CS 217 Final Project
Point in Polygon

Daniel Vyenielo
`dvyen001@ucr.edu`
`https://github.com/DVNLO`

December 10, 2021

Abstract

For this project I implement solutions to the point in polygon problem. Each solution was designed to run on cpu xor gpu due to limitations with the `nvc++` compiler's support for the cpp parallel algorithms library. Runtime performance for both implementations was recorded (unfortunately on different systems; discussed later) using a randomly generated collection of 10^9 points over 32 trials. Each point was queried using a point in polygon implementation to determine if the point existed within a unit square situated in an xy-plane at $\{(0,0),(0,1),(1,1),(1,0)\}$. Finally the determination of each points location, inside or outside, of the polygon was verified sequentially. No inconsistent points were discovered across all trials and implementations.

1 Technical Description

This project contains two implementations. One implementation target a cpu runtime environment and the other targets a gpu runtime environment. Each implementation shares the same general algorithm. First, I will discusss the general algorithm. Then follow up with implementation specific details.

1.1 The Point in Polygon Algorithm

Multiple algorithms are known to determine if a point exists within a polygon. This project focuses on a ray casting algorithm.

The ray casting algorithm works be projecting an infinite ray from a query point (p_q). For each edge of a polygon the ray intersects a count is incremented. Initially, making the assumption that the point exists within the polygon, each intersection of an edge will leave the interior of the pollygon, then enter the polygon, leave, then enter, etc. After considering each edge, and determining if the ray intersects, a determination of the points location, inside or outside, of the polygon is possible. A point inside a polygon will have a odd intersection count, while a point outside a polygon will have an even intersection count. This prescription is easier said than implemented.

The crux of the algorithm relies on determining the intersection point of two lines and determining if the intersection point lies within the polygon edge's line segment. Mathematically we can model the algorithm as a system of two parametric lines, one line eminating from the query point called l_{pq} and another forming the edge line segment connecting two points from the polygon's vertices p_i and p_j. We will call the edge line segment l_{ji}. Constructing the system we can enumerate each parametric line by introducing the parametric variable t for each line, yielding the following system.

$$\begin{aligned}\vec{l_{pq}} &= \begin{bmatrix}1\\0\end{bmatrix} t_0 + \begin{bmatrix}p_{qx}\\p_{qy}\end{bmatrix}\\ \vec{l_{ji}} &= \begin{bmatrix}p_{ix}-p_{jx}\\p_{iy}-p_{jy}\end{bmatrix} t_1 + \begin{bmatrix}p_{jx}\\p_{jy}\end{bmatrix}\end{aligned}$$

The intersection occurs when $\vec{l_{pq}}(t_0) = \vec{l_{ji}}(t_1)$, therefore rearranging,

$$\begin{aligned}\begin{bmatrix}1\\0\end{bmatrix} t_0 + \begin{bmatrix}p_{qx}\\p_{qy}\end{bmatrix} &= \begin{bmatrix}p_{ix}-p_{jx}\\p_{iy}-p_{jy}\end{bmatrix} t_1 + \begin{bmatrix}p_{jx}\\p_{jy}\end{bmatrix}\\ \begin{bmatrix}1\\0\end{bmatrix} t_0 - \begin{bmatrix}p_{ix}-p_{jx}\\p_{iy}-p_{jy}\end{bmatrix} t_1 &= \begin{bmatrix}p_{jx}\\p_{jy}\end{bmatrix} - \begin{bmatrix}p_{qx}\\p_{qy}\end{bmatrix}\\ \begin{bmatrix}1 & p_{jx}-p_{ix}\\0 & p_{jy}-p_{iy}\end{bmatrix}\begin{bmatrix}t_0\\t_1\end{bmatrix} &= \begin{bmatrix}p_{jx}-p_{qx}\\p_{jy}-p_{qy}\end{bmatrix}\end{aligned}$$

Which is a deceptively simple system to work with. Naievely, trying to compute t_1, we find

$$t_1 = \frac{p_{jy}-p_{qy}}{p_{jy}-p_{iy}} \tag{1}$$

Substuting t_1 into $\vec{l_{ji}}(t_1)$ we can compute the x-coordinate of intersection between the infinite ray and the edge line segment. Unfortunately, the equation for t_1 suffers from the possibility of division by zero when $p_{jy} = p_{iy}$. These inconsistent situations proved to be quite complicated to resolve, and lead to many edge cases which I couldn't quite get right in my implementation (I tried for about a day) even for what seemed to be a simple unit square.

Conceeding defeat I looked up a solution which projected an infinite ray vertically, rather than horizontally, as I originally derived, and avoided division by zero through a nifty trick which shortcutted evaluation if division by zero would have occured (PNPOLY). Using the PNPOLY solution the remainder of the implementation focused on parallelization.

For both CPU and GPU the parallel code focuses on splitting the determination of a points location at the granulaity of a point, since each determination can be made independently against a constant polygon. However, this independence, while obvious, proved to be challenging to implement with `nvc++` due to aparent differences between compilers support of the parallel algorithms library.

1.2 Parallelizing on CPU

Implementation on CPU was fairly trivial thanks to the parallel algorithms library available in C++17. The implementation leverages use of a parallel execution policy in conjunction with `std::transform()`. These features are well specificed and described on cppreference.

However, during compilation it became apparent that `nvc++` and `g++` offer varying levels of support of the parallel algorithms library. For example, the same code which compiled with `g++-10` on my local system would not compile with `nvc++` version 21.9 on bender. I discuss the issue later.

Inspite of these shortcomings, I was able to implement a solution using `g++-10` on my local system. `nvc++` does not appear to parallel lambda's in the same way as `g++-10`.

1.3 Parallelizing on GPU

Parallelization on GPU was surprisingly similar to the Histogram kernel. My implementation leverages shared memory in much the same way histogram did. Each block is responsible for loading polygon vertices into shared memory. With the polygon loaded, each thread begins to determine if each point from the input set is within the polygon. Each thread strides through the input points in the same pattern as histogram, but since the point data each thread uses is exclusive to that thread there is no need to synchronize accesses to the points or to the output.

1.4 Branchless Code

One additional optimization focused on minimizing branching. For both implementations the `is_point_in_polygon()` attempts to not branch. However, there is shortcutting when evaluating the intersection condition. Which is ultimately implemented as branching. So despite by best efforts, there is branching in disguise as logical operators in my code. The shortcircuiting proved to be necessary to stop division by zero.

2 Project Status

Is it feature complete? Does everything work? What does not work? What are limitations of your project? (e.g. only works on square matrix) What major technical challenges did you encounter?

3 Results

Timing of parallel code vs serial code. Timing of various implementations. Bottleneck analysis using profiling. Screenshots of your project running.

4 Documentation

Documentation and outline of how to compile and run your project. Description of expected results when running your project. Which input should be used for testing?