dataBitPlanes)
      dataBitPlane.Index += numberOfShifts
   end foreach

   // Add new zero-value bit planes to fill in the lower-ordered bits that were shifted
   for(integer i = 0; i < numberOfShifts; i++)
      dataBitPlane.Add(new BitPlane(length: planeSize, index:i, value:0)
   end for
```

Figure 4: Multiplication via bit shifting.

After any of these shifting operations, there are three options for handling overflow in this kind of shift operation:

1. Remove the three highest-order bit planes. This is analogous to a shift without carry. This retains bit depth of the data throughout the operation.
2. In the left shift scenario, we can extend precision by three bits. This requires no extra action outside of the initial adjustment to the bit plane index values.

3. Reassign the three highest-order bit planes as the lowest-order bit planes (or the lowest to highest for right shift). This approach is not generally valid for multiplication, but can be used for other bit shift scenarios.

Multiplication by non-powers of two can be performed as a multiplication by the nearest power of two and repeated addition operations as previously defined.

Division operation

```
Function Division(dataBitPlanes, operand)
   // First determine how many shifts are required
   let numberOfShifts = Math.LogBase2(operand)

   // Perform the shifts
   foreach (dataBitPlane in dataBitPlanes)
      dataBitPlane.Index -= numberOfShifts
   end foreach

   // Remove the planes whose data is lost by division
   dataBitPlanes = dataPlanes.Where(Index >= 0)
```

Figure 5: Division by bit shifting.

Division by an operand whose value is of the form 2^n can be performed by a *right shift without carry* operation against the bit planes. This is accomplished by adjusting indexes of the planes, like multiplication. This becomes integer division with floor function rounding, though other rounding algorithms are possible. A common rounding algorithm can be implemented by referencing the largest bit plane index with index > 0 after the reindex operation to decide if the value should round up or down to the nearest whole number. This bit plane can be added to the new plane at index 0.

<u>Aligning the data vertically for average operation</u>

As part of calculating the sliding window averages using a common 'corner-based' sampling method, data from four different positions of the overall data structure participate in each window calculation.

Table 3: An example of two-dimensional data, with four example 2x2 windows.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

This sample data's first window consists of the values at indexes 0, 1, 4, 5. The second consists of the average of values at indexes 1, 2, 5, 6. The third window is 2, 3, 6, 7 and so on.

For extremely large data sets, random accesses to data at various index values can cause challenges to computation, especially if the data set cannot fully exist in memory or the computation needs to be parallelized.

To simplify the indexing while calculating averages of values in our data planes, we can align all values that contribute to an output value such that all contributing values reside at the same index. This simplifies the implementation of the adder, such that it needs to not be interested in randomly accessing data in the operand bit planes. In the example above, we aim to align, in the bit planes that represent the data, the values that contribute to the output at index n. In this case, aligning the values at indexes 0, 1, 4, and 5 is ideal. For each bit-indexed bit vector in our bit plane set, we can perform shifts of the vectors to accomplish this alignment.

```
Function AlignPlane(dataBitPlanes, indexOffset)
   let shiftedPlanes = new BitPlane[];

   // Copy from the offset to the end of each plane into the new one
   foreach (bitPlane in dataBitPlanes)
       shiftedPlanes[bitPlane.index] =
           bitPlane.copy(indexOffset, bitPlane.vector.length – indexOffset;

   return shiftedPlanes;
```

Figure 6: Creating a copy of a bit plane, with data records offset.

The bit planes can be cloned in memory or storage, and then each bit vector shifted as desired to gain the desired alignment. This aligns all values at index *n* such that the resulting value at *n*, after calculating the average, is the average of the values at *n* in each copy of the data. This alignment takes place for each bit plane.

Table 4: Sample data for a sliding window calculation.

	data[0]	data[1]	data[2]	data[3]	data[4]	data[5]	data[6]	data[7]	data[8]
Data copy 1	0	1	2	3	4	5	6	7	8
Data copy 2	1	2	3	4	5	6	7	8	9
Data copy 3	4	5	6	7	8	9	10	11	12
Data copy 4	5	6	7	8	9	10	11	12	13

One of the major advantages of this alignment, performed at the bit vector level, is to help reduce random accesses into the data, when adding the initial vector to the first copy such that the operands of a single operation align, The underlying processor can simply load data at the same index from each array as operands.

Calculating the average of a resulting data point

Given the previously defined operations and implementations of addition and division operations, these two operations can be used to calculate averages of sets of numbers with cardinality 2^n for positive integer values of *n*. Data comprised of values of other quantities can be considered in the average should the division algorithm be adjusted to accommodate for them.

Reverse pivot

The calculation of the averages results in bit planes that represent the output values. After all calculations are completed, to get these back to the common non-vertical format, the data must be pivoted once more. We can now perform the pivot in reverse, deconstructing the individual bit planes back into common integer data structures.

```
// For a set of bit planes, reconstructs the integer array that it represents
function CreateColorDataPlanes(BitPlane[] planes)
    let planeSize = planes[0].Vector.Length;
    var data = new int[planeSize];

    // For each bit plane
    for(int i = 0; i < planes.Count; i++)
        // For each item in the plane
        for(int j = 0; j < planeSize; j++)
            let bit = planes[i].Vector[j];
            if (bit = 1)
                let mask = 1 << i;
                data[j] = data[j] OR mask;
        end for
    end for

    return data;

end function
```

Figure 7: Reverse pivot of a bit plane structure back to an array of integers.

EXPERIMENTS

To verify the viability of the vertical approach, the pivot, reverse pivot, addition, division, and shift operations were implemented in C# alongside a more traditional implementation of a sliding window calculation using a traditional two-dimensional array data structure. For various scenarios, the implementations were both run and results gathered and analyzed. In these experiments, measured times are for the core algorithm to operate. The loading and saving times of the images were not considered in the results.

The experiments were performed on a Windows-based PC, using a standard x64 processor supporting Advanced Vector Extensions (AVX) instructions. Pivot, unpivot, addition, division, and shift operations were implemented in C# as described in earlier sections. The C# BitArray class was used as the underlying implementation of binary vectors as it provides software-level support for AVX and related processor extensions to assist with optimizing common XOR, AND, OR, and NOT operations on large bit vectors.

Regarding pivot and reverse pivot operations

The pivot and reverse pivot operations are preprocessing and postprocessing steps that involve loading the image and converting it to the vertical format. This operation is necessary to convert the data but is not considered as part of the core algorithm under analysis.

Over several experiments, the load and pivot operations were found to take, on average, 770 milliseconds. The reverse pivot and save operation were found to take approximately 1026 milliseconds on average. These operations are independent of the algorithms under test and were not necessarily optimized.

Experiment: Adding a value to each pixel

This initial experiment was designed to ascertain the magnitude of performance change when using the vertical-based data processing methods versus using simple array addressing and repeated addition. This also acted as a test to help validate the accuracy of the results.

An image with multiple visible bands was chosen for this experiment, to provide a baseline that would contain image data that may reasonably be typical of GIS images for which a sliding window algorithm is likely to be applied. The image was of a PNG file format, and contained four color channels: red, green, blue, and alpha. The sample image is square and is 1,200 pixels wide and tall.

Figure 8: Sample image used as input for experiments.

Each color channel was extracted from the image, and a constant value was added to each pixel of each color. Each resulting color channel component was recombined back into an RGBA image. This allowed for quick visual examination to determine the accuracy of the vertical method.

The measurement of this experiment was the runtime of the operations of addition and saving the resulting image to a file format similar to that from which it was originally stored. The initial loading of the data into either the data or vertical structure was not considered.

In the traditional approach, each color channel from the bitmap image data was loaded into a standard array of integers. Subsequently, each data item had a constant value added to it and the result was stored back into the array. This required random accesses into the array of numbers with one array access to read the data, and another to save it, for each item.

In this vertical structure and approach, the image had the same constant value added to each color channel, using the vertical data operations. These helped to parallelize additions and reduce random data accesses.

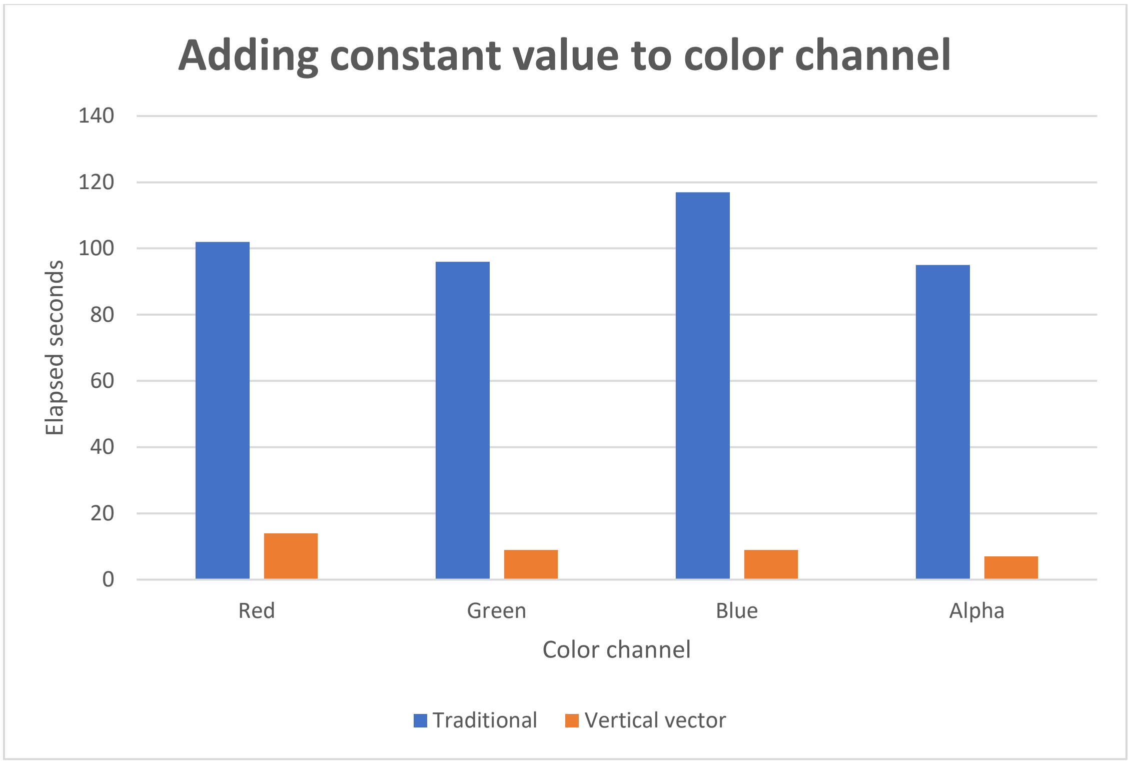

Figure 9: Elapsed time to add constant value to each pixel - by color channel.
A total of 1.44 million addition operations were executed during the test.

The experiment shows a reduction in elapsed time for the vertical method by factors ranging from 7 to 13. This time reduction is attributed to a massive reduction in data object and memory accesses, as well as a reduction in the number of operations overall. This reduction in operation count is attributed to a reduction in random memory access, as well as a reduction in the number of distinct and separate non-parallel operations.

Sliding window averages

These experiments involved calculation of a sliding window average data from a multi-color GIS image that contains an image of a lake and surrounding area in RGBA format. Each experiment involved two implementations of the sliding window average algorithm, one using a traditional implementation, and the other using the bit plane structure and operations. The three different sliding window experiments represent three different approaches to choosing data points to calculate the sliding window average. The image used in all three sliding window experiments is a four-channel image, with red, green, blue and alpha (transparency) bands, with size 1200 pixels by 1200 pixels. The image was originally in PNG format, and was converted to a bitmap before processing. Each pixel of each band contained data in integer format of bit depth 8.

Multiple sliding window patterns were considered, and averages calculated using the vertically structured approach and the addition/division algorithms previously described. The resulting outputs were compared to determine accuracy of the vertically structured approach. Additionally, computation of different window sizes was performed to ensure that the vertical implementation scaled similarly to standard implementations, with regards to window sizes.

Experiment: Sliding window estimate via four-corners method

The first approach is to sample the four corners of the window being considered, which can work well for small window size and it should approximate the full-window average method at those small window sizes. It has an advantage of having a straightforward implementation and requiring a constant number of input values for each window being calculated, independent of window size.

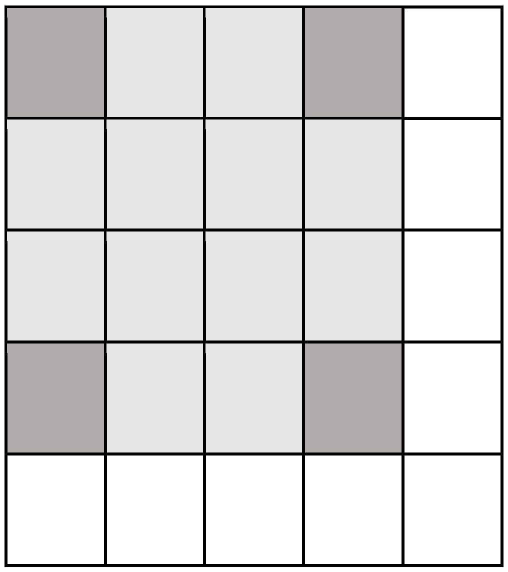

Figure 10: Sample sliding window (shaded) with four corner sampling (darker shading).

The sampling technique used focuses on speed in both cases and reduces the number of additions required to calculate the average of a sliding window. However, as the window size increases, the four corners may become less representative of the entirety of the window.

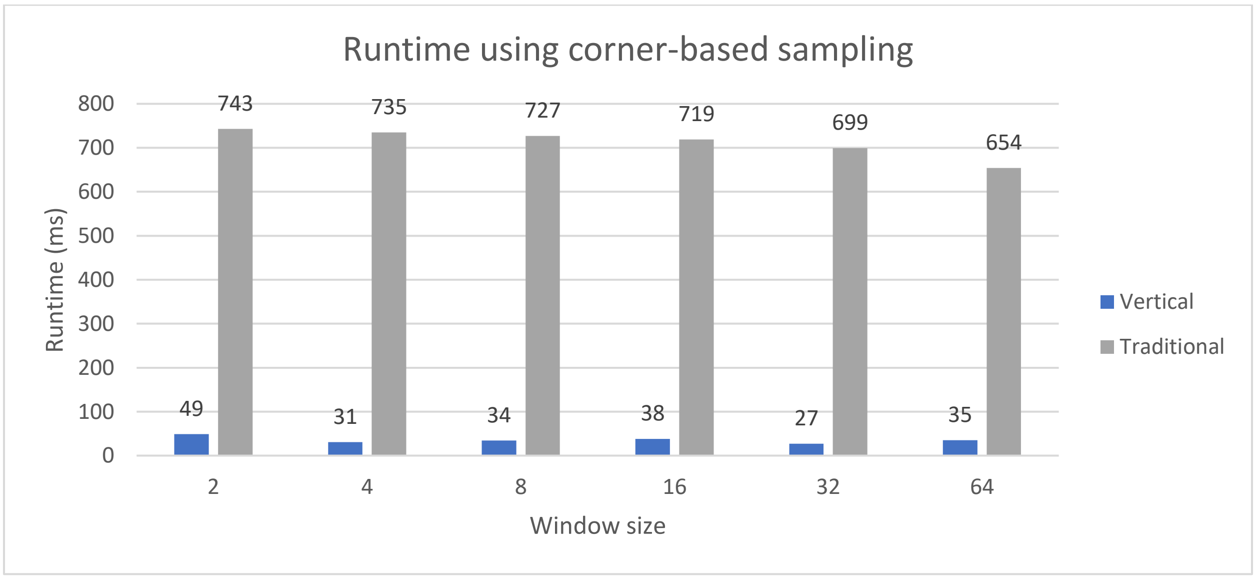

Figure 11: Comparison of runtime durations for corner-based sampling.

Results were reasonably consistent across window size variations, with the vertical approach requiring approximately, on average, 5% of the time to execute compared with the traditional implementation, resulting in processing that it 20 times faster.

Note the decrease in time for the large window sizes in the traditional implementation. This is a consequence of having fewer windows to calculate, as fewer windows fit within the image. Some potential windows on the right-most and bottom-most data could not be calculated, as the entire window did not fit on the image. For those data positions, there is not enough data to make the calculation given the window size. This does not apply to the vertical implementation in the same way, as operations are applied to the entire data set.

Experiment: Sliding window average via full-window method

This approach considers all data inside of a window and averages all values together for each window. The parameters for the image for which the average is calculated is the same as in the previous experiment. This will provide a more accurate result; however, it requires more calculations in the form of more addition operations. Larger windows in large image sets can be slow to process due to this introduction of additional operations.

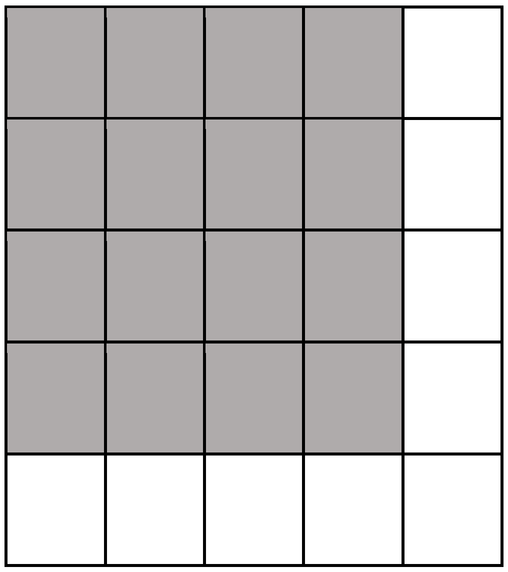

Figure 12: Sample sliding window (shaded) with all points sampled.

Full window sampling will provide a more accurate result but requires more calculations for larger window sizes. This sampling approach gains accuracy at the expense of processing time and memory usage.

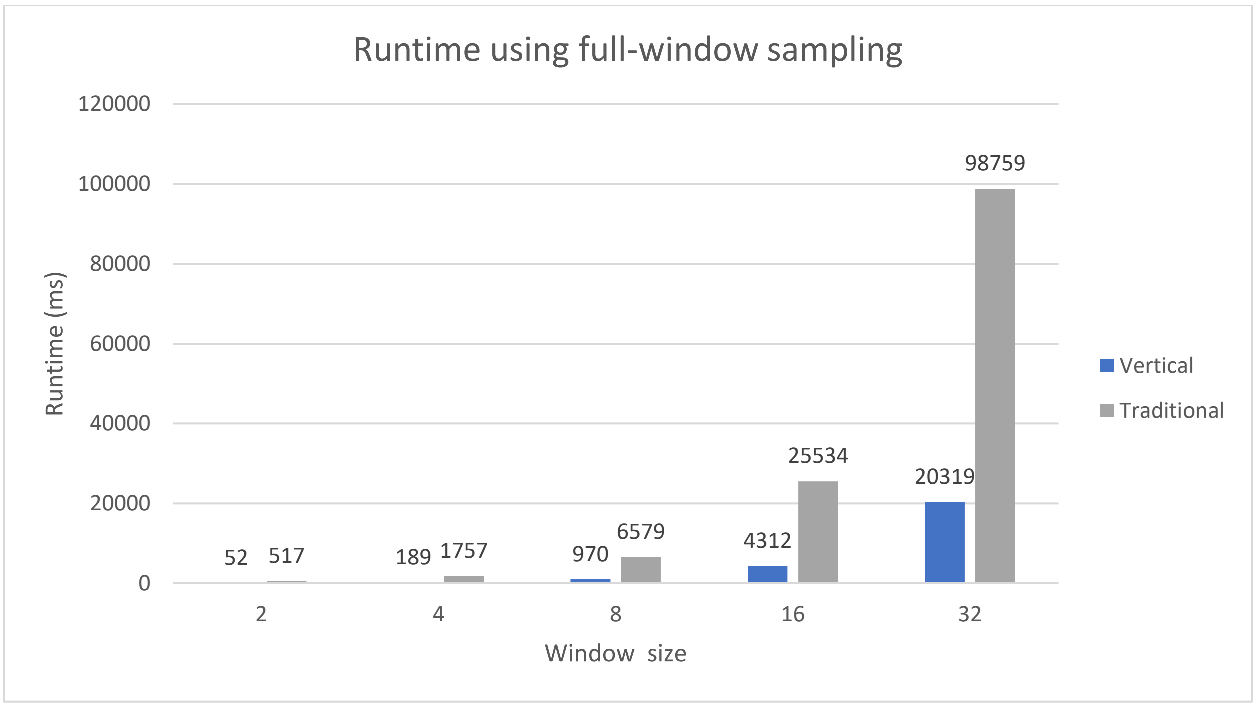

Figure 13: Comparison of runtime durations for full-window sampling.

The same exponential growth is present in both methods. The vertical method shows improvements in speed ranging between 4 and 10 times, depending on the window size. Note that the larger window sizes tend to show a lower improvement. This implies that there may be other factors that contribute to processing time at window samples of this scale such as overhead related to memory management.

Experiment: Sliding window average via 2^n iterative method

Full windows, of size 2^n can be calculated using an iterative method, where smaller windows of size 2 are calculated initially. The results of four adjacent averages of window size 2 can then themselves be aggregated to create results of a window size of 4, and subsequently those results are then averaged to create averages over larger windows (Denton, Gomes, & Franzen, 2018). This takes advantage of a property of averages, where a result of the subset of data can contribute to the result of the average of a larger set of data without loss of data or information.

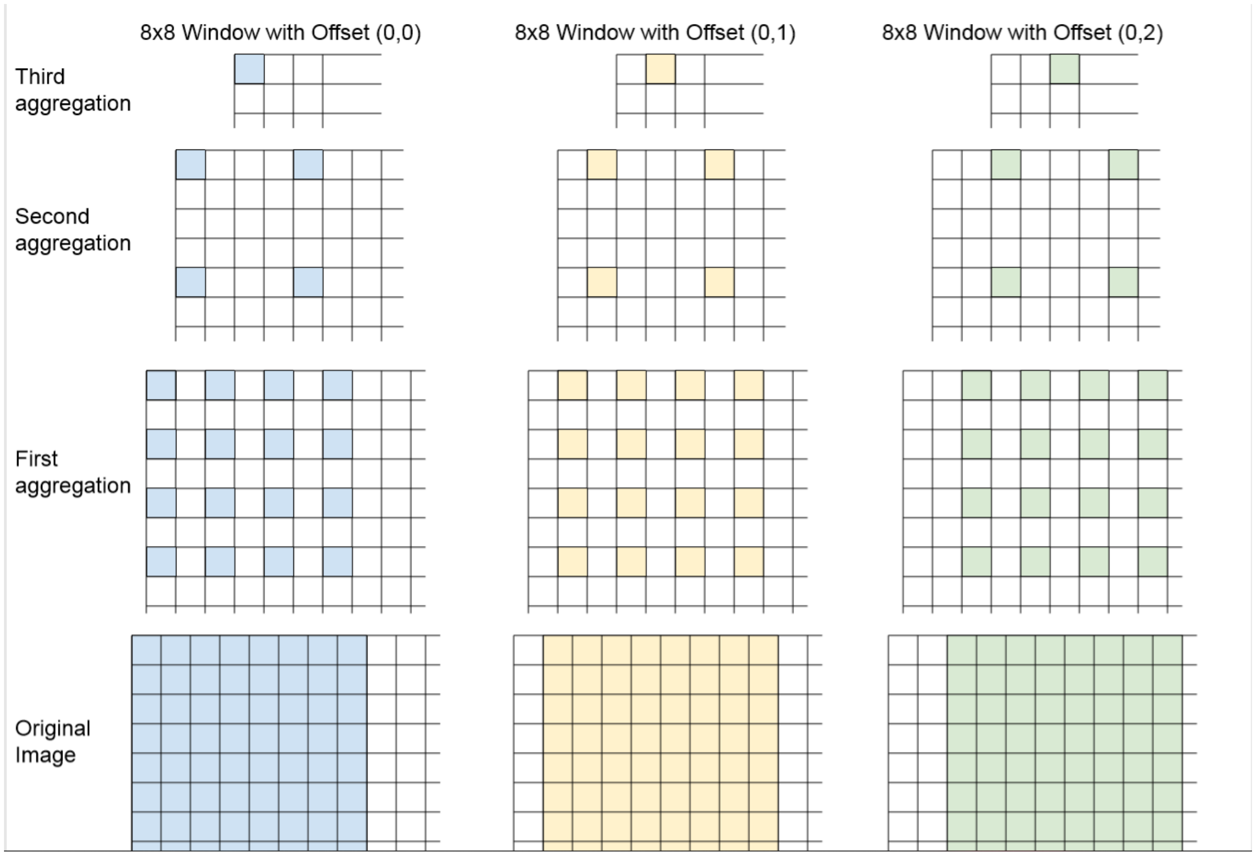
8x8 Window with Offset (0,0)
8x8 Window with Offset (0,1)
8x8 Window with Offset (0,2)
Third aggregation
Second aggregation
First aggregation
Original Image

Figure 14: An example of aggregation steps in the iterative aggregation approach. In each aggregation the shaded cell is the top-left cell of a sub-window.

Calculating sliding window averages in this manner involves a series of smaller aggregations, where each aggregation considers a smaller number of points. For a large window size, the first aggregation, for instance, will consider small sub-windows of the full window. Once the first iteration is completed, the second iteration considers the result of the first iteration to calculate larger windows. In an example of a window size of 8, the first aggregation has a window size of 2. The second aggregation calculates a window size of 4, but only need to use four data points since the sub-window averages have already been calculated in the first aggregation. This allows the second aggregation to need only 4 input values, but since each of those values represents a window of size 2, the second aggregation's output is the average of windows size 4. Similarly, the third aggregation considers four points, which are the results of sub-windows of size 4, in order to create the result for a window size of 8.

The optimizations of the 2^n-sized iterative window approach hold when compared to the other approaches, even when using the vertical structure and calculation process. The computation speed of a single iteration remains comparable to that of corner-based sampling. This shows the vertically structured data approach can be applied more generally and to different methods of choosing data points within a window, whether that method provides an approximation such as the window corner method, or a full average.

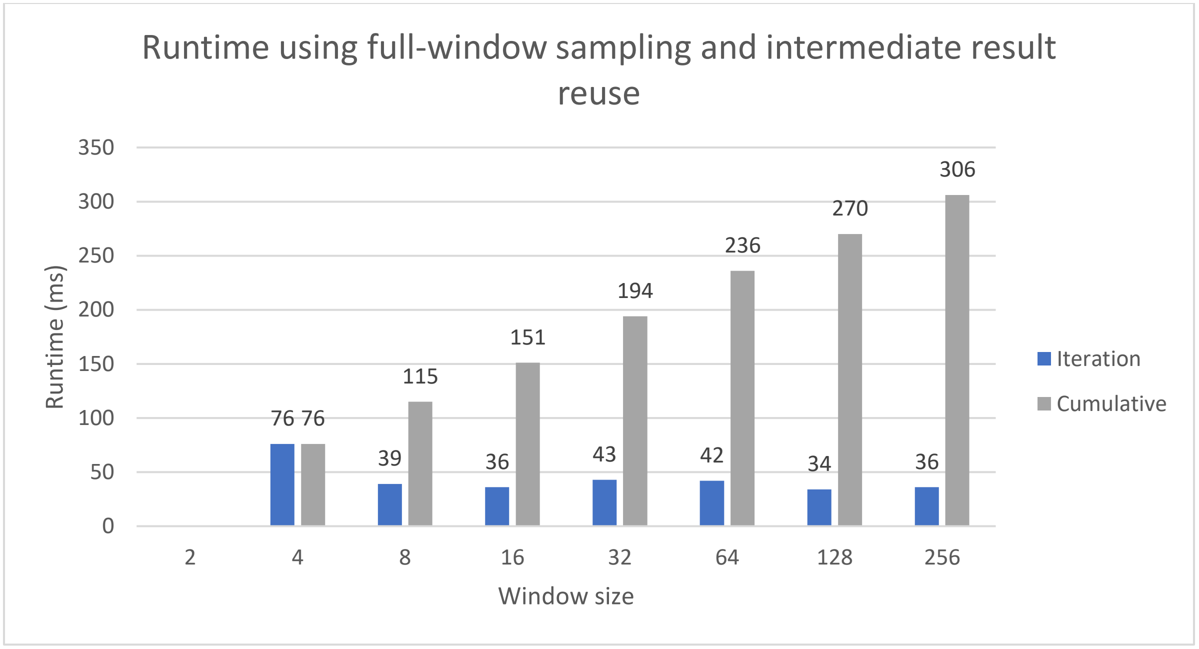

Figure 15: Comparison of runtime duration for full-window sampling with reuse of intermediate results.

Each individual iteration had a runtime comparable to that of the corner-based sampling of window size of 2, this is expected, as each new iteration is a calculation of an average of four numbers. The number of overall operations in a single iteration is equivalent to that of a corner-based sampling run or a full window sampling of window size 2. The average runtime of each iteration matches closely with those results. Each iteration, from a processing perspective, is identical to any other iteration, each considering four inputs and generating a single output value for every sliding window position. Though each iteration is functionally identical to each other, there was a slight variance in the runtime execution length for each iteration which can be attributed to minor environmental variance in the operating system and environment. The host runtime environment and operating system in which the experiment was run are subject to variances that can impact running tasks, these may include memory management and collection, execution of internal system tasks that may compete for CPU or other resources, or any other unpredictable tasks that may have run at the same time as the different iterations of the experiment run.

For the same image analyzed in previous experiments, this iterative approach was leveraged to calculate very large window sizes. This approach, as anticipated, was much more efficient at calculating large window sizes than approaches that calculate a full average of a large window in a single pass. In the implementations of the vertical adder, exceptionally large window sizes require larger amounts of memory as the individual planes are cloned and shifted in memory to perform the addition steps. Using this iterative approach, memory requirements for the vertical approach remain minimal, while computations remain fast.

Experiment: Variable bit depth

Because the bit plane approach represents data in vectors that each contain a single bit of data from all records, it's a natural extension to allow for the use of *n* bit vectors, where n is the bit depth of the data being stored. The consequence of this is that data of any bit depth can be easily stored and worked with, and algorithms written against the bit plane formats will naturally apply to data of any bit depth, and it not strictly limited to any constraints outside practical concerns of system memory.

To validate usage of large bit depths, multiple executions of sliding window averages of window size 4 were performed against data sets of multiple bit depths. For this experiment, bit depths of the form 2^n with $1 <= n <= 8$ were considered for simplicity and their common usage. Random data, representing a single band of data with bit depth n, for a simulated image with width and height 1200 was used as input data.

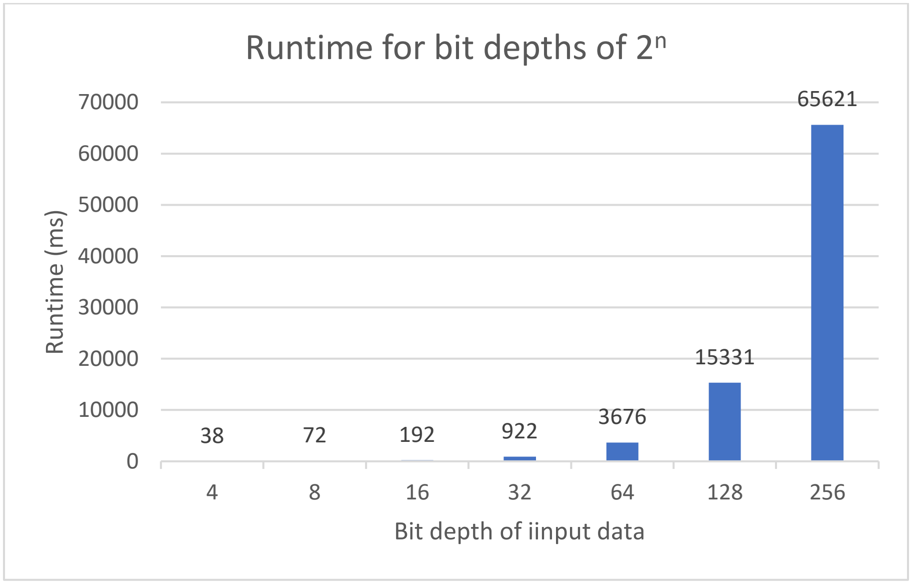

Figure 16: Duration of runtimes for a 1200x1200 pixel image, using a single data channel and window size of 4.

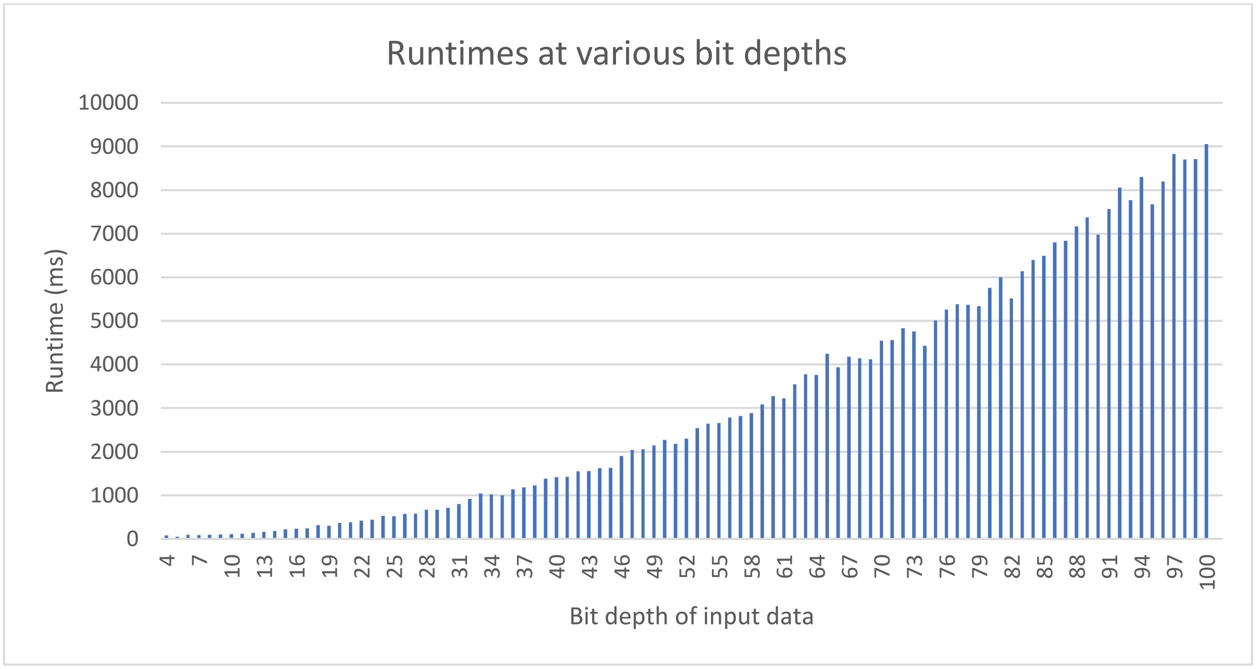

Figure 17: Runtime for bit depths up to 100 bits.

For small values of *n,* data fit comfortably in memory and the execution was smooth as expected. For larger bit depth values, more time was spent performing more steps in the addition operation and spent by the underlying software framework performing memory management operations. The addition operation requires one more step for each extra bit, and as bit depth increases, data size is impacted, and the system begins to require more memory as the bit depth increased. This is attributable to the simple adder implementation and unoptimized memory usage.

From this result of being able to handle data with large bit depth, it is conceivable that other numerical structures, say quadruple precision floating point numbers, could be implemented to allow for mass calculations as a vertical structure. Using hardware implementations as a guideline to implement pivoting, and with corresponding implementations of arithmetic operations, the handling of data at very high depth or precision is possible.

DISCUSSION

<u>Sources of improvement</u>

Much of the improvement shown in runtime speed of the algorithm is attributed to processor operations being requested to do more useful work per processor request, and to reduce the number of overall operations being requested. Consider an array consisting of common eight-bit positive integers, even the straightforward process of adding a constant value to each element requires each individual item to be read from memory, processed, and stored again. For data elements of *b* bits where *b* is not strictly the bit-depth of the processor, this leaves some bits unused during the CPU's operation as the *b* is expanded to fit a convenient register size of the CPU. To add values to each of 64 different 8-bit integers, the CPU would handle 128 reads, 64 add operations, and 64 writes of the result. Each of these operations is performed sequentially (read, add, write), and the addition operation will take multiple processor cycles to complete. This results in 256 operations, with the addition operation taking multiple cycles to complete.

In the vector-based structure, the addition operations are simple AND, OR, and XOR operations performed across multiple data items simultaneously, albeit a single bit at a time. This allows operations to be performed against a single bit of data across *n* data points at a time, where *n* is the amount of data a processor can handle in a single operation. As an example, for modern processors with 256-bit SIMD vector instructions, 256 bits of data from a vector can be processed at a single time, regardless of the bit depth of the original data. For an example of 256 different 8-bit integers, once in the vertical format, the addition operation uses five basic vector operations to add 256 bits at once. This involves two reads for the operand data, thus in this scenario, a total of sixteen reads of 256-bits each are required. For each of those eight bits of data to be operated upon, five operations are required for addition per bit across the full vector, and a total of eight more operations to write results back to memory. This results in addition of 256 different 8-bit integers in approximately 64 operations, most of which can be completed in minimal processor cycles as basic AND, XOR, and NOT operations are highly optimized and fast operations for hardware.

For modern processors with CPU-level vector optimizations, vector-based processing can handle 128 bits, 256 bits, or even 512 bits at a time in a single operation. Processor extensions such as SSE, AVX, AVX2, and AVX-512 (Intel Corp., 2020) focus heavily on allowing large vectors to be processed. Implementations of the vectors that underlie the experiments have support for these optimizations (Microsoft Corp., 2020), and so future enhancements in this area will continue to improve these vector-based algorithms using vertical data structuring. With AVX-512 extensions and the vertical vector structure, addition of values to each of 512 different 8-bit integers could theoretically be done in one read operation, 40 AND, OR, and XOR operations, and one write operation.

Tradeoffs

Using the binary vertical approach does currently have a few tradeoffs. When in the vertical structure, some of the operator implementations cannot work on the data in place, that is, operating against the original source of the data directly. For instance, the ripple-carry adder implementation must work against a copy of the data, incurring a tradeoff where more memory and extra memory allocation operations are required.

There is an expectation that the data is already in a vertical structure, whether it is stored in that manner or transformed at runtime. Transforming the data from the traditional format to and from the vertical format frequently incurs runtime penalties in the form of the pivot and unpivot operations. Depending on the usage, this may not always be appropriate.

<u>Current limitations of the approach</u>

So far, the vertical vector-based approach has only had a limited set of basic arithmetic operations implemented and tested. The implementation of addition is currently known to be suboptimal, and there are known approaches to optimize certain aspects of it in the same way it has been optimized for use by hardware adders.

Multiplication and division operations are currently limited to multiplicands and dividends whose values are multiples of two. Naturally, other implementations will be more complex but will allow for a wider range of potential operands and uses.

Only positive integer data is currently considered, floating point data stored in IEEE floating point format would make the implementation of pivoting and reverse pivoting significantly more complicated than the current implementation that supports positive integer data.

CONCLUSIONS AND FUTURE WORK

<u>Conclusions</u>

Sliding window techniques are used in GIS systems for several purposes. As GIS image data grows more detailed over time, due to higher resolution images and equipment, this processing can become more intense and require significant computing resources.

A new data structure, defined as a set of bit planes, for use with parallel data calculations was explored and investigated. This structure holds data in a vertical-style structure, like that of those that are often leveraged in vertical data mining. The data is pivoted at a binary level so that bits of equal significance are stored together in the resulting structure. This structure can be effectively acted upon by addition, division, and bit shift algorithms that resemble those implemented at a hardware logic level. By implementing this structure and algorithms in this manner, efficiencies can be obtained by obtaining more output data per operation than the traditional implementation of these operations on large data sets.

This data structure also can leverage algorithms that have implementations analogous to existing hardware-level arithmetic operations, while providing built-in benefits and enhancements. These include arbitrary bit precision of the initial data objects and efficient usage of memory at the CPU level, all while preserving the ability to parallelize calculation and retaining the ability to leverage already-known algorithms for common arithmetic operations.

Experiments show performance increases for simple operations as well as algorithms used to implement sliding window average operations that are commonly used for GIS data. Multiple styles of sliding window data selection were tested and examined, and the algorithms for the bit plane data structure were able to finish their calculations significantly sooner than a traditional implementation.

Future work

The vertical data approach combines elements from different techniques and other approaches, as such, there are multiple areas where the approach can be refined, expanded, or optimized.

The pivot and reverse pivot operations can be leveraged by specific GIS platforms via implementation as plugins and usage in that way can be studied and optimized. It's possible for data to be generally stored and persisted in the vertical format if significant processing is expected to be performed.

The same operations can be implemented at a hardware level, on GPU or FPGA hardware. These hardware architectures may have benefits that apply specifically to these transformative operations.

Arithmetic operations can be expanded via implementation of new operators and binary-level data storage formats that leverage the vertical structures. Where positive integer data was considered for the GIS scenario, negative integers and floating-point numbers of various precisions can also be good targets for storing in the binary vertical format. With these expanded number storage formats, operations such as floating-point division could be implemented similarly to how addition was implemented for binary vertical integer data

The experiment on higher order depths of integer values shows that the format and operators can also be leveraged as means to perform arithmetic against arbitrary-precision data. This is especially useful and important in the realm of floating-point decimals and high-precision operations.

REFERENCES

Denton, A. M., Ahsan, M., Franzen, D., & Nowatzki, J. (2016). Multi-scalar Analysis of Geospatial Agricultural Data for Sustainability. *Proceedings of the 2016 IEEE International Conference on Big Data* (pp. 2139-2146). Wachington, DC: IEEE.

Denton, A., Gomes, R., & Franzen, D. (2018). Scaling up Window-Based Slope Computations for Geographic Information System. *IEEE International Conference on Electro/Information Technology (EIT)* (pp. 554-559). Rochester, MI: IEEE.

Gomes, R., Denton, A., & Franzen, D. (2019). Quantifying Efficiency of Sliding-Window Based Aggregation Technique by Using Predictive Modeling on Landform Attributes Derviced from DEM and NDVI. *Internation Journal of Geo-Information*.

Han, J., & Kamber, M. (2006). *Data Mining: Concepts and Techniques*. San Francisco, CA, United States of America: Eslevier.

Intel Corp. (2020, January 7). *Intel Instruction Set Extensions Technology*. Retrieved April 7, 2020, from https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html

Jensen, J. R. (2005). *Introductory Digital Image Processing* (Third ed.). Pearson Prentice Hall.

Longley, P. A., Goodchild, M. F., Maguire, D. J., & Rhind, D. W. (2011). *Geographic Information Systems & Science* (Third Edition ed.). Hoboken, NJ, USA: John Wiley & Sons.

Microsoft Corp. (2020, March 30). *runtime/BitArray.cs at master - dotnet/runtime - GitHub*. Retrieved April 05, 2020, from https://github.com/dotnet/runtime/blob/master/src/libraries/System.Collections/src/System/Collections/BitArray.cs

Null, L., & Lobur, J. (2010). *The Essentials of Computer Organization and Architecture*. Jones and Bartlett Learning.

Pradhan, S. (2001). Crop area estimation using GIS, remote sensing and area frame sampling. *International Journal of Applied Earth Observation and Geoinformation, 1*, 86-92.