Optimizing and Analyzing the Performance of Hough Transform using SIMD, OpenMP and OpenMPI

Rashmi Anil, Ishaan Gupta

[https://ishaan66.github.io/]

Summary

For our project, we plan on optimizing and analyzing the performance of the Hough Transform algorithm – an algorithm commonly used for line and shape detection in computer vision – using AVX compiler intrinsics, OpenMP and OpenMPI against the OpenCV implementation on a x86-64 CPU.

Background

The Hough transform is a feature extraction technique that is primarily used in image analysis and digital image processing. 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. While this algorithm is an effective method to detect lines in an image, the Hough Transform is known to be notoriously slow and is not used for real-time applications.

How it works

The algorithm uses a two-dimensional array – the accumulator – to detect the existence of a line described in polar coordinates using the Hesse normal form as . The dimension of the accumulator matrix is equal to the number of unknown parameters and . For each pixel at *(x,y)* and its neighborhood, the Hough transform algorithm determines if there is enough evidence of a straight line at that pixel. . If so, it will calculate the parameters (r,θ) of that line, and then look for the accumulator's bin that the parameters fall into, and increment the value of that bin. By finding the bins with the highest values, typically by looking for local maxima in the accumulator space, the most likely lines can be extracted.

The final result of the Hough transform is a two-dimensional array with the same dimensions as the accumulator. Each element of the matrix has a value equal to the sum of the points or pixels that are positioned on the line represented by quantized parameters (r, θ). So, the element with the highest value indicates the straight line that is most represented in the input image.

High-level Algorithm Outline of the core of Hough Transformation

```
·       Initialize accumulator H to all zeros
·       For each edge point (x,y) in the image
·               For θ = 0 to 180
                    ρ = x cos θ + y sin θ
                    H(θ, ρ) = H(θ, ρ) + 1
·       Find the value(s) of (θ, ρ) where H(θ, ρ) is a local maximum
·       The detected line in the image is given by ρ = x cos θ + y sin θ
```

There are many independent operations involved in the Hough transform algorithm which provides us various opportunities for optimization. The lines highlighted in yellow are lines where parallelism can likely be exploited. In the outer for loop, we are looping over every single 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.

The Challenge

From the above snippet, we can see that we can easily parallelize over either the outer for-loop which involves iterating over the pixels in the image or the inner for loop which involves iterating over 180 fixed theta values. However, to get a desirable speedup, parallelizing over one of these loops is not enough. We need to consider more subtle aspects of our code such as caching and pipelining.

In terms of memory accesses, accessing each coordinate in the image and checking to see if that coordinate has an edge involves a lot of memory accesses. For each of these image pixels, we need to do heavy computation and update the accumulator matrix. Depending on how we choose to implement this, updating the accumulator matrix also involves a lot of memory accesses. In order to get good speedup, we would have to find a way to mitigate these memory accesses.

Inside the loop, we see that there are a lot of dependent operations. For example, we need to compute the value of ρ before we can update the accumulator matrix. To get the speedup we desire, we need to find a way to take advantage of the CPU to do these operations in a more efficient manner. This means reducing communication costs of the memory accesses and pipelining instructions.

Resources

We plan on using the Hough Transform from the OpenCV library as a baseline for benchmarking. However, we plan on implementing the Hough Transform from scratch ourselves for AVX intrinsic, OpenMP and OpenMPI. We will be using the OpenCV implementation to ensure correctness. Since both of the members of our team have taken courses in computer vision, we will not be referencing any additional textbook or papers.

Access to the Bridges machine will be useful since it has many cores so we can fully test speedup as the number of cores increased. Also, it has more MPI cores available than the latedays machine so we can fully test an OpenMPI implementation. As an additional benefit, we already have access to the Bridges machine so we don't have to do extra work to obtain access.

Goals and Deliverables

Plan

Our goal is to speedup the Hough Transform from the fast serial implementation found in the OpenCV library. We want to compare and contrast different techniques to parallelize the algorithm to see what works best. Our first technique to speedup the Hough Transform is using SIMD using AVX intrinsics. Since we are doing the same operation inside the for loops, we can do the fused multiply add AVX instruction to do multiple of the iterations of the loop together. We need to determine how many hardware units there are that perform a fused multiply add to maximize the work we can do at the same time.

Our second technique will be to use OpenMP for speedup. The H accumulator array is a shared global array all threads will be writing to and the image pixels is a shared array all threads will be reading from. We can split the loop iterations between threads and determine how to best use locks around the shared memory to maximize speedup.

Once we have implemented both versions of the parallel Hough Transform, we will compare and contrast the implementations. More specifically, we will graph the speedup of both implementations running with 1, 2, 4, …, 32 processors. We will compare how the speedup scales with both implementations as the number of processors increases. We will also try varying image sizes with each number of processors so we can see how problem size affects the speedup. For our demo, we will show the speedup graphs for the different implementations with increasing number of processors and varying image sizes.

Stretch Goals

If we have time, we hope to also implement an OpenMPI version of the Hough Transform. We still need to decide the architecture of this implementation and if it is better to have a root thread assigning work or if all threads should do work. Messages will have to be sent to write to the H accumulator array and read from the image array since they cannot be accessed globally. If we can finish this implementation, we also want to graph the speedup with increasing number of processors as well as varying image sizes. We can then compare this implementation to the other 2 parallel implementations and the serial one. We can add these speedup graphs to our demo.

Platform Choice

Language

We are choosing to do our project in C++. This is because C++ provides in-built data structures such as queues that we can use for a potential OpenMP / OpenMPI solution. It is also compatible with AVX intrinsics. Both members of our group are very comfortable with this language and we feel that we can be the most efficient in C++.

Computer

We plan to use the Bridges machine to run each of our parallel implementations. The high number of cores and MPI cores will allow us to fully test the extent of parallelism available in each of the implementations we plan on doing.

Schedule

Week	**Goal**
11/02/2020 – 11/08/2020	Set up a machine and benchmark OpenCV Hough Transform so that we have a baseline to work from.
11/09/2020 – 11/16/2020	Implement SIMD version of Hough Transform

11/16/2020 – 11/22/2020	Implement OpenMP version of Hough Transform Work on checkpoint report
11/23/2020 – 11/29/2020	Benchmark SIMD and OpenMP implementation. Compare and Contrast the Speedup.
11/30/2020 – 12/06/2020	Implement OpenMPI version of Hough Transform and benchmark it. Compare with SIMD and OpenMP implementation.
11/07/2020 – 12/14/2020	Work on poster presentation and final writeup