HWSW Project:
Benchmark Optimization, Analysis, and Hardware Acceleration Proposal

In this project, you will explore, analyze, optimize, and accelerate the standard Python interpreter and runtime. As Python code is so common these days, improving its performance can save a lot of CPU cycles and money. To this end, you will focus on benchmarks from the **pyperformance** benchmark suite, simulate real-world Python workloads and test the performance of specific operations such as encryption, data serialization, logging, or memory usage. Ultimately, you will be expected to propose software (and possibly hardware) optimizations that can accelerate your selected Python benchmarks. These optimizations can either accelerate the benchmarks themselves or, better yet, accelerate the entire Python interpreter and runtime.

Each team will select two benchmarks from the **pyperformance** benchmark suite and deep dive into the selected benchmarks, how Python is executed (hint: bytecode), the Python interpreter, and the run time libraries it uses. You will need to gain that deep understanding to identify the execution bottlenecks and explain why they may be inefficient.

Once you understand the behavior of the benchmarks, you'll use the performance profiling tool **perf** and flame graphs to visualize and analyze the program's runtime behavior. These tools will help you detect bottlenecks and specific parts of the code that slow down execution or use too many resources. After detecting these hotspots, your next goal is to optimize the benchmark. This could mean replacing an algorithm with a faster one, using a more efficient library, or restructuring the code for better performance.

To demonstrate the effect of your changes, you will compare performance results from before and after the optimization. You will then take your analysis one step further by proposing a hardware acceleration solution for a selected part of the benchmark that could benefit from it. For example, you might suggest using a hardware accelerator to speed up encryption or memory access. You will need to describe the hardware module's function, how it integrates with software, and its performance trade-offs. A block diagram is required to visualize how the hardware connects with the software system.

Your work must be documented and uploaded to a **Git** repository, including reports, scripts, optimized code, and a clear README file explaining how to reproduce your results. You are encouraged to use **AI** tools such as ChatGPT or GitHub Copilot to assist you during the project, but you must also submit the prompts or instructions you used when interacting with these tools. The goal is not just to write faster code, but to demonstrate a deep and thoughtful understanding of performance tuning and how software optimizations and hardware acceleration can work together to improve computational efficiency.

Besides the technical deliverables, you will present your work to the course staff in a (hopefully) frontal 30 minute presentation. You will present your work in 20 minutes (prepare slides) and leave 10 minutes for questions. In this presentation **you will explain your decisions, discuss your analysis, and answer our questions** about your optimizations and proposed hardware solutions.

Guide: Setting Up pyperformance in a Virtual Environment

Step 0 Copy the Image File:

Copy the prepared Ubuntu image from the server to your local machine:

```
scp <your_username>@tangerine/csl/ohad.eitan/ohad/jammy-server-cloudimg-amd64-disk-kvm.img
```

Step 1 Launch QEMU VM:

Start your QEMU environment using the copied image.
(see previous guide)

Step 2: Run perf with Python Debug Symbols:

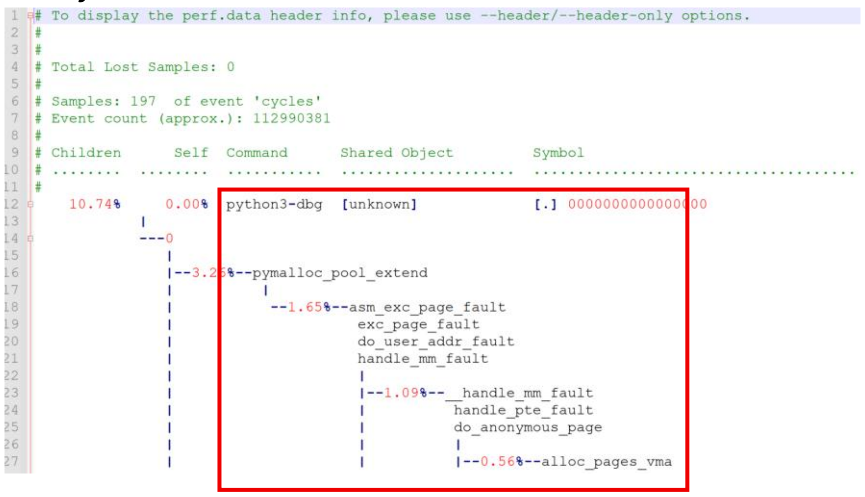
To profile a specific benchmark, for instance json_dumps using perf, make sure you're using the debug build of Python (python3-dbg):

```
perf record -F 999 -g -- python3-dbg -m pyperformance run --bench json_dumps
```

After the run, export the report to a text file:

```
perf report --stdio > perf_report.txt
```

What you will see:



```
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 197  of event 'cycles'
# Event count (approx.): 112990381
#
# Children      Self  Command      Shared Object         Symbol
# ........  ........  ...........  ....................  ...................................
#
    10.74%     0.00%  python3-dbg  [unknown]             [.] 0000000000000000
            |
            ---0
               |
               |--3.2[illegible]%--pymalloc_pool_extend
               |          |
               |           --1.65%--asm_exc_page_fault
               |                     exc_page_fault
               |                     do_user_addr_fault
               |                     handle_mm_fault
               |                     |
               |                     |--1.09%--__handle_mm_fault
               |                     |          handle_pte_fault
               |                     |          do_anonymous_page
               |                     |          |
               |                     |          |--0.56%--alloc_pages_vma
```

Thanks to the presence of Python debug symbols, the report will reveal internal Python function calls and stack traces used during the benchmark. This helps identify performance bottlenecks within Python itself. We encourage you to read the **perf** man page and google usage examples to learn about more options the tool provides.

Proposed workflow:

1. **Benchmark Analysis:**

- Begin by reviewing and understanding the code for the selected benchmarks:
    - o Identify the purpose of the benchmark and the libraries used.
    - o Explore the data structures and algorithms employed in the benchmark.
    - o If necessary, dive into any third-party libraries or dependencies used by the benchmark to fully understand their inner workings (e.g., libraries like numpy, pycrypto).
- Gain a deep understanding of the benchmark's behavior, which will be essential for detecting bottlenecks and proposing effective improvements.
- The Python code is executed using the Python interpreter, so at some point you will need to examine interpreter code and understand how the interpreter works.

2. **Understand pyperformance:**

- Familiarize yourself with how pyperformance works. You should learn how to:
    - o Run the benchmark tests.
    - o Capture performance results using pyperformance.
    - o Interpret performance data effectively
- Ensure that you are comfortable running benchmarks and comparing performance data, as this will form the foundation of your analysis and optimization process.
-

3. **Generate a Flame Graph:**

- Utilize pyperformance to generate flame graphs. These graphs will help visualize the performance hotspots in the benchmark by showing which parts of the code are consuming the most resources.
- Create flame graphs for each of the selected benchmarks to identify the most resource-intensive areas in the code.

4. **Detect Bottlenecks:**

- Analyze the generated flame graphs and other profiling data to detect performance bottlenecks. Note that bottlenecks will likely appear as part of the Python interpreter or libraries.
- Pinpoint key areas in the benchmark where performance can be improved. These might include inefficient algorithms, slow function calls, or excessive memory usage.

5. Suggest Improvements:

- Based on your analysis, propose improvements to the benchmark code. You are encouraged to:

    - Use more efficient libraries, algorithms, or data structures.
    - Implement code refactoring or other optimization strategies.
- Provide clear and well-reasoned suggestions for optimizing the benchmark, backed by data that show improvements such as reduced execution time or better memory efficiency.

6. Show Performance Improvements:

- After implementing the suggested optimizations, run the benchmarks again and capture the performance data.
- Demonstrate the improvements in performance after applying the optimizations, with concrete data comparing the optimized benchmark to the original one.
- If you can identify two or more benchmarks where your optimizations result in a performance improvement of **at least 5%**, that will be considered sufficient.

7. Propose Hardware Acceleration:

- For one or two key components of the benchmark, propose a hardware acceleration solution. Examples include:
    - Accelerating dictionary operations with specialized hardware.
    - Speeding up AES encryption using custom hardware modules.
- Provide a comprehensive proposal for hardware acceleration that includes:
    - Inputs and outputs: What are the inputs and outputs of the hardware solution?
    - Hardware/software interface: How will the hardware module interact with the existing code? What software changes are needed to integrate the hardware?
    - Acceleration justification: Explain how the proposed hardware solution improves performance, providing expected performance metrics.
    - Block diagram: Include a block diagram showing the hardware solution and its integration with the system.
    - Performance/area/power trade-offs: Discuss the trade-offs between performance gains, hardware complexity, and power consumption.
- Demonstrating a solid understanding of both hardware and software is required, but you are not expected to build or physically test any hardware. Don't worry—focus on the concepts, not implementation.

8. Git Repository and Documentation:

- Upload all your project files to a Git repository. Ensure that your repository contains:
    - The benchmark reports (report_<name_of_benchmark>.txt).

    - Shell scripts for running the benchmarks (script_<name_of_benchmark>.sh).
    - Any additional files you have created (Python scripts, configuration files, etc.).
    - A README.md file that explains the structure of the repository and provides instructions on how to run the scripts.
- Maintain good version control practices by using clear commit messages that demonstrate the development process and your understanding of the material. Properly structured commits and a well-organized repository will earn you bonus points (+5)

9. Presentation:

- You are required to prepare a 20–25 minutes presentation and present it at a scheduled time determined by the course staff.
- The best structure for your presentation is to simply follow the flow of your project work, from initial analysis to optimization and hardware proposal.
- During the session, you will be asked questions for 5–10 minutes, so be ready to explain your choices clearly.
- Make sure you have working code available to demonstrate and support your explanations (**No need to add all your code to the presentation!**). Your goal is to walk the course staff through your project as if you're explaining it to a fellow ECE student (someone with a technical background who didn't do the project themselves). Focus on teaching, guiding, and showing your understanding of the work.

10. AI Tools:

- You may use AI tools (such as ChatGPT, GitHub Copilot, etc.) to assist you in the project. However, you must provide the prompts or instructions you used to interact with the AI tool.
- The goal is to assess how well you understand the material, so the AI tools should be used as an aid, not a substitute for your own analysis and work.

Approved pyperformance benchmarks:

We have selected the following 10 benchmarks for you to choose from. You need to select **2 benchmarks** for this project. You can refer to the following link for more details about the benchmarks:

pyperformance Benchmark Documentation

Index	**Name of Benchmark**
1	crypto_pyaes
2	Deepcopy
3	Logging
4	Mdp
5	Pathlib
6	Pickle and pickle_dict
7	Pyflate
8	unpack_sequence
9	json_dumps
10	gc_collect

Alternative benchmarks:

If there's a different software project of similar scale you are interested in optimizing rather than Python, you may ask the course staff for permission to examine that code instead of Python.

What You Need to Submit:

For each benchmark you select, you are required to submit the following:

1. Benchmark Reports:

- **File Name**: report_<name_of_benchmark>.txt
  - **Contents**:
    - Overview: Description of the benchmark, libraries used, and data structures employed.
    - Initial Analysis: Performance analysis, flame graphs, profiling data.
    - Optimizations: Description of improvements made, including any external libraries or algorithms used.
    - Performance Comparison: Show how performance improved after optimizations.

        - Hardware Acceleration Proposal: If applicable, explain the proposed hardware solution with inputs, outputs, trade-offs, and a block diagram.
        - Conclusion: Summarize the impact of the optimizations and hardware acceleration.

2. Benchmark Execution Scripts:

- **File Name**: script_<name_of_benchmark>.sh
    - **Contents**:
        - Environment setup and dependency installation.
        - Benchmark execution using pyperformance or other profiling tools.
        - Flame graph and performance data generation.
        - Post-optimization benchmark execution with performance comparison.

3. Additional Files (Optional but encouraged):

- Python scripts (*.py), performance logs, configuration files, or any additional files that assist in your analysis or optimizations.

4. AI Tool Prompts:

- **File Name**: prompt.txt or prompt.docx
    - **Contents**:
        - Document the prompts or instructions used when interacting with AI tools.

The course staff will be available to assist you on the course forum.