Setup:

The project setup was carried out on a Windows system using PowerShell for automation. FIO was installed and configured to run consistently across trials. PowerShell scripts were written to automate test execution, capture output logs, and save results in CSV format. System settings like CPU affinity and clock control were applied where possible to reduce noise and ensure repeatability. Python scripts were then used to process the collected data, compute averages and standard deviations, and generate plots for analysis. This methodology provided a structured workflow from environment preparation through data visualization.

All the operations (reads and writes) were done on an external flash drive. Due to this, it can be expected that latency and bandwidth is heavily restricted. However, even under higher latencies and bandwidth's, the comparison is valid since all operations were done on this flash drive.

**couldn't upload the .json files as they were over the upload limit of 100 files.
**use scripts\run_all.ps1 to make all files

**All code and scripts have been submitted (contain comments/greater detail on the specific implementation of each dataset).

1. **Zeroqueue baselines**

Used the specific script: scripts\run_zeroqueue.ps1

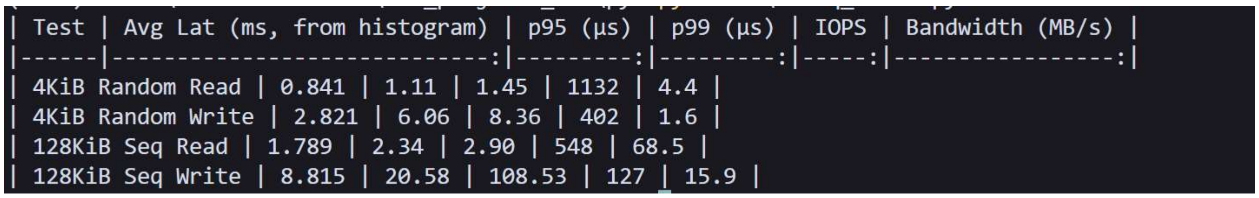
```
& $FioExe `
  --name=randwrite_XX --rw=YY --bs=4k --size=4G --iodepth=1 --numjobs=1 `
  --time_based=1 --runtime=$Runtime --ramp_time=$RampTime --direct=1
--group_reporting=1 `
  --filename=$TestFile --output=$out2 --output-format=json
--percentile_list=95:99
```

Where XX and YY differed based on specific variables we needed to plot.

Explicitly set --iodepth=1, which enforces queue depth = 1.

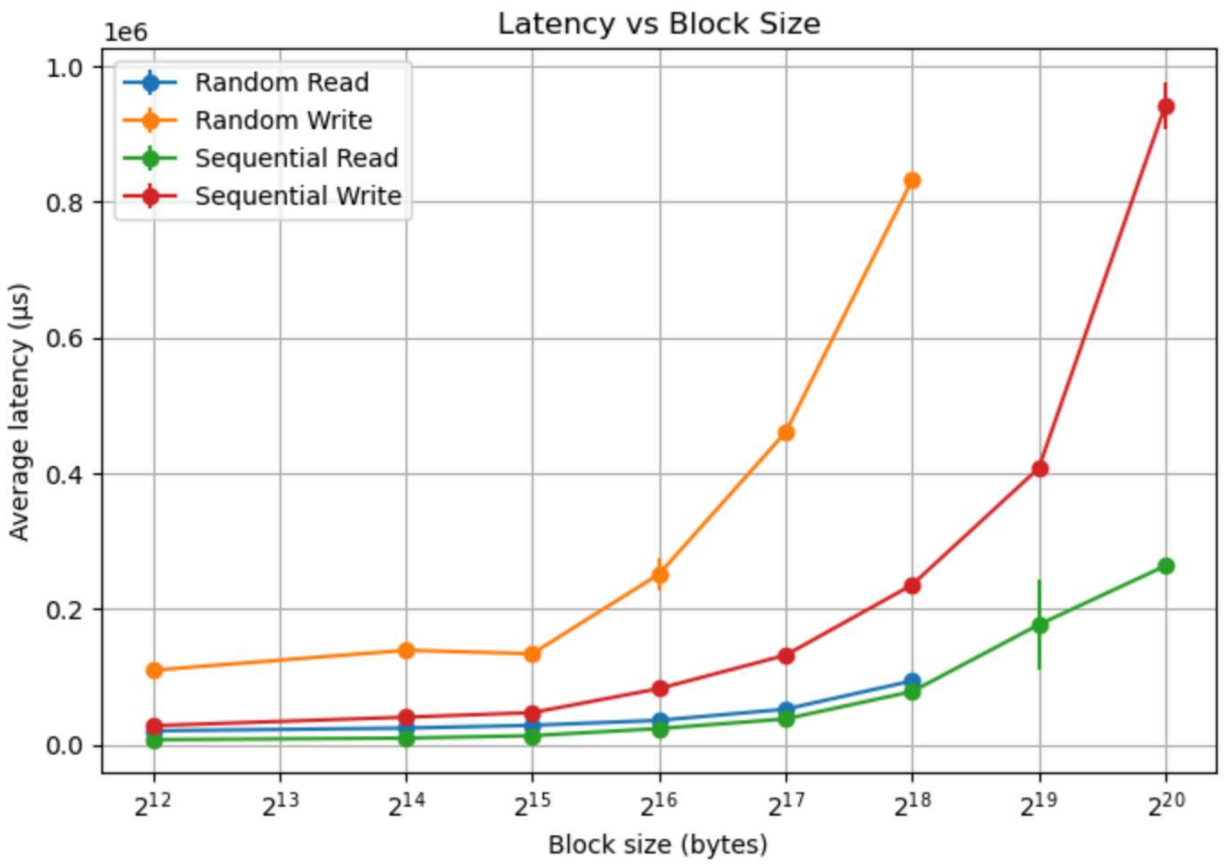
Explicitly set --direct=1, which bypasses the page cache and forces I/O to go directly to the device, ensuring you measure storage latency rather than OS caching

By running **\py\python zeroqueue_table.py,** we get the following table output:

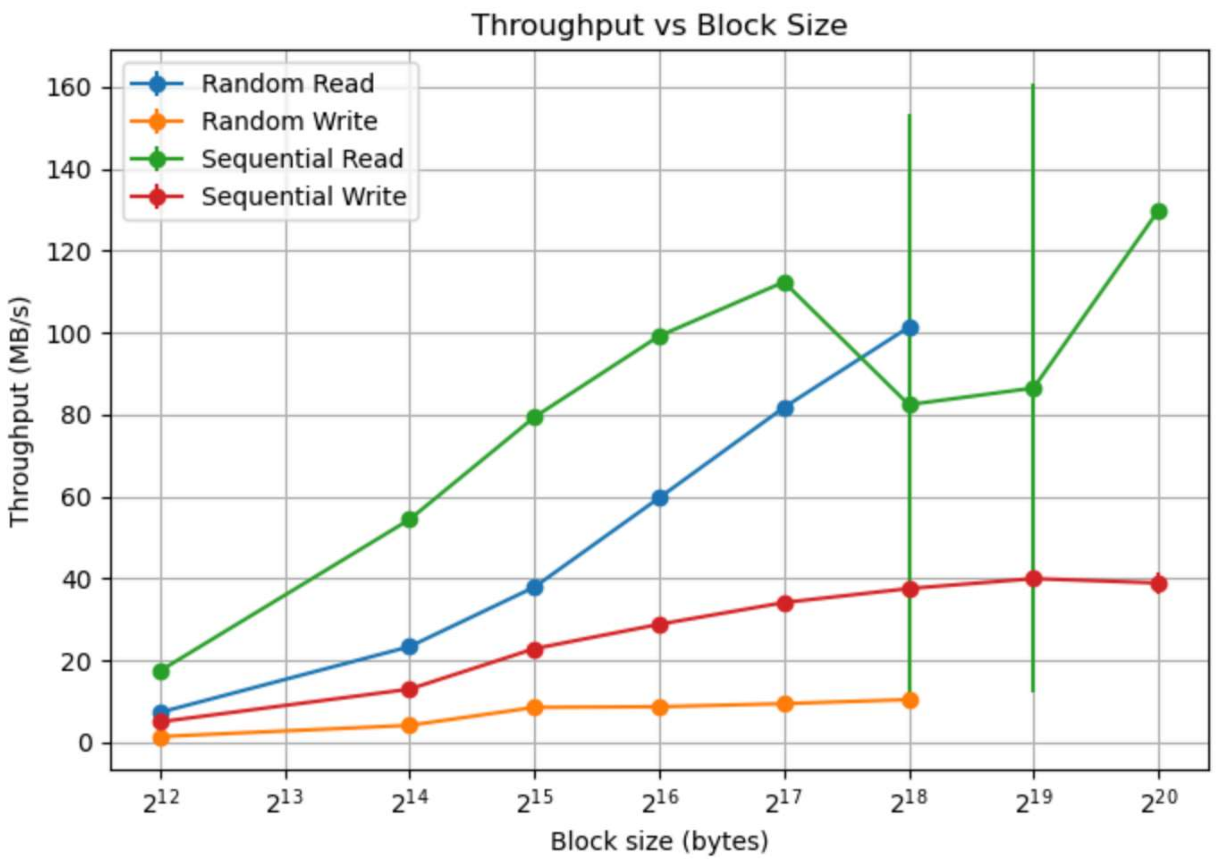


```
| Test | Avg Lat (ms, from histogram) | p95 (µs) | p99 (µs) | IOPS | Bandwidth (MB/s) |
|------|----------------------------:|---------:|---------:|-----:|-----------------:|
| 4KiB Random Read | 0.841 | 1.11 | 1.45 | 1132 | 4.4 |
| 4KiB Random Write | 2.821 | 6.06 | 8.36 | 402 | 1.6 |
| 128KiB Seq Read | 1.789 | 2.34 | 2.90 | 548 | 68.5 |
| 128KiB Seq Write | 8.815 | 20.58 | 108.53 | 127 | 15.9 |
```

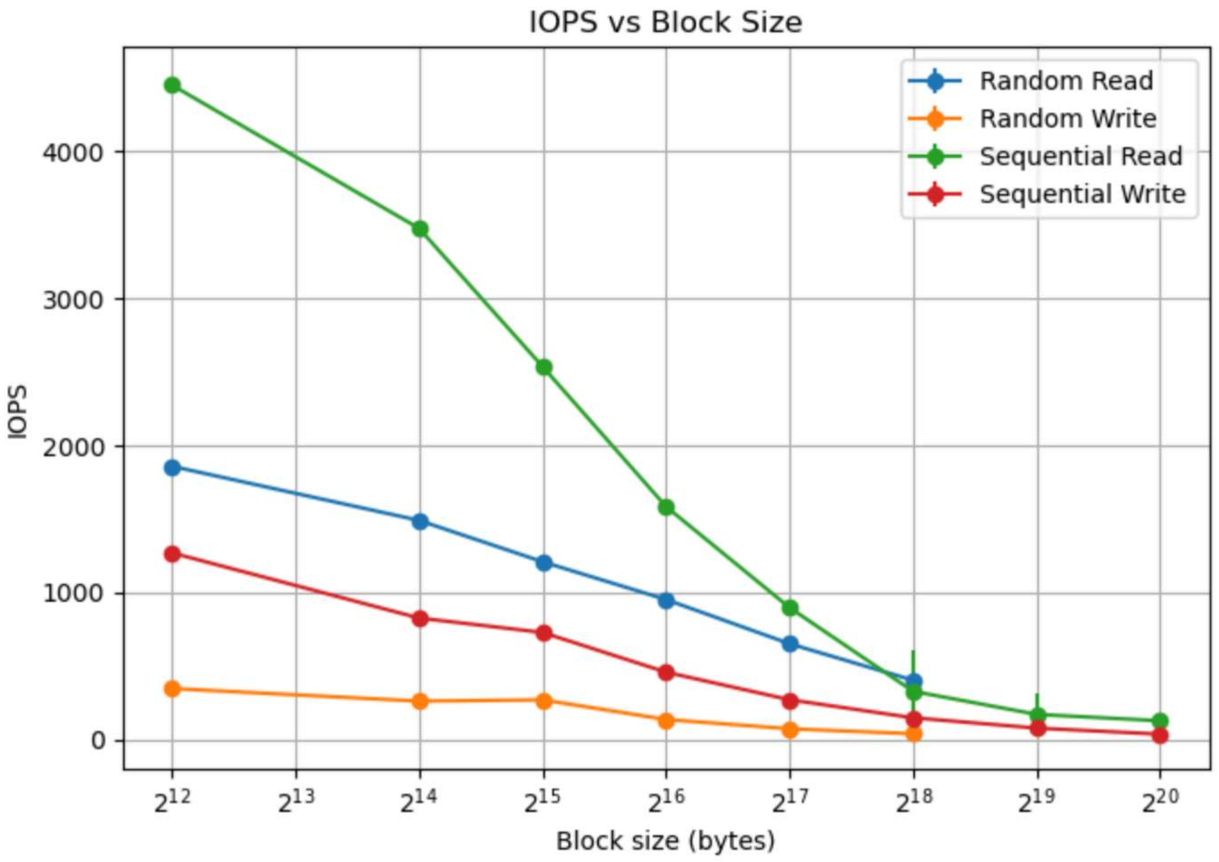
2. Block Size and Patterns sweep



The latency trends reflect both the intrinsic access cost of the media and controller-side effects like prefetching and queue coalescing. Random writes begin at the highest baseline, about 0.1 µs above the others, because each small, scattered update triggers full erase-program operations internally and cannot be coalesced early on. Sequential writes follow, as they benefit somewhat from buffering but still incur write-amplification overhead. Reads, by contrast, start lower because the processor can issue long bursts and exploit prefetching from adjacent addresses. Importantly, both writes exhibit similar exponential growth as queue depth/block size increases, while both reads scale with a smaller growth factor, illustrating the computer's ability to decrease latency through reordering and prefetching of read commands but its limited ability to hide program costs on writes.

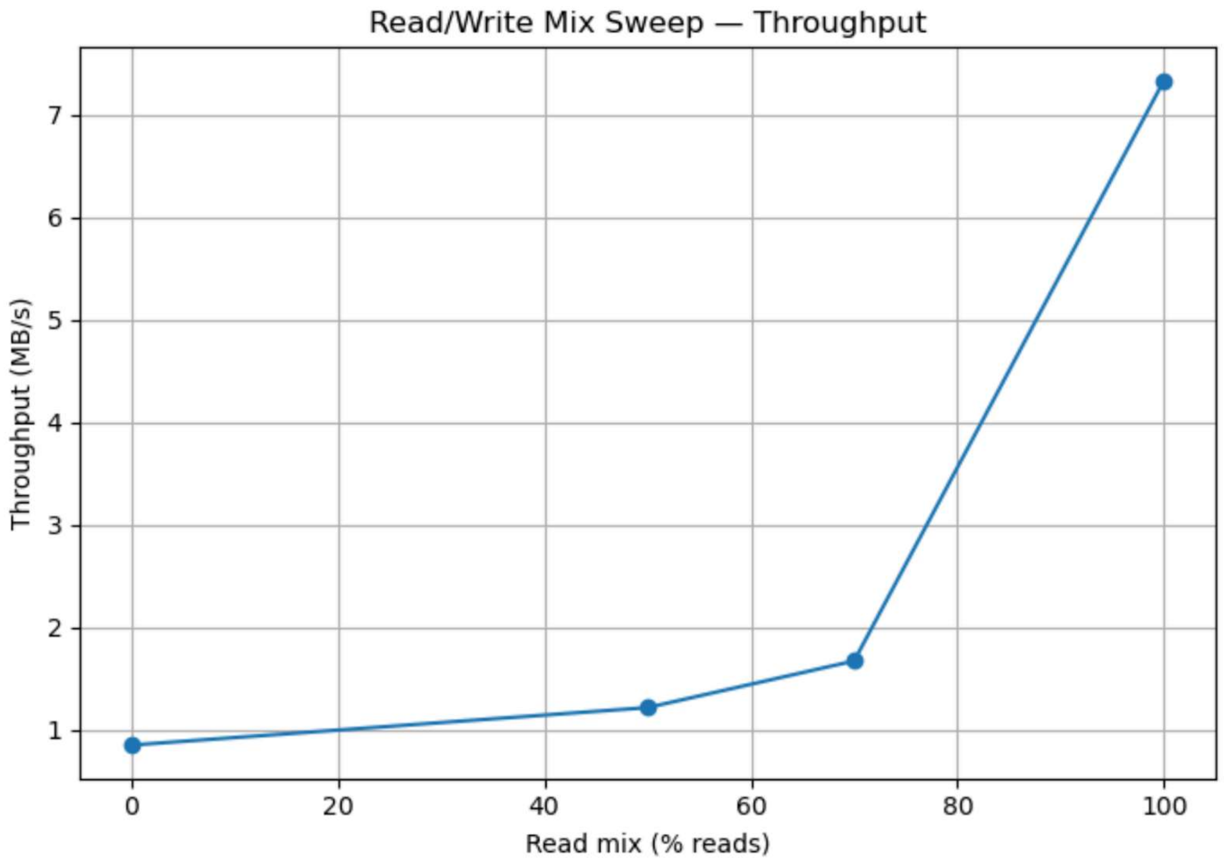


Sequential read achieves the highest slope and absolute throughput, a result of efficient large transfers and controller prefetching across contiguous addresses. Random read follows closely, demonstrating that the computer can still coalesce some random accesses into full-page operations. Both writes are lower, with random write trailing significantly since it lacks spatial locality and is bound by garbage collection and program-erase costs. All curves appear roughly linear until the 2^18–2^19 region, where sequential random drops sharply before recovering. This anomaly and the large variance suggest a computer state change: crossing an internal buffer boundary or triggering garbage collection, which temporarily reduces effective throughput. The re-rise indicates recovery once block allocations stabilize.

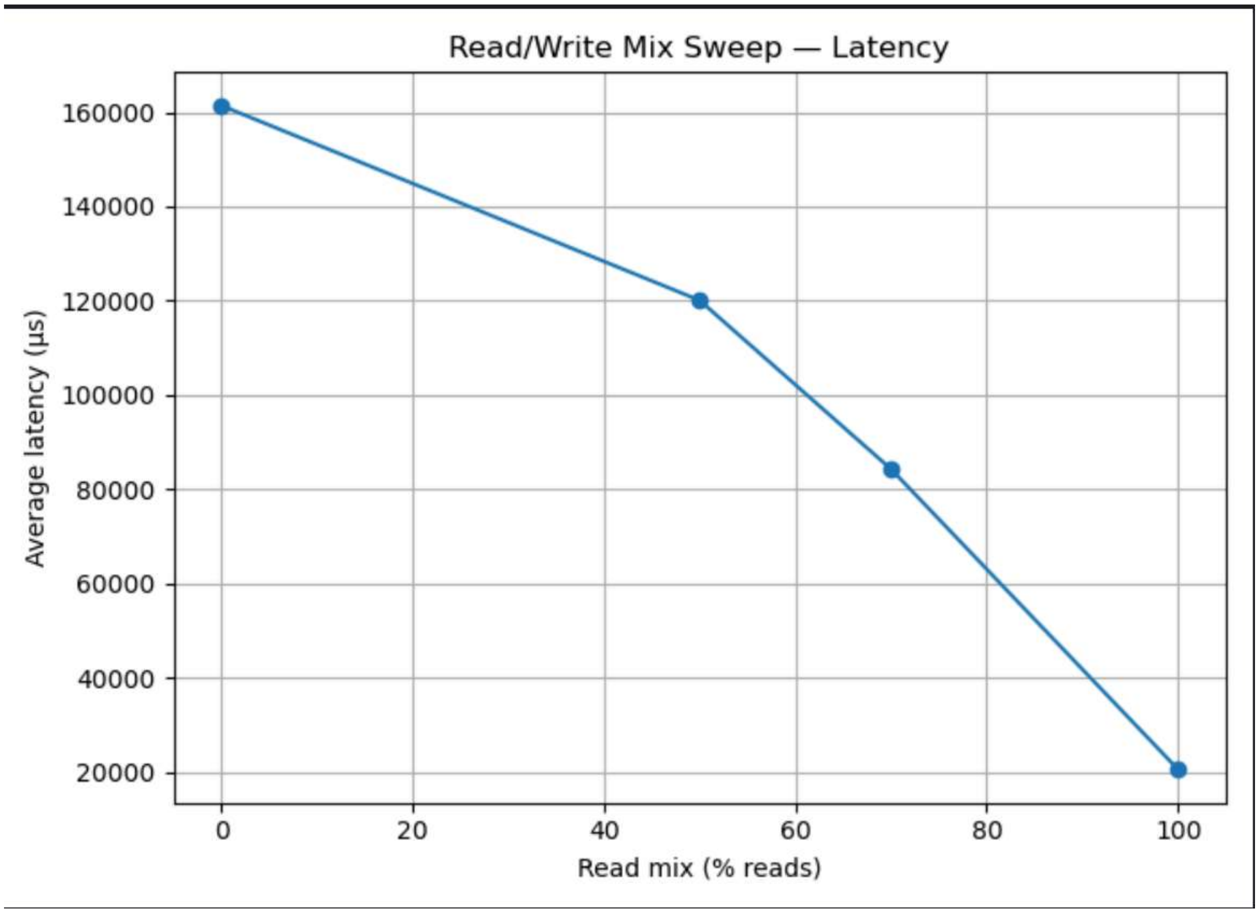


At small block sizes, read and write lines are separated, with reads much higher than writes, because per-operation overhead dominates and reads benefit from coalescing and prefetch. As block sizes grow, IOPS naturally decline on a near-linear slope (fewer operations per second for larger requests), but effective bandwidth rises. By ~2^18 bytes, all four curves group closely, with the gap between read and write shrinking. This convergence indicates the transition where per-request latency matters less, and the interface saturates on raw throughput. This graph clearly shows the initial advantages sequential read/write have in smaller block sizes, and the higher IO-rate of read instructions compared to writes.

3. **Read/Write Mix Sweep**

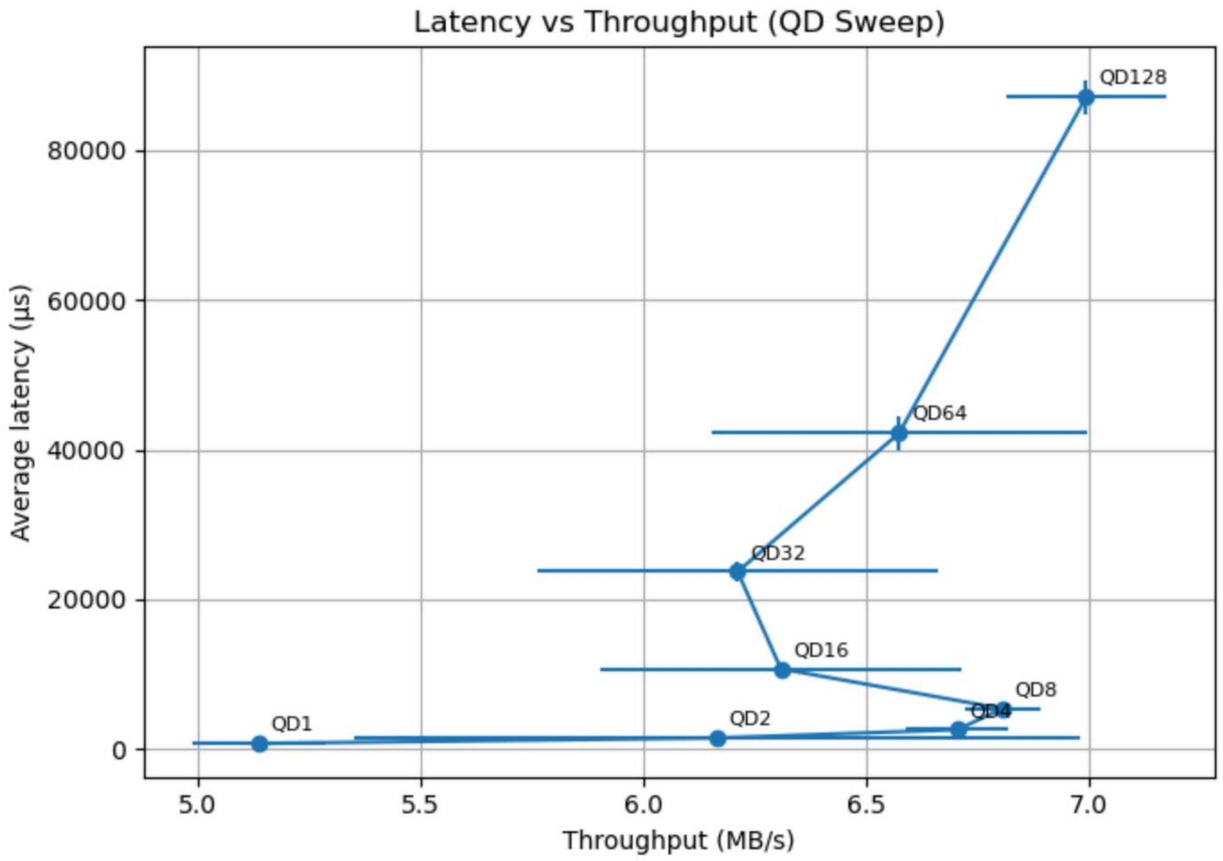


Average latency decreases as read % increases: This is because writes are more expensive than reads. Every write usually involves an erase program, which is much slower than simply fetching pages.

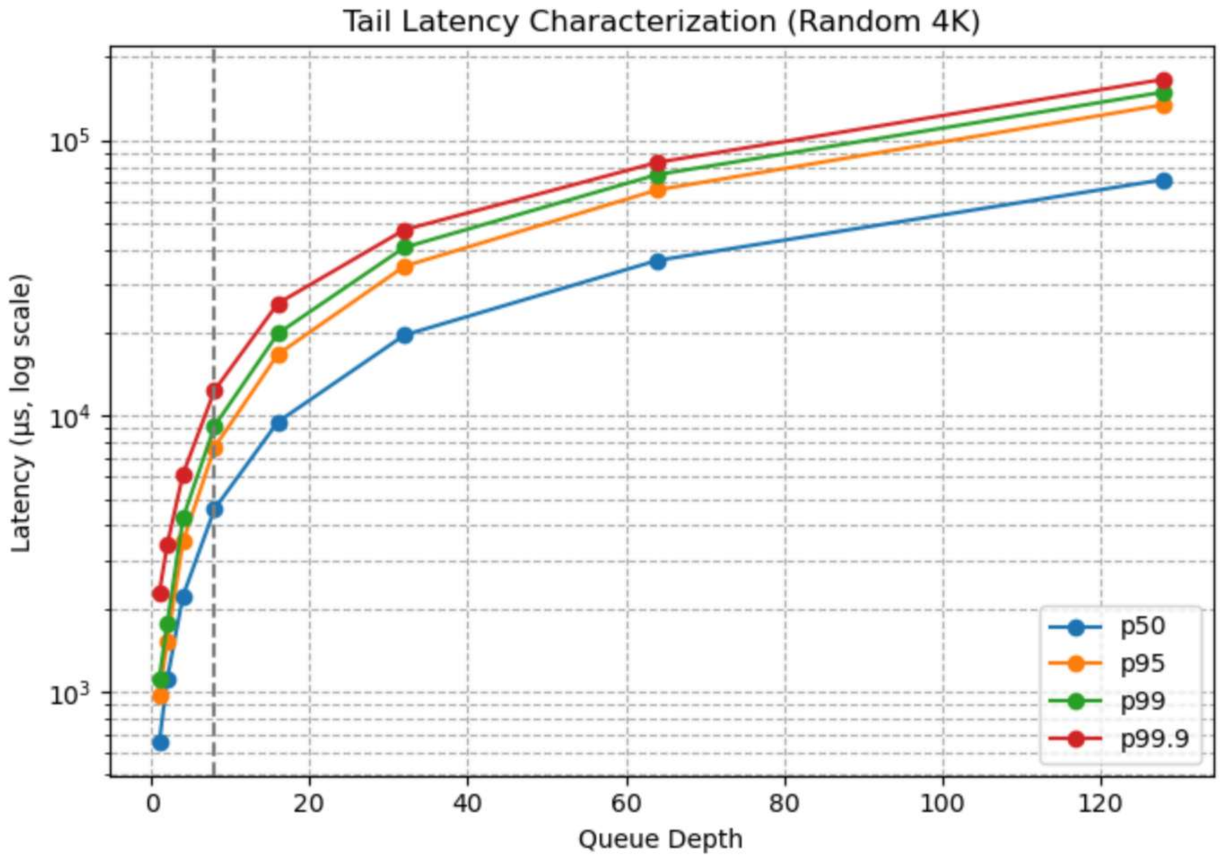


Average latency decreases as read % increases: This is mainly because reads can be done in parallel. Higher read % can support higher parallelism because SSDs can read requests across multiple channels, while writes stalls channels. This stalls this channel till the write is finished processing. Higher read % also means lower dirty pages, greatly reducing the OS's stress on garbage collection, indirectly increasing bandwidth for IO operations.

4. Queue-depth/parallelization sweep



This intensity curve deviates from the ideal case. The knee appears at QD≈8, where the drive reaches ~96% of peak bandwidth while still at ~9.3% of peak latency. The error bars are shown, for +/- one standard deviation. Beyond this point, theory predicts only marginal throughput gains at sharply rising latency; instead, we observe a throughput dip across QD16–64. This anomaly is consistent with the characteristics of an external USB flash drive: shallow internal parallelism, small/slow write caches, aggressive background management, and a much lower host-link ceiling than NVMe. Together, these factors cap sustainable service rate and can temporarily depress throughput as queue depth rises.



The tail-latency curves (p50, p90, p95, and p99.9) highlight how queue depth impacts not just average latency but also the distribution of service times. Near the knee at QD≈8, all four curves show the steepest slope, meaning that small increases in queue depth translate into disproportionately large increases in tail latency. This region offers the most balanced trade-off between throughput gain and latency cost. At lower depths (e.g., QD=4), throughput is still below peak, so we are not fully utilizing the device; at much higher depths (e.g., QD=40), the drive is already saturated, and latency rises sharply without added benefit. Thus, operating near QD=8 optimizes performance: it delivers close to peak bandwidth while keeping p95 and p99 latencies bounded, and it avoids the long-tail blowup visible at higher depths. This observation reinforces the importance of tuning queue depth around the knee to balance throughput efficiency against QoS guarantees