Unikernel Based Implementation of a Cloud Powered Mandelbrot Viewer

Tiger Mou
School of Engineering
Vanderbilt University
Nashville, TN USA
tiger.h.mou@vanderbilt.edu

Jiayao Wang
School of Engineering
Vanderbilt University
Nashville, TN USA
jiayao.wang@vanderbilt.edu

Yufei Yan
School of Engineering
Vanderbilt University
Nashville, TN USA
yufei.yan@vanderbilt.edu

***Abstract*—Our team's final project is building a cloud-based Mandelbrot viewer using unikernels. It is a program that displays the Mandelbrot set using a parallel computation backend running in the cloud. The front end of the application is written in Fyne.io in Go with the back end written in IncludeOS. The application is deployed on Google Cloud. Latency and bandwidth were likely the performance limiting factors when comparing to local computation.**

I. Introduction

In the Operating Systems class at Vanderbilt, there was an assignment in which students visualized the Mandelbrot set. The set of numbers is infinitely recursive, making the calculation computationally expensive. However, since the calculation of every pixel of the output image is independent of one another, the computation can be easily parallelized. The assignment from the Operating Systems class was to be completed using multithreading. However, for the Cloud Computing final project, we decided to completely rewrite the project by using cloud computing to perform the parallel computation. We think it will be interesting to compare the two different techniques and see the advantages and disadvantages of each approach.

Since the CPU usage in the cloud can rapidly increase from the load of new clients, each with varying client window sizes, we thought that unikernels could be a powerful solution to high speed scalability and efficient CPU scheduling. Since unikernels are tiny disk images of around 5 MB with only the minimal feature set, they are very quick to boot from, making them easy to rapidly scale in under 10 seconds. Since they also don't have extras such as package managers running in the background, CPU cycles are dedicated towards the unikernel application. Unikernels could also comfortably run on cheaper hardware with very little memory (<10 MB) and disk space (<10MB) for each CPU core. This could bring new life to older or less capable hardware.

Furthermore, the task for Mandelbrot viewer is a good candidate for testing parallel computing and scalability. A unique computation is run on each pixel of the image, and each pixel is independent of the other pixels. This independence makes it easy to break up the computation to different Unikernels.

II. Technology

A. *Unikernels*

The core part of our project is the use of Unikernels. Unikernels are specialised, machine images made from library operating systems, which also shrinks the attack surface and resource footprint of the resulting virtual machine. Unikernels also have many benefits compared to a traditional operating systems such as improved security, smaller footprints, more code optimizations, and faster boot times.

B. *IncludeOS*

IncludeOS is a minimal unikernel operating system for C++ services running in the cloud and other hardware. When we want to use it in our project, we can start a program with #include <os>. This will include a tiny operating system into the service during link-time.

We choose IncludeOS instead of other available tools because we were the most comfortable working with C++. Moreover, it is also the most one of the most actively maintained unikernels. Originally, we wanted to use Unik, which is a higher level tool that helps run and deploy unikernels of different types and languages. However, we were unable to run Unik, so we switched to purely IncludeOS.

C. *Google Cloud*

The cloud service we used is Google Cloud. Google Cloud also provides an auto scaling service, and since the simplest way to monitor resource usage is through the VM container, it was the best way to quickly implement auto scaling unikernels. The unikernels could then run in an auto scaling instance group within a private subnet. One downside however, is that it takes longer to boot the unikernel on Google Cloud than within an operating system, likely because Google Cloud needs to set up all the networking and other resources, which introduces some latency to the boot. Additionally, all the virtual machines on Google Cloud are too big for the resource needs of unikernels. so we used the smallest virtual machines with 614 MB RAM and 1GB HDD.

D. *Front End*

For the front-end part of our application, we used fyne.io libraries to build a cross-platform GUI in Go. Fyne.io uses OpenGL underneath.

III. Architecture

Our application consists of three parts: a local application, a web server, and the unikernels, as shown in *Figure 1*.

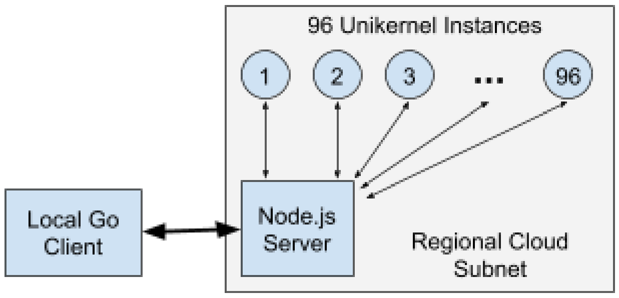

Figure 1. Diagram of system architecture.

A. *Local Go Application*

Our local client is written in Go, using fyne-io and OpenGL to build a cross-platform UI for the local app. The app runs a canvas that displays the Mandelbrot set as a color image. The application processes mouse interactions to examine what parts of the image need to be recalculated and redrawn. Accordingly, it sends protobuf formatted requests to the Node.js server. Each protobuf message contains parameters for the pixel boundaries, coordinate boundaries, and number of iterations. The protobuf messages are sent as the body of a POST request to the node server at port 80.

After server-side calculations are done, our local Go client receives a response from the Node.js server. The response contains an array of the number of iterations at each pixel. The number of iterations can then be translated into colors at the corresponding pixels. The Go client processes the entire array in this way and draws the updated image on the canvas. Since each mouse move can create up to three rectangles that need to be redrawn, the client app uses an asynchronous function for sending requests and processing responses, thus improving efficiency.

Some potential areas for improvement are that the conversion from iterations to color can be parallelized. This would result in improvements particularly when the client is redrawing the entire image. Additionally, it is extremely inefficient to send pixel data over the internet as an array of integers, which slows down requests.

B. *Node.js Web Server*

Our web server is written in Node.js. It has a dedicated external IP address on Google Cloud and receives protobuf message requests from the local Go client. It partitions each request into smaller sub-requests by dividing the image along the y-axis according to the minimum of either the number of rows or the number of unikernel instances. The server then sends the sub-requests to each unikernel in a Round-Robin scheduling algorithm. The web server keeps track of the number of unikernels and their private subnet IPs by periodically querying the Compute Engine API.

After all the unikernels finish their job and send back their responses, our Node.js server joins all of the sub-responses into one large response with an array representing the number of iterations at each point. The Node server then sends the single response back to the Go client. The Node server sends and receives the same Protobuf message type as the Go application.

There is room for improvement with the Node.js server. because Node is single threaded, there is potentially room for improvement with processing all the network data from the unikernels. This could speed up the response times. Additionally, it was discovered that the function used to merge the protobuf array data was inefficiently copying the array each time. A change to this may further improve the server latencies.

C. *Unikernels*

Our unikernels run on Google Cloud as virtual machine instances. We used IncludeOS for unikernels. Therefore they are written in C++. Unikernels communicate with Node.js server via TCP. They receive protobuf requests from the Node.js server, do the computations for calculating the number of iterations, and send back protobuf responses to the Node.js server. Each unikernel has a private IP within the subnet. After asking Google Cloud for an increase in the number of maximum instances, we were able to create up to 96 unikernels.

We set Google Cloud to handle auto scaling according to the average CPU usage of the unikernels. For example, initially we may have 10 unikernels running in an Instance Group. When their average CPU usage as measured by Google Cloud exceeds 20%, the server will slowly start spawning more unikernels.

IV. Case performance

To test the performance of our IncludeOS unikernels, tests were run from a local Go program that would simulate requests sent from the actual Go client. The test sends requests that ask for the computation of the Mandelbrot image shown in *Figure 2*. It is a slightly misshapen square render of the Mandelbrot image between the coordinates (-1.5, -1) and (1, 1).

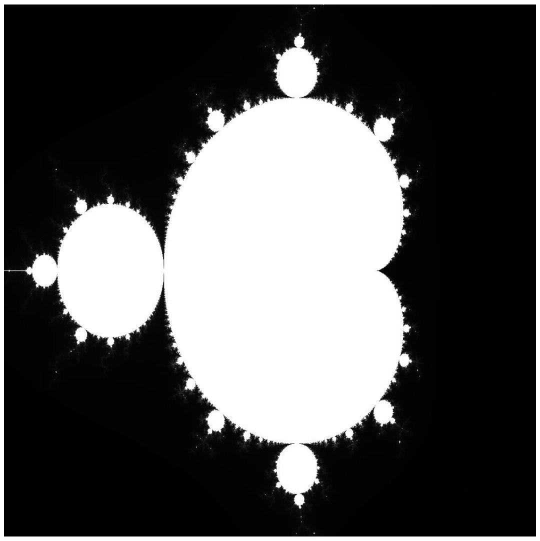

Figure 2. Image of the test request.

For the test, 20 of the requests described above were sent configured at seven different resolutions to better understand the performance of our architecture as the image gets larger. The dimensions chosen were (1x1), (10x10), (100x100), (500x500), and (1000x1000). It is important to remember that the total number of pixels scales with the square of the width. This resulted in a protobuf encoded array of one million integers being sent over the network for the largest image.

Relative response times were measured from the Go client using a timer function and also from the Node client using the Express.js morgan logger middleware. By measuring in two different places, we were able to see results with and without the network latency between the node server and the local client. Figure 3 and Figure 4 show the median time for each configuration divided by the median time for a 1 unikernel request of that configuration. Dividing by the time taken on 1 unikernel gives the relative performance boost gained from having multiple unikernels to compute the request, which is better for comparison. Median times were used because some of the averages are heavily skewed by abnormally slow response times.

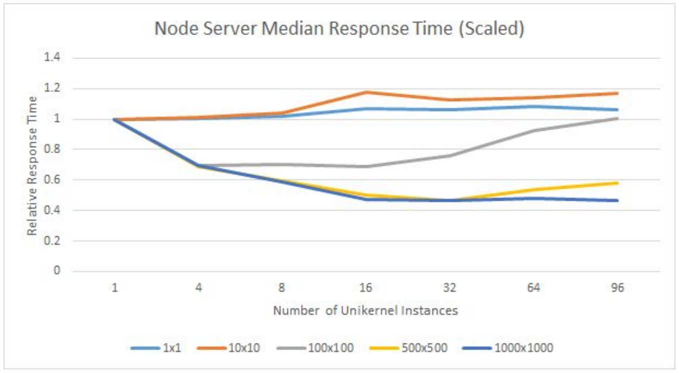

Figure 3. Graph of response times from the Node Server.

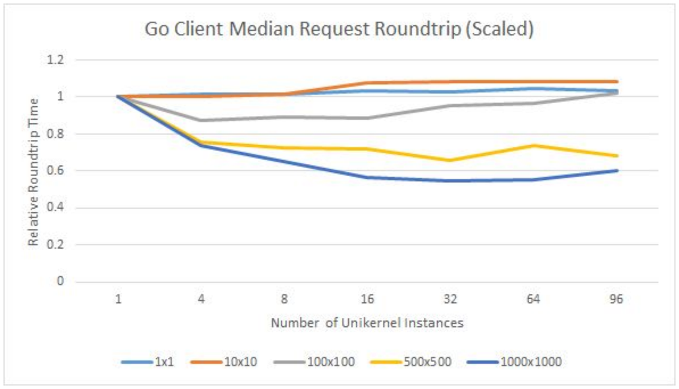

Figure 4. Graph of response times from the Go client

The Node Server times showed a slightly better percent improvement with 1000x1000 at 96 servers than with the Go Client times (0.5 vs 0.6). This was likely because of the network latency between the client and server, which diminished the relative speed boost from parallel computing. Unexpectedly, there was a general lack of improvement in performance after 16 unikernel instances. This may have been a limitation of Node’s single threaded architecture, bandwidth limits, or a result of the slightly inefficient algorithm for merging the responses of the unikernels.

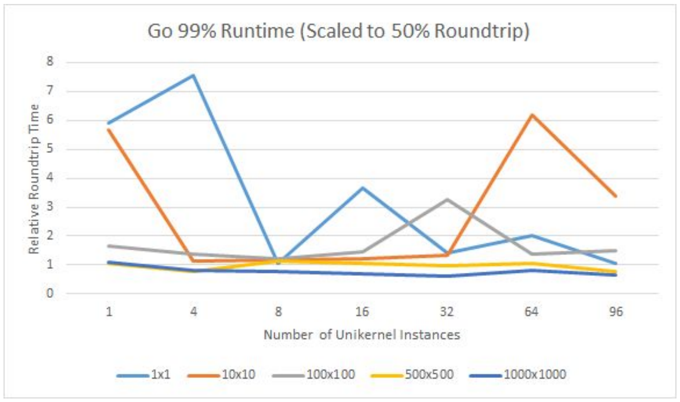

Figure 5. Graph of the 99% percentile response time measured from the Go client scaled according to the median time.

Figure 5 is a scaled graph of the 99th percentile runtimes. Surprisingly, tail latency issues do not seem to be an issue with 96 servers and large images. However, issues do show up with the smaller 1x1 and 10x10 images, with the 99% percentile times being up to 7.5 times longer than the response times at the median. This was an extremely unexpected result, and was likely just a result from random latency between the node server and the local client, which would show up as a larger percent of the total roundtrip time. More testing definitely needs to be done to explore the effects of tail latency.

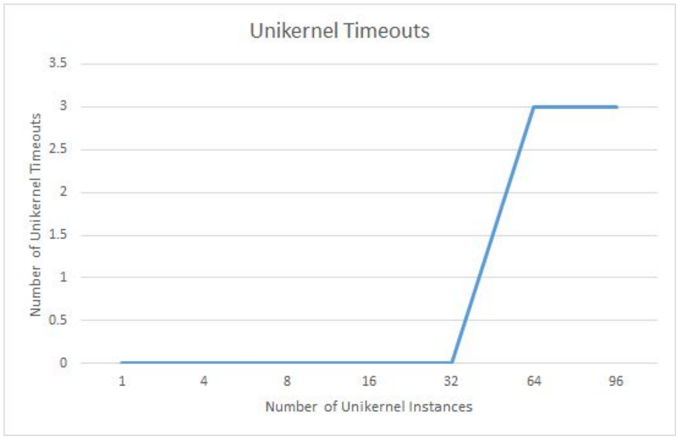

Figure 6. Graph of the number of unikernel timeouts out of 20 test requests.

One unfortunate result that was likely related to a poor implementation of TCP on the Node.js server is that unikernels will occasionally fail to connect. Although a quick fix to this was implemented, it usually only happened on requests sent in rapid succession. Unsurprisingly, this happened three times on the test runs with 64 and 96 unikernel instances.

Tests were not done to directly compare the remote Mandelbrot calculation with local calculation. However, local calculation would definitely result in a smoother and more reliable experience since our implementation introduced large amounts of latency.

V. Future Work

There are many things than can be done to further improve this application.

A. Latency

There were many sources of latency in our program, such as latency in translating to/from protobuf and to the image color. There were also quite a few places in the code that can have the algorithms improved. Some future improvements to the code would definitely be a priority. This would very likely improve the latency and reliability of our program.

One interesting approach to reducing latency would be to run multiple unikernels on a locally hosted server. This would drastically reduce the overall latency and make the remote Mandelbrot calculation have a comparable performance with that of a locally calculated application.

B. Algorithm

One thing was that we can try improving the run time of the algorithm. The implementations of all the algorithms were relatively basic, so there are many optimizations that can be done to improve the overall efficiency.

C. Overall Communication

Another thing is that we can try to improve the parts for the Unikernel, the Web Server, and our local application. For example, currently we used array concatenation instead of array push in Node.js in our web server. This can be improved to further improve the execution of our program.

D. Cloud Application we used

The fourth thing is that, although Google Cloud turned out to be a very good choice for our project, there are still some limitations such as the relatively slow autoscaling. However, an auto scaling service good for the startup speed of unikernels may not be common on cloud providers. Additionally, most cloud providers will likely not have instances with extremely small amounts of hard disk and memory. However, we can try other cloud platforms such as Microsoft Azure or Amazon AWS to see which one best fits our application.

E. Investigate Lower Performance Hardware

Finally, we would like to investigate lower performance hardware for Unikernels, namely Raspberry Pi, to see if we can achieve similar performance in our application.

VI. Conclusion

We learned a lot throughout this project and had a positive experience. The IncludeOS library is good despite certain problems we had during the development phase of our project. We had some struggles with including the protobuf file to the service and spent a lot of time to fix this issue. Turned out the problem was the ordering of the imports of the libraries and we think it would be helpful that the IncludeOS had this information on the documentation. Additionally, the unikernels in Google Cloud seemed to run extremely well. Although the main performance difference between Mandelbrot computation running locally and remotely was latency and network reliability, the unikernels themselves did a superb job at scaling and adjusting to the workload. Other than the struggles, we think that IncludeOS is still a very good tool to use with Unikernel when developing applications.

References

[1] A. Bratterud, A.Walla, H. Haugerud, P. Engelstad, and K. Begnum "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services", 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom). Vancouver, BC, Canada. Nov. 2015

The code for this project can be found at this url:
https://github.com/tgmeow/mandel-canvas