Comparing the Performance of the Jarvis March Algorithm on Various GPU Frameworks

Ivan Castaneda

Overview

The original scope of this project was to implement and analyze algorithms related to three well-known computational geometry problems using modern C++ and CUDA. These problems include finding a convex hull, performing Delaunay triangulation, and constructing a Voronoi Diagram from a given set of points. Due to a lack of time, I narrowed the scope of the project to implementing and analyzing the performance of finding a convex hull.

To accomplish this, I experimented with multiple convex hull algorithms, such as Graham's scan and Quickhull. Ultimately, however, I opted to use the Jarvis March algorithm for ease of implementation and to ensure the completion of the project.

The project includes three implementations of the Jarvis March algorithm, namely a standard, sequential C++ implementation, a modern, parallel C++ implementation, and a CUDA implementation. In addition, I used python to help generate random points and as a visualizer using matplotlib library.

How is the GPU used to accelerate the application?

Description of the algorithm

The Jarvis March algorithm, also known as the Gift wrapping algorithm, finds the convex hull of a set of two-dimensional points. A convex hull is the smallest polygon that contains an entire set of two-dimensional points. The algorithm first starts by discovering the farthest left point, then iteratively selects the point with the smallest counterclockwise angle from the current point. This process is repeated until the algorithm returns to the starting point. The time complexity of this algorithm is O(nh) where ‘n’ is the size of the input and ‘h’ is the size of the hull. Each solution point must iterate through the entire set of points before iterating to the next solution point.

Details related to the parallel algorithm

Due to the rigid sequential nature of the algorithm (e.g. we do not know the size of the hull until the algorithm is completed), this algorithm is not the optimal candidate for parallelization. However, there are opportunities to enhance its performance.

The algorithm benefits from parallel preprocessing steps. The algorithm must find the farthest left point before it can begin the core algorithm, where in the worst case it can take O(n) time. Rather than sequentially checking each point and updating the farthest seen point, parallelization allows multiple threads to perform checks simultaneously. Minimum-finding reduction can be applied to this preprocessing step.

How is the problem space partitioned?

In this kernel implementation, I partitioned the problem space using CUDA Grid-Stride pattern.

Implementation details

My implementation of the algorithm in CUDA includes using the GPU to launch a 'findLeftMost' kernel which retrieves the leftmost point, or the smallest X value, with a tiebreaker of the smaller Y value.

The kernel uses shared memory for local indices, local X values, and local Y values to reduce memory overhead. Threads populate these shared memory arrays, synchronize, and begin minimum reduction. The reduction checks for smaller X values and breaks ties using Y values if X's are equal. If a value is found, replace the current point with the point at the current stride and update the index. Compare blocks to find which contains the minimum-most index, using atomic operations to prevent race conditions.

A similar, yet more intuitive method is utilized in the case for Modern C++. We can assign integer variable 'l' an index using std::distance() passing in the beginning of the points vector, and most left elements in the vector using std::min_element() utilizing execution policy std::execution::par. We use a custom comparator using C++17 lambda expression by defining that the boolean should return 'a.x < b.x' finding the minimum x value given two points.

Documentation

My project directory contains three directories label cuda, moderncpp, and naive. Each directory contains another directory label convex-hull. All directories contain a Makefile script containing everything needed to build their respective Jarvis March algorithm.

This binary file requires an argument of an input.txt files containing the number of points at the top of the file and the points in the file itself. This can be generated by

‘python3 input.py <num_points> <min_val> <max_val>’

Which specifies an input.txt file that contains the number of points to be generated, the minimum value a point can have, and the maximum value a point can have. This will be written to an input.txt file.

Running the binary using this text file will generate an output.txt file and the time it took to generate that file.

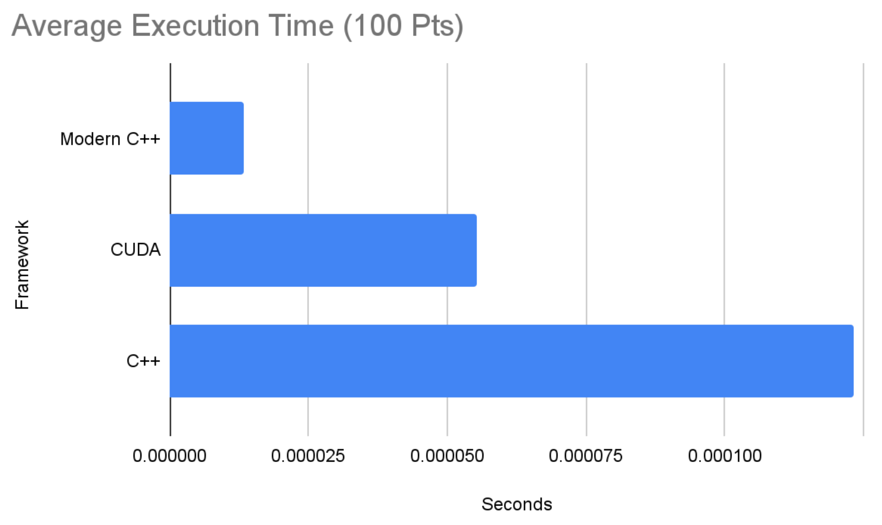
‘python3 plot.py’

Generates a plot.png which takes the input values from input.txt and the hull values from output.txt and uses matplotlib to plot them on a graph.

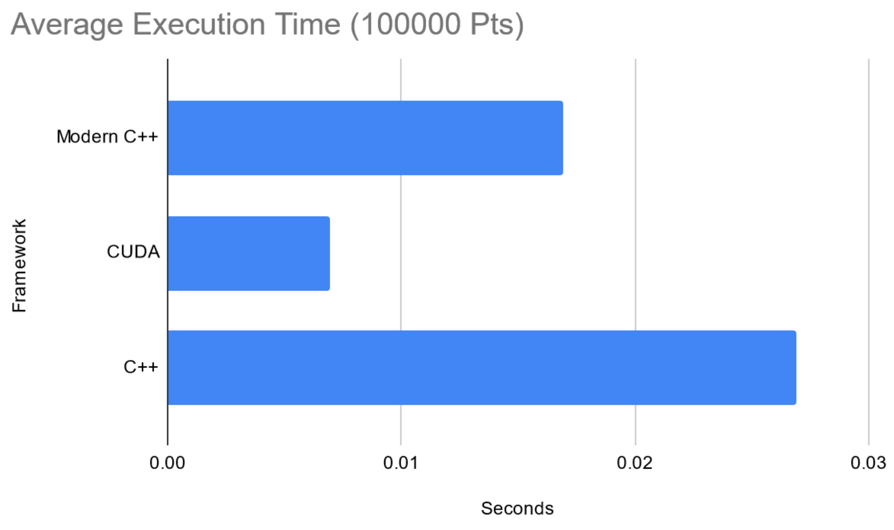
‘python3 run.py <num_points> <min_val> <max_val> [plot]’

This script automates this entire process, given that the binary file has been generated, where an input.txt and output.txt will be created and an optional plot can be created where plot is specified at the end or not.

Evaluation



Average Execution Time of 100 Points Across 10 Runs.



Average Execution Time of 100000 Points Across 10 Runs.

It is evident that C++ is outclassed by CUDA and Modern C++ in both small and large datasets. This is expected however, as it performs all instructions sequentially. Modern C++ outclasses CUDA in small datasets, while CUDA performs best in large datasets.

The Modern C++ framework requires, overall, less management from the user to utilize effectively. The compiler, and the standard libraries, handle both memory management and parallel algorithm implementation. The programmer has the flexibility to decide when, where, and why they want to utilize execution policies within their code. However, this convenience comes at the cost of potentially losing out on memory optimizations, and only being able to utilize algorithms that support execution policies. To this end, larger datasets may execute slower than other GPU frameworks. In this project, Modern C++ had almost a 91% performance improvement over its C++ sequential counterpart for the small datasets.

The CUDA framework requires significantly more involvement than its Modern C++ counterpart. CUDA gives the programmer everything they need to assemble their own kernel instructions, including memory management and synchronization timings. While implementing a kernel is not as trivial as calling an execution policy in Modern C++, CUDA allows for better optimization opportunities as it scales with a programmer's knowledge and mastery. CUDA is more suited for extreme-performance and larger datasets. In this project, CUDA had almost a 76% performance improvement over its C++ sequential counterpart for the large datasets.

Problems faced

I faced several problems throughout the duration of this project. Prior to this project, I had no experience on implementing computational geometry algorithms. I underestimated the time needed to research and even implement a sequential version of an algorithm. Most of my time was spent researching implementation methods for these algorithms, limiting my time engineering parallel solutions.

Another problem I faced was compiling with modern C++ source code using the NVIDIA HPC SDK and nvc++ with -stdpar. Although I was able to leverage the new features provided by C++17, such as execution policies, it was evident that these features were primarily designed for experienced, senior developers. I suffered numerous verbose compiler errors that hindered my progress, which resulted in multiple rewrites of the same algorithm in order to resolve these issues.

A technical-related problem I faced was attempting to implement a parallel Quickhull algorithm. As a new student to GPU programming, I struggled to convert the recursive nature of the algorithm to something that can utilize threads effectively. As time was a crucial resource at this point of the project, I decided to scrap the design all-together and reworked it using Jarvis March Algorithm. In retrospect, it would have been more efficient to focus on parallelizing specific aspects of the algorithm, rather than committing to a complete device-only algorithm.

Video Link

https://youtu.be/RgKWciodbas