System:
Model name: AMD Ryzen 5 5500U with Radeon Graphics
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Address sizes: 48 bits physical, 48 bits virtual
Byte Order: Little Endian
CPU(s): 12
On-line CPU(s) list: 0-11
Vendor ID: AuthenticAMD
CPU family: 23
Model: 104
Thread(s) per core: 2
Core(s) per socket: 6

Right After Boot:

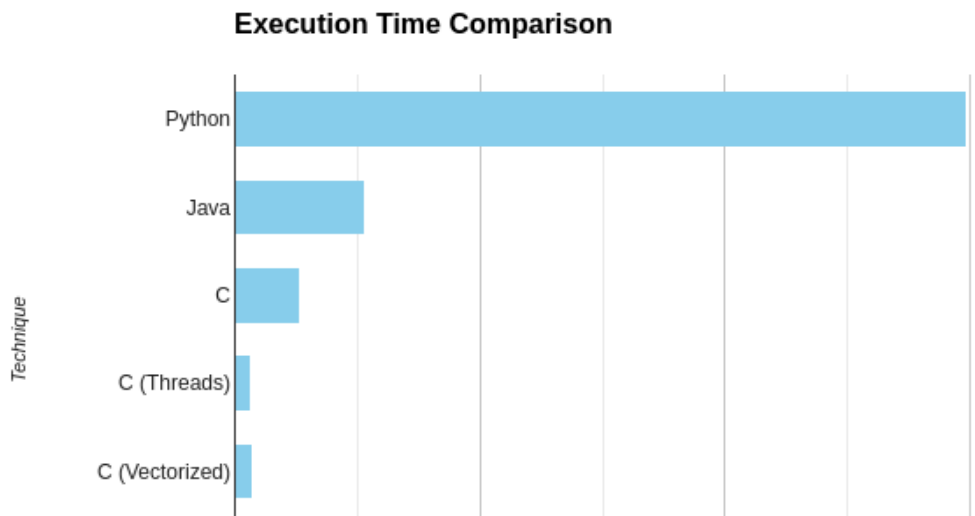
Journal:

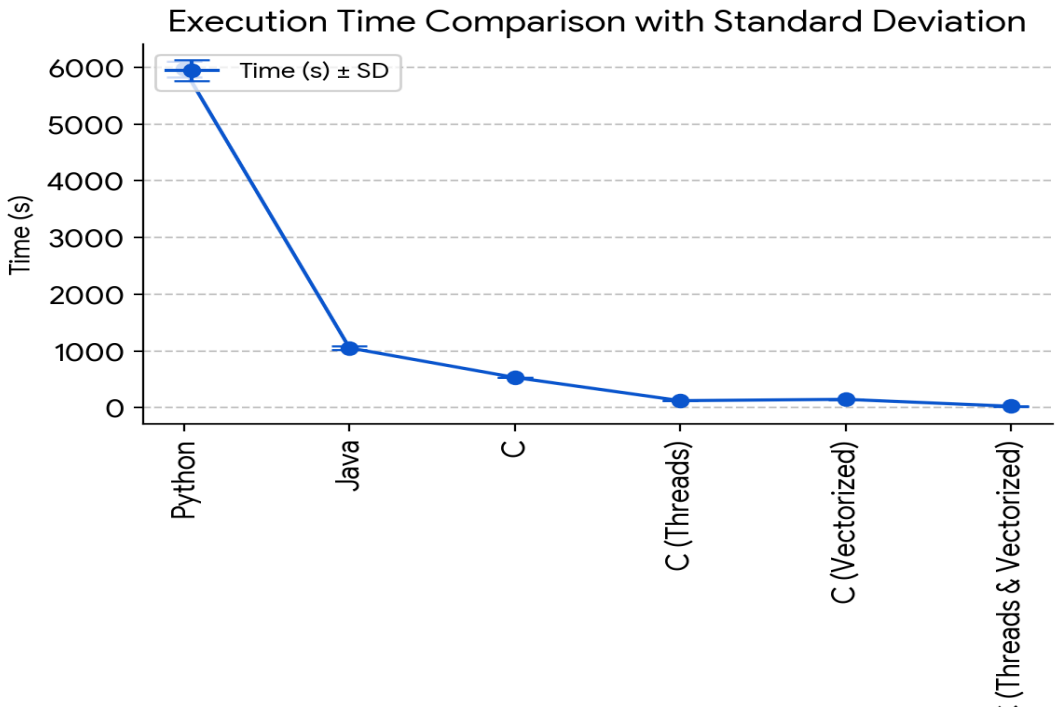
- The code was susceptible to segmentation faults due to out-of-bounds memory access and potential thread-unsafe writes to the result matrix.

- While not explicitly implemented, using mutexes or atomic operations on the result matrix can prevent simultaneous writes from multiple threads.
- I encountered attempts to assign entire arrays directly to structure members. This was fixed by assigning the address of the first element in the array.
- In vectorization of threads, code didn't consider thread safety when accessing matrices A, B, and the result matrix. This was solved by passing these matrices as arguments to the thread function within a dedicated structure.
- The main thread needed to distribute the workload among multiple threads. This was achieved by calculating the workload per thread based on the number of columns in the result matrix and handling any remaining columns for the last thread.
- The code needed to create and manage threads for parallel execution. This was implemented using pthread.create() and pthreadjoin() functions within a loop to handle thread creation and synchronization.
- Memory allocated for thread arguments and the result matrix needed proper deallocation after use. This was addressed by freeing the memory within dedicated loops after the threads finished execution. Clearing is important as large dataset such as this one quickly started causing heap problems and compiler threw fatal errors and the easiest solution was to restart the device which i assumed would clear heap and it worked.
- While multithreading introduces parallelism, the optimal number of threads depends on hardware and workload. This wasn't explicitly addressed in the code, but it's important to consider for real-world use.
- Casting integer calculations to pointers caused warnings. We addressed this by using `size_t` for memory allocation size calculations or by explicitly casting the calculation result to a pointer type.
- To enable AVX during compilation, using flags such as march-native was necessary.

.

Analysis Graph:




Execution Time Comparison with Standard Deviation
Time (s) ± SD
Time (s)
6000
5000
4000
3000
2000
1000
0
Python
Java
C
C (Threads)
C (Vectorized)
(Threads & Vectorized)

(a) What did you learn in this assignment? Did you have some interesting insights?

Key Learnings:

- **Language Nuances:**
    - **Performance:** C consistently outperformed Python and Java due to its lower-level nature and direct hardware control.
    - **Readability:** Python's cleaner syntax made algorithms easier to grasp, while Java's type safety caught potential errors early.
- **Parallelism with Threads:**
    - **Performance Improvement:** Breaking down matrix multiplication into smaller tasks using pthreads significantly accelerated execution, especially for larger matrices.
    - **Overhead Considerations:** Thread creation should be balanced with physical threads available otherwise mapping isnt effiecient.
- **Vector Instructions:**
    - **Exploitation of Hardware:** Using vector instructions (SSE, AVX, etc.) allowed me to harness CPU vector units for parallel computations within a single core, leading to substantial speedups for computationally intensive tasks like matrix multiplication.
    - **Complementary Approaches:** Combining pthreads for multi-core parallelism and vector instructions for single-core vectorization led to even more dramatic performance improvements.

Interesting Insights:

- **Language Choice:**
    - Python's simplicity often outweighed its performance limitations for smaller-scale tasks and prototyping.
    - C's performance made it suitable for demanding numerical computations.
- **Hardware Considerations:**
    - The effectiveness of vector instructions and threading varied depending on the underlying hardware architecture.

(b) Which single way gave you the maximum speed up in relative terms? Programming language change, vector instructions or multithreading?

Multithreading.

(c) The Karp-Flatt metric looks at serial part+overhead while we change the number of processors, though if we extend the definition of “processor” we might be able to use that metric. Assuming that any speed up from language change to vectors to multithreading to (vector+multithreading) is due to increasing processor count, what do your metric numbers suggest about the quality of your speed up and is it plateauing?