rocWMMA Documentation

Advanced Micro Devices

Sep 02, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

rocWMMA is AMD's C++ library for accelerating mixed precision matrix multiply-accumulate operations leveraging specialized GPU matrix cores on AMD's latest discrete GPUs.

A C++ API is provided to facilitate decomposition of matrix multiply-accumulate problems into discretized block fragments and to parallelize block-wise operations across multiple GPU wavefronts.

The API is implemented in GPU device code: it empowers user device kernel code with direct use of GPU matrix cores. Moreover, this code can benefit from inline compiler optimization passes and does not incur additional overhead of external runtime calls or extra kernel launches.

Acronym	Expansion
GEMM	**GE**neral **M**atrix **M**ultiply
WMMA	**W**avefront **M**ixed precision **M**ultiply **A**ccumulate
ROCm	**R**adeon **O**pen **C**ompute platfor**m**
HIP	**H**eterogeneous-Compute **I**nterface for **P**ortability

rocWMMA is written in C++14 and may be applied directly in device kernel code. Library code is templated for modularity and uses available meta-data to provide opportunities for compile-time inferences and optimizations.

The rocWMMA API exposes block-wise data load / store and matrix multiply-accumulate functions appropriately sized for thread-block execution on data fragments. Matrix multiply-accumulate functionality supports mixed precision inputs and outputs with native fixed-precision accumulation. The rocWMMA Coop API provides wave/warp collaborations within the thread-blocks for block-wise data load and stores. Supporting code is required for GPU device management and for kernel invocation. Kernel code samples and tests provided are built and launched via the HIP ecosystem within ROCm.

Below is a simple example code for calling rocWMMA functions load_matrix_sync, store_matrix_sync, fill_fragment, mma_sync.

```
#include <hip/hip_ext.h>
#include <hip/hip_fp16.h>
#include <hip/hip_runtime.h>

#include <iostream>
#include <vector>

#include <rocwmma/rocwmma.hpp>

using rocwmma::float16_t;
using rocwmma::float32_t;

// Matrix data initialization
```

(continues on next page)

(continued from previous page)

```
template <typename DataT>
__host__ static inline void fill(DataT* mat, uint32_t m, uint32_t n)
{
    auto ld = n;
    for(int i = 0; i < m; ++i)
    {
        for(int j = 0; j < n; ++j)
        {
            // Generated data
            // Alternate sign every 3 elements
            auto value      = (i * n + j) % 13;
            mat[i * ld + j] = (value % 3) ? -static_cast<DataT>(value) : static_cast
↪<DataT>(value);
        }
    }
}

// Supports BlockM/N square sizes of
// : 16 x 16
// : 32 x 32
const int ROCWMMA_M = 16;
const int ROCWMMA_N = 16;

// Supports ROCWMMA_K sizes as
// : multiples of 16.
const int ROCWMMA_K = 16;

// AMDGCN default wave size
const int WAVE_SIZE = rocwmma::AMDGCN_WAVE_SIZE;

// Thread block
// : T_BLOCK_X must be multiple of WAVE_SIZE.
// Note: Each wave will compute one BLOCK_M x BLOCK_N output block
// Note: Workgroup will compute
//  T_BLOCK_X / WAVE_SIZE x T_BLOCK_Y output blocks
// This thread block will compute (4 x 4 output blocks)
const int T_BLOCK_X = 4 * WAVE_SIZE;
const int T_BLOCK_Y = 4;

// The following device kernel is a naive implementation
// of blocked GEMM. Each wave will compute one BLOCK_M x BLOCK_N
// output block of the M x N x K GEMM, generalized as:
// D = alpha * (A x B) + beta * C
//
// In this simplified example, we assume:
// : A is in row-major format     (m x k)
// : B is in col-major format     (k x n)
// : C, D are in row-major format (m x n)
// : Multiplication is NOT in-place, output is written to D matrix
// : No LDS required
//
// Disclaimer: This is a simplified implementation to demonstrate API usage in
```

(continues on next page)

(continued from previous page)

```
// context of wave-level GEMM computation, and is not optimized.
//
// Launchable device kernel function:
//
__global__ void gemm_wmma_d(uint32_t         m,     // matrix free dim m
                            uint32_t         n,     // matrix free dim n
                            uint32_t         k,     // matrix fixed dim k
                            float16_t const* a,     // device data ptr for matrix A
                            float16_t const* b,     // device data ptr for matrix B
                            float32_t const* c,     // device data ptr for matrix C
                            float32_t*       d,     // device data ptr for matrix D
                            uint32_t         lda,   // leading dimension for matrix A
                            uint32_t         ldb,   // leading dimension for matrix B
                            uint32_t         ldc,   // leading dimension for matrix C
                            uint32_t         ldd,   // leading dimension for matrix D
                            float32_t        alpha, // uniform scalar
                            float32_t        beta)  // uniform scalar
{
    // Create frags with meta-data context for block-wise GEMM decomposition
    // @tp0: fragment context = matrix_a, matrix_b or accumulator
    // @tp1: block size M
    // @tp2: block size N
    // @tp3: block size K
    // @tp4: fragment data type
    // @tp5: data layout = row_major, col_major or void (default)
    auto fragA = rocwmma::fragment<rocwmma::matrix_a, ROCWMMA_M, ROCWMMA_N, ROCWMMA_K,␣
↪float16_t, rocwmma::row_major>();
    auto fragB = rocwmma::fragment<rocwmma::matrix_b, ROCWMMA_M, ROCWMMA_N, ROCWMMA_K,␣
↪float16_t, rocwmma::col_major>();
    auto fragC   = rocwmma::fragment<rocwmma::accumulator, ROCWMMA_M, ROCWMMA_N, ROCWMMA_
↪K, float32_t>();
    auto fragAcc = rocwmma::fragment<rocwmma::accumulator, ROCWMMA_M, ROCWMMA_N, ROCWMMA_
↪K, float32_t>();

    // Initialize accumulator fragment
    rocwmma::fill_fragment(fragAcc, 0.0f);

     // Tile using a 2D grid
     auto majorWarp = (blockIdx.x * blockDim.x + threadIdx.x) / WAVE_SIZE;
     auto minorWarp = (blockIdx.y * blockDim.y + threadIdx.y);

     // Target C block
     auto cRow = majorWarp * ROCWMMA_M;
     auto cCol = minorWarp * ROCWMMA_N;

    // Bounds check
    if(cRow < m && cCol < n)
    {
         // fragAcc = A x B
         for(int i = 0; i < k; i += ROCWMMA_K)
         {
             // Load the inputs
```

(continues on next page)

(continued from previous page)

```
            rocwmma::load_matrix_sync(fragA, a + (cRow * lda + i), lda);
            rocwmma::load_matrix_sync(fragB, b + (i + cCol * ldb), ldb);

            // Matrix multiply - accumulate using MFMA units
            rocwmma::mma_sync(fragAcc, fragA, fragB, fragAcc);
        }

        // Fetch C matrix
        rocwmma::load_matrix_sync(fragC, c + (cRow * ldc + cCol), ldc, rocwmma::mem_row_
↪major);

        // D = alpha * A x B + beta * C
        for(int i = 0; i < fragC.num_elements; ++i)
        {
            fragC.x[i] = alpha * fragAcc.x[i] + beta * fragC.x[i];
        }

        // Store to D
        rocwmma::store_matrix_sync(d + (cRow * ldd + cCol), fragC, ldd, rocwmma::mem_
↪row_major);
    }
}

// Host side supporting device mgmt and launch code
__host__ void gemm_test(uint32_t m, uint32_t n, uint32_t k, float32_t alpha, float32_t␣
↪beta)
{
    // Problem size check
    if((m < (ROCWMMA_M * T_BLOCK_X / WAVE_SIZE) || n < (ROCWMMA_N * T_BLOCK_Y) || k <␣
↪ROCWMMA_K)
       || (m % ROCWMMA_M || n % ROCWMMA_N || k % ROCWMMA_K))
    {
        std::cout << "Unsupported size!\n";
        return;
    }

    int lda = k;
    int ldb = k;
    int ldc = n;
    int ldd = ldc;

    std::cout << "Initializing host data..." << std::endl;

    // Initialize input matrices
    std::vector<float16_t> matrixA(m * k);
    std::vector<float16_t> matrixB(k * n);
    std::vector<float32_t> matrixC(m * n);
    // Fill outputs with NaN to catch contamination
    std::vector<float32_t> matrixD(m * n, std::numeric_limits<float32_t>::signaling_
↪NaN());

    fill(matrixA.data(), m, k);
```

(continues on next page)

(continued from previous page)

```
    fill(matrixB.data(), k, n);
    fill(matrixC.data(), m, n);

    std::cout << "Initializing device data..." << std::endl;

    // Allocate and copy device memory
    float16_t* d_a;
    float16_t* d_b;
    float32_t* d_c;
    float32_t* d_d;

    const size_t bytesA = matrixA.size() * sizeof(float16_t);
    const size_t bytesB = matrixB.size() * sizeof(float16_t);
    const size_t bytesC = matrixC.size() * sizeof(float32_t);
    const size_t bytesD = matrixD.size() * sizeof(float32_t);

    CHECK_HIP_ERROR(hipMalloc(&d_a, bytesA));
    CHECK_HIP_ERROR(hipMalloc(&d_b, bytesB));
    CHECK_HIP_ERROR(hipMalloc(&d_c, bytesC));
    CHECK_HIP_ERROR(hipMalloc(&d_d, bytesD));

    CHECK_HIP_ERROR(hipMemcpy(d_a, matrixA.data(), bytesA, hipMemcpyHostToDevice));
    CHECK_HIP_ERROR(hipMemcpy(d_b, matrixB.data(), bytesB, hipMemcpyHostToDevice));
    CHECK_HIP_ERROR(hipMemcpy(d_c, matrixC.data(), bytesC, hipMemcpyHostToDevice));
    CHECK_HIP_ERROR(hipMemcpy(d_d, matrixD.data(), bytesD, hipMemcpyHostToDevice));

     auto blockDim = dim3(T_BLOCK_X, T_BLOCK_Y);
     auto gridDim  = dim3(rocwmma::ceilDiv(m, ROCWMMA_M * T_BLOCK_X / WAVE_SIZE),
            rocwmma::ceilDiv(n, ROCWMMA_N * T_BLOCK_Y));

     std::cout << "Launching GEMM kernel..." << std::endl;

     hipEvent_t startEvent, stopEvent;
     CHECK_HIP_ERROR(hipEventCreate(&startEvent));
     CHECK_HIP_ERROR(hipEventCreate(&stopEvent));

     hipExtLaunchKernelGGL(gemm_wmma_d,
                     gridDim,
                     blockDim,
                     0, // sharedMemBytes
                     0, // stream
                     startEvent, // Event start
                     stopEvent, // event stop
                     0, // flags
                     m,
                     n,
                     k,
                     d_a,
                     d_b,
                     d_c,
                     d_d,
                     lda,
```

(continues on next page)

(continued from previous page)

```
                        ldb,
                        ldc,
                        ldd,
                        alpha,
                        beta);

      auto elapsedTimeMs = 0.0f;
      CHECK_HIP_ERROR(hipEventSynchronize(stopEvent));
      CHECK_HIP_ERROR(hipEventElapsedTime(&elapsedTimeMs, startEvent, stopEvent));
      CHECK_HIP_ERROR(hipEventDestroy(startEvent));
      CHECK_HIP_ERROR(hipEventDestroy(stopEvent));

      // Release device memory
      CHECK_HIP_ERROR(hipFree(d_a));
      CHECK_HIP_ERROR(hipFree(d_b));
      CHECK_HIP_ERROR(hipFree(d_c));
      CHECK_HIP_ERROR(hipFree(d_d));

      std::cout << "Finished!" << std::endl;
}

int main()
{
    gemm_test(256, 256, 256, 2.1f, 2.1f);
    return 0;
}
```

1.1 Synchronous API

In general, rocWMMA API functions (load_matrix_sync, store_matrix_sync, mma_sync) are assumed to be synchronous when used in context of global memory.

When using these functions in the context of shared memory (e.g. LDS memory), additional explicit workgroup synchronization may be required due to the nature this memory usage.

1.2 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

Ti / To / Tc	BlockM	BlockN	BlockK
i8 / i32 / i32	16	16	Min: 16, pow2
i8 / i32 / i32	32	32	Min: 8, pow2
i8 / i8 / i32	16	16	Min: 16, pow2
i8 / i32 / i32	32	32	Min: 8, pow2
f16 / f32 / f32	16	16	Min: 16, pow2
f16 / f32 / f32	32	32	Min: 8, pow2
f16 / f16 / f32	16	16	Min: 16, pow2
f16 / f16 / f32	32	32	Min: 8, pow2
f16 / f16 / f16*	16	16	Min: 16, pow2
f16 / f16 / f16*	32	32	Min: 8, pow2
__half / f32 / f32	16	16	Min: 16, pow2
__half / f32 / f32	32	32	Min: 8, pow2
__half / __half / f32	16	16	Min: 16, pow2
__half / __half / f32	32	32	Min: 8, pow2
__half / __half / __half*	16	16	Min: 16, pow2
__half / __half / __half*	32	32	Min: 8, pow2
bf16 / f32 / f32	16	16	Min: 8, pow2
bf16 / f32 / f32	32	32	Min: 4, pow2
bf16 / bf16 / f32	16	16	Min: 8, pow2
bf16 / bf16 / f32	32	32	Min: 4, pow2
bf16 / bf16 / bf16*	16	16	Min: 8, pow2
bf16 / bf16 / bf16*	32	32	Min: 4, pow2
f32 / f32 / f32	16	16	Min: 4, pow2
f32 / f32 / f32	32	32	Min: 2, pow2
f64** / f64** / f64**	16	16	Min: 4, pow2

*= matrix unit accumulation is natively 32 bit precision, and is converted to desired type.

**= f64 datatype is only supported on MI-200 class AMDGPU and successors.

1.3 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

1.3.1 Using rocWMMA API

This section describes how to use the rocWMMA library API.

1.4 rocWMMA Datatypes

struct **matrix_a**
Input Matrix A.

struct **matrix_b**
Input Matrix B.

struct **accumulator**
Input/Output Matrix Accumulator.

struct **row_major**
Data/In-memory Layout as Row Major.

struct **col_major**
Data/In-memory Layout as Column Major.

class **VecT**
Functional vector class.

tparam T Vector data type

tparam VecSize Vector storage size

template<typename **T**, int **Elements**, typename **IsNativeType** = typename std::is_fundamental<*T*>::type>

struct **VectorStorage**
Vectorized internal storage.

tparam T Storage type

tparam Elements No of Elements in the vector

tparam IsNativType Native or rocWMMA defined

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**, typename **DataLayoutT**>
struct **IOConfig**
Definition of ROCWMMA fragment input / output configurations in specific matrix context.

tparam Matrix fragment context

tparam BlockM/N/K block dimensions

tparam DataT data type

tparam DataLayout in-memory layout as *col_major* or *row_major*

param BlockDim leading block dimension (row / col size)

param KDim minor block dimension (row / col count)

param MaxVectorWidth maximum allowable vector width

param VectorWidth currently used vector width

param CoopIndex shared wave index (0 = row, 1 = col)

param IOTraits Input/output traits specific to AMDGCN architecture

param Packer Packs raw fragment data into register

param Unpacker Unpacks registers to raw fragment data

param Broadcaster Sets all fragment data to a desired value

param MatrixLayout Maps GPU threads to matrix shape or geometry

param Loader Issues load instructions for raw fragment data

param Storer Issues store instructions for raw fragment data

param CoopLoader Issues cooperative load instructions for raw fragment data

param CoopStorer Issues cooperative store instructions for raw fragment data

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**, typename **DataLayoutT**>
struct **IOShape**

Definition of rocWMMA data and matrix mapping utilities in specific matrix context.

tparam MatrixT fragment context

tparam BlockM/N/K block dimensions

tparam DataT data type

tparam DataLayoutT in-memory layout as *col_major* or *row_major*

1.5 rocWMMA Enumeration

Enumeration constants have numbering that is consistent with standard C++ libraries.

enum `rocwmma::layout_t`

Values:

enumerator **mem_row_major**

enumerator **mem_col_major**

1.6 rocWMMA API functions

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**, typename **DataLayout**>
void `rocwmma::fill_fragment`(fragment<*MatrixT*, *BlockM*, *BlockN*, *BlockK*, *DataT*, *DataLayout*> &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** – Value of type DataT.

Template Parameters

- **Matrix** – fragment context
- **BlockM/N/K** – block dimensions
- **DataT** – data type
- **DataLayout** – in-memory layout as *col_major* or *row_major*

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**, typename **DataLayout**>
void `rocwmma::load_matrix_sync`(fragment<*MatrixT*, *BlockM*, *BlockN*, *BlockK*, *DataT*, *DataLayout*> &frag, const *DataT* *data, uint32_t ldm)

Loads the entire fragment from 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

- **MatrixT** – fragment context
- **BlockM/N/K** – block dimensions
- **DataT** – data type
- **DataLayout** – in-memory layout as *col_major* or *row_major*

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**>
void `rocwmma::load_matrix_sync`(fragment<*MatrixT*, *BlockM*, *BlockN*, *BlockK*, *DataT*> &frag, const *DataT* *data, uint32_t ldm, *layout_t* layout)

Loads the entire fragment from the data pointer according to its matrix layout.Data pointer may point to either local or global memory. This overload provides a run-time ability to choose the data layout of the target 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** – Matrix layout

Template Parameters

- **MatrixT** – fragment context
- **BlockM/N/K** – block dimensions
- **DataT** – data type
- **DataLayout** – in-memory layout as *col_major* or *row_major*

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**, typename **DataLayout**>
void `rocwmma::store_matrix_sync`(*DataT* *data, fragment<*MatrixT*, *BlockM*, *BlockN*, *BlockK*, *DataT*, *DataLayout*> 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

- **MatrixT** – fragment context
- **BlockM/N/K** – block dimensions
- **DataT** – data type
- **DataLayout** – in-memory layout as *col_major* or *row_major*

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**>
void `rocwmma::store_matrix_sync`(*DataT* *data, fragment<*MatrixT*, *BlockM*, *BlockN*, *BlockK*, *DataT*> const &frag, uint32_t ldm, *layout_t* layout)

Stores the entire fragment to the data pointer according to its matrix layout. Data pointer may point to either local or global memory. This overload provides a run-time ability to choose the data layout of the target 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

- **MatrixT** – fragment context
- **BlockM/N/K** – block dimensions
- **DataT** – data type
- **DataLayout** – in-memory layout as *col_major* or *row_major*

template<uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **InputT**, typename **ComputeT**, typename **LayoutA**, typename **LayoutB**>
void `rocwmma::mma_sync`(fragment<*accumulator*, *BlockM*, *BlockN*, *BlockK*, *ComputeT*> &d, fragment<*matrix_a*, *BlockM*, *BlockN*, *BlockK*, *InputT*, *LayoutA*> const &a, fragment<*matrix_b*, *BlockM*, *BlockN*, *BlockK*, *InputT*, *LayoutB*> const &b, fragment<*accumulator*, *BlockM*, *BlockN*, *BlockK*, *ComputeT*> const &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

- **BlockM/N/K** – block dimensions
- **InputT** – data type of input frags A and B
- **ComputeT** – data type of accumulator fragment C / D
- **LayoutA** – in-memory layout of frag A as *col_major* or *row_major*
- **LayoutB** – in-memory layout of frag B as *col_major* or *row_major*

void rocwmma::**synchronize_workgroup**()
Synchronization point for all wavefronts in a workgroup.

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**, typename **DataLayout**>
void rocwmma::**load_matrix_coop_sync**(fragment<*MatrixT*, *BlockM*, *BlockN*, *BlockK*, *DataT*, *DataLayout*> &frag, const *DataT* *data, uint32_t ldm, uint32_t waveIndex, uint32_t waveCount, uint32_t splitCount)

Cooperative Load Matrix - Loads the entire fragment with data from memory address cooperatively across waves. Each cooperative wave is responsible in loading a portion of the final fragment. Note that the full fragment data is not cohesive for individual waves as they only load a piece of the data. This function may be paired with store_matrix_sync to move a single fragment collaboratively between memory locations.

The full load is split into work items (splitCount). Work items are assigned in round robin fashion to waves in the range of [0, waveCount). The current wave index determines the order of the current wave in the collaboration pool. Work items are consumed in order by waves [0, waveCount) until there are no more work items and the operation is completed.

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
- **waveIndex** – Index assignment of current wave in collaboration
- **waveCount** – Number of waves assigned for collaboration
- **splitCount** – Number of work items to split the operation

Template Parameters

- **MatrixT** – fragment context
- **BlockM/N/K** – block dimensions
- **DataT** – data type

- **DataLayout** – in-memory layout as *col_major* or *row_major*

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**, typename **DataLayout**>
inline void `rocwmma::load_matrix_coop_sync`(fragment<*MatrixT*, *BlockM*, *BlockN*, *BlockK*, *DataT*, *DataLayout*> &frag, const *DataT* *data, uint32_t ldm, uint32_t waveIndex, uint32_t waveCount)

Cooperative Load Matrix - Loads the entire fragment with data from memory address cooperatively across waves. Each cooperative wave is responsible in loading a portion of the final fragment. Note that the full fragment data is not cohesive for individual waves as they only load a piece of the data. This function may be paired with store_matrix_sync to move a single fragment collaboratively between memory locations.

The full load is split into work items (default = waveCount). Work items are assigned in round robin fashion to waves in the range of [0, waveCount). The current wave index determines the order of the current wave in the collaboration pool. Work items are consumed in order by waves [0, waveCount) until there are no more work items and the operation is completed.

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
- **waveIndex** – Index assignment of current wave in collaboration
- **waveCount** – Number of waves assigned for collaboration

Template Parameters

- **MatrixT** – fragment context
- **BlockM/N/K** – block dimensions
- **DataT** – data type
- **DataLayout** – in-memory layout as *col_major* or *row_major*

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**, typename **DataLayout**>
void `rocwmma::load_matrix_coop_sync`(fragment<*MatrixT*, *BlockM*, *BlockN*, *BlockK*, *DataT*, *DataLayout*> &frag, const *DataT* *data, uint32_t ldm)

Cooperative Load Matrix - Loads the entire fragment with data from memory address cooperatively across waves. Each cooperative wave is responsible in loading a portion of the final fragment. Note that the full fragment data is not cohesive for individual waves as they only load a piece of the data. This function may be paired with store_matrix_sync to move a single fragment collaboratively between memory locations.

The full load is split into work items. This overload is conducive to GEMM workload where *matrix_a* fragments collaboratively load common data with other waves in the same row. Likewise, *matrix_b* fragments collaboratively load common data with other waves in the same column. Workload is split evenly across all waves in collaborative dimension. Split count = wave count = workgroupDim<1> (*matrix_a*) | workgroupDim<0> (*matrix_b*) Wave index = waveCoord<1> (*matrix_a*) | waveCoord<0> (*matrix_b*)

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

- **MatrixT** – fragment context
- **BlockM/N/K** – block dimensions
- **DataT** – data type
- **DataLayout** – in-memory layout as *col_major* or *row_major*

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**, typename **DataLayout**>
void `rocwmma::store_matrix_coop_sync`(*DataT* *data, fragment<*MatrixT*, *BlockM*, *BlockN*, *BlockK*, *DataT*, *DataLayout*> const &frag, uint32_t ldm, uint32_t waveIndex, uint32_t waveCount, uint32_t splitCount)

Cooperative Store Matrix - Stores the entire fragment to data address cooperatively across waves. Each cooperative wave is responsible in storing a portion of the final fragment. Note that the full fragment data is not required to be cohesive for individual waves as they only store a piece of the data. This function may be paired with load_matrix_sync to move a single fragment collaboratively between memory locations.

The full store is split into work items (splitCount). Work items are assigned in round robin fashion to waves in the range of [0, waveCount). The current wave index determines the order of the current wave in the collaboration pool. Work items are consumed in order by waves [0, waveCount) until there are no more work items and the operation is completed.

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
- **waveIndex** – Index assignment of current wave in collaboration
- **waveCount** – Number of waves assigned for collaboration
- **splitCount** – Number of work items to split the operation

Template Parameters

- **MatrixT** – fragment context
- **BlockM/N/K** – block dimensions
- **DataT** – data type
- **DataLayout** – in-memory layout as *col_major* or *row_major*

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**, typename **DataLayout**>
void `rocwmma::store_matrix_coop_sync`(*DataT* *data, fragment<*MatrixT*, *BlockM*, *BlockN*, *BlockK*, *DataT*, *DataLayout*> const &frag, uint32_t ldm, uint32_t waveIndex, uint32_t waveCount)

Cooperative Store Matrix - Stores the entire fragment to data address cooperatively across waves. Each cooperative wave is responsible in storing a portion of the final fragment. Note that the full fragment data is not required to be cohesive for individual waves as they only store a piece of the data. This function may be paired with load_matrix_sync to move a single fragment collaboratively between memory locations.

The full store is split into work items (default = waveCount). Work items are assigned in round robin fashion to waves in the range of [0, waveCount). The current wave index determines the order of the current wave in the collaboration pool. Work items are consumed in order by waves [0, waveCount) until there are no more work items and the operation is completed.

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
- **waveIndex** – Index assignment of current wave in collaboration
- **waveCount** – Number of waves assigned for collaboration

Template Parameters

- **MatrixT** – fragment context
- **BlockM/N/K** – block dimensions
- **DataT** – data type
- **DataLayout** – in-memory layout as *col_major* or *row_major*

template<typename **MatrixT**, uint32_t **BlockM**, uint32_t **BlockN**, uint32_t **BlockK**, typename **DataT**, typename **DataLayout**>
void `rocwmma::store_matrix_coop_sync`(*DataT* *data, fragment<*MatrixT*, *BlockM*, *BlockN*, *BlockK*, *DataT*, *DataLayout*> const &frag, uint32_t ldm)

Cooperative Store Matrix - Stores the entire fragment to data address cooperatively across waves. Each cooperative wave is responsible in storing a portion of the final fragment. Note that the full fragment data is not required to be cohesive for individual waves as they only store a piece of the data. This function may be paired with load_matrix_sync to move a single fragment collaboratively between memory locations.

The full store is split into work items. This overload is conducive to GEMM workload where *matrix_a* fragments collaboratively use common data with other waves in the same row. Likewise, *matrix_b* fragments collaboratively use common data with other waves in the same column. Workload is split evenly across all waves in collaborative dimension.

Split count = wave count = workgroupDim<1> (*matrix_a*) | workgroupDim<0> (*matrix_b*) Wave index = waveCoord<1> (*matrix_a*) | waveCoord<0> (*matrix_b*)

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

- **MatrixT** – fragment context
- **BlockM/N/K** – block dimensions
- **DataT** – data type
- **DataLayout** – in-memory layout as *col_major* or *row_major*

CHAPTER

TWO

GETTING STARTED GUIDE FOR LINUX

2.1 Introduction

This document contains instructions for installing, using, and contributing to rocWMMA. The quickest way to install is from prebuilt packages. Alternatively, there are instructions to build from source. The document also contains an API Reference Guide, Programmer's Guide, and Contributor's Guides.

2.1.1 Documentation Roadmap

The following is a list of rocWMMA documents in the suggested reading order:

- Getting Started Guide (this document): Describes how to install and configure the rocWMMA library; designed to get users up and running quickly with the library.
- API Reference Guide : Provides detailed information about rocWMMA functions, data types and other programming constructs.
- Programmer's Guide: Describes the code organization, Design implementation detail, Optimizations used in the library and those that should be considered for new development and Testing & Benchmarking detail.
- Contributor's Guide : Describes coding guidelines for contributors.

2.2 Prerequisites

- A ROCm enabled platform, more information here.

2.3 Installing pre-built packages

rocWMMA can be installed on Ubuntu or Debian using

```
sudo apt-get update
sudo apt-get install rocWMMA
```

rocWMMA can be installed on CentOS using

```
sudo yum update
sudo yum install rocWMMA
```

rocWMMA can be installed on SLES using

```
sudo dnf upgrade
sudo dnf install rocWMMA
```

Once installed, rocWMMA can be used just like any other library with a C++ API. The rocwmma.hpp header file will need to be included in the user code in order to make calls into rocWMMA.

Once installed, rocwmma.hpp can be found in the /opt/rocm/include directory. Only this installed file should be used when needed in user code. Other rocWMMA files can be found in /opt/rocm/include/internal, however these files should not be directly included.

2.4 Building and Installing rocWMMA

For most users building from source is not necessary, as rocWMMA can be used after installing the pre-built packages as described above. If desired, the following instructions can be used to build rocWMMA from source.

2.4.1 System Requirements

As a general rule, 5GB of system memory is required for a full rocWMMA build. This value can be lower if rocWMMA is built without tests. This value may also increase in the future as more functions are added to rocWMMA.

2.4.2 Minimum GPU Requirements

- AMD Instinct™ class GPU with matrix core support: Minimum MI-100
- Note: Double precision FP64 datatype support minimum MI-200 +

2.4.3 Download rocWMMA

The rocWMMA source code is available at the rocWMMA github page. rocWMMA has a minimum ROCm support version 4.3. Check the ROCm Version on your system. For Ubuntu use

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

For Centos use

```
yum info rocm-libs
```

The ROCm version has major, minor, and patch fields, possibly followed by a build specific identifier. For example the ROCm version could be 4.0.0.40000-23, this corresponds to major = 4, minor = 0, patch = 0, build identifier 40000-23. There are GitHub branches at the rocWMMA site with names rocm-major.minor.x where major and minor are the same as in the ROCm version. For ROCm version 4.0.0.40000-23 you need to use the following to download rocWMMA:

```
git clone -b release/rocm-rel-x.y https://github.com/ROCmSoftwarePlatform/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 5.0 installed, then replace release/rocm-rel-x.y with release/rocm-rel-5.0

The user can build either

- library

- library + samples
- library + tests
- library + tests + assembly

You only need (library) if you call rocWMMA from your code. The client contains the test samples and benchmark code.

Below are the project options available to build rocWMMA library with/without clients.

Option	Description	Default Value
AMDGPU_TARGETS	Build code for specific GPU target(s)	gfx908:xnack-;gfx90a:xnack-;gfx90a:xnack+
ROCWMMA_BUILD_TESTS	Build Tests	ON
ROCWMMA_BUILD_SAMPLES	Build Samples	ON
ROCWMMA_BUILD_ASSEMBLY	Generate assembly files	OFF
ROCWMMA_BUILD_VALIDATION_TESTS	Build validation tests	ON (requires ROCWMMA_BUILD_TESTS=ON)
ROCWMMA_BUILD_BENCHMARK_TESTS	Build benchmark tests	OFF (requires ROCWMMA_BUILD_TESTS=ON)
ROCWMMA_BUILD_EXTENDED_TESTS	Build extended testing coverage	OFF (requires ROCWMMA_BUILD_TESTS=ON)
WMMA_VALIDATE_WITH_ROCBLAS	Use rocBLAS for validation tests	ON (requires ROCWMMA_BUILD_VALIDATION_TESTS=ON)
WMMA_BENCHMARK_WITH_ROCBLAS	Include rocBLAS benchmarking data	OFF (requires ROCWMMA_BUILD_BENCHMARK_TESTS=ON)

2.4.4 Build only library

CMake has a minimum version requirement 3.5.

Minimum ROCm version support is 4.3.

By default, the project is configured as Release mode.

To build only library, run the following comomand :

```
CC=hipcc CXX=hipcc cmake -B<build_dir> . -DROCWMMA_BUILD_TESTS=OFF -DROCWMMA_BUILD_SAMPLES=OFF
```

Here are some other example project configurations:

Configuration	Command
Basic	CC=hipcc CXX=hipcc cmake -B<build_dir> .
Targeting MI100	CC=hipcc CXX=hipcc cmake -B<build_dir> . -DAMDGPU_TARGETS=gfx908:xnack-
Debug build	CC=hipcc CXX=hipcc cmake -B<build_dir> . -DCMAKE_BUILD_TYPE=Debug
Build without rocBLAS(default on)	CC=hipcc CXX=hipcc cmake -B<build_dir> . -DROCWMMA_VALIDATE_WITH_ROCBLAS=OFF -DROCWMMA_BENCHMARK_WITH_ROCBLAS=OFF

After configuration, build with

```
cmake –build <build_dir> – -j
```

2.4.5 Build library + samples

To build library and samples, run the following comomand :

CC=hipcc CXX=hipcc cmake -B<build_dir> . -DROCWMMA_BUILD_TESTS=OFF -DROCWMMA_BUILD_SAMPLES=ON

After configuration, build with

cmake –build <build_dir> – -j

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

executable name	description
simple-gemm	a simple GEMM operation [D = alpha * (A x B) + beta * C] using rocWMMA API
sgemv	a simple GEMV operation [y = alpha * (A) * x + beta * y] using rocWMMA API
simple-dlrm	a simple DLRM operation using rocWMMA API

2.4.6 Build library + tests

rocWMMA has several test suites that can be built:

- DLRM tests
- GEMM tests
- Unit tests

DLRM tests cover the dot product interactions between embeddings used in DLRM.

GEMM tests cover block-wise Generalized Matrix Multiplication (GEMM) implemented with rocWMMA.

Unit tests cover various aspects of rocWMMA API and internal functionality.

rocWMMA can build both validation and benchmark tests. The library uses CPU or rocBLAS methods for validation (where available) and benchmark comparisons based on the provided project option. By default, the project is linked against rocBLAS for validating results. Minimum ROCBLAS library version requirement is 2.39.0 for ROCm 4.3.0

To build library and tests, run the following command :

CC=hipcc CXX=hipcc cmake -B<build_dir> .

After configuration, build with

cmake –build <build_dir> – -j

The tests in <build_dir> contains executables in the table below.

executable name	description
dlrm/dlrm_dot_test-*	a DLRM implementation using rocWMMA API
dlrm/dlrm_dot_lds_test-*	a DLRM implementation using rocWMMA API with LDS shared memory
gemm/mma_sync_test-*	a simple GEMM operation [D = alpha * (A x B) + beta * C] using rocWMMA API
gemm/mma_sync_multi_test-*	a modified GEMM operation, each wave targets a sub-grid of output blocks using rocWMMA API
gemm/mma_sync_multi_adaptive_test-*	adaptive version of mma_sync_multi_test-*
gemm/mma_sync_multi_lds_test-*	a modified GEMM operation, each wave targets a sub-grid of output blocks using LDS memory and rocWMMA API and wave-level collaboration
gemm/mma_sync_multi_lds_adhoc_test-*	ad-hoc version of mma_sync_multi_lds_test-*
gemm/mma_sync_coop_wg_test-*	a modified GEMM operation, each wave targets a sub-grid of output blocks using LDS memory and rocWMMA API and workgroup-level collaboration
gemm/mma_sync_coop_wg_adhoc_test-*	ad-hoc version of mma_sync_coop_wg_test-*
gemm/barrier_test-*	a simple GEMM operation with wave synchronization
unit/fill_fragment_test	tests fill_fragment API function
unit/load_store_matrix_sync_test	tests load_matrix_sync and store_matrix_sync API functions
unit/load_store_matrix_coop_sync_test	tests load_matrix_coop_sync and store_matrix_coop_sync API functions
unit/contamination_test	tests against contamination of pristine data for loads and stores
unit/layout_test	tests accuracy of internal matrix layout patterns
unit/mapping_util_test	tests mapping utilities used in rocWMMA implementations
unit/vector_iterator_test	tests internal vector storage implementation

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

*= bench: executables that measure kernel execution speeds and may compare against those of rocBLAS references.

2.4.7 Build library + Tests + Assembly

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

```
CC=hipcc CXX=hipcc cmake -B<build_dir> . -DROCWMMA_BUILD_ASSEMBLY=ON
```

After configuration, build with

```
cmake –build <build_dir> – -j
```

The assembly folder in <build_dir> contains assembly generation of test executables in the format [test_executable_name.s]

CHAPTER

THREE

PROGRAMMER'S GUIDE

3.1 Library Source Code Organization

The rocWMMA code is split into four major parts:

- The *library* directory contains all source code for the library.
- The *samples* directory contains real-world use-cases of the rocWMMA API.
- The *test* directory contains all validation, performance and unit tests of rocWMMA API.
- Infrastructure

3.1.1 The *library* directory

3.1.1.1 library/include/rocwmma/

Contains C++ include files for the rocWMMA API. These files also contain Doxygen comments that document the API.

3.1.1.2 library/include/internal

Internal include files for:

- Type support
- Input / output configuration, shapes and traits
- Layout
- Mapping Utility
- Packing and unpacking
- Conversion and broadcasting
- Load and store
- Matrix multiply-accumulate
- Cooperative load and store
- Threadblock synchronization
- Utility code

3.1.2 The *samples* directory

3.1.2.1 samples/sgemmv.cpp

sample code for calling Simple matrix multiply-accumulate with a vector demonstration, without LDS and no transpose.

3.1.2.2 samples/simple_gemm.cpp

Sample code for calling Simple GEMM algorithm demonstration without LDS memory usage and no transpose.

3.1.2.3 samples/simple_dlrm.cpp

Sample code for calling Simple Deep Learning Recommendation Model (DLRM) for machine learning.

3.1.2.4 samples/common.hpp

Common code used by all the above rocWMMA samples files.

3.1.3 The *test* directory

3.1.3.1 test/bin

Script to generate benchmark plots from the gtest output dumps of benchmark tests of rocWMMA.

3.1.3.2 test/dlrm

Test code for various strategies of DLRM application. This test is used to validate dlrm functions using rocWMMA API.

3.1.3.3 test/gemm

Test Code for various strategies of GEMM application. This test is used to validate and benchmark GEMM functions using rocWMMA API.

3.1.3.4 test/unit

Test code for testing the basic functional units of rocWMMA library.

3.1.4 Infrastructure

- CMake is used to build and package rocWMMA. There are CMakeLists.txt files throughout the code.
- Doxygen/Breathe/Sphinx/ReadTheDocs are used to produce documentation. Content for the documentation is from:
  - Doxygen comments in include files in the directory library/include
  - files in the directory docs/source.
- Jenkins is used to automate Continuous Integration testing.

- clang-format is used to format C++ code.

CHAPTER

FOUR

CONTRIBUTOR'S GUIDE

4.1 License Agreement

1. The code I am contributing is mine, and I have the right to license it.
2. By submitting a pull request for this project I am granting you a license to distribute said code under the MIT License for the project.

4.2 Pull-request guidelines

Our code contriubtion guidelines closely follows the model of GitHub pull-requests. The rocWMMA repository follows a workflow which dictates a /master branch where releases are cut, and a /develop branch which serves as an integration branch for new code. Pull requests should:

- target the **develop** branch for integration
- ensure code builds successfully.
- do not break existing test cases
- new functionality will only be merged with new unit tests
- new unit tests should integrate within the existing googletest framework.
- tests must have good code coverage
- code must also have benchmark tests, and performance must approach the compute bound limit or memory bound limit.

4.3 StyleGuide

This project follows the CPP Core guidelines, with few modifications or additions noted below. All pull-requests should in good faith attempt to follow the guidelines stated therein, but we recognize that the content is lengthy. Below we list our primary concerns when reviewing pull-requests.

4.3.1 Interface

- Library code should use C++14
- Our minimum supported compiler is hipcc 4.4
- Avoid CamelCase
- This rule applies specifically to publicly visible APIs, but is also encouraged (not mandated) for internal code

4.3.2 Philosophy

- P.2: Write in ISO Standard C++14 (especially to support windows, linux and macos plaforms)
- P.5: Prefer compile-time checking to run-time checking

4.3.3 Implementation

- SF.1: Use a `.cpp` suffix for code files and an `.hpp` suffix for interface files if your project doesn't already follow another convention
- SF.5: A `.cpp` file must include the `.hpp` file(s) that defines its interface
- SF.7: Don't put a global `using`-directive in a header file
- SF.8: Use `#include` guards for all `.hpp` files
- SF.21: Don't use an unnamed (anonymous) `namespace` in a header
- SL.10: Prefer using `std::array` or `std::vector` instead of a C array
- C.9: Minimize the exposure of class members
- F.3: Keep functions short and simple
- F.21: To return multiple 'out' values, prefer returning a `std::tuple`
- R.1: Manage resources automatically using RAII (this includes `std::unique_ptr` & `std::shared_ptr`)
- ES.11: Use `auto` to avoid redundant repetition of type names
- ES.20: Always initialize an object
- ES.23: Prefer the `{}` initializer syntax
- CP.1: Assume that your code will run as part of a multi-threaded program
- I.2: Avoid global variables

4.3.4 Format

C++ code is formatted using `clang-format`. To run clang-format use the version in the `/opt/rocm/llvm/bin` directory. Please do not use your system's built-in `clang-format`, as this may be an older version that will result in different results.

To format a file, use:

```
/opt/rocm/llvm/bin/clang-format -style=file -i <path-to-source-file>
```

To format all files, run the following script in rocWMMA directory:

```
#!/bin/bash
git ls-files -z *.cc *.cpp *.h *.hpp *.cl *.h.in *.hpp.in *.cpp.in | xargs -0 /opt/rocm/
↪llvm/bin/clang-format -style=file -i
```

Also, githooks can be installed to format the code per-commit:

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

CHAPTER

FIVE

DISCLAIMER

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

- AMD, the AMD Arrow logo, Radeon, Ryzen, Epyc, and combinations thereof are trademarks of Advanced Micro Devices, Inc.
- Google(R) is a registered trademark of Google LLC.
- PCIe(R) is a registered trademark of PCI-SIG Corporation.
- Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.
- Ubuntu and the Ubuntu logo are registered trademarks of Canonical Ltd.
- Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

INDEX

R

V