# **NVidia GDS on OCI: Benchmark Results & Recommendations**

## Introduction

Today, artificial intelligence is flourishing in constantly expanding spectrum of industries. As business problems solved by AI grow in scale, so does performance requirement of underlying infrastructure.

Key element of AI performance heavily influencing its overall performance remains storage. Furthermore, AI applications tend to involve different IO workloads at the same time.

To tackle the IO challenge for AI, NVidia announced GDS (GPU Direct Storage), technology for direct access of GPU to storage subsystem.

As OCI aims to deliver utmost performance, we adopted GDS on OCI Compute Shapes equipped with NVidia GPU, both [bare metal](https://docs.oracle.com/en-us/iaas/Content/Compute/References/computeshapes.htm#bm-gpu) and [virtual](https://docs.oracle.com/en-us/iaas/Content/Compute/References/computeshapes.htm#vm-gpu) ones.

In this article we are answering questions of high demand in the market.

* What is GDS, exactly?
* Does GDS really increase performance?
* What are the recommended use cases for GDS on OCI?
* What are best practices for configuring application to gain the best of GDS on OCI?

Before processing with the questions, let us agree on benchmarking methodology and T&D.

# Benchmark Methodology

TBD - Andrea

# Terms and Definitions

## GPU Modes

GDS, abbreviated GPU Direct Storage, is a technology to provide a shortcut from GPU to storage, in our case local NVMe SSD. The idea of such shortcut is to bypass CPU, thus reducing potential delays and offloading CPU cores.

GPU operates in one of the following IO modes:

|  |  |
| --- | --- |
| 0 | GPU\_DIRECT |
| 1 | CPU\_ONLY |
| 2 | CPU\_GPU |
| 3 | CPU\_ASYNC\_GPU |
| 4 | CPU\_CACHED\_GPU |
| 5 | GPU\_DIRECT\_ASYNC |
| 6 | GPU\_BATCH |

Technically, all modes involving CPU are not GDS. However, we will include CPU\_ONLY and CPU\_GPU into consideration to have a solid baseline for the benchmarks which will tell us if GDS actually brings advantages compared to CPU modes, and how great it is, if any.

Having said that, we will investigate the following GPU modes in this article:

CPU\_ONLY, CPU\_GPU, GPU\_BATCH, GPU\_DIRECT, GPU\_DIRECT\_ASYNC.

## Visual Style

### GPU Modes

To make benchmark result charts as informative and concise as possible, we will apply a rich visual style which takes a bit of explanation. Practically, however, the visual style proves natural and easy to understand.

GPU modes are visualized with their own colours.

|  |
| --- |
| CPU\_GPU |
| CPU\_ONLY |
| GPU\_DIRECT |
| GPU\_BATCH |
| GPU\_DIRECT\_ASYNC |

### Performance

Performance is visualized in shades of green, red, and yellow.

In particular, top 10% performance results are marked bright green, and bottom 10% bright red:

|  |
| --- |
| Top 10% |
| Bottom 10% |

“Top” and “bottom” correspond to the metric of a given chart, either latency or bandwidth.

Lighter shades of green and red stand for intermediate performance between these areas of 10% extreme results.

Additionally, some charts require deeper detail of visualization. In such charts, colour spectrum is extended:

* Bottom 10% is moved from bright red to dark red, with correspondent enrichment of intermediate shades of red for poor side of performance
* Shades of yellow are included for intermediate performance between shades of green and red and aren’t tied to any specific percentage range of performance
* Most importantly, extended colour spectrum marks each decile of the performance with its own colour

### Performance Extremes

Wherever a chart includes quantified performance, performance extremes on the chart are marked bold white within correspondent area (either green for top performance or red for poor performance).

## Performance Units

Mostly, performance units are explicitly given in the charts or tables. For the sake of nice and clean formatting, however, units are omitted sometimes. In all such cases bandwidth is measured in KB/sec, latency in microseconds, and block size in KB.

# Benchmark Results

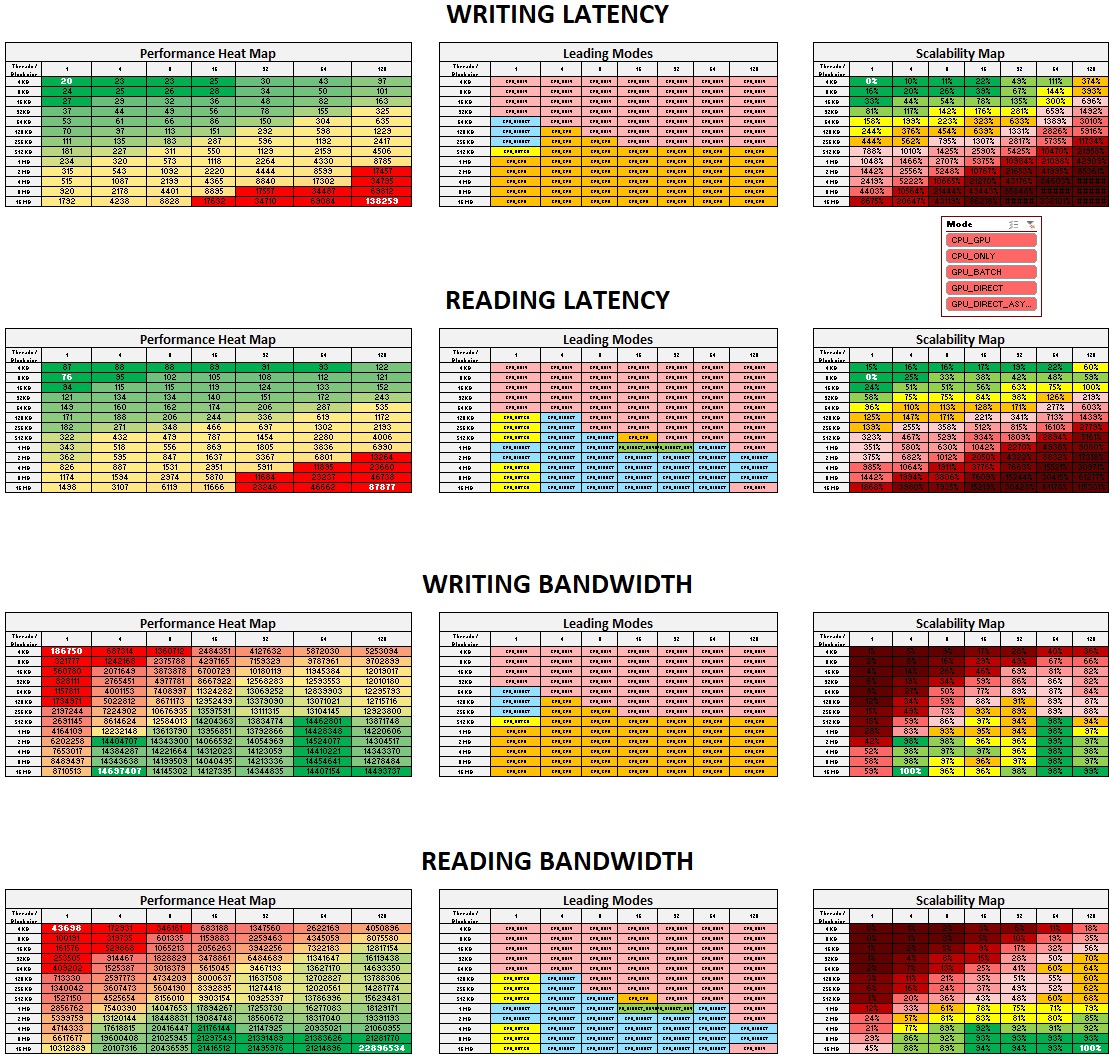
We are representing benchmark results for any IO metric (reading latency, writing latency, reading bandwidth, writing bandwidth) as a combination of three tables.

* **Performance Heat Map** to demonstrate actual performance numbers and its dependency from workload parameters (block size, number of threads)
* **Leading Modes** to demonstrate which GPU mode provides best performance for a given combination of workload parameters
* **Scalability Map** to demonstratethe way performance degrades as input parameters drift away from those that provide best performance

It is comfortable to analyse performance in the order as given above to deep-dive from high-level overview to details.

## GPU Modes Altogether

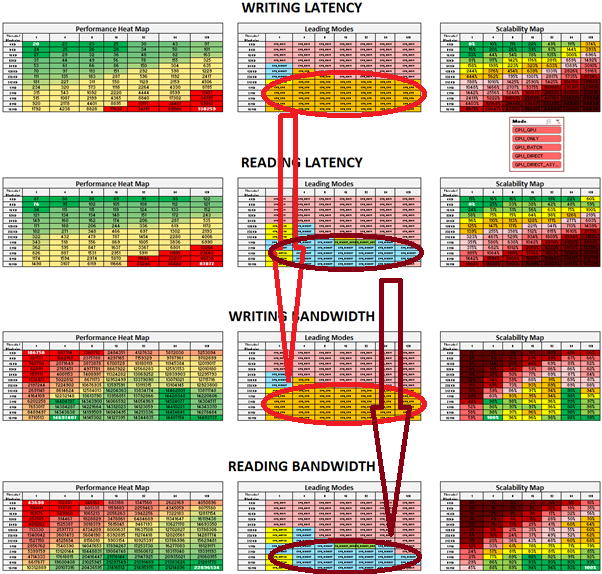
First, let us compare GPU IO modes altogether (including those baseline modes that involve CPU and are not GDS as such).



Notes based on the overview are the following.

* Regarding Latency
  + Latency better scales by number of threads than block size. Namely, up to 8 threads keep latency within 20% proximity of its best. Increase of block size, however, immediately falls out of 20% proximity.
  + Curiously, best block size differs for reading (8KB) and writing (4KB).
  + For lower half of block sizes, CPU\_ONLY is absolute winner
  + For higher half of block sizes, best latency is provided by different GPU modes
    - CPU\_GPU for writing
    - GPU\_DIRECT for reading
* Regarding Bandwidth
  + Reading bandwidth better scales by number of threads than block size. Namely, any number of threads from 4 to 128 keeps performance within 20% proximity of its best, while only one step down of the block size from 16M to 8M keeps it within same proximity.
  + Curiously, writing bandwidth doesn’t scale comprehensibly. In particular, its almost-best results are spread across non-contiguous areas of parameters.
  + For lower half of block sizes, CPU\_ONLY is absolute winner
  + For higher half of block sizes, best latency is provided by different GPU modes
    - CPU\_GPU for writing
    - GPU\_DIRECT for reading

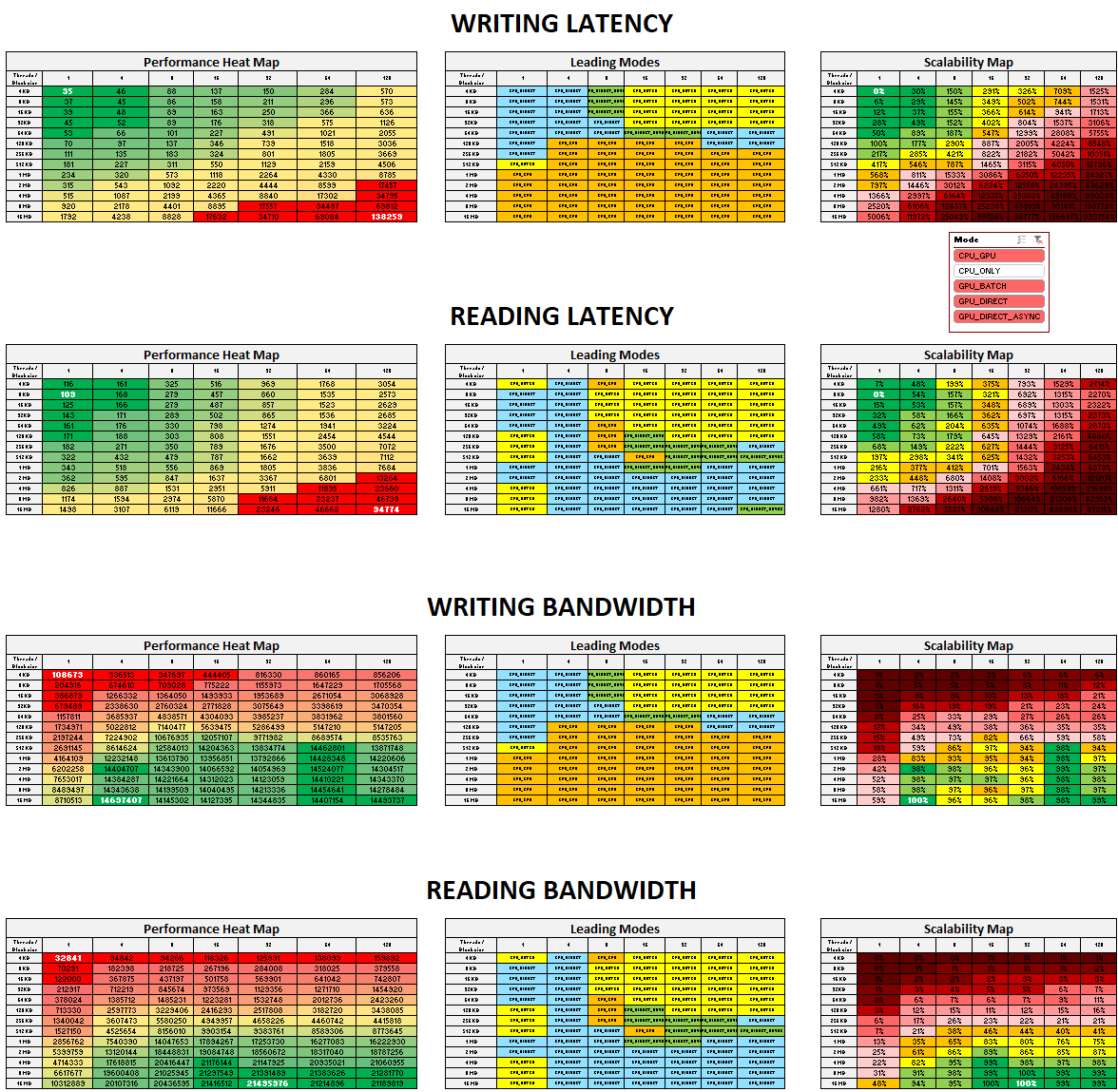
As we compared full spectrum of GPU modes, non-GDS included, the final conclusions are the following:



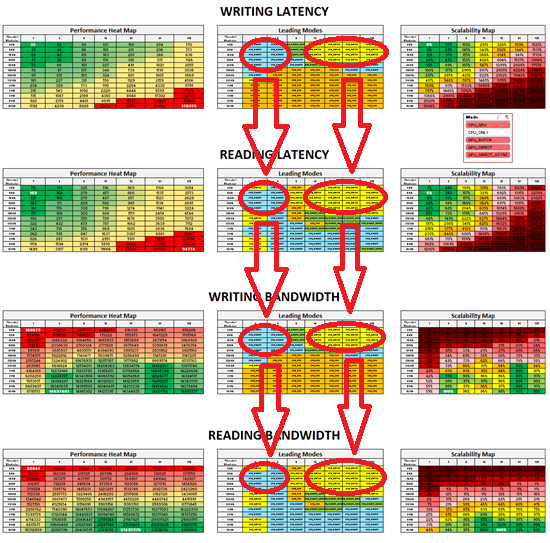
1. GPU\_DIRECT brings absolute best READING performance at higher half of block sizes, both for latency and bandwidth. It scales up to 8 threads with 20% proximity of its best and doesn’t scale by block size.
2. CPU\_GPU brings absolute best WRITING performance at higher half of block sizes, both for latency and bandwidth. Its scalability looks contradictory.

## GPU-specific Modes Only

Now we are excluding CPU\_ONLY baseline mode from consideration to deeper investigate features of GPU-specific modes.



The big picture has changed as the total domination of CPU\_ONLY disappeared in lower half of block sizes and replaced by two new distinct areas of input parameters:



* GPU\_DIRECT performs best in lower half of block sizes & lower half of number of threads, regardless of operation and metric
* GPU\_BATCH performs best in lower half of block sizes & higher half of number of threads, regardless of operation and metric

Absolute best extremes of the performance.

* As for latency, GPU\_DIRECT delivers absolute best latency regardless of operation
* Latency allows moderate scaling by block size only. Namely, 20% proximity area is kept for 4-16KB block sizes and doesn’t allow for multithreading at all.
* As for bandwidth, GPU\_DIRECT delivers absolute best reading, and CPU\_GPU delivers absolute best writing
* Scaling of bandwidth is contradictory.

Absolute best performance numbers for truly GPU-specific modes are these.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Absolute Best GPU Mode | | | | | **Comparison to CPU\_ONLY baseline** | |
| Metric | **Value** | **GPU Mode** | **Block Size** | **Number of Threads** |
| Writing Latency | **35 microsec** | GPU\_DIRECT | 4 KB | 1 | **75% slower** | **20 microsec** |
| Reading Latency | **109 microsec** | GPU\_DIRECT | 8 KB | 1 | **45% slower** | **76 microsec** |
| Writing Bandwidth | **14.7 GB/sec**  (14,697,407 KB/sec) | CPU\_GPU | 16 MB | 4 | **CPU\_GPU is absolute best** | **14.7 GB/sec**  (14,697,407 KB/sec) |
| Reading Bandwidth | **21,5GB/sec** (21,495,976 KB/sec) | GPU\_DIRECT | 16 MB | 128 | **6% lower** | **22.9 GB/sec**  (22,896,534 KB/sec) |

The table above gives a clear understanding of the performance difference when the data truly flows from GPU to NVMe compared to the “cheat” with CPU in the middle (presumably, CPU is able of smarter caching techniques based on sophisticated algorithms of Linux kernel IO subsystem).

1. Latency is by far higher on GPU/NVMe route compared to GPU/CPU
2. Bandwidth is virtually the same at GPU/NVMe and GPU/CPU routes

Now, let us put best performance parameters for GPU-specific modes together.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Best Performance Numbers per GPU Mode | | | | | | | | | | |
| Metric | GPU\_DIRECT | | GPU\_DIRECT\_ASYNC | | GPU\_BATCH | | CPU\_GPU | | CPU\_ONLY (baseline) | |
| Value | Block Size / threads | Value | Block Size / threads | Value | Block Size / threads | Value | Block Size / threads | Value | Block Size / threads |
| Writing Latency, microsec | 35 | 4 KB / 1 | 77 | 4 KB / 1 | 79 | 8 KB / 1 | 46 | 4K / 1 | 20 | 4KB / 1 |
| Reading Latency, microsec | 109 | 8 KB / 1 | 136 | 8 KB / 1 | 116 | 4 KB / 1 | 118 | 8K / 1 | 76 | 8KB / 1 |
| Writing Bandwidth, KB/sec | 5,180,706 | 1 MB / 8 | 5,171,187 | 512 KB / 16 | 5,071,804 | 2 MB / 16 | 14,697,407 | 16M / 4 | 14,171,452 | 16M / 128 |
| Reading Bandwidth, KB/sec | 21,495,976 | 16 MB / 32 | 20,929,270 | 16 MB / 128 | 17,750,841 | 16 MB / 64 | 14,778,180 | 8M / 64 | 22,896,534 | 16M / 128 |

# Recommendations

In this section, we are putting all the conclusions together and map them onto most demanded questions, both actual and expected, from our customers.

**Q: Why do you think your low-level IO benchmark results are relevant for my high-level GPU application?**

A: As low-level as it is, the IO benchmark sets hard limits for the performance of any workload placed onto the correspondent OCI Compute Shape. Since the hard limits are imposed by hardware configuration, any application will have to respect them and impose its own individual performance limits within the hard limits, not overcoming them. That said, we are elucidating theoretical limits of IO performance as well as universal steps to get as close to those limits as possible without knowing specifics of your application. All further optimizations specific for your application are up to you, they may or may not get you even closer to the same hard limits.

However, the IO benchmark is still an algorithm coded as software. Moreover, the IO benchmark is designed by NVidia, and as such is a black box for us. Therefore, there is always a chance for your application to demonstrate even more efficient software architecture than the IO benchmark. If you find a way to challenge the limits with your algorithm, do not hesitate to let us know 😉

**Q: What maximum IO performance should I expect on my OCI 1xA100?**

A: You should plan your architecture based on the following average numbers

* Writing latency 35 microseconds
* Reading latency 109 microseconds
* Writing bandwidth 14.7GB/sec
* Reading bandwidth 21.5GB/sec

Note that these numbers require special configurations of block size and number of threads. If you have no freedom to set these parameters, you are likely to have lower performance (much lower, possibly), and the only optimization available for you will be choosing most relevant GPU mode.

**Q: What are those GPU modes you mention in your recommendations?**

A: GPU can operate in one of predefined IO modes, each of them optimized for a certain workload. Intentionally simplified yet practical introduction of IO modes on OCI is that

* If parameters of your workload are predefined
  + GPU\_DIRECT is the best at small block size <=64KB and small number of threads <=4
  + GPU\_DIRECT is the best at reading of large blocks >=2M
  + GPU\_BATCH is the best at small block sizes <=32KB and big number of threads >=16
  + CPU\_GPU is the best at writing of large blocks >=512KB
* If you are tuning parameters of your workload to hit absolute maximum performance
  + GPU\_DIRECT is designed to maximize latency
  + GPU\_DIRECT is designed to maximize reading bandwidth
  + CPU\_GPU is designed to maximize writing bandwidth

**Q: I have OCI 1xA100 and I need to run GPU workload to read and/or write a great throughput of tiny files. How do I maximize IO performance?**

A: You should enable GPU\_DIRECT and read/write in as few threads as possible, 1 thread in ideal case unless your throughput mandates a greater number of threads. In that case you may go up to 4 threads and assess the trade-off of higher latency versus multiplied number of threads. Beyond 4 threads, degradation of the performance is too significant to bring benefit.

Optimal block size depends on your workload:

* If you read from/write to same files, you can take 8KB block size as a good choice both for reading and writing
* If, however, you read from/write to different files, you may test slightly more complex configuration with separate storage for reading (8KB block size) and writing (4KB block size) to try hitting highest individual performance of each operation
* If your workload favours one operation over another, you may favour that operation by either separating storage as mentioned above or optimizing block size of your single storage for that operation

In any case, beyond 32KB block size, degradation of the performance is too significant to bring any benefit.

**Q: I have OCI 1xA100 and I need to run GPU workload which reads and/or writes huge files. How do I maximize IO performance?**

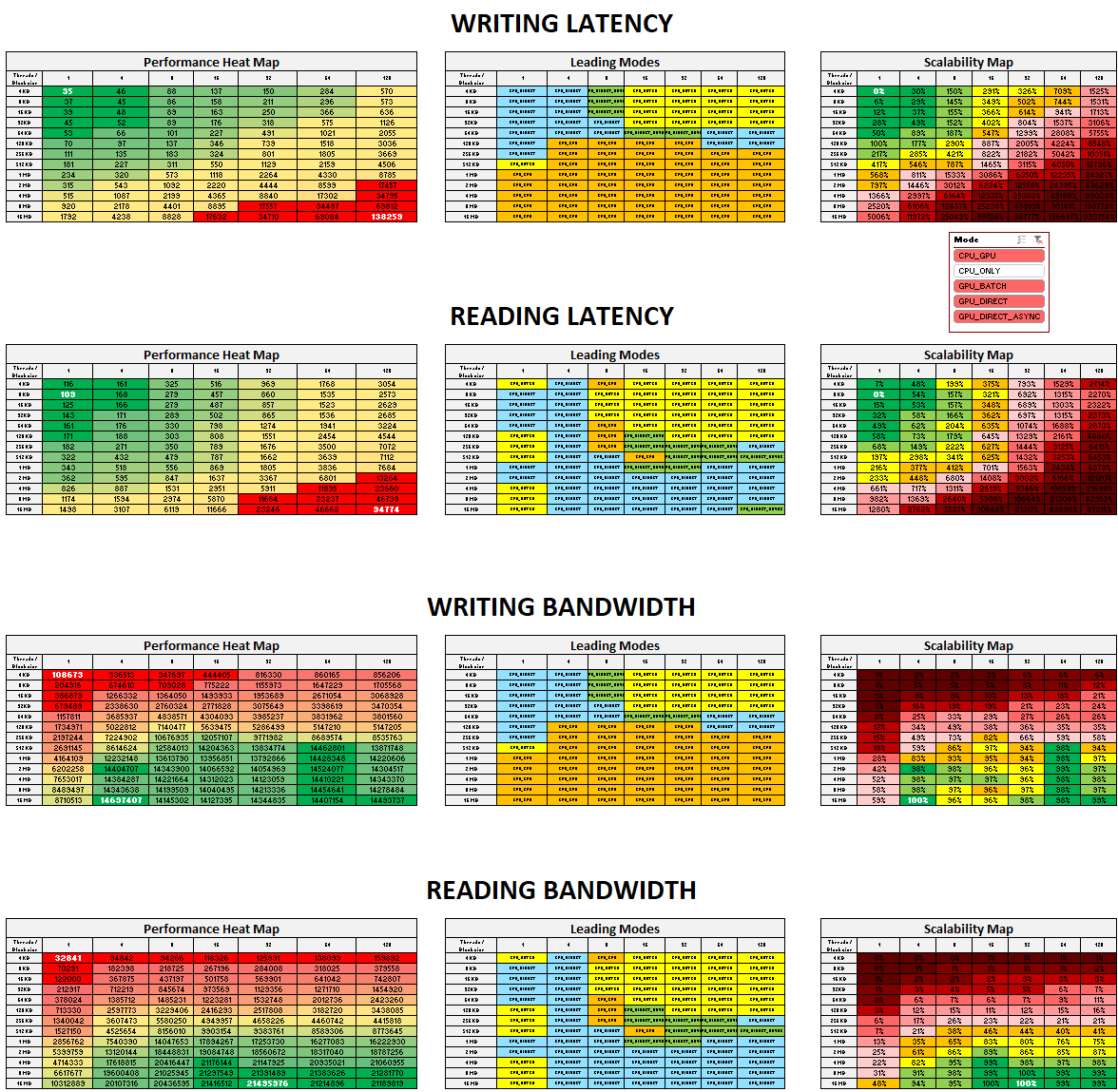
A: It depends on which operation prevails in your workload, reading or writing. The rules are the following.

* If you only need to read huge files, you should enable GPU\_DIRECT, set 16MB block and run 32 threads to have 21.5GB/sec
* If you only need to write huge files, you should enable CPU\_GPU, set 16MB block and run 4 threads to have 14.7GB/sec
* Finally, if you evenly read/write huge files, you should enable CPU\_GPU, set 8M block size and run 64 threads to have 14.45GB/sec writing and 14.78GB/sec reading

**Q: I have OCI 1xA100 and I am not in a position to change some parameters of my workload (block size, number of threads). Do I still have an option to raise performance?**

A: You should take the following steps based on Performance Charts below.

* Based on your tuple “Operation / Block size / Number of threads”, locate your workload on relevant Performance Heat Map
* Move to the relevant Leading Modes Map and enable leading GPU mode for your workload
* Finally, move to Scalability Map and check area in the proximity of your workload. Scalability map demonstrates deviation of the workload compared to absolute best (marked white). This will tell you A). how far your performance is from the best performance, B). if changing your parameters worth the effort, C). which parameter is worth changing most (or both maybe)



**Q: My application doesn’t seem to benefit from the tunings you described here. Why so?**

A: Your application doesn’t seem to utilize hardware resource effectively enough as to approach the described hardware limits. If you modify and/or reconfigure your application to do so, the tunings will have the greater effect the closer you approach the hardware limits. You are welcome to contact us and let us analyse performance of your app. When we disclose its bottlenecks and suggest ways to eliminate them, you will unleash previously hidden performance of your application.