# MIOpen Documentation

***Release 3.4.0***

**Advanced Micro Devices, Inc.**

**May 21, 2025**

# INSTALL

MIOpen is the AMD open-source, deep-learning primitives library for GPUs. It implements fusion to optimize for memory bandwidth and GPU launch overheads, providing an auto-tuning infrastructure to overcome the large design space of problem configurations. It also implements different algorithms to optimize convolutions for different filter and input sizes.

MIOpen is one of the first libraries to publicly support the `bfloat16` data type for convolutions, which allows for efficient training at lower precision without loss of accuracy.

The MIOpen public repository is located at https://github.com/ROCm/MIOpen.

Install

- *MIOpen prerequisites*
- *Install MIOpen*
- *Build MIOpen from source*
- *Build MIOpen for embedded systems*
- *Build MIOpen using Docker*

Conceptual

- *Find database*
- *Kernel cache*
- *Performance database*
- *Manual tuning*
- *MI200 alternate implementation*
- *Porting to MIOpen*

How to

- *Use the fusion API*
- *Log and debug*
- *Use the find APIs and immediate mode*

Reference

- *API library*
  - *Modules*
  - *Datatypes*

For information on contributing to the MIOpen code base, see Contributing to ROCm.

For licensing information for all ROCm components, see ROCm licensing.

CHAPTER

ONE

# MIOPEN PREREQUISITES

To install MIOpen, you must first install these prerequisites. These prerequisites apply to all types of MIOpen installations.

- A ROCm-enabled platform
- A base software stack that includes either:
  - HIP (HIP and HCC libraries and header files)
  - OpenCL (OpenCL libraries and header files) (Using MIOpen with OpenCL is now deprecated.)
- ROCm CMake: CMake modules for common build tasks needed for the ROCm software stack
- Half: An IEEE 754-based, half-precision floating-point library
- Boost: Version 1.79 is recommended, because older versions might need patches to work on newer systems
  - MIOpen uses the `boost-system` and `boost-filesystem` packages to enable persistent *kernel cache*
- SQLite3: A read-write performance database
- lbzip2: A multi-threaded compression and decompression utility
- rocBLAS: AMD's library for Basic Linear Algebra Subprograms (BLAS) on the ROCm platform.
  - Minimum version for pre-ROCm 3.5 master-rocm-2.10
  - Minimum version for post-ROCm 3.5 master-rocm-3.5
- Multi-Level Intermediate Representation (MLIR), with an MIOpen dialect to support and complement kernel development
- Composable Kernel: A C++ templated device library for GEMM-like and reduction-like operators.

CHAPTER

TWO

# INSTALL MIOPEN

To install MIOpen from a package, choose either a pre-built package for your Linux distribution or choose a pre-compiled kernels package. For a list of MIOpen prerequisites, see *MIOpen prerequisites*. To build MIOpen from source, see *build MIOpen from source*.

## 2.1 Installing using a pre-built package

To install MIOpen on Ubuntu, use `apt-get install miopen-hip`.

If you are using OpenCL, use `apt-get install miopen-opencl`. (This is not recommended because OpenCL is deprecated.)

**Note**

You can't install both backends on the same system simultaneously. To switch to a different backend, completely uninstall the existing backend prior to installing the new backend.

## 2.2 Installing using a kernels package

MIOpen provides an optional pre-compiled kernels package to reduce startup latency. These precompiled kernels consist of a select set of popular input configurations. This collection of kernels will continue to expand to include additional coverage.

**Note**

All compiled kernels are locally cached in the `$HOME/.cache/miopen/` folder, so these pre-compiled kernels only reduce the startup latency for the first run of a neural network. Pre-compiled kernels don't reduce the startup time on subsequent runs.

To install the kernels package for your GPU architecture, use the following command:

```
apt-get install miopen-hip-<arch>kdb
```

Where <arch> is the GPU architecture, for example, `gfx900`, `gfx906`, or `gfx1030`.

**Note**

If you don't install these packages, it doesn't impact the functioning of MIOpen. This is because MIOpen compiles them on the target machine after you run the kernel. However, the compilation step might significantly increase the startup time for certain operations.

The `utils/install_precompiled_kernels.sh` script provided as part of MIOpen automates the preceding process. It queries the user machine for the GPU architecture and then installs the appropriate package. To run it, use the following command:

```
./utils/install_precompiled_kernels.sh
```

The preceding script depends on the `rocminfo` package to query the GPU architecture.

## 2.3 Installing dependencies

To install the MIOpen dependencies, use the `install_deps.cmake` command:

```
cmake -P install_deps.cmake
```

By default, this installs the dependencies in `/usr/local`, but you can specify another location using the `--prefix` argument:

```
cmake -P install_deps.cmake --prefix <miopen-dependency-path>
```

The following example demonstrates how to use `cmake` with a specific installation directory:

```
cmake -P install_deps.cmake --minimum --prefix /root/MIOpen/install_dir
```

You can specify this directory during the configuration phase using `CMAKE_PREFIX_PATH`.

MIOpen's HIP backend uses rocBLAS by default. You can install the rocBLAS minimum release using `apt-get install rocblas`. To disable rocBLAS, set the configuration flag `-DMIOPEN_USE_ROCBLAS=Off`. rocBLAS is **not** available with OpenCL.

MIOpen's HIP backend can use hipBLASLt. To install the minimum release of hipBLASLt, use `apt-get install hipblaslt`. In addition to installing hipBLASLt, you must also install hipBLAS. To install the hipBLAS minimum release, use `apt-get install hipblas`. To disable hipBLASLt, set the configuration flag `-DMIOPEN_USE_HIPBLASLT=Off`. hipBLASLt is **not** available with OpenCL.

CHAPTER

THREE

# BUILD MIOPEN FROM SOURCE

This topic discusses how to build MIOpen from source and configure the resulting application. It also explains how to build the library and driver and run the tests. For a list of MIOpen prerequisites, see *MIOpen prerequisites*. To install MIOpen from a package, see *install MIOpen*.

## 3.1 Building MIOpen

You can build MIOpen form source using either a HIP backend or an OpenCL backend.

### 3.1.1 HIP backend

To build MIOpen using the HIP backend (in ROCm 3.5 and later), follow these steps:

1. Create the build directory:

```
mkdir build; cd build;
```

2. Configure CMake. Set the backend using the `-DMIOPEN_BACKEND` CMake variable and set the C++ compiler to `clang++`. The command to build MIOpen with a HIP backend follows this format:

```
export CXX=<location-of-clang++-compiler>
cmake -DMIOPEN_BACKEND=HIP -DCMAKE_PREFIX_PATH="<hip-installed-path>;<rocm-
↪installed-path>;<miopen-dependency-path>" ..
```

   An example of a CMake build is:

```
export CXX=/opt/rocm/llvm/bin/clang++ && \
cmake -DMIOPEN_BACKEND=HIP -DCMAKE_PREFIX_PATH="/opt/rocm/;/opt/rocm/hip;/root/
↪MIOpen/install_dir" ..
```

> **Note**
>
> When specifying the path for the `CMAKE_PREFIX_PATH` variable, **do not** use the tilde (~) symbol to represent the home directory.

### 3.1.2 OpenCL backend

To build MIOpen using an OpenCL backend, run the following command:

```
cmake -DMIOPEN_BACKEND=OpenCL ..
```

> **Note**
>
> OpenCL is deprecated and the HIP backend is recommended instead. To install MIOpen using HIP, follow the instructions in the preceding section.

The preceding code assumes OpenCL is installed in one of the standard locations. If not, then manually set these CMake variables:

```
cmake -DMIOPEN_BACKEND=OpenCL -DMIOPEN_HIP_COMPILER=<hip-compiler-path> -DOPENCL_
↪LIBRARIES=<opencl-library-path> -DOPENCL_INCLUDE_DIRS=<opencl-headers-path> ..
```

Here's an example showing how to configure the dependency path for an environment (applies to ROCm version 3.5 and later):

```
cmake -DMIOPEN_BACKEND=OpenCL -DMIOPEN_HIP_COMPILER=/opt/rocm/llvm/bin/clang++ -DCMAKE_
↪PREFIX_PATH="/opt/rocm/;/opt/rocm/hip;/root/MIOpen/install_dir" ..
```

### 3.1.3 Setting the install location

By default, the install location is set to `/opt/rocm`. To change this, use `CMAKE_INSTALL_PREFIX`:

```
cmake -DMIOPEN_BACKEND=HIP -DCMAKE_INSTALL_PREFIX=<miopen-installed-path> ..
```

### 3.1.4 System performance database and user database

The default path to the system performance database (System PerfDb) is `miopen/share/miopen/db/` within the install location. The default path to the user performance database (User PerfDb) is `~/.config/miopen/`. Setting `BUILD_DEV` for development purposes changes the default path for both database files to the source directory:

```
cmake -DMIOPEN_BACKEND=HIP -DBUILD_DEV=On ..
```

To customize the database paths, use the `MIOPEN_SYSTEM_DB_PATH` (for the System PerfDb) and `MIOPEN_USER_DB_PATH` (for the User PerfDb) CMake variables.

To learn more, see *using the performance database*.

### 3.1.5 Persistent program cache

By default, MIOpen caches device programs in the `~/.cache/miopen/` directory. Within the cache directory, there is a directory for each version of MIOpen. To change the location of the cache directory during configuration, use the `-DMIOPEN_CACHE_DIR=<cache-directory-path>` flag.

To disable the cache during runtime, set the `MIOPEN_DISABLE_CACHE=1` environmental variable.

#### 3.1.5.1 For MIOpen version 2.3 and earlier

If the compiler changes or you modify the kernels, then you must delete the cache for the MIOpen version in use, for example, `rm -rf ~/.cache/miopen/<miopen-version-number>`. For more information, see *kernel cache*.

#### 3.1.5.2 For MIOpen version 2.4 and later

MIOpen's kernel cache directory is versioned, so cached kernels don't collide when upgrading from an earlier version.

### 3.1.6 Changing the CMake configuration

The configuration can be changed after running CMake by using `ccmake`:

```
ccmake ..
```

or

```
cmake-gui: cmake-gui ..
```

The `ccmake` program can be downloaded using the Linux package `cmake-curses-gui`, but is not available on Windows.

## 3.2 Building the library

You can build the library from the `build` directory using the `Release` configuration:

```
cmake --build . --config Release
```

or

```
make
```

You can install it using the `install` target:

```
cmake --build . --config Release --target install
```

or

```
make install
```

This command installs the library to the `CMAKE_INSTALL_PREFIX` directory path.

## 3.3 Building the driver

MIOpen provides an application driver that can run any layer in isolation and measure library performance and verification.

To build the driver, use the `MIOpenDriver` target:

```
cmake --build . --config Release --target MIOpenDriver
```

or

```
make MIOpenDriver
```

## 3.4 Running the tests

To run tests, use the `check` target:

```
cmake --build . --config Release --target check
```

or

```
make check
```

To build and run a single test, use the following commands:

```
cmake --build . --config Release --target test_tensor
./bin/test_tensor
```

## 3.5 Formatting the code

To format the code using `clang-format`, use this command:

```
clang-format-10 -style=file -i <path-to-source-file>
```

To format the code per commit, install githooks:

```
./.githooks/install
```

## 3.6 Storing large file using Git Large File Storage

Git Large File Storage (LFS) replaces large files, such as audio samples, videos, datasets, and graphics with text pointers inside Git, while storing the file contents on a remote server. MIOpen uses Git LFS to store large files, such as kernel database files (`*.kdb`), which are normally > 0.5 GB.

To install Git LFS, use these commands:

```
sudo apt install git-lfs
git lfs install
```

In the Git repository where you want to use Git LFS, track the file type using the following code. If the file type is already being tracked, you can skip this step:

```
git lfs track "*.file_type"
git add .gitattributes
```

To pull all files or a single large file, use:

```
git lfs pull --exclude=
```

or

```
git lfs pull --exclude= --include "filename"
```

Update the large files and push to GitHub using the following sequence of commands:

```
git add my_large_files
git commit -m "the message"
git push
```

## 3.7 Installing the dependencies manually

If you're using Ubuntu v16, you can install the `Boost` packages using:

```
sudo apt-get install libboost-dev
sudo apt-get install libboost-system-dev
sudo apt-get install libboost-filesystem-dev
```

**Note**

By default, MIOpen attempts to build with Boost statically linked libraries. To build with dynamically linked Boost libraries, use the `-DBoost_USE_STATIC_LIBS=Off` flag during the configuration stage. However, this is not recommended.

You must install the `half` header from the half website.

CHAPTER

FOUR

# BUILD MIOPEN FOR EMBEDDED SYSTEMS

Follow these steps to build and configure MIOpen for an embedded system.

1. Install the dependencies. The default install location is `/usr/local`:

```
cmake -P install_deps.cmake --minimum --prefix /some/local/dir
```

2. Create the build directory.

```
mkdir build; cd build;
```

3. Add the embedded build configuration.

   The minimum static build configuration, without an embedded precompiled kernels package or FindDb, is the following:

```
CXX=/opt/rocm/llvm/bin/clang++ cmake -DMIOPEN_BACKEND=HIP -DMIOPEN_EMBED_BUILD=On -
↪DCMAKE_PREFIX_PATH="/some/local/dir" ..
```

   To enable HIP kernels in MIOpen for embedded builds, add `-DMIOPEN_USE_HIP_KERNELS=On` to the command line. For example:

```
CXX=/opt/rocm/llvm/bin/clang++ cmake -DMIOPEN_BACKEND=HIP -DMIOPEN_USE_HIP_
↪KERNELS=On -DMIOPEN_EMBED_BUILD=On -DCMAKE_PREFIX_PATH="/some/local/dir" ..
```

4. Embed FindDb and PerfDb.

   FindDb provides a database of known convolution inputs. It allows you to use the best tuned kernels for your network. To embed on-disk databases in the binary with FindDb, use `DMIOPEN_EMBED_DB` with a semicolon-separated list of architecture-CU pairs (for example, `gfx906_60;gfx900_56`).

```
CXX=/opt/rocm/llvm/bin/clang++ cmake -DMIOPEN_EMBED_BUILD=On -DMIOPEN_EMBED_
↪DB=gfx900_56 ..
```

   This configures embedding for the build directory for both FindDb and PerfDb.

5. Embed the precompiled kernels package.

   An MIOpen build can embed the precompiled kernels package, preventing reduced performance due to compile-time overhead. This package contains the convolution kernels of known inputs to avoid the runtime compilation of kernels.

   There are two options for embedding the precompiled kernels package.

   - Embed the precompiled package using a package install.

```
apt-get install miopenkernels-<arch>-<num cu>
```

Where <arch> is the GPU architecture (for example, gfx900 or gfx906) and <num cu> is the number of compute units (CUs) available in the GPU (for example, 56 or 64).

There is no functional impact to MIOpen if you choose not to install the precompiled kernel package. This is because MIOpen compiles these kernels on the target machine after the kernel is run. However, the compilation step might significantly increase the startup time for some operations.

The utils/install_precompiled_kernels.sh script automates this process. It queries your system for the GPU architecture and then installs the appropriate package. To invoke it, use:

```
./utils/install_precompiled_kernels.sh
```

To embed the precompiled kernels package, configure CMake using MIOPEN_BINCACHE_PATH.

```
CXX=/opt/rocm/llvm/bin/clang++ cmake -DMIOPEN_BINCACHE_PATH=/path/to/package/
↪install -DMIOPEN_EMBED_BUILD=On ..
```

Here's an example that uses the gfx900 architecture and 56 CUs:

```
CXX=/opt/rocm/llvm/bin/clang++ cmake -DMIOPEN_BINCACHE_PATH=/opt/rocm/miopen/
↪share/miopen/db/gfx900_56.kdb -DMIOPEN_EMBED_BUILD=On ..
```

- Embed the precompiled package using the URL of a kernels binary. Use the MIOPEN_BINCACHE_PATH flag with the URL of the binary.

```
CXX=/opt/rocm/llvm/bin/clang++ cmake -DMIOPEN_BINCACHE_PATH=/URL/to/binary -
↪DMIOPEN_EMBED_BUILD=On ..
```

The precompiled kernels packages are installed in /opt/rocm/miopen/share/miopen/db.

As of ROCm version 3.8 and MIOpen version 2.7, precompiled kernels binaries are located at repo.radeon.com.

Here's an example that uses the gfx906 architecture and 64 CUs:

```
CXX=/opt/rocm/llvm/bin/clang++ cmake -DMIOPEN_BINCACHE_PATH=http://repo.radeon.
↪com/rocm/miopen-kernel/rel-3.8/gfx906_60.kdb -DMIOPEN_EMBED_BUILD=On ..
```

6. Full configuration line.

   To build MIOpen statically and embed the performance database, FindDb, and the precompiled kernels binary, follow this example:

```
CXX=/opt/rocm/llvm/bin/clang++ cmake -DMIOPEN_BINCACHE_PATH=/path/to/package/
↪install -DMIOPEN_EMBED_BUILD=On -DMIOPEN_EMBED_DB=gfx900_56 ..
```

   After configuration is complete, run the following command:

```
make -j
```

CHAPTER

FIVE

# BUILD MIOPEN USING DOCKER

You can build MIOpen using Docker by either downloading a prebuilt image or creating your own.

**Note**

For ease of use, the prebuilt Docker image is recommended.

- Downloading a prebuilt image

  You can find prebuilt Docker images at ROCm Docker Hub.

- Building your own image

  1. To build the Docker image, use `docker build`:

     ```
     docker build -t miopen-image .
     ```

  2. To enter the development environment, use `docker run`, for example:

     ```
     docker run -it -v $HOME:/data --privileged --rm --device=/dev/kfd --device /dev/
     ↪dri:/dev/dri:rw
     --volume /dev/dri:/dev/dri:rw -v /var/lib/docker/:/var/lib/docker --group-add␣
     ↪video
     --cap-add=SYS_PTRACE --security-opt seccomp=unconfined miopen-image
     ```

  3. Enter the Docker environment and run `git clone MIOpen`. You can now build MIOpen using CMake. For instructions on how to build MIOpen from source, see *building MIOpen*.

CHAPTER

SIX

# USING THE FIND DATABASE

MIOpen 2.0 introduced *immediate mode*, which is based on a find database called FindDb. This database contains the results of calls to the legacy `Find()` stage.

**Note**

Prior to MIOpen 2.0, you could use calls (such as `miopenFindConvolution*Algorithm()`) to gather a set of convolution algorithms in the form of an array of `miopenConvSolution_t` structs. This process is time consuming because it requires online benchmarking of competing algorithms.

FindDb consists of two parts:

- **System FindDb**: A system-wide storage that holds pre-run values for the most applicable configurations.
- **User FindDb**: A per-user storage that holds results for arbitrary user-run configurations. It also serves as a cache for the `Find()` stage.

The User FindDb *always takes precedence* over the System FindDb.

By default, System FindDb resides within the MIOpen install location, while User FindDb resides in your home directory.

**Note**

- The System FindDb is *not* modified upon installation of MIOpen.
- There are separate Find databases for the HIP and OpenCL backends.

## 6.1 Populating User FindDb

MIOpen collects FindDb information during the following API calls:

- `miopenFindConvolutionForwardAlgorithm()`
- `miopenFindConvolutionBackwardDataAlgorithm()`
- `miopenFindConvolutionBackwardWeightsAlgorithm()`

During the call, find data entries are collected for one specific "problem configuration", which is implicitly defined by the tensor descriptors and convolution descriptor passed to the API function.

## 6.2 Updating MIOpen and User FindDb

When you install a new version of MIOpen, the new version ignores old User FindDb files. Therefore, you don't need to move or delete the old User FindDb files.

To collect the previous information again into the new User FindDb, use the same steps you followed in the previous version. Re-collecting information keeps immediate mode optimized.

## 6.3 Disabling FindDb

To disable FindDb, set the `MIOPEN_DEBUG_DISABLE_FIND_DB` environmental variable to `1`:

```
export MIOPEN_DEBUG_DISABLE_FIND_DB=1
```

**Note**

System FindDb can be cached into memory, which might dramatically increase performance. To disable this option, set the `DMIOPEN_DEBUG_FIND_DB_CACHING` CMake configuration flag to off.

```
-DMIOPEN_DEBUG_FIND_DB_CACHING=Off
```

CHAPTER

SEVEN

# KERNEL CACHE

MIOpen caches binary kernels to disk so they don't need to be compiled the next time you run the application. This cache is stored in `$HOME/.cache/miopen` by default, but you can change this at build time by setting the `MIOPEN_CACHE_DIR` CMake variable.

## 7.1 Clear the cache

You can clear the cache by deleting the cache directory (for example, `$HOME/.cache/miopen`). However, you should only do this for development purposes or to free disk space. You don't need to clear the cache when upgrading MIOpen.

## 7.2 Disabling the cache

Disabling the cache is generally useful for development purposes. You can disable the cache in this following situations:

- During the build, either set `MIOPEN_CACHE_DIR` to an empty string or set `BUILD_DEV=ON` when configuring CMake.
- At runtime, set the `MIOPEN_DISABLE_CACHE` environment variable to `true`.

## 7.3 Updating MIOpen and removing the cache

For MIOpen version 2.4 and later, MIOpen's kernel cache directory is versioned, so any existing cached kernels won't collide when upgrading.

> **Note**
>
> For MIOpen version 2.3 and earlier, if the compiler changes or you modify the kernels, then you must delete the cache for the existing MIOpen version using the command `rm -rf $HOME/.cache/miopen/<miopen-version-number>`.

## 7.4 Installing precompiled kernels

GPU architecture-specific, precompiled kernel packages are available in the ROCm package repositories. These packages reduce the startup latency of MIOpen kernels. They contain a kernel cache file, which they install in the ROCm installation directory along with other MIOpen artifacts. When MIOpen launches a kernel, it first checks for a kernel in the kernel cache within the MIOpen installation directory. If the file doesn't exist, or the required kernel isn't found, it compiles the kernel and places it in the kernel cache.

These packages are optional and must be separately installed from MIOpen. To conserve disk space, you can choose not to install these packages (which would result in higher startup latency). You also have the option to only install kernel packages for your device architecture, which helps save disk space.

If the MIOpen kernels package is not installed, or if the kernel doesn't match the GPU, you'll get a warning message similar to:

```
> MIOpen(HIP): Warning [SQLiteBase] Missing system database file:gfx906_60.kdb␣
↪Performance may degrade
```

The performance degradation mentioned in the warning only affects the network start-up time (the "initial iteration time") and can be safely ignored.

Refer to the *installation instructions* for guidance on installing the MIOpen kernels package.

CHAPTER

EIGHT

# USING THE PERFORMANCE DATABASE

Many MIOpen kernels have parameters that affect their performance. Setting these parameters to optimal values allows for the best possible throughput. The optimal values depend on many factors, including the network configuration, GPU type, clock frequencies, and ROCm version.

Due to the large number of possible configurations and settings, MIOpen provides a set of pre-tuned values for the "most applicable" network configurations and a method for expanding the set of optimized values. MIOpen's performance database (PerfDb) contains these pre-tuned parameter values in addition to any user-optimized parameters.

The PerfDb consists of two parts:

- **System PerfDb**: A system-wide storage that holds pre-run values for the most applicable configurations.
- **User PerfDb**: A per-user storage that holds optimized values for arbitrary configurations.

The User PerfDb *always takes precedence* over System PerfDb.

MIOpen also has auto-tuning functionality, which is able to find optimized kernel parameter values for a specific configuration. The auto-tune process might take a long time, but after the optimized values are found, they're stored in the User PerfDb. MIOpen then automatically reads and uses these parameter values.

By default, System PerfDb resides within the MIOpen install location, while User PerfDb resides in your home directory. For more information, see *setting up locations*.

The System PerfDb is not modified during the MIOpen installation.

## 8.1 Auto-tuning kernels

MIOpen performs auto-tuning during the these API calls:

- `miopenFindConvolutionForwardAlgorithm()`
- `miopenFindConvolutionBackwardDataAlgorithm()`
- `miopenFindConvolutionBackwardWeightsAlgorithm()`

Auto-tuning is performed for only one "problem configuration", which is implicitly defined by the tensor descriptors that are passed to the API function.

In order for auto-tuning to begin, the following conditions must be met:

- The applicable kernels must have tuning parameters
- The value of the `exhaustiveSearch` parameter is set to `true`
- Neither the System nor User PerfDb can contain values for the relevant "problem configuration".

You can override the latter two conditions and force the search using the `-MIOPEN_FIND_ENFORCE` environment variable. You can also use this variable to remove values from User PerfDb, as described in the following section.

To optimize performance, MIOpen provides several find modes to accelerate find API calls. These modes include:

- normal find
- fast find
- hybrid find
- dynamic hybrid find

For more information about the MIOpen find modes, see *Find modes*.

### 8.1.1 Using MIOPEN_FIND_ENFORCE

`MIOPEN_FIND_ENFORCE` supports case-insensitive symbolic and numeric values. The possible values are:

- `NONE/(1)`: No change in the default behavior.
- `DB_UPDATE/(2)`: Do not skip auto-tune (even if PerfDb already contains optimized values). If you request auto-tune via API, MIOpen performs it and updates PerfDb. You can use this mode for fine-tuning the MIOpen installation on your system. However, this mode slows down the processes.
- `SEARCH/(3)`: Perform auto-tune even if not requested via API. In this case, the library behaves as if the `exhaustiveSearch` parameter is set to `true`. If PerfDb already contains optimized values, auto-tune is not performed. You can use this mode to tune applications that don't anticipate any means of getting the best performance from MIOpen. When in this mode, your application's first run might take substantially longer than expected.
- `SEARCH_DB_UPDATE/(4)`: A combination of `DB_UPDATE` and `SEARCH`. MIOpen performs auto-tune and updates User PerfDb on each `miopenFindConvolution*()` call. This mode is only recommended for debugging purposes.
- `DB_CLEAN/(5)`: Removes optimized values related to the "problem configuration" from User PerfDb. Auto-tune is blocked, even if explicitly requested. System PerfDb is left intact.

> **Caution**
>
> Use the `DB_CLEAN` option with care.

## 8.2 Updating MIOpen and User PerfDb

If you install a new version of MIOpen, it is recommended that you move or delete your old User PerfDb file. This prevents older database entries from affecting configurations within the newer system database. The User PerfDb is named `miopen.udb` and can be found at the User PerfDb path location.

CHAPTER

NINE

# TUNING PERFORMANCE DATABASES

A key element for ensuring the best performance is to do tuning on the shapes used by your model. MIOpen uses the following to decide upon the best solver to be used for a requested convolution:

- User DB: This stores the results of previous tunings and by default is found in "~/.config/miopen", although this can be changed.
- System DB: This is part of the MIOpen install files and has specific shapes that the MIOpen team has saved as tuning choices.
- Heuristics: This is a model trained on the shapes and solvers so that it picks the best solver and parameters. The goal is that this is within 90% of what can be achieved by specific tuning for the given shape, but manual tuning is expected to often give somewhat better results.

Manual tuning can either be incremental or exhaustive as detailed in the next sections.

## 9.1 Incremental tuning

This method conducts tuning when MIOpen encounters a new shape that doesn't exist in the user or system DBs. Tuning is conducted once for each missing shape and the results are saved to the user DB. Use this mode to add missing tuning information to the user database. This method defaults to heuristic find results when FDB entries are missing.

Enable this feature using these commands:

```
export MIOPEN_FIND_MODE=3
export MIOPEN_FIND_ENFORCE=3
export MIOPEN_USER_DB_PATH="/user/specified/directory"
```

### 9.1.1 Exhaustive tuning

#### 9.1.1.1 Method 1

Exhaustive tuning can be executed similarly to incremental tuning by ignoring the system DB. Redirecting the system DB path causes MIOpen to miss system entries and heuristic models, forcing a user DB entry to be generated. Each unique configuration is tuned exactly once, with results aggregated in the user DB files.

Enable this feature using these commands:

```
export MIOPEN_FIND_MODE=3
export MIOPEN_FIND_ENFORCE=3
export MIOPEN_USER_DB_PATH="/user/specified/directory"
export MIOPEN_SYSTEM_DB_PATH="$MIOPEN_USER_DB_PATH"
```

The default location for the MIOpen user database is "$HOME/.config/miopen"

#### 9.1.1.2 Method 2

To exhaustively tune all shapes explicitly, use `MIOPEN_ENABLE_LOGGING_CMD=1` with the target application to pull out all `MIOpenDriver` commands. Then enable exhaustive tuning and run the unique commands.

```
export MIOPEN_FIND_MODE=1
export MIOPEN_FIND_ENFORCE=4
export MIOPEN_USER_DB_PATH="/user/specified/directory"
```

## 9.2 Kernel cache

`MIOPEN_CUSTOM_CACHE_DIR` can be used to set the location of the user kernel cache, which is where compiled versions of kernels needed on your specific GPU will be saved. The presence of entries in this cache reduces first time run delay by reducing the compile time.

The cache directory defaults to "$HOME/.cache/miopen".

## 9.3 Post tuning

Unset `MIOPEN_FIND_MODE` and `MIOPEN_FIND_ENFORCE` to return to the default behavior. The behavior when these are set is expected to be the same as the default when all shapes are tuned, but some first run delay has been observed.

If `MIOPEN_SYSTEM_DB_PATH` was set for exhaustive tuning, it can now be unset to revert to the default system database install directory. The tuning entries created in the user database directory are preferred over the system entries.

The user DB at `MIOPEN_USER_DB_PATH` now contains all of the shapes that have been tuned. This path must continue to point to the newly created/updated user database files.

Unset these variables using these commands:

```
unset MIOPEN_FIND_MODE
unset MIOPEN_FIND_ENFORCE
unset MIOPEN_SYSTEM_DB_PATH
```

CHAPTER

# TEN

# MI200 MATRIX FUSED MULTIPLY-ADD (MFMA) BEHAVIOR SPECIFICS

On the MI200, `MFMA_F16`, `MFMA_BF16`, and `MFMA_BF16_1K` flush subnormal input/output data to zero. This behavior might affect the convolution operation in certain workloads due to the limited exponent range of the half-precision floating-point datatypes.

MIOpen offers an alternate implementation for the half-precision datatype via conversion instructions to utilize the larger exponent range of the `BFloat16` data type, albeit with reduced accuracy. The following caveats apply to this alternate implementation:

- It's disabled by default in the forward convolution operations.
- It's enabled by default in the backward data and backward weights convolution operations.
- You can override the default MIOpen behavior by using the `miopenSetConvolutionAttribute` API call. To use the alternate implementation, pass the convolution descriptor for the appropriate convolution operation and the `MIOPEN_CONVOLUTION_ATTRIB_FP16_ALT_IMPL` convolution attribute (with a non-zero value).
- You can also override the behavior using the `MIOPEN_DEBUG_CONVOLUTION_ATTRIB_FP16_ALT_IMPL` environment variable. When set to `1`, `MIOPEN_DEBUG_CONVOLUTION_ATTRIB_FP16_ALT_IMPL` engages the alternate implementation. When set to `0`, it's disabled. This environment variable impacts the convolution operation in all directions.

CHAPTER

# ELEVEN

# PORTING TO MIOPEN

The following is a summary of the key differences between MIOpen and NVIDIA CUDA cuDNN.

- Calling `miopenFindConvolution*Algorithm()` is **mandatory** before calling any Convolution API
- The typical calling sequence for the MIOpen Convolution APIs is:
  - `miopenConvolution*GetWorkSpaceSize()` (returns the workspace size required by `Find()`)
  - `miopenFindConvolution*Algorithm()` (returns the performance information for various algorithms)
  - `miopenConvolution*()`

MIOpen supports:

- 4D tensors in the NCHW and NHWC storage format. The CUDA cuDNN __"*Nd*"__ APIs don't have a corresponding MIOpen API.
- The __`float(fp32)`__ datatype
- __2D Convolutions__ and __3D Convolutions__
- __2D Pooling__

MIOpen doesn't support:

- __Preferences__ for convolutions
- Softmax modes (MIOpen implements the __SOFTMAX_MODE_CHANNEL__ flavor)
- __Transform-Tensor__, __Dropout__, __RNNs__, and __Divisive Normalization__

Useful MIOpen environment variables include:

- `MIOPEN_ENABLE_LOGGING=1`: Logs all the MIOpen APIs that are called, including the parameters passed to those APIs
- `MIOPEN_DEBUG_GCN_ASM_KERNELS=0`: Disables hand-tuned ASM kernels (the fallback is to use kernels written in a high-level language)
- `MIOPEN_DEBUG_CONV_FFT=0`: Disables the FFT convolution algorithm
- `MIOPEN_DEBUG_CONV_DIRECT=0`: Disables the direct convolution algorithm

## 11.1 CUDA cuDNN versus MIOpen APIs

The following sections compare the CUDA cuDNN and MIOpen APIs with similar functions.

### 11.1.1 Handle operations

| cuDNN | MIOpen |
|---|---|
| `cudnnStatus_t cudnnCreate( cudnnHandle_t *handle)` | `miopenStatus_t miopenCreate( miopenHandle_t *handle)` |
| `cudnnStatus_t cudnnDestroy( cudnnHandle_t handle)` | `miopenStatus_t miopenDestroy( miopenHandle_t handle)` |
| `cudnnStatus_t cudnnSetStream( cudnnHandle_t handle, cudaStream_t streamId)` | `miopenStatus_t miopenSetStream( miopenHandle_t handle, miopenAcceleratorQueue_t streamID)` |
| `cudnnStatus_t cudnnGetStream( cudnnHandle_t handle, cudaStream_t *streamId)` | `miopenStatus_t miopenGetStream( miopenHandle_t handle, miopenAcceleratorQueue_t *streamID)` |

### 11.1.2 Tensor operations

| cuDNN | MIOpen |
| --- | --- |
| `cudnnStatus_t cudnnCreateTensorDescriptor( cudnnTensorDescriptor_t *tensorDesc)` | `miopenStatus_t miopenCreateTensorDescriptor( miopenTensorDescriptor_t *tensorDesc)` |
| `cudnnStatus_t cudnnDestroyTensorDescriptor( cudnnTensorDescriptor_t tensorDesc)` | `miopenStatus_t miopenDestroyTensorDescriptor( miopenTensorDescriptor_t tensorDesc)` |
| `cudnnStatus_t cudnnSetTensor( cudnnHandle_t handle, const cudnnTensorDescriptor_t yDesc, void *y, const void *valuePtr)` | `miopenStatus_t miopenSetTensor( miopenHandle_t handle, const miopenTensorDescriptor_t yDesc, void *y, const void *alpha)` |
| `cudnnStatus_t cudnnSetTensor4dDescriptor( cudnnTensorDescriptor_t tensorDesc, cudnnTensorFormat_t format, cudnnDataType_t dataType, int n, int c, int h, int w)` | `miopenStatus_t miopenSet4dTensorDescriptor( miopenTensorDescriptor_t tensorDesc, miopenDataType_t dataType, int n, int c, int h, int w)` Only the NCHW format is supported |
| `cudnnStatus_t cudnnGetTensor4dDescriptor( cudnnTensorDescriptor_t tensorDesc, cudnnDataType_t *dataType, int *n, int *c, int *h, int *w, int *nStride, int *cStride, int *hStride, int *wStride)` | `miopenStatus_t miopenGet4dTensorDescriptor( miopenTensorDescriptor_t tensorDesc, miopenDataType_t *dataType, int *n, int *c, int *h, int *w, int *nStride, int *cStride, int *hStride, int *wStride)` |
| `cudnnStatus_t cudnnAddTensor( cudnnHandle_t handle, const void *alpha, const cudnnTensorDescriptor_t aDesc, const void *A, const void *beta, const cudnnTensorDescriptor_t cDesc, void *C)` | `miopenStatus_t miopenOpTensor( miopenHandle_t handle, miopenTensorOp_t tensorOp, const void *alpha1, constmiopenTensorDescriptor_t aDesc, const void *A, const void *alpha2, const miopenTensorDescriptor_t bDesc, const void *B, const void *beta, const miopenTensorDescriptor_t cDesc,` |

### 11.1.3 Filter operations

| cuDNN | MIOpen |
| --- | --- |
| `cudnnStatus_t cudnnCreateFilterDescriptor( cudnnFilterDescriptor_t *filterDesc)` | All `TensorDescriptor` APIs substitute for the respective `FilterDescriptor` APIs. |

### 11.1.4 Convolution operations

| cuDNN | MIOpen |
|---|---|
| cudnnStatus_t<br>cudnnCreateConvolutionDescriptor(<br>cudnnConvolutionDescriptor_t␣<br>↪*convDesc) | miopenStatus_t<br>miopenCreateConvolutionDescriptor(<br>miopenConvolutionDescriptor_t␣<br>↪*convDesc) |
| cudnnStatus_t<br>cudnnDestroyConvolutionDescriptor(<br>cudnnConvolutionDescriptor_t convDesc) | miopenStatus_t<br>miopenDestroyConvolutionDescriptor(<br>miopenConvolutionDescriptor_t␣<br>↪convDesc) |
| cudnnStatus_t<br>cudnnGetConvolution2dDescriptor(<br>const cudnnConvolutionDescriptor_t␣<br>↪convDesc,<br>int *pad_h,<br>int *pad_y,<br>int *u,<br>int *v,<br>int *upscalex,<br>int *upscaley,<br>cudnnConvolutionMode_t *mode) | miopenStatus_t<br>miopenGetConvolutionDescriptor(<br>miopenConvolutionDescriptor_t␣<br>↪convDesc,<br>miopenConvolutionMode_t *mode,<br>int *pad_h,<br>int *pad_y,<br>int *u,<br>int *v,<br>int *upscalex,<br>int *upscaley) |
| cudnnStatus_t<br>cudnnGetConvolution2dForwardOutputDim(<br>const cudnnConvolutionDescriptor_t␣<br>↪convDesc,<br>const cudnnTensorDescriptor_t␣<br>↪inputTensorDesc,<br>const cudnnFilterDescriptor_t␣<br>↪filterDesc,<br>int *n,<br>int *c,<br>int *h,<br>int *w) | miopenStatus_t<br>miopenGetConvolutionForwardOutputDim(<br>miopenConvolutionDescriptor_t␣<br>↪convDesc,<br>const miopenTensorDescriptor_t␣<br>↪inputTensorDesc,<br>const miopenTensorDescriptor_t␣<br>↪filterDesc,<br>int *n,<br>int *c,<br>int *h,<br>int *w) |
| cudnnStatus_t<br>cudnnGetConvolutionForwardWorkspaceSize(<br>cudnnHandle_t handle,<br>const cudnnTensorDescriptor_t xDesc,<br>const cudnnFilterDescriptor_t wDesc,<br>const cudnnConvolutionDescriptor_t␣<br>↪convDesc,<br>const cudnnTensorDescriptor_t yDesc,<br>cudnnConvolutionFwdAlgo_t algo,<br>size_t *sizeInBytes) | miopenStatus_t<br>miopenConvolutionForwardGetWorkSpaceSize(<br>miopenHandle_t handle,<br>const miopenTensorDescriptor_t wDesc,<br>const miopenTensorDescriptor_t xDesc,<br>const miopenConvolutionDescriptor_t␣<br>↪convDesc,<br>const miopenTensorDescriptor_t yDesc,<br>size_t *workSpaceSize) |
| cudnnStatus_t<br>cudnnGetConvolutionBackwardFilterWorkspaceSi<br>cudnnHandle_t handle,<br>const cudnnTensorDescriptor_t xDesc,<br>const cudnnTensorDescriptor_t dyDesc, | miopenStatus_t<br>miopenConvolutionBackwardWeightsGetWorkSpaceSize(<br>miopenHandle_t handle,<br>const miopenTensorDescriptor_t dyDesc,<br>const miopenTensorDescriptor_t xDesc, |

### 11.1.5 Softmax operations

| cuDNN | MIOpen |
| --- | --- |
| `cudnnStatus_t cudnnSoftmaxForward( cudnnHandle_t handle, cudnnSoftmaxAlgorithm_t algo, cudnnSoftmaxMode_t mode, const void *alpha, const cudnnTensorDescriptor_t xDesc, const void *x, const void *beta, const cudnnTensorDescriptor_t yDesc, void *y)` | `miopenStatus_t miopenSoftmaxForward( miopenHandle_t handle, const void *alpha, const miopenTensorDescriptor_t xDesc, const void *x, const void *beta, const miopenTensorDescriptor_t yDesc, void *y)` |
| `cudnnStatus_t cudnnSoftmaxBackward( cudnnHandle_t handle, cudnnSoftmaxAlgorithm_t algo, cudnnSoftmaxMode_t mode, const void *alpha, const cudnnTensorDescriptor_t yDesc, const void *y, const cudnnTensorDescriptor_t dyDesc, const void *dy, const void *beta, const cudnnTensorDescriptor_t dxDesc, void *dx)` | `miopenStatus_t miopenSoftmaxBackward( miopenHandle_t handle, const void *alpha, const miopenTensorDescriptor_t yDesc, const void *y, const miopenTensorDescriptor_t dyDesc, const void *dy, const void *beta, const miopenTensorDescriptor_t dxDesc, void *dx)` |

### 11.1.6 Pooling operations

| cuDNN | MIOpen |
|---|---|
| `cudnnStatus_t cudnnCreatePoolingDescriptor( cudnnPoolingDescriptor_t *poolingDesc)` | `miopenStatus_t miopenCreatePoolingDescriptor( miopenPoolingDescriptor_t *poolDesc)` |
| `cudnnStatus_t cudnnSetPooling2dDescriptor( cudnnPoolingDescriptor_t poolingDesc, cudnnPoolingMode_t mode, cudnnNanPropagation_t maxpoolingNanOpt, int windowHeight, int windowWidth, int verticalPadding, int horizontalPadding, int verticalStride, int horizontalStride)` | `miopenStatus_t miopenSet2dPoolingDescriptor( miopenPoolingDescriptor_t poolDesc, miopenPoolingMode_t mode, int windowHeight, int windowWidth, int pad_h, int pad_w, int u, int v)` |
| `cudnnStatus_t cudnnGetPooling2dDescriptor( const cudnnPoolingDescriptor_t poolingDesc, cudnnPoolingMode_t *mode, cudnnNanPropagation_t *maxpoolingNanOpt, int *windowHeight, int *windowWidth, int *verticalPadding, int *horizontalPadding, int *verticalStride, int *horizontalStride)` | `miopenStatus_t miopenGet2dPoolingDescriptor( const miopenPoolingDescriptor_t poolDesc, miopenPoolingMode_t *mode, int *windowHeight, int *windowWidth, int *pad_h, int *pad_w, int *u, int *v)` |
| `cudnnStatus_t cudnnGetPooling2dForwardOutputDim( const cudnnPoolingDescriptor_t poolingDesc, const cudnnTensorDescriptor_t inputTensorDesc, int *n, int *c, int *h, int *w)` | `miopenStatus_t miopenGetPoolingForwardOutputDim( const miopenPoolingDescriptor_t poolDesc, const miopenTensorDescriptor_t tensorDesc, int *n, int *c, int *h, int *w)` |
| `cudnnStatus_t cudnnDestroyPoolingDescriptor( cudnnPoolingDescriptor_t poolingDesc)` | `miopenStatus_t miopenDestroyPoolingDescriptor( miopenPoolingDescriptor_t poolDesc)` |
| `cudnnStatus_t cudnnPoolingForward( cudnnHandle_t handle, const cudnnPoolingDescriptor_t poolingDesc,` | `miopenStatus_t miopenPoolingForward( miopenHandle_t handle, const miopenPoolingDescriptor_t poolDesc,` |

### 11.1.7 Activation operations

| cuDNN | MIOpen |
|---|---|
| cudnnStatus_t<br>cudnnCreateActivationDescriptor(<br>cudnnActivationDescriptor_t␣<br>↪*activationDesc) | miopenStatus_t<br>miopenCreateActivationDescriptor(<br>miopenActivationDescriptor_t␣<br>↪*activDesc) |
| cudnnStatus_t<br>cudnnSetActivationDescriptor(<br>cudnnActivationDescriptor_t␣<br>↪activationDesc,<br>cudnnActivationMode_t mode,<br>cudnnNanPropagation_t reluNanOpt,<br>double reluCeiling) | miopenStatus_t<br>miopenSetActivationDescriptor(<br>const miopenActivationDescriptor_t␣<br>↪activDesc,<br>miopenActivationMode_t mode,<br>double activAlpha,<br>double activBeta,<br>double activPower) |
| cudnnStatus_t<br>cudnnGetActivationDescriptor(<br>const cudnnActivationDescriptor_t␣<br>↪activationDesc,<br>cudnnActivationMode_t *mode,<br>cudnnNanPropagation_t *reluNanOpt,<br>double *reluCeiling) | miopenStatus_t<br>miopenGetActivationDescriptor(<br>const miopenActivationDescriptor_t␣<br>↪activDesc,<br>miopenActivationMode_t *mode,<br>double *activAlpha,<br>double *activBeta,<br>double *activPower) |
| cudnnStatus_t<br>cudnnDestroyActivationDescriptor(<br>cudnnActivationDescriptor_t␣<br>↪activationDesc) | miopenStatus_t<br>miopenDestroyActivationDescriptor(<br>miopenActivationDescriptor_t␣<br>↪activDesc) |
| cudnnStatus_t<br>cudnnActivationForward(<br>cudnnHandle_t handle,<br>cudnnActivationDescriptor_t␣<br>↪activationDesc,<br>const void *alpha,<br>const cudnnTensorDescriptor_t xDesc,<br>const void *x,<br>const void *beta,<br>const cudnnTensorDescriptor_t yDesc,<br>void *y) | miopenStatus_t<br>miopenActivationForward(<br>miopenHandle_t handle,<br>const miopenActivationDescriptor_t␣<br>↪activDesc,<br>const void *alpha,<br>const miopenTensorDescriptor_t xDesc,<br>const void *x,<br>const void *beta,<br>const miopenTensorDescriptor_t yDesc,<br>void *y) |
| cudnnStatus_t<br>cudnnActivationBackward(<br>cudnnHandle_t handle,<br>cudnnActivationDescriptor_t␣<br>↪activationDesc,<br>const void *alpha,<br>const cudnnTensorDescriptor_t yDesc,<br>const void *y,<br>const cudnnTensorDescriptor_t dyDesc,<br>const void *dy,<br>const cudnnTensorDescriptor_t xDesc, | miopenStatus_t<br>miopenActivationBackward(<br>miopenHandle_t handle,<br>const miopenActivationDescriptor_t␣<br>↪activDesc,<br>const void *alpha,<br>const miopenTensorDescriptor_t yDesc,<br>const void *y,<br>const miopenTensorDescriptor_t dyDesc,<br>const void *dy,<br>const miopenTensorDescriptor_t xDesc, |

### 11.1.8 LRN operations

| cuDNN | MIOpen |
|---|---|
| `cudnnStatus_t cudnnCreateLRNDescriptor( cudnnLRNDescriptor_t *normDesc)` | `miopenStatus_t miopenCreateLRNDescriptor( miopenLRNDescriptor_t *lrnDesc)` |
| `cudnnStatus_t cudnnSetLRNDescriptor( cudnnLRNDescriptor_t normDesc, unsigned lrnN, double lrnAlpha, double lrnBeta, double lrnK)` | `miopenStatus_t miopenSetLRNDescriptor( const miopenLRNDescriptor_t lrnDesc, miopenLRNMode_t mode, unsigned lrnN, double lrnAlpha, double lrnBeta, double lrnK)` |
| `cudnnStatus_t cudnnGetLRNDescriptor( cudnnLRNDescriptor_t normDesc, unsigned* lrnN, double* lrnAlpha, double* lrnBeta, double* lrnK)` | `miopenStatus_t miopenGetLRNDescriptor( const miopenLRNDescriptor_t lrnDesc, miopenLRNMode_t *mode, unsigned *lrnN, double *lrnAlpha, double *lrnBeta, double *lrnK)` |
| `cudnnStatus_t cudnnDestroyLRNDescriptor( cudnnLRNDescriptor_t lrnDesc)` | `miopenStatus_t miopenDestroyLRNDescriptor( miopenLRNDescriptor_t lrnDesc)` |
| `cudnnStatus_t cudnnLRNCrossChannelForward( cudnnHandle_t handle, cudnnLRNDescriptor_t normDesc, cudnnLRNMode_t lrnMode, const void* alpha, const cudnnTensorDescriptor_t xDesc, const void *x, const void *beta, const cudnnTensorDescriptor_t yDesc, void *y)` | `miopenStatus_t miopenLRNForward( miopenHandle_t handle, const miopenLRNDescriptor_t lrnDesc, const void *alpha, const miopenTensorDescriptor_t xDesc, const void *x, const void *beta, const miopenTensorDescriptor_t yDesc, void *y, bool do_backward, void  *workspace)` |
| `cudnnStatus_t cudnnLRNCrossChannelBackward( cudnnHandle_t handle, cudnnLRNDescriptor_t normDesc, cudnnLRNMode_t lrnMode, const void* alpha, const cudnnTensorDescriptor_t yDesc, const void *y, const cudnnTensorDescriptor_t dyDesc, const void *dy, const cudnnTensorDescriptor_t xDesc,` | `miopenStatus_t miopenLRNBackward( miopenHandle_t handle, const miopenLRNDescriptor_t lrnDesc, const void *alpha, const miopenTensorDescriptor_t yDesc, const void *y, const miopenTensorDescriptor_t dyDesc, const void *dy, const miopenTensorDescriptor_t xDesc, const void *x, const void *beta,` |

### 11.1.9 Batch normalization operations

| cuDNN | MIOpen |
|---|---|
| cudnnStatus_t<br>cudnnBatchNormalizationForwardTraining(<br>cudnnHandle_t handle,<br>cudnnBatchNormMode_t mode,<br>void *alpha,<br>void *beta,<br>const cudnnTensorDescriptor_t xDesc,<br>const void *x,<br>const cudnnTensorDescriptor_t yDesc,<br>void *y,<br>const cudnnTensorDescriptor_t<br>bnScaleBiasMeanVarDesc,<br>void *bnScale,<br>void *bnBias,<br>double exponentialAverageFactor,<br>void *resultRunningMean,<br>void *resultRunningVariance,<br>double epsilon,<br>void *resultSaveMean,<br>void *resultSaveInvVariance) | miopenStatus_t<br>miopenBatchNormalizationForwardTraining(<br>miopenHandle_t handle,<br>miopenBatchNormMode_t bn_mode,<br>void *alpha,<br>void *beta,<br>const miopenTensorDescriptor_t xDesc,<br>const void *x,<br>const miopenTensorDescriptor_t yDesc,<br>void *y,<br>const miopenTensorDescriptor_t<br>bnScaleBiasMeanVarDesc,<br>void *bnScale,<br>void *bnBias,<br>double expAvgFactor,<br>void *resultRunningMean,<br>void *resultRunningVariance,<br>double epsilon,<br>void *resultSaveMean,<br>void *resultSaveInvVariance) |
| cudnnStatus_t<br>cudnnnBatchNormalizationForwardInference(<br>cudnnHandle_t handle,<br>cudnnBatchNormMode_t mode,<br>void *alpha,<br>void *beta,<br>const cudnnTensorDescriptor_t xDesc,<br>const void *x,<br>const cudnnTensorDescriptor_t yDesc,<br>void *y,<br>const cudnnTensorDescriptor_t<br>bnScaleBiasMeanVarDesc,<br>const void *bnScale,<br>void *bnBias,<br>const void *estimatedMean,<br>const void *estimatedVariance,<br>double epsilon) | miopenStatus_t<br>miopenBatchNormalizationForwardInference(<br>miopenHandle_t handle,<br>miopenBatchNormMode_t bn_mode,<br>void *alpha,<br>void *beta,<br>const miopenTensorDescriptor_t xDesc,<br>const void *x,<br>const miopenTensorDescriptor_t yDesc,<br>void *y,<br>const miopenTensorDescriptor_t<br>bnScaleBiasMeanVarDesc,<br>void *bnScale,<br>void *bnBias,<br>void *estimatedMean,<br>void *estimatedVariance,<br>double epsilon) |
| cudnnStatus_t<br>cudnnBatchNormalizationBackward(<br>cudnnHandle_t handle,<br>cudnnBatchNormMode_t mode,<br>const void *alphaDataDiff,<br>const void *betaDataDiff,<br>const void *alphaParamDiff,<br>const void *betaParamDiff,<br>const cudnnTensorDescriptor_t xDesc,<br>const void *x,<br>const cudnnTensorDescriptor_t dyDesc,<br>const void *dy,<br>const cudnnTensorDescriptor_t dxDesc,<br>void *dx, | miopenStatus_t<br>miopenBatchNormalizationBackward(<br>miopenHandle_t handle,<br>miopenBatchNormMode_t bn_mode,<br>const void *alphaDataDiff,<br>const void *betaDataDiff,<br>const void *alphaParamDiff,<br>const void *betaParamDiff,<br>const miopenTensorDescriptor_t xDesc,<br>const void *x,<br>const miopenTensorDescriptor_t dyDesc,<br>const void *dy,<br>const miopenTensorDescriptor_t dxDesc,<br>void *dx, |

CHAPTER

# TWELVE

# USING THE FUSION API

Increasing the depth of deep-learning networks requires novel mechanisms to improve GPU performance. One mechanism to achieve higher efficiency is to *fuse* separate kernels into a single kernel in order to reduce off-chip memory access and avoid kernel launch overhead.

Using MIOpen's fusion API, you can specify operators that you want to fuse into a single kernel, compile that kernel, and then launch it. While not all combinations are supported, the API is flexible enough to allow the specification of several operations, in any order, from the set of supported operations. The API provides a mechanism to report unsupported combinations.

You can find a complete example of MIOpen's fusion API in the MIOpen GitHub repository example folder. The code examples in this document are taken from this example project.

**Note**

The example project creates a fusion plan to merge the convolution, bias, and activation operations. For a list of supported fusion operations and associated constraints, see the *Supported fusions* section. For simplicity, the example doesn't populate the tensors with meaningful data and only shows the basic code without any error checking.

After you've initialized an MIOpen handle object, the workflow for using the fusion API is:

- Create a fusion plan
- Create and add the convolution, bias, and activation operators
- Compile the fusion plan
- Set the runtime arguments for each operator
- Run the fusion plan
- Cleanup

The order in which you create operators is important because this order represents the order of operations for the data. Therefore, a fusion plan where convolution is created before activation differs from a fusion plan where activation is added before convolution.

**Note**

The primary consumers of the fusion API are high-level frameworks, such as TensorFlow/XLA and PyTorch.

## 12.1 Creating a fusion plan

A *fusion plan* is the data structure that holds all the metadata regarding fusion intent along with the logic to compile and run a fusion plan. The fusion plan not only contains the order in which different operations are applied on the data, but also specifies the *axis* of fusion. Currently, only *vertical* (sequential) fusions are supported, implying the flow of data between operations is sequential.

You can create a fusion plan using `miopenCreateFusionPlan`, as follows:

```
miopenStatus_t
miopenCreateFusionPlan(miopenFusionPlanDescriptor_t* fusePlanDesc,
const miopenFusionDirection_t fuseDirection,const miopenTensorDescriptor_t inputDesc);
```

The *input tensor descriptor* specifies the geometry of the incoming data. Because the data geometry of the intermediate operations can be derived from the input tensor descriptor, this is only required for the fusion plan. The input tensor descriptor isn't required for the individual operations.

```
miopenCreateFusionPlan(&fusePlanDesc, miopenVerticalFusion, input.desc);
```

In the previous example, `fusePlanDesc` is an object of type `miopenFusionPlanDescriptor_t` and `input.desc` is the `miopenTensorDescriptor_t` object.

## 12.2 Creating and adding operators

Operators represent the different operations to fuse. Currently, the API supports these operators:

- Convolution forward
- Activation forward
- BatchNorm inference
- Bias forward

> **Note**
>
> Although bias is a separate operator, it's typically only used with convolution.

MIOpen plans to add support for more operators, including operators for backward passes, in the future.

The fusion API provides calls for the creation of the supported operators. To learn more, refer to the Fusion API documentation.

After you've created the fusion plan descriptor, you can add two or more operators to it by using the individual operator creation API calls. If the API doesn't support the fusion of the operations you add, the creation might fail.

This example adds the convolution, bias, and activation operations to the newly created fusion plan.

```
miopenStatus_t
miopenCreateOpConvForward(miopenFusionPlanDescriptor_t fusePlanDesc,
                          miopenFusionOpDescriptor_t* convOp,
                          miopenConvolutionDescriptor_t convDesc,
                          const miopenTensorDescriptor_t wDesc);
miopenStatus_t
miopenCreateOpBiasForward(miopenFusionPlanDescriptor_t fusePlanDesc,
                          miopenFusionOpDescriptor_t* biasOp,
```

(continues on next page)

(continued from previous page)

```
                            const miopenTensorDescriptor_t bDesc);

miopenStatus_t
miopenCreateOpActivationForward(miopenFusionPlanDescriptor_t fusePlanDesc,
                                miopenFusionOpDescriptor_t* activOp,
                                miopenActivationMode_t mode);
```

`conv_desc` is the regular MIOpen convolution descriptor. For more information on creating and setting this descriptor, see the example code and the Convolution API documentation.

`weights.desc` refers to `miopenTensorDescriptor_t` for the convolution operations. `bias.desc` refers to the object of the same type for the bias operation.

In the preceding code, the convolution operation is the first operation to run on the incoming data, followed by the bias, and then activation operations.

During this process, it is important to verify the return codes to ensure the operations and sequence are supported. The operator insertion can fail for a number of reasons, such as an unsupported operation sequence, unsupported input dimensions, or, in the case of convolution, unsupported filter dimensions. In the preceding example, these aspects are ignored for the sake of simplicity.

## 12.3 Compiling the fusion plan

Following the addition of the operators, you can compile the fusion plan. This populates the MIOpen kernel cache with the fused kernel and gets it ready to run.

```
miopenStatus_t
miopenCompileFusionPlan(miopenHandle_t handle, miopenFusionPlanDescriptor_t␣
↪fusePlanDesc);
```

The corresponding code snippet in the example is:

```
auto status = miopenCompileFusionPlan(mio::handle(), fusePlanDesc);
if (status != miopenStatusSuccess) {
return -1;
}
```

To compile the fusion plan, you must acquire an MIOpen handle object. In the preceding code, this is accomplished using the `mio::handle()` helper function. While a fusion plan is itself not bound to an MIOpen handle object, it must be recompiled separately for each handle.

Compiling a fusion plan is a costly operation in terms of run-time, and compilation can fail for a number of reasons. Therefore, the recommendation is to only compile your fusion plan once and reuse it with different runtime parameters, as described in the next section.

## 12.4 Setting runtime arguments

While the fusion operator for the underlying MIOpen descriptor specifies the data geometry and parameters, the fusion plan still needs access to the data to run a successfully compiled fusion plan. The arguments mechanism in the fusion API provides this data before a fusion plan can be run. For example, the convolution operator requires *weights* to carry out the convolution computation, and the bias operator requires the actual bias values. Therefore, before you can run a fusion plan, you must specify the arguments required by each fusion operator.

First create the `miopenOperatorArgs_t` object using this code:

```
miopenStatus_t miopenCreateOperatorArgs(miopenOperatorArgs_t* args);
```

After it is created, you can set the runtime arguments for each operation. In this example, the forward convolution operator requires the convolution weights argument, which is supplied using:

```
miopenStatus_t
miopenSetOpArgsConvForward(miopenOperatorArgs_t args,
                           const miopenFusionOpDescriptor_t convOp,
                           const void* alpha,
                           const void* beta,
                           const void* w);
```

Similarly, the parameters for bias and activation are supplied by:

```
miopenStatus_t miopenSetOpArgsBiasForward(miopenOperatorArgs_t args,
                                          const miopenFusionOpDescriptor_t biasOp,
                                          const void* alpha,
                                          const void* beta,
                                          const void* bias);

miopenStatus_t miopenSetOpArgsActivForward(miopenOperatorArgs_t args,
                                          const miopenFusionOpDescriptor_t activOp,
                                          const void* alpha,
                                          const void* beta,
                                          double activAlpha,
                                          double activBeta,
                                          double activGamma);
```

In the example code, the arguments for the operations are set as follows:

```
miopenSetOpArgsConvForward(fusionArgs, convoOp, &alpha, &beta, weights.data);
miopenSetOpArgsActivForward(fusionArgs, activOp, &alpha, &beta, activ_alpha,
                           activ_beta, activ_gamma);
miopenSetOpArgsBiasForward(fusionArgs, biasOp, &alpha, &beta, bias.data);
```

Having a separation between the fusion plan and the arguments required by each operator allows better reuse of the fusion plan with different arguments. It also avoids the necessity to recompile the fusion plan to run the same combination of operators with different arguments.

As previously mentioned, the compilation step for a fusion plan can be costly. Therefore, it is recommended that you only compile a fusion plan once in its lifetime. A fusion plan doesn't need to be recompiled if the input descriptor or any of the parameters in the `miopenCreateOp*` API calls are different. You can repeatedly reuse a compiled fusion plan with a different set of arguments.

In the example, this is demonstrated in `main.cpp`, lines 77 through 85.

## 12.5 Running a fusion plan

Once you've compiled the fusion plan and set the arguments for each operator, you can run it as follows:

```
miopenStatus_t
miopenExecuteFusionPlan(const miopenHandle_t handle,
                        const miopenFusionPlanDescriptor_t fusePlanDesc,
                        const miopenTensorDescriptor_t inputDesc,
```

(continues on next page)

(continued from previous page)

```
                        const void* input,
                        const miopenTensorDescriptor_t outputDesc,
                        void* output,
                        miopenOperatorArgs_t args);
```

The following code snippet runs the fusion plan:

```
miopenExecuteFusionPlan(mio::handle(), fusePlanDesc, input.desc, input.data,
                        output.desc, output.data, fusionArgs);
```

If you try to run a fusion plan that is not compiled, or has been invalidated by changing the input tensor descriptor or any of the operation parameters, you'll get an error.

## 12.6 Cleanup

After the application is finished with the fusion plan, you can destroy the fusion plan and the fusion `args` objects:

```
miopenStatus_t miopenDestroyFusionPlan(miopenFusionPlanDescriptor_t fusePlanDesc);
```

After the fusion plan object is destroyed, all the operations are automatically destroyed. You don't need to worry about additional cleanup.

## 12.7 Supported fusions

The following tables outline the supported fusions for `FP32` and `FP16`, including any applicable constraints.

> **Note**
>
> Fusion Plans with grouped convolutions are not supported.

**C = convolution, B = bias, N = batch normalization, A = activation**

## Convolution based FP32 Fusion for Inference

| Single Precision Floating Point | | | | | | |
|---|---|---|---|---|---|---|
| **Combination** | **Conv Algo** | **Stride** | **Filter Dims** | **N Mode*** | **Activations** | **Other Constraints** |
| CBNA | Direct | 1 and 2 | 3x3, 5x5, 7x7, 9x9, 11x11 | All | All | stride and padding must be either 1 or 2 |
| CBA | Direct | | 1x1 | | All | stride/ padding not supported |
| | Winograd | 1 | 1x1, 2x2 | N/A | Relu, Leaky Relu | c >= 18 |
| | | 1 | 3x3 | | Relu, Leaky Relu | c >= 18 and c is even |
| | | 1 | 4x4, 5x5, 6x6 | | Relu, Leaky Relu | 4 x c >= 18 |
| | | 1 | 7x7, 8x8, 9x9 | | Relu, Leaky Relu | 12 x c >= 18 |
| | | 1 | 10x10, 11x11, 12x12 | | Relu, Leaky Relu | 16 x c >= 18 |
| | | 1 | larger filter sizes | | Relu, Leaky Relu | none |
| | | 2 | 1x1 | | Relu, Leaky Relu | 2 x c >= 18 |
| | | 2 | 2x2, 3x3, 4x4, 5x5, 6x6 | | Relu, Leaky Relu | 4 x c >= 18 |
| | | 2 | 7x7 | | Relu, Leaky Relu | 12 x c >= 18 |
| | | 2 | 8x8, 9x9, 10x10, 11x11, 12x12 | | Relu, Leaky Relu | 16 x c >= 18 |
| | | 2 | larger filter sizes | | Relu, Leaky Relu | none |
| NA | - | | - | All | All | Padding not supported |

*N mode is either spatial, or per activation. For CBA other asymmetric kernels are supported as well, but are not enumerated here for brevity.

### Convolution based FP16 Fusion for Inference

| Half Precision Floating Point | | | | | | |
|---|---|---|---|---|---|---|
| Combination | Conv Algo | Stride | Filter Dims | N Mode* | Activations | Other Constraints |
| CBNA | Direct | 1 and 2 | 3x3, 5x5, 7x7, 9x9, 11x11 | All | All | stride and padding must be either 1 or 2 |
| CBA | Direct | | 1x1 | | All | stride/ padding not supported |

*N mode is either spatial, or per activation.

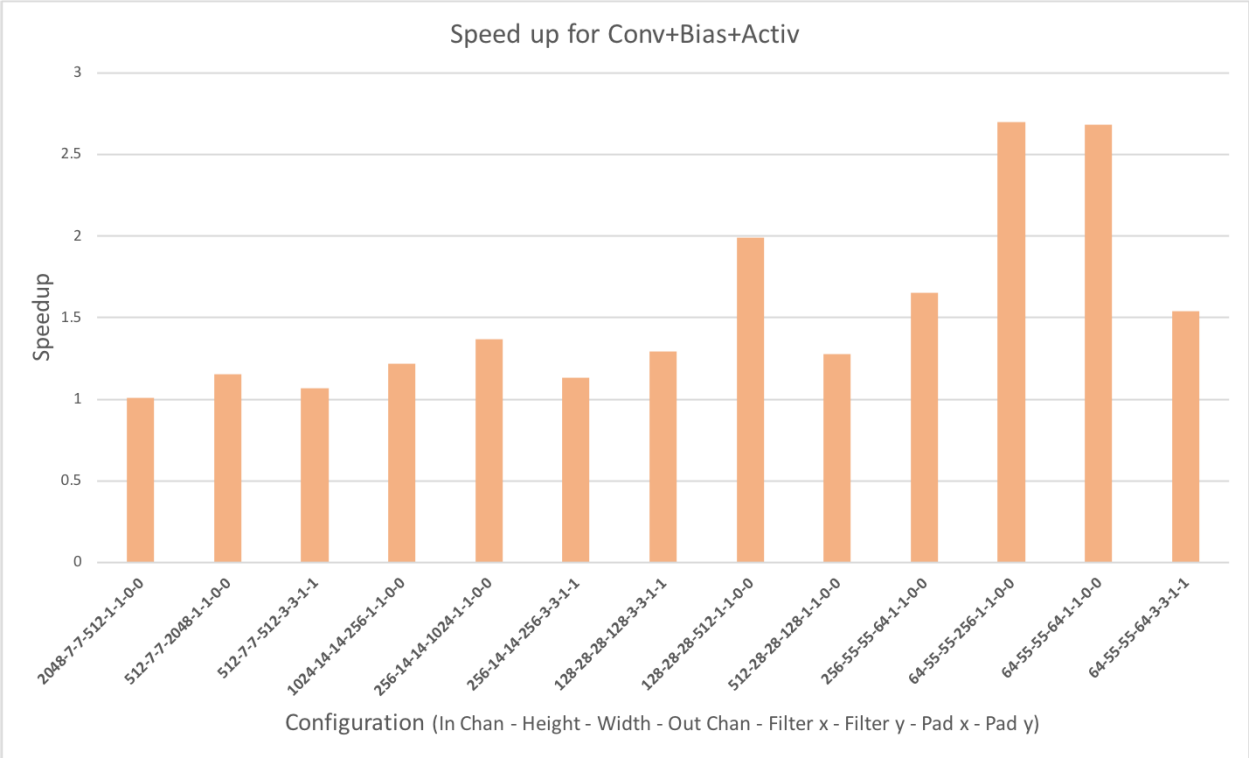
### Batch Normalization based fusion for FP32 and FP16 for Inference and Training

| Combination | N mode* | Activations | Constraints |
|---|---|---|---|
| NA for inference | All | All | None |
| NA forward training | All | All | None |
| NA backward training | All | All | None |

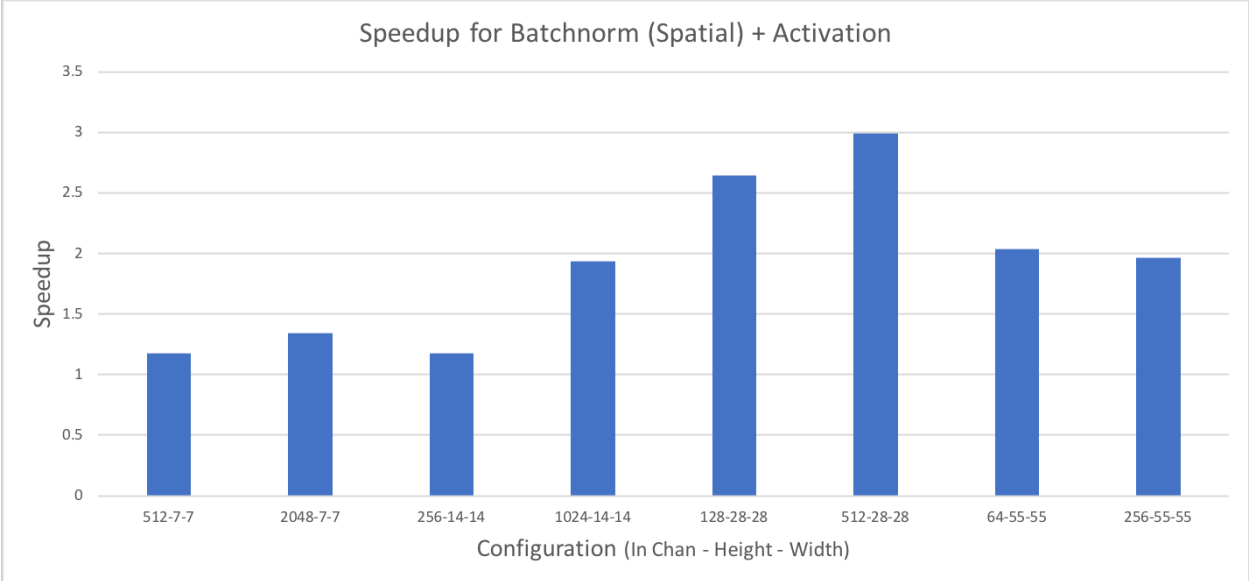
*N mode is either spatial, or per activation.

## 12.8 Comparing performance with non-fused kernels

The following graph depicts the speedup gained for a fused convolution+bias+activation over a non-fused version. All configurations have a batch size of 64:




The following graph depicts the speedup obtained by fusing BatchNorm (in spatial mode) with activation:



CHAPTER

# THIRTEEN

# LOGGING AND DEBUGGING

All logging messages are output to the standard error stream (`stderr`). You can use the following environmental variables to control logging. Both variables are disabled by default.

- `MIOPEN_ENABLE_LOGGING`: Prints the basic layer-by-layer MIOpen API call information with the actual parameters and configurations. This information is important for debugging.
  - To enable the feature: `1`, `on`, `yes`, `true`, `enable`, or `enabled`
  - To disable the feature: `0`, `off`, `no`, `false`, `disable`, or `disabled`
- `MIOPEN_ENABLE_LOGGING_CMD`: Outputs the associated `MIOpenDriver` command lines to the console.
  - To enable the feature: `1`, `on`, `yes`, `true`, `enable`, or `enabled`
  - To disable the feature: `0`, `off`, `no`, `false`, `disable`, or `disabled`
- `MIOPEN_LOG_LEVEL`: In addition to API call information and driver commands, MIOpen logs information related to the progress of its internal operations. This information can be useful for debugging and understanding how the library works. `MIOPEN_LOG_LEVEL` controls the verbosity of these messages. The allowed values are:
  - `0`: Default. Works as level `4` for release builds and level `5` for debug builds.
  - `1`: Quiet. No logging messages.
  - `2`: Fatal errors only (unused).
  - `3`: Errors, including fatal errors.
  - `4`: All errors and warnings.
  - `5`: Info. All the preceding information, plus information for debugging purposes.
  - `6`: Detailed information. All the preceding information, plus more detailed information for debugging.
  - `7`: Trace. All the preceding information, plus additional details.
- `MIOPEN_ENABLE_LOGGING_MPMT`: Each log line is prefixed with information to identify records printed from different processes or threads. This is useful for debugging multi-process and multi-threaded applications.
- `MIOPEN_ENABLE_LOGGING_ELAPSED_TIME`: Adds a timestamp to each log line that indicates the time elapsed (in milliseconds) since the previous log message.

**Note**

If you require technical support, include the console log that is produced from these settings:

```
export MIOPEN_ENABLE_LOGGING=1
export MIOPEN_ENABLE_LOGGING_CMD=1
export MIOPEN_LOG_LEVEL=6
```

## 13.1 Layer filtering

The following sections describe environment variables for enabling or disabling various types of kernels and algorithms. These are helpful for debugging MIOpen and framework integrations.

For these environment variables, you can use the following values:

- To enable the kernel/algorithm: `1`, `yes`, `true`, `enable`, or `enabled`
- To disable the kernel/algorithm: `0`, `no`, `false`, `disable`, or `disabled`

**Warning**

When you use the library with layer filtering, the results of `*Find()` calls become narrower than during normal operation. This means that relevant FindDb entries won't include all the solutions that are normally present. Therefore, the subsequent immediate mode `*Get()` calls may return incomplete information or run into the fallback path.

In order to repair immediate mode, you can:

- Re-enable all solvers and rerun the same `*Find()` calls from before
- Completely remove the User FindDb

If a variable is not set, MIOpen treats it as `enabled`. This means that kernels and algorithms are enabled by default.

### 13.1.1 Filtering by algorithm

These variables control the sets (families) of convolution solutions. For example, the direct algorithm is implemented in several solutions that use OpenCL and GCN assembly. The corresponding variable is used to disable them.

- `MIOPEN_DEBUG_CONV_FFT`: FFT convolution algorithm.
- `MIOPEN_DEBUG_CONV_DIRECT`: Direct convolution algorithm.
- `MIOPEN_DEBUG_CONV_GEMM`: GEMM convolution algorithm.
- `MIOPEN_DEBUG_CONV_WINOGRAD`: Winograd convolution algorithm.
- `MIOPEN_DEBUG_CONV_IMPLICIT_GEMM`: Implicit GEMM convolution algorithm.

### 13.1.2 Filtering by build method

- `MIOPEN_DEBUG_GCN_ASM_KERNELS`: Kernels written in assembly language. These are used in many convolutions (some direct solvers, Winograd kernels, and fused convolutions) and batch normalization.
- `MIOPEN_DEBUG_HIP_KERNELS`: Convolution kernels written in HIP. These implement the ImplicitGemm algorithm.

- MIOPEN_DEBUG_OPENCL_CONVOLUTIONS: Convolution kernels written in OpenCL. This only affects convolutions.

### 13.1.3 Filtering out all but one solution

- MIOPEN_DEBUG_FIND_ONLY_SOLVER=solution_id: Only directly affects *Find() calls. However, there is an indirect connection to immediate mode (see the previous warning).
  - solution_id must be a numeric or a string identifier of some solution.
  - If the solution_id denotes an applicable solution, then only that solution is found (in addition to GEMM and FFT, if applicable). See the note in this section.
  - If the solution_id is valid, but not applicable, then *Find() fails with all algorithms (except for GEMM and FFT). See the note in this section.
  - Otherwise, the solution_id is invalid (for instance, it doesn't match any existing solution) and the *Find() call fails.

**Note**

This environmental variable doesn't affect the GEMM and FFT solutions. For now, GEMM and FFT can only be disabled at the algorithm level.

### 13.1.4 Filtering the solutions on an individual basis

Some of the solutions have individual controls, which affect both find and immediate modes.

- Direct solutions:
  - MIOPEN_DEBUG_CONV_DIRECT_ASM_3X3U – ConvAsm3x3U
  - MIOPEN_DEBUG_CONV_DIRECT_ASM_1X1U – ConvAsm1x1U
  - MIOPEN_DEBUG_CONV_DIRECT_ASM_1X1UV2 – ConvAsm1x1UV2
  - MIOPEN_DEBUG_CONV_DIRECT_ASM_5X10U2V2 – ConvAsm5x10u2v2f1`, `ConvAsm5x10u2v2b1
  - MIOPEN_DEBUG_CONV_DIRECT_ASM_7X7C3H224W224 – ConvAsm7x7c3h224w224k64u2v2p3q3f1
  - MIOPEN_DEBUG_CONV_DIRECT_ASM_WRW3X3 – ConvAsmBwdWrW3x3
  - MIOPEN_DEBUG_CONV_DIRECT_ASM_WRW1X1 – ConvAsmBwdWrW1x1
  - MIOPEN_DEBUG_CONV_DIRECT_OCL_FWD11X11 – ConvOclDirectFwd11x11
  - MIOPEN_DEBUG_CONV_DIRECT_OCL_FWDGEN – ConvOclDirectFwdGen
  - MIOPEN_DEBUG_CONV_DIRECT_OCL_FWD – ConvOclDirectFwd
  - MIOPEN_DEBUG_CONV_DIRECT_OCL_FWD1X1 – ConvOclDirectFwd1x1
  - MIOPEN_DEBUG_CONV_DIRECT_OCL_WRW2 – ConvOclBwdWrW2<n> (where n = {1,2,4,8,16}) and ConvOclBwdWrW2NonTunable
  - MIOPEN_DEBUG_CONV_DIRECT_OCL_WRW53 – ConvOclBwdWrW53
  - MIOPEN_DEBUG_CONV_DIRECT_OCL_WRW1X1 – ConvOclBwdWrW1x1
- Winograd solutions:
  - MIOPEN_DEBUG_AMD_WINOGRAD_3X3 – ConvBinWinograd3x3U, FP32 Winograd Fwd/Bwd, filter size fixed to 3x3

  - MIOPEN_DEBUG_AMD_WINOGRAD_RXS – ConvBinWinogradRxS, FP32/FP16 F(3,3) Fwd/Bwd and FP32 F(3,2) WrW Winograd. Subsets include:
    - MIOPEN_DEBUG_AMD_WINOGRAD_RXS_WRW – FP32 F(3,2) WrW convolutions only
    - MIOPEN_DEBUG_AMD_WINOGRAD_RXS_FWD_BWD – FP32/FP16 F(3,3) Fwd/Bwd
  - MIOPEN_DEBUG_AMD_WINOGRAD_RXS_F3X2 – ConvBinWinogradRxSf3x2, FP32/FP16 Fwd/Bwd F(3,2) Winograd
  - MIOPEN_DEBUG_AMD_WINOGRAD_RXS_F2X3 – ConvBinWinogradRxSf2x3, FP32/FP16 Fwd/Bwd F(2,3) Winograd, serves group convolutions only
  - MIOPEN_DEBUG_AMD_WINOGRAD_RXS_F2X3_G1 – ConvBinWinogradRxSf2x3g1, FP32/FP16 Fwd/Bwd F(2,3) Winograd, for non-group convolutions
- Multi-pass Winograd:
  - MIOPEN_DEBUG_AMD_WINOGRAD_MPASS_F3X2 – ConvWinograd3x3MultipassWrW<3-2>, WrW F(3,2), stride 2 only
  - MIOPEN_DEBUG_AMD_WINOGRAD_MPASS_F3X3 – ConvWinograd3x3MultipassWrW<3-3>, WrW F(3,3), stride 2 only
  - MIOPEN_DEBUG_AMD_WINOGRAD_MPASS_F3X4 – ConvWinograd3x3MultipassWrW<3-4>, WrW F(3,4)
  - MIOPEN_DEBUG_AMD_WINOGRAD_MPASS_F3X5 – ConvWinograd3x3MultipassWrW<3-5>, WrW F(3,5)
  - MIOPEN_DEBUG_AMD_WINOGRAD_MPASS_F3X6 – ConvWinograd3x3MultipassWrW<3-6>, WrW F(3,6)
  - MIOPEN_DEBUG_AMD_WINOGRAD_MPASS_F5X3 – ConvWinograd3x3MultipassWrW<5-3>, WrW F(5,3)
  - MIOPEN_DEBUG_AMD_WINOGRAD_MPASS_F5X4 – ConvWinograd3x3MultipassWrW<5-4>, WrW F(5,4)
  - MIOPEN_DEBUG_AMD_WINOGRAD_MPASS_F7X2:
    - ConvWinograd3x3MultipassWrW<7-2>, WrW F(7,2)
    - ConvWinograd3x3MultipassWrW<7-2-1-1>, WrW F(7x1,2x1)
    - ConvWinograd3x3MultipassWrW<1-1-7-2>, WrW F(1x7,1x2)
  - MIOPEN_DEBUG_AMD_WINOGRAD_MPASS_F7X3:
    - ConvWinograd3x3MultipassWrW<7-3>, WrW F(7,3)
    - ConvWinograd3x3MultipassWrW<7-3-1-1>, WrW F(7x1,3x1)
    - ConvWinograd3x3MultipassWrW<1-1-7-3>, WrW F(1x7,1x3)
  - MIOPEN_DEBUG_AMD_MP_BD_WINOGRAD_F2X3 – ConvMPBidirectWinograd<2-3>, FWD/BWD F(2,3)
  - MIOPEN_DEBUG_AMD_MP_BD_WINOGRAD_F3X3 – ConvMPBidirectWinograd<3-3>, FWD/BWD F(3,3)
  - MIOPEN_DEBUG_AMD_MP_BD_WINOGRAD_F4X3 – ConvMPBidirectWinograd<4-3>, FWD/BWD F(4,3)
  - MIOPEN_DEBUG_AMD_MP_BD_WINOGRAD_F5X3 – ConvMPBidirectWinograd<5-3>, FWD/BWD F(5,3)
  - MIOPEN_DEBUG_AMD_MP_BD_WINOGRAD_F6X3 – ConvMPBidirectWinograd<6-3>, FWD/BWD F(6,3)
  - MIOPEN_DEBUG_AMD_MP_BD_XDLOPS_WINOGRAD_F2X3 – ConvMPBidirectWinograd_xdlops<2-3>, FWD/BWD F(2,3)
  - MIOPEN_DEBUG_AMD_MP_BD_XDLOPS_WINOGRAD_F3X3 – ConvMPBidirectWinograd_xdlops<3-3>, FWD/BWD F(3,3)
  - MIOPEN_DEBUG_AMD_MP_BD_XDLOPS_WINOGRAD_F4X3 – ConvMPBidirectWinograd_xdlops<4-3>, FWD/BWD F(4,3)

  - MIOPEN_DEBUG_AMD_MP_BD_XDLOPS_WINOGRAD_F5X3 – ConvMPBidirectWinograd_xdlops<5-3>, FWD/BWD F(5,3)
  - MIOPEN_DEBUG_AMD_MP_BD_XDLOPS_WINOGRAD_F6X3 – ConvMPBidirectWinograd_xdlops<6-3>, FWD/BWD F(6,3)
  - MIOPEN_DEBUG_AMD_MP_BD_WINOGRAD_EXPEREMENTAL_FP16_TRANSFORM – ConvMPBidirectWinograd*, FWD/BWD FP16 experimental mode (use at your own risk). Disabled by default.
  - MIOPEN_DEBUG_AMD_FUSED_WINOGRAD – Fused FP32 F(3,3) Winograd, variable filter size.

Implicit GEMM solutions:

- ASM implicit GEMM
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_ASM_FWD_V4R1 – ConvAsmImplicitGemmV4R1DynamicFwd
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_ASM_FWD_V4R1_1X1 – ConvAsmImplicitGemmV4R1DynamicFwd_1x1
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_ASM_BWD_V4R1 – ConvAsmImplicitGemmV4R1DynamicBwd
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_ASM_WRW_V4R1 – ConvAsmImplicitGemmV4R1DynamicWrw
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_ASM_FWD_GTC_XDLOPS – ConvAsmImplicitGemmGTCDynamicFwdXdlops
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_ASM_BWD_GTC_XDLOPS – ConvAsmImplicitGemmGTCDynamicBwdXdlops
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_ASM_WRW_GTC_XDLOPS – ConvAsmImplicitGemmGTCDynamicWrwXdlops
- HIP implicit GEMM
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_FWD_V4R1 – ConvHipImplicitGemmV4R1Fwd
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_FWD_V4R4 – ConvHipImplicitGemmV4R4Fwd
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_BWD_V1R1 – ConvHipImplicitGemmBwdDataV1R1
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_BWD_V4R1 – ConvHipImplicitGemmBwdDataV4R1
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_WRW_V4R1 – ConvHipImplicitGemmV4R1WrW
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_WRW_V4R4 – ConvHipImplicitGemmV4R4WrW
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_FWD_V4R4_XDLOPS – ConvHipImplicitGemmForwardV4R4Xdlops
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_FWD_V4R5_XDLOPS – ConvHipImplicitGemmForwardV4R5Xdlops
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_BWD_V1R1_XDLOPS – ConvHipImplicitGemmBwdDataV1R1Xdlops
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_BWD_V4R1_XDLOPS – ConvHipImplicitGemmBwdDataV4R1Xdlops
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_WRW_V4R4_XDLOPS – ConvHipImplicitGemmWrwV4R4Xdlops
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_FWD_V4R4_PADDED_GEMM_XDLOPS – ConvHipImplicitGemmForwardV4R4Xdlops_Padded_Gemm
  - MIOPEN_DEBUG_CONV_IMPLICIT_GEMM_HIP_WRW_V4R4_PADDED_GEMM_XDLOPS – ConvHipImplicitGemmWrwV4R4Xdlops_Padded_Gemm

## 13.2 GEMM logging and behavior

The ROCBLAS_LAYER environmental variable can be set to output GEMM information when using the rocBLAS GEMM backend:

- ROCBLAS_LAYER=: No logging (not set)

- ROCBLAS_LAYER=1: Trace logging
- ROCBLAS_LAYER=2: Bench logging
- ROCBLAS_LAYER=3: Trace and bench logging

The HIPBLASLT_LOG_LEVEL environmental variable can be set to output GEMM information when using the hipBLASLt GEMM backend:

- HIPBLASLT_LOG_LEVEL=0: Off – no logging (default)
- HIPBLASLT_LOG_LEVEL=1: Error logging
- HIPBLASLT_LOG_LEVEL=2: Trace - API calls that launch HIP kernels log their parameters and important information
- HIPBLASLT_LOG_LEVEL=3: Hints - Hints that can potentially improve the application's performance
- HIPBLASLT_LOG_LEVEL=4: Info - Provides general information about the library execution. Can contain details about heuristic status
- HIPBLASLT_LOG_LEVEL=5: API Trace - API calls log their parameters and important information

You can also set the MIOPEN_GEMM_ENFORCE_BACKEND environment variable to override the default GEMM backend, which is rocBLAS:

- MIOPEN_GEMM_ENFORCE_BACKEND=1: Use rocBLAS if enabled
- MIOPEN_GEMM_ENFORCE_BACKEND=2: Reserved
- MIOPEN_GEMM_ENFORCE_BACKEND=3: No GEMM is called
- MIOPEN_GEMM_ENFORCE_BACKEND=4: Reserved
- MIOPEN_GEMM_ENFORCE_BACKEND=5: Use hipBLASLt if enabled
- MIOPEN_GEMM_ENFORCE_BACKEND=<any other value>: Use default behavior

To disable the use of rocBLAS entirely, set the -DMIOPEN_USE_ROCBLAS=Off configuration flag during MIOpen configuration. To disable the use of hipBLASLt entirely, set the -DMIOPEN_USE_HIPBLASLT=Off configuration flag during MIOpen configuration.

For more information on how to use logging with rocBLAS, see the rocBLAS programmer guide.

## 13.3 Numerical checking

Use the MIOPEN_CHECK_NUMERICS environmental variable to debug potential numerical abnormalities. When this variable is set, MIOpen scans all inputs and outputs for each kernel called and attempts to detect infinities (infs), not-a-number (NaN), and all zeros. This environment variable has several settings to help with debugging:

- MIOPEN_CHECK_NUMERICS=0x01: Fully informative. Prints results from all checks to console.
- MIOPEN_CHECK_NUMERICS=0x02: Warning information. Prints results only if an abnormality is detected.
- MIOPEN_CHECK_NUMERICS=0x04: Throw error on detection. MIOpen runs MIOPEN_THROW upon an abnormal result.
- MIOPEN_CHECK_NUMERICS=0x08: Abort upon an abnormal result. Lets you drop into a debugging session.
- MIOPEN_CHECK_NUMERICS=0x10: Print stats. Computes and prints mean/absmean/min/max (note that this is slow).

## 13.4 Controlling parallel compilation

MIOpen's convolution `*Find()` calls compile and benchmark a set of `solvers` contained in `miopenConvAlgoPerf_t`. This is done in parallel with `miopenConvAlgorithm_t`. Parallelism per algorithm is set to 20 threads. Typically, there are far fewer threads spawned due to the limited number of kernels under any given algorithm.

You can control the level of parallelism using the `MIOPEN_COMPILE_PARALLEL_LEVEL` environment variable.

To disable multi-threaded compilation, run:

```
export MIOPEN_COMPILE_PARALLEL_LEVEL=1
```

## 13.5 Experimental controls

Using experimental controls might result in:

- Performance drops
- Computation inaccuracies
- Runtime errors
- Other kinds of unexpected behavior

We strongly recommended only using these controls at the explicit request of the library developers.

### 13.5.1 Code Object version selection (experimental)

Different ROCm versions use Code Object (CO) files from different versions (formats). The library automatically uses the most suitable version. The following variables allow for experimenting and triaging possible problems related to the CO version:

- `MIOPEN_DEBUG_AMD_ROCM_METADATA_ENFORCE`: Affects kernels written in GCN assembly language.
  - `0` (or unset): Automatically detects the required CO version and assembles the files to that version. This is the default.
  - `1`: Do not auto-detect the CO version and always assemble v2 COs.
  - `2`: Behave as if both v2 and v3 COs are supported (see `MIOPEN_DEBUG_AMD_ROCM_METADATA_PREFER_OLDER`).
  - `3`: Always assemble v3 COs.
- `MIOPEN_DEBUG_AMD_ROCM_METADATA_PREFER_OLDER`: This variable only affects assembly kernels and only applies when ROCm supports both v2 and v3 COs. By default, the newer format is used (v3 CO). When this variable is enabled, the behavior is reversed.
- `MIOPEN_DEBUG_OPENCL_ENFORCE_CODE_OBJECT_VERSION`: Enforces the CO format for OpenCL kernels. This only works with the HIP backend, when `cmake ... -DMIOPEN_BACKEND=HIP...` is used.
  - Unset - Automatically detects the required CO version. This is the default.
  - `2`: Always build to v2 CO.
  - `3`: Always build to v3 CO.
  - `4`: Always build to v4 CO.

### 13.5.2 Winograd multi-pass maximum workspace throttling

- `MIOPEN_DEBUG_AMD_WINOGRAD_MPASS_WORKSPACE_MAX`: `ConvWinograd3x3MultipassWrW`, WrW
- `MIOPEN_DEBUG_AMD_MP_BD_WINOGRAD_WORKSPACE_MAX`: `ConvMPBidirectWinograd*`, FWD BWD

Syntax of value:

- A decimal or hex (with `0x` prefix) value that must fit into a 64-bit unsigned integer
- If the syntax is invalid, then the behavior is unspecified

Usage notes:

- Sets the limit (max allowed workspace size) in bytes for multi-pass (MP) Winograd solutions.
- The value affects all MP Winograd solutions. If a solution needs more workspace than the limit, it doesn't apply.
- If the value is not set, then the default limit is used. The current default is `2000000000` (~1.862 GiB) for the gfx900 and gfx906/60 (or fewer CUs). No default limit is defined for other GPUs.
- Special values:
  - `0`: Use the default limit, as if the variable is unset
  - `1`: Completely prohibit the use of workspace
  - `-1`: Remove the default limit

CHAPTER

FOURTEEN

# USING THE FIND APIS AND IMMEDIATE MODE

MIOpen contains several convolution algorithms for each stage of training or inference. Prior to MIOpen version 2.0, you had to call find methods to generate a set of applicable algorithms.

Here's a typical workflow for the find stage:

```
miopenConvolutionForwardGetWorkSpaceSize(handle,
                                         weightTensorDesc,
                                         inputTensorDesc,
                                         convDesc,
                                         outputTensorDesc,
                                         &maxWorkSpaceSize);

// < allocate workspace >


// NOTE:
// The miopenFindConvolution*() call is expensive in terms of run time and required␣
↪workspace.
// Therefore, we highly recommend reserving the required algorithm and workspace so that␣
↪you can
// reuse them later (within the lifetime of the same MIOpen handle object).
// With this approach, there should be no need to invoke miopenFind*() more than once per
// application lifetime.

miopenFindConvolutionForwardAlgorithm(handle,
                                      inputTensorDesc,
                                      input_device_mem,
                                      weightTensorDesc,
                                      weight_device_mem,
                                      convDesc,
                                      outputTensorDesc,
                                      output_device_mem,,
                                      request_algo_count,
                                      &ret_algo_count,
                                      perf_results,
                                      workspace_device_mem,
                                      maxWorkSpaceSize,
                                      1);

// < select fastest algorithm >
```

(continues on next page)

(continued from previous page)

```
// < free previously allocated workspace and allocate workspace required for the␣
↪selected algorithm>

miopenConvolutionForward(handle, &alpha,
                         inputTensorDesc,
                         input_device_mem,
                         weightTensorDesc,
                         weight_device_mem,
                         convDesc,
                         perf_results[0].fwd_algo, // use the fastest algo
                         &beta,
                         outputTensorDesc,
                         output_device_mem,
                         workspace_device_mem,
                         perf_results[0].memory); //workspace size
```

The results of the find call are returned in an array of `miopenConvAlgoPerf_t` structures in order of performance, with the fastest at index `0`.

This call sequence is only run once per session, as it's inherently expensive. Within the sequence, `miopenFindConvolution*()` is the most expensive call. `miopenFindConvolution*()` caches its own results on disk so subsequent calls during the same MIOpen session run faster.

Internally, MIOpen's find calls compile and benchmark a set of `solvers` contained in `miopenConvAlgoPerf_t`. This is performed in parallel with `miopenConvAlgorithm_t`. You can control the level of parallelism using an environmental variable. See the debugging section on *controlling parallel compilation* for more information.

## 14.1 Immediate mode

MIOpen v2.0 introduces immediate mode, which removes the requirement for `miopenFindConvolution*()` calls, thereby reducing runtime costs. In this mode, you can query the MIOpen runtime for all of the supported solutions for a given convolution configuration. The sequence of operations for immediate mode is similar to launching regular convolutions in MIOpen, for instance, through the `miopenFindConvolution*()` API. However, in this case, the different APIs have a lower runtime cost.

A typical convolution call is similar to the following sequence:

- Construct the MIOpen handle and relevant descriptors, such as the convolution descriptor.
- With the above data structures, call `miopenConvolution*GetSolutionCount` to get the maximum number of supported solutions for the convolution descriptor.
- Use the obtained count to allocate memory for the `miopenConvSolution_t` structure (introduced in MIOpen v2.0).
- Call `miopenConvolution*GetSolution` to populate the `miopenConvSolution_t` structures allocated above. The returned list is sorted in order of best performance, where the first element is the fastest.
- While the above structure returns the amount of workspace required for an algorithm, you can query the amount of a workspace required for a known solution ID using `miopenConvolution*GetSolutionWorkspaceSize`. However, this is not a requirement because the structure returned by `miopenConvolution*GetSolution` already has this information.
- Initiate the convolution operation in `immediate` mode by calling `miopenConvolution*Immediate`. This populates the output tensor descriptor with the respective convolution result. However, the first call to `miopenConvolution*Immediate` might take more time because the kernel must be compiled if it isn't present in the kernel cache.

- Optionally, you can compile the solution of choice by calling `miopenConvolution*CompileSolution`. This ensures that the kernel represented by the chosen solution is populated in the kernel cache, removing the need to compile it.

```
miopenConvolutionForwardGetSolutionCount(handle,
                                         weightTensorDesc,
                                         inputTensorDesc,
                                         convDesc,
                                         outputTensorDesc,
                                         &solutionCount);


// < allocate an array of miopenConvSolution_t of size solutionCount >


miopenConvolutionForwardGetSolution(handle,
                                    weightTensorDesc,
                                    inputTensorDesc,
                                    convDesc,
                                    outputTensorDesc,
                                    solutionCount,
                                    &actualCount,
                                    solutions);

// < select a solution from solutions array >

miopenConvolutionForwardGetSolutionWorkspaceSize(handle,
                                                 weightTensorDesc,
                                                 inputTensorDesc,
                                                 convDesc,
                                                 outputTensorDesc,
                                                 selected->solution_id,
                                                 &ws_size);

// < allocate solution workspace of size ws_size >


// This stage is optional.
miopenConvolutionForwardCompileSolution(handle,
                                         weightTensorDesc,
                                         inputTensorDesc,
                                         convDesc,
                                         outputTensorDesc,
                                         selected->solution_id);



miopenConvolutionForwardImmediate(handle,
                                  weightTensor,
                                  weight_device_mem,
                                  inputTensorDesc,
                                  input_device_mem,
                                  convDesc,
```

(continues on next page)

(continued from previous page)

```
                                    outputTensorDesc,
                                    output_device_mem,
                                    workspace_device_mem,
                                    ws_size,
                                    selected->solution_id);
```

### 14.1.1 Immediate mode fallback

Although immediate mode is underpinned by *FindDb*, it might not contain every configuration of interest. If FindDb encounters a database miss, there are two fallback paths it can take, depending on whether the CMake variable `MIOPEN_ENABLE_AI_IMMED_MODE_FALLBACK` is set to `ON` or `OFF`.

If you require the best possible performance, run the find stage at least once.

#### 14.1.1.1 AI-based heuristic fallback (default)

If `MIOPEN_ENABLE_AI_IMMED_MODE_FALLBACK` is set to `ON` (the default), the immediate mode behavior upon encountering a database miss is to use an AI-based heuristic to pick the optimal solution.

It first checks the applicability of the AI-based heuristic for the given configuration. If the heuristic is applicable, it feeds various parameters of the given configuration into a neural network that has been tuned to predict the optimal solution with 90% accuracy.

#### 14.1.1.2 Weighted throughput index-based fallback

When `MIOPEN_ENABLE_AI_IMMED_MODE_FALLBACK` is set to `OFF` or the AI heuristic is not applicable for the given convolution configuration, immediate mode uses a weighted throughput index-based mechanism when encountering a database miss. This mechanism estimates which solution would be optimal based on the convolution configuration parameters.

### 14.1.2 Limitations of immediate mode

System FindDb has only been populated for these architectures:

- gfx906 with 64 CUs
- gfx906 with 60 CUs
- gfx900 with 64 CUs
- gfx900 with 56 CUs

If your architecture isn't listed, you must run the find API on your system (once per application) to take advantage of immediate mode's more efficient behavior.

### 14.1.3 Backend limitations

OpenCL support for immediate mode via the fallback is limited to `FP32` datatypes. This is because the current release's fallback path uses GEMM, which is serviced through MIOpenGEMM (on OpenCL). MIOpenGEMM only contains support for `FP32`.

The HIP backend uses rocBLAS as its fallback path, which contains a more robust set of data types.

## 14.2 Find modes

MIOpen provides a set of find modes that are used to accelerate find API calls. Set the different modes by using the `MIOPEN_FIND_MODE` environment variable with one of these values:

- `NORMAL/1` (normal find): This is the full find mode call, which benchmarks all the solvers and returns a list.
- `FAST/2` (fast find): Checks *FindDb* for an entry. If there's a FindDb hit, it uses that entry. If there's a miss, it uses the immediate mode fallback. This mode offers fast start-up times at the cost of GPU performance.
- `HYBRID/3` or unset `MIOPEN_FIND_MODE` (hybrid find): Checks *FindDb* for an entry. If there's a FindDb hit, it uses that entry. If there's a miss, it uses the existing find machinery. This mode offers slower start-up times than fast find without the GPU performance drop.
- `4`: This value is reserved and should not be used.
- `DYNAMIC_HYBRID/5` (dynamic hybrid find): Checks *FindDb* for an entry. If there's a FindDb hit, it uses that entry. If there's a miss, it uses the existing find machinery, skipping non-dynamic kernels. It offers faster start-up times than hybrid find, but GPU performance might decrease.

The default find mode is `DYNAMIC_HYBRID`. To run the full `NORMAL` find mode, use `export MIOPEN_FIND_MODE=NORMAL` or `export MIOPEN_FIND_MODE=1`.

CHAPTER

# FIFTEEN

# API REFERENCE LIBRARY

The MIOpen API library is structured as follows:

- *Datatypes*
- Modules:
  - Handle
  - Tensor
  - Activation
  - Convolution
  - Recurrent neural networks (RNN)
  - Batch normalization
  - Local response normalization (LRN)
  - Pooling
  - Softmax
  - Fusion
  - Loss function
  - Dropout
  - Reduction
  - Find
  - Layernorm (experimental)
  - Sum (experimental)
  - GroupNorm (experimental)
  - Cat (experimental)
  - SGD (experimental)
  - ReduceExtreme (experimental)
  - Getitem (experimental)
  - ReduceCalculation (experimental)
  - RotaryPositionalEmbeddings (experimental)
  - ReLU (experimental)

  - Kthvalue (experimental)
  - GLU (experimental)

## 15.1 Modules

## 15.2 Datatypes

MIOpen contains several datatypes at different levels of support. The enumerated datatypes are:

```
typedef enum {
    miopenHalf     = 0,
    miopenFloat    = 1,
    miopenInt32    = 2,
    miopenInt8     = 3,
    /* Value 4 is reserved. */
    miopenBFloat16 = 5,
    miopenDouble   = 6,
    miopenFloat8   = 7,
    miopenBFloat8  = 8
} miopenDataType_t;
```

Of these types only `miopenFloat` and `miopenHalf` are fully supported across all layers in MIOpen. Refer to the individual *Modules* in the API library for specific datatype support and limitations.

Type descriptions:

- `miopenHalf`: 16-bit floating point
- `miopenFloat`: 32-bit floating point
- `miopenInt32`: 32-bit integer, used primarily for `int8` convolution outputs
- `miopenInt8`: 8-bit integer; supported by `int8` convolution forward path, tensor set, tensor copy, tensor cast, tensor transform, tensor transpose, and im2col
- `miopenBFloat16`: brain float fp-16 (8-bit exponent, 7-bit fraction); supported by convolutions, tensor set, and tensor copy
- `miopenDouble`: 64-bit floating point; supported by reduction, layerNorm, and batchNorm
- `miopenFloat8`: 8-bit floating point (layout 1.4.3, exponent bias 7); supported by convolutions
- `miopenBFloat8`: 8-bit floating point (layout 1.5.2, exponent bias 15); supported by convolutions

In addition to these standard datatypes, pooling also contains its own indexing datatypes:

```
typedef enum {
    miopenIndexUint8  = 0,
    miopenIndexUint16 = 1,
    miopenIndexUint32 = 2,
    miopenIndexUint64 = 3,
} miopenIndexType_t;
```

CHAPTER

SIXTEEN

# LICENSE



---

The following files

- src/include/miopen/kernel_cache.hpp
- src/kernel_cache.cpp

are licensed using the MIT license described at the top of this file in addition to an Apache-2.0 license using the following text:



---

driver/mloSoftmaxHost.hpp is available under a BSD-2-Clause license

src/include/miopen/mlo_internal.hpp is licensed using the MIT described above and a BSD-2-Clause license

Both files use the following license text for their BSD license text:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

---

The file src/md5.cpp is derived from a public domain implementation. The original license text is as follows:

Author: Alexander Peslyak, better known as Solar Designer

This software was written by Alexander Peslyak in 2001. No copyright is claimed, and the software is hereby placed in the public domain. In case this attempt to disclaim copyright and place the software in the public domain is deemed null and void, then the software is Copyright © 2001 Alexander Peslyak and it is hereby released to the general public under the following terms:

Redistribution and use in source and binary forms, with or without modification, are permitted.

There's ABSOLUTELY NO WARRANTY, express or implied.

(This is a heavily cut-down "BSD license".)

This differs from Colin Plumb's older public domain implementation in that no exactly 32-bit integer data type is required (any 32-bit or wider unsigned integer data type will do), there's no compile-time endianness configuration, and the function prototypes match OpenSSL's. No code from Colin Plumb's implementation has been reused; this comment merely compares the properties of the two independent implementations.

The primary goals of this implementation are portability and ease of use. It is meant to be fast, but not as fast as possible. Some known optimizations are not included to reduce source code size and avoid compile-time configuration.