rocWMMA Documentation

Release 2.2.1

Advanced Micro Devices, Inc.

May 11, 2026

CONTENTS

rocWMMA is a C++ header library for accelerating mixed-precision matrix multiply-accumulate operations. It leverages specialized GPU matrix cores on the latest AMD discrete GPUs. For more information, see *What is rocWMMA?*

The rocWMMA public repository is located at https://github.com/ROCm/rocm-libraries/tree/develop/projects/rocwmma.

Note: The rocWMMA repository for ROCm 7.1.1 and earlier is located at https://github.com/ROCm/rocWMMA.

Install

- *Installation guide*

Conceptual

- *Programming guide*
- *Migration guide*

Examples

- Samples

API reference

- *API reference guide*

To contribute to the documentation, see Contributing to ROCm.

You can find licensing information on the Licensing page.

CHAPTER

ONE

WHAT IS ROCWMMA?

rocWMMA is a C++ header library for accelerating mixed-precision matrix multiply-accumulate operations that leverage specialized GPU matrix cores on the latest AMD discrete GPUs. “roc” indicates rocWMMA is an AMD-specific component belonging to the ROCm ecosystem, while WMMA stands for Wavefront Mixed precision Multiply Accumulate.

rocWMMA leverages modern C++ techniques, is templated for modularity, and uses meta-programming paradigms to provide opportunities for customization and compile-time inferences and optimizations. The API is seamless across the supported CDNA and RDNA architectures. The API also offers a high degree of compatibility with common CUDA WMMA library interfaces, simplifying the migration of existing code to the AMD platform.

The API is implemented as GPU device code which empowers you to directly use GPU matrix cores right from your kernel code. Major benefits include kernel-level control, which allows authoring flexibility and accessibility, and compiler optimization passes in-situ with other device code. You can decide when and where kernel run-time launches are required, which is not dictated by the API.

The rocWMMA API facilitates the decomposition of matrix multiply-accumulate problems into discretized blocks (also known as fragments) and enables parallelization of block-wise operations across multiple GPU wavefronts. Programmers only have to manage the wavefront handling of fragments, while individual threads are handled internally. This can lead to faster development times and a more seamless experience across multiple architectures. API functions include data loading and storing, matrix multiply-accumulate, and helper transforms that operate on data fragment abstractions. Data movement between global and local memory can be handled cooperatively amongst the wavefronts in a thread block to enable data sharing and re-use. Matrix multiply-accumulate functionality supports mixed-precision inputs and outputs with native fixed-precision accumulation.

Supporting code is required for GPU device management and kernel invocation. The kernel code samples and tests provided are built and launched using HIP as part of the ROCm ecosystem.

CHAPTER

TWO

INSTALLING AND BUILDING ROCWMMA

This topic provides instructions for installing and configuring the rocWMMA library. The quickest way to install rocWMMA is to use the prebuilt packages that are released with ROCm. Alternatively, there are instructions to build the component from the source code.

The available ROCm packages are:

- rocwmma-dev (source files for development)
- rocwmma-samples (sample executables)
- rocwmma-samples-dbgsym (sample executables with debug symbols)
- rocwmma-tests (test executables)
- rocwmma-tests-dbgsym (test executables with debug symbols)
- rocwmma-clients (samples, tests, and benchmarks executables)

2.1 Prerequisites

rocWMMA requires the following prerequisites:

- A ROCm-enabled platform, using ROCm version 6.4 or later. For more information, see ROCm installation.
- rocBLAS version 4.3.0 for ROCm version 6.3 or later. (Optional: only required if rocWMMA is configured to use rocBLAS for validation. See below for more information.)

2.2 Installing prebuilt packages

To install rocWMMA on Ubuntu or Debian, use these commands:

```
sudo apt-get update
sudo apt-get install rocwmma-dev rocwmma-samples rocwmma-tests
```

To install rocWMMA on RHEL-based platforms, use:

```
sudo yum update
sudo yum install rocwmma-dev rocwmma-samples rocwmma-tests
```

To install rocWMMA on SLES, use:

```
sudo dnf upgrade
sudo dnf install rocwmma-dev rocwmma-samples rocwmma-tests
```

After rocWMMA is installed, it can be used just like any other library with a C++ API.

Note: The prebuilt package client executables support the gfx908, gfx90a, gfx942, gfx950, gfx1100, gfx1101, gfx1102, gfx1200, and gfx1201 targets.

After rocWMMA is installed, you can find the `rocwmma.hpp` header file in the `/opt/rocm/include/rocwmma` directory. To make calls into rocWMMA, ensure you only include the `rocwmma.hpp`, `rocwmma_coop.hpp`, and `rocwmma_transforms.hpp` files in the user code. Don't directly include any other rocWMMA files from `/opt/rocm/include/internal`.

2.3 Building and installing rocWMMA from source

It isn't necessary to build rocWMMA from source because it's ready to use after installing the prebuilt packages, as described above. To build rocWMMA from source, follow the instructions in this section.

2.3.1 System requirements

8GB of system memory is required for a full rocWMMA build. This value might be lower if rocWMMA is built without tests.

2.3.2 GPU support

rocWMMA is supported on the following GPUs.

- AMD CDNA class GPUs featuring matrix core support, including the gfx908, gfx90a, gfx942, and gfx950 GPUs (collectively labeled as gfx9).

  Note: Double precision `FP64` data type support requires the gfx90a, gfx942, or gfx950.

  `F8` and `BF8` data type support requires the gfx942 or gfx950.

- AMD RDNA class GPUs featuring AI acceleration support, including the gfx1100, gfx1101, and gfx1102 (collectively known as gfx11), and the gfx1200 and gfx1201 (collectively known as gfx12).

  Note: `F8` and `BF8` data type support requires the gfx1200 or gfx1201.

2.3.3 Dependencies

rocWMMA is designed to have minimal external dependencies so it's lightweight and portable. The following dependencies are required:

- ROCm (Version 6.4 or later)
- CMake (Version 3.14 or later)
- rocm-cmake (Version 0.8.0 or later)
- HIP runtime (Version 6.4.0 or later) (Or the ROCm hip-runtime-amd package)
- LLVM OpenMP runtime dev package (Version 10.0 or later) (Also available as the ROCm rocm-llvm-dev package)
- (Optional, only required to use rocBLAS for validation) rocBLAS (Version 4.3.0 for ROCm 6.3 or later) (Or the ROCm rocblas and rocblas-dev packages).

Note: It's best to use ROCm packages from the same release where applicable.

2.3.4 Downloading rocWMMA

The rocWMMA source code is available from the rocWMMA GitHub. ROCm version 6.4 or later is required.

Note: The rocWMMA repository for ROCm 7.1.1 and earlier is located at https://github.com/ROCm/rocWMMA.

To verify the ROCm version installed on an Ubuntu system, use this command:

```
apt show rocm-libs -a
```

On a RHEL-based system, use:

```
yum info rocm-libs
```

The ROCm version has major, minor, and patch fields, possibly followed by a build-specific identifier. For example, a ROCm version of `6.0.0.40000-23` corresponds to major release = `6`, minor release = `0`, patch = `0`, and build identifier `40000-23`. The rocWMMA GitHub repository has branches with names like `rocm-major.minor.x`, where `major` and `minor` are the same as for the ROCm version. To download rocWMMA on ROCm version `x.y`, use this command:

ROCm 7.2.0 and later

```
git clone -b release/rocm-rel-x.y https://github.com/ROCm/rocm-libraries.git
cd projects/rocwmma
```

Alternatively, you can use `sparse-checkout` to clone only the rocWMMA project from the `rocm-libraries` monorepo. For more information, see Contributing to the ROCm Libraries.

ROCm 7.1.1 and prior

```
git clone -b release/rocm-rel-x.y https://github.com/ROCm/rocWMMA.git
cd rocWMMA
```

Replace `x.y` in the above command with the version of ROCm installed on your machine. For example, if you have ROCm 6.0 installed, then replace `release/rocm-rel-x.y` with `release/rocm-rel-6.0`.

2.3.5 Build configuration

You can choose to build any of the following combinations:

- The rocWMMA library only
- The library and samples
- The library and tests (validation and benchmarks)
- The library, samples, tests, and (optionally) assembly

rocWMMA is a header library, so you only need the header includes to call rocWMMA from your code. The client contains the tests, samples, and benchmark code.

Here are the available options to build the rocWMMA library, with or without clients.

Option	Description	Default value
GPU_TARGETS	Build code for specific GPU target(s)	gfx908; gfx90a; gfx942; gfx950; gfx1100; gfx1101; gfx1102; gfx1200; gfx1201
ROCWMMA_BUILD_TES	Build the tests	ON
ROCWMMA_BUILD_SAM	Build the samples	ON
ROCWMMA_BUILD_ASS	Generate assembly files	OFF
ROCWMMA_BUILD_VAL	Build validation tests	ON (requires ROCWMMA_BUILD_TESTS=ON)
ROCWMMA_BUILD_REG	Build regression tests	ON (requires ROCWMMA_BUILD_TESTS=ON)
ROCWMMA_BUILD_BEN	Build benchmark tests	OFF (requires ROCWMMA_BUILD_TESTS=ON)
ROCWMMA_BUILD_EXT	Build extended testing coverage	OFF (requires ROCWMMA_BUILD_TESTS=ON)
ROCWMMA_VALIDATE_	Use rocBLAS for validation tests	ON (requires ROCWMMA_BUILD_VALIDATION_TESTS=ON)
ROCWMMA_BENCHMARK	Include rocBLAS benchmarking data	OFF (requires ROCWMMA_BUILD_BENCHMARK_TESTS=ON)
ROCWMMA_USE_SYSTE	Use the system GoogleTest library instead of downloading and building it	OFF (requires ROCWMMA_BUILD_TESTS=ON)

Building the library alone

By default, the project is configured in Release mode.

To build the library alone, run this command:

```
CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B <build_dir> . -DROCWMMA_
↪BUILD_TESTS=OFF -DROCWMMA_BUILD_SAMPLES=OFF
```

Here are some other example project configurations:

Configuration	Command
Basic	`CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B <build_dir>`
Targeting gfx908	`CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B <build_dir> . -DGPU_TARGETS=gfx908:xnack-`
Debug build	`CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B <build_dir> . -DCMAKE_BUILD_TYPE=Debug`
Build without rocBLAS (default on)	`CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B <build_dir> . -DROCWMMA_VALIDATE_WITH_ROCBLAS=OFF -DROCWMMA_BENCHMARK_WITH_ROCBLAS=OFF`

After configuration, build the library using this command:

```
cmake --build <build_dir> -- -j<nproc>
```

Note: It's recommended to use a minimum of 16 threads to build rocWMMA with any tests, for example, using `-j16`.

Building the library and samples

To build the library and samples, run the following command:

```
CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B <build_dir> . -DROCWMMA_
↪BUILD_TESTS=OFF -DROCWMMA_BUILD_SAMPLES=ON
```

After configuration, build using this command:

```
cmake --build <build_dir> -- -j<nproc>
```

The samples folder in `<build_dir>` contains the executables in the table below.

Executable name	Description
simple_sgemm	A simple GEMM operation [D = alpha * (A x B) + beta * C] using the rocWMMA API for single-precision floating point types
simple_dgemm	A simple GEMM operation [D = alpha * (A x B) + beta * C] using the rocWMMA API for double-precision floating point types
simple_hgemm	A simple GEMM operation [D = alpha * (A x B) + beta * C] using the rocWMMA API for half-precision floating point types
perf_sgemm	An optimized GEMM operation [D = alpha * (A x B) + beta * C] using the rocWMMA API for single-precision floating point types
perf_dgemm	An optimized GEMM operation [D = alpha * (A x B) + beta * C] using the rocWMMA API for double-precision floating point types
perf_hgemm	An optimized GEMM operation [D = alpha * (A x B) + beta * C] using the rocWMMA API for half-precision floating point types
simple_sgemv	A simple GEMV operation [y = alpha * (A) * x + beta * y] using the rocWMMA API for single-precision floating point types
simple_dgemv	A simple GEMV operation [y = alpha * (A) * x + beta * y] using the rocWMMA API for double-precision floating point types
simple-dlrm	A simple DLRM operation using the rocWMMA API
hipRTC_gemm	A simple GEMM operation [D = alpha * (A x B) + beta * C] demonstrating runtime compilation (hipRTC) compatibility

Building the library and tests

rocWMMA provides the following test suites:

- **DLRM tests**: Cover the dot product interactions between embeddings used in the Deep Learning Recommendation Model (DLRM) implemented with rocWMMA.
- **GEMM tests**: Cover block-wise Generalized Matrix Multiplication (GEMM) implemented with rocWMMA.
- **Unit tests**: Cover various aspects of the rocWMMA API and internal functionality.

rocWMMA can build both validation and benchmark tests. Validation tests verify the rocWMMA implementations against a reference model, providing a `PASS` or `FAIL` result. Benchmark tests invoke the tests multiple times. They return the average compute throughput in teraflops/sec (TFLOPS/sec) and, in some cases, gauge the efficiency as a percentage of the expected peak performance. The library uses CPU or rocBLAS methods for validation, when available and benchmarks comparisons based on the selected project configurations provided. By default, the project is linked with rocBLAS to validate results more efficiently.

To build the library and tests, run this command:

```
CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B <build_dir> . -DROCWMMA_
↪BUILD_TESTS=ON
```

After configuration, build using this command:

```
cmake --build <build_dir> -- -j<nproc>
```

The tests in `<build_dir>` contain executables, as shown in the table below.

Executable name	Description
dlrm/dlrm_dot_test-*	A DLRM implementation using the rocWMMA API
dlrm/dlrm_dot_lds_test-*	A DLRM implementation using the rocWMMA API with LDS shared memory
gemm/gemm_PGR0_LB0_MP0_SB_N(	A simple GEMM operation [D = alpha * (A x B) + beta * C] using the rocWMMA API
gemm/gemm_PGR0_LB0_MP0_MB_N(	A modified GEMM operation where each wave targets a sub-grid of output blocks using the rocWMMA API
gemm/gemm_PGR1_LB2_MP0_MB_CI	A modified GEMM operation where each wave targets a sub-grid of output blocks using LDS memory, the rocWMMA API, and block-level collaboration
gemm/gemm_PGR1_LB2_MP0_MB_CI	A modified GEMM operation where each wave targets a sub-grid of output blocks using LDS memory, the rocWMMA API, and wave-level collaboration
gemm/gemm_PGR1_LB2_MP0_MB_CI	A modified GEMM operation where each wave targets a sub-grid of output blocks using LDS memory, the rocWMMA API, and workgroup-level collaboration
gemm/gemm_PGR0_LB0_MP0_SB_N(	An adhoc version of gemm_PGR0_LB0_MP0_SB_NC-*
gemm/gemm_PGR0_LB0_MP0_MB_N(	An adhoc version of gemm_PGR0_LB0_MP0_MB_NC-*
gemm/gemm_PGR1_LB2_MP0_MB_CI	An adhoc version of gemm_PGR1_LB2_MP0_MB_CP_BLK-*
gemm/gemm_PGR1_LB2_MP0_MB_CI	An adhoc version of gemm_PGR1_LB2_MP0_MB_CP_WV-*
gemm/gemm_PGR1_LB2_MP0_MB_CI	An adhoc version of gemm_PGR1_LB2_MP0_MB_CP_WG-*
unit/contamination_test	Tests against contamination of pristine data for loads and stores
unit/cross_lane_ops_test	Tests the cross-lane vector operations
unit/fill_fragment_test	Tests the fill_fragment API function
unit/io_shape_test	Tests input and output shape meta data
unit/io_traits_test	Tests input and output logistical meta data
unit/layout_test	Tests the accuracy of internal matrix layout patterns
unit/layout_traits_test	Tests the accuracy of internal matrix layout traits
unit/load_store_matrix_sync_	Tests the load_matrix_sync and store_matrix_sync API functions
unit/load_store_matrix_coop_	Tests the load_matrix_coop_sync and store_matrix_coop_sync API functions
unit/map_util_test	Tests the mapping utilities used in rocWMMA implementations
unit/pack_util_test	Tests the vector packing utilities used in rocWMMA implementations
unit/transforms_test	Tests the transform utilities used in rocWMMA implementations
unit/tuple_test	Tests the additional transform utilities used in rocWMMA implementations
unit/unpack_util_test	Tests the vector unpacking utilities used in rocWMMA implementations
unit/vector_iterator_test	Tests the internal vector storage iteration implementation
unit/vector_test	Tests the internal vector storage implementation
unit/vector_util_test	Tests the internal vector manipulation utilities implementation

Note: *= validate: Executables that compare outputs for correctness against reference sources such as CPU or rocBLAS calculations.

*= bench: Executables that measure kernel execution speeds, which might be compared against the rocBLAS references.

Build the library, tests, and assembly

To build the library and tests with assembly code generation, run the following command:

```
CC=/opt/rocm/bin/amdclang CXX=/opt/rocm/bin/amdclang++ cmake -B <build_dir> . -DROCWMMA_
↪BUILD_ASSEMBLY=ON -DROCWMMA_BUILD_TESTS=ON
```

After configuration, build using this command:

```
cmake --build <build_dir> -- -j<nproc>
```

Note: The `assembly` folder within the `<build_dir>` contains a hierarchy of assembly files generated by the executables in the format `test_executable_name.s`. These files can be viewed in a text editor.

Make targets list

When building rocWMMA during the `make` step, you can specify the Make targets instead of defaulting to `make all`. The following table highlights the relationships between high-level grouped targets and individual targets.

Group target	Individual targets
rocwmma_samples	simple_sgemm
	simple_dgemm
	simple_hgemm
	perf_sgemm
	perf_dgemm
	perf_hgemm
	simple_sgemv
	simple_dgemv
	simple_dlrm
	hipRTC_gemm
rocwmma_gemm_tests_va	gemm_PGR0_LB0_MP0_SB_NC-validate
	gemm_PGR0_LB0_MP0_SB_NC_ad_hoc-validate
	gemm_PGR0_LB0_MP0_MB_NC-validate
	gemm_PGR0_LB0_MP0_MB_NC_ad_hoc-validate
	gemm_PGR1_LB2_MP0_MB_CP_BLK-validate
	gemm_PGR1_LB2_MP0_MB_CP_WV-validate
	gemm_PGR1_LB2_MP0_MB_CP_WG-validate
	gemm_PGR1_LB2_MP0_MB_CP_ad_hoc-validate
rocwmma_gemm_tests_be	gemm_PGR0_LB0_MP0_SB_NC-bench
	gemm_PGR0_LB0_MP0_SB_NC_ad_hoc-bench
	gemm_PGR0_LB0_MP0_MB_NC-bench
	gemm_PGR0_LB0_MP0_MB_NC_ad_hoc-bench
	gemm_PGR1_LB2_MP0_MB_CP_BLK-bench
	gemm_PGR1_LB2_MP0_MB_CP_WV-bench

continues on next page

Table 1 – continued from previous page

Group target	Individual targets
	gemm_PGR1_LB2_MP0_MB_CP_WG-bench
	gemm_PGR1_LB2_MP0_MB_CP_ad_hoc-bench
rocwmma_dlrm_tests_va	dlrm_dot_test-validate
	dlrm_dot_lds_test-validate
rocwmma_dlrm_tests_be	dlrm_dot_test-bench
	dlrm_dot_lds_test-bench
rocwmma_unit_tests	contamination_test
	layout_test
	layout_traits_test
	map_util_test
	load_store_matrix_sync_test
	load_store_matrix_coop_sync_test
	fill_fragment_test
	vector_iterator_test
	vector_test
	vector_util_test
	pack_util_test
	io_traits_test
	cross_lane_ops_test
	io_shape_test
	tuple_test
	transforms_test
	unpack_util_test

2.3.6 Build performance

Depending on the resources available to the build machine and the selected build configuration, rocWMMA build times can take an hour or more. Here are some things you can do to reduce build times:

- Target a specific GPU, for instance, with `-D GPU_TARGETS=gfx908:xnack-`.
- Use a large number of threads, for instance, `-j32`.
- Select `ROCWMMA_BUILD_ASSEMBLY=OFF`.
- Select `ROCWMMA_BUILD_DOCS=OFF`.
- Select `ROCWMMA_BUILD_EXTENDED_TESTS=OFF`.
- Specify one of `ROCWMMA_BUILD_VALIDATION_TESTS` or `ROCWMMA_BUILD_BENCHMARK_TESTS` as `ON` and the other as `OFF` instead of building both.
- During the `make` command, build a specific target, for instance, `rocwmma_gemm_tests`.

2.3.7 Test runtimes

Depending on the resources available to the machine running the selected tests, rocWMMA test runtimes can last an hour or more. Here are some things you can do to reduce test runtimes:

- CTest runs the entire test suite, but you can invoke tests individually by name.
- Use GoogleTest filters to target specific test cases:

```
<test_exe> --gtest_filter=\*name_filter\*
```

- Use ad hoc tests to focus on a specific set of parameters.
- Manually adjust the test case coverage.

2.3.8 Test verbosity and output redirection

GEMM tests support logging arguments to control verbosity and output redirection.

```
<test_exe> --output_stream "output.csv" --omit 1
```

Compact	Verbose	Description
-os <output_file>.csv	--output_stream <output_file>.csv	redirect GEMM testing output to CSV file
	--omit <code>	code = 1: Omit gtest SKIPPED tests
		code = 2: Omit gtest FAILED tests
		code = 4: Omit gtest PASSED tests
		code = 8: Omit all gtest output
		code = <N>: OR'd combination of 1, 2, 4

CHAPTER

THREE

PROGRAMMING GUIDE

This document outlines the library design choices, source code organization and helpful information for new development.

3.1 Infrastructure

- Doxygen and Sphinx are used to generate the project's documentation.
- Jenkins is used to automate Continuous Integration (CI) testing, with configurations stored in the `.jenkins` folder.
- rocWMMA is hosted and maintained by AMD on GitHub.

> **Note:** The rocWMMA repository for ROCm 7.1.1 and earlier is located at https://github.com/ROCm/rocWMMA.

- The rocWMMA project is organized and configured using `CMake`, with `CMakeLists.txt` files in the root of each directory.
- `clang-format` is used to format C++ code. `.githooks/install` ensures that a clang-format pass will run on each committed file.
- `GTest` is used to implement test suite organization and execution.
- `CTest` is used to consolidate and invoke multiple test targets. The `<rocWMMA_install_dir>/bin/rocWMMA/CTestTestfile.cmake` file lists the testing targets executed when `ctest` is invoked.
- The preferred compiler for rocWMMA is `CC=<path_to_rocm>/bin/amdclang` and `CXX=<path_to_rocm>/bin/amdclang++`. `hipcc` is also supported, however may be deprecated in future ROCm releases.

3.2 hipRTC support

The HIP runtime compilation (hipRTC) environment enables on-the-fly runtime compilation, loading, and execution of device code on AMD GPUs. The rocWMMA library is compatible with hipRTC, so it can be used for runtime-generated kernels. A simple GEMM sample is included to demonstrate compatibility.

For more information, refer to the HIP API Reference

3.3 Design concepts

rocWMMA is a header-only library written in C++17 and contained in the `rocwmma` namespace. It leverages template meta-programming to optimize code at compile time and generate efficient GPU kernels. rocWMMA offers full implementation visibility, allowing developers to integrate rocWMMA API calls directly into their own kernels. The integrated code is more visible to compiler optimizations for generating efficient device code. The API also avoids expensive host-device transfers or external kernel invokations.

The programming model best suited for the rocWMMA API is wavefront-centric. Data loading, storing, and MMA functions are assumed to involve the entire wavefront (or warp). Undefined behaviour is expected if not all threads in each wavefront are active while using rocWMMA. Small block sizes representing edge cases will be automatically padded and will not affect thread masking.

The data of collaborative fragments is distributed across participating waves in the same thread block. Collaborative fragments optimize collective data movement between memory locations, such as data movement from global memory to LDS, to balance shared data responsibilities across wavefronts. However, collaborative fragments are not supported in MMA functions. Fragment instances express wavefront collaboration with fragment scheduler meta-data which is specified by the developer.

In general, larger fragment sizes are better able to maximize memory bandwidth when transferring data between memory spaces, as well as ordering MMA functions. Beginning with rocWMMA 2.0.0 for ROCm 7.0, the mma_sync API can handle partial or large tiles in a piece-wise manner automatically. This enhanced functionality is demonstrated in the performance samples, which feature simplified workflow with larger tile sizes and good performance.

The rocWMMA API reduces the boiler-plate code by providing tools to decompose matrix multiply-accumulate based problems into blocks, or fragments of data that individual wavefronts can efficiently process. Wavefronts abstract blocks of data into `fragments`, which encapsulate meta-data properties about the blocks in different contexts and the data itself, including:

- a general geometric shape, for example, the BlockM/N/K dimensions.
- a context of the provenance of the data, for example, `matrix_a` or `matrix_b`.
- a context of how the data is stored, for example, row-major or column-major.
- a data type, for example, single or half-precision floating point.
- a fragment scheduler, for example, a class that specifies thread block collaboration, the number of participating waves, and their execution order.

These basic traits determine storage requirements and how rocWMMA organizes individual threads in a layout to fetch or store data. `Fragments` are powerful objects because they are versatile in configuring and representing data. Their meta-properties propagate easily via Argument Dependent Lookup (ADL) and other deduction techniques. In practice, users configure fragments for their use case, and rocWMMA handles the underlying details.

The implementation code is encapsulated into layered objects, fulfilling specific interface communications from low level functions to high-level API interactions:

- **Unit backend operations:** Act as wrappers around device-specific functions, such as intrinsics typically prefixed with `amdgcn_*`. These functions translate inputs into raw vector forms and addresses required by the low-level intrinsics, and handle architecture or device-specific behavior.
- **Vector operations:** This level of objects such as `OpaqueLoad` or `OpaqueStore` handle variable-sized vector inputs and marshall them into unrolled unit backend operations. They encompass thread-wise details of vector operations. These classes provide a consistent functional interface for input vectors of all sizes, independent of device architecture, whose details are handled at a lower level.
- **Fragment operations:** At the API level, user data is stored and managed in `fragment` objects. Fragment objects can be visualized as geometric blocks of data from the perspective of a wavefront and stored as vectors. rocWMMA's load, store, and MMA operations operate at the wavefront level, assuming that all threads in the

wavefront are participating under the hood. This layer's implementation translates wavefront fragment operations into vector operations. rocWMMA's layered design enables seamless API experience across diverse device architectures and platforms.

3.4 Nomenclature

3.4.1 GEMM

Generalized Matrix-Matrix multiplication (GEMM) is a fundamental application for rocWMMA, solving the equation `D = alpha * A x B + beta * C` , where `A`, `B`, `C`, and `D` are matrices, and `alpha` and `beta` are scalars. Matrices are sized by `M x N x K`, where `A = M x K`, `B = K x N` and `C, D = M x N`. rocWMMA includes a range of test and sample kernels covering various parameters. Test kernels are grouped into executables named using parameter strings that describe their specific implementations.

```
PGR# - Prefetch Global Read lookup stages. PGR0 = no global read prefetch. PGR1 = 1␣
↪stage global read prefetch.
LB# - LDS buffer count. LB0 = no LDS usage, LB2 = 2 LDS buffers used for swap.
MP# - MFMA instruction priority. MP0 = default MFMA instruction priority of 0. MP1 =␣
↪raise MFMA instruction priority to 1.
MB - Multiple output blocks targeted per wave
SB - Single output block target per wave
NC - Non-Cooperative load / store
CP - Cooperative load / store
BLK - Cooperative load / store per block tile
WV - Cooperative load / store per wave tile
WG - Cooperative load / store per macro tile
```

- `gemm_PGR0_LB0_MP0_SB_NC`: The simplest blocked GEMM example, which targets one output block of matrix multiplication per wave. No prefetch, no LDS usage, default MFMA prioritization, single block output and non-collaborative.
- `gemm_PGR0_LB0_MP0_MB_NC`: Implements a multi-block GEMM where each wave is responsible for a BlocksX x BlocksY grid of output blocks. No prefetch, no LDS usage, default MFMA prioritization, multiple blocks output, and non-collaborative.
- `gemm_PGR1_LB2_MP0_MB_CP_BLK`: Implements a multi-block GEMM where each wave is responsible for a BlocksX x BlocksY grid of output blocks. This kernel leverages shared memory to implement a data prefetching pipeline and collaborates with other waves to improve performance. Implements single stage prefetch, double LDS buffer, default MFMA prioritization, multiple blocks output, and is block-tile collaborative in global read and local write.
- `gemm_PGR1_LB2_MP0_MB_CP_WV`: Implements a multi-block GEMM where each wave is responsible for a BlocksX x BlocksY grid of output blocks. This kernel leverages shared memory to implement a data prefetching pipeline and collaborates with other waves to improve performance. Implements single stage prefetch, double LDS buffer, default MFMA prioritization, multiple blocks output, and is wave-tile collaborative in global read and local write.
- `gemm_PGR1_LB2_MP0_MB_CP_WG`: Implements a multi-block GEMM where each wave is responsible for a BlocksX x BlocksY grid of output blocks. This kernel leverages shared memory to implement a data prefetching pipeline and collaborates with other waves to improve performance. Implements single stage prefetch, double LDS buffer, default MFMA prioritization, multiple blocks output and is macro-tile collaborative in global read and local write.
- `Ad Hoc Test`: An executable targeting a specific set of kernel parameters. This is used as a quick mock-up for investigating a particular GEMM kernel scenario.

Validation tests are postfixed with `-validate`. Benchmark tests are postfixed with `-bench`.

Sample kernels are built with minimal infrastructure as possible and use more approachable names to appeal to a broader audience.

- `simple_sgemm`: a simple GEMM kernel with `s` denoting single-precision floating-point data type.
- `simple_dgemm`: a simple GEMM kernel with `d` denoting double-precision floating-point data type.
- `simple_hgemm`: a simple GEMM kernel with `h` denoting half-precision floating-point data type.
- `perf_sgemm`: a performant GEMM kernel with `s` denoting single-precision floating-point data type.
- `perf_dgemm`: a performant GEMM kernel with `d` denoting double-precision floating-point data type.
- `perf_hgemm`: a performant GEMM kernel with `h` denoting half-precision floating-point data type.

3.4.2 Community Samples

rocWMMA provides a `samples/community/` directory for community-contributed samples that demonstrate advanced techniques, specialized use cases, or experimental approaches extending beyond the core competency demonstrations in the official samples directory.

Community samples have reduced review requirements compared to official samples and may not be maintained with the same rigor. They are provided as-is and may not work on all GPU architectures or with all data types. Users are encouraged to adapt these examples for their specific use cases.

The community samples directory is currently available for contributions. As samples accumulate, they may be organized into subdirectories by technique or domain (e.g., fusion/, ml-models/, optimizations/, advanced/, experimental/).

To build community samples, configure with `-DROCWMMA_BUILD_COMMUNITY_SAMPLES=ON`.

3.4.3 GEMV

Generalized Matrix-Vector multiplication (GEMV) is another application for rocWMMA that solves the equation `y = alpha * A * x + beta * y`, where `A` is a matrix, `x and y` are vectors and `alpha and beta` are scalars. `Matrix A` is sized as `M x K`, vector `X` is `K x 1`, and vector `Y` is `M x 1`.

rocWMMA implements the following simple GEMV demonstration samples:

- `simple_sgemv`: Simple GEMV kernel with `s` denoting single-precision floating-point data type.
- `simple_dgemv`: Simple GEMV kernel with `d` denoting double-precision floating-point data type.

3.4.4 DLRM

rocWMMA implements a simple component of Deep Learning Recommendation Model (DLRM) for machine learning. Both forward and backward passes using half-precision inputs and outputs are demonstrated.

- `simple_dlrm`: Simple GEMV kernel with `s` denoting single-precision floating-point data type.

3.5 Library source code organization

The rocWMMA code consists of four major parts:

- The `library` directory contains the header library API and its implementation.
- The `samples` directory contains real-world sample use-cases using the rocWMMA API.
- The `test` directory contains testing infrastructure for rocWMMA.
- The `docs` directory contains sources for documentation generation.

3.5.1 `library` directory

The `library` directory is structured as follows:

- `library/include/rocwmma/`: C++ include files for the rocWMMA API. These files also contain Doxygen content that documents the API.
  - `rocwmma.hpp`: The main API for rocWMMA, defining fragment data abstractions, wave-wise storing, loading, matrix multiply-accumulate (mma) and thread block synchronization. This API offers function signatures highly compatible with common CUDA WMMA interfaces.
  - `rocwmma_transforms.hpp`: A complimentary API for rocWMMA, defining functionality to manipulate fragment data, for example, transpose and data layout changes. These are unique to rocWMMA.
- `library/include/internal`: Internal include files which define the main infrastructure driving the rocWMMA API:
  - Configuration of platforms and architectures.
  - Type support.
  - Input and output configuration, shapes and traits.
  - Loading and storing utilities.
  - Layouts of memory and registers.
  - Mapping utilities.
  - Intrinsic wrappers and hardware abstraction.
  - Vector class implementations.
  - Vector conversion, permutation, and transform utilities.
  - Vector packing and unpacking.
  - Matrix multiply-accumulate.
  - Cooperative loading and storing.
  - Thread block synchronization and flow control.
  - Utility code.
  - Data layout transformation utilities.

3.5.2 `samples` directory

The `samples` directory contains the sample codes for the following use cases:

- `samples/hipRTC_gemm.cpp`: Simple General Matrix Multiply (GEMM) algorithm demonstration without LDS memory usage or transposition, running within the hipRTC environment.
- `samples/simple_sgemv.cpp`: Simple matrix multiply-accumulate with a vector demonstration without LDS or transposition for single-precision floating-point types.
- `samples/simple_dgemv.cpp`: Simple matrix multiply-accumulate with a vector demonstration without LDS or transposition for double-precision floating-point types.
- `samples/simple_sgemm.cpp`: Simple GEMM algorithm demonstration without LDS memory usage or transposition for single-precision floating-point types.
- `samples/simple_dgemm.cpp`: Simple GEMM algorithm demonstration without LDS memory usage or transposition for double-precision floating-point types.
- `samples/simple_hgemm.cpp`: Simple GEMM algorithm demonstration without LDS memory usage or transposition for half-precision floating-point types.
- `samples/perf_sgemm.cpp`: High performing multi-block GEMM algorithm demonstration with LDS memory, macro tile collaboration, data reuse and optimized pipeline for single-precision floating-point types.
- `samples/perf_dgemm.cpp`: High performing multi-block GEMM algorithm demonstration with LDS memory, macro tile collaboration, data reuse and optimized pipeline for double-precision floating-point types.
- `samples/perf_hgemm.cpp`: High performant multi-block GEMM algorithm demonstration with LDS memory, macro tile collaboration, data reuse and optimized pipeline for half-precision floating-point types.
- `samples/simple_dlrm.cpp`: Simple Deep Learning Recommendation Model (DLRM) for machine learning.
- `samples/common.hpp`: Common code used by all the above rocWMMA sample files.
- `samples/community/`: Community-contributed samples demonstrating advanced techniques. See the Community Samples section above for details.

3.5.3 `test` directory

The `test` directory contains the following test code support:

- `test/bin`: To generate benchmark plots from the `gtest` output dumps of rocWMMA's benchmark tests.
- `test/device`: Device utility kernels to support test setup and validation on GPU.
- `test/dlrm`: For various strategies of DLRM application. This test is used to validate DLRM functions using rocWMMA API.
- `test/gemm`: For various strategies of GEMM application. This test is used to validate and benchmark GEMM functions using rocWMMA API.
- `test/unit`: For testing the basic functional units of rocWMMA library.

3.5.4 docs directory

- Sphinx and Doxygen are used to generate the project's documentation.
- `api-reference-guide.rst` pulls from Doxygen documentation to format the API documentation.
- `installation.rst` builds installation and build instructions for rocWMMA.
- `license.rst` includes information about rocWMMA licensing.
- `programmers-guide.rst` includes information about project organization and expectations.
- `what-is-rocwmma.rst` includes a description of rocWMMA.

3.5.5 Contributing

To contribute to the project, see Contributing to rocWMMA.

CHAPTER

FOUR

MIGRATION GUIDE FOR ROCWMMA 2.0

This document outlines the key API changes and new features introduced in rocWMMA 2.0, with examples to help you migrate from earlier versions.

Starting with version 2.0, rocWMMA introduces significant changes to its API, including:

- Removal of the cooperative API
- Transforms API no longer requires wave count template parameters
- rocWMMA fragments now have a fragment scheduler template argument
- rocWMMA fragments now support partial fragment sizes

4.1 Cooperative API changes

Previous releases began deprecating cooperative API functions such as those defined in `rocwmma/rocwmma_coop.hpp`:

```
template <uint32_t WaveCount, typename MatrixT, uint32_t BlockM, uint32_t BlockN, uint32_
↪t BlockK, typename DataT, typename DataLayoutT>
ROCWMMA_DEVICE void load_matrix_coop_sync(fragment<MatrixT, BlockM, BlockN, BlockK,␣
↪DataT, DataLayoutT>& frag,
                                          const DataT*                                ␣
↪                 data,
                                          uint32_t                                    ␣
↪                 ldm,
                                          uint32_t                                    ␣
↪                 waveIndex);

template <uint32_t WaveCount, typename MatrixT, uint32_t BlockM, uint32_t BlockN, uint32_
↪t BlockK, typename DataT, typename DataLayoutT>
ROCWMMA_DEVICE void store_matrix_coop_sync(DataT*                                     ␣
↪                       data,
                                          fragment<MatrixT, BlockM, BlockN, BlockK,␣
↪DataT, DataLayoutT> const& frag,
                                          uint32_t                                    ␣
↪                       ldm,
                                          uint32_t                                    ␣
↪                       waveIndex);
```

These functions previously required `WaveCount` as a template parameter and passed `waveIndex` as an argument to the API calls. This information was used to distribute data responsibilities across participating waves, aiming to balance

and optimize data transactions within a thread block. Cooperation between wavefronts in a thread block requires the use of a separate cooperative API, along with propagation of wave count and wave index values.

Example of deprecated cooperative API:

```
// Global read (macro tile)
using GRBuffA = fragment<matrix_a, MACRO_TILE_X, ROCWMMA_N, ROCWMMA_K, InputT,␣
↪DataLayoutA>;

// Local warp coordinate relative to current thread block (wg).
constexpr auto warpDims        = make_coord2d(WARPS_X, WARPS_Y);
auto           localWarpCoord  = make_coord2d(threadIdx.x / WARP_SIZE, threadIdx.y);

// WorkItems will be split up by minimum IOCount to perform either global read or local␣
↪write.
// These are inputs to cooperative functions.
constexpr auto warpCount = get<0>(warpDims) * get<1>(warpDims);

// Scheduling warp order is analogous to row major priority.
// E.g. Wg = (128, 2) = 2x2 warps
// (0, 0)   (0, 1)   Share Schedule: w0 = (0, 0), w1 = (0, 1),
// (1, 0)   (1, 1)                   w2 = (1, 0), w3 = (1, 1), count = 4
const auto warpIndex = get<0>(localWarpCoord) * get<1>(warpDims) + get<1>
↪(localWarpCoord);

// Transfer data from global memory to local memory
GRBuffA grBuffA;
load_matrix_coop_sync<warpCount>(grBuffA, gAddrA, lda, warpIndex);
store_matrix_coop_sync<warpCount>(ldsAddr, applyDataLayout<DataLayoutLds, warpCount>
↪(applyTranspose(grBuffA)), ldsld, warpIndex);
```

Calculating the warp count and warp index requires extra boilerplate code. It is important to supply the same warp count and warp index values to matching pairs of load, store, and transform APIs. Providing mismatched values to APIs that depend on matching warp count and index poses a risk of incorrect behavior. Embedding the warp count and index into the fragment object helps mitigate the risk.

As a result, fragments in rocWMMA 2.0 are augmented with an additional fragment scheduler template parameter. Fragment schedulers are classes that represent thread block scheduling models. These models provide static values for both the wave count and wave order (wave index). Fragment schedulers are classified as either non-cooperative (the default, where waves act independently) or cooperative (where waves collaborate within a thread block). Their names reflect their ordering scheme.

Example:

```
namespace fragment_scheduler
{
   //! @struct default
   //! @brief The default fragment scheduler; each wave operates independently.
   using default_schedule = IOScheduler::Default;

   //! @struct coop_row_major_2d
   //! @brief  A cooperative scheduling strategy where each wave in the 2d thread block
   //! will contribute to the fragment operation in row_major grid order.
   //! All waves are scheduled in row_major order.
   //! E.g. (TBlockX, TBlockY) => 2x2 waves
```

(continues on next page)

(continued from previous page)

```
    //! w0 = (0, 0),  w1 = (0, 1),
    //! w2 = (1, 0),  w3 = (1, 1)
    //! @tparam TBlockX the size of the thread-block in the X dimension
    //! @tparam TBlockY the size of the thread-block in the Y dimension
    template <uint32_t TBlockX, uint32_t TBlockY>
    using coop_row_major_2d = IOScheduler::RowMajor2d<TBlockX, TBlockY>;

    ...
}
```

Here is the simplified usage with new cooperative fragment changes:

```
// Global read (macro tile)
// Distribute segments of macro tile data between waves of the thread block in
// row major order.
using CoopScheduler = fragment_scheduler::coop_row_major_2d<TBLOCK_X, TBLOCK_Y>;
using GRBuffA = fragment<matrix_a, MACRO_TILE_X, ROCWMMA_N, ROCWMMA_K, InputT,␣
↪DataLayoutA, CoopScheduler>;

// Transfer data from global memory to local memory
GRBuffA grBuffA;
load_matrix_sync(grBuffA, gAddrA, lda);
store_matrix_sync(ldsAddr, apply_data_layout<DataLayoutLds>(apply_transpose(grBuffA)),␣
↪ldsld);
```

To summarize, the `CoopScheduler` template parameter allows you to express the required cooperative behavior with the fragment class declaration. Boilerplate code for calculating wave count and wave indices is wrapped into the `CoopScheduler` class. You can use fragments with the standard rocWMMA API without the need to externally propagate matching wave counts or wave indices, making rocWMMA more compact and expressive than previous versions.

Note: Cooperative fragment schedulers require template parameters for `TBLOCK_X` and `TBLOCK_Y` dimensions. This design enables various optimizations by allowing the schedulers to provide a static wave count at compile time. As a result, rocWMMA no longer supports run-time wave count calculations in favor of better performance.

4.2 Partial fragment support

In previous rocWMMA versions, fragment sizes were required to be a multiple of the minimum block sizes, as described in the *Programming guide*. This was a function of the MMA implementation of hardware acceleration. Thus, rocWMMA serves as a direct hardware enablement to employ block-wise decomposition of matrix-multiply problems. In absence of perfect block-wise decompositions, there is a need to accommodate odd-sized blocks or partials. To increase the utility of rocWMMA to more applications, rocWMMA was extended to include support for partial tile sizes, allowing fragment dimensions (FragMNK) to differ from the minimum block-wise dimensions required for MMA (BlockMNK). rocWMMA now pads FragMNK dimensions to meet the minimal BlockMNK dimensions, ensuring compatibility with MMA hardware.

```
// Fragment types, assuming ROCWMMA_MNK are minimum block sizes.
// These fragments will not use any padding.
using FragA = fragment<matrix_a, ROCWMMA_M, ROCWMMA_N, ROCWMMA_K, InputT, DataLayoutA>;
using FragB = fragment<matrix_b, ROCWMMA_M, ROCWMMA_N, ROCWMMA_K, InputT, DataLayoutB>;
```

(continues on next page)

(continued from previous page)

```
using Accum = fragment<accumulator, ROCWMMA_M, ROCWMMA_N, ROCWMMA_K, AccumT>;

FragA fragA;
FragB fragB;
Accum accum;
fill_fragment(accum, 0);

load_matrix_sync(fragA, gAddrA, lda);
load_matrix_sync(fragB, gAddrB, ldb);
mma_sync(accum, fragA, fragB, accum);

store_matrix_sync(gResC, accum, ldc, layout_t::mem_row_major);

// Now also supported

// Fragment types, which are partial fragments.
// These fragments will use padding to minimum block sizes internally.
// Note: The dimensions (2, 3, 1) are smaller than BlockMNK, creating partial fragments
using FragA = fragment<matrix_a, 2, 3, 1, InputT, DataLayoutA>;
using FragB = fragment<matrix_b, 2, 3, 1, InputT, DataLayoutB>;
using Accum = fragment<accumulator, 2, 3, 1, AccumT>;

FragA fragA;
FragB fragB;
Accum accum;
fill_fragment(accum, 0);

load_matrix_sync(fragA, gAddrA, lda);
load_matrix_sync(fragB, gAddrB, ldb);
mma_sync(accum, fragA, fragB, accum);

store_matrix_sync(gResC, accum, ldc, layout_t::mem_row_major);
```

In summary, partial tiles are padded to the minimum MMA block dimensions to accommodate a wider range of fragment sizes. However, this added flexibility comes at a cost: extra registers used for padding might increase kernel register pressure for small tiles and incur extra overhead for checking boundary conditions. Padded fragments are logically restricted to writing in FragMNK dimensions and zeroing boundary conditions.

Note: Padded fragment internals always use padded-sized resources instead of fragment-sized resources. However, fragment element-wise accesses, such as uniform FMA, should continue to use `fragment.num_elements`, assuming that any padded elements will be zero.

Example:

```
// Fused multiply-add still valid for partials as padded elements are 0
for(int i = 0; i < frag.num_elements; i++)
{
   frag.x[i] = frag.x[i] * (alpha + 1);
}
```

CHAPTER

FIVE

API REFERENCE GUIDE

This document provides information about rocWMMA functions, data types, and other programming constructs.

5.1 Synchronous API

rocWMMA API functions such as `load_matrix_sync`, `store_matrix_sync`, and `mma_sync` are synchronous when used with global memory. However, when you use these functions with shared memory, for example, LDS memory, explicit workgroup synchronization (`synchronize_workgroup`) might be required.

5.2 Supported GPU architectures

Supported CDNA architectures (wave64):

- gfx908
- gfx90a
- gfx942
- gfx950

Note: gfx9 refers to gfx908, gfx90a, gfx942, and gfx950.

Supported RDNA architectures (wave32):

- gfx1100
- gfx1101
- gfx1102
- gfx1200
- gfx1201

Note: gfx11 refers to gfx1100, gfx1101, and gfx1102. gfx12 refers to gfx1200 and gfx1201.

5.3 Supported data types

rocWMMA mixed precision multiply-accumulate operations support the following data type combinations.

Data Types **<Ti / To / Tc>** = <Input type / Output Type / Compute Type>, where:

- Input Type = Matrix A / B
- Output Type = Matrix C / D
- Compute Type = Math / Accumulation type

Supported data types:

- i8: 8-bit precision integer
- f8: 8-bit precision floating point
- bf8: 8-bit precision brain floating point
- f16: half-precision floating point
- bf16: half-precision brain floating point
- f32: single-precision floating point
- i32: 32-bit precision integer
- xf32: single-precision tensor floating point
- f64: double-precision floating point

Note: f16 includes support for both _Float16 and __half types.

f8 NANOO (optimized) format is only supported on gfx942, otherwise f8 OCP is assumed on targets that support f8 datatypes.

Ti / To / Tc	BlockM	BlockN	BlockK Range* (Powers of 2)	CDNA Support	RDNA Support
bf8 / f32 / f32	16	16	32+	gfx940, gfx950	gfx12
	32	32	16+		-
f8 / f32 / f32	16	16	32+	gfx940, gfx950	gfx12
	32	32	16+		-
i8 / i32 / i32	16	16	16	gfx908, gfx90a	gfx11, gfx12
			32	gfx940, gfx950	-
			64+	gfx950	-
	32	32	8	gfx908, gfx90a	-
			16	gfx940, gfx950	-

continues on next page

Table 1 – continued from previous page

Ti / To / Tc	BlockM	BlockN	BlockK Range* (Powers of 2)	CDNA Support
RDNA Support				
			32+	gfx950
-				
i8 / i8 / i32	16	16	16	gfx908, gfx90a
gfx11, gfx12				
			32	gfx940, gfx950
-				
			64+	gfx950
-				
	32	32	8	gfx908, gfx90a
-				
			16	gfx940, gfx950
-				
			32+	gfx950
-				
f16 / f32 / f32	16	16	16	gfx9
gfx11, gfx12				
			32+	gfx950
-				
	32	32	8	gfx9
-				
			16+	gfx950
-				
f16 / f16 / f32	16	16	16	gfx9
gfx11, gfx12				
			32+	gfx950
-				
	32	32	8	gfx9
-				
			16+	gfx950
-				
f16 / f16 / f16**	16	16	16	gfx9
gfx11, gfx12				
			32+	gfx950
-				
	32	32	8	gfx9
-				
			16+	gfx950
-				
bf16 / f32 / f32	16	16	8	gfx908
-				
			16	gfx90a, gfx942, gfx950
gfx11, gfx12				
			32+	gfx950
-				
	32	32	4+	gfx908
-				
			8	gfx90a, gfx942, gfx950
-				

continues on next page

Table 1 – continued from previous page

Ti / To / Tc	BlockM	BlockN	BlockK Range* (Powers of 2)	CDNA Support
RDNA Support				
			16+	gfx950
-				
bf16 / bf16 / f32	16	16	8	gfx908
-				
			16	gfx90a, gfx942, gfx950
gfx11, gfx12				
			32+	gfx950
-				
	32	32	4+	gfx908
-				
			8	gfx90a, gfx942, gfx950
-				
			16+	gfx950
-				
bf16 / bf16 / bf16**	16	16	8	gfx908
			16	gfx90a, gfx942, gfx950
gfx11, gfx12				
			32+	gfx950
-				
	32	32	4+	gfx908
-				
			8	gfx90a, gfx942, gfx950
-				
			16+	gfx950
-				
f32 / f32 / f32	16	16	4+	gfx9
-				
	32	32	2+	gfx9
-				
xf32 / xf32 / xf32	16	16	8+	gfx942
	32	32	4+	
f64 / f64 / f64	16	16	4+	gfx90a, gfx942, gfx950
-				

Note: BlockM/N values are minimum recommended values. Below these values padding is used which may impact performance. Above this value powers of 2 are acceptable.

* BlockK range specifies the minimum recommended value. Below this value padding is used which may impact performance. Above this value powers of 2 are acceptable. In practice, BlockK values are typically 32 or less.

** On CDNA architectures, matrix unit accumulation is performed in natively 32-bit precision and then converted to the target data type.

Note: rocWMMA supports partial fragment sizes where `FragMNK` may be smaller than the `BlockMNK` sizes listed in

the table above. These fragments are internally padded to nearest supported `BlockMNK` sizes.

5.4 Supported matrix layouts

(N = col major, T = row major)

LayoutA	LayoutB	Layout C	LayoutD
N	N	N	N
N	N	T	T
N	T	N	N
N	T	T	T
T	N	N	N
T	N	T	T
T	T	N	N
T	T	T	T

5.5 Supported thread block sizes

rocWMMA supports up to four wavefronts per thread block. The X dimension should be a multiple of the wave size and is scaled accordingly.

TBlock_X	TBlock_Y
WaveSize	1
WaveSize	2
WaveSize	4
WaveSize*2	1
WaveSize*2	2
WaveSize*4	1

Note: WaveSize (RDNA) = 32

WaveSize (CDNA) = 64

5.6 Using rocWMMA API

This section describes how to use the rocWMMA library API.

5.7 rocWMMA datatypes

5.7.1 matrix_a

struct **matrix_a**

Meta-tag indicating data context is input Matrix A.

5.7.2 matrix_b

struct **matrix_b**

Meta-tag indicating data context is input Matrix B.

5.7.3 accumulator

struct **accumulator**

Meta-tag indicating data context is Accumulator (also used as Matrix C / D).

5.7.4 row_major

struct **row_major**

Meta-tag indicating 2D in-memory data layout as row major.

5.7.5 col_major

struct **col_major**

Meta-tag indicating 2D in-memory data layout as column major.

5.7.6 default_schedule

typedef IOScheduler::Default `rocwmma::fragment_scheduler::`**default_schedule**

The default fragment scheduler; each wave operates independently.

5.7.7 coop_row_major_2d

typedef IOScheduler::RowMajor2d<TBlockX, TBlockY>
`rocwmma::fragment_scheduler::`**coop_row_major_2d**

A cooperative scheduling strategy where each wave in the 2d threadblock will contribute to the fragment operation in *row_major* grid order. All waves are scheduled in *row_major* order. E.g. (TBlockX, TBlockY) => 2x2 waves w0 = (0, 0), w1 = (0, 1), w2 = (1, 0), w3 = (1, 1)

Template Parameters

- **TBlockX** – the size of the thread-block in the X dimension
- **TBlockY** – the size of the thread-block in the Y dimension

5.7.8 coop_col_major_2d

typedef IOScheduler::ColMajor2d<TBlockX, TBlockY>
`rocwmma::fragment_scheduler::`**coop_col_major_2d**

A cooperative scheduling strategy where each wave in the 2d threadblock will contribute to the fragment operation in *col_major* grid order. All waves are scheduled in *row_major* order. E.g. (TBlockX, TBlockY) => 2x2 waves w0 = (0, 0), w2 = (0, 1), w1 = (1, 0), w3 = (1, 1)

Template Parameters

- **TBlockX** – the size of the thread-block in the X dimension
- **TBlockY** – the size of the thread-block in the Y dimension

5.7.9 coop_row_slice_2d

typedef IOScheduler::RowSlice2d<TBlockX, TBlockY>
`rocwmma::fragment_scheduler::`**coop_row_slice_2d**

A cooperative scheduling strategy where each row of waves in the 2d threadblock will contribute to the fragment operation. Waves are partitioned into rows. Only waves in the same row participate together. E.g. (TBlockX, TBlockY) = 2x2 waves RowSlice0: w0 = (0, 0), w1 = (0, 1) RowSlice1: w0 = (1, 0), w1 = (1, 1)

Template Parameters

- **TBlockX** – the size of the thread-block in the X dimension
- **TBlockY** – the size of the thread-block in the Y dimension

5.7.10 coop_col_slice_2d

typedef IOScheduler::ColSlice2d<TBlockX, TBlockY> `rocwmma::fragment_scheduler::`**coop_col_slice_2d**

A cooperative scheduling strategy where each col of waves in the 2d threadblock will contribute to the fragment operation. Waves are partitioned into cols. Only waves in the same col participate together. E.g. (TBlockX, TBlockY) = 2x2 waves ColSlice0: ColSlice1: w0 = (0, 0), w0 = (0, 1), w1 = (1, 0) w1 = (1, 1)

Template Parameters

- **TBlockX** – the size of the thread-block in the X dimension
- **TBlockY** – the size of the thread-block in the Y dimension

5.7.11 single

typedef IOScheduler::Single<TBlockX, TBlockY, WaveIdx> `rocwmma::fragment_scheduler::single`

A cooperative scheduling strategy where only one wave in the thread block will participate.

Template Parameters

- **TBlockX** – the size of the thread-block in the X dimension
- **TBlockY** – the size of the thread-block in the Y dimension
- **WaveIdx** – the index of the wave which will participate

5.7.12 fragment

template<typename **MatrixT**, uint32_t **FragM**, uint32_t **FragN**, uint32_t **FragK**, typename **DataT**, typename **DataLayoutT** = void, typename **Scheduler** = fragment_scheduler::*default_schedule*>
class **fragment**

rocWMMA fragment class. This is the primary object used in block-wise decomposition of the matrix multiply-accumulate (mma) problem space. In general, fragment data is associated with a matrix context (*matrix_a*, *matrix_b* or accumulator), a block size (BlockM/N/K), a datatype (e.g. single-precision float, etc.) and an in-memory 2D layout (e.g. *row_major* or *col_major*). These fragment properties are used to define how data is handled and stored locally, and to drive API implementations for loading / storing, mma and transforms. Fragment abstractions are designed to promote a simple wavefront programming model, which can accelerate development time. Internal thread-level details are handled by rocWMMA which frees the user to focus on wavefront block-wise decomposition. Written purely in device code, the programmer can use this object in their own device kernels.

Note: Fragments are stored in packed registers, however vector elements have no guaranteed order or locality.

Template Parameters

- **MatrixT** – fragment context
- **FragM/N/K** – fragment dimensions
- **DataT** – datatype
- **DataLayoutT** – in-memory layout as *col_major* or *row_major*
- **Scheduler** – wave-wise scheduler

Public Types

using **IOTraits** = typename IOConfig<*MatrixT*, *FragM*, *FragN*, *FragK*, *DataT*, *DataLayoutT*, *Scheduler*>::IOTraits

Input / output traits specific to AMDGCN architecture.

Public Functions

inline *DataT* &**operator[]**(uint32_t index)

Parameters
index – Element index

Returns
Mutable unpacked element accessor at given index

inline *DataT* const &**operator[]**(uint32_t index) const

Parameters
index – Element index

Returns
Immutable unpacked element accessor at given index

inline *Traits*::*StorageT* &**operator***()

Returns
Mutable packed storage vector accessor

inline *Traits*::*StorageT* const &**operator***() const

Returns
Immutable packed storage vector accessor

Public Members

union rocwmma::*fragment*::[anonymous] **[anonymous]**

Internal data storage views. Compatibility with nvcuda::wmma.

Public Static Functions

static inline constexpr uint32_t **height**()

Returns
The geometric height of fragment

static inline constexpr uint32_t **width**()

Returns
The geometric width of fragment

static inline constexpr uint32_t **blockDim**()

Returns
The leading block dimension (non-K)

static inline constexpr uint32_t **kDim**()

Returns
The k dimension

static inline constexpr uint32_t **size**()

Returns
The size of the unpacked elements vector

struct **Traits**

Public Types

using **AccessT** = VecT<UnpackedElementT, *Size*>
Unpacked data access view.

using **StorageT** = VecT<PackedElementT, *IOTraits*::PackedSize / WaveCount>
Packed data storage view.

Public Static Attributes

static constexpr uint32_t **Size** = *IOTraits*::UnpackedSize / WaveCount
Assert the fragment occupies at least one packed register.

Assert the fragment is equally splittable among the wave count

5.8 rocWMMA enumeration

5.8.1 layout_t

enum `rocwmma::layout_t`
Values:

enumerator **mem_row_major**

enumerator **mem_col_major**

5.9 rocWMMA API functions

template<typename **FragT**, typename **DataT**>
void `rocwmma::fill_fragment`(*FragT* &frag, *DataT* value)
Fills the entire fragment with the desired value.

Parameters

- **frag** – Fragment of type MatrixT with its associated block sizes, data type and layout
- **value** – Fill value of type DataT

Template Parameters

- **FragT** – Opaque fragment type
- **DataT** – Datatype

template<typename **FragT**, typename **DataT**>
void `rocwmma::load_matrix_sync`(*FragT* &frag, const *DataT* *data, uint32_t ldm)

Loads the entire fragment from the data pointer according to its matrix and data layout contexts. Data pointer may point to either local or global memory.

Parameters

- **frag** – Fragment of type MatrixT with its associated block sizes, data type and layout
- **data** – Data pointer to global or local memory
- **ldm** – Leading dimension size

Template Parameters

- **FragT** – Opaque fragment type
- **DataT** – Datatype

template<typename **FragT**, typename **DataT**>
void `rocwmma::load_matrix_sync`(*FragT* &frag, const *DataT* *data, uint32_t ldm, *layout_t* layout)

Loads the entire fragment from the data pointer according to its matrix layout and data layout contexts. Data pointer may point to either local or global memory. This overload provides manual selection of data layout of the incoming memory pointer, which will be transformed to conform to the data layout of the fragment.

Parameters

- **frag** – Fragment of type MatrixT with its associated block sizes, data type and layout
- **data** – Data pointer to global/local memory
- **ldm** – Leading dimension size
- **layout** – Data layout

Template Parameters

- **FragT** – Opaque fragment type
- **DataT** – Datatype

template<typename **FragT**, typename **DataT**>
void `rocwmma::store_matrix_sync`(*DataT* *data, *FragT* const &frag, uint32_t ldm)

Stores the entire fragment to the data pointer according to its matrix and data layouts. Data pointer may point to either local or global memory.

Parameters

- **frag** – Fragment of type MatrixT with its associated block sizes, data type and layout
- **data** – Data pointer to global/local memory
- **ldm** – Leading dimension size

Template Parameters

- **FragT** – Opaque fragment type
- **DataT** – Datatype

template<typename **FragT**, typename **DataT**>

void rocwmma::**store_matrix_sync**(*DataT* *data, *FragT* const &frag, uint32_t ldm, *layout_t* layout)

Stores the entire fragment to the data pointer according to its matrix layout and data layout contexts. Data pointer may point to either local or global memory. This overload provides manual selection of data layout of the outgoing memory pointer, which the data layout of the fragment will be transformed to.

Parameters

- **frag** – Fragment of type MatrixT with its associated block sizes, data type and layout
- **data** – Data pointer to global/local memory
- **ldm** – Leading dimension size
- **layout** – Data layout

Template Parameters

- **FragT** – Opaque fragment type
- **DataT** – Datatype

template<typename **FragA**, typename **FragB**, typename **FragAccumIn**, typename **FragAccumOut**>
void rocwmma::**mma_sync**(*FragAccumOut* &d, *FragA* const &a, *FragB* const &b, *FragAccumIn* &c)

Performs the Multiply-Accumulate operation on the fragments A, B, C and D (D = A * B + C)

Note: Frag c = d is valid

Parameters

- **d** – Accumulator output D
- **a** – Input fragment A
- **b** – Input fragment B
- **c** – Input accumulator fragment C

Template Parameters

- **FragA** – Opaque fragment type for matrix A data
- **FragB** – Opaque fragment type for matrix A data
- **FragAccumIn** – Opaque fragment type for input accumulation data
- **FragAccumOut** – Opaque fragment type for output accumulation data

void rocwmma::**synchronize_workgroup**()

Synchronization point for all wavefronts in a workgroup. Guarantees pending reads / writes to LDS are flushed.

5.9.1 rocWMMA transforms API functions

template<typename **FragT**>
static inline T rocwmma::**apply_transpose**(*FragT* &&frag)

Applies the transpose transform the input fragment. Transpose is defined as orthogonal matrix and data layout. E.g. T(fragment<matrix_a, BlockM, BlockN, BlockK, DataT, row_major>) = fragment<matrix_b, BlockN, BlockM, BlockK, DataT, col_major>

Parameters

frag – Fragment of type MatrixT with its associated block sizes, data type and layout

Template Parameters
FragT – The incoming fragment type

Returns
Transposed (orthogonal) fragment

template<typename **DataLayoutT**, typename **FragT**>
static inline T rocwmma::**apply_data_layout**(*FragT* &&frag)

Transforms the input fragment to have the desired data layout.

Parameters
frag – Fragment of type MatrixT with its associated block sizes, data type and layout

Template Parameters

- **DataLayoutT** – The desired fragment data layout to apply
- **FragT** – The incoming fragment type

Returns
Fragment with transformed data layout

template<typename **DstFragT**, typename **FragT**>
static inline T rocwmma::**apply_fragment**(*FragT* &&frag)

Transforms the input fragment to the target fragment type. This could include changing matrix context and/or changing data layout, as long as there is a path from the source register layout to the destination register layout.

Parameters
frag – Source fragment of type MatrixT with its associated block sizes, data type and layout

Template Parameters

- **DstFragT** – The target fragment type to transform to
- **FragT** – The source incoming fragment type

Returns
Target fragment after transformation

template<typename **FragT**>
static inline T rocwmma::**to_register_file**(*FragT* &&frag)

Transforms the input fragment to a "register file" fragment type. Register contents are directly mapped to a 2D matrix space represented by [RegCount x WaveSize]. This transform is a geometry reinterpretation.

Parameters
frag – Source fragment of type MatrixT with its associated block sizes, data type and layout

Template Parameters
FragT – The source incoming fragment type

Returns
Target fragment after transformation

template<typename **DstFragT**, typename **FragT**>
static inline T rocwmma::**from_register_file**(*FragT* &&frag)

Transforms the "register file" fragment type to a target fragment type. Register contents are directly mapped to a 2D matrix space represented by [RegCount x WaveSize]. This transform is a geometry reinterpretation.

Parameters
frag – Source fragment of type MatrixT with its associated block sizes, data type and layout

Template Parameters

- **DstFragT** – The target frag to transform to
- **FragT** – The source incoming fragment type as register file

Returns
Fragment after transformation

5.10 Sample programs

A sample demonstrating the use of rocWMMA functions `load_matrix_sync`, `store_matrix_sync`, `fill_fragment`, and `mma_sync` is available here. For more sample programs, refer to the samples directory.

5.11 Emulation tests

The emulation test is a smaller test suite designed for emulators. It includes a subset of ROCWMMA test cases for faster execution on emulated platforms. It supports `smoke`, `regression`, and `extended` modes.

For example, to run a smoke test:

```
rtest.py --install_dir <build_dir> --emulation smoke
```

CHAPTER

SIX

LICENSE

INDEX

R