Benchmark Report: GPUs Performance Comparison of Ipazia and Sofia for Machine Learning and AI Workloads

Abstract

This document presents a comparative performance analysis of two high-performance computing (HPC) systems, Ipazia and Sofia. The former is a server HPC platform, while the latter is a workstation, specifically designed for machine learning (ML) and artificial intelligence (AI) workloads. Both are available in the HPC facility of the Department of Mathematics. The benchmarks focus on matrix multiplication performance, evaluated using standardised Compute Unified Device Architecture (CUDA) sample tests and custom scripts for PyTorch and TensorFlow. The results provide an in-depth understanding of each system's capabilities in handling matrix operations, a fundamental component of deep learning and numerical computing.

1. INTRODUCTION

The rapid advancements in artificial intelligence (AI) and machine learning (ML) have placed increasing demands on computational resources, particularly for deep learning workloads. The performance of these workloads is heavily dependent on the ability to perform matrix operations efficiently, as these constitute the core computations in neural networks. GPUs (Graphics Processing Units) have become the cornerstone of modern AI research and deployment due to their ability to execute massive parallel computations, making them ideal for matrix multiplications.

1.1 The Role of GPUs in Machine Learning and AI

Matrix multiplication is a fundamental operation in deep learning, where weight matrices are multiplied by input tensors to perform transformations crucial for training and inference. Efficient execution of these operations directly impacts model training time and inference speed.

1.2 Technologies and Frameworks for GPU Computing

The effectiveness of GPUs in AI applications is amplified by software frameworks that provide optimised libraries for numerical computation. Key technologies include:

- CUDA (Compute Unified Device Architecture) [1]: NVIDIA’s parallel computing platform that allows developers to execute highly optimised GPU-based computations.
- PyTorch [2] and TensorFlow [3]: Leading deep learning frameworks that leverage GPU acceleration for matrix computations essential in training and inference.

1.3 Rationale for Benchmarking Approach

To evaluate the performance of two HPC systems, Ipazia and Sofia, a benchmarking methodology was established using CUDA sample tests and custom-developed scripts for PyTorch and TensorFlow. The benchmarks were selected to:

- Establish a base test with CUDA Samples: The CUDA samples include a matrix multiplication test, which measures raw computational power in floating-point operations per second (FLOPS).
- Assess Real-World AI Workloads with PyTorch and TensorFlow: while CUDA samples provide fundamental performance metrics, deep learning frameworks introduce additional

computational overhead. By benchmarking matrix multiplications in PyTorch and TensorFlow, we gain insight into how each system performs under typical AI workloads.

2. SYSTEM SPECIFICATIONS

A fundamental aspect of this benchmark study is understanding the underlying hardware architecture of each system. The specifications of both systems under examination (Ipazia and Sofia) are summarised in **Table 1**.

Table 1. Hardware configurations of the HPC systems Ipazia and Sofia, detailing GPU models, memory capacities, and other relevant specifications.

Component(s)	Ipazia	Sofia
Processors	2 x Intel Xeon 5220, 18 core 2.2 GHz	2 x Intel Xeon Gold 6438Y+, 32core, 2.0 GHz
Thread per core	1	2
L3 Cache	60 MB	24.75 MB
GPUs	2 x NVIDIA Tesla V100/V100S 32GB PCIe	2 x NVIDIA RTX 6000 Ada Generation 48GB GDDR6, 4xDP
Memory	384GB DDR4 (12 x 32GB)	832GB DDR5 (13 x 64GB) 4800 MHz
Storage	2 x 480GB SSD 2 x 1.6TB NVMe SSD	2 x 2TB SSD M.2 PCIe Gen4, 2 x 4TB SSD M.2 PCIe Gen4
Operating System	Debian 12.5	Debian 12.9
Cuda Toolkit Version	12.4	12.4
Cuda Driver Version	550.54.15	550.54.15
cuDNN Version	9.8.0	9.8.0

Ipazia and Sofia present key architectural differences that impact their performance across CPU, GPU, memory, and storage subsystems. Sofia's dual Intel Xeon Gold 6438Y+ processors (64 cores, 2.0 GHz) vastly outperform Ipazia's dual Intel Xeon 5220 CPUs (36 cores, 2.2 GHz), resulting in superior multi-threaded and single-threaded performance for AI/ML workloads, and parallel computing.

In terms of GPUs, Sofia's NVIDIA RTX 6000 Ada Generation [4] far exceed Ipazia's 2 x NVIDIA Tesla V100 [5], offering higher CUDA core counts, improved Tensor Core performance, 48GB of GDDR6 per GPU (vs. 32GB HBM2), and PCIe Gen4 support, significantly enhancing AI training and inference.

Additionally, Sofia's 832GB DDR5 memory at 4800 MHz provides greater bandwidth and lower latency compared to Ipazia's 384GB DDR4, allowing it to efficiently handle large datasets and intensive computations.

Storage is also a key differentiator, as Sofia's 2 x 2TB + 2 x 4TB M.2 PCIe Gen4 SSDs deliver far superior read/write speeds and lower I/O latency than Ipazia's SATA SSDs and PCIe Gen3 NVMe drives, improving system responsiveness in data-heavy applications.

While both systems run Debian and CUDA 12.4 with the same driver version, ensuring software compatibility, Sofia's vastly superior compute power, memory bandwidth, and storage speed make

it the clear choice for workloads requiring extreme parallelism, high AI acceleration, and fast data processing, leaving Ipazia significantly behind in overall performance.

3. BENCHMARKING METHODOLOGY

The performance assessment was carried out using a combination of standard and custom benchmarks, specifically designed to evaluate the efficiency of matrix multiplication.

3.1 CUDA Sample Tests

As an initial step, several tests from the CUDA Samples suite [6] were executed to gain a preliminary understanding of the computational power and efficiency of each GPU. Among these, two tests were particularly relevant:

Memory Bandwidth Test, which measures data transfer speeds between GPU memory and system memory, providing insight into bandwidth management capabilities.

Matrix Multiplication, a fundamental test for assessing computational performance in floating-point operations. To ensure comparability across devices, the benchmark was run using fixed matrix dimensions (Matrix A: 320 × 320, Matrix B: 640 × 320).

3.2 Custom PyTorch and TensorFlow Benchmarks

To gain a deeper understanding of performance in machine learning workloads, custom scripts were developed and executed using both PyTorch and TensorFlow.

PyTorch 2.6.0 and TensorFlow 2.18.0 were used to analyse matrix multiplication scalability, recording computation times for increasing matrix sizes, ranging from 2^2 to 2^{15}. This approach allowed for an evaluation of computational efficiency under progressively larger workloads. The codes for these benchmarks are reported in appendix **7**.

3.4 Computation of TFLOPS for Matrix Multiplication with Square Matrices of Size $2^n \times 2^n$

Given two square matrices A and B, each of size $2^n \times 2^n$, the task at hand is to compute the number of floating-point operations (FLOP) required for their multiplication and subsequently determine the throughput in TeraFLOPS (TFLOPS).

In the case of matrix multiplication, each element c_{ij} of the resulting matrix $C = A \cdot B$ is computed as:

$$c_{ij} = \sum_{k=1}^{2^n} a_{ik} \cdot b_{kj}$$

Each such computation involves $2^n \times 2^n$ multiplications and $2^n - 1$ additions. Hence, for each element of the resulting matrix C, leading to a total of $2^{n+1} - 1$ FLOP are required.
The resulting matrix C has $2^n \times 2^n$ elements, and therefore the total number of FLOP required for the entire matrix multiplication process can be expressed as:

$$FLOP = 2 \cdot 2^{2n} \cdot (2^n - 1) = 2^{3n+1} - 2^{2n+1}$$

Thus, the total number of floating-point operations required for multiplying two square matrices of size $2^n \times 2^n$ is $2^{3n+1} - 2^{2n+1}$.

To compute the throughput in TFLOPs, we use the formula:

$$TFLOPS = \frac{Total\ FLOP}{Execution\ Time\ in\ seconds \times 10^{12}}$$

Where:

- Total FLOP is given by equation 2,
- Execution Time in seconds is the time taken to complete the matrix multiplication.

This derivation illustrates how to estimate the computational cost and performance of matrix multiplication, providing a straightforward method to compute the floating-point operations involved and the corresponding TFLOPS for a given execution time.

4. RESULTS AND DISCUSSION

4.1 CUDA Sample Results

The CUDA samples provided a baseline for GPU performance.

4.1.1 Bandwidth Test

The memory bandwidth highlights the RTX 6000 Ada's superior data transfer capabilities. These results demonstrate a substantial advantage in Host-to-Device (H2D), Device-to-Host (D2H), and Device-to-Device (D2D) communication, underscoring the efficiency improvements introduced with the Ada Lovelace architecture. Such enhancements are particularly relevant for workloads heavily reliant on fast memory access, further distinguishing Sofia's hardware from the older Volta-based V100/V100S GPUs in Ipazia.

Table 2. Results of the bandwidth test, comparing memory transfer speeds and computational efficiency between Ipazia and Sofia under identical benchmarking conditions.

Test Type	Ipazia (V100/V100S)	Ipazia (Cumulative)	Sofia (RTX 6000 Ada)	Sofia (Cumulative)	Speedup Factor (Sofia vs. Ipazia)
H2D	7.2 / 7.7 GB/s	14.9 GB/s	24.9 / 25.1 GB/s	50.1 GB/s	3.4x
D2H	8.2 / 9.2 GB/s	17.5 GB/s	27.0 / 27.0 GB/s	54.0 GB/s	3.2x
D2D	726.7/906.7 GB/s	1633.7 GB/s	4184.1/4296.6 GB/s	8508.8 GB/s	5.9x

The RTX 6000 Ada on Sofia demonstrates a transfer bandwidth approximately 3.4 times higher than the Tesla V100/V100S on Ipazia. This significant difference persists even when considering cumulative benchmarks, with the RTX 6000 Ada achieving 50.1 GB/s compared to 14.9 GB/s for H2D transfers and 54.0 GB/s versus 17.5 GB/s for D2H transfers.

In terms of D2D transfers, the RTX 6000 Ada exhibits a throughput roughly 5.9 times greater than the V100/V100S for internal GPU operations. This advantage remains pronounced in

dual-GPU configurations, where the RTX 6000 Ada achieves 8508.8 GB/s against the V100/V100S's 1633.7 GB/s.

Overall, Sofia's superior memory handling capabilities make it significantly more efficient for applications requiring high-intensity communication between the GPU and RAM. In contrast, Ipazia, equipped with the V100/V100S, shows more limited performance compared to newer hardware, making it better suited for workloads less dependent on memory access speed.

4.1.2 Matrix Multiplication

The CUDA sample *matrixMul* CUDA sample results indicate a substantial performance advantage of the RTX 6000 Ada Generation over the Tesla V100(S). Specifically, the Tesla V100 achieved 1.96 TFLOPS with a computation time of 67 µs, whereas the RTX 6000 Ada recorded 3.27 TFLOPS, reducing the computation time to 40 µs. This translates to a 67 % increase in performance.

While these results provide a preliminary assessment of raw computational throughput, it is important to note that the *matrixMul* CUDA sample is not designed for rigorous performance benchmarking, as explicitly stated in its documentation.

The observed performance disparity aligns with the expected generational improvements in NVIDIA's Ada Lovelace architecture. The RTX 6000 Ada benefits from:

- Enhanced Streaming Multiprocessors (SMs) with increased per-core efficiency.
- Faster memory bandwidth, reducing data transfer bottlenecks.
- Improved CUDA core performance, which, although not directly investigated in this specific benchmark, contributes to the overall architectural efficiency.

4.2 PyTorch Performance Benchmarking

As expected, both HPC systems displayed linear scaling up to a certain threshold, beyond which memory bandwidth and GPU communication bottlenecks became evident. **Figure 1** shows that Tesla V100S GPUs in Ipazia exhibited stable but slightly lower throughput compared to Sofia's 6000 Ada Generation GPUs.

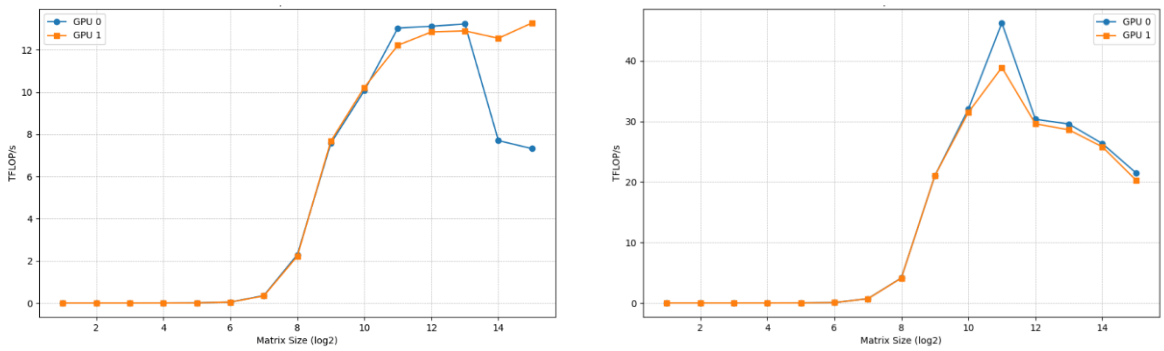

Figure 1. Performance comparison of Ipazia and Sofia in terms of TFLOPS as a function of matrix size using PyTorch. The left graph represents Ipazia's results, while the right graph illustrates Sofia's performance with RTX 6000 Ada GPUs.

Performance scaling across matrix sizes exhibits a clear trend where Sofia demonstrates superior throughput, particularly for large-scale matrix multiplications, whereas Ipazia shows early saturation and a significant drop in efficiency at extreme sizes. At smaller matrix dimensions (2x2

to 128x128), both systems exhibit negligible computational throughput, constrained by memory latency and kernel launch overheads rather than raw FLOP capability. In this range, Sofia maintains slightly higher TFLOPS values, but the difference is minimal due to the minimal computational workload. As matrix size increases beyond 256x256, Sofia begins to significantly outperform Ipazia. At 512x512, Sofia reaches over 21 TFLOPS, while Ipazia lags below 8 TFLOPS. At 1024x1024 and 2048x2048, Sofia achieves 32 and 46 TFLOPS, respectively, whereas Ipazia remains constrained to approximately 10-13 TFLOPS. Beyond 4096x4096, Sofia continues to sustain high performance with marginal decline, peaking at 46 TFLOPS before gradually reducing to 21 TFLOPS at 32768x32768. In contrast, Ipazia exhibits an abrupt performance drop at extreme sizes, particularly in GPU 0, which falls to 7.4 TFLOPS at 16384x16384 and 7.3 TFLOPS at 32768x32768, while GPU 1 remains significantly faster, peaking at 13.3 TFLOPS. This discrepancy suggests potential interconnect bottlenecks or memory contention issues in the V100 architecture when handling large matrix operations. The substantial disparity between the two systems at higher problem sizes can be primarily attributed to the architectural advancements of the Ada-Generation-based RTX 6000, including its higher theoretical FP16/FP32 throughput and increased memory bandwidth.

4.3 TensorFlow Performance Benchmarking

TensorFlow mirrored the trends observed in PyTorch, with Sofia's GPUs handling larger batch sizes more efficiently and exhibiting superior memory bandwidth utilisation. The performance gap between the two systems widened as model complexity increased, as shown in **Figure 2**.

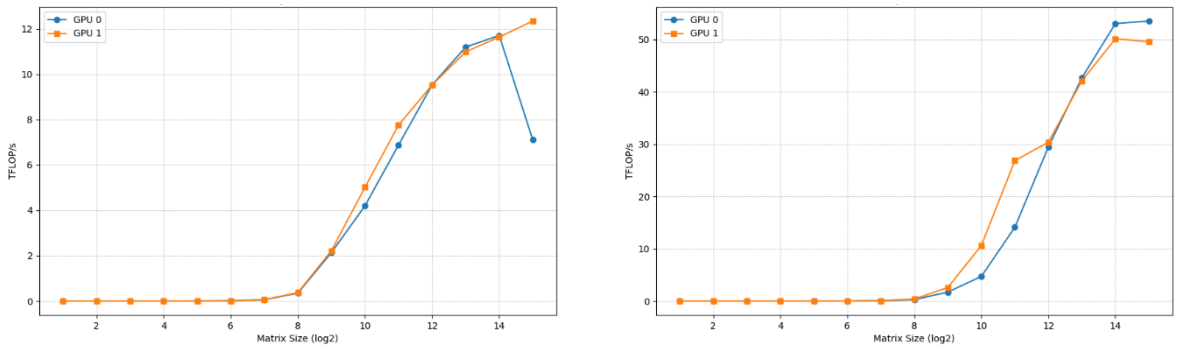

Figure 2. Performance comparison of Ipazia and Sofia in terms of TFLOPS as a function of matrix size using TensorFlow. The left graph corresponds to Ipazia, while the right graph displays the performance of Sofia's RTX 6000 Ada GPUs.

For small matrix sizes (2x2 to 128x128), the performance of both systems remains relatively low due to the overhead associated with kernel launch and memory transfers, which dominate execution time in such scenarios. In this regime, execution times are in the order of microseconds, and TFLOPS values remain negligible. However, as the matrix size increases beyond 256x256, the computational efficiency improves considerably.

Ipazia exhibits a steady increase in performance up to matrix sizes of 16384x16384, where the peak throughput on GPU 0 reaches approximately 11.71 TFLOPS, and GPU 1 follows closely at 11.64 TFLOPS. However, at 32768x32768, a significant discrepancy emerges between the two GPUs: while GPU 1 achieves 12.35 TFLOPS, GPU 0's performance drops to 7.10 TFLOPS, suggesting potential memory bandwidth limitations.

In contrast, Sofia demonstrates a far superior scaling trend, with its GPUs reaching considerably higher computational throughput. For large matrices (4096x4096 and above), the RTX 6000 Ada GPUs outperform the V100-based system by a substantial margin. The peak performance is observed at 32768x32768, where GPU 0 achieves 53.50 TFLOPS, while GPU 1 reaches 49.54 TFLOPS. This marks a more than fourfold increase in computational efficiency compared to Ipazia's best results.

Additionally, the significantly lower execution times on Sofia suggest more efficient kernel execution and data transfer mechanisms, further contributing to the observed performance gap.

5. CONCLUSION

The comparative analysis of the two HPC systems, Sofia and Ipazia, has revealed substantial performance disparities, particularly in memory bandwidth and computational throughput. Across all evaluated benchmarks, the RTX 6000 Ada GPUs in Sofia demonstrated significant advantages over the Tesla V100/V100S GPUs in Ipazia, confirming the impact of architectural advancements in NVIDIA's Ada Lovelace generation.

Memory bandwidth assessments highlighted a crucial bottleneck in the older Volta-based architecture, with the RTX 6000 Ada Generation achieving up to 5.9 times the transfer bandwidth of the Tesla V100/V100S in D2D communication. This discrepancy suggests that workloads with intensive memory transactions will benefit considerably from the improved data transfer capabilities of the RTX 6000 Ada.

The matrix multiplication experiments corroborated these findings, demonstrating a 67% increase in raw computational throughput for the RTX 6000 Ada compared to the Tesla V100. The enhanced Streaming Multiprocessors (SMs), increased memory bandwidth, and superior architectural efficiency of the Ada Lovelace GPUs collectively contributed to this substantial performance uplift.

Deep learning benchmarks using PyTorch and TensorFlow further reinforced the generational gap between the two GPU architectures. While both systems exhibited predictable scaling behaviour, the RTX 6000 Ada consistently outperformed the Tesla V100, particularly for large-scale tensor operations. Sofia maintained significantly higher computational throughput in matrix multiplications beyond 4096×4096, where the Tesla V100 suffered from early performance saturation. Notably, the RTX 6000 Ada achieved peak performance above 50 TFLOPS in TensorFlow experiments, more than four times the maximum performance observed on the V100S.

The results suggest that the performance gap widens with increasing problem size, implying that newer architectures are increasingly optimized for large-scale AI and scientific computing workloads. The presence of memory bandwidth limitations in the Tesla V100, especially in extreme-scale computations, highlights the necessity of transitioning to modern GPU architectures to mitigate such bottlenecks.

In conclusion, the RTX 6000 Ada represents a significant leap in GPU performance, driven by architectural refinements, enhanced memory handling, and superior execution efficiency. These results underscore the importance of hardware selection in HPC environments, particularly for AI, deep learning, and large-scale numerical simulations. Future research should explore the implications of these architectural differences in multi-GPU and distributed computing scenarios, where interconnect bandwidth and parallelism play an even more critical role.

6. REFERENCES

[1] https://docs.nvidia.com/cuda/
[2] https://pytorch.org/docs/stable/index.html
[3] https://www.tensorflow.org
[4] https://www.nvidia.com/en-us/design-visualization/rtx-6000/
[5] https://www.nvidia.com/en-gb/data-center/tesla-v100/
[6] https://github.com/NVIDIA/cuda-samples

7. APPENDICES

7.1 PyTorch Benchmark

```
import torch
import time
import logging
import matplotlib.pyplot as plt
import pynvml
import numpy as np

# Logging configuration
logging.basicConfig(filename = "matrix_multiplication.log", level
= logging.INFO, format = "%(asctime)s - %(message)s")

# Initialisation of NVML for temperature monitoring
def get_gpu_temperature(device):
    pynvml.nvmlInit()
    handle = pynvml.nvmlDeviceGetHandleByIndex(device)
    temp = pynvml.nvmlDeviceGetTemperature(handle,
pynvml.NVML_TEMPERATURE_GPU)
    pynvml.nvmlShutdown()
    return temp

sizes = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096,
8192, 16384, 32768]
num_trials = 100

torch.manual_seed(42)
torch.cuda.manual_seed(42)

def benchmark_single_gpu(device, s):
    torch.cuda.set_device(device)
    tensor_a = torch.randn(s, s, device=f"cuda:{device}")
    tensor_b = torch.randn(s, s, device=f"cuda:{device}")

    # Warm up
    for _ in range(50):
        torch.matmul(tensor_a, tensor_b)

    torch.cuda.synchronize()
```

```
    memory_before = torch.cuda.memory_allocated(device)

    start = time.time()
    for _ in range(num_trials):
        torch.matmul(tensor_a, tensor_b)
    torch.cuda.synchronize()
    end = time.time()

    memory_after = torch.cuda.memory_allocated(device)

    avg_time = (end - start) / num_trials

    logging.info(f"GPU {device}: Allocated memory: {memory_after -
memory_before} bytes")

    torch.cuda.empty_cache()

    return avg_time

def tflops(s, avg_time):
    return (1e-12) * (2 * s ** 3 - s ** 2) / avg_time

gpu_0_results = []
gpu_1_results = []

temp_0 = get_gpu_temperature(0)
temp_1 = get_gpu_temperature(1)
logging.info(f"Initial temperature GPU 0: {temp_0}°C")
logging.info(f"Initial temperature GPU 1: {temp_1}°C")

for size in sizes:
    print(f"\nBenchmarking matrix size {size}x{size}...")
    logging.info(f"Benchmarking matrix size {size}x{size}")

    time_gpu_0 = benchmark_single_gpu(0, size)
    time_gpu_1 = benchmark_single_gpu(1, size)

    tflops_gpu_0 = tflops(size, time_gpu_0)
    tflops_gpu_1 = tflops(size, time_gpu_1)

    gpu_0_results.append(tflops_gpu_0)
    gpu_1_results.append(tflops_gpu_1)

    print(f" Average time on GPU 0: {time_gpu_0:.6f} s")
    print(f"TFLOP/s on GPU 0: {tflops_gpu_0}")
    print(f"Average time on GPU 1: {time_gpu_1:.6f} s")
    print(f"TFLOP/s on GPU 1: {tflops_gpu_1}")

    temp_0 = get_gpu_temperature(0)
    temp_1 = get_gpu_temperature(1)
    logging.info(f"Temperature GPU 0: {temp_0}°C")
```

```
    logging.info(f"Temperature GPU 1: {temp_1}°C")

    logging.info(f"GPU 0 - Tempo medio: {time_gpu_0:.6f} s,
TFLOP/s: {tflops_gpu_0}")
    logging.info(f"GPU 1 - Tempo medio: {time_gpu_1:.6f} s,
TFLOP/s: {tflops_gpu_1}")

# Plot
plt.figure(figsize=(10, 6))
plt.plot(np.log2(sizes), gpu_0_results, marker='o', label='GPU 0')
plt.plot(np.log2(sizes), gpu_1_results, marker='s', label='GPU 1')
plt.xlabel("Matrix Size (log2)")
plt.ylabel("TFLOP/s")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.savefig("gpu_performance_comparison.png")
```

7.2 TensorFlow Benchmark

```
import tensorflow as tf
import time
import logging
import matplotlib.pyplot as plt
import pynvml
import numpy as np

# Logging configuration
logging.basicConfig(filename="matrix_multiplication_tf.log",
level=logging.INFO, format="%(asctime)s - %(message)s")

# Initialisation of NVML for temperature monitoring
def get_gpu_temperature(device):
    pynvml.nvmlInit()
    handle = pynvml.nvmlDeviceGetHandleByIndex(device)
    temp = pynvml.nvmlDeviceGetTemperature(handle,
pynvml.NVML_TEMPERATURE_GPU)
    pynvml.nvmlShutdown()
    return temp

sizes = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096,
8192, 16384, 32768]
num_trials = 100

tf.random.set_seed(42)

def benchmark_single_gpu(device, s):
    with tf.device(f"/device:GPU:{device}"):
        tensor_a = tf.random.normal((s, s), dtype=tf.float32)
        tensor_b = tf.random.normal((s, s), dtype=tf.float32)

        # Warm-up
```

```
        for _ in range(50):
            tf.linalg.matmul(tensor_a, tensor_b)

        # Synchronisation
        tf.identity(tensor_a).numpy()  # Forced Synchronisation

        # Time measurement
        start = time.time()
        for _ in range(num_trials):
            tf.linalg.matmul(tensor_a, tensor_b)
        # Synchronisation
        tf.identity(tensor_a).numpy()  # Forced Synchronisation
        end = time.time()

        avg_time = (end - start) / num_trials

        # Clearing mem
        tf.keras.backend.clear_session()

        return avg_time

def tflops(s, avg_time):
    return (1e-12) * (2 * s ** 3 - s ** 2) / avg_time

gpu_0_results = []
gpu_1_results = []

temp_0 = get_gpu_temperature(0)
temp_1 = get_gpu_temperature(1)
logging.info(f"Initial temperature GPU 0: {temp_0}°C")
logging.info(f"Initial temperature GPU 1: {temp_1}°C")

for size in sizes:
    print(f"\nBenchmarking matrix size {size}x{size}...")
    logging.info(f"Benchmarking matrix size {size}x{size}")

    time_gpu_0 = benchmark_single_gpu(0, size)
    time_gpu_1 = benchmark_single_gpu(1, size)

    tflops_gpu_0 = tflops(size, time_gpu_0)
    tflops_gpu_1 = tflops(size, time_gpu_1)

    gpu_0_results.append(tflops_gpu_0)
    gpu_1_results.append(tflops_gpu_1)

    print(f"Average time on GPU 0: {time_gpu_0:.6f} s")
    print(f"TFLOP/s on GPU 0: {tflops_gpu_0}")
    print(f"Average time on GPU 1: {time_gpu_1:.6f} s")
    print(f"TFLOP/s on GPU 1: {tflops_gpu_1}")

    temp_0 = get_gpu_temperature(0)
```

```
    temp_1 = get_gpu_temperature(1)
    logging.info(f"Temperature GPU 0: {temp_0}°C")
    logging.info(f"Temperature GPU 1: {temp_1}°C")

    logging.info(f"GPU 0 - Average time: {time_gpu_0:.6f} s,
TFLOP/s: {tflops_gpu_0}")
    logging.info(f"GPU 1 - Average time: {time_gpu_1:.6f} s,
TFLOP/s: {tflops_gpu_1}")

# Plot
plt.figure(figsize=(10, 6))
plt.plot(np.log2(sizes), gpu_0_results, marker='o', label='GPU 0')
plt.plot(np.log2(sizes), gpu_1_results, marker='s', label='GPU 1')
plt.xlabel("Matrix Size (log2)")
plt.ylabel("TFLOP/s")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.savefig("gpu_performance_comparison_tf.png")
```