Raspberry PI GPU Auto-Overclocker

Sajeeb Roy Chowdhury
Department of Computer Science
Colorado State University
Fort Collins, Colorado, United States
srchowdh@rams.colostate.edu

Caleb Maranian
Department of Computer Engineering
Colorado State University
Fort Collins, Colorado, United States
cmar1299@rams.colostate.edu

***Abstract*—The goal of this project is to develop a raspberry PI GPU auto overclocker and evaluate the results against two criterions: cost and marketability of the PI overclocker, and power consumption of the overclocked GPU. There are third party vendor overclocking utilities such as MSI Afterburner and EVGA Precision X1. The problem with these libraries is that they are only designed to work on Windows Operating Systems. These libraries use the OC Scanner algorithm to automatically overclock (OC) a GPU. The OC Scanner algorithm sets both the frequency and voltage for overclocking. NVIDIA does not allow Linux to modify the core voltage of their GPUs, instead Linux can only modify core and memory frequency. Therefore we have developed our own auto overclocking algorithm which resides in the Raspberry PI. In this paper we discuss our proposed solution and show that our solution is energy efficient and does not increase the power consumption of an overclocked GPU unlike the OC Scanner algorithm. In this paper we also evaluate the cost and marketability of the project.**

***Index Terms*—Overclocking, GPU, Raspberry PI, Auto-Overclocking, MSI After Burner, EVGA Precision X1**

I. Introduction

A GPU, graphics processing unit, is used in many different capacities. Playing video games, playing simulations, and even 3D Rendering all use the GPU of a computer to process the graphical components of a project. GPU's can be pushed past their standard limitations with a process called overclocking. Overclocking is when you apply overclock profiles and stress test a GPU to increase its performance. With overclocking you get more out of a GPU and utilize it to the best of it's abilities.

II. Problem Characterization

In this project we propose to develop an automatic GPU (Graphics Processing Unit) overclocker using a raspberry pi. The purpose of overclocking is to get more out of a system than what was initially paid for. Often GPU manufacturers such as NVIDIA will release a chip with a certain clock speed, and then release a slightly modified version of that chip with slightly higher clock speeds with a higher price tag. Since people do not want to purchase the more expensive chip, they buy the lower end chip and overclock it to as close as possible to the next tier chip. This process of overclocking is tedious for humans as it is an iterative process. The process of overclocking starts by first maxing out the power limit of the chip, and then iteratively incrementing the clock speed and stress testing the chip after every increment to ensure stability. To ensure the max overclock is reached, the increment is usually very small: 10 – 40 MHz on a GHz scale. If an increment causes instability or crashes the system, the clock is reset to what it was before the last increment and the increment amount is reduced. This reduced increment is then applied incrementally till the system is unstable or crashed again. At which point the reduced increment is again reduced, and this new reduced increment is applied incrementally till the next crash point is found. This process continues till the increment is less than 1MHz or the overclocker loses patience and is happy with the results. For GPUs this process is repeated twice: once for the core clock, and once for the memory clock. It is due to this tedious nature of overclocking, companies such as MSI or EVGA have their own auto-overclockers for GPUs. These are Afterburner [1] and Precision X1 [2] respectively. Auto-overclockers are programs that implement the OC Scanner algorithm that automatically overclocks the GPU using a curve of voltage-frequency values to find the right pair of voltage-frequency values that give a stable overclock. Unfortunately these programs are only for Windows. Under Linux, there exists no auto overclocking programs. The OC Scanner algorithm cannot be used under Linux, since NVIDIA does not allow modification of power and/or voltage offsets for post Maxwell architecture cards under Linux. Therefore, the Linux user has to overclock their GPU using the manual procedure.

III. Proposed Solution

Since we do not have any good auto-overclocking utilities for Linux, and Linux cannot modify the power offset or the voltage offset for NVIDIA cards after Maxwell generation, we are limited to only offsetting the core and memory offsets to achieve an overclock. We also are unable to use the OC Scanner [3] algorithm used by many overclocking utilities under Windows since NVIDIA does not allow setting power offsets under Linux. It is due to this reason, we propose to develop our own algorithm using a raspberry PI server.

A. Overclock Profile Representation

Since we have to send the overclock profiles back and forth between the server and the client, we have decided to represent

each OC profile as a 41 bit integer encoding in order to save bandwidth. This can be seen in Fig. 1. This 41 bit integer contains information about the status of the OC profile, the alive status of the system when stressing with the applied profile, the system identification number, the core clock offset, the memory clock offset, the power offset, the temperature status of the system and the computation error status. The OC profile status is the last bit - bit 40 - in the 41 bit integer. This bit contains information as to whether or not the profile passed the stress test. A value of 1 in this bit position represents that the OC profile passed the stress test. A value of 0 indicates that it failed the test. A value of 1 in this bit position is attained if the OC profile, during the stress testing process, did not crash the system or made the system go into critical temperature regions or did not affect the accuracy of the system. A value of 0 is attained otherwise. The second to last bit - bit 39 - in the 41 bit integer is the alive status of the system. This bit indicates whether or not the system crashed during the stress test with the applied OC profile. A value of 1 in this bit position indicates that the system was stable and did not crash during the stress testing process. A value of 0 indicates that the system crashed during the testing process. Bits ranging from bits 38 to 31 represent the unique identification number of the system. This is for future expandability of the algorithm to be able to overclock multiple systems at once instead of one system at a time. The next 11 bits, ranging from bit 30 to 20 represent the core clock offset that is being applied by the OC profile. Bits 19 to 9 represent the memory offset that is being applied by the OC profile. Bits 8 to 2 represent the power offset. As of the time of writing this report, setting the power offset under Linux is not allowed for cards starting from the Pascal generation - i.e. cards starting from GTX 1060. Therefore, these 8 bits are left unused and are there for future expandability of the algorithm when post pascal cards support setting power offsets under Linux. Bit 1 is the temperature status bit. A value of 1 for this bit means that the system did not reach the critical temperature region when running stress tests with the overclocked profile. A value of 0 means that the system entered the critical temperature region at least once. Bit 0 represents the computation error status. A value of 0 for this bit indicates that there was a computation error during stress testing. A value of 1 indicates that there was no computational error. A value of 1 is only attained if the OC profile did not compromise the accuracy of the system.

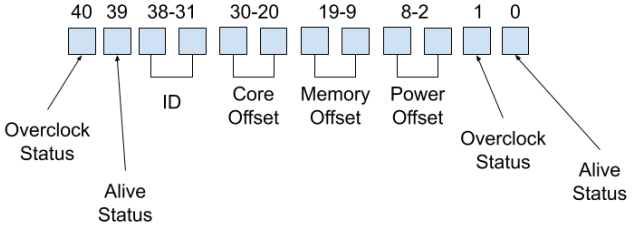

Fig. 1. Diagram representing the bit encoding of the OC profile

B. Algorithm overview

Our OC algorithm for GPU core overclock is divided into two parts: the part that resides on the host system and the part that is on the raspberry PI. The part of the algorithm that is on the raspberry PI is the core of the algorithm. This is the part of the algorithm that generates new OC profiles based on the status of the current OC profile on the client and sends this profile to the host system over Ethernet to be evaluated. In this section we provide a generalized overview of host/client and server portions of the algorithm.

1) Host/Client: The host machine after receiving a new profile from the server applies it and runs the stress testing and monitor processes in parallel. The stress test process stresses the system by training a fully connected feed forward neural network for the classification of hand drawn digits for 20 iterations and evaluating the accuracy of the training afterwards. The monitor process constantly monitors the temperature, power draw and clock frequency at one second time intervals. The temperature parameter collected by this process does have an effect on the status of the OC profile. If the temperature at any one second time interval is above or equal to the slowdown temperature of the GPU, then the program on the host machine that started the stress and monitor process terminates both of these processes, sets the appropriate flags to the 41 bit integer encoding of the OC profile and sends it to the server to get a new profile.

If the program on the host machine detects that it had to terminate the stress and monitor process due to temperature reasons, then the temperature bit in the OC profile encoding is set to 0. Otherwise it is set to 1. If the stress process started by the program on the host machine caused the system to crash, then the host program sets the OC status, the system alive status, and the computation error status in the OC profile encoding to 0. If the stress process did not crash the system and terminated successfully but the numerical error in the computed result is not within 2^{-52}, then the computation error status bit of the OC profile is set to 0, otherwise it is set to 1. The numerical error, ϵ, in the computed result is calculated by subtracting the expected value of the weight tensor from the reference model from the weight tensor obtained in the current training process run by the stress program in a separate process and calculating the 2-norm of the resultant tensor. This is represented formally in the following equation where W_{ref} and W_c are the reference and current weight tensors respectively with dimensions m, n and p:

$$\epsilon = \sum_{i=0}^{m}\sum_{j=0}^{n}\sum_{k=0}^{p}(W_{ref}[i,j,k] - W_c[i,j,k])^2$$

After the program running on the host machine sets the appropriate bits of the encoding it is sent to the raspberry PI server along with a string appended to the end of the encoding that either reads $MEMORY$ or $CORE$ separated by a comma character. This string tells the algorithm on the PI whether to generate a new profile by changing the memory offset or the core offset. Decision to append the $MEMORY$

or $CORE$ string is determined by the program running on the host machine. The program running on the host machine starts off by first applying the core offset in the OC profile encoding while setting the memory offset to zero. During this time every request containing the encoding sent to the server has the $CORE$ string appended to it. Once the host program is able to run an OC profile - that changes only the core offset - successfully for 20 iterations in a row, it determines the core offset specified in that profile to be the final stable offset that when applied to the core clock does not compromise accuracy or system stability. Once a stable core offset has been found, the host program then moves on to overclocking the memory by setting memory and core offsets to that specified in the profile. During this time every request sent to the server containing the encoding has the $MEMORY$ string appended to it. Once the host program is able to run an OC profile that has both the memory and core offsets for 20 iterations in a row it is done overclocking the system. Generally the core offset in the profile is left unchanged by the sever if a stable core offset with zero memory offset was found.

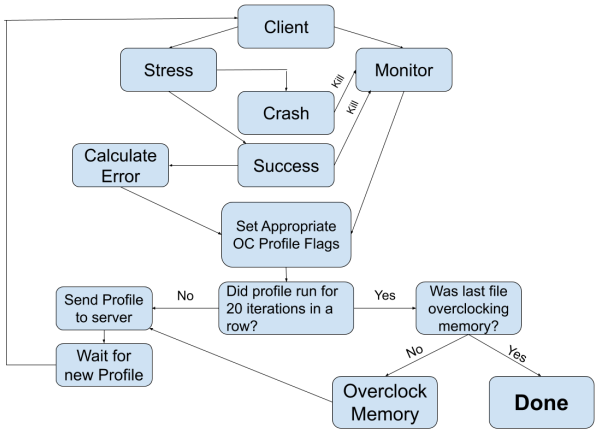

Fig. 2. Schematic diagram for client side implementation

2) Server: The algorithm on the server maintains seven variables which it uses to generate new OC profiles. These variables are: last known good profile, a list of failed core clock offsets, a list of failed memory clock offsets, max core clock offset, max memory clock offset, core clock offset increment, and memory clock offset increment. This algorithm can be broken up into two components: the max offset finder, and the stabilizer.

The purpose of the max offset finder is to find and set values for the max clock offset and max memory offset variables. The max clock offset value is found by first setting the core clock offset increment to a value of 40MHz and generating a last known good profile whose core and memory offsets are set to zero. A new profile is then generated from this last known good profile by applying an increment of core clock offset increment value to the core clock offset in the last known good profile. This new generated profile is then sent to the client/host to be evaluated. If the evaluation passes, then this new profile is set as the last known good profile and another increment of core clock offset increment is applied to the core clock offset of this last known good profile and sent to the client to be evaluated. This process continues till the server receives a profile that failed. When this happens, the value of the core clock offset in the failed profile is set as the value of max core clock offset variable. This same procedure is repeated when finding the max memory clock offset value. When this portion of the algorithm finds a max value for the memory or clock offset, it adds them to the list of failed memory and clock offsets respectively. This is because the max offset finder does not assign an offset as max if it did not fail the host evaluation. This portion of the algorithm has to run before the stabilizer portion.

The stabilizer portion of the algorithm is responsible for finding a stable clock offset for both the core clock and the memory clock given the max clock offsets for both. This is done by incrementing the core clock or memory clock offset in the last known good profile by the core clock or memory clock offset increment values while also making sure that after the application of the increment values, the offsets in the profile do not exceed their respective max values. If either the core offset or the memory offset in the last known good profile exceeds their respective maximum values, then the respective increments are halved and reapplied. This process of halving the increment continues till an increment is found that after application does not exceed the offset maximum values: max core clock offset for application to core offset, or max memory clock offset for application to memory offset. If, after halving, the increment is zero, then the increment is set to -1. This applies to both the memory and core increments. After the application of the offset increments, the newly generated OC profile is sent to the client to be evaluated. If the evaluation passed, the last known good profile is set to the current OC profile that passed and the above procedure repeated to generate a new OC profile. If the evaluation fails, the max offsets are set to the offsets in the failed OC profile and the above procedure repeated to generate a new profile. This portion of the algorithm either stabilizes the core or the memory clock offset depending on the string that was appended to the end of the OC profile encoding in the request received from the client. If the string read $CORE$, then the core offset would be stabilized, else if the string read $MEMORY$, then the memory offset would be stabilized following the above procedure.

When the server first receives the base OC profile from the client, it runs the max offset finder till it finds a max offset for either the core or the memory offset depending on the appended string at the end of the client request following the encoding separated by a comma. If this string reads $MEMORY$, then the memory max offset finder is run, else if it reads $CORE$, the core max offset finder is run. After the algorithm finds a max offset for the memory or the core, it runs the stabilization portion to find a stable offset for core and memory. The stabilizer also modifies the max offsets so that

the algorithm converges to the right stable OC profile offsets.

IV. Implementation of Proposed Solution

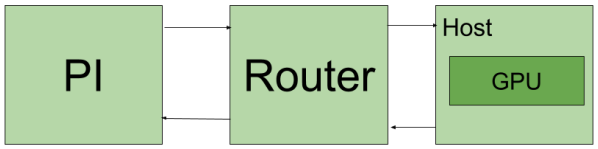

Fig. 3. Current Setup for Testing.

Due to certain complexities associated with socket programming, we have implemented the proposed algorithm in Python. The implementation contains 8 modules. These are: the instruction decoder module, the monitor module, the neural networks module written using PyTorch, the stress module, the GPU module, the OC algorithm module and the server and the client module.

The instruction decoder module is responsible for decoding the 41 bit integer OC profile encoding and returning or setting flags and values such as the temperature status flag and/or the core offset value.

The monitor module is one of the two processes forked by the program on the host/client system when running stress tests on the OC profile. The purpose of this module is to monitor and report the temperature of the system, power draw, current core clock speed, and current memory clock speed.

The neural network module is part of the stress module, in that this module is used in the stress module to stress the system by running a training task and assessing the stability and accuracy of the system. This module is entirely implemented using PyTorch [4].

The purpose of the stress module is to use the neural network module to create and train a neural network model for the classification of hand drawn digits obtained from the MNIST hand drawn digits dataset [5].

The GPU module is a wrapper for NVIDIA settings and smi programs [6] [7] [8]. The nvidia-settings program is used to set certain attributes to the GPU such as setting the core clock offset, and the memory offset. The nvidia-smi program is used to collect information about the GPU such as temperature, power draw, core clock and memory speeds. This module wraps the settings program from NVIDIA and uses it to set the core and memory offset by providing high level function abstractions to low level call of the settings program. This module also does the same to the smi program, in that it wraps the smi program and provides high level function abstractions to low level smi program calls for getting critical information such as temperature, power draw, and clock speeds from the GPU.

The OC algorithm module contains the algorithm that is applied by the server to the OC profile received from the client to generate a new OC profile. This module is entirely and only used by the server module. This module contains 7 variables that it uses to keep track of its progress and the changes it made to the received OC profile to generate a new profile. This module contains only two functions: the adjustment function and the generate new profile function.

The adjustment function can be divided into two components: the max offset finder and the stabilizer. The max offset finder is used to find the initial maximum offset for either the core or the memory offset depending on the appended string at the end of the request from the client. This process continues for either the core or memory overclock - depending the $CORE$ or $MEMORY$ string - until this algorithm generates an OC profile that is able to run for 20 times in a row without causing the host system to become unstable or compromising its accuracy.

The server module is the part of the entire process that resides on the raspberry PI and applies the overclock algorithm in the OC algorithm module to OC profiles it receives from the client to generate a new OC profile and send the new profile to the client. The server continuously waits for a request from the client. Once the client – in the client module – establishes a connection, it sends the OC profile it evaluated to the server. The server passes this received OC profile through the algorithm module to generate a new OC profile and send it to the client.

The client module is the last module in the entire system. In this paper, the client and the host are used interchangeably, since in the context of this paper we are referring to the host as a machine that has a GPU that is connected to the PI. The client module is responsible for sending and receiving OC profiles back and forth from the server. When the client receives an OC profile from the server it immediately starts evaluating it and sends the result of the evaluation to the server to get a new profile. This process is repeated till an OC profile is found that runs successfully for 20 iterations in a row.

V. Evaluation

We have evaluated out PI overclocker against two criterions: the cost and marketability of the product, and the power consumption of the overclocked GPU using this product.

A. Power Consumption

We have evaluated the power consumption of our PI overclocker using the current setup in Fig 2. In particular, we collected power consumption data - in watts - during stress testing of the system during the overclocking process on a GTX1080Ti. Figure 3 to 9 displays the power drawn by the GPU during stress testing with different core offset amounts. From these plots, it can be seen that the maximum power rating for the 1080Ti is around 38 Watts, and the average of the power consumption over time hovers around 30 Watts regardless of the core offset. This implies that the overclocking algorithm that is implemented in our PI is able to push the core clock of the GPU without consuming much power, since the average power consumption with an offset of 0MHz is around 32 Watts, and the average power consumption with an offset of 211MHz is around similar 30MHz. One explanation for the

minute change in power consumption to the significant change in core offset is due to the fact that our algorithm does not modify the power/voltage offset of the GPU unlike the OC Scanner algorithm which uses the voltage-frequency curve to change both the voltage and the frequency.

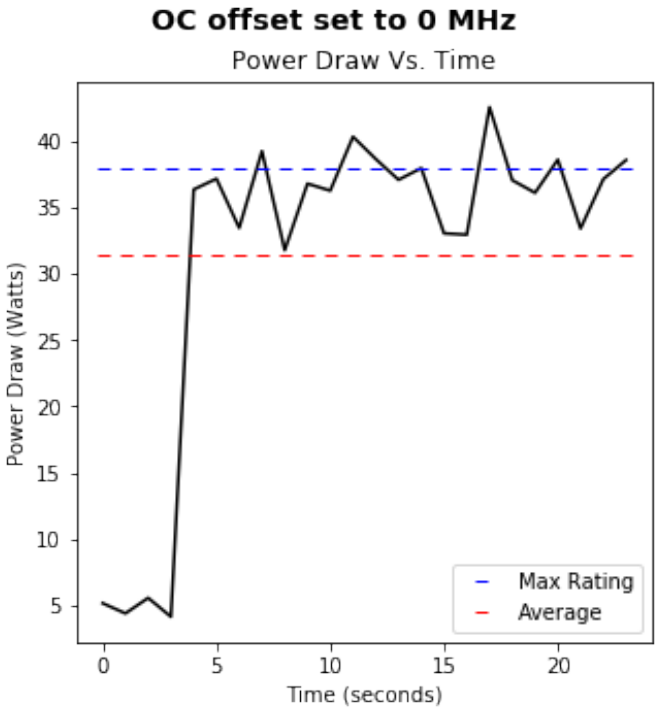

Fig. 4. Power consumption over time during stress testing with 0MHz offset

B. Cost and Marketability

We also evaluated the marketability of a raspberry PI implementing our algorithm. The marketability of an automatic GPU overclocker for Linux machines is tremendous since no other third party programs exits. This technology is genuinely useful and with a limited amount of competitors. The cost for such a software is also very low since we have already written the code and all a user would need to run it would be a NVIDIA GPU and a raspberry pi. The total cost for the entire set up is about 80 dollars for a pi and a router. We could not sell the algorithm for sever wide distribution as the software we developed only works on one device at a time.

VI. Conclusion

Being able to push the boundaries of a GPU makes it, in theory, a higher end product. In this paper we have developed our own overclocking algorithm for Linux machines and have evaluated it against two criterion: costs and marketability and power consumption. We have shown that our developed algorithm is energy efficient and does not consume more power when overclocked as shown in Figures 3 - 9. The power consumption remains about the same regardless of the overclocked frequency. A raspberry PI implementing our algorithm is also marketable to Linux users who want to get

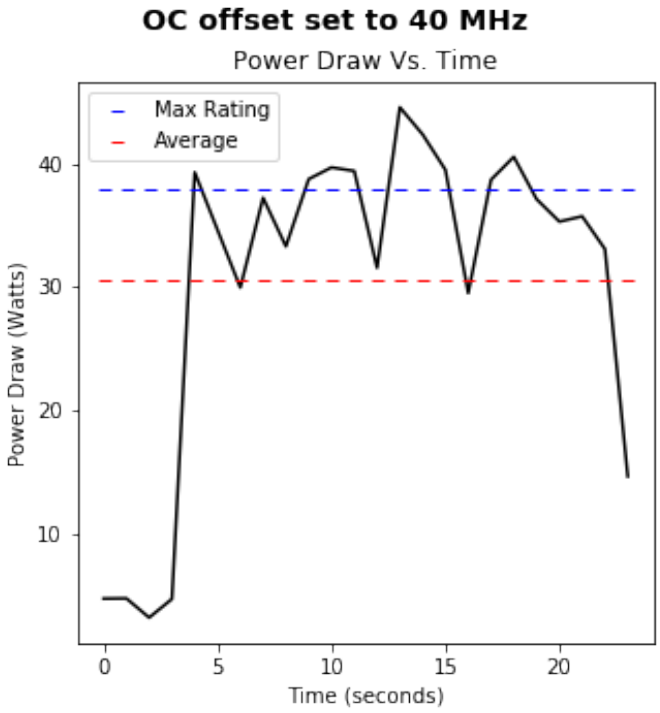

Fig. 5. Power consumption over time during stress testing with 40MHz offset

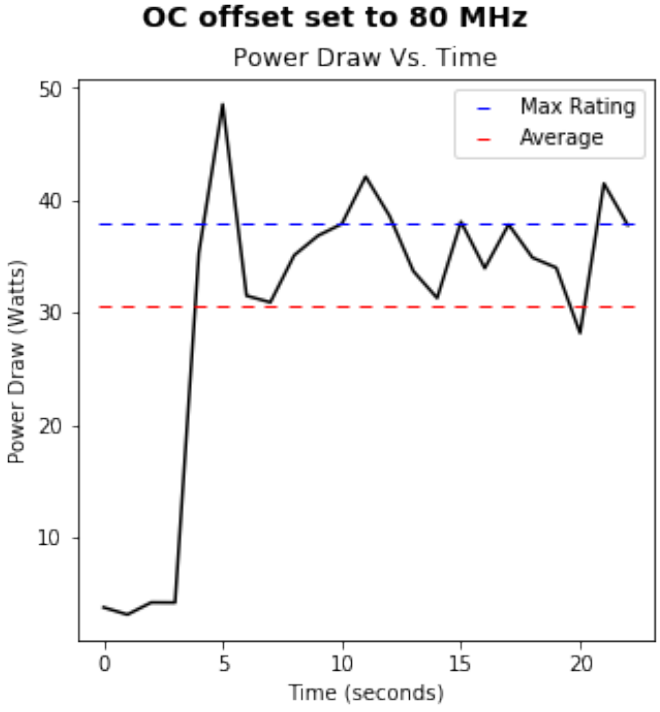

Fig. 6. Power consumption over time during stress testing with 80MHz offset

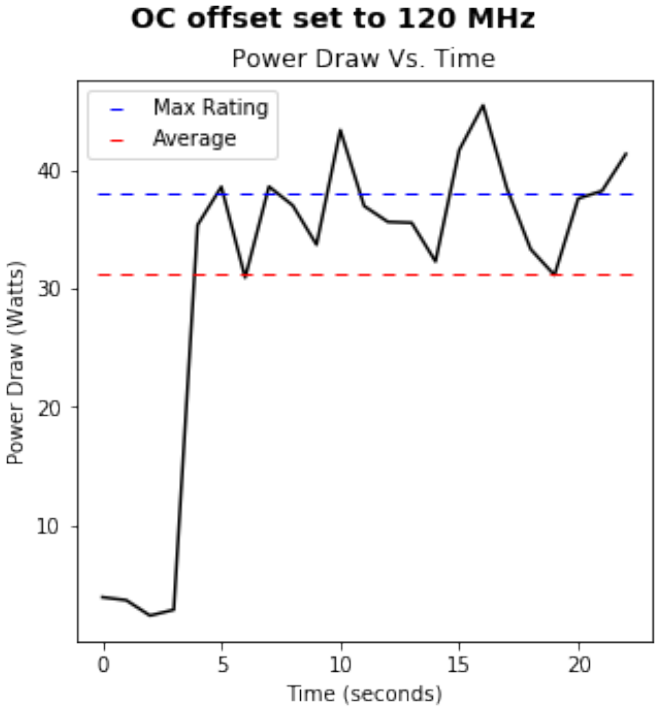

Fig. 7. Power consumption over time during stress testing with 120MHz offset

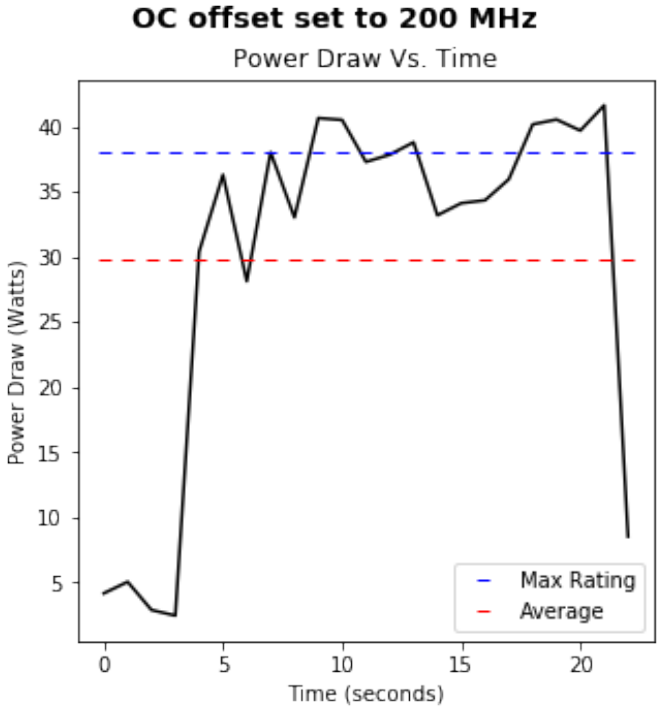

Fig. 9. Power consumption over time during stress testing with 200MHz offset

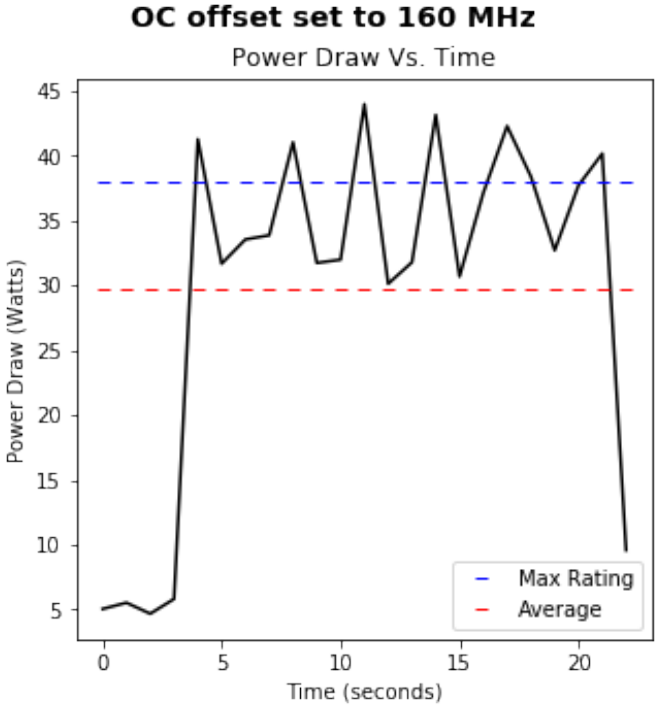

Fig. 8. Power consumption over time during stress testing with 160MHz offset

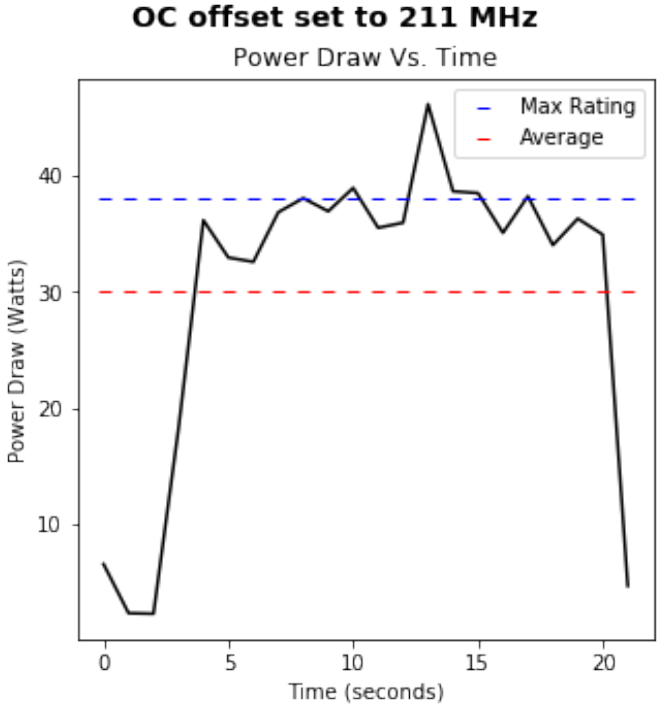

Fig. 10. Power consumption over time during stress testing with 211MHz offset

more performance out of their GPUs without having to spend tedious number of hours manually overclocking. This project is currently not as marketable to high performance computing server applications since it can only overclock one GPU at a time. In a server firm where there are thousands of GPUs this becomes tedious. This project might be marketable to server applications if the algorithm can be modified to overclock multiple GPUs at once instead of one GPU at a time.

REFERENCES

[1] Msi.com. (2019). Afterburner. [online] Available at: https://www.msi.com/page/afterburner [Accessed 6 Dec. 2019].

[2] EVGA. (2019). EVGA - Software - EVGA Precision X1. [online] Available at: https://www.evga.com/precisionx1/ [Accessed 6 Dec. 2019].

[3] Thubron, Rob. "Nvidia's 'OC Scanner' Automatic Overclocking Feature Is Now Available for Pascal Cards." TechSpot, TechSpot, 28 Dec. 2018, www.techspot.com/news/78031-nvidia-oc-scanner-automatic-overclocking-feature-now-available.html.

[4] "Welcome to PyTorch Tutorials¶." Welcome to PyTorch Tutorials - PyTorch Tutorials 1.3.1 Documentation, pytorch.org/tutorials/.

[5] "Welcome to Deep Learning." Deep Learning, deeplearning.net/.

[6] "Useful Nvidia-Smi Queries." NVIDIA, 6 Nov. 2017, nvidia.custhelp.com/app/answers/detail/a_id/3751/ /useful-nvidia-smi-queries.

[7] Developer.download.nvidia.com. (2019). [online] Available at: https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf [Accessed 6 Dec. 2019].

[8] GitHub. (2019). NVIDIA/nvidia-settings. [online] Available at: https://github.com/NVIDIA/nvidia-settings/blob/master/src/parse.c [Accessed 6 Dec. 2019].