# Workload Characterization Tools

## Release 1.0.0

**Advanced Micro Devices (AMD)**

**Research and Advanced Development (RAD)**

**Advanced Architectures and Accelerators (AAA)**

**February 09, 2024**

# 1.0 Overview

Understanding a workload's bottlenecks is critical in improving the performance of an application, but also proposing future hardware and software features that target those bottlenecks. AMD's Research and Development (RAD) have developed tools such as Omnitrace and Omniperf to assist with this, but finding performance bottlenecks can still be time consuming and tedious.

Our workload characterization tool analyzes the results from both Omnitrace and Omniperf to generate an end-to-end analysis including CPU, GPU, and communication timings as well as a thorough breakdown of the GPU's performance bottlenecks. The scripts are written in Python, are lightweight, and easy to use and modify.

Our workload characterization tools consist of three Python scripts:

1. **workload-characterizaion.py** – the main script that parses all the Omnitrace and Omniperf data and generates a CSV file with all the timing and bottleneck information.
2. **plot-roofline.py** – a script that uses the Omniperf data to generate the hierarchical roofline for the application.
3. **plot-characterization.py** – a script that generates the end-to-end CPU/GPU analysis and GPU bottleneck analysis.

# 2.0 Features

Our workload characterization scripts come with the following features:

- Generate detailed timing information for the CPU, GPU, and communication (CPU-to-GPU, GPU-to-GPU, etc.).
- Create a roofline plot that the user can interact with.
- Generate detailed breakdown of GPU performance bottlenecks.
- The tool currently supports MI200 hardware, but the tools are extensible and future hardware support is coming.

# 3.0 Installation

Although our scripts are written in Python and only need to be downloaded, the scripts take Omnitrace and Omniperf data as inputs and as such, it is assumed the user has already installed and used these.

- Instructions for installing and using Omnitrace can be found **here.**
- Instructions for installing and using Omniperf can be found **here.**

# 4.0 Usage

## 4.1 Omnitrace Usage

1. To use Omnitrace on the desired benchmark, go to the directory with the executable and type the following command:

   ***omnitrace-instrument -- <name_of_your_executable> <options if applicable>***

   For example, if the workload were DGEMM, the command would be:

   ***omnitrace-instrument -- ./gemm R_64F R_64F R_64F R_64F OP_N OP_T 8640 8640 8640 8640 8640 8640 36 300***

2. Upon completion, Omnitrace will create a new directory and place a Perfetto file in it usually named ***perfetto-trace-XYZ.proto*** where XYZ is a number. This Perfetto file is used by our tool.

3. Go to the location where our Python scripts reside and create a new folder named “omnitrace.” For example, if you are working within UNIX/Linux:

   ***mkdir omnitrace***

4. Move the Perfetto file that Omnitrace generated into the newly created omnitrace folder.

5. It is suggested that the Perfetto file be renamed since the naming convention from Omnitrace is not very descriptive. For example, if you are working within UNIX/Linux:

   ***mv perfetto-trace-XYZ.proto coraldgemm-trace-XYZ.proto***

## 4.2 Omniperf Usage

1. To use Omniperf on the desired benchmark, go to the directory with the executable and type the following command:

   ***omniperf profile -n <your_desired_workload_name> -- <name_of_your_executable> <options if applicable>***

   For example, if the workload were Mixbench, the command would be:

   ***omniperf profile -n my_Mixbench_run -- ./mixbench-hip***

2. Upon completion, Omniperf will create a new directory named “workloads” and inside this directory will be another directory named after the desired workload name given with the previous command.

3. Navigate the newly created directories until you find the files ***pmc_perf.csv*** and ***roofline.csv***.

4. Go to the location where our Python scripts reside and create a new folder named “omniperf” and in that omniperf folder create another folder named after your workload.

5. Move the *pmc_perf.csv* and *roofline.csv* files into the newly created workload folder within the omniperf folder.

## 4.3 Workload Characterization Usage

### 4.3.1 Workload Characterization Script

1. Open the ***workload-characterization.py*** script in a text editor or Integrated Development Environment (IDE).

2. Near the middle of the script there should be two variables highlighted in **Figure 1.** The variables ***peak_valu_flops*** and ***peak_mfma_flops*** are used to establish the peak vector (VALU) and matrix fused multiply add (MFMA) performance. These values were found using Omniperf's microbenchmarks when you ran Omniperf.

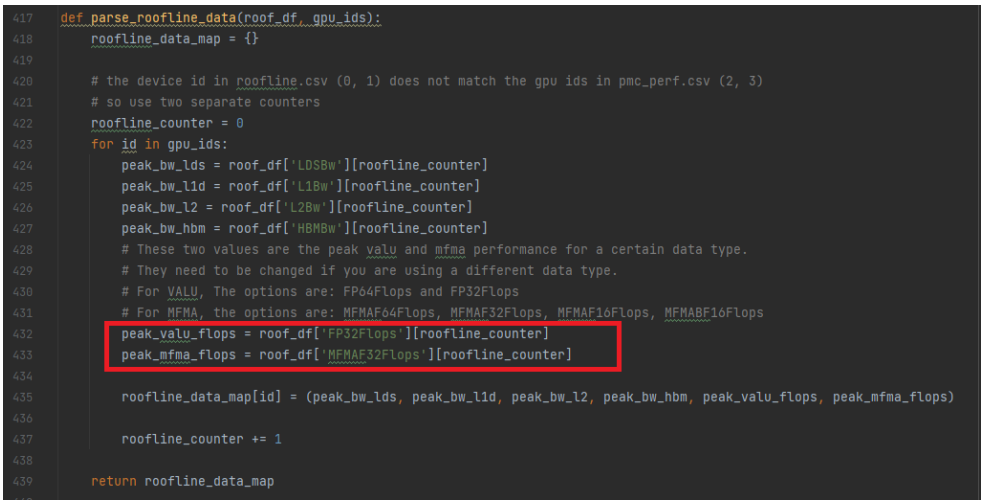


**Figure 1:** Changing to the correct data types

3. Based on the data type used by your workload, change the two variables to the appropriate value. For example, for 32-bit floating-point (FP32) the setting should be ***FP32Flops*** and ***MFMAF32Flops.***

4. Immediately below the previous variables should be a variable named ***util_ratio,*** which is highlighted in **Figure 2**. This variable is used to set the threshold for percentage of peak Floating-Point Operations per second (FLOPS) we want to compare against.

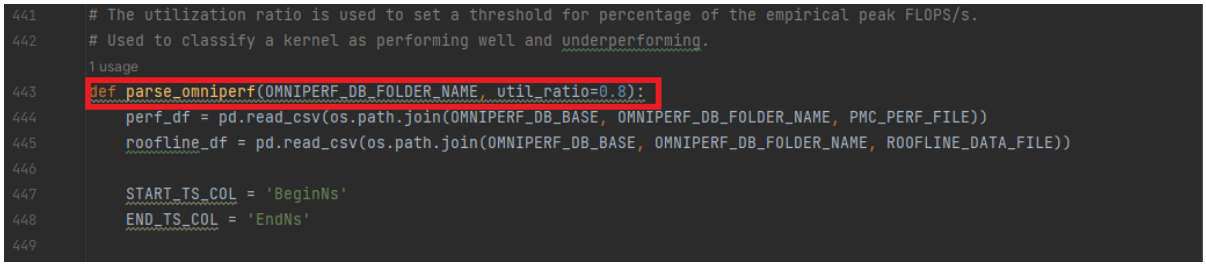


**Figure 2:** Changing the utilization ratio

5. By default, the script is set at a threshold of 80% (i.e., 0.8) meaning that kernels that perform below 80% of the peak performance will be classified as underperforming while the kernels above 80% of the peak performance are performing well. If a different threshold is desired, set the variable accordingly.

6. Near the bottom of the script, there are four variables that **need to be modified every time** a new workload is to be profiled. Those variables are highlighted in **Figure 3**.

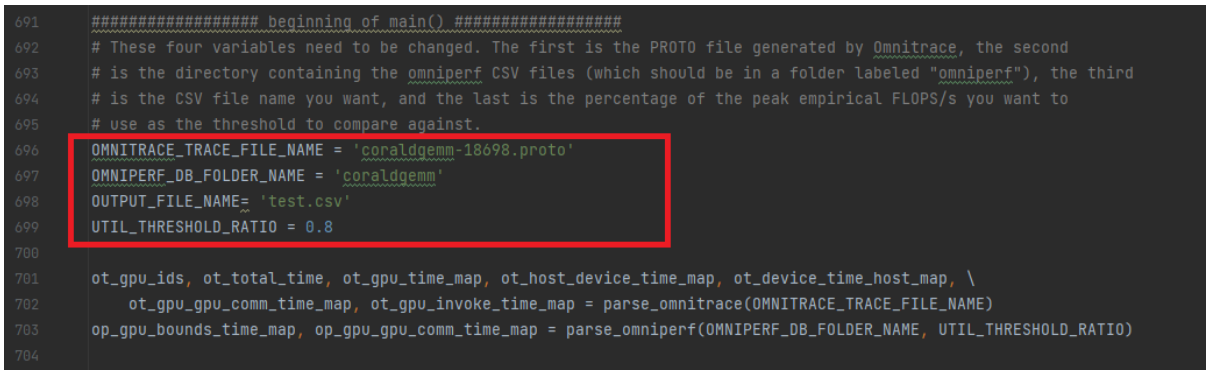


**Figure 3:** Changing the input and output settings

7. The ***OMNITRACE_TRACE_FILE_NAME*** variable should be set to the name of the Perfetto file generated by Omnitrace in an earlier step.

8. The ***OMNIPERF_DB_FOLDER_NAME*** variable should be set to the name of the folder containing the ***pmc_perf.csv*** and ***roofline.csv*** files generated by Omniperf in an earlier step.

9. The ***OUTPUT_FILE_NAME*** variable is used to set the file name for the Comma Separated Values (CSV) output that the script will generate. Set this variable to the desired file name.

10. Lastly, the ***UTIL_THRESHOLD_RATIO*** variable is used to set the threshold for the percentage of peak performance. By default, it has been set to 0.8 (80% threshold) but can be modified if desired.

11. The script is now ready to be run and will generate a CSV file named after the output given earlier.

### 4.3.2 Plot Roofline Script

1. Open the ***plot-roofline.py*** script in a text editor or IDE.

2. Near the bottom of the script there is one variable that needs to be modified. That variable is highlighted in **Figure 4.**

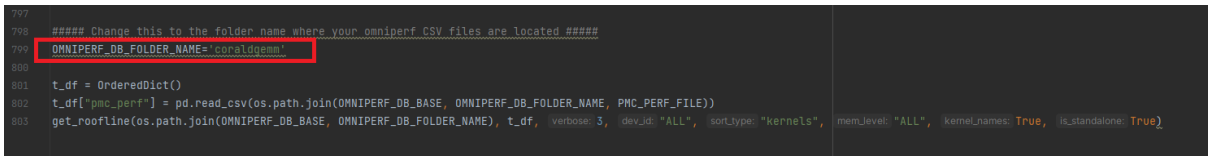


**Figure 4:** Changing the input setting

3. The ***OMNIPERF_DB_FOLDER_NAME*** variable should be set to the name of the folder containing the ***pmc_perf.csv*** and ***roofline.csv*** files generated by Omniperf in an earlier step.

4. The script is now ready to be run and will generate two interactable HTML files containing the FP32 and FP16/INT8 roofline plots, respectively.

### 4.3.3 Plot Characterization Script

1. Open the ***plot-characterization.py*** script in a text editor or IDE.
2. Near the top of the script there is one variable that needs to be modified. That variable is highlighted in **Figure 5.**

```
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import pandas as pd
plt.rcParams['figure.figsize'] = [20, 8]
plt.rcParams['font.size'] = 24

# This needs to be changed to the CSV filename that was generated by the workload-characterization.py script
FILENAME = 'test.csv'
data_df = pd.read_csv(FILENAME)
GRAPH_NAME = FILENAME.replace( _old: ".csv",  _new: "")

```

**Figure 5:** Changing the input setting

3. The ***FILENAME*** variable should be set to the name of the CSV file generated by the ***workload-characterization.py*** script earlier.
4. The script is now ready to be run and will generate two PDF files, ***<your_filename>-e2e_time.pdf*** and ***<your_filename>-gpu_time.pdf.*** The e2e_time file contains the end-to-end CPU and GPU timing and the gpu_time file contains a breakdown of the workload's bottlenecks on the GPU.

## 5.0 Interpreting the Results

### 5.0.1 Workload Characterization CSV File

1. The ***workload-characterization.py*** script generates a CSV file that contains a detailed analysis of the Omnitrace and Omniperf profiling data.
2. The first eight columns show the amount of time the workload spent on the CPU, GPU, communication such as CPU-to-GPU, and kernel invoke time. This portion of data is used to create the visualized analysis found in the e2e_time.pdf file.
3. The 18 columns to the right are separated into three buckets for classifying a workload's bottlenecks. Those buckets are the amount of time the workload's kernels are performing no operations (no_flops), performing under the threshold set earlier (under_util), and performing above the threshold (above_util).
4. Each of those three buckets have six columns representing the four levels of the memory hierarchy (Local Data Store (LDS), L1 cache, L2 cache, and High Bandwidth Memory (HBM)) and the two types of compute units (VALU and MFMA). In other words, there are a total of 18 buckets that break down the amount of time the workload spent in each category. This allows the user to see the workload's bottlenecks at a quantifiable and fine-grain level.
5. Although the raw data can be useful, the main purpose of the CSV file is to be fed into the ***plot-characterization.py*** script so that the data can be visualized for the user.

### 5.0.2 Roofline HTML Files

1. The ***plot-roofline.py*** script generates two interactable HTML files that show the roofline for the workload. These roofline plots give the user a high-level view of the workload's performance. Specifically, a performance estimate of the individual kernels and can be used in conjunction with the other outputs. **Figure 6** shows an example roofline plot for the benchmark Mixbench. The roofline plot shows that Mixbench has kernels that are both compute (vector) and memory bound.

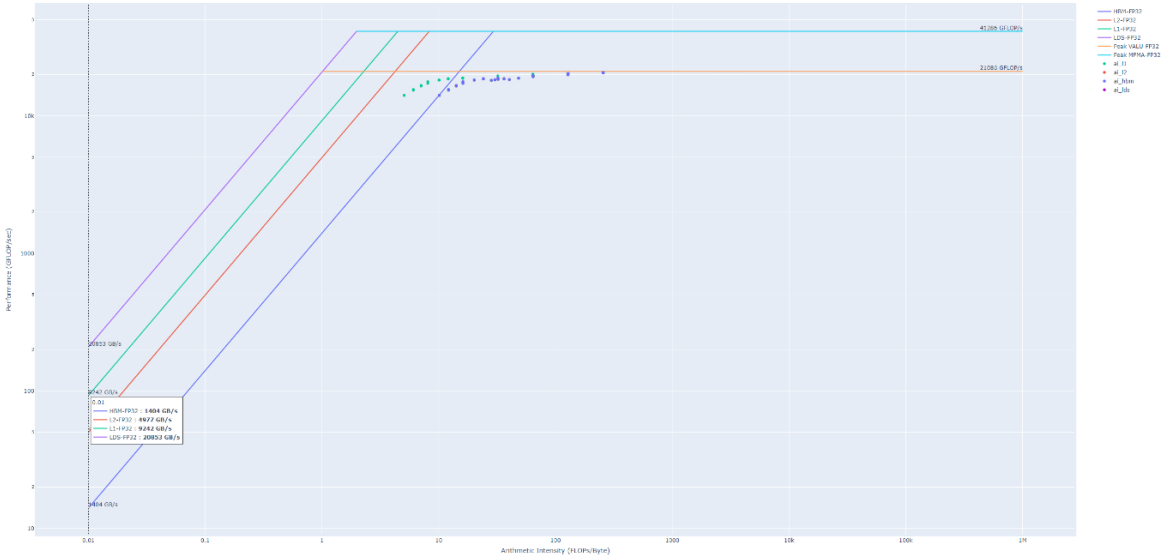

**Figure 6:** An example roofline plot for Mixbench

### 5.0.3 Plot Characterization PDF Files

1. The ***plot-characterization.py*** script generates two PDF files that visualize the analysis data created by the ***workload-characterization.py*** script (i.e., the CSV file).

2. The ***e2e_time.pdf*** file contains an end-to-end breakdown of the amount of execution time the workload spent on the CPU, GPU, and communication time such as CPU-to-GPU communication (Host to Device), GPU-to-CPU communication (Device to Hos) and GPU-to-GPU communication (GPU-GPU). **Figure 7** shows an example of an end-to-end timing breakdown for Mixbench, which spends most of the time on the CPU, GPU, and communicating from the CPU to the GPU.

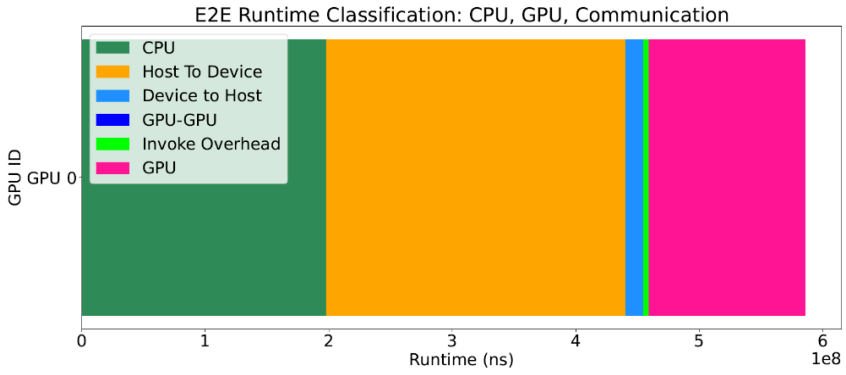


**Figure 7:** An example of an end-to-end analysis for Mixbench

3. The ***gpu_time.pdf*** file contains a breakdown and analysis of the workload's GPU bottlenecks. It quantifies the amount of time the workload spent performing above the set threshold, the amount of time spent below the threshold (the portion marked with '/'), and lastly the amount of time spent performing no operations (the portion marked with 'X'). **Figure 8** shows an example quantifying Mixbench's performance bottlenecks. A substantial portion of Mixbench performs above the threshold in both vector and memory operations, a small portion of Mixbench performs below the threshold on vector operations and lastly, a portion of the workload's time is spent performing no operations.

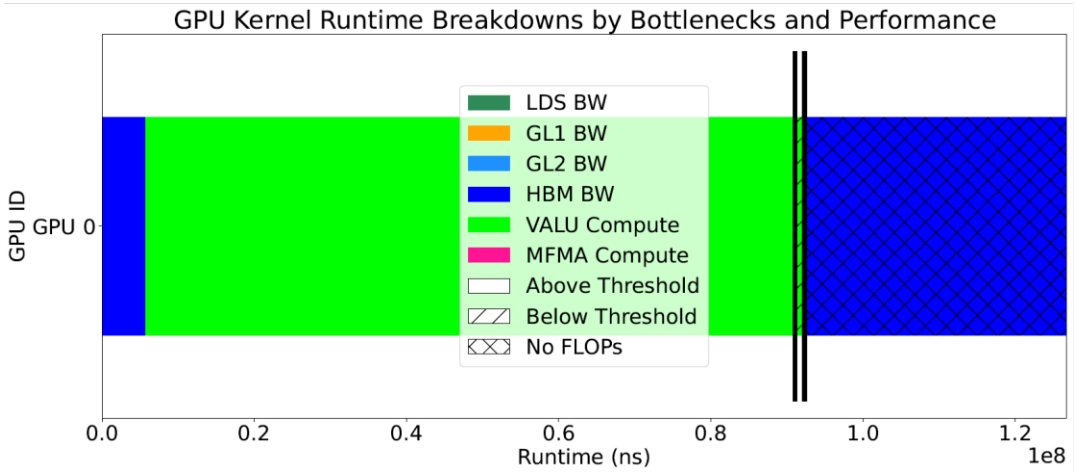


**Figure 8:** An example analysis of Mixbench's GPU bottlenecks

## 6.0 Known Bugs and Workarounds

### 6.0.1 Omnitrace "Did not end" Bug

1. At the time of writing there is a known Omnitrace bug where a minority of workloads do not end and the ***workload-characterization.py*** script will log an incorrect total trace time of -1 (i.e., column B in the CSV file). **Figure 9** shows a visualization of the entire workload in the Perfetto UI and highlights where the error occurs.

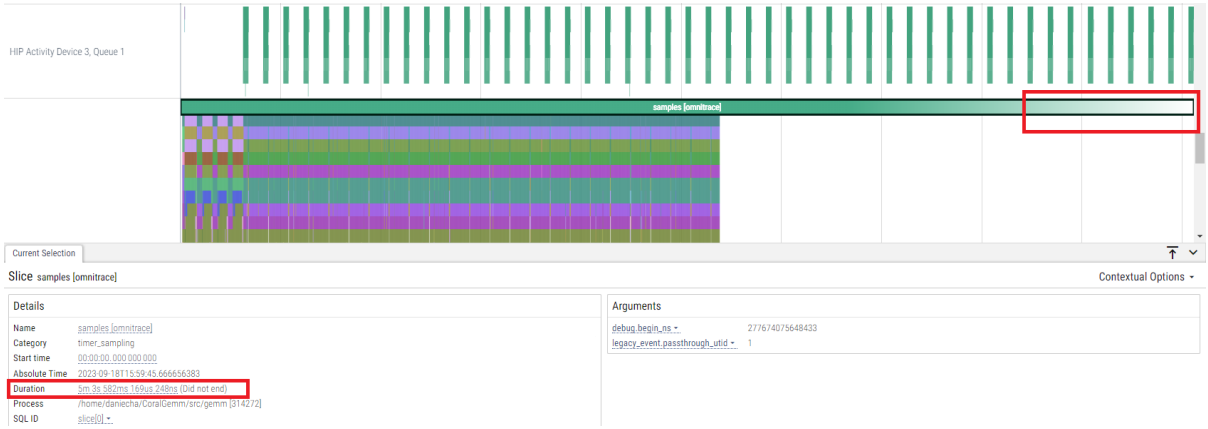


**Figure 9:** An example of the "Did not end" bug in Omnitrace

2. The semi-transparent color near the top right shows the trace not ending and the information in the bottom left corner says, “did not end.” When a trace does not end, the total trace time is logged to -1. Since the total trace time is used to create the end-to-end, a value of -1 causes the ***plot-characterization.py*** script to generate an inaccurate end-to-end analysis.

3. To fix this, the total trace time in the CSV file needs to be manually updated with the correct value. If possible, the correct value can be found by opening the Perfetto Proto file generated by Omnitrace in the Perfetto UI (https://ui.perfetto.dev/).

4. Manually navigate the trace in the Peretto UI and find a trace at the end that does complete and note its end time. Similarly, find a trace at the beginning and note its start time. The difference between these is the total trace time. An example of this workaround can be seen in **Figure 10.**

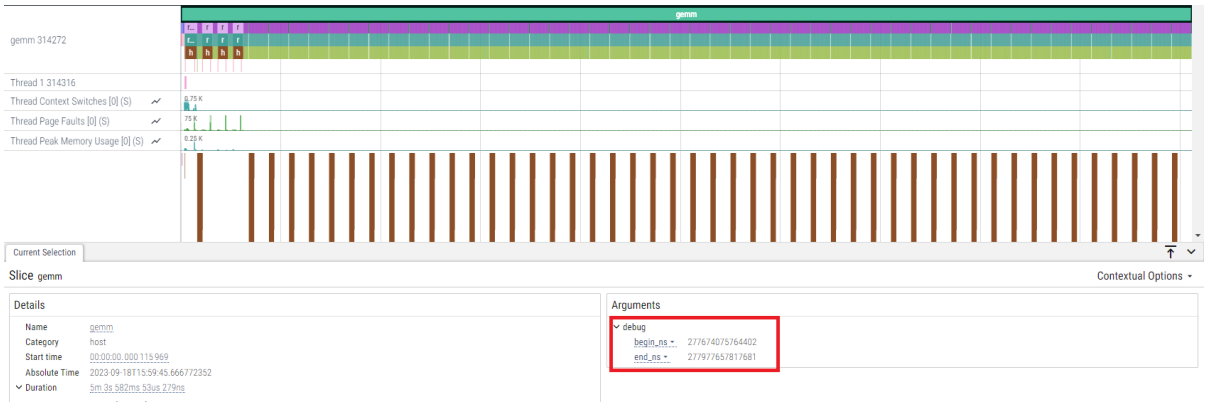


**Figure 10:** An example showing a workaround to the “Did not end” bug

5. The Omnitrace team is aware of this bug and hopefully it will be fixed in a future Omnitrace revision. Until then, unfortunately, the manual workaround can be tedious. The bug ticket has been submitted (https://github.com/AMDResearch/omnitrace/issues/311).

### 6.0.2 Total Trace Time does not add up Bug

1. Occasionally the CSV file generated by **workload-characterization.py** will have a total trace time that is too small. Where the summation of the GPU and communication timing is greater than the total trace time. The **plot-characterization.py** script will then create an end-to-end analysis with negative times as shown in **Figure 11.**

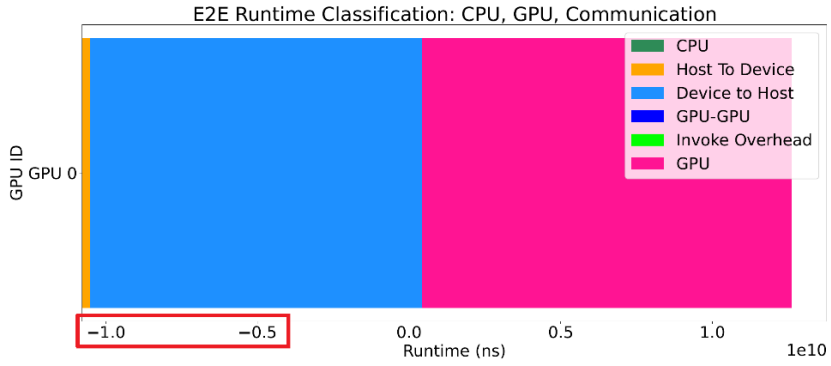


**Figure 11:** An example of the total trace time being too small

2. It is believed that this bug is occurring due to overlap of the GPU kernels and the device to host communication. Adding these two values together creates a value greater than the total trace time. The reality is, the Device to Host and GPU time should overlap and only counted once, but our tool cannot show this overlapped time correctly. A truncated version of the Perfetto file is shown in **Figure 12.** In the figure the hipMemcpy operation completely overlaps with the previously invoked GPU kernel, bodyForce().

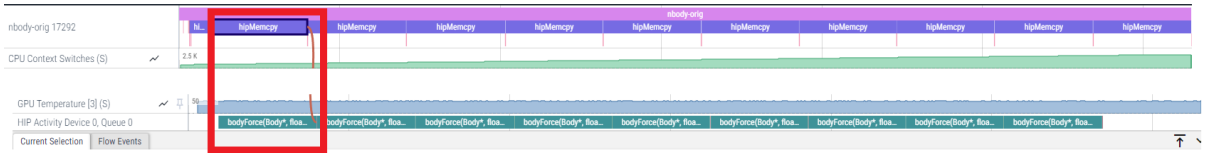


**Figure 12:** Overlapping GPU kernels with GPU kernel invocations

3. Looking at the Perfetto file in the Perfetto UI shows that the total trace time recorded by our tool is correct and that value should be left alone.

4. Although there is currently no solution to completely fix this, the user can open the CSV file and manually edit the device to host entry that is being double counted and zero it out. Admittedly, this is not 100% correct since device to host timing is now not being tracked, but the CPU, GPU, and other remaining traffic should be correct.
5. Re-run the **plot-characterization.py** script and it should generate a more accurate end-to-end analysis. A corrected end-to-end analysis is shown in **Figure 13.**

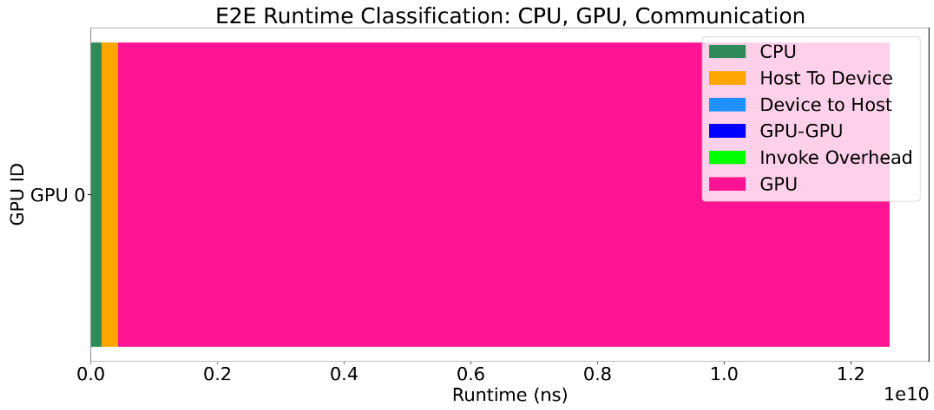


**Figure 13:** A corrected end-to-end analysis

6. Although this is admittedly not a 100% accurate end-to-end analysis, we have confidence that the GPU bottleneck breakdown is correct. Therefore, the user can be confident with the conclusions drawn regarding the workload's bottlenecks on the GPU. Also, the Device to Host communication time in **Figure 12** is the correct amount of time, but again, the communication time would actually overlap with the GPU, and we have no way of visualizing this correctly.

7. This bug will be investigated further and hopefully solved in a future revision.