NVIDIA Performance Testing for Emulation of the Grace CPU

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

By: Gong Fan, Muyun He, Shundong Li

Decemeber 31, 2021

Report Submitted to Professor Mark Claypool, Worcester Polytechnic Institute

Sponsored by NVIDIA Corporation

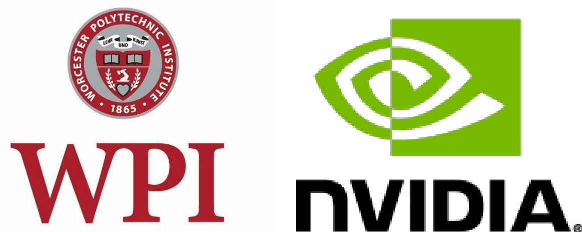

Table of Contents

Chapter 1: Introduction **1**

Chapter 2: Background **3**

GitLab 3
Amazon Web Service Graviton 4
Linux Perf 5
NUMA and Numactl 6
Control Groups 7
PostgreSQL Database 8
Tableau 9

Chapter 3: Methodology **11**

Requirement Definition 12
Prototype 12
Receive Feedback 13
Finalize Software 13

Chapter 4: Implementation **14**

Backend 14
Shell script 15
Utility Scripts 16
benchmarks.jsonc 16
benchmarkUtility.py 16
shellUtility.py 17
SpinCursor.py 17
Benchmark Scripts 17
Database 18
Visualization 22

Chapter 5: Expandability **32**

Python Script 32
Case 1: Add a sub-task 32
Case 2: Add a new benchmark suite 33
Visualization with Tableau 33

Chapter 6: Conclusion **34**

Chapter 7: Future Work **36**

Python Script 36
Database 37
Tableau Visualization 37

Chapter 8: References 38

Appendix A: Tableau Guide 40
Get Permission on Tableau 40
Add a New Benchmark Feature 40
Application (Benchmark) doesn’t Appear. 41
General Notes for Editing Worksheets 42

Appendix B: Our GitLab project code 43

Abstract

NVIDIA is developing a new CPU and needs to test and compare the performance of competitors' chips against their simulated CPU. However, testing is cumbersome and time-consuming with current test suites, especially on simulated hardware. Our project is to create a new tool that consolidates other test suites for easier testing for their simulated CPU. We developed a series of Python scripts, defined a database solution for data storage, and created notebooks on Tableau for data visualization. The tester uses a Python script (shell) to run all tests in one place and upload the results to a database, and the developer views and compares the results in Tableau. For ease of use, testers can mount this software on any Linux machine, which simplifies multiple running benchmarks at once and compares results.

1. Introduction

NVIDIA plans to design a new CPU and evaluate its performance against competitors before making the silicon. Since it is time-consuming and expensive to manufacture a chip, NVIDIA wants to compare their competitors' chip's performance and test their CPU design without making the silicon. To do so, NVIDIA's CPU needs to be simulated on other machines so tests can be simulated.

Many test suites and benchmarks are available to stress-test the hardware to evaluate performance, even on a simulated chip. However, current test suites need to be installed from different places and run separately with different commands. Furthermore, all tests have to be run on different machines to compare the results. Since the new CPU will be simulated and much slower than a real chip, it is time-consuming and complicated to run tests with current tools. The challenge is consolidating other tests and creating a pipeline by automating most processes, such as installing and uploading data.

A simple solution is to create a bash script that installs and runs all the tests one by one, then manually records the data into Excel. This solution might be sufficient, but it also has many drawbacks. A bash script is hard to customize and automate based on NVIDIA's workflow. Also, NVIDIA might want to add new tests. Bash is designed to execute commands and is much harder to implement logic, making it hard to understand and maintain bash script. For instance, output formatting code could be incredibly time-consuming and confusing to write or understand in a bash script compared to a high-level programming language. Furthermore, it is hard to automate a process based on different situations with a bash script. On the other

hand, a high-level programming language, such as Python, is designed to be easier to read and simpler to implement than a bash script. If NVIDIA wants to add new tests to the case, an engineer can easily navigate to the correct position and make changes.

We created a new tool with the tech stack of Python shell, Postgres database, and Tableau visualization to address this problem. The test suite follows a linear workflow (Figure 1), intended to be customizable and easy to use. The user can run benchmarks with the Python shell and store the data in local storage. The script automatically uploads the test results to NVIDIA's database. Then the user can visualize the results in Tableau. Most of the work has been automated, such as running multiple tests and uploading results to the database; the user only needs to specify which tests they want to run. Users can also quickly create new tests with different specifications. The tool is concise, such that it can be incorporated into a Linux image to be loaded and run on a simulated CPU from NVIDIA.

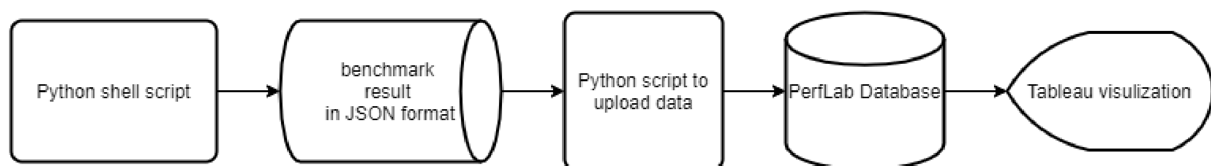

Figure 1. Workflow: Run tests, upload data, visualize data

This report provides details on the project and our process. Chapter 2 describes the background and related technologies on software and concepts used. Chapter 3 provides our approach to designing and building our new test suite. Chapter 4 gives a detailed explanation of our implementation and decision processes. Chapter 5 provides expandability for maintainers to develop new benchmarks. Chapter 6 summarizes our conclusions. Finally, Chapter 7 lists possible future work.

2. Background

This chapter describes the background and related technologies on softwares and concepts we used in our project. Each section is dedicated to a different technology; inside, we give an introduction to each technology, and how we utilized it inside our project.

2.1. GitLab

GitLab is a fully integrated software development platform that enables users to collaborate and produce software by using public or private web-based git repositories in a transparent, effective, and cohesive way on the same platform [1].

Developers can install GitLab on multiple operating systems, including Linux, macOS, Windows, and FreeBSD. Once GitLab is installed on Linux, developers could create a project with a remote repository and a local copy. Standard command lines like *git fork*, *git clone*, *git pull*, *git push*, and *git checkout* allow users to modify, update, and share the code. Git fork creates a copy of the project in the user's namespace on GitLab for modifying project files, settings, and permissions. Git clone creates a copy of the project on the user's local computer. Git pull fetches the content from the remote repository and updates users' local repository to match the latest content. Git push uploads local repository content to a remote repository so that other users can use git pull to update their code or repository. Git branch allows users to copy files in the repository and work on an independent line of development. Users usually use the git branch to work on new features or fix bugs and later merge the individual branch

with the main branch. Overall, GitLab's command lines provide an efficient working function for software development groups.

Our repository is located on NVIDIA's private GitLab space. Throughout this project, git is heavily utilized to synchronize code between teammates and our sponsors.

2.2. Amazon Web Service Graviton

Amazon Web Service (AWS) is a cloud service platform provided by Amazon.com, Inc. The cloud-computing platform allows users to rent virtual computers to run their computer applications [2]. AWS Graviton is an Elastic Compute Cloud (EC2) instance that uses ARM architecture cores; Compared to x86 instances; Graviton has a significant advantage on price. Also, AWS Graviton processes have extensive software support from other AWS services. For example, Cloudwatch can collect logs and metrics from each instance when pairing with Graviton; also Auto Scaling can use the metrics to create a new instance based on traffic and utilization of current infrastructure. Service (ECS) can pull docker containers from container registries to run on any EC2 instances.

In our project, Graviton simply serves as a benchmarking machine. We runned some benchmarking scripts with the tool we built.

2.3. Linux Perf

Linux Perf is a performance analysis tool for Linux that instruments CPU performance counter events [3]. It is used for statistical profiling of the entire system. Perf can run benchmarks on both hardware and software for performance data.

Perf's commands, including stat, record, report, top, and bench, are the most useful in our projects. We use perf stat to gather performance counter statistics. Perf record runs the requested command and records its profile into perf data. Pref report reads specific perf data and displays the profile. Perf top is a system profiling tool that generates and shows a live performance counter profile. Perf bench is a general framework for benchmark suites that run different kernel microbenchmarks. We use perf bench to capture kernel microbenchmarks. For example, *perf bench cpu-clock* and *perf top* first runs a CPU-clock benchmark and then display a live performance counter profile. We can also capture CPU performance counter events during external benchmark execution. For instance, *perf stat Phoronix-test-suite benchmark systemd-boot-total* first runs a test that uses system-analyze to report the entire boot time (the time it takes for a device to be ready to operate after the power has been turned on) by Phoronix test suite in Figure 2. Then it captures CPUperformance counter events by *perf stat* in Figure 3.

```
Systemd Total Boot Time:
    pts/systemd-boot-total-1.0.6 [Test: Total]
    Test 1 of 1
    Estimated Trial Run Count:    1
    Estimated Time To Completion: 2 Minutes [10:58 EDT]
        Started Run 1 @ 10:57:00
        The test run ended quickly.

    Test: Total:
        5859

    Average: 5859 ms
```

Figure 2. Systemd-boot-total Test Result

```
Performance counter stats for 'phoronix-test-suite benchmark systemd-boot-total':

         4,967.61 msec task-clock                #    0.111 CPUs utilized
            8,541      context-switches          #    0.002 M/sec
              590      cpu-migrations            #    0.119 K/sec
          521,498      page-faults               #    0.105 M/sec
  <not supported>      cycles
  <not supported>      instructions
  <not supported>      branches
  <not supported>      branch-misses

     44.937478911 seconds time elapsed

      3.439836000 seconds user
      1.562797000 seconds sys
```

Figure 3. Perf Performance Counter Statistics for Systemd-boot-total Test

Linux perf allows users to capture CPU performance counter events for low-level performance analysis or tuning.

2.4. NUMA and Numactl

The scalability of the Grace CPU, relies heavily on non-uniform memory access (NUMA) [4]. Uniform memory access (UMA) systems usually have only one CPU and one memory controller, limiting the total memory pool size; since NUMA nodes (one CPU package, one

memory controller, and the controller's designated RAM pools) have interconnections between each node, NUMA provides the potential to expand.

Although NUMA provides potential for increased server performance, it has some drawbacks. Mixing the RAM pool among multiple NUMA nodes can increase RAM accessing latency. This arises from the data taking more time to travel through the interconnections when accessed across nodes. Therefore, restricting and planning the memory access policy is crucial in NUMA platforms.

Numactl is a Linux package that controls NUMA policy for processes or shared memory [5]. We looked into several flags that were useful for our project such as '–physcpubind', which only escute processes on certain specified CPU cores.

NUMA and numactl, provide a more thorough understanding of the system that we run our benchmarks on, and give us more control as to how we want to customize some test suites.

2.5. Control Groups

Control Group (cgroup) is a mechanism in the Linux kernel to organize processes hierarchically and distribute system resources along the hierarchy [6]. It allows users to determine how much system resources are allocated to a given process. The Control group consists of a core, which establishes and maintains the hierarchy, and controllers, which distribute specific types of resources along the hierarchy.

Command lines like:

- *blkio* allows users to limit per cgroup block IO performance.
- *cpu* is the percentage of CPU for a particular application.
- *cpuacct* provides per cgroup usage accounting.
- *cpusets* provides a Linux Kernel mechanism to constrain which CPUs and memory nodes are used by a process or set of processes.
- *Devices* build a device access whitelist or blacklist with each cgroup.
- *freezer* freezes the tasks if they are not scheduled.
- *memory* tracks and limits userspace memory and kernel memory.
- *net_cls* tags packets to specific tasks, and traffic controllers can use those tags to assign priorities.

Users can use cgroup to organize different GPU processes hierarchically and distribute system resources along the hierarchy. Cgroups are used in many benchmarking tools to control the resources used in a specific flag, mainly through passing in different flags. Although it is not directly used in our project, we need to record the corresponding flags for our sponsors to understand better and analyze the data gathered by our tool.

2.6. PostgreSQL Database

Databases are used to store and organize structured information or data. Databases have many different paradigms such as key-value, wide column, and document. Those paradigms have different benefits; for example, key-value paradigms can be extremely fast to access data and

the document paradigms are schema-less. In this project, we used a relational database for the accessibility and consistency provided by a Structured Query Language (SQL).

Postgres is an open-source relational database management system and SQL compliant [7]. The system is ACID (atomicity, consistency, isolation, and durability) compliant, which means all transactions in the database guarantee data validity even through network or hardware failure. SQL allows the Python scripts to generate the insertion queries automatically and uploads data to the database after each benchmark run. Visualization steps with structured data can also be automated. Potential drawbacks for this database is that it requires schemas and is difficult to scale

We used a PostgreSQL database to store our data generated by our benchmarking suite. The data can be effortlessly accessed by Tableau, our visualization tool (introduced in section 2.7), to generate graphics for NVIDIA to analyze and compare results.

2.7. Tableau

Tableau helps to see and understand data [8]. It is also easy to create a dashboard on Tableau. The dashboard is similar to the Excel Pivot table but much more powerful. Users can grab data realtime from the database, and Tableau gives various options on how the data is displayed.

We used Tableau in our workflow and served as the users' front end. The user can run benchmarks on multiple machines; the results are updated in Tableau for comparison and analysis within seconds.

3. Methodology

This chapter describes our methodology. Our team switched from a simplistic development approach to the Rapid Application Development approach. We will also break down how this approach is performed and why it is more effective than our initial approach.

At the planning stage of our project, we were given a list of technologies to explore as background. Starting from those backgrounds with a known development timeline, we began the development serially by first creating a Python script backend, then the database, and finally the visualization.

After discussing with Professor Claypool and our mentor roughly a week into the project, we decided the three sections depend on each other and need input from developers and users to make changes correspondingly. Therefore, we switched to Rapid Application Development (RAD). RAD mainly breaks down into four parts: requirement definition, prototype, feedback reception, and software finalization. We describe these four parts in the following paragraphs.

3.1. Requirement Definition

Our requirement was to provide a robust and expandable workflow that contains the functionalities of the Python script backend, the database, and the visualization. Our project does not only need to include the Python script backend, the database, and the visualization, we also should provide a robust and expandable workflow that contains these three functionalities. Robust means that the failure rate of the workflow should be low. Furthermore, even if some part fails, it should be easy to find the error source and fix it correspondingly. Expandable means that adding those benchmarks to the workflow should be relatively easy and hassle-free.

3.2. Prototype

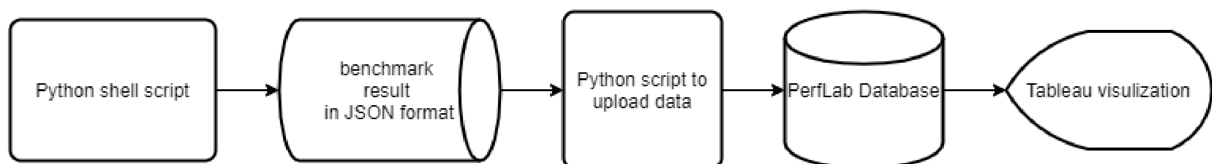

Figure 4. The workflow for our project.

As shown in the workflow above (Figure 4), our project comprises five relatively independent parts, which can be developed, tested, and demonstrated in isolation. The five parts include two storage systems: JSON files on local disk and final storage in PerfLab's PostgreSQL database. The remaining three parts consist of the Python shell to handle benchmark execution and result parsing to JSON, the Python script to parse the JSON files and upload them to the database, and the Tableau visualization. We can quickly develop

working prototypes that our mentors can review by breaking down the project into these five portions.

3.3. Receive Feedback

With the help of our mentor, we have reached out to other teams who had experience with similar projects but for different hardware or technology stacks. Therefore, whenever we had a working demo, we got feedback from our mentor and specialists, which sped up the development process aided by advice on implementing our tasks.

3.4. Finalize Software

After refining prototypes, adding/removing features, and debugging, our last step is connecting the python script, the database, and the visualization to form a functional workflow. We also had to document the maintenance and expansion of our project.

4. Implementation

This chapter describes our implementation test tool: Python script backend, Perflab database, and Tableau visualization frontend.

4.1. Backend

With the idea of comparing hardware performance in mind, during the development process, we generated test results from different test suites, different hardware, and different kernel patches. We store our initial benchmarking results into JSON files since some of the benchmarking tools we use to yield results in a JSON file or provide functions to convert the result into a JSON file. Figure 5 is an example of how Phoronix can convert its result into JSON file format.

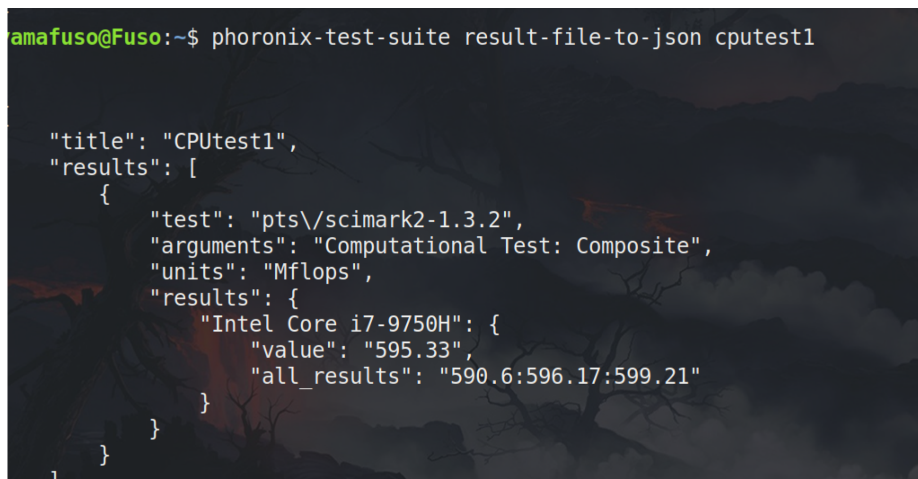

Figure 5. Two Phoronix Test Results Parse as Json Files

In our project, the Python script acts as the backend of the workflow: executing benchmarks, formating results, saving them to local disk, and uploading them to the database. The Python script can format those default JSON results into our designed schema. Furthermore, if the benchmark does not provide any built-in function to parse results into a file, the Python script can also directly capture the standard output in the Linux terminal and parse the results. We

designed our script to run the benchmark, capture and store the results locally before storing them in the database.

Shell script

There are two important functions in the **benchmark_shell.py** file, which fully make up the shell users invoke and interact with if they want to execute the benchmark using our workflow.

- auto_shell(step_flag,argv)

Invoked by the main() function, this is the structure for the shell's workflow to go step by step (gather info, execute benchmark, upload, etc.). We previously had two nearly identical shells, one auto and one interactive. We merged those two functions into one at the finalization stage of our project to reduce repetitive code and make maintenance easier.

- configure_benchmark(argv)

This is the key function to match the user input benchmarks to the default values recorded in the benchmarks.jsonc file.

Utility Scripts

A few utility scripts and files are essential to the backend. They act as helper functions for either the shell or the benchmark scripts. They are located under the **utilities** folder.

benchmarks.jsonc

This file saves the defaults for each benchmark and sub-benchmarks. One example is shown below in Figure 6:

```
// The following spawn micro-benchmark exercises process generation
// (using the POSIX spawn API) and process deletion, a basic functionality of operating systems.
{
    "benchmark":"stress-ng",
    "specification":["spawn","cpus"],
    "flags":["--spawn","0", "-t", "30", "--metrics-brief"]
},
```

Figure 6. An example of one benchmark preset with comment

Each JSON object should include **benchmark**, **specification**, and **flags** as the fields. The **benchmark** is the application itself, the **specification** marks the keywords for the sub-benchmarks, and the **flags** are the default arguments entered in the Linux terminal. **Benchmarks** and **specifications** are used when matching the user input string with those defaults, and **specifications** and **flags** could be left empty if not needed. Also, since this file is JSONC, not JSON, the developer can add comments to document the defaults if needed.

benchmarkUtility.py

This file contains several helper functions the shell uses in the execution process.

- benchmark_call(benchmark)

This function takes a JSON object parsed from benchmarks.jsonc and adds additional information like directory or timeout setting. At the end of the function, it invokes the

result_add_specification(path, benchmark) method to add the specification information to the result JSON object and file

- script_call(...) and command_call(...)

These functions are invoked by the benchmark_call() functions above and take in the parsed arguments and configurations, import the corresponding benchmark script, and invoke the function to generate the result JSON files. The main difference between script_call and command_call is that script_call should invoke those benchmarks in a particular directory, preferably at the same level as this repository. In contrast, command_call is suitable for those installed Linux packages and can be invoked regardless of the directory in which the command is called.

shellUtility.py

This file contains the function exec_in_terminal. That takes in the argument and executes them in the Linux terminal.

SpinCursor.py

This file contains the class that is used to create the execution animation.

Benchmark Scripts

The benchmark scripts are not executed directly per benchmark; instead, the shell matches the user input with our supported benchmarks using configure_benchmark(argv) mentioned above. Benchmark execution is only one portion of our workflow; one of the most important goals is to gather the data from tests and save them to a uniform schema. The benchmark scripts are dynamically imported into the shell to perform precisely this task.

Each benchmark script file must have a function toJSON(p_stdout,p_stderr,argv) with this exact signature dynamically imported into the shell and the same name with the benchmark field listed in the benchmarks.jsonc file. Our project documentation marked these requirements to make adding new benchmarks and maintaining previous benchmarks easier.

4.2. Database

While a document-based database such as MongoDB seemed to be an obvious choice for our use cases, it requires maintenance of accounts such as MongoDB or AWS outside NVIDIA's network.

Instead, our project utilizes the existing infrastructure that PerfLab owns. A new Postgres database is created on Perflab's database systems. Their database undergoes daily backup to protect the data space and enables the use of Tableau for visualization (Section 4.3).

After the database was online and the credentials sent to the team, we defined a schema to represent the data. Two options for the database schema are given at first:

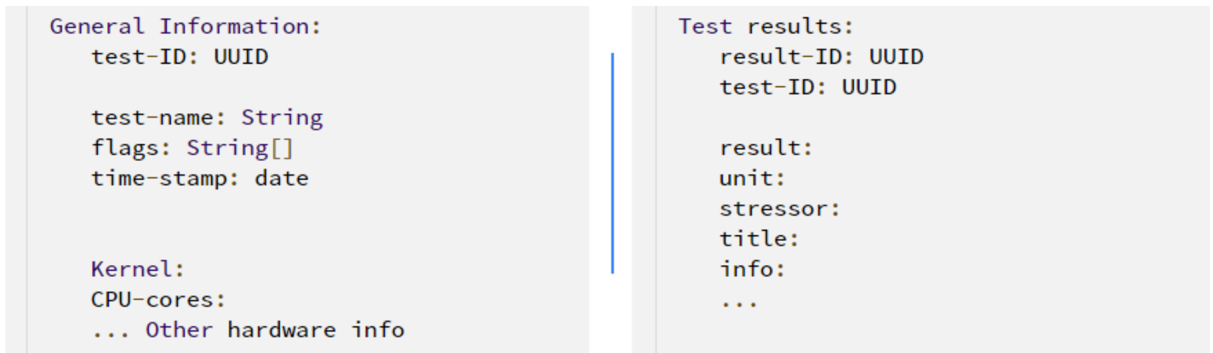

Figure 7. Right: Option one for database schema;

Left: Option twofor database schema

1. As shown in figure 7, one table stores general test information and another table is initiated with all possible result titles as columns.

   This approach results in two tables, but the second table has extraneous fields that most tests will not use.

2. The other option is a table to store general test information, and each test has its specific table. The different tables for each test are created when running a new test for the first time.

   This approach creates more tables for each test, reducing the database storage used but increasing software complexity. It also requires more maintenance and versioning to ensure there are no conflicts.

The third option chosen based on feedback from the Perflabs team is for output files from the test to follow a strict format:

```
"results": [
    {
        -- METADATA
        "recordid": string,
        "timestamp": string,            -
        "uploader": string,

        -- Data from tests
        "application": string,
        "application_version": string,
        "flags": string,
        "specification" string,
        "description": string,
        "score": float,
        "metric": string,
        "bigger_is_better": string,

        -- Hardware data
        "cpu_model_name": string,
        "cpu_family": string,
        "cpu_model" : integer,
        "cpu_stepping" : interger,
        "cpu_virtualization_type": string,
        "cpu_address_sizes": string,
        "cpu_frequency": float,
        "cpu_l1d_kb": float,
        "cpu_l1i_kb": float,
        "cpu_l2_mb": float,
        "cpu_l3_mb": float,
        "BogoMIPS": float,
        "number_of_cpu_sockets": string,
        "cpu_cores_per_socket": string,
        "threads_per_core": string,
        "number_of_cpu_threads": string,

        "os_architecture": string,
        "os_name": string,
        "os_version": string,
        "kernel_version": string,

        "system_memory_gb": interger,
        "system_memory_speed_mhz": interger,

        "gpu_model": string
        "gpu_video_memory_mb": integer,
        "number_of_gpus": integer
    },
     ...
 ]
```

Figure 8. Format the benchmark results need to follow

The format as shown in figure 8 consists of an object named results, and inside results, there is an array of record objects.

The record object consists of three parts:

- Metadata
- Data from tests
- Hardware data

Figure 9 is a brief representation of how the data from JSON is stored in the database.

A single run of a test

Data of the run

recordid	application	score	metric	cpu_model_name
stress-ng-12.03.21-05:46:25-a8b18ec4-c32c-4702-b191-bd2af0a54b1d	stress-ng	59524250.00	bogo ops	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
stress-ng-12.03.21-05:46:25-a8b18ec4-c32c-4702-b191-bd2af0a54b1d	stress-ng	30.00	real time, secs	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
stress-ng-12.03.21-05:46:25-a8b18ec4-c32c-4702-b191-bd2af0a54b1d	stress-ng	84.78	usr time, secs	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
stress-ng-12.03.21-05:46:25-a8b18ec4-c32c-4702-b191-bd2af0a54b1d	stress-ng	322.16	sys time, secs	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
stress-ng-12.03.21-05:46:25-a8b18ec4-c32c-4702-b191-bd2af0a54b1d	stress-ng	1984117.78	bogo ops/s, real time	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
stress-ng-12.03.21-05:46:25-a8b18ec4-c32c-4702-b191-bd2af0a54b1d	stress-ng	146272.79	bogo ops/s, usr+sys time	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
stress-ng-12.03.21-05:54:08-d3926241-af19-405b-ad3e-e6e4f8cd0068	stress-ng	30.00	real time, secs	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
stress-ng-12.03.21-05:54:08-d3926241-af19-405b-ad3e-e6e4f8cd0068	stress-ng	83.09	usr time, secs	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
stress-ng-12.03.21-05:54:08-d3926241-af19-405b-ad3e-e6e4f8cd0068	stress-ng	331.16	sys time, secs	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
stress-ng-12.03.21-05:54:08-d3926241-af19-405b-ad3e-e6e4f8cd0068	stress-ng	1943823.16	bogo ops/s, real time	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
stress-ng-12.03.21-05:54:08-d3926241-af19-405b-ad3e-e6e4f8cd0068	stress-ng	140773.25	bogo ops/s, usr+sys time	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
stress-ng-12.03.21-06:00:27-9c4f87ef-768e-462b-afa7-506c7c4e3b93	stress-ng	58597483.00	bogo ops	fake cpu
stress-ng-12.03.21-06:00:27-9c4f87ef-768e-462b-afa7-506c7c4e3b93	stress-ng	30.00	real time, secs	fake cpu
stress-ng-12.03.21-06:00:27-9c4f87ef-768e-462b-afa7-506c7c4e3b93	stress-ng	83.52	usr time, secs	fake cpu
stress-ng-12.03.21-06:00:27-9c4f87ef-768e-462b-afa7-506c7c4e3b93	stress-ng	329.61	sys time, secs	fake cpu
stress-ng-12.03.21-06:00:27-9c4f87ef-768e-462b-afa7-506c7c4e3b93	stress-ng	1953231.13	bogo ops/s, real time	fake cpu
stress-ng-12.03.21-06:00:27-9c4f87ef-768e-462b-afa7-506c7c4e3b93	stress-ng	141837.88	bogo ops/s, usr+sys time	fake cpu
ipc_benchmark-12.03.21-06:24:56-a5c4e36b-55ce-4796-bf13-4b6b12f393b2		128.00	data_byte	fake cpu
ipc_benchmark-12.03.21-06:24:56-a5c4e36b-55ce-4796-bf13-4b6b12f393b2	ipc_benchmark	101.00	result1_MB_s	fake cpu
ipc_benchmark-12.03.21-06:24:56-a5c4e36b-55ce-4796-bf13-4b6b12f393b2		825553.00	result2_msg_s	fake cpu
ipc_benchmark-12.03.21-06:24:56-a5c4e36b-55ce-4796-bf13-4b6b12f393b2		256.00	data_byte	fake cpu
ipc_benchmark-12.03.21-06:24:56-a5c4e36b-55ce-4796-bf13-4b6b12f393b2	ipc_benchmark	196.00	result1_MB_s	fake cpu
ipc_benchmark-12.03.21-06:24:56-a5c4e36b-55ce-4796-bf13-4b6b12f393b2		803599.00	result2_msg_s	fake cpu
ipc_benchmark-12.03.21-06:24:56-a5c4e36b-55ce-4796-bf13-4b6b12f393b2		512.00	data_byte	fake cpu
ipc_benchmark-12.03.21-06:24:56-a5c4e36b-55ce-4796-bf13-4b6b12f393b2	ipc_benchmark	505.00	result1_MB_s	fake cpu
ipc_benchmark-12.03.21-06:24:56-a5c4e36b-55ce-4796-bf13-4b6b12f393b2		1033691.00	result2_msg_s	fake cpu

Benchmark Data

Figure 9. How data is represented in database

Each object in the results array is a row in the database.

The metadata is used to identify which test run the record is for and other information such as when the user ran the test.

The actual data from the test suites are stored in the next section - data from tests.

The data is represented in two columns, namely: metric and score, as seen in the blue box.

The orange box represents a run of a test. Different metrics are stored in different columns with their respective score.

There are also two databases: production and staging databases. The staging database is used for testing newly added scripts such that if something goes wrong, undesired data does not pollute the production database. The user can use the staging database bypassing the ‘--staging’ flag to the testing shell.

4.3. Visualization

The visualization aims to display benchmarks’ performance in a user-friendly format. For implementation, we considered three options: a custom web app, Google Data Studio [9], and Tableau [8].

A custom web app allows us to design unique functions based on our needs. However, it would need extensive development and maintenance effort. On the other hand, Google Data Studio has already developed functions for users to use. It supports an easy connection to data sources, and it does not require much maintenance. Google Data Studio is a better choic than a custom web app.

Property / Platform	Google Data Studio	Tableau
Price	Free, but to add some data sources will be required Supermetrics: https://supermetrics.com/pricing/data-studio ($19/mo)	$70 per user a month: https://www.tableau.com/pricing/teams-orgs
Trial	Unnecessary	14 days
Users	Unlimited	1
Dashboards	Unlimited	Unlimited
Data refresh	12 Hours	Can be scheduled for any time
Application speed	Loads data from free cloud storage, so takes longer to load on complex dashboards	Displays all received info right away, making the heaviest dashboards load fast
Implementation	Cloud	Program for any desktop + changes in the Cloud
Other	Low loading speed if you use blended data/high quantity of data/a lot of calculated metrics Doesn't have all required connectors – will be required to use some third-party tools	Handles large volumes of data with fast performance Mobile-friendly A broad range of databases and servers, and native connectors

Figure 10. Google Data Studio Vs. Tableau [10]

Figure 10 shows the differences between Google Data Studio and Tableau. Tableau can have a data refresh, whereas Google Data Studio requires twelve hours after the previous data refresh. Tableau displays and loads dashboards faster than Google Data Studio. It also works well with perf Lab's Postgres database. Based on the comparison, we choose to use Tableau for a visualization based on its primary data refresh function, fast loading and high displaying performance, low maintenance effort, and compatibility with live perf Lab databases.

Users of the visualization tool would face three situations. First, users may want to view a single benchmark performance. Second, users may want to check the same benchmark performance on various machines to compare their CPU TH500. Third, users may want to visualize the same benchmark performance that runs on the same machine over time to

analyze its development improvement. Three worksheets, Benchmark Performance, Data on Benchmark Performance, and Machine Information Raw Data, are designed with Tableau to support those use-cases. A design theory, Zero One Infinity Rule, avoiding arbitrary limits on the number of instances of a particular type of data or structure [11], provides no limits for users to select their preferred benchmark, CPU type, and time that a benchmark is running.

First, a worksheet Benchmark Performance with filters shows bar graphs for all benchmarks' performance. The worksheet allows users to use filters to narrow down to a specific benchmark with details including application, application version, CPU model name, description, metric, and score:

- Application refers to a type of benchmark.
- CPU model name refers to the CPU model of a machine that runs a benchmark
- Description refers to different tests of the same benchmark. For example, the Glibc benchmark has tested ffsll, ffs, pthread_once, tanh, and sqrt.
- Metric refers to specific test columns in each description. For example, ffs of Glibc benchmarks have test columns like duration, iterations, min, max, and mean.
- Score refers to a specific value for each metric.

A general view of visualizations on all benchmark performances is shown in Figure 11. When users decide to view a specific benchmark performance, they could use filters on the right side to narrow down the application (benchmark type), CPU model name, uploader, score, timestamp, flags, description, and metric. The column in this figure shows benchmarking applications, the CPU that ran the benchmark, and the scores from each metric. A single row represents the data from a single benchmark run. We can compare any two rows from the same application to analyze the performance difference of the benchmark on different hardware. Furthermore, the rows can also be filtered by flags (specifications when running the benchmark that controls the application's behavior, such as using a single core or all CPU cores) to analyze the data more accurately.

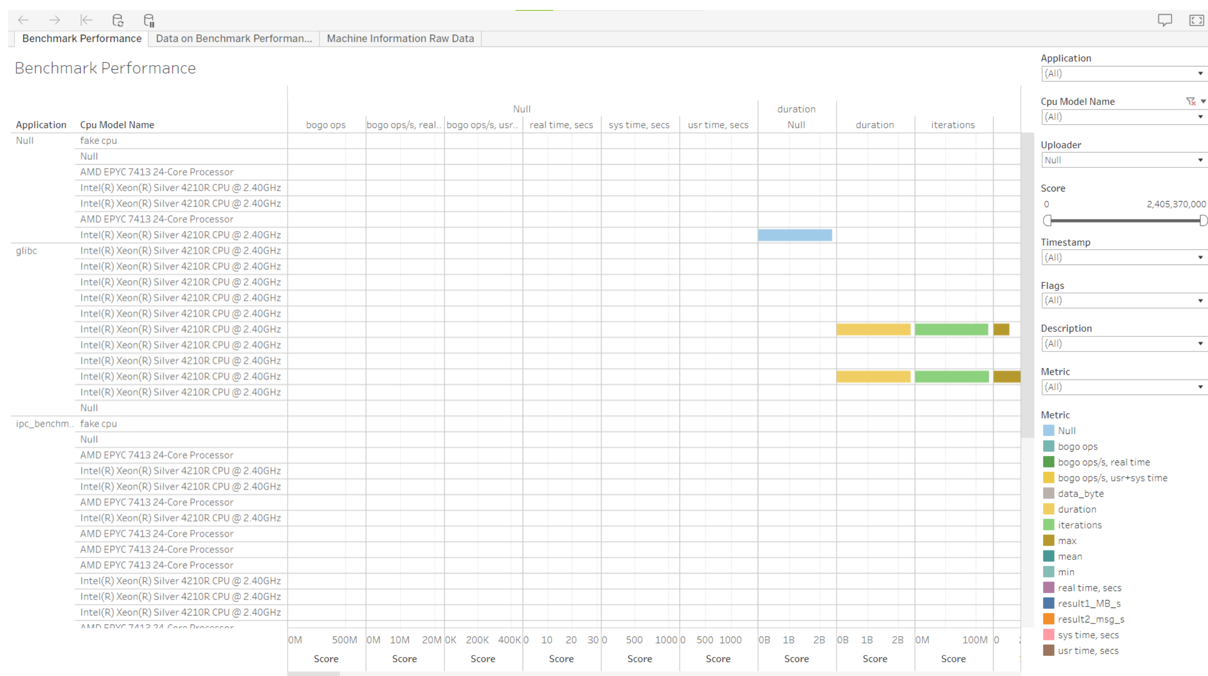

Figure 11. Performance of All Benchmarks

For example, a user can filter data to visualize a Glibc benchmark performance for a sinh test that runs on a machine with Intel Xeon Silver 4201R CPU@ 2.40 GHz at 09:13 AM on Dec 16, 2021, depicted in Figure 12. On this particular CPU, the Glibc benchmark ran with an interaction score of about 84 million and a mean score of about 30.

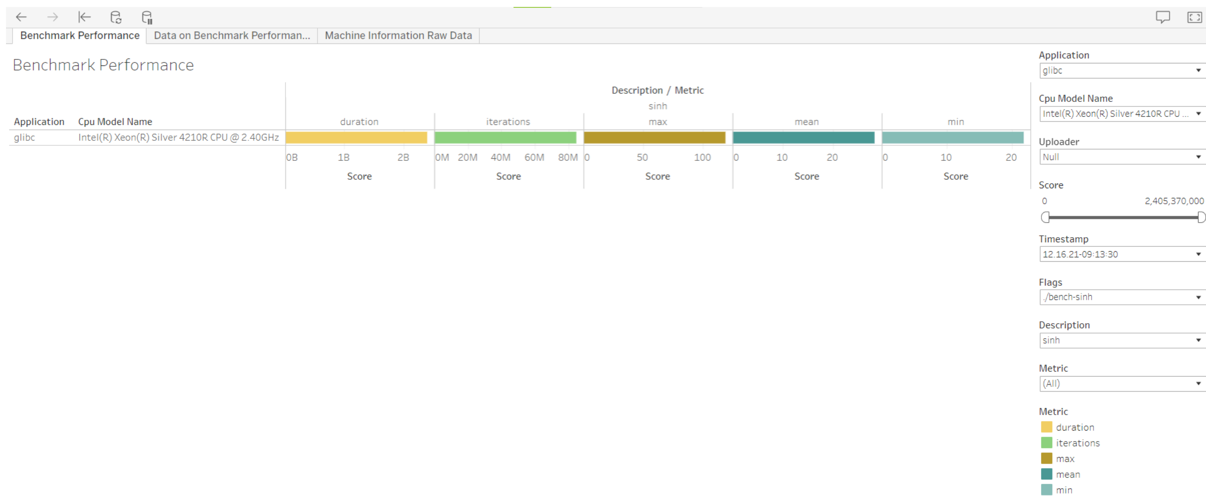

Figure 12. A Glibc Benchmark Performance for sinh test with Intel Xeon Silver 4201R CPU@ 2.40 GHz at 09:13 AM on Dec 16, 2021

Each metric has a unique y-axis. Moving the mouse to a bar graph displays detailed data. Users can switch to the Data on Benchmark Performance worksheet shown in Figure 13. This worksheet shows numeric data values for benchmark performance, including application, CPU model name, description, metric, and score.

Figure 13. Data of a Glibc Benchmark Performance for sinh test with Intel Xeon Silver 4201R CPU@ 2.40 GHz at 09:13 AM on Dec 16, 2021

If users want to see the machine information that runs this Glibc benchmark, they could switch to Machine Information Raw Data shown in Figure 14. This worksheet Machine Information Raw Data shows machine information of a specific machine that runs the benchmarks. If users choose not to capture machine information or there is no machine information to be captured, the display of the column is null.

Benchmark Performance | Data on Benchmark Performan... | Machine Information Raw Data

Machine Information Raw Data

Application	Gpu Model	Number Of Gpus	Gpu Video Memory Mb	Os Architecture	Os Name	Os Version	Kernel Version	System Memory Gb	System Memory Speed Mhz	Cpu Family	Cpu Model	Cpu Stepping
glibc	Null	Null	Null	x86_64	Null	Null	Null	Null	Null	6	85	7

Cpu Model Name: Intel(R) Xeon(R) Silver 4210R CPU ...
Application: glibc
Uploader: Null
Timestamp: 12.16.21-09:13:30
Flags: ./bench-sinh

Figure 14. 1. Machine Information of a Glibc Benchmark Performance for sinh test with Intel Xeon Silver 4201R CPU@ 2.40 GHz at 09:13 AM on Dec 16, 2021

Cpu Address Sizes	Cpu Virtualization Type	Cpu L1D Kb	Cpu L1I Kb	Cpu L2 Mb	Cpu L3 Mb	Bogomips	Number Of Cpu Threads	Number Of Cpu Sockets	Cpu Cores Per Socket	Threads Per Core
46 bits physical, 48 bits virtual	Null	320	320	10	13.8	4,800	20	1	10	2

Figure 14. 2. Machine Information of a Glibc Benchmark Performance for sinh test with Intel Xeon Silver 4201R CPU@ 2.40 GHz at 09:13 AM on Dec 16, 2021

Similarly, users could use filters to visualize the same benchmark performance on multiple machines and dates, as shown in Figure 15. Each row represents a benchmarking run with the specific application, a specific flag, and a specific CPU. The user can compare the performance difference of this particular configured benchmark on different CPUs. For instance, we can see that AMD EPYC 7413 CPU is outperformed by Intel Xeon Silver 4210R CPU from Figure 15; it takes the AMD CPU much longer than a custom web app at around 400 seconds in system time to run the benchmark compared to around 300 seconds from Intel CPU.

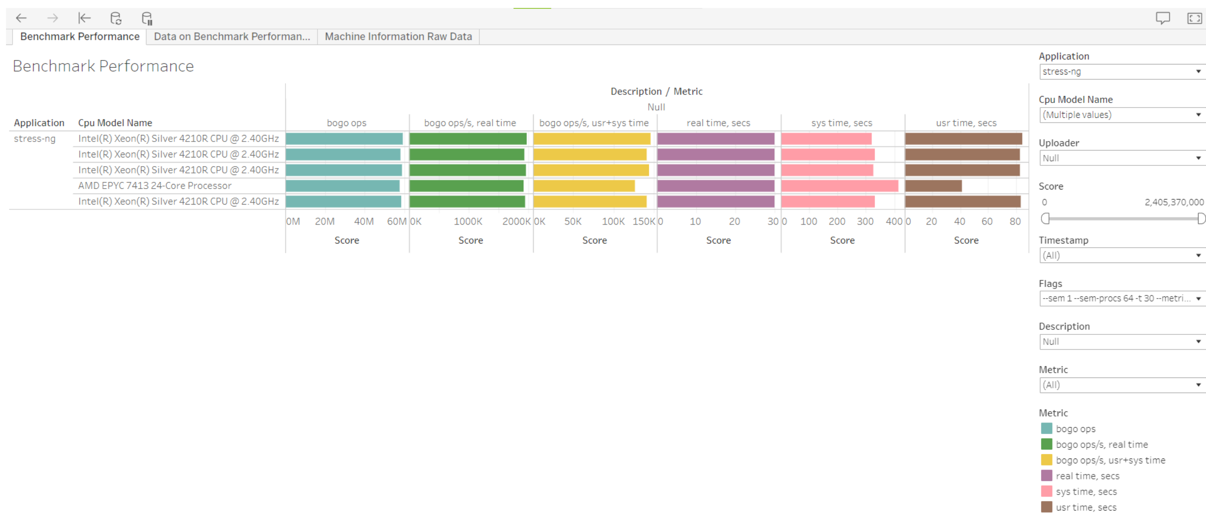

Figure 15. stress-ng Benchmark Performance with flag -sem1. Comparison between Intel Xeon Silver 4201R CPU@ 2.40 GHz and AMD EPYC 7413 24-Core Processor

For future analysis and data modification, Tableau supports the interface allows users to download worksheets as an image, an excel file, a CSV file, a text file with raw data, a pdf, a PowerPoint, and a Tableau workbook.

5. Expandability

The following chapter describes the expandability of the project. It demonstrates how a developer can maintain the Python script and visualization on Tableau if a new benchmark is added.

5.1. Python Script

There are two cases when adding a benchmark to the workflow: adding subtasks or adding a complete benchmark. Adding a sub-benchmark does not need to be configured within the workflow; for most cases, adding a sub-benchmark requires editing the jsonc file and changing the formatting function. Adding a new benchmark needs more effort in result parsing and minor additions in the framework.

Case 1: Add a sub-task

Adding a sub-task means that the benchmark is already in the workflow; the developer just needs to add a sub-benchmark (the benchmark folder has the corresponding file). This is relatively easy to implement; the developer can just:

1. Add the new sub-benchmark and required information into the benchmarks.jsonc file.
2. Initialize the shell and select the sub-benchmark to see if everything works.
3. There may likely need slight alternations in capturing the results, so small changes are also needed for the file under the benchmark folder.

Case 2: Add a new benchmark suite

In this case, the developer needs to make a more significant effort:

1. Document the new defaults into the benchmarks.jsonc file.
2. Add the corresponding directory and placeholder file in the Output folder
3. If the dependency auto checking feature is done, update the initialization script
4. Update the benchmark_call(benchmark) function with the arguments and paths for the benchmark execution.
5. Capture the standard output for the benchmark and decide which part of the result goes to the JSON File on Internal Storage
6. Create the corresponding file in the benchmark folder with the conventions listed in Benchmark Scripts.
7. Do test runs by starting the shell and seeing if there are remaining bugs.

5.2. Visualization with Tableau

The visualization interface built with Tableau is designed to be user-friendly. In general, the visualization with Tableau has three worksheets: Benchmark Performance, Data on Benchmark Performance, and Machine Information Raw Data. Appendix A includes a detailed guide on how users can add a new benchmark feature.

6. Conclusion

NVIDIA plans to design a new CPU and evaluate its performance against competitors before making the silicon. NVIDIA also wants to test their competitors' chip's performance without making the silicon. Fortunately, many test suites and benchmarks are available to stress-test the hardware to evaluate performance, even on a simulated chip.

However, installing, running, and collecting data manually from all test suites on multiple machines is cumbersome and time-consuming. Furthermore, it takes about ten seconds for the simulated chip from NVIDIA to run one second of system time. Running commands one by one to install and run all test suites is impractical; it can easily take days to do so. Thus an automated workflow is needed to simplify testing and data collection on different machines and emulated CPUs.

To address this problem, we create a new tool consisting of a Python shell, database, and Tableau to address this problem. Based on background information and technology, we used the rapid application development approach to design and implement our solution; We carefully made each decision after evaluating the pros and cons of multiple options; communicating with NVIDIA frequently to ensure the tool fits their desired workflow and use-cases.

The tester can use the tool to download all needed test suites. After downloading is complete, the entire tool can be incorporated into a Linux image to be loaded on the simulated CPU from NVIDIA, significantly reducing the time needed to start running benchmarks. The

utilities provided by the new tool save time for NVIDIA to install and run the test suites and simplify the work to compare the collected data. Data collected from different machines are brought to a centralized database. Tableau, our visualization tool, can grab the data in real-time to feed to the predefined dashboards.

There are also many low-level details in this new tool. The user can customize data flow behaviors in the python shell. For instance, the user can upload data to a staging database instead of the production database. This function can be helpful when testing out new test suites. The user can also save the results to JSON files instead of uploading them to the database to reduce emulation time on simulated CPUs. Users can also define new test specifications, add additional tests, and have the configuration saved to be used later.

The tool has error-catching code to handle most failures by either restarting or prompting the user at which step the error has occurred. The tool is automated in that the user can run multiple benchmarks and upload results with just one command. Our project addresses NVIDIA's problem with significantly reduced time and effort for testing their new CPU development process.

7. Future Work

This chapter lists some potential future work that a project immediately following this one could undertake. It includes three topics based on the three main parts of this project: Python script backend, Perflab database, and Tableau visualization frontend.

7.1. Python Script

This project's most immediate future work is adding new benchmarks to the workflow. We have covered over forty different benchmarks during our project's timespan, but NVIDIA has more on their list for their benchmarking needs. Based on our instructions, other users have already successfully added some benchmarks just after the end of our project.

Future work also includes polishing the profiling feature in the shell script. Currently, the profiling provides information about the CPU times of each part of the benchmark execution. Profiling should also include restricting which hardware (CPU core or RAM) to participate in the benchmark execution. We can use some Linux packages we explored to add those modifiers at the execution stage of the shell script.

Last but not least, the future teams working after this project could upgrade the initialization script of the entire Python script backend. Merging the script into the Python shell script can also grant the user the ability to install the benchmark test suites while using the shell. This would include adding the utility script to check the machine's Internet connection and installation status and creating another jsonc file to store the automated scripts to install each supported benchmark.

7.2. Database

Future database expansion could consider adding versioning to grant the database manager the ability to change the schema without losing previous data. This should include scripts to migrate the database with the help of applications like Flyway [12].

7.3. Tableau Visualization

Our Tableau currently only reflects the production database, not the staging database. Adding the ability to visualize the staging database would speed up the development processes on the staging database before deploying them. The future team can follow the Get Permission on Tableau steps in Appendix A to create another Tableau folder to connect with the staging database.

The three tables in Tableau already provide plenty of information to the user, but they could use more visualization types other than bar charts. Providing a variety of visualization graphics can make the user spot performance differences more efficiently. Future work could explore more on the use of Tableau and consult with other teams to add more visualization types.

8. References

[1] *Gitlab Docs*. GitLab. (n.d.). Retrieved December 30, 2021, from https://docs.gitlab.com/ee/

[2] Newspaper PM. (1945). *AWS Graviton Processor*. Amazon. Retrieved December 30, 2021, from https://aws.amazon.com/pm/ec2-graviton/

[3] Gregg, B. (n.d.). *Perf examples*. Linux perf Examples. Retrieved December 30, 2021, from https://www.brendangregg.com/perf.html

[4] *TechTarget*, T. T. (2011, March 24). *What is Numa (non-uniform Memory access)?* WhatIs.com. Retrieved December 30, 2021, from https://whatis.techtarget.com/definition/NUMA-non-uniform-memory-access

[5] *NUMACTL: Control numa policy for processes or shared memory*. numactl: Linux Man Pages (8). (n.d.). Retrieved December 30, 2021, from https://www.systutorials.com/docs/linux/man/8-numactl/

[6] *Introduction to control groups (cgroups) red hat enterprise linux 6*. Red Hat Customer Portal. (n.d.). Retrieved December 30, 2021, from https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01

[7] *About PostgreSQL*. PostgreSQL. (n.d.). Retrieved December 30, 2021, from https://www.postgresql.org/about/

[8] *What is Tableau?* Tableau. (n.d.). Retrieved December 30, 2021, from https://www.tableau.com/why-tableau/what-is-tableau

[9] Google. (n.d.). *Google data studio help*. Google. Retrieved December 30, 2021, from https://support.google.com/datastudio/answer/6283323?hl=en

[10] *Google Data-Studio vs Tableau: A comparative analysis of visualization tools*. Digital Analytics, Conversion Rate Optimization and Business Intelligence. (2019, December 6). Retrieved December 30, 2021, from https://insightwhale.com/google-data-studio-vs-tableau-a-comparative-analysis-of-visualization-tools/

[11] Zero-one-infinity rule. (n.d.). Retrieved December 30, 2021, from http://www.catb.org/jargon/html/Z/Zero-One-Infinity-Rule.html

[12] *Flyway db*. Flyway. (2020, December 16). Retrieved December 30, 2021, from https://flywaydb.org/

Appendix A: Tableau Guide

Visualization with Tableau is designed to be user-friendly. In general, visualization with Tableau has three worksheets: Benchmark Performance, Data on Benchmark Performance, and Machine Information Raw Data. If users want to add a new benchmark feature, here is a step by step guide:

Get Permission on Tableau

1. Alexey is helping us to create a site on Tableau. Once it's created, ask Sean for permission to join the site.
2. Submit a Tableau Access Request here or search a Tableau Access Request on its portal.
3. After the request is approved, you will have access to the site as an explorer (default), allowing you to view worksheets. Click on your Tableau profile and then click *My Account Settings.* Check your site's role.
4. If you are an explorer (can publish), you can create a worksheet.
5. If you want to edit others' worksheets, you need to ask a site administrator to permit you. You could change permission by selecting a specific worksheet and clicking on *more actions (figure: ...) → permissions.*

Add a New Benchmark Feature

1. Click *Edit* and go to the worksheet that you want to edit
2. Click *Refresh* to load live data from the database. It takes some time for Tableau to update to live data. If you want to see new data immediately, use *Refresh.*

3. Drag new benchmark feature from *Tables* to *Rows* or *Columns*. If you want to display this feature as printouts rather than graphs, right-click on this feature and choose *Convert to discrete.*
4. Modify features under *Rows* or *Columns.* We are using bar graphs in the Benchmark Performance worksheet. If you want to apply other graphs, click on *Show me* and select other types of graphs.
5. If you want to add a filter for this new feature, drag this feature from *Tables* to *Filters.* Select *From List* and *Only Relevant Values* under General in most cases. *Only Relevant Values* show suitable selectable choices of other filter selections. Be careful at choosing all filters as *Only Relevant Values* as users may not get to some data simply because they don't know what exact combination of filter choices would allow them to see that data. To prevent this, *All Values in Database* is used for *Applications* to ensure users have full access to all benchmarks.
6. Adjust types of filter display. In most cases, *multiple values (dropdown)* are used.
7. If you want to link this filter with other worksheets and change visualization on other worksheets when this filter is selected, click on the triangle on the right side of this filter. Select *Apply to worksheets → All using this data source* or *All using corresponding data source.*
8. Save the change.

Application (Benchmark) doesn’t Appear.

If you are running a new benchmark and it doesn’t appear under the *Application* filter, you need to edit the *Application* filter:

1. Click *Edit* and go to the worksheet that you want to edit

2. Click *Refresh* to load live data from the database. It takes some time for Tableau to update to live data. If you want to see new data immediately, use *Refresh.*
3. Click *Revert* to clean filter status to default.
4. Go to *Filter* in the second area beside *Tables*, right-click on *Application.* Click on *Edit Filter.* Choose *Select From List* and *All Values in Database* under General. Make sure the *All* choice box is selected and click *Apply* or *OK*.
5. If the previous step doesn't work, choose *Use All* under General as a filter setting.
6. Save the change.

Application filter is the majority filter, and it is selected as showing all values. The rest of the filters are selected as only showing relevant values. This means that once the application is selected, the selections in the filters are only related to the selected application. When a new filter is applied, please consider only displaying relevant values.

General Notes for Editing Worksheets

7. Ensure all filters on each worksheet select the *All* choice box when you save your change on worksheets.
8. Consider the relationship between worksheets. If you want to have the exact visualization content, make sure to *Apply it to worksheets → All using this data source* is selected on each filter.
9. Be careful about using *Only Relevant Values* under filters.
10. If you are looking forward to learning more about Tableau, go and check their tutorial videos.

Appendix B: Our GitLab project code

gitlab-master.nvidia.com/wpi-project/th500-software-performance