NVIDIA Tegra Performance:
Power Sensing Daemon and NvPerfScan Profiling Support

A Major Qualifying Project
submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

WPI routinely publishes these reports
without editorial or peer review

by
Danilo Augusto Correia Da Silva
Eren Zeyn Eroglu
Adam Grabowski
Emmanuel Ola

Date:
2 December 2022

Advisor:
Mark Claypool

NVIDIA Mentors:
Mitch Luban
Allen Martin
May Narayanaswamy
Zubair Waheed

Abstract

NVIDIA collects performance metrics to compare their products to competitive counterparts and develop solutions to any bottlenecks. Power sensing data is crucial to create low overhead programs and minimize resource wastage. One project task was to make power data collection easier by converting NVIDIA's power sensing program into a daemon with a well-defined API using named pipes and multi-threaded programming. Another project task was to update NVIDIA's performance scanning infrastructure to support profiling tools. This involved updating its command line to execute a chosen profile, uploading collected data to a database, and developing a method for visualizing that data. We held daily scrums and tested iteratively to implement a power sensing daemon and add profiling support.

Table of Contents

Table of Figures

Chapter 1: Introduction

As any organization that creates products to be widely used by a variety of consumers, NVIDIA's development cycle has a rigorous testing cycle to go along with the rest of their development. An integral part of this testing cycle is several benchmark applications that are run on NVIDIA proprietary machines and GPUs as well as competitive counterparts. NVIDIA needs profiling tools that run against benchmark applications on these machines to measure specific CPU and GPU metrics that occur. When collecting profiling information, NVIDIA developers have to manually run benchmark applications to collect data which is cumbersome and time-consuming.

One goal of this project was to add support for tools such as Linux perf and Nsight Systems to NvPerfScan, NVIDIA's performance scanning framework, allowing developers to collect CPU and GPU metrics from Linux perf or Nsight Systems during a benchmark run. This automates the process of running numerous benchmarks so that developers so not have to run tests manually. We designed, implemented, and evaluated an update to the NvPerfScan framework that adds options for users to specify which benchmark applications need to be profiled with chosen options.

One other form of performance metric data is power usage. NVIDIA is specifically interested in making power usage data easier to acquire as it is an instrumental part of the software design process. NVIDIA has an existing power sensing program; it is tedious and time consuming to use. This means developers are wasting time trying to collect and parse data that is essential. Ultimately, the goal of both projects is to make specific performance data easier to access and collect.

In order to tackle the power usage performance data problem, we also converted an existing proprietary NVIDIA power sensing program to a daemon with a well-defined API. This daemon enables NVIDIA developers to successfully collect system power-sensing data to measure application power usage. To test our program, we created test code that increased system power usage. More information regarding testing can be found in chapter 5.1 and Appendix A respectively.

We participated in daily scrums, communicated with mentors, and iteratively tested our codebases to complete each project task. Results of this project include a power sensing daemon implementation for NVIDIA's DriveOS application and the addition of profiling support to NVIDIA's performance scanning framework.

Subsequent chapters are broken down into subsections based on the project task. Chapter 2 provides background on related technologies including descriptions of applications and tools that were used in implementation. Chapter 3 delineates our methods for communication and code development throughout the project. Chapter 4 describes in detail our implementation of the project tasks: power sensing daemon and performance scan profiling support. Chapter 5 outlines our evaluation of these solutions, including verification and points of success. Chapter 6 concludes each project task and describes potential features to be added in the future.

Chapter 2: Background

2.1 Power Sensing Daemon

DriveOS is NVIDIA's "embedded real-time operating system" (1). A real-time operating system is used in applications requiring potentially time sensitive tasks; DriveOS is specifically designed to be used in autonomous vehicles. Since autonomous vehicles are systems that must behave predictably, often with life-or-death consequences if they do not, ensuring that programs running on DriveOS utilize as few resources as possible to limit potential waste is essential.

This is where power sensing programs are integrated into the design of autonomous vehicles. Power sensing enables developers to test their applications to ensure that as few system resources are being utilized as possible. Additionally, ensuring that power sensing tools are accessible and easy to use can reduce the amount of time developers have to spend optimizing their programs. Our team's goal was to make the existing power sensing program used by NVIDIA developers much easier and less time consuming to use and parse information from.

2.2 NvPerfScan Profiling Support

TH500 is an ARMv9a based architecture and is the first chip of its kind at NVIDIA. They want to understand the abilities and limitations of TH500 against competitive systems like those developed by Intel or AMD. Platforms for development and performance evaluation of this chip include Colossus, a virtual machine leasing service, and PerfLab, a performance testing lab which provides the PostgreSQL database used in NvPerfScan.

To facilitate easy collection of benchmarking data, NVIDIA has a framework called NvPerfScan. NvPerfScan is a lightweight utility that automates the process of running various benchmarks, parsing collected results, and uploading scores to a database. It is easier to run numerous benchmarks with an automated infrastructure for testing new CPU architectures against other competitive systems. Records from the database are linked to a Tableau dashboard where results from various competitive systems can be viewed and analyzed together. As shown in Figure 1, the NvPerfScan tool includes python scripts for running the tests which generate JSON result files. Then, there are upload scripts that make results available on a PerfLab PostgreSQL database to be visualized in Tableau.

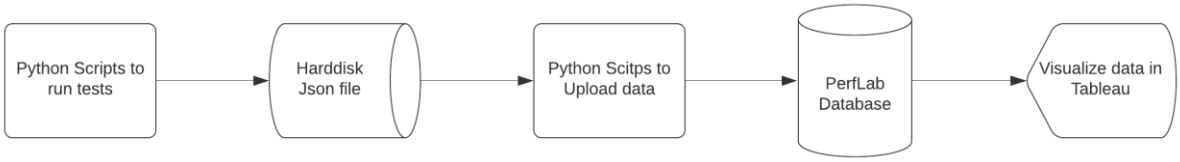

Figure 1 - NvPerfScan flowchart

Linux perf is capable of lightweight profiling, as it can instrument CPU performance counters, tracepoints, kprobes, and uprobes. A list of measurable events which can come from different sources, including hardware, software, and tracepoint events, is supported by the tool and underlying kernel interface. For any of the supported events, perf can keep a running count during process execution by aggregating the occurrences of events during a benchmark run using perf stat. When using perf stat profiling, the cumulative counts can be presented in a csv file which correspond to their respective events. The perf tool can also be used to collect profiles by collecting samples using perf record which generates a perf data file to be analyzed using perf report, perf script, or perf annotate. A hardware counter is used to signal the kernel to record a sample about the execution of a program which depends on the options specified by the user and tool. For reporting purposes, developers can use the perf report command to read the generated perf data file and display a concise execution profile.

NVIDIA's Nsight Systems is a system-wide performance analysis tool designed to visualize an application's algorithms, help identify opportunities to optimize, and tune to scale efficiently across any quantity or size of CPUs and GPUs. Nsight Systems latches on to a target application to expose GPU and CPU activity, events, annotations, throughput, and performance metrics in a chronological timeline. For example, toggling on GPU metrics sampling will collect low-level IO activity such as PCIe throughput, NVIDIA NVLink, and DRAM activity. Nsight Systems can generate profiling information using nsys profile which generates a nsys report, a .nsys-rep file to be analyzed by the Nsight Systems desktop application or nsys analyze command.

PostgreSQL, also known as Postgres, is an open-source relational database management system emphasizing extensibility and SQL compliance. NVIDIA uses a PostgreSQL database to store performance data collected by its scanning tool. Talos is a versatile web interface exclusive to NVIDIA employees for accessing files on engineering scratch spaces, which includes an API to enable interaction with these files. Files uploaded to this network share can later be fetched via http as a GET request. SCP is a command-line utility that allows users to securely copy files and directories between two locations. This tool was used to copy files generated by a profiling run into the network share owned by NVIDIA's performance scanning infrastructure.

Chapter 3: Methodology

Daily scrum meetings were held to update our mentors on the progress we had made that day and ask any questions we had related to the development of our solutions. Communication between our group and mentors was primarily through Slack and daily scrum meetings were held on Microsoft Teams. When required, members of our group would meet after the daily meeting to discuss a problem or plan a project task.

3.1 Power Sensing Daemon

The original power sensing daemon application was created for boards equipped with the NVIDIAs Tegra processor running DriveOS as its operating system. Because of that, every code produced must be tested in that specific hardware. To assist with it, NVIDIA has a proprietary cloud system where programmers can remotely connect to boards called Colossus. In addition to that, development was made through a virtual Linux Ubuntu machine that could be connected to Visual Studio Code using SSH connections.

At the beginning of every week, new boards were reserved for each day of week through Colossus. After a board was reserved, the first thing that would need to be done is connect to the host through the virtual Linux terminal. Once the connection has been established via IP, a build of DriveOS could be copied to the host and flashed. After flashing is done, the board can be accessed using Linux minicom. Some boards could be defective regarding their minicom access. Therefore, remembering specific boards marked as in “good state” is recommended so development is not blocked.

After the board is successfully accessed, connection between both the board itself and the host needs to be established. This is done by turning on specific IP ports on both. This connection must be created so that it is possible to transfer the code from host to board.

Finally, once all the process above is concluded, code from VS Code connected to the SSH Linux can be built into executables and transferred to host, then to the board. As soon as the binaries executables are installed, code can be run and tested. In addition to that, a proprietary version control platform called Gerrit was used to track.

3.2 NvPerfScan Profiling Support

Benchmarks were run on remote machines that had hardware requirements for the benchmark applications (i.e., a competitor GPU, or NVIDIA proprietary GPU). These benchmark applications had to be run on a variety of machine configurations. For this reason, a lot of development was done within remote machines on a cluster of Colossus machines. These machines were leased for the duration of the project with any essential specifications provided for the requirements of the task. In testing Nsight Systems for example, we needed a machine that had an NVIDIA proprietary GPU on it. We also needed our machines to be running Ubuntu, which is a Linux-based operating system.

Since we were interacting remotely with these machines it would have been particularly difficult to write code within the terminal while remotely connected to the machines. For this reason, we use an extension on Visual Studio Code called Remote Explorer. This allowed us to remotely connect to the Linux machines but have a visual representation of the file directory on the machine and additionally make changes via the Visual Studio Code UI.

The NvPerfScan code repository is hosted on GitLab, so we created a new development branch off the main branch to organize our changes. Code development was done primarily asynchronously, and specific tasks were assigned to contributing members during the daily syncs. These task assignments were coordinated so that changes to the code repository did not conflict with each other.

Chapter 4: Implementation

4.1 Power Sensing Daemon

In order to optimize the usability of the existing power sensing program, we first assessed what the existing solution was lacking. One problem with the existing power sensing program was that the program would have to be started and stopped manually. During its runtime, the power sensing program would log all system power usage measurements such as CPU, GPU, Total System, and RAM voltages to a log file. The main issue with this design is that there was no distinct separation between measurements recording during the desired test application's runtime, and idle measurements that were recorded before the test application's start, but after the power sensing program's start.

Implementation of a Daemon

The first solution was to convert the program into a daemon. A daemon is a process/service that runs in the background. We first created a simple program that would just spawn a process and decouple it from the terminal so that it could run in the background. The most important technical detail related to a daemon is that the built-in methods fork, and exit is called to create the background process and leave the parent after its creation.

Inter-process Communication

After we had implemented the daemon functionality, we needed a way to communicate with the daemon process. We explored two separate ways to tackle inter-process communication: named pipes and TCP sockets. Both solutions work in Linux. Since TCP socket communication is far more resource expensive on certain operating systems such as QNX, we used named pipes to increase its potential use cases (2). Named pipes are special types of files that can be accessed by multiple processes to provide bidirectional inter-process communication. Accessing a pipe in reading mode is a blocking operation that waits until something is written in it.

Multithreading and Asynchronous Tasks

After we had inter-process communication working, we converted our existing daemon code into a multi-threaded program. One thread would listen to the unnamed pipe created so that the user can interact with the daemon, and the second thread collected the power usage data and logging it to a file. This multi-threaded approach prevents the program from being blocked by reading the pipe if nothing is written to it.

Migration from Previous Implementation and Refactoring

Lastly, we moved the power sensing code from the existing power sensing program into our new multi-threaded daemon. This process also involved pruning some of the configuration options that had become obsolete such as the runtime execution and reading the configuration file. The first, used to describe the amount of time that the application would run, but since the program had been converted to a daemon, it runs on background independently. The reading of the config file, used to describe a method that was called once a configuration file has been defined. In our current design, a configuration file is always present, so this functionality was merged into the main function. Additionally, we had to create one function that reset the board's statistics structures and one that returned the statistical analysis as a string, instead of the previous implementation that would automatically write it to the output file.

Pre-Application Cache Implementation

Now that the power-sensing daemon was up and running, we added a cache to the daemon. The motivation for this cache was to help clearly distinguish pre-application power usage measurements from application measurement data by storing the 10 last recorded power sensing measurements before the daemon receives the start command. This cache data is used for system idle measurements. These idle measurements are the system's power usage when the application is not running. Power sensing data is useful if the context of how much power the machine uses while idling is also acquired. All data is then output to a log file and statistical analysis consisting of averages and standard deviations for each channel is returned via pipes to the user's terminal. The entire application's sequence diagram and flow can be seen in Figure 2.

Implementation overview

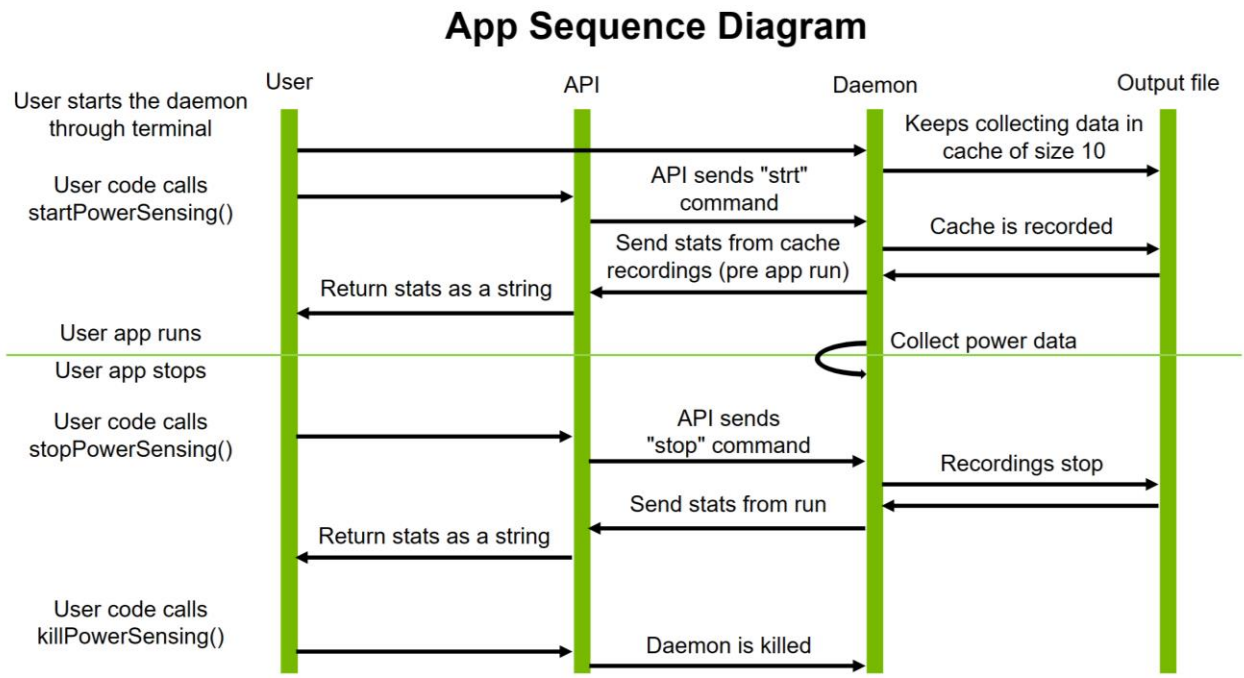

Figure 2 - Power Sensing Daemon Sequence Diagram

As shown in Figure 2, the application is divided into four main entities: the user, API, the daemon itself, and the output file. For the application to start, the user needs to run the program from the terminal. Once that is running, the daemon can be integrated into the user's code, and recordings are continuously being saved in the cache of size 10. From user code, the method *startPowerSensing*, can be called. This function defined in the API will send "strt" through the daemon pipe, telling it to compute the statistics of the cache (pre app run) recordings, and start recordings for user code. Data is collected until the user calls *stopPowerSensing*, that tells the daemon to compute statistics of the data collected while the code was running and stop the daemon recordings. Statistics are sent to the user as strings using another pipe dedicated only for daemon responses. In addition to that, there is also the method *killPowerSensing* that stops the daemon process.

4.2 NvPerfScan Profiling Support

First, we updated the NvPerfScan framework to add options that allow users to specify which benchmarks need to be profiled with Linux perf. We allowed users to specify options for perf stat and perf record, such as CPU and memory statistics, and kernel Performance Monitoring Unit (PMU) counters. Additionally, we collected this data as a sweep, a collection of benchmark applications, and uploaded it into the database by copying profile data files into a Talos scratch space. We then designed a method for viewing and downloading perf data on the Tableau dashboard through clickable links in worksheet views. Lastly, we added similar support for Nsight Systems, but with additional options for GPU metrics.

Profiling Support Commands

Developers using NvPerfScan can specify either one benchmark or a sweep of benchmarks to profile which generates and executes a command for each benchmark. Three profile types are supported: stat or 'perf stat', record or 'perf record', and nsys or 'nsys profile'. This was implemented through the parsing of command line arguments and flags. Using the -p flag with the associated stat, record, or nsys identifiers, users can choose the type of profiling they want to run against a particular set of benchmarks. Developers can also specify any options they want to associate with the chosen profile type using the -o command. Using Python's operating system library, a command was generated and executed based on the options specified by the user. Some examples of how a developer may use this profiling support to generate profiles for various benchmark applications and the associated commands are below.

One Benchmark:

```
./benchmark_shell.py -b sysbench -p stat -o '-e cache-misses' --delay
```

Command Executed:

```
perf stat -e cache-misses -o <output_directory> <benchmark_command>
```

Figure 3 - Perf stat with cache misses event option

Sweep of Benchmarks:

```
./run_sweep.py -b stress-ng -p nsys -o '--trace cuda' --delay
```

One Command Executed in Sweep:

```
nsys profile --trace cuda -o <output_directory> <one_benchmark_command>
```

Figure 4 - Nsys profile with Cuda trace option

As shown in Figure 3, users can run perf stat profiling against the sysbench benchmark application and then specify events to be recorded using the Linux perf '-e' option. This executes a perf record command collecting the total number cache misses that occur during that run. As shown in Figure 4, a developer could run nsys profile against the collection of stress-ng benchmark applications and then specify Cuda API tracing, which collects system events associated with the Cuda tool. The nsys profile command executed for each benchmark generates a .nsys-rep file at the output destination containing the chosen performance metrics.

JSON Results and Upload Perf Report

Results already include high-level system information: CPU model details, CPU and memory frequency, and kernel version. Developers can also view profiling information collected by the perf record, perf stat, or nsys profile commands. Two new columns were added to the JSON results and database to store each profile metric and corresponding score: profile_metric and profile_score. As shown in Figure 5, for the record and nsys profile types, results holds the path to its profile data in a results JSON file but developers can choose to upload to the database later. As shown in Figure 6, if uploaded to the database, results holds the link to download its profile data from our Talos share, which will be available on Tableau. Results collected by the perf stat profile type stores aggregate event counters regardless of whether it is uploaded.


```
{
    "uploader": "NvPerfScan",
    "application": "sysbench",
    "recordid": "sysbench-12.13.22-05:30:16-7fe02e45-3d63-4fab-908b-bd13dfbf3b01",
    "timestamp": "12.13.22-05:30:16",
    "description": null,
    "flags": "mutex --mutex-num=4096 --mutex-locks=50000 --mutex-loops=300000 run",
    "application_version": "1.0.20",
    "specification": "system mutex",
    "profile_score": "output/sysbench/record/sysbench-perf-record-cca2d558-7b90-410d-937e-0d7564774a69.data",
    "profile_metric": "profile path"
}
```


Figure 5 - Perf Record Local results JSON


```
INSERT INTO records(
                recordid, timestamp, dateadded, uploader,
                application, application_version, flags, description, score, metric, profile_score, profile_metric,
                bigger_is_better, specification, cpu_model_name, cpu_family, cpu_model, cpu_stepping,
                cpu_virtualization_type, cpu_address_sizes, cpu_frequency,
                cpu_l1i_kb, cpu_l1d_kb, cpu_l2_mb, cpu_l3_mb,
                number_of_cpu_sockets, cpu_cores_per_socket,
                threads_per_core, number_of_cpu_threads,
                bogomips, os_architecture,
                os_version, os_name, kernel_version, system_memory_gb, system_memory_speed_mhz,
                gpu_model, number_of_gpus
            )
            VALUES (
                'sysbench-12.14.22-18:39:38-8d65d33c-619b-4b55-b6cd-a2d4ad969ac5', '12.14.22-18:39:38', LOCALTIMESTAMP, 'NvPerfScan',
                'sysbench', '1.0.20', 'mutex --mutex-num=4096 --mutex-locks=50000 --mutex-loops=300000 run', 'None', NULL, NULL, 'https://sc.talos
.nvidia.com/api/v1/file-download/home/nvperfscan/profile_data/sysbench-perf-record-eaa97b10-e135-48bd-b110-a72b72a196b4.zip', 'profile download',
                NULL, 'system mutex', NULL, NULL, NULL, NULL,
                NULL, NULL, NULL,
                NULL, NULL, NULL, NULL,
                NULL, NULL,
                NULL, NULL,
                NULL,NULL,
                NULL, NULL, NULL, NULL, NULL,
                NULL, NULL
            );
```


Figure 6 - Upload Link to Perf Record Data

Alternate Solution for Uploading Perf Report

We also implemented a python script that uploaded the perf report binary file directly to the PostgreSQL database by uploading the binary file as psycogp2 binary object. The script was responsible for retrieving the binary file when it is needed by the developer. The group ended up going with the download links approach as we suspect most users would not be retrieving an historic perf report file via CLI.

Tableau Visualization

Profile data is copied into a Talos network share and then a download link is added to the results of a benchmark run. Test results are compiled across various systems and displayed on views within a worksheet on Tableau. Profile metric and profile score were added as columns to our database for storing the type of metric and its associated value that were collected during a particular run. As shown in Figure 7, cumulative event counts collected by perf stat and download links collected by either perf record or nsys profile are associated with a run timestamp, CPU model name, and benchmark application. As shown in Figure 8, links were made to be clickable, so that developers can download the profiling data associated with a given timestamp.

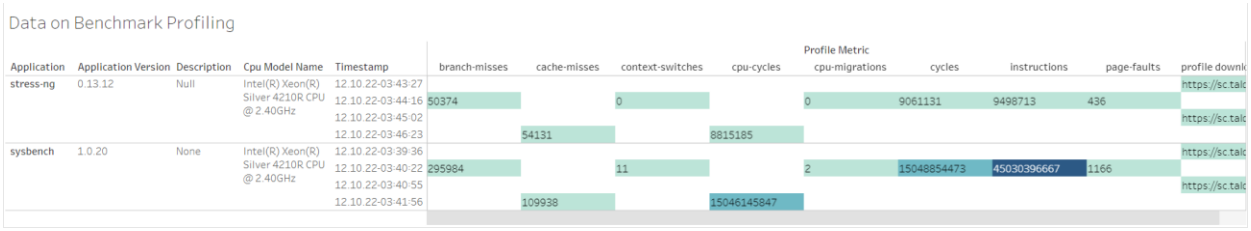

Figure 7 - Tableau visualization for profile data

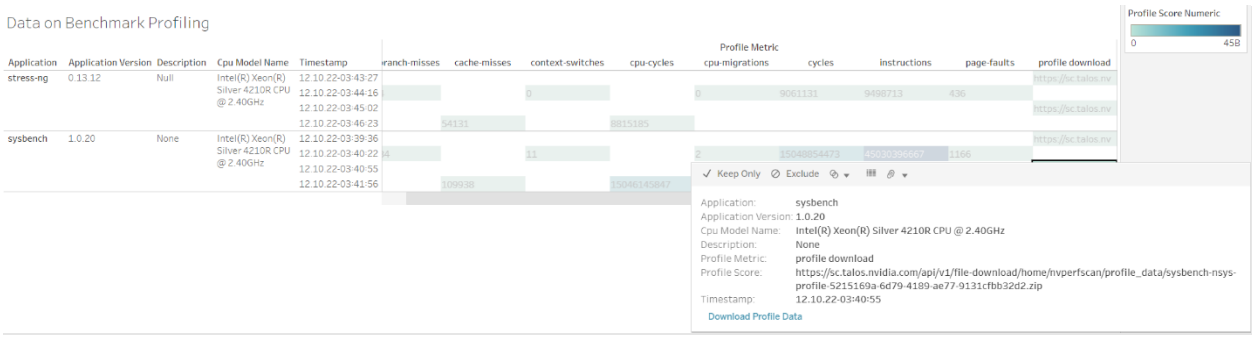

Figure 8 - Profile data download link

Utility Functions

To implement the features described above, several utility functions were written inside of the NvPerfScan Python framework. These functions carry out the primary steps used in our profiling implementation which include getting the profile data, adding profile data to the results JSON, and uploading results to the database.

getProfileData(profile, benchmark, options)

This function takes the profiling type, specified benchmark, and profiling options and returns a python dictionary containing profiling data.

- If perf stat is selected, a csv file is generated and parsed so that each dictionary entry corresponds to the name of an event and its aggregate count.
- If perf record is selected, a perf data file is generated, and the dictionary will contain one entry corresponding to the path of the file.
- If nsys profile is selected, a nsys report file is generated, and the dictionary will contain one entry corresponding to the path of the file.

uploadProfileData(path)

This function takes the path to a file generated by some profile type and uploads a zipped version to our Talos network share by calling 'scp'. It returns an updated 'profile_data' dictionary containing the download link specified by the Talos API. It is only necessary to call this function when uploading results to our database, since profiling results need to be accessed via hyperlink.

scp <profile_data_path> <username>@<nvidia_machine>:/home/nvperfscan/profile_data

As shown in the 'scp' command above, any generated profiling data is copied into a Talos network share folder on one of NVIDIA's Talos machines. Since the nsys profile and perf record profile types generate a data file to be analyzed later, the file is accessible through our database.

result_add_profile(path, profile_data)

This function takes the path to JSON results and appends to its list of dictionaries, adding one new result dictionary for each 'profile_data' entry. Each new profiling dictionary entry will have new keys, 'profile_metric' and 'profile_score', which correspond to the key(s) and value(s) present in 'profile_data'.

stat_perf(opts, path), record_perf(opts, path), nsys_profile(opts, path)

There are also several functions for forming and executing the system calls for each profiling type.

- Stat perf executes a perf stat command with options, generating a .csv file to be parsed.
- Record perf executes a perf record command with options, generating a .data file.
- Nsys profile executes the nsys profile command with options, generating a .nsys-rep file.

Chapter 5: Evaluation

5.1 Power Sensing Daemon

To ensure that our architecture works as expected, a sample of a use case code was developed. The same sample was also tested using different configuration files entered by the user. Results were then analyzed and compared to the original implementation in order to track any anomalies and bugs. Figure 9 describes the architecture of the sample code developed.

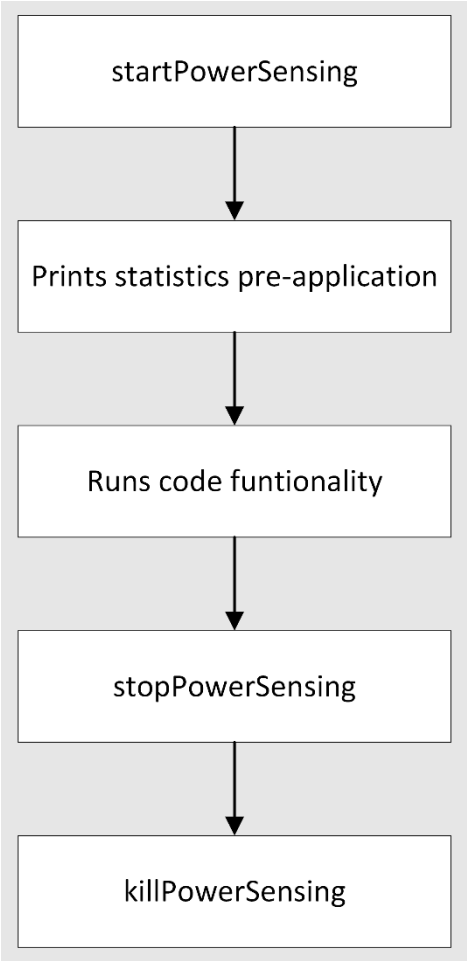

Figure 9 - Sample Code Flowchart

In addition to that, different configurations were tested. Users can add different configurations through the configuration file, that can be passed an argument when starting the daemon. The structure of it is described below:

```
sudo ./tegra-power-sensing <optional-absolute-path-to-config-file>
```

The daemon must be run using sudo commands because of its use of pipes, that are written into a system's files, and not the specific user. The configuration file can have the following options written by the user:

- Number of bus conversions
- Disable any component
- Number of samples per average
- Number of shunts
- Conversion time in milliseconds

5.2 NvPerfScan Profiling Support

Our team collected profiling data as part of a sweep and uploaded it into the staging database. We developed a means by which this recorded information can be downloaded and viewed from the Tableau dashboard. Finally, we ran perf stat, perf record, and nsys profiling with different options on multiple competitive systems to ensure compatibility. We demonstrated these use cases to our sponsor and received approval for our changes to the NvPerfScan codebase and associated PostgreSQL database. Based on our tests, demonstration, and the expectations of our sponsor, the additions successfully integrated profiling support into the NvPerfScan framework.

Verification

With the pieces together, we made sure that the profiling tools worked as intended with the scripts we had written. We evaluated our code iteratively during implementation and upon adding each profiling tool to the script. This involved checking the results JSON to verify that the profile metrics and scores made sense based on the user input. For example, after a perf stat run, we expected to see a list of aggregate event counts, but after a nsys profile run, we expected to see one additional entry with a path to its .nsys-rep file. We first made sure that with a single benchmark run, we are able to get a result file for an individual benchmark run in the appropriate output directory. We also included logs in the script to ensure everything was working as needed in the intermediary steps.

From Staging to Production

In testing, we used a staging database to collect profiling information and debug any issues that arose. This way, we did not have to worry about removing erroneous benchmark runs, as they did not affect the data used by other NVIDIA developers. At the end of this project, when all use cases were tested and verified, we replicated the changes from the staging database on the production database. As described in prior sections, this involved adding two columns to the records table: 'profile_metric' and 'profile_score'. Also, we updated the production database Tableau worksheets to include our profile data view shown in Figure 5.

Chapter 6: Conclusion and Future Work

It is important that NVIDIA has efficient and effective methods for collecting performance data, as they need to test the reliability, stability, and speed of their new applications and hardware systems. Both project tasks successfully made it easier for NVIDIA to collect performance statistics in power consumption and computer metrics.

6.1 Power Sensing Daemon

The power sensing daemon has been successfully implemented. Teams working on DriveOS applications can now measure power consumption for specific programs and blocks of code. CPU, RAM, system, and total power data is outputted into a file that can later be exported and analyzed as a CSV (Comma Separated Values). In addition to that, users can have their own configuration file passed as an argument at the start of the program or use the default configuration. The use of the power sensing daemon tool can help NVIDIA teams to produce more power efficient programs, that consequentially can turn into better products for their customers.

There is still work that can be done to improve the application including, but not limited to: implementation of a terminal interface where the user can call daemon functions directly from terminal, adding the difference in measurements as part of the *stopPowerSensing* method, improving build system integration for easy usability by user programs, and finally, expanding the program systems running QNX as well.

6.2 NvPerfScan Profiling Support

Our project team successfully updated the NvPerfScan framework, allowing users to specify benchmarks that need to be profiled with Linux perf and Nsight systems. Even though NVIDIA has benchmark data for different applications and use cases, they only collected high level information such as CPU model details and kernel version. When results were collected from systems NVIDIA wanted to investigate further, it was required that tests be run manually. Adding profiling support to the NvPerfScan framework addressed this problem, as developers can now collect profiling information associated with a particular benchmark run. Evaluation of this result described in Chapter 5 proves the viability of our solution.

In terms of potential future work, NVIDIA may add support for default profiles and options to be ran for each specific benchmark. As a part of the initialize script for the NvPerfScan tool, a JSON file that stores the name, specification, and flags for each benchmark is generated. If NVIDIA developers decide they want a particular benchmark to always be run with a certain profiling type, this could be specified as an additional field in the JSON. This way, developers would not need to specify a profile type in the CLI if a default profile is always provided. Also, they may add support for additional profiling types besides perf stat, perf record, and Nsight Systems. For example, gprof, Valgrind and Google's gperftools are all potential candidates for profile tools that collect performance metrics and provide reporting mechanisms.

References

McGraph, R. (2022, 12 13). *Pipes and FIFOs.* Retrieved from The GNU C Library: https://www.gnu.org/software/libc/manual/html_node/Pipes-and-FIFOs.html

NVIDIA. (2022, 12 12). *NVIDIA DRIVE SDK.* Retrieved from NVIDIA Developer: https://developer.nvidia.com/drive/drive-sdk

NVIDIA. (2022, 12 14). *NVIDIA Nsight Systems User Guide.* Retrieved from NVIDIA Developer Zone: https://docs.nvidia.com/nsight-systems/UserGuide/index.html

perf: Linux Profiling with Performance Counters. (2022, 12 14). Retrieved from Perf Wiki: https://perf.wiki.kernel.org/index.php/Main_Page

Appendix A

Sample Test Daemon Code

This sample code was used to test the power sensing daemon application. Its functionality can be summarized as a loop that executes squares of numbers and prints them at the end of the execution; before the loop is executed, the power sensing daemon is called and the pre application measurements are printed; after the main block of code is executed, the daemon is stopped and the measurements for the application runtime are printed; finally, the daemon is killed. The language written is C:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#includes "pipes.h"

int main(int argc, char *argv[])
{
// call the function to start power sensing
char* start = startPowerSensing();
// print stats from cache (pre app running)
printf("Start: %s\n", start);
fflush(stdout);
int two = 2;
// loop that prints squares of a number
while (two < 10000) {
two = two * two;
}
printf("number: %d\n", two);
char* final = stopPowerSensing();
// print stats from logfile (during app running)
printf("Final: %s\n", final);
fflush(stdout);
// kills the daemon
int isDead = killPowerSensing();
printf("Is dead: %s\n", isDead == 0 ? "yes, daemon is dead" : "no, daemon is alive");
fflush(stdout);
return 0;
}
```