Optimizing and Analyzing the Performance of Hough Transform using Single Instruction Multiple Data Instructions and OpenMP

Rashmi Anil, Ishaan Gupta
Carnegie Mellon University, Pittsburgh, PA, 15213
{ranil, ishaang}@andrew.cmu.edu

Abstract

This paper introduces an optimized implementation for the Hough Transform using SIMD, OpenMP and certain hardware optimizations that significantly speed up the algorithm. Our algorithm is bench marked against OpenCV's *HoughLines()* - a fast serial implementation of this algorithm - on Bridges, an $x86\_64$ multi-core machine. With the combination of SIMD and OpenMP, we were able to achieve a speedup of about 8x over OpenCV's *HoughLines()*.

Introduction

The Hough Transform is a feature extraction technique that is primarily used in image analysis and digital image processing within computer vision. It is a popular technique to detect any shape that can be represented mathematically. This algorithm has been widely used for straight line detection in low resolution images and still images but suffers from execution time and resource requirements. The Hough transform is a robust technique that works well even in the presence of noise and occlusion. While this algorithm is an effective method to detect lines in an image, the Hough Transform is known to be notoriously slow and is commonly not used for real-time applications.

A complete application of the Hough Transform to detect shapes in images usually consists of several steps:

1. Smoothing the image
2. Edge Detection and Thresholding
3. Voting in Hough Space, finding local maximums identifying lines
4. Displaying the detected lines

These steps are illustrated in figure 1 below. In this paper we will focus only on the third step: voting and identifying the local maximums in the Hough space. This is primarily because there are many different methods one could use to compute the results of steps 1 and 2. For example, there are many filters that can be used to detect edges such as the Sobel filter or the Canny filter. Since these algorithms can vary and already use highly optimized convolutions, we decided to focus our efforts to making the voting procedure more efficient. In this paper, we dive deep into the methods we used to optimize the part of the algorithm that updates the votes in the accumulator and identify the lines. We describe in detail the kernel design, parallelization techniques used to optimize memory accesses, and discuss the speedup we achieved. We compare the performance of our implementation against the fast serial implementation that is used by the OpenCV library. Specifically, we compare the performance of our algorithm to OpenCV's *HoughLines()*. OpenCV's *HoughLines()* computes the results of voting in the Hough Space, find the local maximums and identifies the lines in the image. We extracted the *HoughLines()* to use as a baseline performance metric and correctness checker for all of our implementations of this algorithm.

Background

Algorithm Overview

The Hough transform takes as input the output of an edge detector - a binary matrix the same size as the original image where each element is a 1 if an edge is present in the original image at the corresponding pixel and 0 otherwise. It then uses a two-dimensional array – the accumulator – to collect votes for different possible lines in the image, each of which is represented in polar coordinates with the values (ρ, θ). The width of the accumulator matrix is equal to the number of θ values that will be iterated over in the Hough Space which is 180, since our granularity for θ is 1 degree. The height of the accumulator matrix is equal to approximate upper bound for the number of ρ values that can occur which is calculated as $2 * (w + h) + 1$ where w and h are the width and height of the image. Then each element of the accumulator matrix corresponds a value of ρ and θ which represents the line $\rho = x \cos\theta + y \sin\theta$. The value of that element in the accumulator is the number of votes that line received.

For each nonzero index (x,y) of the input matrix, the Hough transform algorithm calculates the parameters (r, θ) of each of the lines passing through that point, then looks for the accumulator's bin that the parameters tho and theta fall into, and increments the value of that bin. By finding the bins with the highest end values in the accumulator matrix, typically by looking for local maxima in the accumulator and then sorting by accumulator value, the most likely lines can be

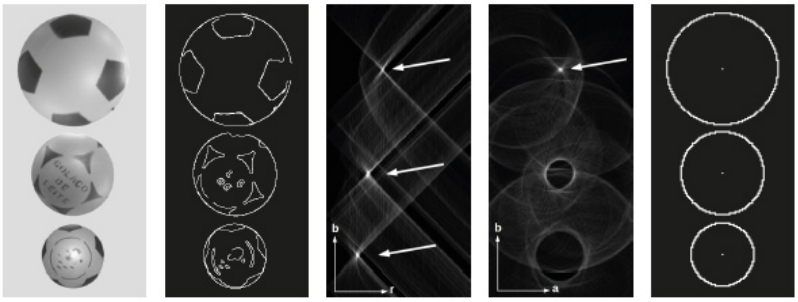

Figure 1: *Example of Hough Transform algorithm for detecting circles. (a) The image is smoothed using Gaussian filter. (b) Edges are detected using Canny filter. (c) Accumulated votes in 3D parameter space are shown in planar sections with (d) arrows highlighting local maximums. (e) The parameters indicated by the local maximums are used to reconstruct the circles.*

extracted.

Algorithm Outline

A description of the algorithm is as follows:

```
1  def hough_lines(M):
2      Initialize accumulator H to all 0s
3      for each point (x,y) in M:
4          if M[x][y] is 1:
5              for theta = 0 to 180:
6                  ρ = x cos(θ) + y sin(θ)
7                  H(θ, ρ) = H(θ, ρ) + 1
8      Find values of (θ, ρ) where H(θ, ρ) is a
           local maximum
9      Sort these pairs (θ, ρ) in descending
           order by the number of votes they
           received in H
10     The N most likely lines in the image
           are given by ρ = x cos(θ) + y sin(θ) for
           the first N pairs in the sorted list
```

In the above algorithm for Hough Transform, the core computation occurs between lines 3-7. While this part of the algorithm may seem small, it contains several regions that are major bottlenecks for the performance of the algorithm. For example, there are several reads that occur into the edge map M that occur on line 4 and several read and writes that occur into the H accumulator matrix on line 7. While the memory access pattern for the former is predictable, memory accesses into the accumulator matrix are highly dependent on the value of ρ and do not form a pattern. If the accumulator data that is being accessed is not in the cache, then we have one load and one store operation per θ per edge pixel. This can severely negatively impact the performance of the algorithm. In terms of computational bottlenecks, on line 6, we have a heavy compute operation that may slow down the speed of the algorithm. The value of ρ is dependent on 2 independent multiplications. Hence, the value of ρ is dependent on both multiplications computed and added to ρ sequentially.

Since this block of code will be used many times when trying to find lines, this will be the main kernel that we will be optimizing. Within these lines, are many independent operations which provide us with various opportunities for optimization. Specifically, lines 3 and 5 are lines where parallelism can likely be exploited. In the outer for loop, we are looping over every single edge pixel in the image. These per-pixel operations can be performed independently. The inner for loop loops over different values of theta and can also be performed independently. Due to the independence between the loops, we feel that SIMD parallelism and OpenMP parallelism between cores can likely be exploited to achieve significant speedup.

Approach

We aimed to optimize the Hough Transform algorithm on the *Pittsburgh Super-Computing* Bridges machine. This machine has an $x86\_64$ architecture and has 28 cores. We chose this machine because it could easily use SIMD and OpenMP and has many cores, making parallelism using OpenMP possible.

We first started with the fast serial implementation of the Hough Transform implemented in OpenCV and copied it over to a file in Bridges to benchmark. We used this implementation as the base line against which we compared our parallel solutions. Then we tried to speed up this serial implementation with a SIMD implementation, an OpenMP implementation, and an implementation which combined SIMD and OpenMP.

SIMD

Using the information about the independence of the core computation of the Hough Transform, we decided to use Single Instruction Multiple Data (SIMD) instructions as a first step towards turning this core computation into a high performance kernel. This decision was made primarily because the computations inside the kernel are independent from each other. Each edge point needs to loop though 180 different θ values. Hence, when computing the ρ values for each theta via $\rho = x * \cos(\theta) + y * \sin(\theta)$, since this computation involves the same instruction, we can input multiple θ values and make use of SIMD to make these computations faster. The architecture that we used to run our optimized version of Hough Transform is $x86\_64$. Specifically,

we used Intel's Haswell micro-architecture. This architecture provides us with 16 SIMD registers that we can use sparingly to improve the performance of the kernel. Since the magnitude of the values for $\sin(\theta)$ and $\cos(\theta)$ range from 0-1 and the values x and y represent the indices for the edge points in the edge map, we know that the value for ρ only needs to be a single precision value. Hence we can use single precision SIMD vectors that are 256 bytes in size which has 8 lanes. This means that inside the inner loop, we can perform 8 simultaneous computations of ρ spanning over 8 consecutive values of θ.

As mentioned earlier, one of the important and reoccurring computation in the kernel is the calculation of ρ. To reiterate, $\rho = x * \cos(\theta) + y * \sin(\theta)$ where x and y and the edge map coordinates and θ is the loop variable being iterated over. In this reoccurring computation, we can see that there are 2 multiplications and 1 addition. We can rewrite this line to be:

```
1 rho = 0
2 rho += x * cos(theta)
3 rho += y * cos(theta)
```

By rewriting this line in this particular fashion, we can see that we have 2 sets of one addition paired with one multiplication. This setup is perfect for using fused multiply add (FMA) instruction set. The FMA instruction set is an extension to the 256-bit SIMD Extensions instructions in the x86 microprocessor instruction set. It can be used to simultaneously perform a floating point multiplication and addition. In the $x86_64$ instruction set, the latency of one multiplication is 5 cycles and the latency of an add is 3 cycles. However, an FMA operation also has a latency of 5 cycles. Hence, we determined that it is more beneficial to us in terms of register usage if we write this operation in such a way that we use 2 FMA operations instead of 2 individual multiplies and 2 separate adds. We also converted the rest of the operations within the innermost loop to SIMD instructions so that we could operate on 8 θ values at the same time. This gave us our initial SIMD implementation which unfortunately gave only a 2x speedup over the serial code.

While SIMD instructions and FMA instructions provide a great start to optimizing the core computation of the Hough Transform algorithm, to take it one step further and really turn it into a high performance kernel, we need to help the compiler help us by taking advantage of compile optimizations. Some of these optimizations include loop peeling and loop unrolling. The architecture $x86\_64$ contains 2 FMA units each with an operational latency of 5 cycles. To take full advantage of these 2 FMA units, we need to pipeline at least 10 *independent* operations. While the way we rewrote the computation of ρ provides us with 2 FMA operations, they are not independent from each other. Since both computations of $\sin(\theta)$ and $\cos(\theta)$ have to update the value of ρ, they have to be serialized. However, if we perform some loop unrolling (i.e we calculate multiple ρ values - ρ_1, ρ_2, ... ρ_{10}), then we can pipeline 10 independent FMA operations and fully utilize the machine.

Putting all of this information together, we designed an updated kernel by unrolling the outer and inner for loops (see Algorithm Design for pseudo-code) in such a way that we would have 10 independent FMA operations. We unrolled the outer loop 5 times so we iterate over 5 independent edge pixels in one iteration and the inner loop 2 times with the use of 256 byte single precision SIMD registers so we iterate over 16 consecutive θ values at once. For each of the resulting ρ vectors, we compute the index inside the accumulator matrix and serially update the value. In order to unroll the outer loop that iterates over the edge pixels, we iterated through the entire edge map and stored all of the entries with a value of 1 into a vector. This gets rid of the if statement on line 4 and allows for easy parallelization on the outer loop. However, when we timed the code, we realized that $3/4$ of the time was spent on looping through the edge map and storing edge pixels. While pipelining improved performance significantly, it was offset by this overhead. To try and improve performance, we got rid of this change and only parallelized over the inner loop by unrolling theta 32 times. While this method only allows us to pipeline 4 independent FMA instructions, it gave us slightly better performance than the previously mentioned techniques. Adding these optimizations to our original SIMD implementation increased our speedup from about 2x to 2.5x.

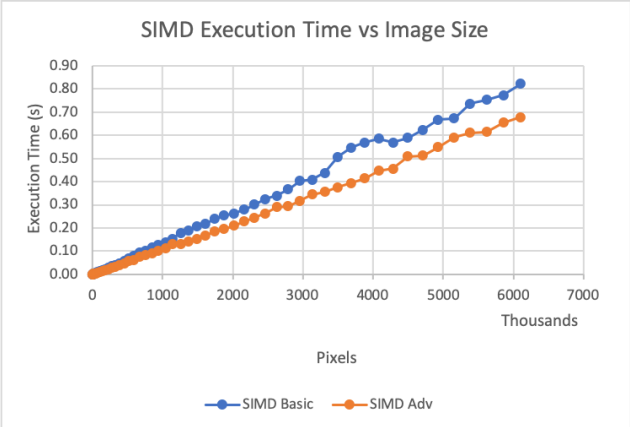

Figure 2: *Execution time for a basic SIMD implementation and an advanced SIMD implementation that uses pipelining*

Memory Optimization

In terms of memory optimizations, the main bottleneck in our code is the serial memory updates to the accumulator matrix. While we parallelized over several θ values, every single rho value that is computed is updated serially in the code. That is, even though the SIMD vector widths are 8 ρ's in length, each of the ρ values are extracted from the vector and updated serially. Since these values are random and don't really have an access pattern associated to them, we could not find a way to get round this serialization. We thought of using a gather or scatter vector operation because that would help us avoid extracting each one of the ρ values. However, after doing some research, we found that this operation is very slow and is implemented serially under the

hood. Hence, we decided to keep the serial update. Since this is accessing random regions in the accumulator matrix which is located in heap memory, we realized that there is going to be a lot of cache misses and many slow loads and stores to memory.

One possible solution to mitigate this issue is to tile the image. The accumulator matrix's dimensions are highly dependent on the width and the height of the image that are passed in. Hence, if we tile the image such that the dimensions that we are passing into the kernel cause the accumulator to fit into the L1 cache, then the serial updates should not take much time to run. We did not explicitly implement this in our code. However, this is a possible optimization that can be made.

OpenMP

Even after trying to make the SIMD as fast as possible, we still only achieved about a 2.5x speedup over the serial code. While we were operating on multiple θ values with SIMD, we saw that the outer loop iterations across all the pixels could be parallized across cores using OpenMP. To see if this would result in a better speedup, we created an OpenMP implementation for our initial serial code. To do this, we first created a parallel section around the loops that updated the accumulator used the #pragma omp for collapse(2) directive to parallize over all the pixels. We also decided to use dynamic scheduling because the work done in each iteration varied greatly depending on the value of the check in the if statement. If the pixel was not an edge pixel, no work would need to be done and if the pixel was an edge pixel, we would have to the the rho calculation and accumulator update. In practice, we saw this give a slight increase in performance over static scheduling.

We also realized that since in each iteration, we had to do a read modify write of one index in the accumulator, the update needed to be protected when multiple cores were accessing it and we initially put this update in a critical section. However, when we tested our code, we saw that it was actually slower than the serial code due to high contention. To improve our design, we realized that we could replace the #pragma omp critical with #pragma omp atomic since the increment operation is supported to be atomic by OpenMP. This simple change gave us a slight speedup (1.7x) over the serial implementation. However, we saw that about 98% of the execution time was still occurring in the update to accumulator due to these writes being serialized.

To make the updates to the accumulator more parallel, we decided that each thread should update its own local copy of the accumulator matrix. Then we wouldn't need to worry about concurrency when updating the accumulator. As a result, each processor could update its accumulator in parallel with other threads. However, the downside is that after each thread computed its local copy of the accumulator matrix, each thread needed to add the local copy to the global accumulator matrix.

In addition, we saw that the second part of the Hough Transform in which we find local maximums in the accumulator matrix had independent iterations. This was because this part of the code checked if each value of the accumulator was larger than its neighbors which could be done in parallel across all elements of the accumulator. To use this to our advantage, we also included this second part in the parallel section. Each time we found a local maximum, we were initially pushing the position of the local maximum onto a vector. However, if we wanted to keep doing this when parallelizing the loop iterations, we would have to place the push operation in a critical section. Trying this resulted in poorer speedup than not including this second part in the parallel section so we tried to think of a way around this.

Our solution was to once again try to convert the critical section to one where we could use OpenMPs atomic construct which is much faster. To do this, we converted the vector into an array and kept track of the index of the next empty index in the array. Then, we could atomically read the current value of the index and increment the index at the same time using #pragma omp atomic capture. This ensured that only the current thread read a single value of the index to write to and all subsequent reads to the index would be a larger index. Then the current thread could write that index without having to worry about another thread writing to it. Once we added these optimizations, our OpenMP implementation had about a 3.5x speedup over the serial code, which wasn't great but was still better than the serial code.

SIMD + OpenMP

While the speedups of the SIMD implementation and OpenMP implementations were ok, we realized that we could combine the 2 approaches and potentially get even more speedup. To do this, we replaced the inner most loop of the OpenMP implementation we had with the fast SIMD implementation. After doing this combination, we reached a total speed up of about 8x.

Results and Discussion

Experimental Setup

To benchmark the performance of our SIMD implementations, we ran the code several times changing the size of the input edge map to scale the amount of work. We did this by resizing our image and varying it from 400 pixels to a little more than 6,000,000 pixels.

To benchmark all the implementations involving OpenMP, used the largest image (2520 x 2520) we used to benchmark for SIMD as our imput image. We then varied the number of processors from 1 to 28. This was because the machine that we used to benchmark our code had 28 processors available for use.

To analyze the execution time and the speedups of the approaches we took to optimize the algorithm, we compare them to the performance of OpenCV's *Houghlines()* function that ran completely serially. We timed the execution of all of the different implementations using wall clock time.

SIMD

From the speedup graph in figure 4, we can see that initial SIMD implementation achieved about a 2x speedup while the improved SIMD achieved a 2.5x speedup. This was much less than the ideal 8x speedup expected when using

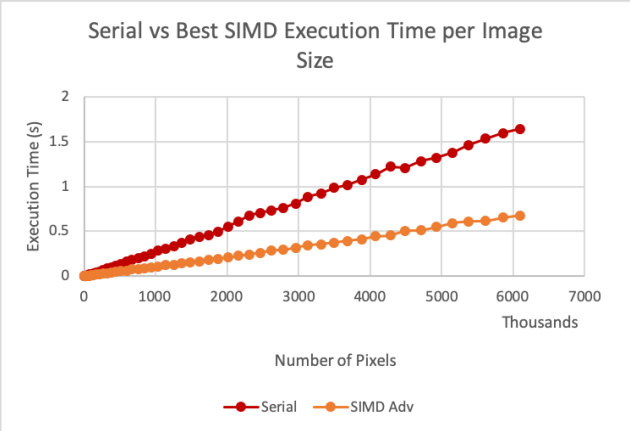

Figure 3: *Execution time for the fast serial code compared to the best SIMD*

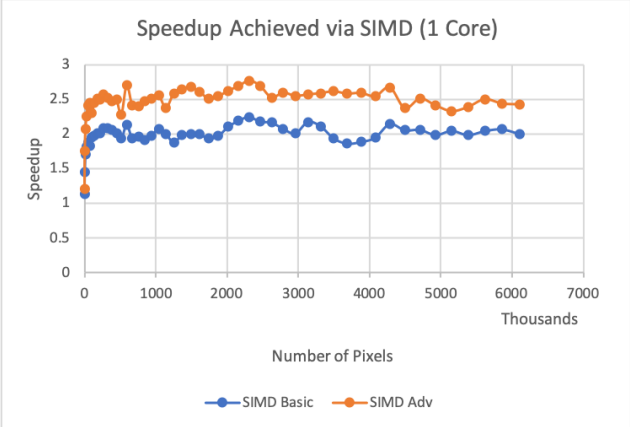

Figure 4: *Speedup achieved for basic SIMD and advanced SIMD*

8 wide vectors. To understand why, we timed different sections of our code and found that 95% of the execution time was spent in the core computation that looped over the pixels to increment the acucmulator and 5% in the other parts including finding local maximums in the accumulator and sorting the local maximums. This was good since most of the time was spent in the section we were parallelizing.

However, further analysis revealed that of the time in the core computation, 98% of the time was spent in the lines incrementing the accumulator matrix. This was the part of the core computation that we did not find a way to parallelize using SIMD and was occurring completely sequentially. As a result, no matter how much we sped up the remainder of the loop, by Amdahl's Law our speedup would not improve unless that updates to the accumulator could occur in parallel as well. We hoped that this issue could be solved using OpenMP.

OpenMP

The speedup graph for our OpenMP implementation shows that our speedup did not reach the ideal scenario which would be linear speedup. The speedup was near linear up till about 4 cores, but then decreased slightly as more cores were added.

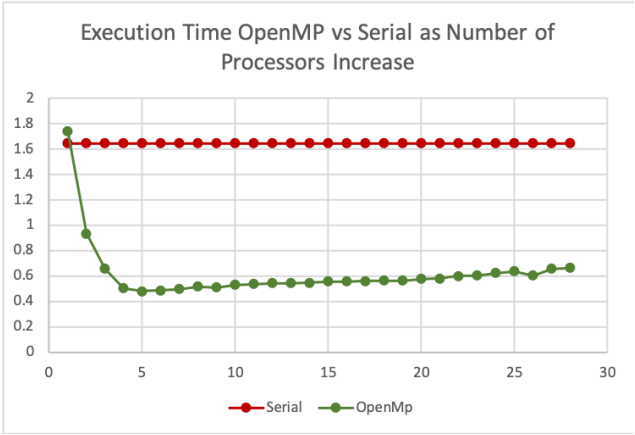

Figure 5: *Execution time for an OpenMP implementation of the algorithm compared to the fast serial implementation*

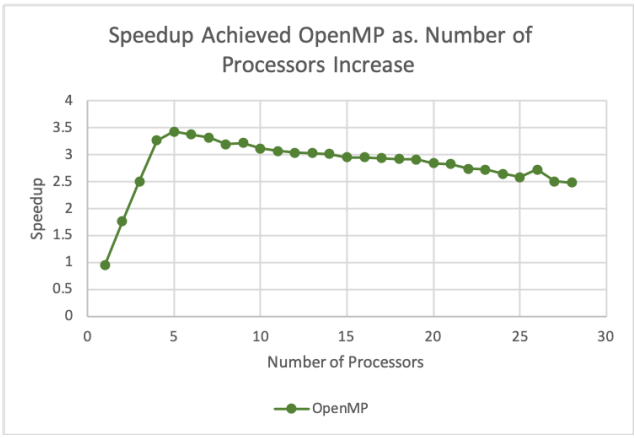

Figure 6: *Speedup achieved for OpenMP Implementation*

To determine why our implementation wasn't scaling, we once again analyzed where execution time was being spent. We saw that 86% of the execution time was spent in the update to the local accumulator matrix. Since this code was reading and writing to a per thread copy of the accumulator matrix, it was not slowed down by any concurrency primitives. This shows that the update to the accumulator was actually bandwidth limited. Up until, 4 cores, we were not bandwidth limited so the implementation scaled well. However, after adding the 5th core, out code is being limited by the rate of reads and writes in can do.

Taking only the memory bandwidth into account, we would expect the speedup to plateau instead of slightly decrease as we see. However, because we increased parallelism

by having each thread write to a local copy of the accumulator, we had to add an additional step where each thread adds its final local copy to the global accumulator. As the number of processors increased, the work needed to be done in this step also increased. Because this step scales linearly with the number of processors and the previous step was not being sped up by adding more processors, the time taken increased as the number of processors increased. If we had not been memory bandwidth limited and achieved more speedup in the previous step, it could offset the small extra cost of the adding to the global accumulator step.

SIMD + OpenMP

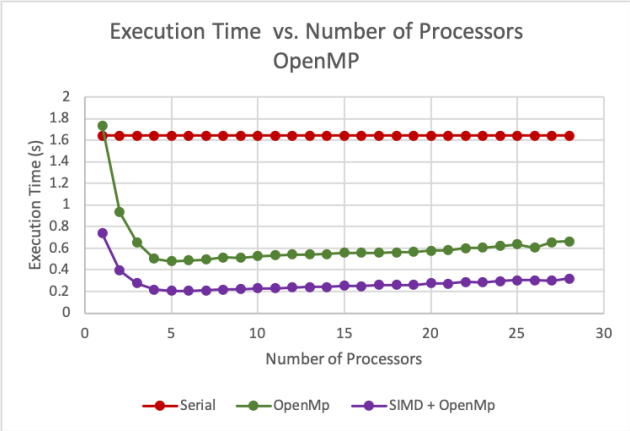

Figure 7: *Execution time for OpenMp and OpenMp combined with SIMD implementation. Significant improvement is seen over serial implementaion.*

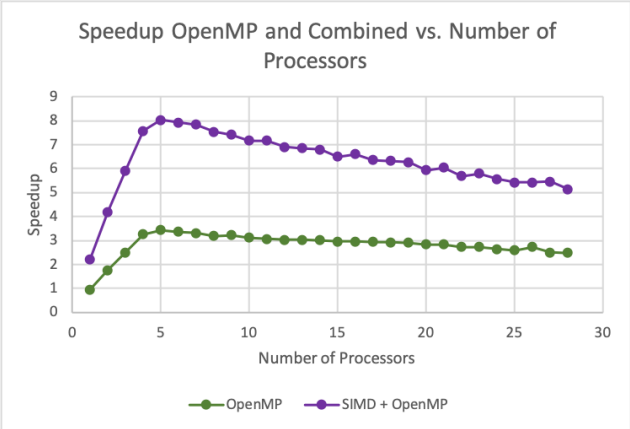

Figure 8: *Speedup achieved for combined OpenMP + SIMD implementation vs. just using OpenMP.*

The speedups for the combined SIMD and OpenMP were expected given the speedups from SIMD and OpenMP individually. Our SIMD implementation gave us about a 2.5x speedup and our OpenMP implementation had a speedup of about 3x to 3.5x. Our combined implementation had a speedup equal to he speedup of the SIMD implementation times the speedup of the OpenMP implementation. This was about an 8x speedup at the peak of our OpenMP implementation's peak performance at 5 cores.

Future Directions

Our results show that our efforts to increase parallelism were hindered due to our program being memory bandwidth limited. Future work on this project needs to focus on how to make the most out of the memory bandwidth we have. Some ideas to look into would be trying to reduce the number of memory accesses by potentially doing extra recomputation. Also, we can try to find a way to minimize the number of cache misses when accessing the accumulator matrix. This is difficult with the current structure of the code since accesses to the accumulator matrix are to random indices. Another potential idea for improvement is to use multiple machines and parallelize with OpenMPI between machines. Having more machines could potentially increase the memory bandwidth.

Acknowledgements

We would like to thank Professor Railing and Professor Mowry for providing us with the foundation of tools and techniques necessary for us to try and optimize this algorithm.

Work Distribution

The work was distributed pretty evenly between both partners and everyone contributed their share. The table below lists the work done by each member of the team.

Item	Team Member
Serial implementation of the algorithm inspired by OpenCV library.	Rashmi
SIMD Basic Implementation - Involved figuring out how Intel AVX intrinsics work and how to use it	Rashmi and Ishaan
SIMD Advanced Version that used pipelining and hardware specs	Rashmi
OpenMP version	Ishaan
Combined version	Ishaan
Scripts to benchmark all of the code	Rashmi
Presentation and Report	Rashmi and Ishaan
Total Division of Labor	50:50

References

1. OpenCV Documentation:
https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html
2. OpenCV Github:
https://github.com/opencv/opencv/blob/master/modules/imgproc/src/hough.cpp