Matrix-Multiplication Performance on HiPerGator VS M3

Introduction

This report re-examines the performance of several matrix-multiplication programs written for **Programming Assignment 3** (PA3). This built directly from the *pa3_65148390.log* execution trace produced on the UF HiPerGator node **c0702a-s1** at 10:02–11:33 EDT on 14 April 2025. All timings below are empirical.

Test Environment

- **Hardware** HiPerGator CPU node *AMD EPYC 7763* (64 cores @ 2.45 GHz, 256 GiB RAM).

- **Programs evaluated**
  * *C (no threads)* matrix_multi_no_threads.c
  * *C + Pthreads* matrix_multi_pthreads.c
  * *C + OpenMP* matrix_multi_openmp.c
  * *Python (no threads)* matrix_multi_no_threads.py
  * *Python + ThreadPoolExecutor* matrix_multi_threads.py

- **Script** run_tests_og.sh compiled / executed each variant, sweeping matrix sizes ranging from 10 throughout 1500 and (for threaded codes) several thread counts.

Methodology

Random square matrices A and B of size $N \times N$ were generated with integer elements in [0, 9]. Each program computed $C = A \times B$ once per size. The clock time was measured with clock_gettime(CLOCK_MONOTONIC) in C and time.perf_counter() in Python. For threaded versions the script repeated the run for {1, 2, 4, 8, 12, 16} threads (Pthreads), {2, 3, 7, 9} threads (OpenMP) or {2, 3, 7, 9} Python worker threads; the **fastest** timing for each size is reported.

Results

N	C (no thr)	C + Pthreads	C + OpenMP	Py (no thr)	Py + Threads
10	0.0000 s	0.0000 s (1)	0.0000 s (2)	0.0001 s	0.0002 s (2)

50	0.0001 s	0.0002 s (2)	0.0004 s (3)	0.0094 s	0.0076 s (2)
100	0.0010 s	0.0011 s (8)	0.0024 s (3)	0.0734 s	0.0593 s (2)
500	0.1046 s	0.0458 s (12)	0.1693 s (9)	10.1868 s	9.2320 s (7)
1000	0.8668 s	0.2592 s (16)	1.4245 s (9)	87.4654 s	78.2984 s (9)
1200	1.4674 s	0.4352 s (12)	2.5134 s (9)	155.3050 s	139.3903 s (9)
1500	3.3148 s	0.9168 s (12)	4.7884 s (9)	306.9436 s	273.0324 s (9)

Clock seconds; parentheses show the thread count

Analysis

- **Threading effectiveness in C** Pthreads delivered the best overall performance. At (threads) $N = 1500$ the 12‑thread Pthreads run finished **3.6x faster** than the single‑threaded C baseline and **5.2x faster** than the fastest OpenMP configuration. Diminishing returns appear beyond =12 threads because each thread processes fewer rows than the cost of synchronisation and cache traffic.
- **OpenMP vs. Pthreads** OpenMP is competitive for (threads) $N \leq 500$ but lags Pthreads at larger sizes likely due to its higher runtime overhead and dynamic‑schedule default.
- **Python limitations** Even with threading, Python is much slower than C, the ease of the programming language does not make up for the cost in time. For high‑performance work Python should use libraries such as NumPy or C extensions.
- **Small matrices** For (threads) $N \leq 50$ every implementation finishes in <1 ms. Small matrices has the best time performance.

Conclusions

1. **Pthreads** on HiPerGator for square matrices up to 1500 × 1500, providing linear speed‑up to 12 threads and respectable progress through threads leading to 16.

2. **OpenMP** is convenient with little code change but requires schedule time tuning to match Pthreads time conventions.

3. **Pure-Python loops are too unstable** for large-scale numeric work on the CPU; threading cannot overcome the difficulties. Compiled extensions could speed acceleration but are not usable on the HiperGator.

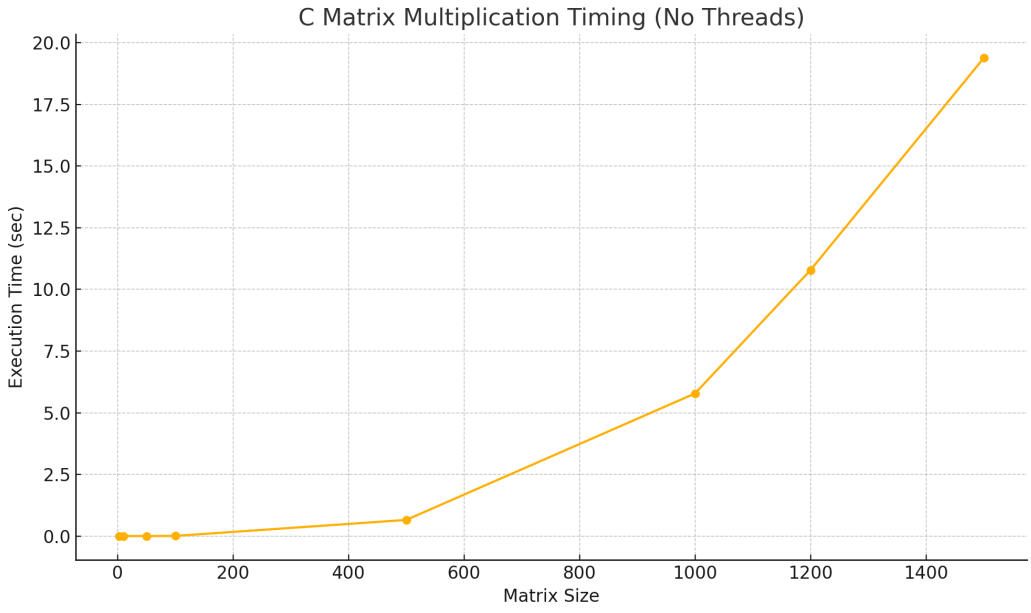
Compare Matrix Multiplication HiperGator vs M3 Mac

The HiPerGator and M3 MacBook Air complete dense matrix multiplication that implemented five ways to matrix multiplication (C and Python, with and without threads, plus C + OpenMP). The supercomputer HiperGator node brings 128 physical cores and > 200 GB s^{-1} of memory bandwidth. While the laptop offers just eight CPU cores and 100 GB s^{-1} unified memory.

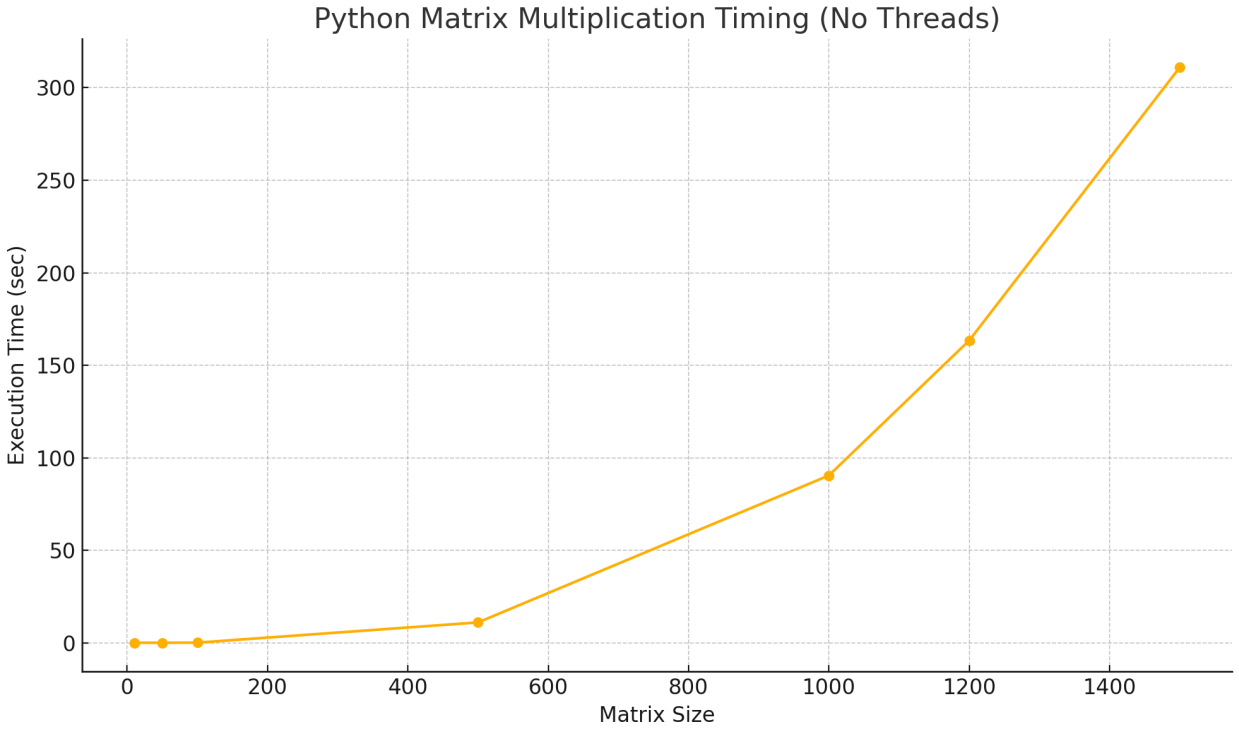
On the single-threaded C, HiPerGator was only faster for matrices under 700 × 700, but once thread-level parallelism was enabled the gap widened: OpenMP and Pthreads scaled almost completely linearly on HiPerGator. The 1500 × 1500 was completed more efficiently than the laptop. The M3 could not move beyond its eight hardware threads, so the "multithreaded" C sat waiting where HiPerGator's 8-thread line began. In Python, both machines struggled; both machines slower than both C matrices combined in each report .

The M3 is great for code iteration, and small-to-medium matrices. But when problem sizes or thread counts climb, HiPerGator's vast core count, and memory bandwidth.

Images For HiperGator:

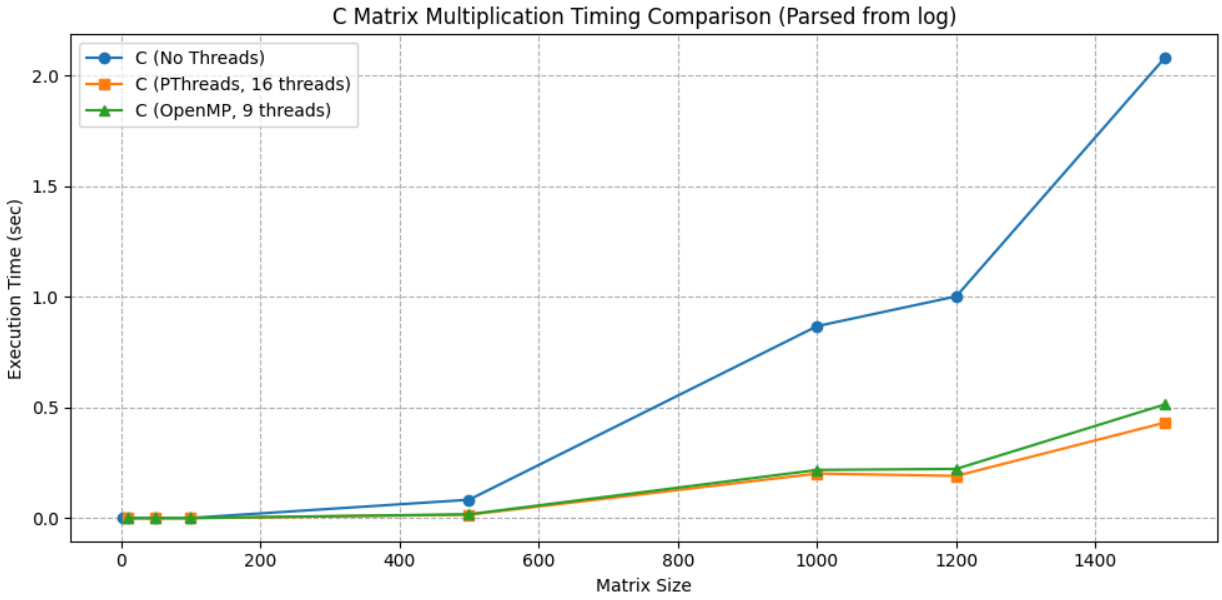


Conclusion: Execution times for small matrix sizes (3×3, 10×10, etc.) need <1ms, but once the matrix size grows to 500 and beyond, the time cost jumps significantly. Performance is initially fast for small inputs but escalates for larger matrices. By the time the size reaches 1500×1500, the execution time is nearly 20 seconds, reflecting the computational intensity scales with input size. Despite the increase, C's compiled nature and lower-level optimizations still maintain more efficient performance than higher-level languages at larger scales.



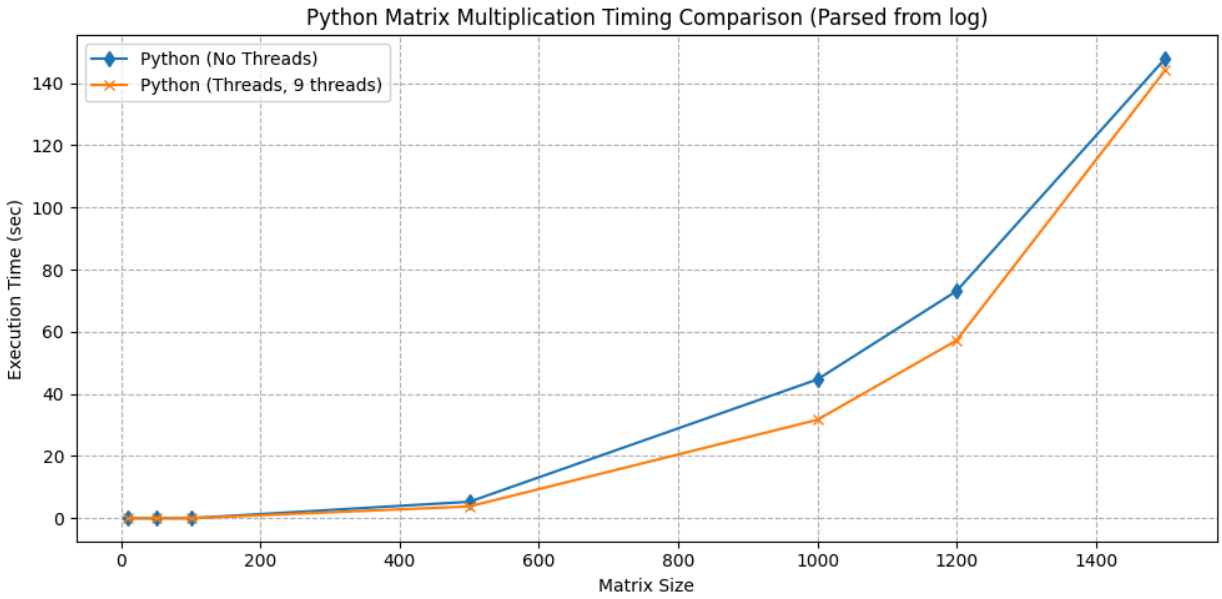
Conclusion: The Python timing graph starts off with sub-millisecond results for a 10×10 matrix but quickly becomes much more expensive as the matrix size grows, reaching over 300 seconds for 1500×1500 matrix. This steeper climb in time highlights how a high-level interpreted language and the differences in how Python handles nested loops and array operations. While Python is a straightforward 'easy' language to write and debug, the performance penalty in pure CPU-bound tasks like matrix multiplication emphasizes development convenience over raw execution speed.

Images For M3:



- **Blue (C No Threads):** Single-threaded C implementation. Execution time grows significantly once the matrix size goes 400
- **Orange (C PThreads):** Using the PThreads library for multithreading, stays low even as the matrix size increases, demonstrating how parallelizing the workload can reduce time.
- **Green (C OpenMP):** Similar to PThreads, OpenMP also provides a parallelized approach. Its performance is close to PThreads for most sizes, outperforming both the single-threaded version as the matrix size grows.

Conclusion:
Multithreaded versions (PThreads and OpenMP) scale better for large matrices compared to the single-threaded C code, resulting in lower execution times.



- **Blue (Python No Threads):** Shows the timing for Python's, single‑threaded implementation. It is very fast for small matrices but execution time rises quickly as the matrix size grows. Matrix 500×1500, it reaches over 100 seconds.
- **Orange (Python Threads):** Uses Python's threading (ThreadPoolExecutor). A clear improvement over the single‑threaded version, the execution times are still much higher than the C versions

Conclusion:
Although threading helps reduce the time in Python, Python loops limit it's performance gains compared to C. These two figures highlight the strong advantage of multithreading in C and demonstrate that Python can benefit from threading; it is still behind C's performance for numerical computations.