MIOpen Documentation

Release 3.3.0

Advanced Micro Devices, Inc.

Dec 03, 2024

CONTENTS

Welcome to our documentation home page. To learn more about MIOpen, refer to *What is MIOpen?*

Our documentation is structured as follows:

Install

- *Install MIOpen*
- *Build MIOpen for embedded systems*
- *Build MIOpen using Docker*

Reference

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

Conceptual

- *MI200 alternate implementation*
- *Cache*
- *Find database*
- *Performance database*
- *Porting to MIOpen*

How to

- *Use fusion*
- *Log & debug*
- *Use the find APIs & immediate mode*

To contribute to the documentation refer to Contributing to ROCm.

You can find licensing information for all ROCm components on the ROCm licensing page.

CHAPTER

ONE

WHAT IS MIOPEN?

MIOpen is AMD's 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 datatype for convolutions, which allows for efficient training at lower precision without loss of accuracy.

CHAPTER

TWO

INSTALLING MIOPEN

To install MIOpen, you must first install these prerequisites:

- 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)–this is now deprecated
- ROCm CMake: provides CMake modules for common build tasks needed for the ROCm software stack
- Half: IEEE 754-based, half-precision floating-point library
- Boost: Version 1.79 is recommended, as older versions may need patches to work on newer systems
  - MIOpen uses `boost-system` and `boost-filesystem` packages to enable persistent *kernel cache*
- SQLite3: A reading and writing performance database
- lbzip2: A multi-threaded compress or decompress utility
- rocBLAS: AMD's library for Basic Linear Algebra Subprograms (BLAS) on the ROCm platform.
  - Minimum version branch for pre-ROCm 3.5 master-rocm-2.10
  - Minimum version branch for post-ROCm 3.5 master-rocm-3.5
- Multi-Level Intermediate Representation (MLIR) with its MIOpen dialect to support and complement kernel development
- Composable Kernel: A C++ templated device library for GEMM-like and reduction-like operators.

2.1 Installing with pre-built packages

You can install MIOpen on Ubuntu using `apt-get install miopen-hip`.

If using OpenCL, you can use `apt-get install miopen-opencl` (but this is not recommended, as OpenCL is deprecated).

Note that you can't install both backends on the same system simultaneously. If you want a different backend other than what currently exists, completely uninstall the existing backend prior to installing the new backend.

2.1.1 Installing with a kernels package

MIOpen provides an optional pre-compiled kernels package to reduce startup latency. These precompiled kernels comprise a select set of popular input configurations. We'll expand these kernels in future releases to include additional coverage.

Note that all compiled kernels are locally cached in the `$HOME/.cache/miopen/` folder, so precompiled kernels reduce the startup latency only for the first run of a neural network. Precompiled kernels don't reduce 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 (e.g., `gfx900`, `gfx906`, `gfx1030`).

> **Note**
>
> Not installing these packages doesn't impact the functioning of MIOpen, since MIOpen compiles them on the target machine once you run the kernel. However, the compilation step may significantly increase the startup time for different 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. You can invoke it using:

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

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

2.1.2 Installing dependencies

You can install dependencies using the `install_deps.cmake` script (`cmake -P install_deps.cmake`).

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

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

An example CMake step is:

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

You can use this prefix to specify the dependency path during the configuration phase using `CMAKE_PREFIX_PATH`.

MIOpen's HIP backend uses rocBLAS by default. You can install 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. You can install hipBLASLt's minimum release using `apt-get install hipblaslt`. In addition to needing hipblaslt, you will also need to install hipBLAS. You can install hipBLAS's minimum release using `apt-get install hipblas`. To disable hipBLASLt, set the configuration flag `-DMIOPEN_USE_HIPBLASLT=Off`. hipBLASLt is **not** available with OpenCL.

2.2 Building MIOpen from source

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

2.2.1 HIP backend

First, create a build directory:

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

Next, configure CMake. You can set the backend using the `-DMIOPEN_BACKEND` CMake variable.

Set the C++ compiler to `clang++`. For the HIP backend (ROCm 3.5 and later), run:

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

An example CMake step 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 (~) shorthand to represent the home directory.

2.2.2 OpenCL backend

Note

OpenCL is deprecated. We recommend using a HIP backend and following the instructions listed in the preceding section.

First, run:

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

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 dependency path for an environment in ROCm 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" ..
```

2.2.3 Setting up locations

By default, the install location is set to `/opt/rocm`. You can change this using `CMAKE_INSTALL_PREFIX`:

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

2.2.4 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/`. For development purposes, setting *BUILD_DEV* changes the default path to both database files to the source directory:

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

Database paths can be explicitly customized using the `MIOPEN_SYSTEM_DB_PATH` (System PerfDb) and `MIOPEN_USER_DB_PATH` (User PerfDb) CMake variables.

To learn more, refer to the *performance database* documentation.

2.2.5 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. You can change the location of the cache directory during configuration using the `-DMIOPEN_CACHE_DIR=<cache-directory-path>` flag.

You can also disable the cache during runtime using the `MIOPEN_DISABLE_CACHE=1` environmental variable.

2.2.5.1 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 (e.g., `rm -rf ~/.cache/miopen/<miopen-version-number>`). You can find more information in the *cache* documentation.

2.2.5.2 For MIOpen version 2.4 and later

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

2.2.6 Changing the CMake configuration

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

`ccmake ..` **or** `cmake-gui`: `cmake-gui ..`

The `ccmake` program can be downloaded as a Linux package (`cmake-curses-gui`), but is not available on Windows.

2.3 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 installs the library to the `CMAKE_INSTALL_PREFIX` path that you specified.

2.4 Building the driver

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

You can build the driver using the `MIOpenDriver` target:

`cmake --build . --config Release --target MIOpenDriver` **or** `make MIOpenDriver`

2.5 Running the tests

You can run tests using the 'check' target:

`cmake --build . --config Release --target check` **or** `make check`

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

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

2.6 Formatting the code

All the code is formatted using *clang-format*. To format a file, use:

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

To format the code per commit, you can install githooks:

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

2.7 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. In MIOpen, we use Gi LFS to store our large files, such as our kernel database files (`*.kdb`) that are normally > 0.5 GB.

You can install Git LFS using the following code:

```
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 has already been tracked, you can skip this step):

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

You can pull all or a single large file using:

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

Update the large files and push to GitHub using:

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

2.8 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. If required, you can build with dynamically linked Boost libraries using 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

THREE

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, we recommended using the prebuilt Docker image.

- Downloading a prebuilt image

  You can find prebuilt Docker images at ROCm Docker Hub.

- Building your own image

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

  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
  ```

  Once in the Docker environment, use `git clone MIOpen`. You can now start building MIOpen using CMake.

3.1 Building MIOpen from source

Use the following instructions to build MIOpen from source.

1. Create a build directory:

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

2. Configure CMake using either an MIOpen or a HIP backend.

   MIOpen backend:

   Set your preferred backend using the `-DMIOPEN_BACKEND` CMake variable.

   HIP backend (ROCm 3.5 and later):

   First, set the C++ compiler to `clang++`:

```
export CXX=<location-of-clang++-compiler>
```

Then, run the following command to configure CMake:

```
cmake -DMIOPEN_BACKEND=HIP -DCMAKE_PREFIX_PATH="<hip-
↪installed-path>;<rocm-installed-path>;<miopen-dependency-
↪path>" ..
```

For example, you can set CMake to:

```
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 `CMAKE_PREFIX_PATH`, **do not** use the tilde (~) shorthand for the home directory.

3.1.1 Choosing an install location

By default, the install location is set to `/opt/rocm`. If you used a different install location, set your install path using `CMAKE_INSTALL_PREFIX`:

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

CHAPTER

FOUR

BUILD MIOPEN FOR EMBEDDED SYSTEMS

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

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

2. Create the build directory.

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

3. Configure for an embedded build.

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

```
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 while using embedded builds, add `-DMIOPEN_USE_HIP_KERNELS=On` to configure 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. This allows you to use the best tuned kernels for your network. Embedding FindDb requires a semicolon-separated list of architecture CU pairs to embed on-disk databases in the binary (e.g., `gfx906_60;gfx900_56`).

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

   This configures the build directory for embedding (FindDb and PerfDb).

5. Embed the precompiled kernels package.

   To prevent the loss of performance due to compile-time overhead, an MIOpen build can embed the precompiled kernels package. This package contains convolution kernels of known inputs, and allows you to avoid compiling kernels during runtime.

   - Embed the precompiled package using a package install.

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

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

If you choose not to install the precompiled kernel package, there is no impact to the functioning of MIOpen because MIOpen compiles these kernels on the target machine once the kernel is run. However, the compilation step may significantly increase the startup time for different operations.

The `utils/install_precompiled_kernels.sh` script automates the above process. It queries your machine for the GPU architecture and then installs the appropriate package. You can invoke it using:

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

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

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

- Using the URL to a kernels binary. You can use the `MIOPEN_BINCACHE_PATH` flag with a URL that contains the binary.

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

Precompiled kernels packages are installed in `/opt/rocm/miopen/share/miopen/db`.

Here's an example with 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 ..
```

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

Here's an example with 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:

```
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:

```
make -j
```

CHAPTER

FIVE

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)

5.1 Modules

5.2 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

SIX

USING THE FIND DATABASE

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.

As of MIOpen 2.0, we introduced an *immediate mode*, which is based on a database that contains the results of calls to the legacy `Find()` stage. We refer to the find database as FindDb.

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.

User FindDb *always takes precedence* over System FindDb.

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

Note that:

- The System FindDb is *not* modified upon installation of MIOpen.
- There are separate Find databases for 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 *problem configuration*, which is implicitly defined by the tensor descriptors and convolution descriptor passed to API function.

6.2 Updating MIOpen and User FindDb

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

If you want to re-collect the information into the new User FindDb, you can use the same steps you followed in the previous version. Re-collecting information keeps immediate mode optimized.

6.3 Disabling FindDb

You can disable FindDb by setting 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 and may 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 (e.g., `$HOME/.cache/miopen`). We recommend that you 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:

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

7.3 Updating MIOpen and removing the cache

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 (e.g., `rm -rf $HOME/.cache/miopen/<miopen-version-number>`).

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

7.4 Installing pre-compiled kernels

GPU architecture-specific, pre-compiled kernel packages are available in the ROCm package repositories. These reduce the startup latency of MIOpen kernels (they contain the kernel cache file and install it in the ROCm installation directory along with other MIOpen artifacts). When launching a kernel, MIOpen 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, the kernel is compiled and placed in your kernel cache.

These packages are optional and must be separately installed from MIOpen. If you want to conserve disk space, you may choose not to install these packages (though without them, you'll have 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 doens'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. Optimal values depend on many factors, including 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 means 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.

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 may take a long time, but once 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 MIOpen's install location, while User PerfDb resides in your home directory. See *setting up locations* for more information.

System PerfDb is not modified during 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 have tuning parameters
- The value of the `exhaustiveSearch` parameter is `true`
- Neither System nor User PerfDb can contain values for the relevant *problem configuration*.

You can override the latter two conditions by enforcing 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 MIOpen find modes, see *Find modes*.

8.1.1 Using MIOPEN_FIND_ENFORCE

`MIOPEN_FIND_ENFORCE` supports symbolic (case-insensitive) and numeric values. 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 processes.
- `SEARCH/(3)`: Perform auto-tune even if not requested via API. In this case, the library behaves as if the `exhaustiveSearch` parameter 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 means for getting the best performance from MIOpen. When in this mode, your application's first run may 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 recommended only 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. **Use this option with care.**

8.2 Updating MIOpen and User PerfDb

If you install a new version of MIOpen, we strongly recommend moving or deleting 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 is located at the User PerfDb path.

CHAPTER

NINE

MI200 MATRIX FUSED MULTIPLY-ADD (MFMA) BEHAVIOR SPECIFICS

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 BFloat16 datatype's larger exponent range, albeit with reduced accuracy. The following salients 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 behaviors by using the `miopenSetConvolutionAttribute` API call: Pass the convolution descriptor for the appropriate convolution operation, and the `MIOPEN_CONVOLUTION_ATTRIB_FP16_ALT_IMPL` convolution attribute (with a non-zero value), to engage the alternate implementation.
- 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. Keep in mind that the environment variable impacts the convolution operation in all directions.

CHAPTER

TEN

PORTING TO MIOPEN

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

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

MIOpen supports:

- 4D tensors in the NCHW and NHWC storage format; the cuDNN __"*Nd*"__ APIs don't have a corresponding MIOpen API
- __`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 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

10.1 cuDNN versus MIOpen APIs

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

10.1.1 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)`

10.1.2 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 ``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

10.1.3 Filter operations

cuDNN	MIOpen
`cudnnStatus_t cudnnCreateFilterDescriptor( cudnnFilterDescriptor_t *filterDesc)`	All `FilterDescriptor` APIs are substituted by their respective `TensorDescriptor` API.

10.1.4 Convolution operations

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

10.1.5 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)`

10.1.6 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,

10.1.7 Activation operations

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

10.1.8 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,`

10.1.9 Batch normalization operations

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

CHAPTER

ELEVEN

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 our GitHub repository example folder. The code examples in this document are from the 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, refer to the *Supported fusions* section. For simplicity, the example doesn't populate the tensors with meaningful data and shows only basic code without any error checking.

Once 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.

11.1 Creating a fusion plan

A **fusion plan** is the data structure that holds all the metadata regarding fusion intent, and 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, only this is required for the fusion plan (i.e. the input tensor descriptor isn't required for the individual operations).

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

Where: `* fusePlanDesc` is an object of type `miopenFusionPlanDescriptor_t * input.desc` is the `miopenTensorDescriptor_t` object

11.2 Creating and adding operators

Operators represent the different operations that you want fused. 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.

We plan 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.

Once 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.

In our example, we add the convolution, bias, and activation operations to our 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 the this descriptor, refer to 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 that you verify the returned codes to make sure the operations (and their order) is supported. The operator insertion can fail for a number of reasons, such as 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.

11.3 Compiling the fusion plan

Following the operator addition, 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;
}
```

In order to compile the fusion plan, you must have acquired 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 for each handle separately.

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

11.4 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.

To begin, create the `miopenOperatorArgs_t` object using:

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

Once created, you can set the runtime arguments for each operation. In our 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 given 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 our example code, we set the arguments for the operations 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);
```

The 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, we recommend only compiling 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 our example, this is demonstrated in `main.cpp`, lines 77 through 85.

11.5 Running a fusion plan

Once you've compiled the fusion plan and set arguments set 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.

11.6 Cleanup

Once the application is done with the fusion plan, you can detroy 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, and you don't need to worry about additional cleanup.

11.7 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.

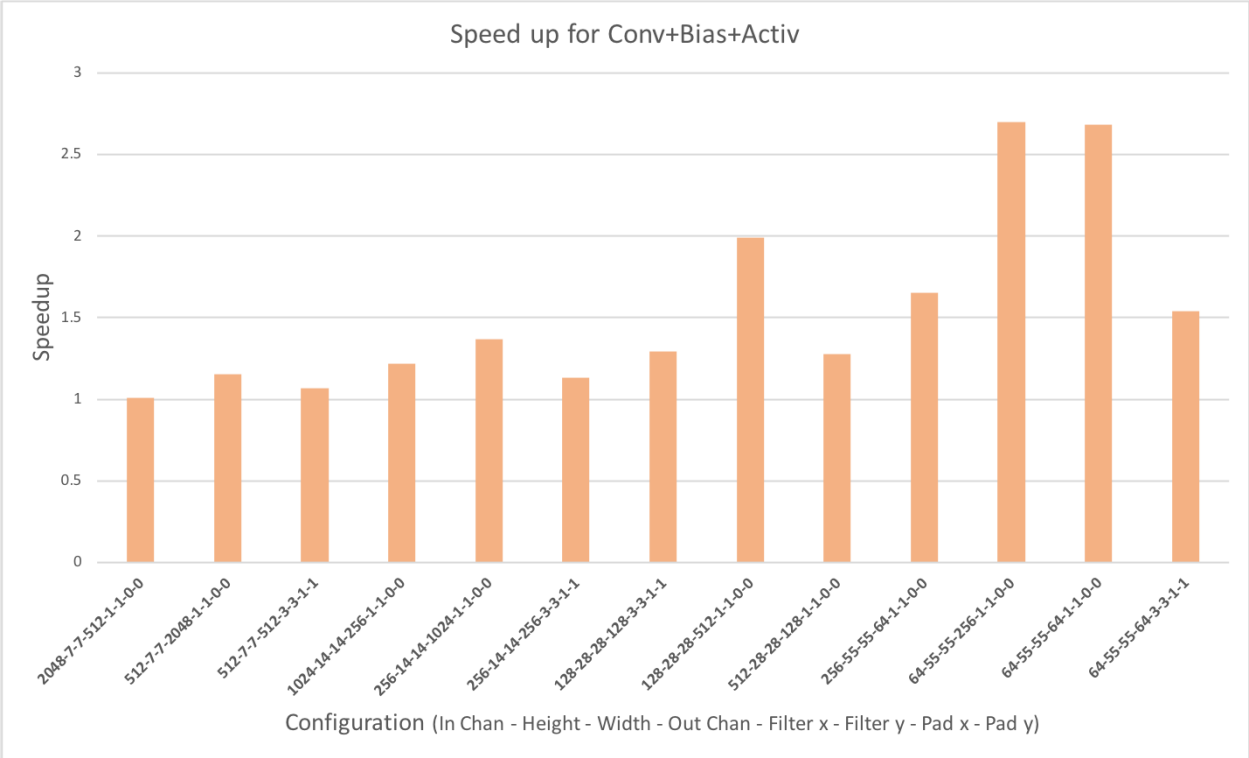
Batch Normalization based fusion for FP32 and FP16 for Inference and Training

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

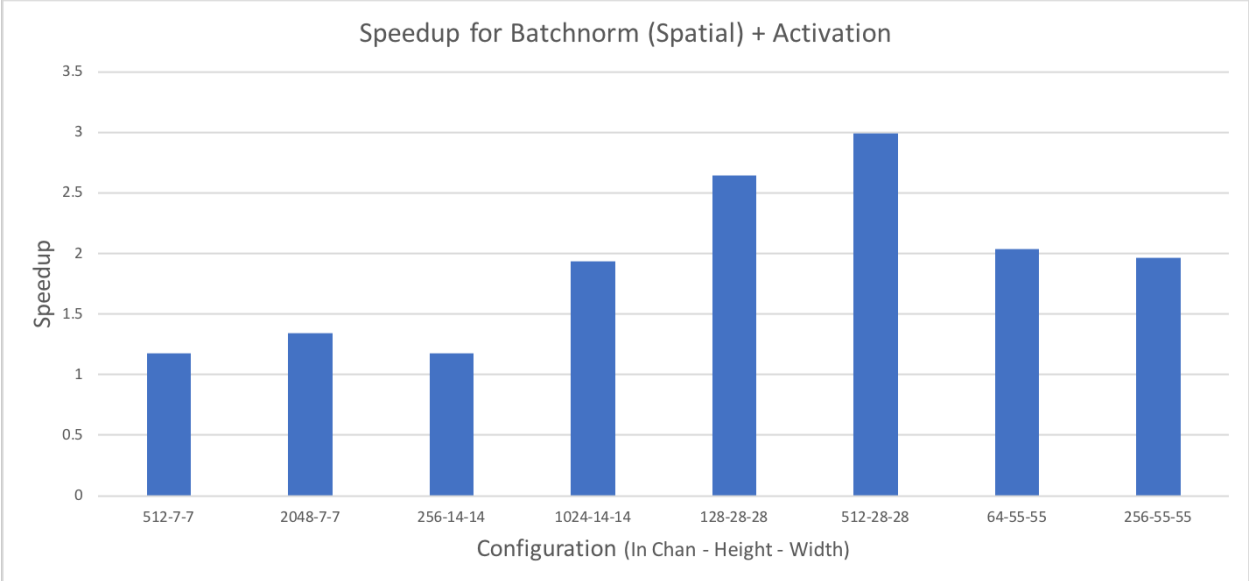
*N mode is either spatial, or per activation.

11.8 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 (spatial mode) with activation:



CHAPTER

TWELVE

LOGGING & 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`: Print the basic layer-by-layer MIOpen API call information with actual parameters (configurations). This information is important for debugging.
  - To enable feature: `1`, `on`, `yes`, `true`, `enable`, `enabled`
  - To disable feature: `0`, `off`, `no`, `false`, `disable`, `disabled`
- `MIOPEN_ENABLE_LOGGING_CMD`: Output the associated `MIOpenDriver` command lines into the console.
  - To enable feature: `1`, `on`, `yes`, `true`, `enable`, `enabled`
  - To disable feature: `0`, `off`, `no`, `false`, `disable`, `disabled`
- `MIOPEN_LOG_LEVEL`: In addition to API call information and driver commands, MIOpen prints information related to the progress of its internal operations. This information can be useful for debugging and understanding the library's principles of operation. *MIOPEN_LOG_LEVEL* controls the verbosity of these messages. 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 (not used yet).
  - `3`: Errors and fatals.
  - `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`: When enabled, each log line is prefixed with information you can use to identify records printed from different processes or threads. This is useful for debugging multi-process/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.

Tip

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

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

12.1 Layer filtering

The following sections contain environment variables that you can use to enable or disable various kinds 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 kernel/algorithm: `1`, `yes`, `true`, `enable`, `enabled`
- To disable kernel/algorithm: `0`, `no`, `false`, `disable`, `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 there. Therefore, the subsequent immediate mode `*Get()` calls may return incomplete information or run into fallback path.

In order to rehabilitate immediate mode, you can:

- Re-enable all solvers and re-run the same `*Find()` calls you previously ran
- Completely remove the User FindDb

If no variable is set, MIOpen behaves as if the variable is set to `enabled`. This means that kernels and algorithms are enabled by default.

12.1.1 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 can disable them all.

- `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.

12.1.2 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.

12.1.3 Filtering out all but one solution

- `MIOPEN_DEBUG_FIND_ONLY_SOLVER=solution_id`: Directly affects only `*Find()` calls. However, there is an indirect connection to immediate mode (per the previous warning).
  - `solution_id` must be a numeric or a string identifier of some solution.
  - If `solution_id` denotes some applicable solution, then only that solution is found (in addition to GEMM and FFT, if applicable–refer to the following note).
  - If `solution_id` is valid, but not applicable, then `*Find()` fails with all algorithms (except for GEMM and FFT,–refer to the following note).
  - Otherwise, the `solution_id` is invalid (i.e., 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.

12.1.4 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:
    - `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

12.2 GEMM logging and behavior

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

- ROCBLAS_LAYER=: Not set–there is no logging
- 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 hipBLASLt GEMM backend:

- HIPBLASLT_LOG_LEVEL=0: Off – there is 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, may 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 (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 using rocBlas entirely, set the *-DMIOPEN_USE_ROCBLAS=Off* configuration flag during MIOpen configuration. To disable using hipBLASLt entirely, set the *-DMIOPEN_USE_HIPBLASLT=Off* configuration flag during MIOpen configuration.

You can find more information on logging with rocBLAS in the rocBLAS programmer guide.

12.3 Numerical checking

You can use the MIOPEN_CHECK_NUMERICS environmental variable to debug potential numerical abnormalities. Setting this variable scans all inputs and outputs of each kernel called and attempts to detect infinities (infs), not-a-number (NaN), and all zeros. This environment variable has several settings that 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 abnormal result.
- MIOPEN_CHECK_NUMERICS=0x08: Abort upon abnormal result. Allows you to drop into a debugging session.
- MIOPEN_CHECK_NUMERICS=0x10: Print stats. Computes and prints mean/absmean/min/max (note that this is slow).

12.4 Controlling parallel compilation

MIOpen's convolution *Find() calls compile and benchmark a set of solvers contained in miopenConvAlgoPerf_t. This is done in parallel per 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
```

12.5 Experimental controls

Using experimental controls may 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.

12.5.1 Code Object version selection (experimental)

Different ROCm versions use Code Object (CO) files from different versions (i.e., formats). The library automatically uses the most suitable version. The following variables allow for experimenting and triaging possible problems related to 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 to that version. This is the default.
  - 1: Do not auto-detect CO version; 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 when ROCm supports both v2 and v3 COs (like ROCm 2.10). 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 CO format for OpenCL kernels. This only works with the HIP backend (cmake ... -DMIOPEN_BACKEND=HIP...).
  - Unset - Automatically detect 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.

12.5.2 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:

- Decimal or hex (with 0x prefix) value that must fit into a 64-bit unsigned integer

- If the syntax is violated, then the behavior is unspecified

Semantics:

- Sets the limit (max allowed workspace size) for multi-pass (MP) Winograd solutions, in bytes.
- Affects all MP Winograd solutions. If a solution needs more workspace than the limit, it doesn't apply.
- If unset, then the default limit is used. The current default is `2000000000` (~1.862 GiB) for gfx900 and gfx906/60 (or less CUs). No default limit is set 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

THIRTEEN

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 in order 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 *find* are returned in an array of `miopenConvAlgoPerf_t` structs 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 the 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. Refer to the debugging section, *controlling parallel compilation* for more information.

13.1 Immediate mode

MIOpen v2.0 introduces immediate more, 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 (i.e., through the use of 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:

- You construct the MIOpen handle and relevant descriptors, such as the convolution descriptor.
- With the above data structures, you call `miopenConvolution*GetSolutionCount` to get the maximum number of supported solutions for the convolution descriptor.
- The obtained count is used to allocate memory for the `miopenConvSolution_t` structure, introduced in MIOpen v2.0.
- You call `miopenConvolution*GetSolution` to populate the `miopenConvSolution_t` structures allocated above. The returned list is in the 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 inquire 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).
- Now you can 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` may consume more time because if the kernel isn't present in the kernel cache, it would need to be compiled.

- 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 the kernel in question.

```
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);
```

13.1.1 Immediate mode fallback

Although immediate mode is underpinned by *FindDb*, it may not contain every configuration of interest. If FindDb encounters a database miss, it has 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.

13.1.1.1 AI-based heuristic fallback (default)

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

First, the applicability of the AI-based heuristic for the given configuration is checked. 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.

13.1.1.2 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, the immediate mode behavior upon encountering a database miss is to use a weighted throughput index-based mechanism to estimate which solution would be optimal (based on the convolution configuration parameters).

13.1.2 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) in order to take advantage of immediate mode's more efficient behavior.

13.1.3 Backend limitations

OpenCL support for immediate mode via the fallback is limited to FP32 datatypes. This is because the current release's fallback path goes through 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 datatypes.

13.2 Find modes

MIOpen provides a set of find modes that are used to accelerate find API calls. The different modes are set 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. 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. 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 may 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

FOURTEEN

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.