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 10^8 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.

$$\vec{l_{pq}} = \begin{bmatrix}1\\0\end{bmatrix} t_0 + \begin{bmatrix}p_{qx}\\p_{qy}\end{bmatrix}$$

$$\vec{l_{ji}} = \begin{bmatrix}p_{ix} - p_{jx}\\p_{iy} - p_{jy}\end{bmatrix} t_1 + \begin{bmatrix}p_{jx}\\p_{jy}\end{bmatrix}$$

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

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

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

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

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 support 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 to prevent division by zero. Ultimately the shortcutting is 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. Note I am only assuming this since I do not know how to look at the GPU assembly code, unfortunately, without spending more time on the topic which I do not have.

2 Project Status

This project is feature complete, but significant issues were encountered during implementation. Multiple problems developed including issues with differences in the parallel algorithms implementation by the `nvc++` and `g++` compilers, lack of a `g++` compiler on bender which supports parallel algorithms library, challenges translating the point in polygon algorithm from math to code, and issues working with the standard template types and cuda simultaneousily. I Discuss each issue below.

By far the most severe issues I encourntered where with compilation. `nvc++` and `g++` do not offer the same implementation guarentees when working with the c++ parallel algorithms library. The first issue is a difference in the way `std::transform()` handles a unary functor. In `g++` there is full support for passing a lambda as the unary function argument to `std::transform()`. However, `nvc++` does not offer the same guarentees with reference captures in the lambda capture field. According to documentation reference captures are supported but with a very obvious warning for race conditions and possible memory issues if compilation targets the gpu(documentation). Nevertheless, when compiling with `nvc++` there is an option to target a cpu with the paramater `-stdpar=multicore`, which should allow the lamba capture to occur my reference the same as g++ offers it, but despite setting this flag `nvc++` will not compile the reference capture.

Another issue I encountered occured with an apparent race condition when using a `std::vector<bool>`. In `g++`, I suspect there is a race when using `std::transform()` and collecting results of a unary functor lambda into the `std::vector<bool>`. I was able to document cases (see `edge_cases()` in main_cpu.cpp) when running on the large point sets where points within the unit square were said to be outside the unit square. Sequential execution for the same points was always correct. Moreover, changing to from bool to uint_fast8_t and re-executing with a parallel execution policy did not cause inconsistent results. I am not sure what the true cause of this is and can only speculate.

The next issue was my own. I could not, despite significant effort, get the point in polygon edge cases correct. This effort was difficult because of the division by zero issue when computing the t_1 parameter, mentioned above. My initial thought was to rationalize the denominator, and then scale the x-axis comparison by the denominator to remove the division by zero. However, this leads to issues when the denominator is negative or zero, since the inqualities break down. Ultamately, I had to give up on this issue to move forward with the project. I may return to this issue since ratioinalizing the denominator will yield truely branchless code since no shortcutting will be necessary.

Lastly, one minor annoyance with this project was the cuda memory operations library. I could not determine how to transfer c++ stl types between host and device. The functions I found all focused on transfering bytes, which is greate for simple types like `std::vector<int>`, but not so great for composite types where the memory layout maybe a bit more complex. In my implementation I originally chose to use a `std::vector<std::pair<float, float>>` which I was unsure if accessing the underlying data of the vector would be ordered pair.first, pair.second, pair.first, pair.second, etc. Because of this limitation it necessitated changing the function interfaces into c-style interfaces with pointers, which works, but is difficult to maintain in a c++ application, not to mention it goes against modern cpp programming practices.

Nevertheless, stepping down from my software engineering soap box, the implementations are feature complete, but as evidenced above are not without issue and getting the systems interactions correct proved to be the most challenging part. Ultimately, I was unable to reconcile the differences within the timeframe and had to take measurements on two different systems.

3 Results

The results of my work are fairly weak due to complications during implementation. As a consequence of the issues I was forced to take measurements of parallel code on seperate systems. The CPU implementation ran on my local computer (2 x Intel Xeon X5650 6 Cores 2.67GHz, 24 logical cores total, 24GB RAM) and the GPU implementation ran on bender. Both ran with the same number of points for the same number of trials, but unfortuntaely the implementations are different, due to the different functionality of the standard library and cuda. So despite comparing apples and oranges I will try to do my best reviewing the results.

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?