Parallelized Localization in an Octree Representation of Point Clouds

John Lore (jlore)
Richard Zhu (rzhu1)

SUMMARY

We implemented localization of iPhones in a space by collecting a point cloud from each iPhone, storing them in octrees, and matching together the objects represented by each point cloud using a parallelized template matching based position and direction agnostic algorithm. Our parallelized localization algorithm achieved 1.24x speedup over serial. We ran our implementation on Intel Core i7.

MOTIVATION

Originally, our goal for this project was to develop a cryptocurrency which was aimed at solving two of the biggest problems with cryptocurrencies today: the scalability problem and the mining integrity problem. Bitcoin, the largest cryptocurrency by market capitalization, can process only about 3-4 transactions per second. This means that a simple transfer of money from one party to another can take hours. For comparison, Paypal can process up to 500 transactions per second, and VISA is able to handle almost 2000 transactions per second. This, along with the fact that each transaction must be validated via miners, which costs an increasing amount of electrical energy, renders bitcoin unusable for day-to-day transactions. The mining integrity problem has to do with consolidation of mining power in a cryptocurrency. If a single entity is able to comprise over 50% of all mining power, then this entity would theoretically be able to out-mine everyone else combined, thus progressing along the blockchain faster, and invalidating their transactions; their currency lost and worthless. The initial goal of this project was to use what we learned in this class to produce a cryptocurrency which was massively scalable, yet largely resistant to mining consolidation.

However, this quickly proved to be outside the scope of this class, and beyond our capabilities as programmers. Instead, we swerved to a similar idea. We are on the cusp of an augmented reality breakthrough. Hugely popular games such as Pokemon Go and Mario Run demonstrate this. Along with this technology will come a demand for accurate mappings of the real world around us. Thus, we have developed a three-part system which contains an app, a server, and an algorithm. The user uses the app to scan the real world into a point cloud, which is uploaded onto the server, which contains an aggregate point cloud containing all the points previously scanned by all users. The algorithm then determines where in the point cloud the new points should go, and inserts the points into their correct position.

BACKGROUND

A point cloud [Figure 1] is a dataset consisting of points in a 3D space relative to some origin. Multiple scans of the same physical space will produce different point clouds that are similar in the distance between notably large structures.

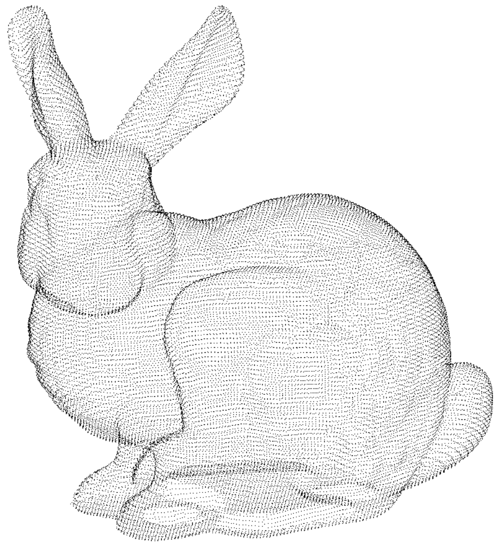


Figure 1: a point cloud for a rabbit statue

An octree is a very efficient method of storing sparse point clouds. It is a tree data structure where each node has up to eight children. The tree rooted at each node contains the information for all the points contained within a cube, which is represented by that node.

We build an octree [Figure 2] with a fixed minimum granularity as a representation of the 3D data which is our point cloud. Key operations on octrees used in our project include the union of two octrees, adding nodes and updating intensity values held in pre-existing nodes, and a function which returns all the points within a certain radius of a given point. During the search and match step, the aggregate octree is read-only, and is shared across threads.

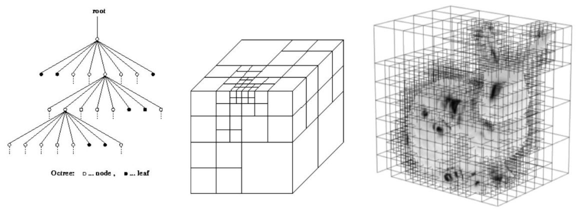


Figure 2: the octree data structure and visualization in 3D space

Our algorithm accepts as inputs an octree which is the representation of the template point cloud, another octree which is the target point cloud,, and a thread count. It outputs either a point which is the xyz coordinate within the target point cloud that the template point cloud matches, effectively localizing and matching the template point cloud to its correct position within the target point cloud, or None if there is no point with a sufficiently high metric indicating a match.

Our localization algorithm is a parallelized variation on template matching [Figure 3] that we developed for 3D space. Template matching is a computational photography process in which a template, or part of an image, is searched for within an image by sliding the template along the image, convolving it with the pixel values, generating a metric and thus calculating a value of certainty to a match for the template in that location in the image. This algorithm requires that the template and the image are oriented and scaled the same. Thus, to find a match for a template of varying size or orientation, many runs across the image must take place. This process is called multiscaling, and the best match for a given scale is taken to be the best match to a template in the image. Our algorithm effectively performs this in 3D space instead of 2D.

At this point, one might be wondering why GPS could not be used to pinpoint the exact location of the user. This is because, according to the United States government, the standard for civilian GPS has a 95% confidence interval accuracy of 7.8 meters. This is not nearly accurate enough for the precise scan of the world that we wish to develop using our aggregate point cloud view.

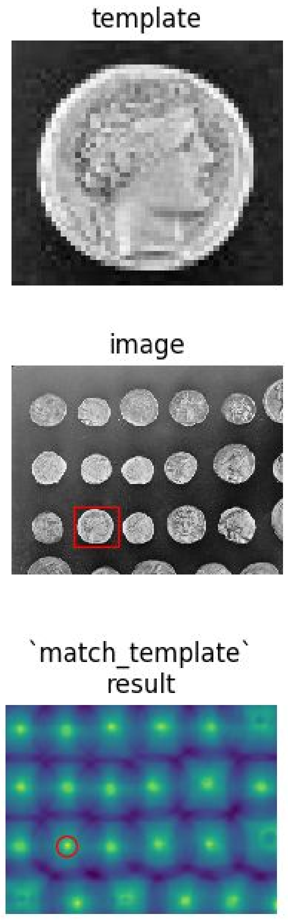


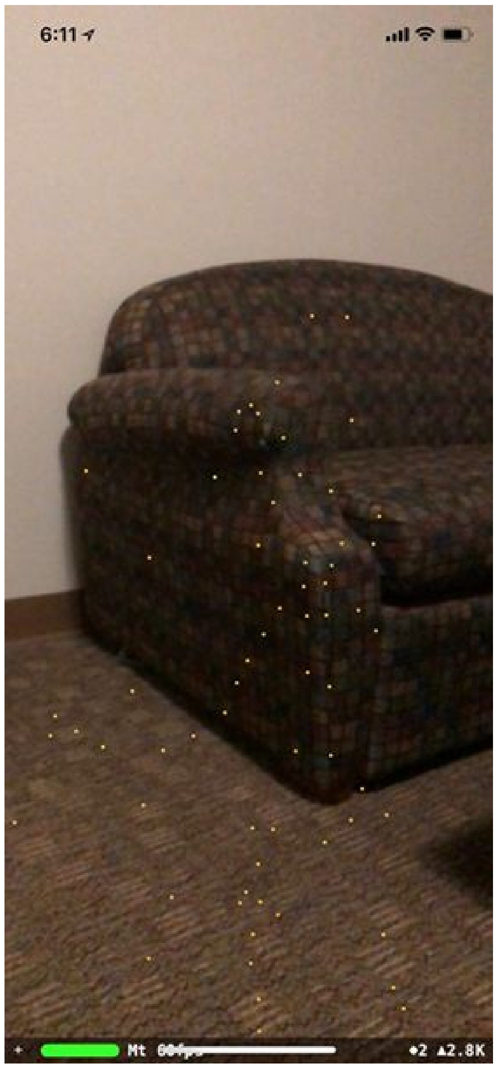
Figure 3: example of template matching for a 2D image

This multiscaling process is both very expensive and highly parallelizable. Our 3D implementation utilizes a variation we describe as multirotating. Different scales of the same template can be performed at the same time in parallel. Furthermore, sliding the same template across the point cloud looking for matches is highly parallel, as the sliding can be done at the same time in different parts. Due to the nature of points in space being naturally near to each other, the locality should be very high.

Parallelism of this algorithm lies in the choice of parallelizing the multirotating step, the sliding of the template, or the simultaneous search for different templates within the space.

APPROACH

We have a large amount of overhead involved in the retrieval and storage of the point clouds. For retrieval of points to store in the point clouds, we used an Apple iPhone X and Apple iPhone 6S running ARKit. ARKit uses the cameras on the phone to detect features, or points, which we use to build our point cloud by logging all the points detected. These points are relative to the position the iPhone is in when the app starts. Additionally, the direction in which the user is facing on app startup is determined as the x-axis.



Using a RESTful API and JSON format, we send our point cloud data to a server on our local PC that saves them. Initially, we were running a PostgreSQL database to save all our points, but ran into issues running that and leveraging an API to store point cloud data as patches. We instead build an octree as we receive points in POST requests. We use GKOctree in ARKit locally on the iPhone in order to create the octree which we initially send to the server. Then, we use the multiprocessing package in Python in order to parallelize our localization algorithm. In Python, we also use a modified open-source implementation of octrees called pyoctree, which was written in C++, but then exposed to Python using Cython. Pyoctree supports OpenMP.

We construct octrees with fixed maximum depth, meaning a limit on granularity of points is set, as we are mapping the physical world and do not need to store data with centimeter accuracy nor have the physical memory space to do so. This effectively performs a gaussian blur on the data as many points may fall into the same bucket in space, but is preferable in our implementation due to the fact that two points clouds don't contain the same points but rather contain different points that characterize the same larger objects. The bottom nodes in our octree represent 1 meter distance.

With standard template matching, the template is slide over a target image in order to determine a metric for each point along the image. Some metrics commonly used are SQDIFF, normalized SQDiff, CCORR, normalized CCORR, CCOEFF, and normalized CCOEFF. Traditionally, the part of template matching which is parallelized is the multiscaling process. Different threads will run in parallel on different scales. However, our algorithm uses a different parallelization. Because we are using an octree implementation, we have devised a way of using the metric for size *n* in order to determine the metric for size *n+1*.

Traditionally, template matching is done on matrices, but we are performing it on octrees. We can leverage the structure of the octrees in order to avoid having to recalculate metrics on different sizes. The metric that we will be using is simply the number of points within a given space. This is simple in order to facilitate easy understanding, but it has the same computational complexity as the other metrics traditionally used in template matching, and is sufficient for our use case. In order to count the number of points within a cube within an octree at the lowest level, we have written a function which returns the list of all the points within a certain radius of a point within the octree. We will call this with the smallest radius of a cube within the octree. This computation only needs to be done once. For size *n+1*, if we have the metric for size *n*, we can just simply sum up the 8 metrics of the 8 children nodes, because the number of points within the large cube will just be the sum of the number of points within each of its children.

This metric also has the added benefit of being direction agnostic. Because we are not caring about the specifics of the points, and we are simply counting them up with respect to a certain cube, this means that for any orientation, we will be able to discover a matching if there is a matching to be found.

Using this strategy means that it is suboptimal to perform parallelism across different scales, as is usually done in template matching. However, because of the nature of octrees, where we have a natural bifurcation of points, we will be using different threads in order to calculate metrics across space. For example, if we are running the algorithm with 2 threads, then we would split the algorithm into 2 half-cubes, and each thread would calculate metrics on each of the half-cubes. Similarly, for 4 threads, we would calculate the metrics for 4 2NxNxN rectangular prisms. When we hit 8 threads, we will be working with 8 NxNxN cubes, which just so happens to be the structure of the octree, so at that point, one thread can take the results of the other 7, and combine them to calculate the metric for the entire thing.

We explored and went down many different routes before arriving at our final implementation. Conventionally, the template matching algorithm is parallelized heavily when the template is multiscaled to search for a good match. Initially we approached the problem in the same way, though instead of multiscaling we performed a sort of multirotating in space. This is due to the fact that when two point clouds are captured, they are captured from different orientation in rotational spaces and positional space, and we seeked to account for the rotational space to localize position.

After coming up with this method, we effectively no longer needed to multirotate and thus were able to parallelize the sliding of our template as we documented.

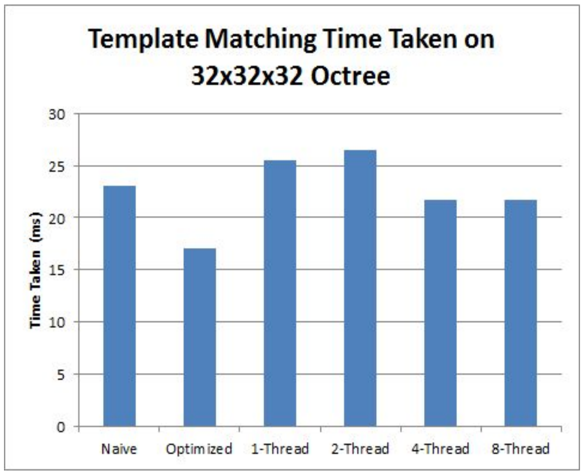
Before doing any of this work for our implementation targeting the i7 CPU, we spent a lot of time trying to parallelize the algorithm on the Apple iPhone iOS SoC. Ultimately, we found the constraints for memory of the iPhone as well as the constraints on the available libraries to be too difficult to use to develop our algorithm.

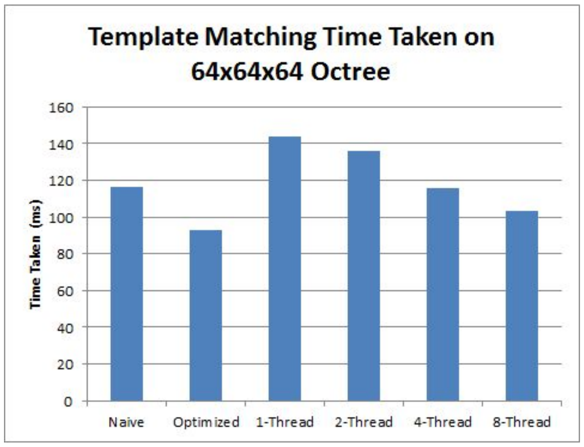
RESULTS

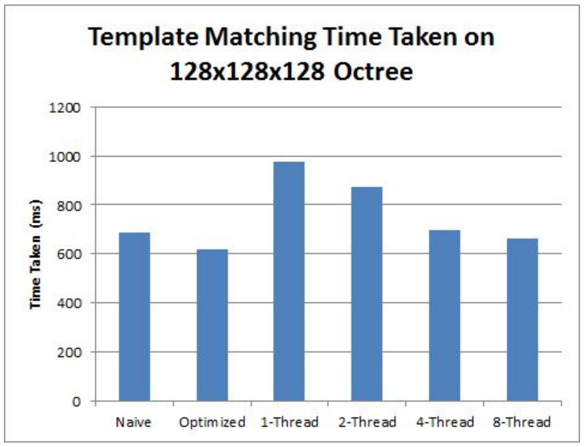
We measured the execution time for the matching process on a naive implementation, which is simply an analog to standard template matching as done in computer vision; a single core implementation of the algorithm which is optimized for octrees; and finally our parallel implementation which we ran on multiple cores (1,2,4,8). We specifically measured the time it takes to find a match after the octrees have been created. We used the execution time to show speedup for various number of threads running.

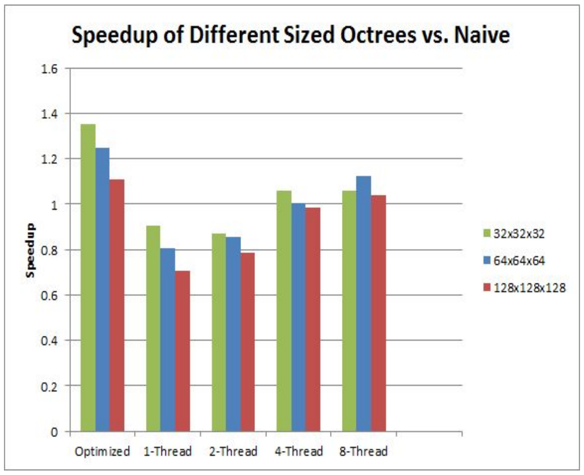
Our experimental setup involved searching for a template of size 8x8x8 within octrees of size 64x64x64 and 128x128x128. Problems of larger size result in taking far too long to generate octrees and execute. The 128x128x128 took 40 seconds to generate, and a 256x256x256 would have taken 40 minutes. This would have been infeasible for testing. Problems of smaller size are not computationally interesting, so we settled on 128x128x128, which is the largest size that was reasonably tractable, and 64x64x64, the next size down, so we could explore how speedup changed as the size of our search space changed.

Here we provide graphs to show our achieved speedup for different thread counts and problem sizes.









As we can see in the results, the time it took to template match on the 128x128x128 was about 6-7x the time it took on the 64x64x64 octree, This is interesting to note, as the size of the search space has been increased by 8. This indicates that our algorithm is scalable to even greater sizes. Furthermore, we had expected a larger increase in time for larger datasets due to the O(nlog(n)) complexity in our algorithm. This makes us speculate that our algorithm performs better on larger data sets relative to the total number of points being iterated on. However, we were unable to measure performance on any octree larger than 128x128x128. A single unit in ARKit approximately correlates to 1 meter in the real world, so a 128x128x128 cube is about 2 million meters cubed of information. Algorithms required to work on large scales such as those found in the real world would need to scale well with respect to the world size.

However, different problem sizes returned very similar speedup with respect to the number of cores used, indicating this solution does not scale very well with more cores. Even after many trials and error, our results were not what we were hoping for in terms of overall speedup. We are able to outperform the naive solution with 4 and 8 cores, but not with 1 or 2 cores.

We considered first poor cache coherency to be the reason why our achieved speedup was non-ideal. However, when we attempted different algorithms and partitioning in order to achieve a better speedup, our performance still didn't see huge returns.

We speculate that due to unexpectedly low arithmetic intensity, our speedup didn't show near linear gains for number of cores. We are doing operations on a huge data structure that ultimately hides the good efforts and smart parallelism we implemented.

This is evident when comparing the speedup of the optimized algorithm to the naive algorithm, as huge gains in speedup were expected at that step since we no longer need to rotate our template, but we showed very small speedup. This leads us to believe that the time spent fetching data from memory contributes the single largest amount of time, since for multirotating the operations are being performed on the same data and thus have great locality. This makes the speedup we achieved from our optimized algorithm very small compared to what we expected, since we thought the iterations on the rotating we most time intensive though in reality it didn't amount to much. as the for the points in our octree low amounts of computation are performed.

Thus, our speedup was limited by low arithmetic intensity, in what is originally a computationally dense problem. However, our use of the octrees made data fetching a larger component, and reduced the arithmetic intensity.

REFERENCES

C. H. Chien and J. K. Aggarwal, "Volume/surface octrees for the representation of three-dimensional objects", ***Computer Vision Graphics Image Proc.*** **36 (1986) 100-113**

C. H. Chien and J. K. Aggarwal, "Reconstruction and matching of 3D objects using quadtrees/octrees", ***Proc. 3rd Workshop on Computer Vision*****, Bellaire, Mich., Oct. 1985 49-54**

H. H. Chen and T. S. Huang, "A survey of the construction and manipulation of octrees", ***Computer Vision Graphics Image Proc.*** **43 (1988) 409-431**

LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT

The app was created with an equal amount of effort by both Richard and John. Neither was very fluent in Swift at the beginning of the project, and much learning had to be done on that front. John was the one who discovered how to use rawFeaturePoints in ARKit in order to extract a point cloud, and Richard was the one who discovered GKOctree, which enables us to save the point cloud as an octree. John solely worked on the server, which is used to store the global octree containing the aggregate points from all users. Richard was the one who developed the algorithm for parallel position and direction agnostic template matching on octrees. The total credit distribution to be fair to both members, should be 50% - 50%, as neither would have been able to complete this project without the other. Whenever one needed help, the other was there.