GCSfuse - Tuning and best practices for AI/ML workloads

Created Jun 30, 2024
Last updated: Jun 23, 2025

go/gcsfuse-tuning
Shared under NDA

THIS DOCUMENT IS NOW OFFICIALLY EXTERNALLY PUBLISHED ON CLOUD.GOOGLE.COM:

- **PERFORMANCE TUNING BEST PRACTICES**
- **SAMPLE YAMLS**
- **AUTOMATED CONFIGURATIONS FOR HIGH PERFORMANCE MACHINE TYPES**
    - Starting with GCSfuse 3.0, a subset of default high performance settings are auto applied for certain high performance machine types.

ALL FUTURE UPDATES WILL BE MADE DIRECTLY ON CLOUD DOCUMENTATION WEBSITE. THIS DOCUMENT WILL NO LONGER BE KEPT UP TO DATE

Change log

Date	Change
Jun 30, 2024	Originally created
Jul 10, 2024	Configuration tested at 1000 node scale, emulating 10k GPUs. Added metadata prefetch (#6), parallel download for serving (#7). Added that checkpointing needs the rename-dir flag (#9), with the guidance that in general we do not currently recommend GCSfuse be used for checkpointing at scale.
Jul 17, 2024	Added list cache (#10), separate YAML examples for Training vs Serving
Jul 22, 2024	• YAML file example separated into serving version and training/checkpointing, updated for List Cache. • Substantial updates to “Summary of the Goals In Tuning” Section
Jul 23, 2024	Debug level logging: • Removed ‘debug_fuse’ from sample YAML under mountOptions. This will improve performance and reduce costs from having trace level logging. We may need this enabled in the future for debugging purposes. • Added #11 on how to enable debug logging in future if needed

Aug 2, 2024	Minor updates
Nov 20, 2024	Added: HNS (GKE 1.31.1-gke.2008000 or higher), Checkpointing, TPU vs GPU considerations, updated GCSfuse version Moved: 'Summary of Goals', and outdated recommendations that are now automatically implemented in code with new versions to 'Other Tab'.
Nov 26, 2024	Added Checkpoint YAML
Dec 13, 2024	Added new automated Metadata prefetch (GKE 1.31.3-gke.1162000 or higher) Requires Training and Serving YAMLs to be deployed in two steps. Added RAM disk instructions, which can be used for the cache and parallel download directory for TPUs that do not have LSSD. Training, Serving, and Checkpoint YAMLs have been updated, with it commented out.
Jan 22, 2025	Checkpoint YAML modified to include metadata cache
Feb 10, 2025	Added kernel read ahead increase to 1MB for serving and checkpoint restore
Feb 20, 2025	Added negative stat cache for checkpointing
Feb 26, 2025	Added streaming write for checkpointing
Apr 1, 2025	Added STS quota
Apr 24, 2025	Major changes: Added JIT cache, Negative Stat Cache set to 0 (disable) now recommended across all, kernel list cache not enabled in all YAMLS (use only if data is not changing) . Updated YAMLs
Jun 23, 2025	This doc has now been posted externally. It includes sample YAMLs, and automated configs starting with GCSfuse version 3.0.

Version Guidance: This guide applies to GCSfuse >= v2.9.1 and the GCSfuse GKE CSI driver running on GKE clusters of version GKE version 1.32.2-gke.1297001 or greater.

IMPORTANT NOTE: This document provides specific configuration recommendations for AI/ML workloads that have been validated for GPU and TPU large machine types at scale. For example, where a single GKE pod consumes an entire A3-Ultra machine or a single TPUv5p machine. In these environments there is a lot of RAM and a high bandwidth network interface (200 gbit/sec). Use with caution on any other machine type.

Further, all of the recommendations cache GCS metadata for extended periods of time (the length of the job) - thus it will no longer be checked after the initial mount of the filesystem. This works best if the filesystem is read only or the filesystem semantics are "write-to-new" -. applications always write to new files (i.e. they do not overwrite files). The normal pattern for the following AI/ML workloads have been observed as "write-to-new".

- **Checkpoints** (e.g. JAX Orbax checkpoints, Pytorch checkpoints, Pytorch torch.distributed async checkpoints)
- **Training data**, including PyGrain and Pytorch NeMo training data
- **Inference/Serving** of model weights
- **JAX JIT Cache** - There is a minor variation here if access time is enabled. Under this condition, multiple writers may attempt to update the same access file, but in the end only one will succeed, so it is still always one file that gets updated (i.e. never over-written).

During troubleshooting please enable Debug Logs. Once troubleshooting has completed, please ensure it is then disabled, as there is a performance and cost impact.

For your convenience, we have added a sample YAML file with best practices already configured here. You can copy and paste this, with at minimum modifying just the bucket name.

GCSfuse Overview

GCSfuse is a Google 1st party, supported, FUSE driver for Cloud Storage (GCS), that allows users to run AI/ML workloads (and other workloads) at scale by efficiently accessing GCS storage as a locally mounted filesystem. It allows users to take advantage of the scale, affordability, throughput, and simplicity that Google Cloud Storage provides, while maintaining compatibility with applications that use filesystem semantics without having to refactor applications to use native GCS APIs. Objects appear as local files, which are read as a stream rather than having to download to local storage and then read.

The GCSfuse file cache feature enables transparent caching of files and can also be used as a large prefetch buffer for very fast parallel download of files. On GPUs, the file cache can be on local SSD (A3 machine types come with 6TB of local SSD (LSSD) bundled) or on a RAM disk (/tmp/fs). On TPUs, the file cache would typically be on a RAM disk (zero incremental cost), but if not enough RAM is available, either a PD or Hyperdisk can be used for additional cost (PD/Hyperdisk configurations are out of scope for this document). The file cache enables serving repeat reads at local SSD performance for AI/ML training workloads and also significantly improves model load time for inference and checkpoint load times for training due to the parallel download feature. TPUs, which don't include LSSDs, should use a RAM disk for parallel downloads with serving workloads, and should only consider it for file caching in training workloads if enough machine memory is available.

Additional product links:

1. GCSfuse overview & documentation landing page
2. GCSfuse flags
3. GCSfuse Github page

4. GCSfuse semantics
5. GCSfuse benchmarks

Configuration Recommendations

Use a Hierarchical Namespace Bucket (HNS)

1. **Use a HNS bucket:** Use HNS buckets for 8x higher initial QPS, and for fast atomic directory renames which we require for checkpointing with GCSfuse.
   How?
   - Create a new bucket with HNS enabled.
   - This is a bucket level setting that is enabled during time of bucket creation. It is currently only available with new buckets. See simple instructions here

   When should this flag be used?
   - We recommend this bucket type be used for all AI/ML workloads, especially when used with GCSfuse to get a better file-like experience.

   Why?
   - HNS buckets provides 8x higher initial QPS: HNS supports 40,000 initial object read requests per second and 8,000 initial object write requests, vs an initial 5,000 object read requests per second and 1,000 initial object write requests in traditional GCS flat buckets. Note: GCS scales automatically based on usage from there. This is advantageous for AI/ML clusters because the workload tends to be highly correlated - all nodes in the AI/ML cluster submit requests at the same time, creating a very high request load.
   - HNS provides atomic directory renames, which are required for checkpointing with GCSfuse to ensure atomicity. ML frameworks use directory renames to finalize checkpoints, which is expected to be a fast atomic command, and is only supported in HNS buckets.

Directory Mount Instead of Entire Bucket

2. **Directory mount only (if applicable):** If accessing only a specific directory within a bucket, mount the specific directory using --only-dir flag if that is an option
   How?
   - Use - only-dir:<directory-path-relative-to-the-bucket-root> flag while mounting GCSfuse

   Example: mounting directory a/b inside of a bucket name MyBucket (mybucket/a/b)

```
None
volumeHandle: MyBucket
    - only-dir:a/b
```

When should this flag be used?

- If accessing only a particular subdirectory in a particular mount, mount only that particular subdirectory instead of mounting the entire bucket.

Why?

- Mounting only a particular subdirectory will make the list calls faster.
- LookUpInode calls/accessing a directory or bucket makes a list+stat call for each file/directory in the path. Using this flag will reduce those calls as the path will have fewer directories to traverse to resolve a filename.

Increase Metadata Cache Values

3. **Increase Metadata cache values:** Set metadata caches capacity to be able to grow very large and TTL to be infinite (e.g. >1 million object buckets). Infinite TTL should only be done for volumes that are either read-only or only “write-to-new”. Enabling the metadata cache to grow very large should only be done on nodes with large memory configurations (large buckets can cause metadata caches, which are stored in RAM, to grow very large). This will allow the entire bucket’s metadata to be cached in each node, eliminating the metadata accesses to GCS for any second access. The settings effectively will cache any metadata accessed, and the time to live (TTL) value is set to infinite (i.e. life of the job), thus any changes in GCS will not be seen (e.g. any over-writes of a file or deletions of a file).

If there is concern on the amount of memory consumed, please validate that the amount of memory consumed by the metadata cache is acceptable. It can grow to gigabytes, depending upon the number of files in the mounted buckets and how many mount points are being used. For example, each file's metadata takes about 1.5MB, so 1 Million files’ metadata will take approx. 1.5GB. More details here.

How?

- Set the following GKE Mount options
    - metadata-cache:ttl-secs:-1
    - metadata-cache:stat-cache-max-size-mb:-1
    - metadata-cache:type-cache-max-size-mb:-1 or a high value based on customer requirements

```
None
mountOptions:
    - metadata-cache:ttl-secs:-1
    - metadata-cache:stat-cache-max-size-mb:-1
    - metadata-cache:type-cache-max-size-mb:-1
```

- Full YAML provided at end of doc below with tuned options by type of mount.

What do these configurations mean?
- Using “-1” for metadata-cache:stat-cache-max-size-mb and metadata-cache:type-cache-max-size-mb lets the cache use as much memory as required. Use this if memory usage is not a constraint.
- Using “-1” or very high value for metadata-cache:ttl-secs bypasses the TTL expiration and serves the metadata from the cache whenever it's available. This might serve stale data if data is updated on GCS.

Why ?
- Avoids metadata GCS calls on repeat reads.
- Performance gains on repeat reads because the GCS metadata lookup is avoided.

-

Pre-populate the Metadata Cache

4. **Pre-populate the metadata cache:** Run “ls -R” before training, if the application doesn’t do this itself. If the application is accessing the entire directory structure from the current location down from where the “ls -R” is run, this can quickly populate the entire metadata cache and kernel list cache (if enabled) rather than have it be done piecemeal when each individual file is accessed. This largely eliminates the metadata calls to GCS, and makes a particularly large difference if the “--implicit-dirs” option is used.

   How?
   - Manually:
     - “ls -R” on the GCSfuse mount to recursively list all files in a more efficient batched manner.
   - Starting with GKE v1.32.1-gke.1357001, use the csi volume attribute flag ‘`gcsfuseMetadataPrefetchOnMount: "true"`‘ to enable metadata prefetch for the given volume which represents the bucket

- Previously, this was done by creating an initContainer, which is no longer needed. See here for the previous method.

```
None
apiVersion: v1
kind: PersistentVolume
metadata:
  name: training-bucket-pv
spec:
  ...
  mountOptions:
  ...
  csi:
    volumeHandle: <training-bucket> # unique bucket name
    volumeAttributes:
      ...
      gcsfuseMetadataPrefetchOnMount: "true"
```

- Full YAML provided at end of doc below with tuned options by type of mount.

When should this flag be used?

- With checkpoint, serving and training workloads: Prior to or while the workload is accessing the files. Typically, ML frameworks running training data workloads will issue this automatically, but we have seen custom training code where it does not.

Why?

- It can drastically reduce the number of list calls by front loading the cache (can be as high as 5000x).
- If the –implicit-dirs option is used, a GCS list probe is used to determine if a directory exists, unless the GCSfuse cache has been populated. By issuing the “ls -R”, this cache is fully populated. This means that a file open will not have any GCS metadata interactions at all - even if –implicit-dirs enabled mount points are used - substantially improving overall performance.

Enable the File Cache + Parallel Downloads

5. **Enable the file cache + parallel downloads:** The file cache feature accelerates repeat reads, serving them from local storage. The parallel download feature uses the file cache to download large files in parallel, using the file cache directory as its staging directly. This provides for extremely fast file cache initialization of the data for multi-gigabyte size

files and it can also provide for much faster read performance for applications doing sequential reads of large files by aggressively prefetching data (we are seeing up to 9x improvements in model loading for example, when file sizes are multi-gigabytes).

The file cache can be hosted on LSSD, RAM, or PD/Hyperdisk. Below are GPU and TPU specific guidance:
GPUs:
- Generally LSSD are excellent for training data and for checkpoint downloads.
- A RAM disk will provide the best performance for the loading of model weights, because they are small compared to the unused amount of RAM on the system and the extra performance by using RAM is worth it.
- PD/Hyperdisk can also be used as a cache

TPUs:
- TPUs do not support LSSD. The default location for the file cache is the boot volume. **IT IS NOT RECOMMENDED TO USE THE DEFAULT**. The boot volume performance will be very poor as it is intended for a boot workload, not for a caching workload.
- The RAM disk is the preferred option, because it has no incremental cost. However it is often constrained in size, and thus most useful for Serving model weights download. Checkpoint downloads may also work, depending on the size of the checkpoint and the available RAM.
- PD/Hyperdisk can also be used as a cache

In all cases, the data, or individual large file, must fit within the file cache directory's available capacity which can be controlled using the max-size-mb property.

How?
- File cache:
    - Provide a value to file-cache:max-size-mb: -1. A value of “-1” allows it to use the volume’s entire capacity, or you can give it a value in MiB
    - Set `metadata-cache:ttl-secs:-1 or a high value based on your requirements. "-1" bypasses TTL expiration and serves the file from the cache if it's available.
    - If doing many random/partial reads from the same file, consider enabling ‘- file-cache:cache-file-for-range-read: true’.
        - This asynchronously loads the entire file into the cache, so that subsequent reads of different offsets from the same file can served from the cache
- Parallel download - Enable for serving:
    - enable-parallel-downloads: true (default, false)
- GKE sample:

- By default file cache uses Local SSD if `ephemeral-storage-local-ssd` mode is enabled for the GKE node (See details here)
- If ephemeral storage on local SSD mode is not configured on the GKE nodepool (e.g. TPUv6 VMs), file cache will be the GKE node's boot disk, **which is not recommended**. Instead,a RAM disk can be used as the cache directory (commented out below), but must consider the amount of RAM available for filecaching vs what is needed by the pod.

GPU Example

```
None
mountOptions:
    - file-cache:max-size-mb:-1
    - file-cache:cache-file-for-range-read:true
    - file-cache:enable-parallel-downloads:true

#RAM disk file cache if preferred. Uncomment out to use#
#volumes:
#  - name: gke-gcsfuse-cache
#    emptyDir:
#      medium: Memory
```

TPU Example

```
None
mountOptions:
    - file-cache:max-size-mb:-1
    - file-cache:cache-file-for-range-read:true
    - file-cache:enable-parallel-downloads:true

volumes:
  - name: gke-gcsfuse-cache
    emptyDir:
      medium: Memory
```

- Full YAML provided at end of doc below, with tuned options for mount points defined for training data, checkpoints, and serving model weights.

When should these features be used?

File Cache Feature

- For Serving model weights and checkpoint reads: Typically model weights are read once, so enable file cache primarily to enable parallel downloads (see below).

- For training data reads: Enable the cache if the data you are accessing is read multiple times (ex: multi-epoch training). For training that typically means the dataset set is less than the file cache size, because on each epoch a specific node typically reads different data. Thus simply doing a large parallel download at the beginning of the training run into the file cache makes all epochs fast. If the dataset is larger than the file cache, typically DO NOT enable the file cache. Enabling cache-file-for-range-read is also recommended.

Parallel Download Feature:

- For serving model weights: Always enable the parallel download feature. Typically best performance is with a RAM disk.
- For checkpoint reads: Always enable the parallel download feature. Typically this is best done with LSSD if available (i.e. on GPUs). On TPUs the best choice is RAM.

Combining the Features:

- For training: Typically enable the file cache without the parallel download feature, although some workloads will see performance improvements with it enabled.
- For training: If the entire dataset fits within the cache capacity. If it does not, then you may have high cache thrashing, which will degrade performance, and it should not be used.
    - Each A3 machine (H100) has 6TiB of Local SSD included, therefore the entire dataset must be 6TiB or less (Note: Each GCSfuse instance has its own cache which is not shared).

Why?

- Improved performance for:
    - Training - Repeat reads: The Cloud Storage FUSE file cache is a client-based read cache that lets repeat file reads to be served from a faster cache media, significantly improving small and random I/O while reducing operations back to GCS.
    - Training - Random reads into large files.
    - Serving - Large file reads: Even first reads of large files are accelerated through parallel multi-threaded downloading

Set negative stat cache

6. **Set negative stat cache:** Negative stat cache entries are cached by default with a ttl of 5 seconds. Negative cache entries can become stale quickly in workloads where files are frequently created or deleted by one node, and accessed by another. We have seen some frameworks use this approach for creating index files for training data workloads and coordinating checkpoint writes. It may also occur if publishing new Model weights for serving. Therefore, we recommend disabling this all together for training data,

checkpointing, and serving model weights. This feature requires GCSfuse >=GCSfuse v2.8 which is 1.32.1-gke.1200000

How?
- Set metadata-cache:negative-ttl-secs to 0
- GKE sample:

```
None
mountOptions:
    - metadata-cache:negative-ttl-secs:0
```

- Full YAML provided at end of doc below

When should this flag be used?
- For training data, checkpointing and serving model weights with GCSfuse

Why?
- Negative cache entries can become stale quickly in workloads where files are frequently created or deleted by one node, and accessed by another, such as how certain frameworks implement distributed checkpointing or create index files for training data.

Checkpointing

Note: Checkpointing with GCSfuse should always be done using an HNS bucket. We do not currently recommend using GCSfuse for checkpointing at scale **without HNS**. This is because ML frameworks use directory renames to finalize checkpoints, which is expected to be a fast atomic command, and is only supported in HNS buckets.

If you can accept the risk of directory renames not being atomic and taking longer, **set rename-dir-limit if checkpointing in non HNS enabled buckets**.

7. **Enable streaming writes:** GCSfuse's default write path stages files in their entirety in a temporary directory on a local volume, and then are written out to GCS upon closing or fsyncing the file. This requires that there is enough free space available on the local volume to handle staged content when writing large files. Streaming writes is a new write path that uploads data directly to GCS as it's written. This improves checkpoint write performance and eliminates disk space usage.

   Streaming writes are designed for sequential writes to a new, single file only. Modifying existing files, or doing out-of-order writes (whether from the same file handle or concurrent writes from multiple file handles) will cause GCSFuse to automatically revert to the behavior of staging writes to a temporary directory on local volume. See additional semantics changes here.

Each file opened for streaming writes will consume approximately 64MB of RAM during the upload process for buffering, which is released when the file handle is closed.

This feature requires GCSfuse >= v2.9.1 available on GKE version 1.32.1-gke.1729000 or later.

How?
- Set enable-streaming-writes to true
- GKE sample:

```None
mountOptions:
    - write:enable-streaming-writes:true
```

- Full YAML provided at end of doc below

When should this flag be used?
- Checkpointing with GCSfuse

8. **Set rename-dir-limit if checkpointing using traditional GCS buckets (flat namespace):** Although not recommended, if using traditional GCS buckets (flat namespace) instead of HNS buckets for checkpointing through GCSfuse at smaller scale, set rename-dir-limit to a high value or checkpointing will fail. We've found that ML frameworks do directory renames at the end of checkpointing when working with a filesystem. Because GCS is a flat namespace, and objects in GCS are by definition immutable, a directory rename requires renaming of all the individual files underneath the directory, and deletion of the old files. This is currently disabled in GCSfuse, so needs to be enabled by setting a value to rename-dir-limit.

   How?
   - Set rename-dir-limit to a high value. We tested with 200000, you may need to adjust this based on your directory and subdirectory structure.
   - GKE sample:

```None
mountOptions:
    - rename-dir-limit=200000 #for checkpointing
```

- Full YAML provided at end of doc below

When should this flag be used?

- **Checkpointing with GCSfuse should always be done using an HNS bucket.** We do not currently recommend using GCSfuse for checkpointing at scale **without HNS**.
- If you can accept the risk of directory renames not being atomic and taking longer, **set rename-dir-limit if checkpointing when checkpointing in non HNS enabled buckets**.

Why?

- ML frameworks do directory renames at the end of checkpointing when working with a filesystem.
- Because GCS is a flat namespace, and objects in GCS are by definition immutable, a directory rename requires the reupload of all files.
- This is currently disabled in GCSfuse, so needs to be enabled by setting a value to rename-dir-limit.

Enable Kernel List Caching

9. **Enable Kernel List caching.** The kernel list cache is a cache for directory and file listing responses designed to improve list operation speeds (e.g. “ls” or “ls -R”). The list cache is kept in memory in the kernel page cache, which is controlled by the kernel based on memory availability, as opposed to the stat and type caches, which are kept in your machine's memory and controlled by Cloud Storage FUSE.

How?

- Enable list caching using the --kernel-list-cache-ttl-secs
    - A positive value represents the TTL in seconds to keep the directory list response in the kernel's page cache.
    - A value of -1 to bypass entry expiration and return the list response from the cache when it's available.
    - It is recommended this be done with a RO mount only to avoid consistency issues
    - GKE sample

```
None
mountOptions:
    - file-system:kernel-list-cache-ttl-secs:-1
```

- Full YAML provided at end of doc below

When should this flag be used?

- List caching is especially useful for workloads that repeat full directory listings of large directories as part of execution or interactive use cases, such as AI/ML libraries, configurations, serving, and in some cases, training runs.
- Note that enabling this flag should be done with caution. If the filesystem is truly read-only, there are no risks. However if the filesystem can be written, and the TTL is set to -1 (as suggested above), the local application will never see the updated file if the directory is cached. Thus this will break checkpoints and some training frameworks that write index files as part of training, and should not be used for those frameworks (e.g. NeMo). Example:
    - Client 1 lists directoryA, which causes directoryA to be resident in the kernel list cache.
    - Client 1 continuously checks for a file B in directoryA, which is simply checking the kernel list cache entry - it never goes over the network.
    - Client 2 creates file B under directoryA in GCS.
    - Client 1will never see that a new file is in the directory because the list of files is always satisfied from the local kernel list cache.
    - Client 1 times out. The program is broken.
-

Why?

- If configuration files, or large million object buckets need to be traversed with excellent performance, this enables completely local performance. “Ls” performance is essentially identical to if all the files were on local SSD (after the cache is populated). Use this capability in conjunction with the Pre-population of the Metadata cache for best performance.
- LookUpInode calls/accessing a directory or bucket makes a list+stat call for each file/directory in the path. Using this flag will eliminate those calls for any cached entries.

Kernel Read Ahead

10. **Increase Kernel Read Ahead:** For workloads that primarily involve sequential large reads, such as model weight serving and checkpoint-restore, increasing the read-ahead size can significantly enhance performance. To increase Cloud Storage FUSE read-ahead, specify the read_ahead_kb flag in mountOptions and set it to 1MB (e.g. read_ahead_kb=1024).

    This feature requires GKE version 1.32.1-gke.1200000.

When should this flag be used

- For workloads that primarily involve large sequential reads, such as serving and checkpoint-restore. Some training data workloads also exhibit this behavior.

How?

- This is a Linux kernel level setting
- If using GCSfuse in a GCE deployment, follow these instructions after your GCSfuse bucket has been mounted
- On GKE deployments, the GCSfuse CSI driver has a managed API to do this, which is simply passed as part of the mountOptions. GKE sample:

```
None
mountOptions:
    - read_ahead_kb=1024
```

- Full YAML provided at end of doc below

Debug logs

11. **Debug logs:** By default, info level logging is enabled. For troubleshooting, you may want to use trace level to understand behaviors and we may request you enable trace level logging. Enabling trace level logging can affect performance and increase logging costs.

    How?
    GKE sample

```
None
mountOptions:
    - log-severity=trace
```

GPU vs TPU considerations

12. GPUs and TPUs differ, amongst many other things, in terms of the number of available resources (CPU, Memory, Local Storage) available within its host node configuration. For example:
    - A3 Mega - 1.8 TiB memory, with 6 TiB LSSD
    - TPU v5e - 188 GiB memory, with no LSSD
    - TPU v5p - 448 GiB memory, with no LSSD
    - TPU v6 (Trillium) - 1.5TiB memory, with no LSSD

In general all guidance is the same for TPUs and GPUs, except for the File Cache and associated Parallel Downloads. Please see that section for specific advice.

Security Token Service

13. **Security Token Service:** The current GCSfuse CSI driver has access checks (to ensure pod recoverability due to user misconfiguration of workload identity bindings between GCS bucket and k8s Service Account) which can hit default Security Token Service API quotas at scale. This can be disabled by setting skipCSIBucketAccessCheck volume attribute of Persistent Volume CSI driver. Please ensure the k8s Service Account has right access to the target GCS bucket to avoid mount failures for the pod.

```
None
volumeAttributes:
    - skipCSIBucketAccessCheck: "true"
```

For GKE based deployments, the STS quota needs to be increased beyond the default value of 6000 if a GKE cluster consists of 6k+ nodes, which can result in 429 errors if not increased in large scale deployments. The STS quota has to be increased through the quotas page. The guidance is to keep quota = number of mounts, e.g. if there are 10k mounts in the cluster then quota should be increased to 10k

JAX Persistent Compilation (JIT) Cache Tuning

JAX supports an optional persistent compilation cache (JIT cache) that stores compiled function artifacts. Utilizing this cache can significantly speed up subsequent script executions by avoiding redundant compilation steps as documented here. This is achieved by setting the environment variable:

```
None
export JAX_COMPILATION_CACHE_MAX_SIZE=-1
```

Ensure Recent JAX: Use JAX version 0.5.1 or newer for the latest cache features and optimizations.
Maximize Cache Capacity: To prevent performance degradation due to cache eviction, consider setting an unlimited cache size, particularly if overriding default settings. Bash

Then, mount the JAX JIT cache using the same config we recommend for checkpointing.

Sample YAMLs

This section provides a set of sample YAMLs for the GCSfuse CSI driver for GKE tuned to specific AI/ML workloads. The intent is to use a different mount point for each workload (Training, Serving, Checkpoints/JIT Cache). Customers may be able to collapse this to fewer mount points, depending upon their individual requirements.

The following parameters are the same for all volume types.

- Meta data cache enabled and:
  - STAT cache maximum size: unlimited
  - Type cache maximum size: unlimited
  - TTL: Infinite
  - Negative entry TTL: 0
- GKE GCSfuse Sidecar parameters:
  - CPU limit: unbounded (default)
  - Memory limit: unbounded (default)
  - Ephemeral storage limit (size of file cache): unbounded (default)
- GKE parameters:
  - Metadata Prefetch on Mount: True
  - Skip CSI Bucket Access Check: True

Mounts for serving model parameters, Checkpoints, and JIT cache all enable the following parameters. Training may also see value in enabling them - see the File Cache, Kernel Read Ahead, and comments below. In general, enable the File Cache for training if the dataset is smaller than the cache size. If it is larger than the cache size, **DO NOT** enable the file cache. Because training data usually is random range reads, setting large read ahead values is not typically will see no benefit.

- File Cache parameters
  - Parallel downloads enabled
  - File cache maximum size: unbounded
  - Range reads enabled
- Read ahead set to 1024 KB

Parameters not used on sample YAMLs:

- List cache - because the coherency model intended is to immediately see “writes to new objects”, and enabling an infinite lifetime in the list cache would prevent that, we do not enable the list cache.
- Logging level Trace - because there is a performance penalty when trace log level is enabled, this is not done by default.

Training

See common parameters set above for all volume types. Parameters specific to the Training volume type are
File cache enabled - Only for GPUs, or with TPUs if using a RAM disk with enough available memory

The training volume spec does not have:
- Parallel download feature
- Kernel read ahead

First, deploy the PVC/PV first (this is required because the GKE pod webhook inspects the PV volume attributes for additional optimizations like injection of a metadata prefetch GKE container)

Training PV Config for GPUs	training-pv.yaml
Training PV Config for TPUs	training-pv.yaml

Then, deploy the pod spec that accesses the PVC

Training Pod Config for GPUs	training-pod.yaml
Training Pod Config for TPUs	training-pod.yaml

Serving Volume Mount Point

Serving spec has the common parameters set, the file cache parameters set and read ahead set.

For best performance, configure the RAM disk to be used if enough available memory is available, regardless of whether a GPU or TPU is used. If not enough RAM is available, and on a GPU platform, use local SSD.

First, deploy the PVC/PV first (this is required because the GKE pod webhook inspect the PV volume attributes for additional optimizations like injection of a metadata prefetch GKE container)

Serving PV Config for GPUs	serving-pv.yaml
Serving PV Config for TPUs	serving-pv.yaml

Then, deploy the pod spec that accesses the PVC

Serving Pod Config for GPUs	serving-pod.yaml
Serving Pod Config for TPUs	serving-pod.yaml

Checkpointing & JIT Cache Volume Mount Point

Checkpointing & JIT Cache spec has the common parameters set. The file cache is enabled and read ahead set to enable fast parallel downloads (checkpoint loads or JIT cache binary downloads). In addition:

- Requires HNS for checkpointing - this YAML was built around this assumption
- Requires write streaming

Checkpointing & JIT Cache spec does not have:

- Kernel list cache

First, deploy the PVC/PV first (this is required because the GKE pod webhook inspect the PV volume attributes for additional optimizations like injection of a metadata prefetch GKE container)

Checkpointing PV Config for GPUs	checkpointing-pv.yaml
Checkpointing PV Config for TPUs	checkpointing-pv.yaml

Then, deploy the pod spec that accesses the PVC

Checkpointing Pod Config for GPUs	checkpointing-pod.yaml
Checkpointing Pod Config for TPUs	checkpointing-pod.yaml

Appendix

Links to All Sample YAMLs

GPUs	README training-pv.yaml training-pod.yaml serving-pv.yaml serving-pod.yaml checkpointing-pv.yaml checkpointing-pod.yaml
TPUs	README training-pv.yaml training-pod.yaml serving-pv.yaml serving-pod.yaml checkpointing-pv.yaml checkpointing-pod.yaml

Deployment Instructions

Note: The sample files are for GCSfuse GKE CSI driver running on GKE clusters of GKE version 1.32.2-gke.1297001 or greater.

To utilize these sample configurations, follow the specified deployment order. For instance, to set up the serving workload:

1. Deploy the PersistentVolume (PV) and PersistentVolumeClaim (PVC): Apply the `*-pv.yaml` file first. This step is crucial as the GKE pod admission webhook inspects the PV's volume attributes to apply potential optimizations, such as the injection of sidecar containers, before the pod is scheduled.

None
```
kubectl apply -f serving-pv.yaml
```

2. Deploy the Pod: After the PV and PVC are successfully created, deploy the pod specification that references the PVC.

None
```
kubectl apply -f serving-pod.yaml
```

Prerequisites and Notes

- Service Account: Ensure the specified Kubernetes Service Account (e.g., `<YOUR_K8S_SA>` in the pod YAML) exists and possesses the necessary permissions to access the target Google Cloud Storage bucket *before* deploying the pod.
- Placeholders: Replace all placeholder values (e.g., `<customer-namespace>`, `<checkpoint-bucket>`, `<YOUR_K8S_SA>`) within the YAML files with your specific environment details before application.

Summary of the Goals in Tuning GCSfuse Configurations

Operating GCSfuse in high scale clusters means that the added “chattiness” that occurs due to file API metadata operations must be mitigated or GCS QPS will become dominated by metadata operations. GCSfuse has multiple optimizations to overcome this “chattiness” which can become complex to manage individually. The simplest way to deploy it is to view the volume as a read-only volume, and configure the optimizations to download the various metadata and authorization state as fast as possible into the per node caches such that essentially no metadata or authorization operations at all happen to GCS during the job. However if an application is run on the volume accidentally assuming different coherency semantics, application inconsistencies can occur. To prevent this, it is recommended that when all optimizations are turned on, the mount point is configured as **read-only**.

Because most of the GCSfuse configuration options are enabled for the entire mount point, it is recommended that separate mount points be used for model weight download for Inference, training data, serving data (if any), and checkpoints (if used) so that you can set parameters differently for each mount point.

The configurations in this document are intended to do several things (in purple are specific recommendations for A3/A3-mega class machines):

- **Substantially reduce GCS List, Metadata, and Authorization operation “chattiness”**. The file API can introduce a lot more GCS metadata operations than might be required if simply accessing GCS directly, due to the nature of how the File API traverses directories at fopen()[1], constrains when errors can be returned at fopen()[2], and how “ls -R” maps to GCS list calls (single sub-directory listing)[3]. In general, the goal is to enable local caching of filesystem directory and file metadata for the entire mounted volume’s metadata (e.g. the GCS bucket) - making metadata and list operations extremely fast and predictable (i.e. no traffic to GCS).

  To more easily achieve this goal, the recommendation is to enable metadata caching to grow to the number of objects in the bucket, even if the bucket is very large (>1 million object buckets for some AI/ML customers can be common), keep the cached entries in cache for the duration of the training or serving/inference job (i.e. TTL = infinite), and enable list/directory caching. Beyond this, there are two basic approaches to mitigate chattiness based on how to fill the metadata caches:

[1] Traversing the directory hierarchy on fopen() means the path and access is validated. If implicit directories is used, worst case a GCS “List=1” call and GCS GetObjectMetaData are made for each level in the hierarchy to validate the path. If GCSfuse’s caches entries for that level are valid, then neither is requested, and no GCS metadata request occurs.

[2] GCS GetObjectMetadata is sent to validate access at fopen(). If GCSfuse’s cache entry is valid, no request is sent.

[3] If the kernel directory cache for the GCSfuse mount point is enabled and valid, then all “ls” calls are satisfied from the kernel cache, and no calls occur to GCS.

  - **Download all the object and directory metadata immediately after mount() completes, but allow application workloads to proceed in parallel** (Prepopulate the metadata cache) **(Recommended).** Once the download completes, this means that implicit-dirs) can be used with no performance impact[4]. In addition, all needed file metadata is downloaded and cached, so opens() are efficient[5] and "ls -R" will be high performance. The downside is the additional GCS load of the listing of the bucket immediately after mount for a short period of time[6].
  - **Leverage Application behavior to initialize the cache.** Some applications behave in such a way that they optimally initialize the caches. We have observed, for example, that some AI/ML training applications perform a "ls -R" as part of planning training data, thus a second "ls -R" is not needed.
  - **Block workloads from starting until after the metadata download completes.** This will ensure a consistent experience for all applications, but also means Node resources are stranded while initialization completes. Thus this is not recommended.
  - **Do not download object and directory metadata.** This approach avoids the up-front cost of downloading the object and directory metadata, but can keep GCS accesses in a "chatty" configuration for an unbounded amount of time. Thus it is recommended in this configuration that explicit directories be used (i.e. –implicit-dirs not be used - see below).[7] See the directions below for a script to create the explicit directories. Anytime new objects are added to the bucket that contain new directories the tool must be run again to maintain coherency.

- **Optimize for high performance workloads (Recommended)**. For large machine instances, it is important to scale to use the full resources available for best performance. The defaults in GCSfuse are set to work well in all machine sizes, and thus must be overridden to get full performance. Specific overrides recommended are the maximum number of TCP connections, and also increase the GKE side-car memory and CPU limits.

- **Enable File Cache**. GCSfuse supports a local SSD data cache, which can be used, for example, with the 6TB local SSD capacity that comes bundled with A3 machine instances. There are many ways that AI/ML applications access data, but three primary modes are **1) traditional cache** to optimize second access of data, **2) parallel prefetch** of an object (download of a large object to enable fast local random range reads, for example) and **3) uncachable dataset**, where each node (GPU) is randomly accessing

[4] "List=1" probes to GCS for directory and file entries will not be generated because the GCSfuse "type cache" (i.e. whether it is a directory or an object) will have been initialized
[5] I.e. not subject to GCS performance overhead of GCS GetObjectMetaData operations in-line with the fopen() if the entry is not in cache.
[6] Testing has shown this can be between 100 msec (10K objects) to ~2 minutes (millions of objects), but can vary substantially depending on how deeply nested the hierarchy of the namespace.
[7] By using explicit directory objects, "List=1" probes to discover directories will not occur when traversing the directory hierarchy (for example, for an open()).

data across the entire dataset, and the dataset is much larger than the amount of SSD on the local node. For 1) and 2). the cache should be enabled. For 3) it should not.

- Some examples of access patterns:
    - For 1) an example on A3 might be a dataset that is less than 6 TB on a 64 GPU cluster. It likely will be most performant to simply do a massively parallel download of the entire dataset into the local GCSfuse File Cache (GCSfuse will do this transparently) for the multi-epoch training. Similarly for Serving to hundreds or even thousands of nodes.
    - For 2) an example on A3 might be a 100 TB multi-modal dataset made up of many files, where each GPU is training on tokens from one or more specific files for a specific epoch. It randomly accesses tokens within the file(s) using a normal batching algorithm. GCSfuse transparently translates that into a massively parallel download of all of the file(s) accessed into local SSD, for very fast low-latency token batching for the rest of the epoch. On the next epoch, when the set of files to be trained are randomly selected to be different files, GCSfuse again does a massively parallel download of the specific file(s) accessed to give low-latency local reads (batch of tokens) for training.
    - For 3) an example might be a 100 TB multi-modal dataset made up of many files where each GPU is training on a randomly selected set of tokens across the entire dataset - so effectively it is doing random range reads across all of the files in the dataset, and is uncachable in the 6 TB of cache available on the A3 instance. It is best to turn off the File Cache for this workload.
- Below are some example workloads and recommendations on when the File Cache and Parallel Downloads should be enabled. We recommend they be enabled together. For serving data, we recommend always enabling both the File Cache and Parallel Prefetch. For training data, it can depend on the specific workload. **Rather than trying to characterize your workload for training data, it might be easiest to simply run the workload with both File Cache Enabled and Parallel Prefetch Enabled and then again with both off to see if there are benefits.**
    - **Serving (Recommended)**
        - Model download during instance boot. Recommended because it will typically be used as a large prefetch buffer (i.e. mode 2 - enable both File Cache and Parallel Prefetch).
    - **Training Data, multi-epoch training**
        - If training application uses Mode 1. **(Recommended)**
        - If training application uses Mode 2. **(Recommended)**
        - If training application uses Mode 3. **(NOT Recommended)**
    - **Training Data, single-epoch training**
        - If training application uses Mode 1. **(NOT Recommended)**
        - If training application uses Mode 2. **(Recommended).**
        - If training application uses Mode 3. **(NOT Recommended)**

        - If training application accesses small files. (**NOT Recommended**) Typically training will only access a small file one time in an epoch, and because the file is small there is no value in a prefetch.
    - **Training, checkpoints**
        - While checkpoint reads will be extremely fast with File Cache parallel prefetch, checkpoint writes through GCSfuse have multiple issues, and thus it is **NOT Recommended** to use GCSfuse for checkpointing at this time. We are actively working on a solution.

- **Other minor improvements (Recommended)**. There are also minor improvements in robustness recommended.
    - Reduce Security Token Transfer load from GKE.
    - Recommendation to scope the mount() to just the dataset being accessed. This will make GCS list operations more efficient.

Workload Identity Federation Setup

If the workloads do not have any access permission issues, please skip this section.

- Follow the documentation Configure access to Cloud Storage buckets using GKE Workload Identity Federation for GKE to configure the bucket access. Using GKE Workload Identity Federation, you don't need to create a GCP service account.
- Please make sure that the bucket for read-write workloads uses Uniform bucket-level access with Workload Identity Federation.

Outdated recommendations

The following recommendations are already automatically implemented in code, so are not needed to be manually applied with newer versions.

Security Token Transfer

Security Token Transfer: This an internal implementation detail, but the current GCSfuse CSI driver has redundant access checks which can hit default Security Token Transfer quotas at scale. For now, this can be disabled via the CSI driver, while we implement a permanent solution that automatically handles this. The skipCSIBucketAccessCheck` will ensure the CSI driver does not perform any additional CSI bucket access checks (which results in additional token exchange requests to the Security Token Service server)

```
None
volumeAttributes:
    - skipCSIBucketAccessCheck: "true"
```

Full YAML provided at the end of the doc below.

Increase TCP Connections

Update: As of GCSfuse version 2.1, GCSfuse automatically defaults to max-conns-per-host=0. Therefore, versions 2.1 and greater automatically default to this and do not require users to pass this

Increase TCP connections: Increase the maximum TCP connections (http1 connections) that GCSfuse can open with the GCS server, setting 'max-conns-per-host' to 0. If there are not enough TCP connections, under certain conditions this can lead to applications deadlocking waiting for resources.

How?

- Mount GCSfuse with "--max-conns-per-host=$VALUE"
- $VALUE is by default 100.
- We recommend setting it to "max-conns-per-host=**0"** which does not limit max TCP connections. Alternatively, if you know the max number of files the workload concurrently opens, you can set it to this number, plus a buffer (because you don't want to hit the deadlock condition).
- On GKE, this is done through mount options

```
None
mountOptions:
    - max-conns-per-host=0
```

- Full YAML provided at end of doc below
- Note: This is in the process of being rolled out as by default in the GKE CSI driver

Why ?

- Shortage of TCP connections on concurrent file access can degrade performance or cause stuck issues.

Pre-populate the Metadata Cache

1. **Pre-populate the metadata cache:** Run “ls -r” before training, if the application doesn’t do this itself. If the application is accessing the entire directory structure from the current location down from where the “ls -r” is run, this can quickly populate the entire metadata cache and list cache (if enabled) rather than have it be done piecemeal when each individual file is accessed. This substantially reduces the metadata calls to GCS, particularly if the “--implicit-dirs” option is used. This can either be done manually, or automatically via an initContainer

   How?
   - Manually:
     - “ls -r” on the GCSfuse mount to recursively list all files in a more efficient batched manner and pre-populates the metadata cache to make the first run faster
   - Automated using a sidecar by following:
     - Full details here

```
None
initContainers:
# Metadata Prefetch native sidecar.
- name: metadata-prefetch-container
  image: ubuntu:22.04
  restartPolicy: Always
  command:
  - "/bin/bash"
  - "-c"
  - |
    echo "Starting ls on the bucket..."
    # Redirect output to /dev/null to prevent storage of output.
    ls -R /data > /dev/null && \
    echo "Metadata prefetch complete. Going to sleep..." && \
    tail -f /dev/null
  resources:
    requests:
      cpu: 250m
      memory: 256Mi
  securityContext:
    allowPrivilegeEscalation: false
    capabilities:
      drop:
      - ALL
    readOnlyRootFilesystem: true
    runAsGroup: 65534
    runAsNonRoot: true
    runAsUser: 65534
    seccompProfile:
      type: RuntimeDefault
  volumeMounts:
  - mountPath: /data
    # Point volume mount to the desired volume.
    name: gcs-fuse-csi-bucket1
```

- Full YAML provided at end of doc below

When should this flag be used?

- With serving and training workloads: Prior to training runs beginning to access files. Typically, ML frameworks will issue this automatically, but we have seen custom training code where it does not.

Why?

- It can drastically reduce the number of list calls by front loading the cache (can be as high as 5000x).
- If the –implicit-dirs option is used, a GCS list probe is used to determine if a directory exists, unless the GCSfuse cache has been populated. By issuing the "ls -R", this cache is fully populated, thus the performance of –implicit-dirs enabled mount points will be the same as mount points with explicit mount points (see above discussion).