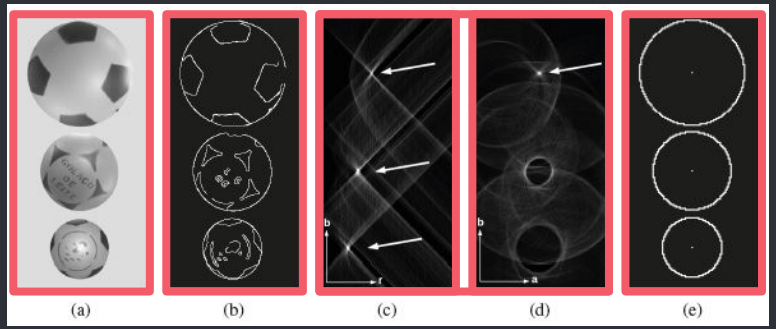
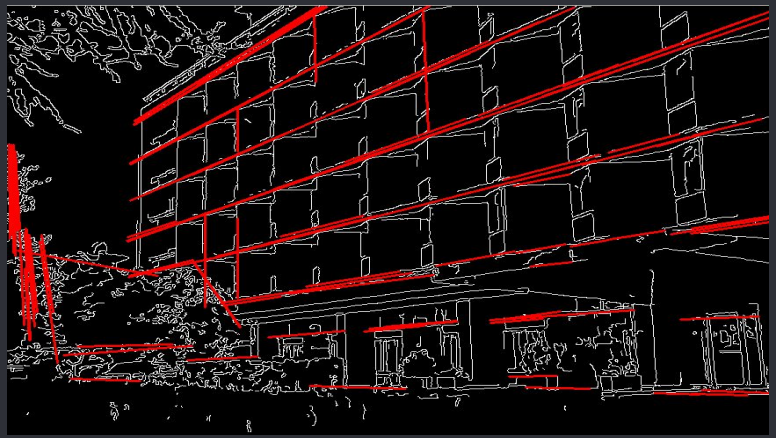
Optimizing Hough Transforms

Rashmi Anil and Ishaan Gupta
15418 - Fall 2020

Introduction

- Hough Transform is a feature extraction technique used in computer vision
  - Commonly used for detecting shapes in the input image
- Usual Pipeline (image on right):
  a. Smoothing the image
  b. Edge Detection and Thresholding
  c. Voting in Hough Space
  d. finding local maximums & identifying lines
  e. Displaying the detected lines
- Steps a and b have variety of algorithms that use highly optimized convolutions so we are focusing on steps **c & d**





Algorithm Description

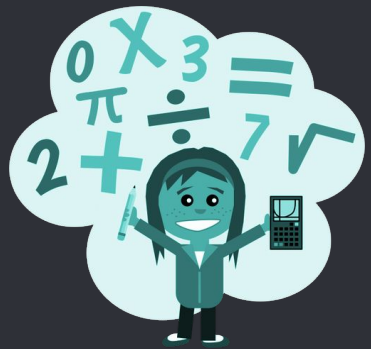
Core Computation:

```
Initialize accumulator H to all zeros
For each point (x,y) in the image
    If (x,y) is edge point
        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
```

Independent Operations

Opportunities for FMA

Lots of Memory Accesses



- Height of H = 180
- Width of H = 2 * (w + h)
- Precompute sin(θ) and cos(θ) for each value of θ

Project Summary:

- The goal of our project is to make this core computation as fast as possible on a multicore CPU using SIMD and OpenMP
- Benchmarked against *HoughLines()* from OpenCV Library

Approach - SIMD

Version 1 - Basic

- Parallelize over theta values in inner most loop
- 8 wide vectors - 8 theta values at once
- Increment to accumulator still sequential
  - Gather and Scatter operations too slow

Version 2 - Advanced

- Parallelized for the hardware
- Hardware Specifications x86_64 - Haswell:
  - 16 SIMD Registers
  - 2 Fused Multiply Add Units (Latency 5)
- Need at least 10 independent FMA operations pipelined to fully utilize hardware
- Parallelize over theta values in the inner most loop
  - Unrolled the loop to try to meet independent operation requirement
    - We were only able to pipeline 4 FMA's (32 thetas at once)
  - Keep register usage under 16 at all all times
- 8 wide vectors

Unrolling the loops and pipelining decreased execution time

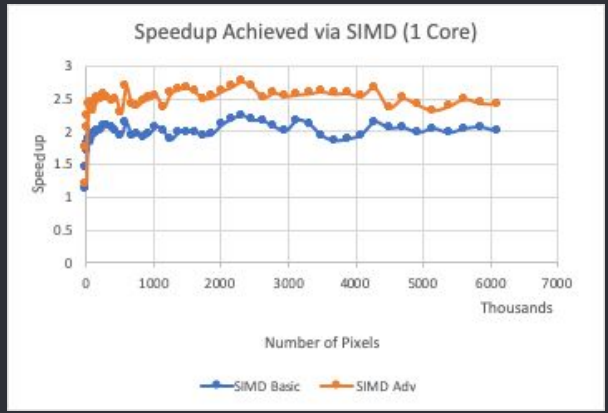
Results - SIMD

Results:

- As expected, execution time for SIMD implementation scales linearly as the image size increases
  - Scales at a slower rate than the fast serial implementation
- Speedup achieved for Basic SIMD implementation was around 2x
- Speedup achieved for Advanced SIMD implementation was around 2.5x

Analysis:

- Speedup MUCH lower than expected.
  - Expected speedup was around 8x
- Time Breakdown:
  - Time spent doing core computation: 95%
    - Finding rho value and computing index into the accumulator: 2%
    - Sequential memory update: 98%
  - Time spent outside core computation: 5%
- Due to that fact that we are spending so much time sequentially updating memory, no matter how fast we make the the parallel computation, our speedups are going to be poor (Amdahl's Law)



Approach - OpenMP

1. Main idea: Parallelize outer loop iteration across pixels using multiple cores.
2. Used Dynamic scheduling since work was uneven depending on which pixels were edge pixels
3. Incrementing Accumulator each iteration required locks:
   - Tried #pragma omp atomic
     - Slower than serial version because of contention
   - Switched to each thread having local copy of accumulator
     - Local copy of accumulator must be added to global copy, making work scale with # of processors
4. Also parallelized over loop for finding local maximums in accumulator
   - Were initially pushing maximums onto a vector in Serial code
     - Switched to atomicly incrementing index of an array to avoid having to use critical section

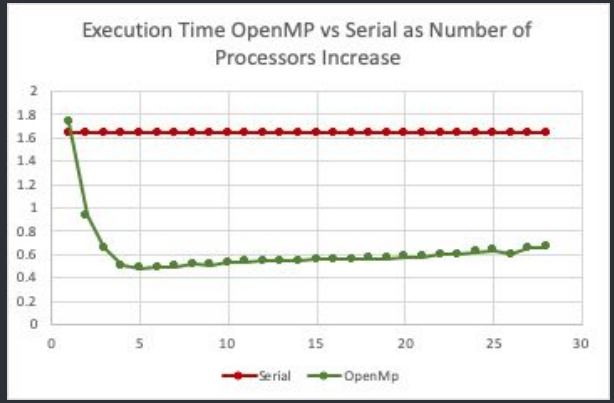
Results - OpenMP

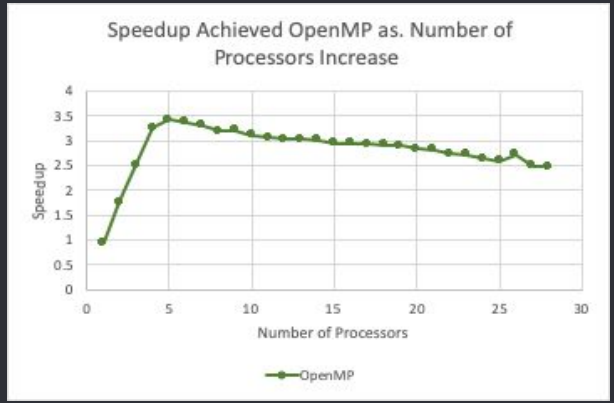
Results:

- Significant improvement on execution time over serial implementation
- Speed up scaled nearly linearly until 4 processors, but then leveled off
  - Does not scale well

Analysis:

- No more speedup after 5 cores:
  - 86% of execution time was in update to local accumulator
  - Memory bandwidth limited
- Slight decrease in Speedup due to step where each thread needs to add local accumulator matrix to global accumulator
  - Work increases as number of processors increases
- Max speedup of about 3.5x





Results SIMD + OpenMP

Analysis:

- Replaced inner most loop of OpenMP implementation with SIMD implementation
- Reached maximum speed up of about 8x, about 3.5x from OpenMP and 2.5x from SIMD

Future Work:

- Severely memory bandwidth limited
  - Try and reduce the number of memory accesses
- Focus on memory issues that inhibited the speedup with SIMD and OpenMP
  - Minimize number of cache misses when accessing accumulator matrix
  - Use OpenMPI to communicate between machines