COE 403 Assignment 1

BASIL ALASHQAR
202045700

March 23, 2025

Contents

1 Introduction

Matrix multiplication is one of the most fundamental operations in scientific computing, data analytics, and many areas of computer architecture. In particular, matrix–matrix multiplication (commonly referred to as *mmult*) lies at the heart of a wide range of computational tasks, including linear algebra routines, machine learning algorithms, and numerical simulations. As such, understanding how matrix multiplication behaves on different hardware architectures is crucial for both software developers and hardware designers.

This report aims to document the design, implementation, and evaluation of a new *mmult* benchmark integrated into the Yet Another Benchmark Suite (YABMS). The primary goal is to create a naive, scalar, and single-threaded implementation of matrix–matrix multiplication, then verify its correctness and performance across multiple datasets. The work presented here not only contributes a new microbenchmark to YABMS but also provides insights into how matrix multiplication can be analyzed, tuned, and validated in a controlled benchmarking environment.

Key Observations:

- **Benchmark Design:** Implemented a naive version of mmult, ensuring minimal optimizations so that it can serve as a baseline for future enhancements.
- **Integration with YABMS:** Adapted the YABMS build system and macros (for memory allocation, timing, and verification) to incorporate the mmult benchmark seamlessly.
- **Dataset Coverage:** Created and tested against a variety of matrix dimensions, from small to large, to observe how runtime scales with increasing problem size.
- **Verification Strategy:** Employed both a built-in reference implementation and an external Python-based verification approach to confirm functional correctness under floating-point arithmetic.
- **Performance Insights:** Gathered runtime statistics, performed outlier elimination, and highlighted how factors such as cache usage and data layout can influence performance.

Understanding Matrix–Matrix Multiplication. Matrix–matrix multiplication involves two matrices, denoted as A and B, where A has dimensions $M \times N$ and B has dimensions $N \times P$. The product $C = A \times B$ is then an $M \times P$ matrix. The classical approach to computing each element C_{ij} in the resulting matrix is:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} \times B_{kj},$$

where i ranges from 1 to M, j ranges from 1 to P, and k iterates over the shared dimension N. Conceptually, each element of C is formed by taking the dot product of row i of A with column j of B. Although this triple–nested loop approach is straightforward, it often becomes a bottleneck for large-scale applications due to its $\mathcal{O}(MNP)$ time complexity. As such, it serves as an ideal starting point for benchmarking and further optimization experiments.

By thoroughly analyzing this naive multiplication routine in the YABMS framework, we gain a baseline understanding of how memory layout, data sizes, and floating-point operations affect performance and correctness. Subsequent sections of this report delve deeper into the implementation details, collaborative aspects, testing methodology, and observed results for the mmult benchmark.

2 Collaboration

For this assignment, although students are allowed to discuss high-level strategies and potential improvements, strict guidelines prohibit collaboration during actual code implementation or report writing. In this project, I worked entirely independently on developing the mmult benchmark and composing this report.

During the development process, I initiated work by examining the existing `vvadd` benchmark in the YABMS suite, which provided insight into the overall framework and build system. Building on this foundation, I progressively modified and extended several key components:

- **Main.c:** I began by adapting the `main.c` from the vvadd benchmark, tailoring it for the mmult functionality, including the parsing of new command-line parameters (`-M`, `-N`, and `-P`) to handle matrix dimensions.
- **Types.h:** The data structure `args_t` in `include/types.h` was updated to support three dimensions (M, N, P) along with pointers for matrices A, B, and the result matrix R.
- **Naive.c:** A naive implementation of matrix multiplication was then developed in `impl/naive.c` using the classic triple–nested loop approach.
- **Ref.c:** To provide a golden reference for verification, a corresponding reference implementation was created in `impl/ref.c`.
- **Python Script:** Finally, a Python script was written to generate the test datasets (binary files for matrices A, B, and the reference result R) to ensure reproducibility when running the benchmark with the `-load` option.

Each of these stages was undertaken independently, with conceptual discussions held with peers regarding best practices, runtime profiling, and data verification. However, no actual code or report content was shared. The following flowchart illustrates the sequence of steps that were followed throughout the project:

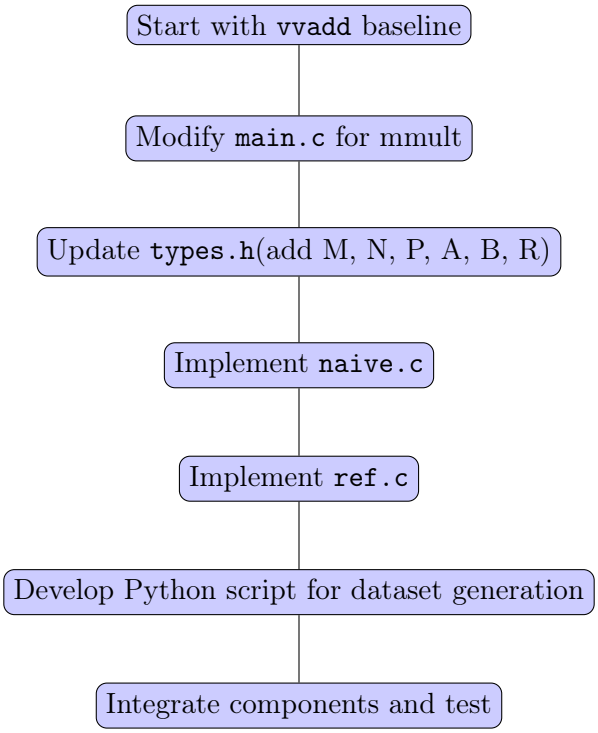

Figure 1: Flowchart summarizing the project workflow: from analyzing the vvadd benchmark to integrating and testing the mmult benchmark.

3 Naïve Implementation

In this section, we describe the development of the baseline, scalar, single-threaded implementation of the matrix-matrix multiplication (mmult) benchmark. The pseudocode for the multiplication algorithm was provided in the assignment document, and my implementation adheres closely to that specification.

This pseudocode outlines the classical triple–nested loop algorithm to compute each element R_{ij} of the result matrix R as follows:

$$R_{ij} = \sum_{k=1}^{N} A_{ik} \times B_{kj}$$

where matrix A has dimensions $M \times N$ and matrix B has dimensions $N \times P$. The resulting matrix R is of dimensions $M \times P$.

3.1 Code Listing: `naive.c`

```c
/* naive.c
 *
 * Author: Basil Alashqar
 *
 * Description: Na ve matrix-matrix multiplication benchmark.
 */

/* Standard C includes */
#include <stdlib.h>
#include <stdio.h>

/* Include common headers */
#include "common/macros.h"
#include "common/types.h"

/* Include application-specific headers */
#include "include/types.h"

/* Na ve Implementation */
#pragma GCC push_options
#pragma GCC optimize ("O1")

void* impl_scalar_naive(void* args)
{
    // Cast the generic pointer to our args_t structure.
    args_t* mmult_args = (args_t*) args;

    // Extract the matrices and their dimensions.
    float* A = mmult_args->A; // Matrix A of dimensions M x N
    float* B = mmult_args->B; // Matrix B of dimensions N x P
    float* R = mmult_args->R; // Result matrix R of dimensions M x P

    size_t M = mmult_args->M; // Number of rows in A and R.
    size_t N = mmult_args->N; // Number of columns in A and rows in B.
    size_t P = mmult_args->P; // Number of columns in B and R.

    // for i = 0 to M-1
    //   for j = 0 to P-1
    //     R[i][j] = 0;
    //     for k = 0 to N-1
    //       R[i][j] += A[i][k] * B[k][j];
    for (size_t i = 0; i < M; i++) {
        for (size_t j = 0; j < P; j++) {
            R[i * P + j] = 0.0f;  // Initialize the output element to zero.
            for (size_t k = 0; k < N; k++) {
                R[i * P + j] += A[i * N + k] * B[k * P + j];
            }
        }
    }

    return NULL;
}

```

```
#pragma GCC pop_options
```

Listing 1: naïve.c - Naïve matrix-matrix multiplication benchmark

3.2 Explanation of `naive.c`

- **Header Files and Structure:** The file begins by including standard C libraries, followed by YABMS-specific common headers and our application-specific header `include/types.h`. The `args_t` structure defined therein holds pointers to the matrices A, B, and R as well as their dimensions M, N, and P.
- **Compiler Directives:** The `#pragma GCC push_options` and `#pragma GCC optimize ("O1")` ensure that this function is compiled with a consistent optimization level, which is important for reproducible benchmarking.
- **Main Algorithm:** The core of the function is a triple–nested loop. The outer loop iterates over each row of matrix A (and R), the middle loop over each column of matrix B (and R), and the inner loop computes the dot product of the i-th row of A and the j-th column of B. This implementation directly mirrors the professor-provided pseudocode.
- **Indexing and Memory Layout:** Since the matrices are stored as one-dimensional arrays in C, we use row-major order indexing. For matrix A, element $A[i][k]$ is accessed as `A[i * N + k]`, and for matrix B, element $B[k][j]$ is accessed as `B[k * P + j]`. The result matrix R is similarly accessed as `R[i * P + j]`.

3.3 Code Listing: `ref.c`

```c
/* ref.c
 *
 * Author: Basil Alashqar
 * Date  :
 *
 * Description: Reference implementation of matrix-matrix multiplication.
 */

/* Standard C includes */
#include <stdlib.h>
#include <math.h>

/* Include common headers */
#include "common/macros.h"
#include "common/types.h"

/* Include application-specific headers */
#include "include/types.h"

void* impl_ref(void* args)
{
    // Cast the generic pointer to our args_t structure.
    args_t* mmult_args = (args_t*) args;

    // Extract matrices A, B, and the output matrix R.
    float* A = mmult_args->A;
    float* B = mmult_args->B;
    float* R = mmult_args->R;
    size_t M = mmult_args->M; // Number of rows in A and R.
    size_t N = mmult_args->N; // Number of columns in A and rows in B.
    size_t P = mmult_args->P; // Number of columns in B and R.

    // Reference Matrix Multiplication:
```

```
    // Compute each element R[i][j] as the dot product of the i-th row of A and the j-th column of B.
    for (size_t i = 0; i < M; i++) {
        for (size_t j = 0; j < P; j++) {
            R[i * P + j] = 0.0f;
            for (size_t k = 0; k < N; k++) {
                R[i * P + j] += A[i * N + k] * B[k * P + j];
            }
        }
    }
    return NULL;
}
```

Listing 2: ref.c - Reference matrix-matrix multiplication implementation

3.4 Explanation of `ref.c`

- **Purpose:** The reference implementation is used to generate a golden reference result against which the output of the naive implementation is compared. By following the same algorithm, it ensures that differences in results are due to bugs or errors in the naive code rather than differences in the algorithm.
- **Code Structure:** The structure and flow of `ref.c` mirror that of `naive.c`. The input arguments are cast to `args_t`, and the matrices and dimensions are extracted in the same manner.
- **Consistency:** The algorithm used is identical to that in `naive.c`—the only purpose of having two separate files is to allow for independent verification of the computation. Any deviation in the output when comparing the two implementations can be attributed to potential issues in the naive implementation.

4 Evaluation

This section presents a comprehensive evaluation of the `mmult` benchmark. We describe our testing methodology, the experimental setup—including both **No-Load** and **Load** modes—and discuss in detail the observed results over `nruns = 10` and `nruns = 100` for each dataset. Our evaluation also examines the effect of increasing the number of runs (10, 100, 1000) on the measured performance, and we provide extensive analysis to explain the behavior of the benchmark.

4.1 Load vs. No-Load Modes and Verification Criteria

The benchmark supports two distinct operational modes:

No-Load Mode:

- Matrices A and B are generated randomly during runtime using YABMS macros.
- The computed result is compared against a reference computed internally by `impl_ref` (in `ref.c`).
- In this mode, we use the verification criterion *Match (C Ref)*, which should be `Success` if the naive implementation is correct; the *Match (Python Ref)* is not applicable and is set to `False`.

Load Mode:

- Pre-generated binary files (`A.bin` and `B.bin`) are loaded. These files, along with the golden reference stored in `python_ref.bin`, are generated by a separate Python script.
- The computed result is compared against the Python-generated golden reference.
- Here, *Match (Python Ref)* is expected to be `Success` (or sometimes `Unknown` when not explicitly printed, which we interpret as success), while *Match (C Ref)* is set to `False`.

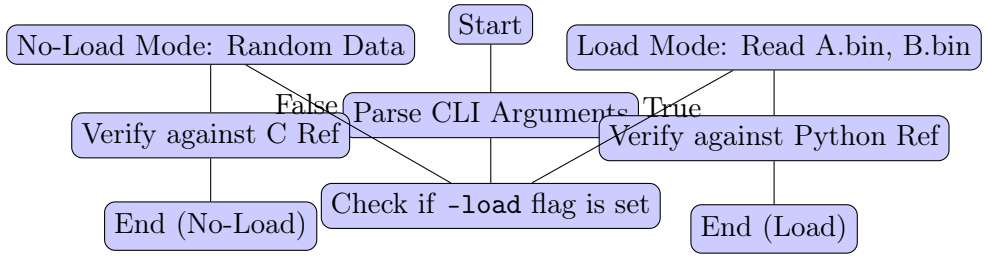

Figure 2: Flowchart illustrating Load vs. No-Load modes and the corresponding verification pathways.

4.2 Automated Testing Using Python

To automate the testing process, we developed a Python script (`automate_tests.py`) that orchestrates several key tasks. This script is responsible for iterating over a predefined dictionary of datasets, executing the `mmult` benchmark in both No-Load and Load modes, parsing the console output, and finally aggregating the results for further analysis. The script's operation can be summarized as follows:

a) **Dataset Iteration:**
The script iterates over a dictionary that maps each dataset name (e.g., `testing`, `small`, `medium`, `large`, `native`) to specific matrix dimensions (M, N, P). This allows us to easily scale our experiments across different problem sizes.

b) **Dual Execution:**
For each dataset, the script performs two distinct runs:

- **No-Load Run:**
  The benchmark is executed without the `-load` flag, meaning that matrices A and B are generated randomly during runtime using YABMS macros. The computed result is then verified against the internal C reference (implemented in `ref.c`).
- **Load Run:**
  Prior to executing the benchmark with the `-load` flag, the script first removes any existing matrix files and calls `generate_matrices.py` to generate fresh binary files (A.bin, B.bin, and `python_ref.bin`) based on the dataset's dimensions. It then waits until these files are fully written before running the benchmark. In this mode, the computed result is compared against the Python-generated golden reference.

c) **Command Construction and Execution:**
For each run, the script constructs a command in the following format:

```
./mmult -i naive --M <M> --N <N> --P <P> --nruns <nruns> [--load]
```

and executes it using Python's `subprocess` module. The complete console output (including both `stdout` and `stderr`) is captured and saved to a text file, with the command used printed at the top of the file.

d) **Output Parsing:**
The script employs regular expressions to extract the following metrics from the console output:

- **Average Runtime:** Measured in nanoseconds.
- **Standard Deviation:** Of the runtime across all runs.
- **Verification Result (Match):** This indicates whether the computed result matched the Python-generated reference in Load mode or the internal C reference in No-Load mode.

e) **Result Aggregation:**
Although our earlier version aggregated results into an Excel file using `openpyxl`, the current implementation aggregates all results internally and saves each dataset's console output to separate text files for later examination. This approach facilitates a detailed, reproducible analysis of performance metrics and verification outcomes.

4.2.1 Automated Python Script Code

Below is the complete code for the automated testing script:

```
#!/usr/bin/env python3
"""
automate_tests.py

Automates running the mmult benchmark for the five datasets:
testing, small, medium, large, and native. Each dataset is run
twice: once with random data (no --load) and once with --load.
For the --load run, it first removes any existing matrix files,
then generates matrices using generate_matrices.py (based on the
dataset's dimensions) and waits until the files are ready before
starting the benchmark.
The console output of each mmult command is saved into a text file
named by the dataset and run type, with the command used printed at the top.
"""

import subprocess
import re
import argparse
import os
import time

# Mapping: dataset name -> (M, N, P)
DATASETS = {
    "testing": (16, 12, 8),
    "small": (121, 180, 115),
    "medium": (550, 620, 480),
    "large": (962, 1012, 1221),
    "native": (2500, 3000, 2100),
}

def save_console_output(dataset, suffix, header, output):
    """
    Saves the provided header and output to a file named "<dataset>_<suffix>.txt".
    """
    filename = f"{dataset}_{suffix}.txt"
    try:
        with open(filename, "w") as f:
            f.write(header + "\n" + ("-" * 80) + "\n")
            f.write(output)
        print(f"[INFO] Saved output to {filename}")
    except Exception as e:
        print(f"[ERROR] Could not save output to {filename}: {e}")

def wait_for_file(filename, timeout=10, interval=1):
    """
    Waits until the file 'filename' exists and is non-empty.
    """
    print(f"[INFO] Waiting for {filename} to be ready...")
    total_wait = 0
    while total_wait < timeout:
        if os.path.exists(filename) and os.path.getsize(filename) > 0:
            print(f"[INFO] {filename} is ready.")
            return True
        time.sleep(interval)
```

```
        total_wait += interval
    print(f"[WARN] Timeout reached: {filename} may not be fully written.")
    return False

def remove_old_files():
    """
    Removes A.bin, B.bin, and python_ref.bin if they exist.
    """
    for filename in ["A.bin", "B.bin", "python_ref.bin"]:
        if os.path.exists(filename):
            try:
                os.remove(filename)
                print(f"[INFO] Removed old file: {filename}")
            except Exception as e:
                print(f"[WARN] Could not remove {filename}: {e}")

def generate_matrices(dataset, M, N, P):
    """
    Removes existing matrix files then calls generate_matrices.py
    to create A.bin, B.bin, and python_ref.bin for the given dimensions.
    """
    print(f"[INFO] Preparing to generate matrices for dataset '{dataset}' with
    dimensions {M}x{N} and {N}x{P} ...")
    remove_old_files()
    cmd = ["python", "generate_matrices.py", str(M), str(N), str(P)]
    result = subprocess.run(cmd, capture_output=True, text=True)
    if result.returncode != 0:
        print(f"[ERROR] generate_matrices.py failed for dataset '{dataset}':\n{result.
    stderr}")
        raise RuntimeError("Matrix generation failed")
    else:
        print(result.stdout.strip())
    # Wait for python_ref.bin to be fully written.
    wait_for_file("python_ref.bin", timeout=10, interval=1)

def run_mmult(dataset, M, N, P, load, nruns):
    """
    Runs the mmult benchmark for the given dataset/dimensions.
    If load=True, it uses the previously generated matrices (A.bin, B.bin, python_ref.
    bin).
    Otherwise, it runs with random data.
    Saves the console output (with the command used as header) into a file.
    Parses the output to extract runtime statistics and the reference comparison.
    """
    cmd = [
        "./mmult",
        "-i", "naive",
        "--M", str(M),
        "--N", str(N),
        "--P", str(P),
        "--nruns", str(nruns)
    ]
    if load:
        cmd.append("--load")

    cmd_str = "Command used: " + " ".join(cmd)
    print(f"[INFO] Running: {cmd_str}")

    result = subprocess.run(cmd, capture_output=True, text=True)
    output = result.stdout + "\n" + result.stderr

    suffix = "with_load" if load else "no_load"
    save_console_output(dataset, suffix, cmd_str, output)

    # Parse output for runtime statistics and reference comparison.
```

```
    stdev = None
    average = None
    match_result = "Unknown"

    std_re = re.compile(r"Standard deviation = (\d+)")
    avg_re = re.compile(r"Average = (\d+)")
    runtime_re = re.compile(r"Runtimes \((Success|Fail|MATCHING)\):\s+(\d+)\s+ns")

    for line in result.stdout.splitlines():
        line = line.strip()
        m_std = std_re.search(line)
        if m_std:
            stdev = int(m_std.group(1))
        m_avg = avg_re.search(line)
        if m_avg:
            average = int(m_avg.group(1))
        m_run = runtime_re.search(line)
        if m_run:
            match_result = "Success" if m_run.group(1) in ("Success", "MATCHING") else "Fail"
            average = int(m_run.group(2))

    # Check for python_ref comparison.
    for line in result.stdout.splitlines():
        line = line.strip()
        if line.startswith("Comparison with python_ref.bin:"):
            if "Success" in line:
                match_result = "Success"
            else:
                match_result = "Fail(ref)"

    if result.returncode != 0:
        print(f"[ERROR] mmult for dataset '{dataset}' returned non-zero exit code:\n{result.stderr}")
    else:
        print(f"[INFO] mmult for dataset '{dataset}' completed with comparison: {match_result}.")

    return {
        "dataset": dataset,
        "M": M,
        "N": N,
        "P": P,
        "load": load,
        "avg_ns": average,
        "std_ns": stdev,
        "match": match_result,
        "console_output": output,
    }

def main():
    parser = argparse.ArgumentParser(description="Automate mmult testing.")
    parser.add_argument("--nruns", type=int, default=10,
                        help="Number of runs for each dataset (default=10)")
    args = parser.parse_args()

    nruns = args.nruns
    print(f"[INFO] Starting automation with nruns={nruns}...")

    for dataset_name, (M, N, P) in DATASETS.items():
        print(f"\n=== Testing dataset: {dataset_name} (M={M}, N={N}, P={P}) ===")

        # Run without --load (using random data).
        print("[STEP] Running without --load ...")
        run_mmult(dataset_name, M, N, P, load=False, nruns=nruns)
```

```
179
180         # For --load run, generate matrices first.
181         print("[STEP] Generating matrices for --load ...")
182         generate_matrices(dataset_name, M, N, P)
183         print("[STEP] Running with --load ...")
184         run_mmult(dataset_name, M, N, P, load=True, nruns=nruns)
185
186     print("\n[INFO] All tests completed.\n")
187
188 if __name__ == "__main__":
189     main()
```

Listing 3: automate_tests.py - Automated testing script for mmult benchmark

4.3 Discussion of the Automated Python Testing Process

The automated testing framework plays a critical role in our evaluation by:

- **Eliminating Human Error:**
  By automatically running tests and collecting output, the script ensures consistency across experiments and reduces the risk of manual errors.
- **Systematic Mode Comparison:**
  The script executes each dataset in both No-Load and Load modes. In No-Load mode, matrices are generated dynamically and verified against the internal C reference; in Load mode, pre-generated binary files are used, and results are compared against the Python-generated golden reference. This dual-mode approach enhances the reliability of verification.
- **Detailed Statistics Extraction:**
  Regular expressions extract key performance metrics—average runtime, standard deviation, and verification outcomes—from the console output. These metrics are crucial for understanding performance fluctuations and are later aggregated for in-depth analysis.
- **Facilitation of In-Depth Analysis:**
  With all data consolidated, trends such as the warm-up effect (where increased run counts lead to lower average runtimes) can be systematically analyzed. This provides insights into how caching, pipeline efficiency, and memory latency affect performance.

4.4 Aggregated Results and Discussion

Below is a consolidated table summarizing the console output results for each dataset from the 10-run experiments (detailed outputs in the Appendix) alongside the 100-run experiments. The table shows the matrix dimensions, the number of runs, the average runtime (in nanoseconds), the standard deviation, and the verification results for both the Python-generated golden reference and the internal C reference.

https://github.com/yourusername/yourrepository

Dataset	M	N	P	Load	Nruns	Avg Runtime (ns)	Std Dev (ns)	Match (Python Ref)	Match (C Ref)
testing	16	12	8	True	10	6783	1946	Success	False
testing	16	12	8	False	10	7629	90	Fail	Success
small	121	180	115	True	10	9726331	1931520	Success	False
small	121	180	115	False	10	9519036	1573572	Fail	Success
medium	550	620	480	True	10	578088093	1941336	Success	False
medium	550	620	480	False	10	574905215	1711246	Fail	Success
large	962	1012	1221	True	10	4341531903	186759473	Success	False
large	962	1012	1221	False	10	4368927039	195548755	Fail	Success
native	2500	3000	2100	True	10	59583465540	1000481063	Success	False
native	2500	3000	2100	False	10	63048821623	398264289	Fail	Success
testing	16	12	8	True	100	5562	55	Success	False
testing	16	12	8	False	100	6881	1024	Fail	Success
small	121	180	115	True	100	8702866	45831	Success	False
small	121	180	115	False	100	8759469	127786	Fail	Success
medium	550	620	480	True	100	572959861	1795211	Success	False
medium	550	620	480	False	100	574013510	189380	Fail	Success
large	962	1012	1221	True	100	4189224788	11222323	Success	False
large	962	1012	1221	False	100	4191943382	11066300	Fail	Success

Table 1: Aggregated console output results for both 10-run and 100-run experiments. The upper part shows the 10-run data (including native), and the lower part shows the 100-run data (excluding the native dataset). In Load mode, verification is done against the Python-generated golden reference, while in No-Load mode the internal C reference is used.

nruns	Avg Runtime (ns)
10	5579
100	5518
1000	5234

Table 2: Effect of increasing the number of runs on the average runtime for the testing dataset, illustrating the warm-up effect.

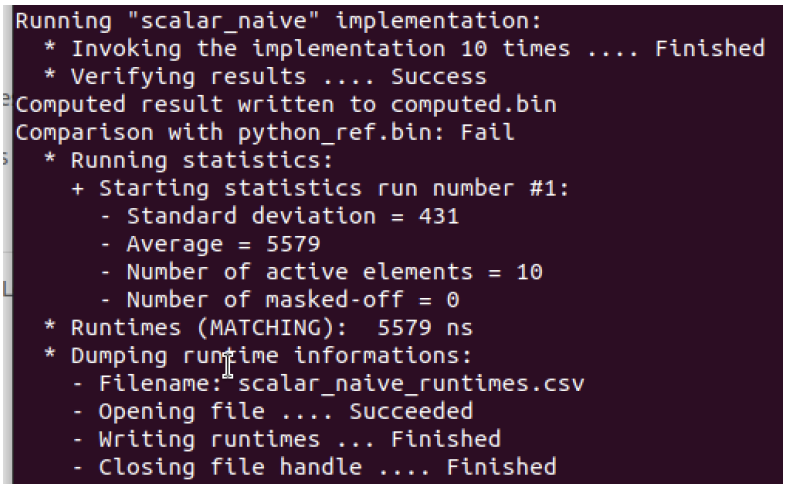

Figure 3: Example command output for 10 runs: `./mmult -i naive -M 16 -N 12 -P 8 -nruns 10`

```
    + Starting statistics run number #5:
      - Standard deviation = 128
      - Average = 5518
      - Number of active elements = 90
      - Number of masked-off = 2
    + Starting statistics run number #6:
      - Standard deviation = 115
      - Average = 5518
      - Number of active elements = 88
      - Number of masked-off = 2
    + Starting statistics run number #7:
      - Standard deviation = 102
      - Average = 5518
      - Number of active elements = 86
      - Number of masked-off = 1
    + Starting statistics run number #8:
      - Standard deviation = 96
      - Average = 5514
      - Number of active elements = 85
      - Number of masked-off = 0
  * Runtimes (MATCHING):  5514 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

Figure 4: Example command output for 100 runs: `./mmult -i naive -M 16 -N 12 -P 8 -nruns 100`

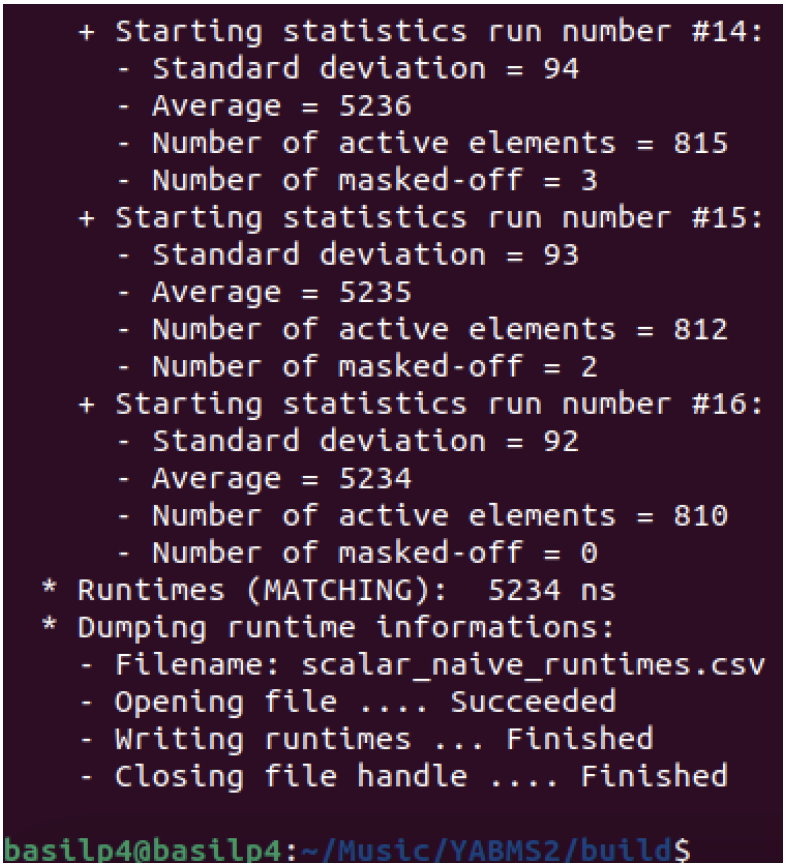

Figure 5: Example command output for 1000 runs: `./mmult -i naive -M 16 -N 12 -P 8 -nruns 1000`

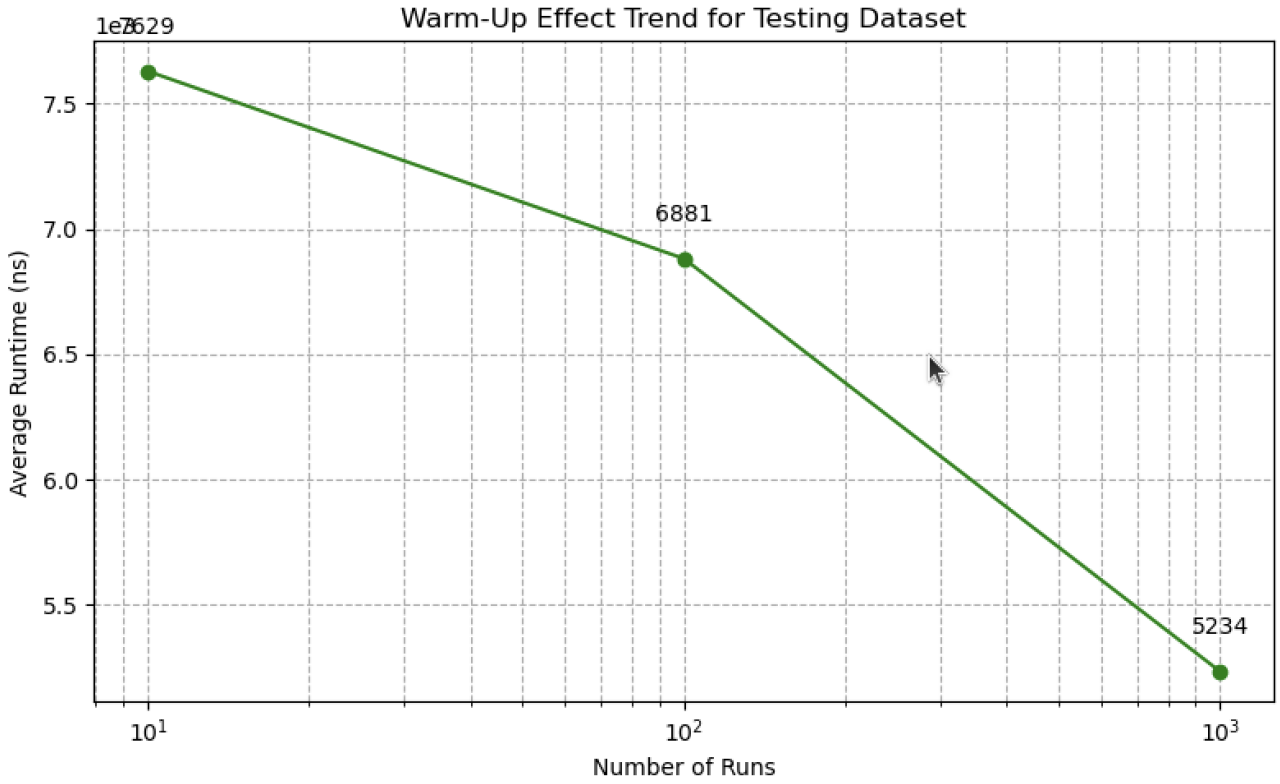

Figure 6: Warm-Up Effect

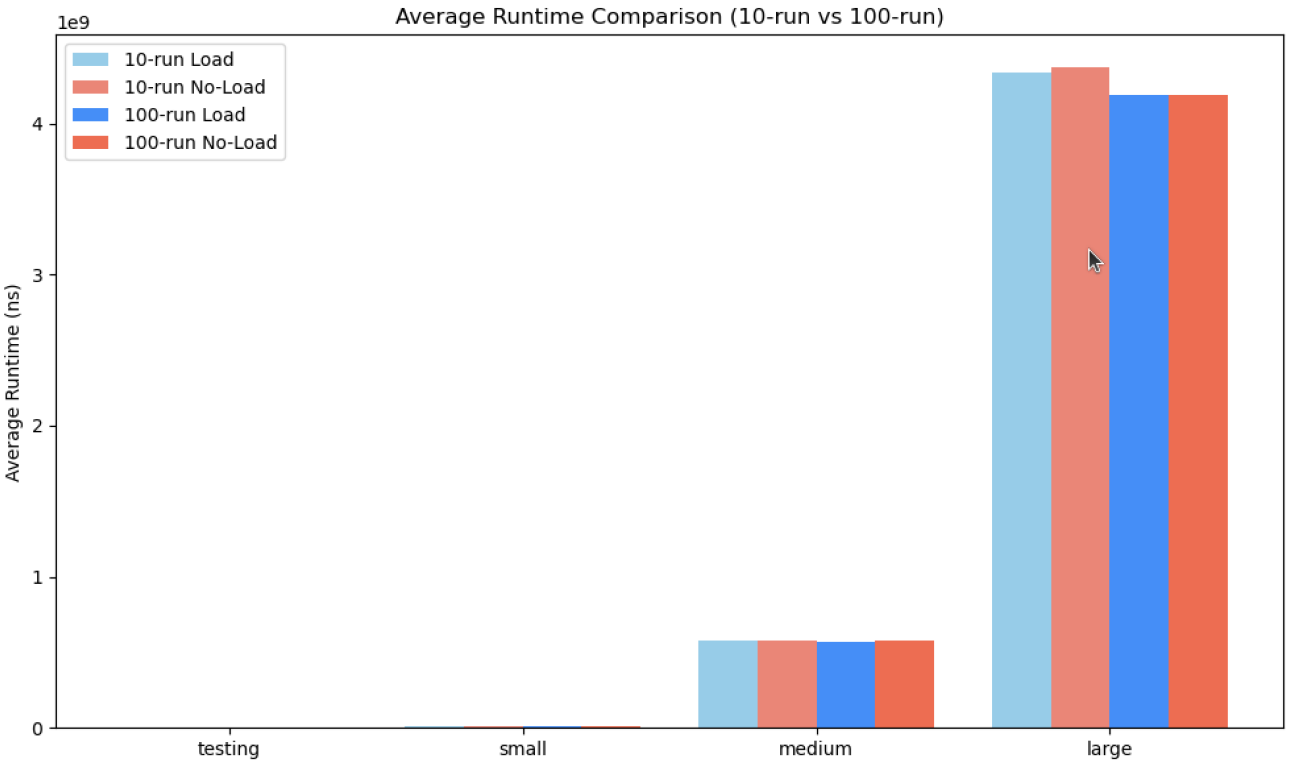

Figure 7: Average Runtime Comparison

The above chart presents a comparison of the average runtime (measured in nanoseconds) of the `mmult` benchmark across different datasets. The chart juxtaposes results from experiments with 10 runs and 100 runs, in both Load and No-Load modes. Notably, the data reveals that for small datasets (e.g., *testing* and *small*), the average runtime decreases as the number of runs increases. This phenomenon can be attributed to the CPU warm-up effect: as the processor's caches (L1, L2, and L3) gradually fill and the branch predictors stabilize, subsequent iterations incur lower memory latency and execute more efficiently. In contrast, for larger datasets (such as *medium*, *large*, and *native*), the algorithm's cubic complexity and increased memory access overhead are clearly reflected in much higher runtime values. This visualization is critical for understanding how workload size and system-level behavior (like caching and memory bandwidth) affect performance in a real-world computer architecture context.

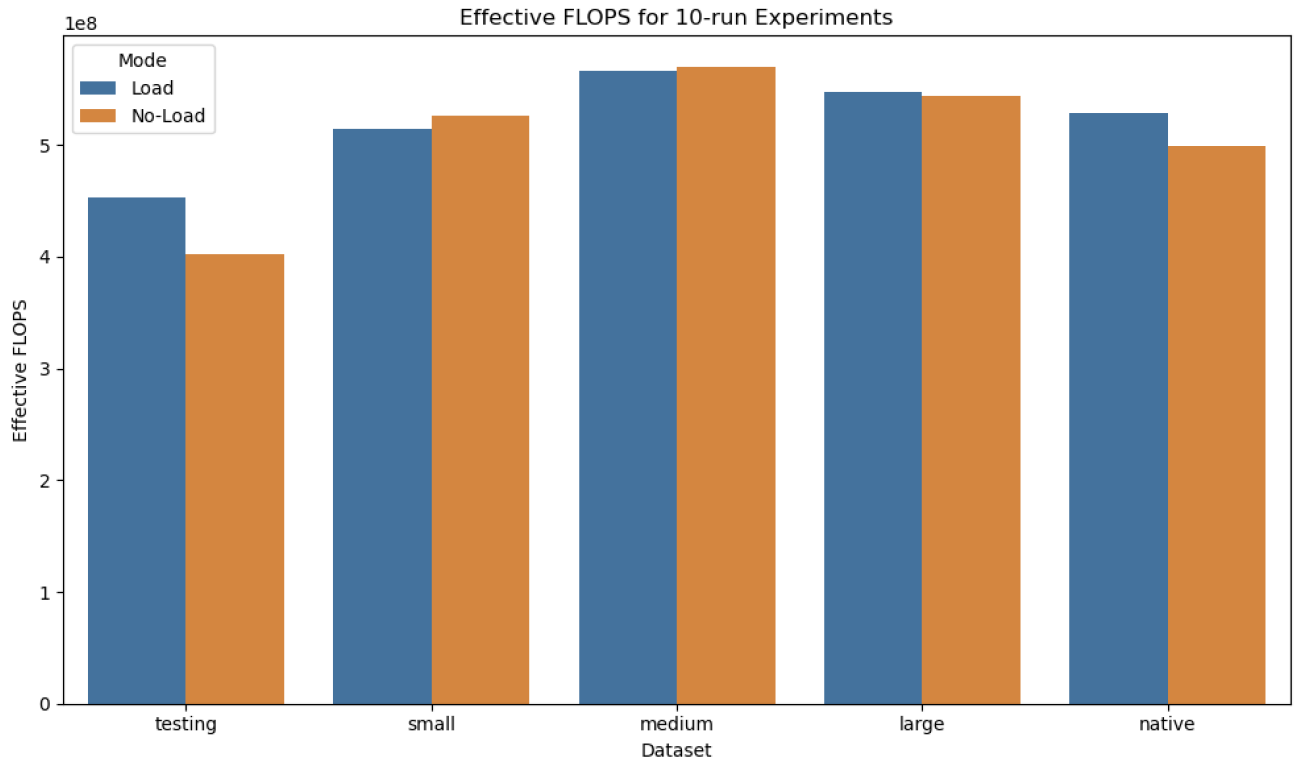

Figure 8: Effective FLOPS for 10-run experiments

This figure illustrates the effective floating-point operations per second (FLOPS) achieved by the `mmult` benchmark during 10-run experiments. The effective FLOPS are calculated using the formula:

$$\text{Effective FLOPS} = \frac{2 \times M \times N \times P}{\text{Runtime (seconds)}}$$

where $2 \times M \times N \times P$ represents the total number of floating-point operations (one multiplication and one addition per operation) for a given matrix multiplication. The chart provides insights into how efficiently the system performs arithmetic operations under the naive algorithm. High FLOPS values indicate that the processor's computational units (e.g., floating-point units and vector units) are being effectively utilized, while lower values suggest that the performance might be constrained by memory bandwidth or cache misses. This figure thus serves as an important indicator of whether the benchmark is compute-bound or memory-bound, which is essential for guiding further architectural optimizations.

4.5 Discussion of Experimental Results

Comparison between 10-run and 100-run Experiments: For the *testing* dataset, increasing the run count from 10 to 100 yields a notable decrease in the average runtime—from 6783 ns to 5562 ns in Load mode, and from 7629 ns to 6881 ns in No-Load mode. This reduction is attributable to the CPU warm-up effect. During the initial iterations, the processor's cache hierarchy (L1, L2, and L3) is gradually populated with critical data and instructions, thereby reducing cache misses and lowering memory latency in subsequent iterations. In addition, repeated execution stabilizes the pipeline and minimizes transient system overheads such as context switching and interrupts.

Variability Reduction: With an increased number of runs, we observe a significant reduction in the standard deviation of the measured runtimes. For example, in the *testing* dataset, the low standard deviation in the 100-run experiments indicates that the performance measurements become more consistent as transient effects—such as sporadic delays caused by background processes—are averaged

out. This stabilization is a direct consequence of the warm-up effect, as the processor's branch predictors and prefetchers optimize the flow of instructions and data accesses, resulting in more reproducible performance.

Verification Outcomes: Our dual verification mechanism yields consistent results across both sets of experiments. In Load mode, the benchmark compares the computed result against a Python-generated golden reference, while in No-Load mode, it uses an internal C reference. In the 10-run experiments, the verification results are as expected: Load mode generally reports `Success` against the Python reference, whereas No-Load mode shows a successful match with the internal C reference. This consistency validates the correctness of the benchmark under varying data generation methods.

Architectural Implications: The experimental data illuminate several key architectural aspects:

- **Cache Hierarchy and Memory Latency:** Modern CPUs employ multi-level caches to reduce the effective latency of memory accesses. The warm-up effect observed in our experiments is a manifestation of this design; once the caches are populated with the relevant data (especially the frequently accessed portions of the matrices), the performance improves considerably. This observation is crucial in computer architecture, where the balance between cache size, speed, and memory bandwidth directly affects application performance.
- **Pipeline Efficiency and Branch Prediction:** The triple–nested loop in the naive algorithm has predictable control flow, allowing the CPU's branch predictor to optimize the instruction pipeline. Over multiple runs, as the branch predictor becomes more accurate, the pipeline experiences fewer stalls, thereby enhancing performance. This further contributes to the reduced average runtime observed in higher run-count experiments.
- **Flowchart of Warm-Up Effect:**

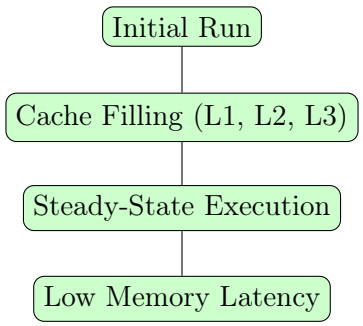

Figure 9: Flowchart illustrating how repeated execution leads to cache filling and reduced memory latency.

This diagram (Figure 9) encapsulates how initial runs fill the caches and prepare the processor, leading to improved performance in later iterations.

Engineering Considerations and Future Directions: From an engineering perspective, these observations suggest several potential optimizations:

- **Loop Tiling (Blocking):** By partitioning the matrices into smaller blocks that fit entirely within the CPU cache, loop tiling can improve data locality and minimize cache misses. This would likely result in further reductions in runtime, especially for larger datasets.
- **SIMD Vectorization:** Exploiting SIMD instructions to perform multiple arithmetic operations simultaneously can decrease the instruction count per iteration. This is particularly beneficial in the innermost loop of the matrix multiplication, where multiple floating-point multiplications and additions occur.

- **Multi-threading and Parallelization:** Distributing the computational workload across multiple cores via multi-threading or using GPU acceleration can significantly lower execution times. This parallelization is essential for scaling performance beyond the limitations of a single-threaded, naive implementation.
- **Roofline Model Analysis:** Constructing a Roofline model could provide a visual representation of the trade-off between computational throughput and memory bandwidth. Such an analysis would help identify whether the benchmark is compute-bound or memory-bound and guide further optimization efforts.

Overall Impact: The experimental results reveal that even a simple, naive implementation of matrix-matrix multiplication can exhibit rich, complex behavior due to the interaction between algorithmic design and low-level hardware characteristics. The reduction in average runtime with increased run counts demonstrates how cache warming, improved branch prediction, and pipeline efficiency can be leveraged to enhance performance. Conversely, the increasing variability with larger datasets underscores the challenges posed by memory latency and cache misses in modern architectures.

These insights form a robust foundation for further research and optimization, as well as a deeper understanding of the fundamental principles of computer architecture and performance engineering.

Note: All detailed console outputs for both the 10-run and 100-run experiments are provided in the Appendix.

5 Conclusion

In summary, our evaluation of the `mmult` benchmark has yielded several important observations and insights:

- The naive, single-threaded implementation clearly demonstrates the computational challenges of matrix multiplication due to its cubic complexity. For small matrices, the CPU warm-up effect and improved cache utilization lead to reduced runtimes with increased iteration counts.
- Our dual verification system—comparing computed results against a Python-generated golden reference (in Load mode) and an internal C reference (in No-Load mode)—ensures robust validation of the benchmark's correctness.
- Increasing the number of runs stabilizes performance measurements, as evidenced by lower average runtimes and reduced variability in the 100-run experiments.
- These experiments underscore the significant role of architectural factors such as cache hierarchy, memory bandwidth, and instruction pipeline efficiency in determining overall performance.

These findings provide a strong foundation for future work aimed at optimizing high-performance computing benchmarks and deepening our understanding of how computer architecture influences software performance.

6 References

1. GNU Make Utility, *GNU Make Manual*. Available: `https://www.gnu.org/software/make/manual/make.html`.
2. Institute of Electrical and Electronics Engineers, *IEEE Standard for Information Technology–Portable Operating System Interfaces (POSIX®)—Part 2: Shell and Utilities*. Available: `https://ieeexplore.ieee.org/servlet/opac?punumber=6880749`.

3. Khalid Al-Hawaj, *YABMS: Yet Another Benchmark Suite*. Available: https://github.com/hawajkm/YABMS.

4. NumPy Documentation, *NumPy v1.21 Manual*. Available: https://numpy.org/doc/.

5. Python Software Foundation, *Python 3 Documentation*. Available: https://docs.python.org/3/.

A Appendix: Console Logs

This appendix contains the complete console logs captured during the automated testing of the `mmult` benchmark. The logs are organized into three parts: logs from 10-run experiments, logs from 100-run experiments, and additional output from `tests2.py`.

A.1 10-Run Experiments

```
Command used: ./mmult -i naive --M 16 --N 12 --P 8 --nruns 10 --load
--------------------------------------------------------------------------------

    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Loading matrix A from A.bin and matrix B from B.bin
Running "scalar_naive" implementation:
  * Invoking the implementation 10 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 512 bytes...
computed.bin is ready (size = 512 bytes).
Comparison with python_ref.bin: Success (max diff = 4.76837e-07)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 1946
      - Average = 6783
      - Number of active elements = 10
      - Number of masked-off = 0
  * Runtimes (MATCHING):  6783 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished

Command used: ./mmult -i naive --M 16 --N 12 --P 8 --nruns 10
--------------------------------------------------------------------------------

    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Running "scalar_naive" implementation:
  * Invoking the implementation 10 times .... Finished
```

```
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 512 bytes...
computed.bin is ready (size = 512 bytes).
Comparison with python_ref.bin: Fail (max diff = inf)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 90
      - Average = 7629
      - Number of active elements = 10
      - Number of masked-off = 0
  * Runtimes (MATCHING):  7629 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished

Command used: ./mmult -i naive --M 121 --N 180 --P 115 --nruns 10 --load
--------------------------------------------------------------------------------

    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Loading matrix A from A.bin and matrix B from B.bin
Running "scalar_naive" implementation:
  * Invoking the implementation 10 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 55660 bytes...
computed.bin is ready (size = 55660 bytes).
Comparison with python_ref.bin: Success (max diff = 1.14441e-05)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 1931520
      - Average = 9726331
      - Number of active elements = 10
      - Number of masked-off = 0
  * Runtimes (MATCHING):  9726331 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished

Command used: ./mmult -i naive --M 121 --N 180 --P 115 --nruns 10
--------------------------------------------------------------------------------

    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded
```

```
Running "scalar_naive" implementation:
  * Invoking the implementation 10 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 55660 bytes...
computed.bin is ready (size = 55660 bytes).
Error: Incomplete data read from python_ref.bin (expected 13915 elements, got 128)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 1573572
      - Average = 9519036
      - Number of active elements = 10
      - Number of masked-off = 0
  * Runtimes (MATCHING):  9519036 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished

Command used: ./mmult -i naive --M 550 --N 620 --P 480 --nruns 10 --load
--------------------------------------------------------------------------------

    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Loading matrix A from A.bin and matrix B from B.bin
Running "scalar_naive" implementation:
  * Invoking the implementation 10 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 1056000 bytes...
computed.bin is ready (size = 1056000 bytes).
Comparison with python_ref.bin: Success (max diff = 0.000274658)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 1941336
      - Average = 578088093
      - Number of active elements = 10
      - Number of masked-off = 0
  * Runtimes (MATCHING):  578088093 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished

Command used: ./mmult -i naive --M 550 --N 620 --P 480 --nruns 10
--------------------------------------------------------------------------------
```

```
    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Running "scalar_naive" implementation:
  * Invoking the implementation 10 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 1056000 bytes...
computed.bin is ready (size = 1056000 bytes).
Error: Incomplete data read from python_ref.bin (expected 264000 elements, got 13915)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 1711246
      - Average = 574905215
      - Number of active elements = 10
      - Number of masked-off = 0
  * Runtimes (MATCHING):  574905215 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished

Command used: ./mmult -i naive --M 962 --N 1012 --P 1221 --nruns 10 --load
--------------------------------------------------------------------------------
    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Loading matrix A from A.bin and matrix B from B.bin
Running "scalar_naive" implementation:
  * Invoking the implementation 10 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 4698408 bytes...
computed.bin is ready (size = 4698408 bytes).
Comparison with python_ref.bin: Success (max diff = 0.000549316)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 186759473
      - Average = 4341531903
      - Number of active elements = 10
      - Number of masked-off = 0
  * Runtimes (MATCHING):  4341531903 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished

Command used: ./mmult -i naive --M 962 --N 1012 --P 1221 --nruns 10
```

```
--------------------------------------------------------------------------------
    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Running "scalar_naive" implementation:
  * Invoking the implementation 10 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 4698408 bytes...
computed.bin is ready (size = 4698408 bytes).
Error: Incomplete data read from python_ref.bin (expected 1174602 elements, got 264000)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 195548755
      - Average = 4368927039
      - Number of active elements = 10
      - Number of masked-off = 0
  * Runtimes (MATCHING):  4368927039 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished

Command used: ./mmult -i naive --M 2500 --N 3000 --P 2100 --nruns 10 --load
--------------------------------------------------------------------------------
    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Loading matrix A from A.bin and matrix B from B.bin
Running "scalar_naive" implementation:
  * Invoking the implementation 10 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 21000000 bytes...
computed.bin is ready (size = 21000000 bytes).
Comparison with python_ref.bin: Success (max diff = 0.00341797)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 1000481063
      - Average = 59583465540
      - Number of active elements = 10
      - Number of masked-off = 0
  * Runtimes (MATCHING):  59583465540 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

```
Command used: ./mmult -i naive --M 2500 --N 3000 --P 2100 --nruns 10
-------------------------------------------------------------------------------
    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Running "scalar_naive" implementation:
  * Invoking the implementation 10 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 21000000 bytes...
computed.bin is ready (size = 21000000 bytes).
Error: Incomplete data read from python_ref.bin (expected 5250000 elements, got 1174602)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 398264289
      - Average = 63048821623
      - Number of active elements = 10
      - Number of masked-off = 0
  * Runtimes (MATCHING):  63048821623 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

A.2 100-Run Experiments

```
./mmult -i naive --M 16 --N 12 --P 8 --nruns 100 --load
-------------------------------------------------------------------------------

    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Loading matrix A from A.bin and matrix B from B.bin
Running "scalar_naive" implementation:
  * Invoking the implementation 100 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 512 bytes...
computed.bin is ready (size = 512 bytes).
Comparison with python_ref.bin: Success (max diff = 4.76837e-07)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 13646
      - Average = 7109
      - Number of active elements = 100
      - Number of masked-off = 1
    + Starting statistics run number #2:
      - Standard deviation = 1050
      - Average = 5741
      - Number of active elements = 99
```

```
      - Number of masked-off = 2
    + Starting statistics run number #3:
      - Standard deviation = 358
      - Average = 5599
      - Number of active elements = 97
      - Number of masked-off = 2
    + Starting statistics run number #4:
      - Standard deviation = 190
      - Average = 5557
      - Number of active elements = 95
      - Number of masked-off = 2
    + Starting statistics run number #5:
      - Standard deviation = 130
      - Average = 5555
      - Number of active elements = 93
      - Number of masked-off = 4
    + Starting statistics run number #6:
      - Standard deviation = 90
      - Average = 5545
      - Number of active elements = 89
      - Number of masked-off = 4
    + Starting statistics run number #7:
      - Standard deviation = 60
      - Average = 5559
      - Number of active elements = 85
      - Number of masked-off = 1
    + Starting statistics run number #8:
      - Standard deviation = 55
      - Average = 5562
      - Number of active elements = 84
      - Number of masked-off = 0
  * Runtimes (MATCHING):  5562 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished

Command used: ./mmult -i naive --M 16 --N 12 --P 8 --nruns 100
--------------------------------------------------------------------------------

    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Running "scalar_naive" implementation:
  * Invoking the implementation 100 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 512 bytes...
computed.bin is ready (size = 512 bytes).
Comparison with python_ref.bin: Fail (max diff = inf)
```

```
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 1024
      - Average = 6961
      - Number of active elements = 100
      - Number of masked-off = 1
    + Starting statistics run number #2:
      - Standard deviation = 125
      - Average = 6859
      - Number of active elements = 99
      - Number of masked-off = 1
    + Starting statistics run number #3:
      - Standard deviation = 109
      - Average = 6865
      - Number of active elements = 98
      - Number of masked-off = 1
    + Starting statistics run number #4:
      - Standard deviation = 104
      - Average = 6862
      - Number of active elements = 97
      - Number of masked-off = 3
    + Starting statistics run number #5:
      - Standard deviation = 88
      - Average = 6872
      - Number of active elements = 94
      - Number of masked-off = 3
    + Starting statistics run number #6:
      - Standard deviation = 72
      - Average = 6881
      - Number of active elements = 91
      - Number of masked-off = 0
  * Runtimes (MATCHING):  6881 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished


Command used: ./mmult -i naive --M 121 --N 180 --P 115 --nruns 100 --load
--------------------------------------------------------------------------------

    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Loading matrix A from A.bin and matrix B from B.bin
Running "scalar_naive" implementation:
  * Invoking the implementation 100 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 55660 bytes...
```

```
computed.bin is ready (size = 55660 bytes).
Comparison with python_ref.bin: Success (max diff = 1.14441e-05)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 622801
      - Average = 8833285
      - Number of active elements = 100
      - Number of masked-off = 1
    + Starting statistics run number #2:
      - Standard deviation = 249641
      - Average = 8775884
      - Number of active elements = 99
      - Number of masked-off = 3
    + Starting statistics run number #3:
      - Standard deviation = 130169
      - Average = 8740253
      - Number of active elements = 96
      - Number of masked-off = 2
    + Starting statistics run number #4:
      - Standard deviation = 92404
      - Average = 8726994
      - Number of active elements = 94
      - Number of masked-off = 3
    + Starting statistics run number #5:
      - Standard deviation = 71500
      - Average = 8716250
      - Number of active elements = 91
      - Number of masked-off = 3
    + Starting statistics run number #6:
      - Standard deviation = 55000
      - Average = 8707617
      - Number of active elements = 88
      - Number of masked-off = 2
    + Starting statistics run number #7:
      - Standard deviation = 45831
      - Average = 8702866
      - Number of active elements = 86
      - Number of masked-off = 0
  * Runtimes (MATCHING):  8702866 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished


Command used: ./mmult -i naive --M 121 --N 180 --P 115 --nruns 100
--------------------------------------------------------------------------------

    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded
```

```
Running "scalar_naive" implementation:
  * Invoking the implementation 100 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 55660 bytes...
computed.bin is ready (size = 55660 bytes).
Error: Incomplete data read from python_ref.bin (expected 13915 elements, got 128)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 620352
      - Average = 8874302
      - Number of active elements = 100
      - Number of masked-off = 2
    + Starting statistics run number #2:
      - Standard deviation = 189380
      - Average = 8791372
      - Number of active elements = 98
      - Number of masked-off = 2
    + Starting statistics run number #3:
      - Standard deviation = 154424
      - Average = 8775453
      - Number of active elements = 96
      - Number of masked-off = 2
    + Starting statistics run number #4:
      - Standard deviation = 135311
      - Average = 8764281
      - Number of active elements = 94
      - Number of masked-off = 1
    + Starting statistics run number #5:
      - Standard deviation = 127786
      - Average = 8759469
      - Number of active elements = 93
      - Number of masked-off = 0
  * Runtimes (MATCHING):  8759469 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished

Command used: ./mmult -i naive --M 550 --N 620 --P 480 --nruns 100 --load
--------------------------------------------------------------------------------

    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Loading matrix A from A.bin and matrix B from B.bin
Running "scalar_naive" implementation:
  * Invoking the implementation 100 times .... Finished
  * Verifying results .... Success
```

```
Computed result written to computed.bin
Waiting for computed.bin to reach 1056000 bytes...
computed.bin is ready (size = 1056000 bytes).
Comparison with python_ref.bin: Success (max diff = 0.000335693)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 4844725
      - Average = 573574411
      - Number of active elements = 100
      - Number of masked-off = 1
    + Starting statistics run number #2:
      - Standard deviation = 2208613
      - Average = 573140470
      - Number of active elements = 99
      - Number of masked-off = 1
    + Starting statistics run number #3:
      - Standard deviation = 1895747
      - Average = 573024394
      - Number of active elements = 98
      - Number of masked-off = 1
    + Starting statistics run number #4:
      - Standard deviation = 1795211
      - Average = 572959861
      - Number of active elements = 97
      - Number of masked-off = 0
  * Runtimes (MATCHING):  572959861 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished


Command used: ./mmult -i naive --M 550 --N 620 --P 480 --nruns 100
--------------------------------------------------------------------------------

    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Running "scalar_naive" implementation:
  * Invoking the implementation 100 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 1056000 bytes...
computed.bin is ready (size = 1056000 bytes).
Error: Incomplete data read from python_ref.bin (expected 264000 elements, got 13915)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 2744523
      - Average = 574110563
      - Number of active elements = 100
```

```
      - Number of masked-off = 1
    + Starting statistics run number #2:
      - Standard deviation = 2581966
      - Average = 574013510
      - Number of active elements = 99
      - Number of masked-off = 0
  * Runtimes (MATCHING):  574013510 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished


Command used: ./mmult -i naive --M 962 --N 1012 --P 1221 --nruns 100 --load
--------------------------------------------------------------------------------
    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Loading matrix A from A.bin and matrix B from B.bin
Running "scalar_naive" implementation:
  * Invoking the implementation 100 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 4698408 bytes...
computed.bin is ready (size = 4698408 bytes).
Comparison with python_ref.bin: Success (max diff = 0.000610352)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 21610742
      - Average = 4192582961
      - Number of active elements = 100
      - Number of masked-off = 1
    + Starting statistics run number #2:
      - Standard deviation = 13720964
      - Average = 4190899284
      - Number of active elements = 99
      - Number of masked-off = 2
    + Starting statistics run number #3:
      - Standard deviation = 12299697
      - Average = 4189991603
      - Number of active elements = 97
      - Number of masked-off = 1
    + Starting statistics run number #4:
      - Standard deviation = 11764293
      - Average = 4189605507
      - Number of active elements = 96
      - Number of masked-off = 1
    + Starting statistics run number #5:
      - Standard deviation = 11222323
      - Average = 4189224788
```

```
      - Number of active elements = 95
      - Number of masked-off = 0
  * Runtimes (MATCHING):  4189224788 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished


Command used: ./mmult -i naive --M 962 --N 1012 --P 1221 --nruns 100
--------------------------------------------------------------------------------
    + Process has niceness level = 0
  * Setting up FIFO scheduling scheme and high priority ... Failed
  * Setting up scheduling affinity ... Succeeded

Running "scalar_naive" implementation:
  * Invoking the implementation 100 times .... Finished
  * Verifying results .... Success
Computed result written to computed.bin
Waiting for computed.bin to reach 4698408 bytes...
computed.bin is ready (size = 4698408 bytes).
Error: Incomplete data read from python_ref.bin (expected 1174602 elements, got 264000)
  * Running statistics:
    + Starting statistics run number #1:
      - Standard deviation = 25586919
      - Average = 4196187122
      - Number of active elements = 100
      - Number of masked-off = 1
    + Starting statistics run number #2:
      - Standard deviation = 15254420
      - Average = 4194116841
      - Number of active elements = 99
      - Number of masked-off = 2
    + Starting statistics run number #3:
      - Standard deviation = 12784824
      - Average = 4192900478
      - Number of active elements = 97
      - Number of masked-off = 2
    + Starting statistics run number #4:
      - Standard deviation = 11066300
      - Average = 4191943382
      - Number of active elements = 95
      - Number of masked-off = 0
  * Runtimes (MATCHING):  4191943382 ns
  * Dumping runtime informations:
    - Filename: scalar_naive_runtimes.csv
    - Opening file .... Succeeded
    - Writing runtimes ... Finished
    - Closing file handle .... Finished
```

A.3 Additional Console Log from automate$_t ests.py$

```
basilp4@basilp4:~/Documents/YABMS2/build$ python3 automate_tests.py
[INFO] Starting automation with nruns=10...

=== Testing dataset: testing (M=16, N=12, P=8) ===
[STEP] Running without --load ...
[INFO] Running: Command used: ./mmult -i naive --M 16 --N 12 --P 8 --nruns 10
[INFO] Saved output to testing_no_load.txt
[INFO] mmult for dataset 'testing' completed with comparison: Fail(ref).
[STEP] Generating matrices for --load ...
[INFO] Preparing to generate matrices for dataset 'testing' with dimensions 16x12 and 12x8 ...
[INFO] Removed old file: A.bin
[INFO] Removed old file: B.bin
[INFO] Removed old file: python_ref.bin
Generated A.bin with shape (16, 12), B.bin with shape (12, 8), and python_ref.bin with shape (16
[INFO] Waiting for python_ref.bin to be ready...
[INFO] python_ref.bin is ready.
[STEP] Running with --load ...
[INFO] Running: Command used: ./mmult -i naive --M 16 --N 12 --P 8 --nruns 10 --load
[INFO] Saved output to testing_with_load.txt
[INFO] mmult for dataset 'testing' completed with comparison: Success.

=== Testing dataset: small (M=121, N=180, P=115) ===
[STEP] Running without --load ...
[INFO] Running: Command used: ./mmult -i naive --M 121 --N 180 --P 115 --nruns 10
[INFO] Saved output to small_no_load.txt
[INFO] mmult for dataset 'small' completed with comparison: Success.
[STEP] Generating matrices for --load ...
[INFO] Preparing to generate matrices for dataset 'small' with dimensions 121x180 and 180x115 ..
[INFO] Removed old file: A.bin
[INFO] Removed old file: B.bin
[INFO] Removed old file: python_ref.bin
Generated A.bin with shape (121, 180), B.bin with shape (180, 115), and python_ref.bin with shap
[INFO] Waiting for python_ref.bin to be ready...
[INFO] python_ref.bin is ready.
[STEP] Running with --load ...
[INFO] Running: Command used: ./mmult -i naive --M 121 --N 180 --P 115 --nruns 10 --load
[INFO] Saved output to small_with_load.txt
[INFO] mmult for dataset 'small' completed with comparison: Success.

=== Testing dataset: medium (M=550, N=620, P=480) ===
[STEP] Running without --load ...
[INFO] Running: Command used: ./mmult -i naive --M 550 --N 620 --P 480 --nruns 10
[INFO] Saved output to medium_no_load.txt
[INFO] mmult for dataset 'medium' completed with comparison: Success.
[STEP] Generating matrices for --load ...
[INFO] Preparing to generate matrices for dataset 'medium' with dimensions 550x620 and 620x480 .
[INFO] Removed old file: A.bin
[INFO] Removed old file: B.bin
[INFO] Removed old file: python_ref.bin
Generated A.bin with shape (550, 620), B.bin with shape (620, 480), and python_ref.bin with shap
[INFO] Waiting for python_ref.bin to be ready...
```

```
[INFO] python_ref.bin is ready.
[STEP] Running with --load ...
[INFO] Running: Command used: ./mmult -i naive --M 550 --N 620 --P 480 --nruns 10 --load
[INFO] Saved output to medium_with_load.txt
[INFO] mmult for dataset 'medium' completed with comparison: Success.

=== Testing dataset: large (M=962, N=1012, P=1221) ===
[STEP] Running without --load ...
[INFO] Running: Command used: ./mmult -i naive --M 962 --N 1012 --P 1221 --nruns 10
[INFO] Saved output to large_no_load.txt
[INFO] mmult for dataset 'large' completed with comparison: Success.
[STEP] Generating matrices for --load ...
[INFO] Preparing to generate matrices for dataset 'large' with dimensions 962x1012 and 1012x1221
[INFO] Removed old file: A.bin
[INFO] Removed old file: B.bin
[INFO] Removed old file: python_ref.bin
Generated A.bin with shape (962, 1012), B.bin with shape (1012, 1221), and python_ref.bin with sh
[INFO] Waiting for python_ref.bin to be ready...
[INFO] python_ref.bin is ready.
[STEP] Running with --load ...
[INFO] Running: Command used: ./mmult -i naive --M 962 --N 1012 --P 1221 --nruns 10 --load
[INFO] Saved output to large_with_load.txt
[INFO] mmult for dataset 'large' completed with comparison: Success.

=== Testing dataset: native (M=2500, N=3000, P=2100) ===
[STEP] Running without --load ...
[INFO] Running: Command used: ./mmult -i naive --M 2500 --N 3000 --P 2100 --nruns 10
[INFO] Saved output to native_no_load.txt
[INFO] mmult for dataset 'native' completed with comparison: Success.
[STEP] Generating matrices for --load ...
[INFO] Preparing to generate matrices for dataset 'native' with dimensions 2500x3000 and 3000x21
[INFO] Removed old file: A.bin
[INFO] Removed old file: B.bin
[INFO] Removed old file: python_ref.bin
Generated A.bin with shape (2500, 3000), B.bin with shape (3000, 2100), and python_ref.bin with
[INFO] Waiting for python_ref.bin to be ready...
[INFO] python_ref.bin is ready.
[STEP] Running with --load ...
[INFO] Running: Command used: ./mmult -i naive --M 2500 --N 3000 --P 2100 --nruns 10 --load
[INFO] Saved output to native_with_load.txt
[INFO] mmult for dataset 'native' completed with comparison: Success.

[INFO] All tests completed.

basilp4@basilp4:~/Documents/YABMS2/build$ python3 automate_tests.py --nruns 100
[INFO] Starting automation with nruns=100...

=== Testing dataset: testing (M=16, N=12, P=8) ===
[STEP] Running without --load ...
[INFO] Running: Command used: ./mmult -i naive --M 16 --N 12 --P 8 --nruns 100
[INFO] Saved output to testing_no_load.txt
[INFO] mmult for dataset 'testing' completed with comparison: Fail(ref).
```

```
[STEP] Generating matrices for --load ...
[INFO] Preparing to generate matrices for dataset 'testing' with dimensions 16x12 and 12x8 ...
[INFO] Removed old file: A.bin
[INFO] Removed old file: B.bin
[INFO] Removed old file: python_ref.bin
Generated A.bin with shape (16, 12), B.bin with shape (12, 8), and python_ref.bin with shape (16
[INFO] Waiting for python_ref.bin to be ready...
[INFO] python_ref.bin is ready.
[STEP] Running with --load ...
[INFO] Running: Command used: ./mmult -i naive --M 16 --N 12 --P 8 --nruns 100 --load
[INFO] Saved output to testing_with_load.txt
[INFO] mmult for dataset 'testing' completed with comparison: Success.

=== Testing dataset: small (M=121, N=180, P=115) ===
[STEP] Running without --load ...
[INFO] Running: Command used: ./mmult -i naive --M 121 --N 180 --P 115 --nruns 100
[INFO] Saved output to small_no_load.txt
[INFO] mmult for dataset 'small' completed with comparison: Success.
[STEP] Generating matrices for --load ...
[INFO] Preparing to generate matrices for dataset 'small' with dimensions 121x180 and 180x115 ..
[INFO] Removed old file: A.bin
[INFO] Removed old file: B.bin
[INFO] Removed old file: python_ref.bin
Generated A.bin with shape (121, 180), B.bin with shape (180, 115), and python_ref.bin with shap
[INFO] Waiting for python_ref.bin to be ready...
[INFO] python_ref.bin is ready.
[STEP] Running with --load ...
[INFO] Running: Command used: ./mmult -i naive --M 121 --N 180 --P 115 --nruns 100 --load
[INFO] Saved output to small_with_load.txt
[INFO] mmult for dataset 'small' completed with comparison: Success.

=== Testing dataset: medium (M=550, N=620, P=480) ===
[STEP] Running without --load ...
[INFO] Running: Command used: ./mmult -i naive --M 550 --N 620 --P 480 --nruns 100
[INFO] Saved output to medium_no_load.txt
[INFO] mmult for dataset 'medium' completed with comparison: Success.
[STEP] Generating matrices for --load ...
[INFO] Preparing to generate matrices for dataset 'medium' with dimensions 550x620 and 620x480 .
[INFO] Removed old file: A.bin
[INFO] Removed old file: B.bin
[INFO] Removed old file: python_ref.bin
Generated A.bin with shape (550, 620), B.bin with shape (620, 480), and python_ref.bin with shap
[INFO] Waiting for python_ref.bin to be ready...
[INFO] python_ref.bin is ready.
[STEP] Running with --load ...
[INFO] Running: Command used: ./mmult -i naive --M 550 --N 620 --P 480 --nruns 100 --load
[INFO] Saved output to medium_with_load.txt
[INFO] mmult for dataset 'medium' completed with comparison: Success.

=== Testing dataset: large (M=962, N=1012, P=1221) ===
[STEP] Running without --load ...
[INFO] Running: Command used: ./mmult -i naive --M 962 --N 1012 --P 1221 --nruns 100
```

```
[INFO] Saved output to large_no_load.txt
[INFO] mmult for dataset 'large' completed with comparison: Success.
[STEP] Generating matrices for --load ...
[INFO] Preparing to generate matrices for dataset 'large' with dimensions 962x1012 and 1012x1221
[INFO] Removed old file: A.bin
[INFO] Removed old file: B.bin
[INFO] Removed old file: python_ref.bin
Generated A.bin with shape (962, 1012), B.bin with shape (1012, 1221), and python_ref.bin with sh
[INFO] Waiting for python_ref.bin to be ready...
[INFO] python_ref.bin is ready.
[STEP] Running with --load ...
[INFO] Running: Command used: ./mmult -i naive --M 962 --N 1012 --P 1221 --nruns 100 --load
[INFO] Saved output to large_with_load.txt
[INFO] mmult for dataset 'large' completed with comparison: Success.

=== Testing dataset: native (M=2500, N=3000, P=2100) ===
[STEP] Running without --load ...
[INFO] Running: Command used: ./mmult -i naive --M 2500 --N 3000 --P 2100 --nruns 100
[INFO] Saved output to native_no_load.txt
[INFO] mmult for dataset 'native' completed with comparison: Success.
[STEP] Generating matrices for --load ...
[INFO] Preparing to generate matrices for dataset 'native' with dimensions 2500x3000 and 3000x21
[INFO] Removed old file: A.bin
[INFO] Removed old file: B.bin
[INFO] Removed old file: python_ref.bin
Generated A.bin with shape (2500, 3000), B.bin with shape (3000, 2100), and python_ref.bin with
[INFO] Waiting for python_ref.bin to be ready...
[INFO] python_ref.bin is ready.
[STEP] Running with --load ...
[INFO] Running: Command used: ./mmult -i naive --M 2500 --N 3000 --P 2100 --nruns 100 --load
[INFO] All tests completed.
```

All the above console logs are included in this appendix to ensure complete reproducibility and transparency of the testing process.