Composable Kernel Documentation

Release 1.3.0

Advanced Micro Devices, Inc.

Nov 18, 2026

INSTALL

1 Install Composable Kernel **3**
- 1.1 Install the ROCm Core SDK . . . 3
- 1.2 Install the Composable Kernel package on Linux . . . 3
- 1.3 Install a nightly build . . . 4

2 Build and install Composable Kernel from source **5**
- 2.1 Prerequisites . . . 5
- 2.2 Build and install . . . 5

3 Composable Kernel structure **9**

4 Composable Kernel mathematical basis **11**

5 Intrawave and interwave scheduling with CK Tile **13**

6 CK Tile conceptual documentation table of contents **15**
- 6.1 CK Tile Conceptual Documentation . . . 15
- 6.2 Introduction and Motivation - Why Tile Distribution Matters . . . 17
- 6.3 Buffer Views - Raw Memory Access . . . 23
- 6.4 Tensor Views - Multi-Dimensional Structure . . . 29
- 6.5 Tile Distribution - The Core API . . . 34
- 6.6 Coordinate Systems - The Mathematical Foundation . . . 41
- 6.7 Terminology Reference - Key Concepts and Definitions . . . 47
- 6.8 Tensor Adaptors - Chaining Transformations . . . 56
- 6.9 Individual Transform Operations . . . 62
- 6.10 Tensor Descriptors - Complete Tensor Specifications . . . 69
- 6.11 Tile Window - Data Access Gateway . . . 74
- 6.12 LoadStoreTraits - Memory Access Optimization Engine . . . 83
- 6.13 Space-Filling Curves - Optimal Memory Traversal . . . 91
- 6.14 Static Distributed Tensor . . . 99
- 6.15 Convolution Implementation with CK Tile . . . 106
- 6.16 Advanced Coordinate Movement . . . 115
- 6.17 Load Data Share Index Swapping . . . 123
- 6.18 Memory Swizzling with Morton Ordering . . . 129
- 6.19 Tensor Coordinates . . . 138
- 6.20 Sweep Tile . . . 145
- 6.21 Encoding Internals . . . 152
- 6.22 Thread Mapping - Connecting to Hardware . . . 159
- 6.23 CK Tile Hardware Documentation . . . 167

7 Composable Kernel examples and tests **183**

8 Composable Kernel precision support **185**
8.1 Supported data types overview . . . 185
8.2 Related references . . . 185

9 Composable Kernel supported scalar data types **187**

10 Composable Kernel custom data types **189**

11 Composable Kernel vector template utilities **191**

12 Composable Kernel wrapper **193**

13 Composable Kernel glossary **195**

14 Contributing to Composable Kernel **201**
14.1 Reporting issues . . . 201
14.2 Contributing to the codebase . . . 201

15 License **203**

Index **205**

The Composable Kernel library provides a programming model for writing performance critical kernels for machine learning workloads across multiple architectures including GPUs and CPUs, through general purpose kernel languages such as HIP C++.

The Composable Kernel project is located in https://github.com/ROCm/rocm-libraries/tree/develop/projects/composablekernel.

Install

- *Install Composable Kernel*
- *Build from source*

Conceptual

- *Composable Kernel structure*
- *Composable Kernel mathematical basis*
- *CK Tile intrawave and interwave scheduling*
- *CK Tile conceptual documentation*

Tutorials

- *Composable Kernel examples and tests*

Reference

- *Composable Kernel precision support*
- *Composable Kernel supported scalar types*
- *Composable Kernel custom types*
- *Composable Kernel vector utilities*
- *Composable Kernel wrapper*
- *Composable Kernel glossary*

To contribute to the documentation refer to Contributing to ROCm.

You can find licensing information on the Licensing page.

CHAPTER

ONE

INSTALL COMPOSABLE KERNEL

Before you begin, verify that your system is supported. For more information, see ROCm Core SDK components.

For advanced workflows, source builds, or custom configurations, see *Build and install Composable Kernel from source*.

1.1 Install the ROCm Core SDK

Composable Kernel (CK) is included with the ROCm Core SDK on Linux and Windows. For the complete Core SDK installation, use the `amdrocm-core-sdk` package. See Install AMD ROCm for more information.

1.2 Install the Composable Kernel package on Linux

Alternatively, you can use the `amdrocm-ck` package to install Composable Kernel without the full ROCm Core SDK.

1. Complete the ROCm installation prerequisites to install dependencies and configure GPU access permissions.
2. Install the Composable Kernel package that matches your desired ROCm version. Package names use the following format:

   ```
   amdrocm-ck<rocm_version>-<llvm_target>
   ```

   Where:

   - `<rocm_version>` is the ROCm Core SDK version to install. Omit this suffix to install the latest available version.
   - `<llvm_target>` (starting with `gfx`) is used if you are installing for a single AMD GPU architecture. Omit this suffix to install for all architectures at the cost of disk space.

   For example, to install the latest Composable Kernel development package release for supported GPU architectures:

   #### Debian-based distros

   ```
   sudo apt install amdrocm-ck<rocm_version>-<llvm_target>
   ```

   #### RHEL-based distros

   ```
   sudo dnf install amdrocm-ck<rocm_version>-<llvm_target>
   ```

SLES

```
sudo zypper install amdrocm-ck<rocm_version>-<llvm_target>
```

1.3 Install a nightly build

The TheRock build system also publishes nightly builds for the ROCm Core SDK and its components, including Composable Kernel. See Nightly release status for details.

> **Note**
>
> If you choose to install from a nightly build artifact, you'll need to set `HIP_PLATFORM=amd` and `LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/rocm/lib` after installation.

CHAPTER

TWO

BUILD AND INSTALL COMPOSABLE KERNEL FROM SOURCE

To build Composable Kernel (CK) as part of the ROCm Core SDK, see TheRock build instructions. TheRock is the recommended way to build ROCm components from source.

Alternatively, you can build Composable Kernel standalone using the following instructions.

2.1 Prerequisites

The following prerequisites are required to build and install Composable Kernel:

- CMake 3.21 or later
- Python 3.8 or later
- Git
- A C++ compiler from the ROCm install, typically `/opt/rocm/llvm/bin/clang++` or `hipcc` on Linux

Composable Kernel uses HIP to compile device code. Set `CMAKE_PREFIX_PATH` to the ROCm install prefix and `CMAKE_CXX_COMPILER` to the ROCm Clang or `hipcc` path when you configure the build.

Pre-built Docker images that bundle ROCm, CMake, and the LLVM toolchain are on Docker Hub.

2.2 Build and install

Before you begin, clone the Composable Kernel project.

Use sparse checkout when cloning the Composable Kernel project:

```
git clone --no-checkout --filter=blob:none https://github.com/ROCm/rocm-libraries.git
cd rocm-libraries
git sparse-checkout init --cone
git sparse-checkout set projects/composablekernel
```

Then use `git checkout` to check out the branch you need.

The develop branch is intended for users who want to preview new features or contribute to the Composable Kernel codebase.

If you don't intend to contribute to the codebase and won't be previewing features, use a branch that matches the version of ROCm installed on your system.

Create the `build` directory under `rocm-libraries/projects/composablekernel`:

```
cd projects/composablekernel
mkdir build
```

Change directory to the `build` directory and generate the makefile using the `cmake` command. Two build options are required:

- `CMAKE_PREFIX_PATH`: The ROCm installation path. ROCm is installed in `/opt/rocm` by default.
- `CMAKE_CXX_COMPILER`: The path to the Clang compiler. Clang is found at `/opt/rocm/llvm/bin/clang++` by default.

```
cd build
cmake ../. -D CMAKE_PREFIX_PATH="/opt/rocm" -D CMAKE_CXX_COMPILER="/opt/rocm/llvm/bin/
↪clang++" [-D<OPTION1=VALUE1> [-D<OPTION2=VALUE2>] ...]
```

Other build options are:

- `DISABLE_DL_KERNELS`: Set this to "ON" to not build deep learning (DL) and data parallel primitive (DPP) instances. DL and DPP instances are useful on architectures that don't support XDL or WMMA.
- `CK_USE_FP8_ON_UNSUPPORTED_ARCH`: Set to `ON` to build FP8 data type instances on gfx90a without native FP8 support.
- `GPU_TARGETS`: Target GPU architectures. The standard HIP variable. Composable Kernel forwards the list to HIP, so HIP's compatibility rules apply. List one architecture or a small set from the same family. Set this option to build the tests, examples, and tutorials. Enclose the list in quotation marks and separate entries with semicolons (`;`). For example, `cmake -D GPU_TARGETS="gfx908;gfx90a"`.
- `GPU_ARCHS`: Target GPU architectures. Composable Kernel-specific. Use this option to build the Composable Kernel library for architectures from different families. If you set `GPU_ARCHS`, Composable Kernel clears `GPU_TARGETS` before configuring HIP. Composable Kernel then builds only the library and skips the tests, examples, and tutorials. Enclose the list in quotation marks and separate entries with semicolons (`;`). For example, `cmake -D GPU_ARCHS="gfx908;gfx1100"`.
- `CMAKE_BUILD_TYPE`: The build type. Can be `None`, `Release`, `Debug`, `RelWithDebInfo`, or `MinSizeRel`. CMake uses `Release` by default.

Note

When both `GPU_TARGETS` and `GPU_ARCHS` are set, Composable Kernel uses `GPU_ARCHS` and clears `GPU_TARGETS`. When neither is set, Composable Kernel picks a default target list based on the detected HIP version. Composable Kernel drops the unsupported architectures `gfx900`, `gfx906`, and `gfx90c`, along with any target the installed compiler can't build for.

Build Composable Kernel using the generated makefile. With a default configuration, the build produces the Composable Kernel libraries, the `ckProfiler` benchmarking tool, the example binaries, the tutorial binaries, and the test binaries. The output is saved to `build/lib/` and `build/bin/`. The Composable Kernel headers stay in the source tree until `make install` copies them to the install location.

Note

A default Composable Kernel build can take an hour or more on a workstation. Most of the time goes to compiling operator instances, and build time scales with the number of GPU architectures you select. Limit `GPU_TARGETS` to your hardware to cut build time.

```
make -j20
```

The `-j` option runs build steps in parallel. For example, `-j20` runs up to twenty jobs at a time. Each parallel job can use several gigabytes of memory because Composable Kernel instance files expand large template instantiations, and link jobs use more memory than compile jobs. Pick a `-j` value that fits your available RAM, and lower it if compiles fail with out-of-memory errors.

> **Note**
>
> Don't run `-j` without a number. Bare `-j` launches an unbounded number of jobs and can exhaust memory.
>
> With Ninja, set `-D CK_PARALLEL_COMPILE_JOBS=N` and `-D CK_PARALLEL_LINK_JOBS=M` at configure time to cap compile and link jobs separately. These options have no effect with Make.

Install the Composable Kernel library:

```
make install
```

After running `make install`, the Composable Kernel files will be saved to the following locations:

- Library files: `/opt/rocm/lib/`
- Header files: `/opt/rocm/include/ck/` and `/opt/rocm/include/ck_tile/`
- Examples, tests, and ckProfiler: `/opt/rocm/bin/`

For information about ckProfiler, see the ckProfiler readme file.

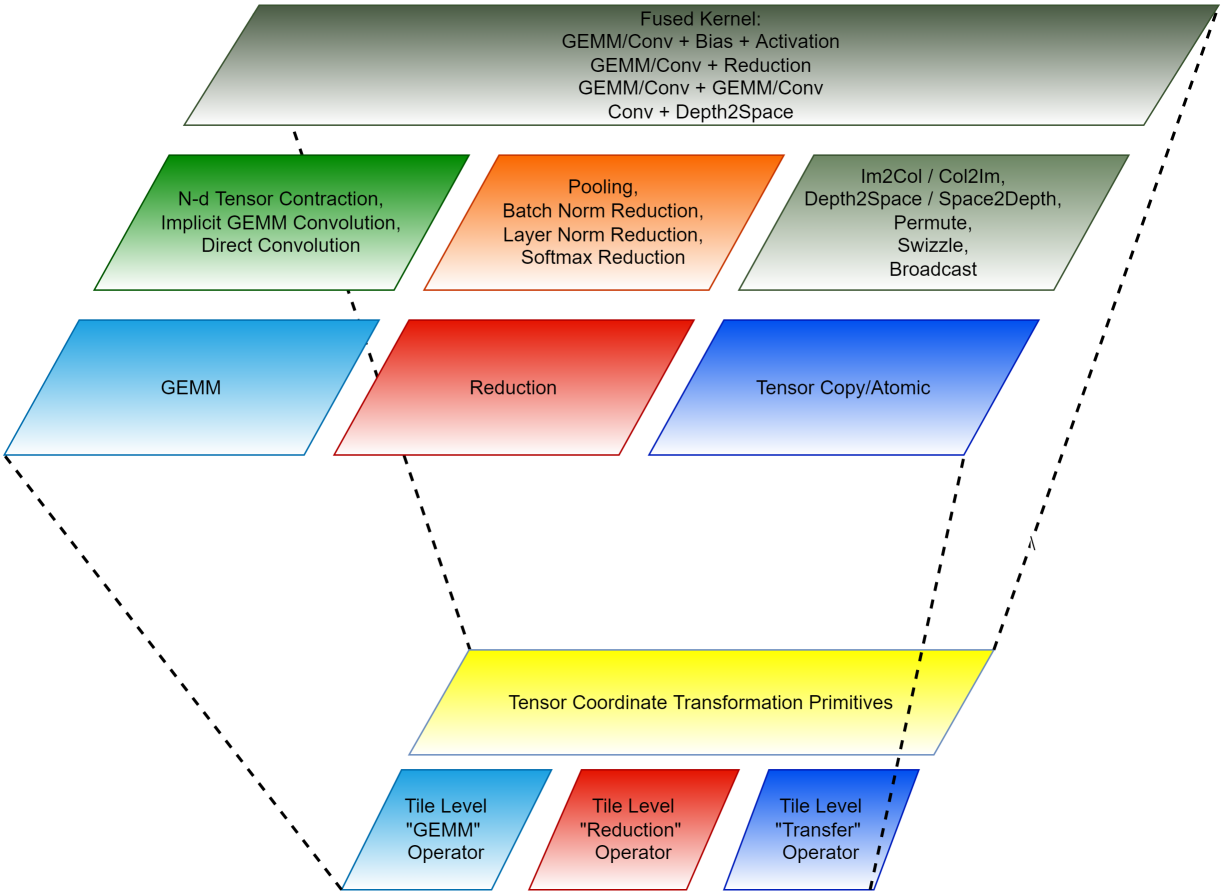
For information about running the examples and tests, see *Composable Kernel examples and tests*.

CHAPTER

THREE

COMPOSABLE KERNEL STRUCTURE

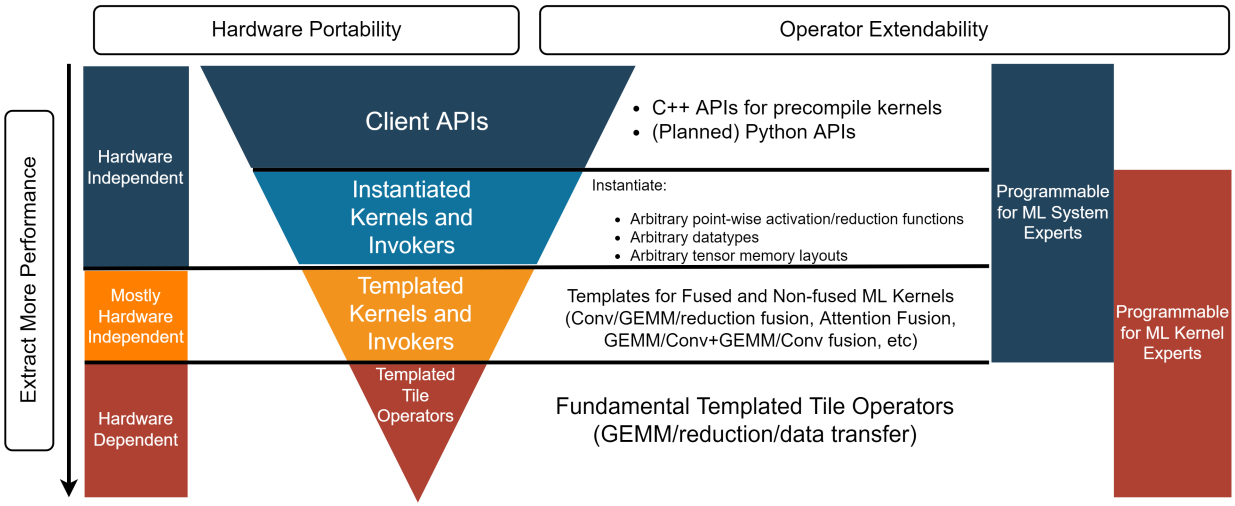
The Composable Kernel library uses a tile-based programming model and tensor coordinate transformation to achieve performance portability and code maintainability. Tensor coordinate transformation is a complexity reduction technique for complex machine learning operators.



The Composable Kernel library consists of four layers:

- a templated tile operator layer
- a templated kernel and invoker layer
- an instantiated kernel and invoker layer
- a client API layer.

A wrapper component is included to simplify tensor transform operations.



CHAPTER

FOUR

COMPOSABLE KERNEL MATHEMATICAL BASIS

This is an introduction to the math which underpins the algorithms implemented in Composable Kernel.

For vectors $x^{(1)}, x^{(2)}, \ldots, x^{(T)}$ of size B you can decompose the softmax of concatenated $x = [x^{(1)} \mid \ldots \mid x^{(T)}]$ as,

$$m(x) = m([x^{(1)} \mid \ldots \mid x^{(T)}]) = \max(m(x^{(1)}), \ldots, m(x^{(T)})) \tag{4.1}$$

$$f(x) = [\exp(m(x^{(1)}) - m(x))f(x^{(1)}) \mid \ldots \mid \exp(m(x^{(T)}) - m(x))f(x^{(T)})] \tag{4.2}$$

$$z(x) = \exp(m(x^{(1)}) - m(x))\ z(x^{(1)}) + \ldots + \exp(m(x^{(T)}) - m(x))\ z(x^{(1)}) \tag{4.3}$$

$$\text{softmax}(x) = f(x)\ /\ z(x) \tag{4.4}$$

where $f(x^{(j)}) = \exp(x^{(j)} - m(x^{(j)}))$ is of size B and $z(x^{(j)}) = f(x_1^{(j)}) + \ldots + f(x_B^{(j)})$ is a scalar.

For a matrix X composed of $T_r \times T_c$ tiles, X_{ij}, of size $B_r \times B_c$ you can compute the row-wise softmax as follows.

For j from 1 to T_c, and i from 1 to T_r calculate,

$$\tilde{m}_{ij} = \text{rowmax}(X_{ij}) \tag{4.5}$$

$$\tilde{P}_{ij} = \exp(X_{ij} - \tilde{m}_{ij}) \tag{4.6}$$

$$\tilde{z}_{ij} = \text{rowsum}(P_{ij}) \tag{4.7}$$

$$\tag{4.8}$$

If $j = 1$, initialize running max, running sum, and the first column block of the output,

$$m_i = \tilde{m}_{i1} \tag{4.9}$$

$$z_i = \tilde{z}_{i1} \tag{4.10}$$

$$\tilde{Y}_{i1} = (\tilde{z}_{ij})^{-1}\tilde{P}_{i1} \tag{4.11}$$

Else if $j > 1$,

1. Update running max, running sum and column blocks $k = 1$ to $k = j - 1$

$$m_i^{new} = \max(m_i, \tilde{m}_{ij}) \tag{4.12}$$

$$z_i^{new} = \exp(m_i - m_i^{new})\ z_i + \exp(\tilde{m}_{ij} - m_i^{new})\ \tilde{z}_{ij} \tag{4.13}$$

$$Y_{ik} = (z_i^{new})^{-1}(z_i)\exp(m_i - m_i^{new})\ Y_{ik} \tag{4.14}$$

2. Initialize column block j of output and reset running max and running sum variables:

$$\tilde{Y}_{ij} = (z_i^{new})^{-1}\exp(\tilde{m}_{ij} - m_i^{new})\tilde{P}_{ij} \tag{4.15}$$

$$z_i = z_i^{new} \tag{4.16}$$

$$m_i = m_i^{new} \tag{4.17}$$

$$\tag{4.18}$$

CHAPTER

FIVE

INTRAWAVE AND INTERWAVE SCHEDULING WITH CK TILE

Two different scheduling pipelines are available to use with CK Tile's GEMM implementation.

The interwave and intrawave scheduling pipelines coordinate waves in K dimension accumulation loops. Whether to use the interwave or intrawave pipeline depends on whether the workload is memory-bound or compute-bound.

In interwave scheduling, the K dimension is separated into chunks. The same chunk is loaded into each wave. When the chunk has been loaded into all the waves, the same operation is run on the chunk.

Once all the waves have completed the operation, the next chunk is loaded into the waves.

Because all the waves are synchronized, memory accesses are coordinated, and the cache hit rate is optimized, interwave scheduling is best for memory-bound workloads.

In intrawave scheduling, the full K dimension is loaded into each wave. Each wave runs its own operation on the K dimension independently of the other waves, and without any synchronization with the other waves. The compute unit (CU) is responsible for interleaving the independent operations.

Because the CU has flexibility in scheduling operations, intrawave scheduling is best for compute-bound workloads.

An example of both interwave and intrawave scheduling can be found in `gemm_utils.hpp`, which is part of the GEMM with CK Tile example.

CHAPTER

SIX

CK TILE CONCEPTUAL DOCUMENTATION TABLE OF CONTENTS

6.1 CK Tile Conceptual Documentation

Welcome to the conceptual documentation for CK Tile, the core abstraction layer of Composable Kernel that enables efficient GPU programming through compile-time coordinate transformations and tile-based data distribution.

See the *CK Tile conceptual documentation table of contents* for the complete CK Tile documentation structure.

6.1.1 Overview

CK Tile provides a mathematical framework for expressing complex GPU computations through:

- **Automatic Memory Coalescing**: Ensures optimal memory access patterns without manual optimization
- **Thread Cooperation**: Coordinates work distribution across the GPU's hierarchical execution model
- **Zero-Overhead Abstractions**: Compile-time optimizations ensure no runtime performance penalty
- **Portable Performance**: Same code achieves high performance across different GPU architectures

6.1.2 Why CK Tile?

Traditional GPU programming requires manual management of:

- Thread-to-data mapping calculations
- Memory coalescing patterns
- Bank conflict avoidance
- Boundary condition handling

CK Tile automates all of these concerns through a unified abstraction that maps logical problem coordinates to physical GPU resources.

6.1.3 Learning Path

1. **Start Here**: *Introduction and Motivation - Why Tile Distribution Matters*

   The fundamental problems CK Tile solves and why it's essential for efficient GPU programming.

2. **Foundation**: *Buffer Views - Raw Memory Access*

   How CK Tile provides structured access to raw GPU memory across different address spaces.

3. **Multi-Dimensional Views**: *Tensor Views - Multi-Dimensional Structure*

   How to work with multi-dimensional data structures and memory layouts.

4. **Core API**: *Tile Distribution - The Core API*

   The tile distribution system that maps work to GPU threads.

5. **Mathematical Framework**: *Coordinate Systems - The Mathematical Foundation*

   The coordinate transformation system that powers CK Tile's abstractions.

6. **Reference**: *Terminology Reference - Key Concepts and Definitions*

   Glossary of all terms and concepts used in CK Tile.

6.1.4 Key Concepts at a Glance

Coordinate Spaces

- **P-space**: Processing element coordinates (thread, warp, block)
- **Y-space**: Local tile access patterns
- **X-space**: Physical tensor coordinates
- **D-space**: Linearized memory addresses

Core Components

- **BufferView**: Type-safe access to GPU memory
- **TileDistribution**: Automatic work distribution
- **TileWindow**: Efficient data loading/storing
- **Encoding**: Compile-time distribution specification

6.1.5 Quick Example

```
// Define how to distribute a 256x256 tile across threads
using Encoding = tile_distribution_encoding<
    sequence<>,                                 // No replication
    tuple<sequence<4,2,8,4>,                    // M dimension hierarchy
          sequence<4,2,8,4>>,                   // N dimension hierarchy
    tuple<sequence<1,2>, sequence<1,2>>,        // Thread mapping
    tuple<sequence<1,1>, sequence<2,2>>,        // Minor indices
    sequence<1,1,2,2>,                          // Y-space mapping
    sequence<0,3,0,3>                           // Y-space minor
>;

// Create distribution and load data
auto distribution = make_static_tile_distribution(Encoding{});
auto window = make_tile_window(tensor_view, tile_size, origin, distribution);
auto tile = window.load();

// Process tile efficiently
sweep_tile(tile, [](auto idx) { /* computation */ });
```

6.1.6 Next Steps

To dive deeper, start with *Introduction and Motivation - Why Tile Distribution Matters* to understand the motivation and core concepts behind CK Tile.

For practical examples, see the example/ck_tile directory in the Composable Kernel repository.

6.2 Introduction and Motivation - Why Tile Distribution Matters

6.2.1 Overview

The evolution of GPU computing has brought unprecedented computational power to modern applications, yet harnessing this power efficiently remains one of the most challenging aspects of high-performance computing. At the heart of this challenge lies a fundamental mismatch between how developers conceptualize algorithms and how GPU hardware executes them. While developers think in terms of mathematical operations on multi-dimensional data structures, GPUs operate through thousands of threads accessing memory in complex patterns that must satisfy stringent hardware constraints.

This conceptual gap manifests most acutely in memory access patterns. Modern GPUs achieve their high performance through massive parallelism, with thousands of threads executing simultaneously. However, this parallelism comes with a critical constraint: memory bandwidth. Despite continuous improvements in computational throughput, memory bandwidth has not scaled proportionally, creating what is often called the "memory wall." The efficiency with which threads access memory determines whether a GPU kernel achieves a few percent or near 100% of the hardware's theoretical performance.

The Composable Kernel (CK) framework addresses this challenge through its tile distribution system, a compile-time abstraction that automatically generates optimal memory access patterns while preserving the natural expression of algorithms. This documentation explores the mathematical foundations and practical implementation of tile distribution, demonstrating how it bridges the gap between algorithmic intent and hardware reality.

In this introduction, we establish the fundamental problems that tile distribution solves, explore why these problems are critical for GPU performance, and provide the conceptual framework necessary to understand the compile-time coordinate transformation system that powers CK's approach to efficient GPU computation.

6.2.2 The GPU Memory Problem

Why Random Memory Access is Slow

The architecture of modern GPUs represents a study in trade-offs. While these devices can execute thousands of threads simultaneously and perform trillions of floating-point operations per second, they remain fundamentally constrained by the physics of memory access. Understanding this constraint is crucial to appreciating why tile distribution is not merely an optimization technique but an essential component of high-performance GPU computing.

GPU memory systems are designed around the assumption of regular, predictable access patterns. The memory controller can service requests from 32 threads (a warp on AMD GPUs) in a single transaction when these threads access consecutive memory locations. This optimization, known as memory coalescing, can improve effective memory bandwidth by up to 32x compared to random access patterns. However, when threads within a warp access memory locations that are scattered throughout the address space, each access requires a separate memory transaction, reducing the effective bandwidth to a fraction of the theoretical maximum.

The impact extends beyond raw bandwidth. Modern GPUs employ cache hierarchies to reduce memory latency, but these caches are effective only when access patterns exhibit spatial or temporal locality. Random access patterns defeat these optimizations, causing frequent cache misses that expose the full latency of global memory access, which can be hundreds of cycles. During these stalls, the computational units sit idle, unable to hide the latency even with the GPU's massive thread count.

Furthermore, the GPU's Single Instruction, Multiple Thread (SIMT) execution model requires that all threads in a warp execute the same instruction at the same time. When threads access memory in unpredictable patterns, the memory controller cannot optimize the requests, leading to serialization of what should be parallel operations. This serialization effect compounds with each level of the memory hierarchy, from L1 cache through L2 cache to global memory, multiplying the performance impact.

The Thread Cooperation Challenge

The challenge of efficient thread cooperation becomes particularly evident when examining a fundamental operation like matrix multiplication. Consider a scenario where 256 threads must cooperate to multiply two matrices. The naive approach, where each thread computes one element of the output matrix, illustrates precisely why GPU programming requires compile-time abstractions.

```
// Inefficient: Random access pattern
__device__ void naive_matrix_multiply()
{
    int thread_id = threadIdx.x + blockIdx.x * blockDim.x;

    // Get this thread's output position
    int row = thread_id / MATRIX_WIDTH;
    int col = thread_id % MATRIX_WIDTH;

    // Each thread computes one element of C = A * B
    float result = 0.0f;
    for (int k = 0; k < MATRIX_WIDTH; k++)
    {
        // Random access pattern - threads in a warp access non-contiguous memory
        // Thread 0: A[0,0], A[0,1], A[0,2]...
        // Thread 1: A[1,0], A[1,1], A[1,2]...
        // These are far apart in memory!
        float a_element = global_memory_A[row * MATRIX_WIDTH + k];

        // Even worse for B - accessing column-wise causes strided access
        // Thread 0: B[0,0], B[1,0], B[2,0]...
        // Thread 1: B[0,1], B[1,1], B[2,1]...
        // Massive stride between accesses!
        float b_element = global_memory_B[k * MATRIX_WIDTH + col];

        result += a_element * b_element;
    }

    // Write result - adjacent threads write to adjacent locations (at least this is␣
↪good)
    global_memory_C[row * MATRIX_WIDTH + col] = result;
}
```

This seemingly straightforward implementation suffers from fundamental inefficiencies that stem from the mismatch between the algorithm's logical structure and the hardware's physical constraints. The memory access pattern is essentially random from the hardware's perspective, as adjacent threads access memory locations separated by large strides. This pattern prevents the memory controller from coalescing accesses, forcing it to issue separate transactions for each thread.

The lack of coordination between threads exacerbates the problem. While all threads in a warp execute the same instruction, they operate on completely different data with no sharing or reuse. This independence, which might seem desirable in traditional parallel programming, actually works against GPU architecture. The hardware cannot exploit any commonality in the access patterns, leading to severe underutilization of memory bandwidth.

Cache utilization suffers dramatically under this access pattern. Each thread traces a unique path through memory, with no overlap between threads' working sets. The L1 and L2 caches, designed to capture and exploit locality, instead thrash continuously as each thread's accesses evict data needed by others. The effective cache capacity approaches zero, exposing every memory access to the full latency of global memory.

Perhaps most critically, this approach fails to utilize the available memory bandwidth efficiently. Modern GPUs can achieve memory bandwidths exceeding 1 TB/s, but only when accesses are properly structured. The random access pattern of the naive implementation might achieve less than 10% of this theoretical maximum, effectively reducing a high-performance GPU to the performance level of a much simpler processor.

6.2.3 The Tile Distribution Solution

Structured Mapping from Logical to Physical Coordinates

The fundamental innovation of tile distribution lies in its approach to the memory access problem. Rather than attempting to optimize the naive access patterns after the fact, tile distribution provides a mathematical framework that generates optimized patterns from the outset. This framework establishes a structured mapping between logical coordinates and physical coordinates that respect hardware constraints.

The essence of tile distribution is the recognition that efficient GPU computation requires a careful choreography of thread cooperation. Instead of each thread operating independently, threads are organized into hierarchical groups that work together on tiles of data. This organization ensures that when threads access memory, they do so in patterns that the hardware can optimize.

```
// Efficient: Tile-based distribution using CK Tile
template<typename AType, typename BType, typename CType>
__device__ void tile_distributed_matrix_multiply()
{
    // 1. Define tile distribution encoding at compile time
    using Encoding = tile_distribution_encoding<
        sequence<>,                                  // No replication
        tuple<sequence<4, 2, 8, 4>,                  // M dimension hierarchy
              sequence<4, 2, 8, 4>>,                 // N dimension hierarchy
        tuple<sequence<1, 2>, sequence<1, 2>>,       // P to RH major
        tuple<sequence<1, 1>, sequence<2, 2>>,       // P to RH minor
        sequence<1, 1, 2, 2>,                        // Y to RH major
        sequence<0, 3, 0, 3>                         // Y to RH minor
    >;

    // 2. Create the distribution
    constexpr auto distribution = make_static_tile_distribution(Encoding{});

    // 3. Create tile window for efficient memory access
    auto tile_window = make_tile_window(
        tensor_view,
        window_lengths,
        origin,
        distribution
    );

    // 4. Load data with coalesced access pattern
    auto loaded_tensor = tile_window.load();

    // 5. Process tile data efficiently
    sweep_tile(loaded_tensor, [](auto y_indices) {
        auto value = loaded_tensor(y_indices);
        // ... efficient computation
    });
}
```

The transformation from inefficient to efficient memory access is profound. Where the naive implementation scattered memory requests across the address space, tile distribution ensures that adjacent threads access adjacent memory locations. This transformation happens through an advanced encoding system that captures the hierarchical nature of both the computation and the hardware.

The encoding shown above demonstrates the multi-level hierarchy that tile distribution employs. The sequence<4, 2, 8, 4> represents a four-level decomposition: four repetitions per thread, two warps per block, eight threads per warp, and four elements per vector operation. This hierarchical structure maps directly to the GPU's hardware organization, ensuring that each level of the hierarchy operates at maximum efficiency.

Memory access patterns become predictable and regular under tile distribution. The hardware's memory coalescing logic can now combine the requests from all threads in a warp into a single transaction, achieving the full memory bandwidth. The predictability extends beyond individual accesses to entire access sequences, enabling the hardware's prefetching mechanisms to anticipate and prepare data before it's needed.

Thread cooperation emerges naturally from the tile distribution structure. Threads within a warp work on adjacent data, enabling efficient data sharing through register shuffle operations. Warps within a block coordinate through shared memory, with access patterns that avoid bank conflicts. This cooperation transforms what was a collection of independent computations into a unified, efficient operation.

Cache utilization improves as well. The structured access patterns ensure that data loaded into cache by one thread is likely to be used by neighboring threads. Temporal locality emerges from the tile-based processing, where all operations on a tile complete before moving to the next tile. This locality transforms the cache from a liability into a high performance accelerator.

The scalability of tile distribution across different GPU architectures represents one of its most key features. The same high-level code can achieve near-optimal performance on GPUs with different numbers of compute units, different cache sizes, and different memory bandwidths. The compile-time nature of the encoding allows the compiler to generate architecture-specific optimizations while maintaining portable source code.

6.2.4 The Coordinate Mapping Insight

At the heart of tile distribution lies a profound mathematical insight: efficient GPU computation requires a systematic framework for mapping between different coordinate spaces. This framework transforms the complex problem of thread-to-data assignment into a series of well-defined mathematical transformations, each serving a specific purpose in the journey from abstract algorithm to concrete hardware execution.

The elegance of this approach emerges from its separation of concerns. Each coordinate space represents a distinct aspect of the computation, and the transformations between them encapsulate specific optimization strategies. This separation allows developers to reason about their algorithms in natural terms while the framework handles the complex mapping to efficient hardware execution patterns.

Thread Position Space (P-space) represents the physical organization of threads on the GPU. This space captures the hierarchical nature of GPU execution, from individual threads identified by their x and y coordinates within a block, to warps that execute in lockstep, to thread blocks that share resources. The coordinates in P-space—thread_x, thread_y, warp_id, and block_id—directly correspond to the hardware's execution model. Understanding P-space is crucial because it determines which threads can cooperate efficiently through shared memory and which threads will execute their memory accesses simultaneously.

Local Data Space (Y-space) embodies the algorithm's perspective on data organization. In this space, each thread reasons about its local portion of work using coordinates like y0, y1, y2, and y3. These coordinates are algorithm-specific and represent the natural way to index the data being processed. For matrix multiplication, Y-space might represent the local tile coordinates within a larger matrix. For convolution, it might represent the spatial dimensions and channels of a local receptive field. The beauty of Y-space is that it allows algorithms to be expressed in their most natural form, without concern for hardware-specific optimizations.

Global Position Space (X-space) serves as the bridge between algorithmic intent and physical reality. This space represents the actual global coordinates of data in the problem domain, such as the row and column indices in a matrix or the spatial coordinates in an image. X-space is where the distributed nature of the computation becomes explicit, as

each thread's local Y-space coordinates combine with its position in P-space to determine which global data elements it accesses.

Memory Address Space (D-space) represents the final destination: linearized memory addresses that the hardware actually uses. This space accounts for the fact that multi-dimensional data structures must ultimately be stored in linear memory. The transformation to D-space incorporates layout optimizations such as padding for alignment, interleaving for better cache utilization, and address space considerations for different memory types (global, shared, or constant memory).

The transformative power of tile distribution emerges from the composition of these mappings. The $\mathbf{P} + \mathbf{Y} \rightarrow \mathbf{X}$ transformation combines a thread's position with its local data coordinates to determine global data positions. This transformation encodes the distribution strategy, determining how work is partitioned across threads. The subsequent $\mathbf{X} \rightarrow \mathbf{D}$ transformation converts these logical positions into physical memory addresses, incorporating layout optimizations that ensure efficient memory access patterns.

The mathematical rigor of this framework enables critical optimizations. Because each transformation is well-defined and composable, the compiler can analyze the complete transformation chain and generate optimal code. The framework can automatically ensure memory coalescing by structuring the $P + Y \rightarrow X$ transformation appropriately. It can minimize bank conflicts in shared memory by carefully designing the $X \rightarrow D$ mapping. Most importantly, it can adapt these optimizations to different hardware architectures by adjusting the transformation parameters while keeping the high-level algorithm description unchanged.

6.2.5 What's Coming Next

Having established the fundamental motivation for tile distribution and its coordinate mapping framework, this documentation now embarks on a systematic journey through the complete CK Tile system. This journey is carefully structured to build understanding layer by layer, starting from the most basic abstractions and progressing to advanced optimization techniques.

The foundation of the exploration begins with raw memory access through *Buffer Views - Raw Memory Access*, the fundamental abstraction that provides type-safe, address-space-aware access to GPU memory. Understanding BufferView is crucial because it establishes the patterns and principles that permeate the entire CK Tile system. From there, it progresses to *Tensor Views - Multi-Dimensional Structure*, which adds multi-dimensional structure to raw memory, enabling natural expression of algorithms while maintaining the efficiency of the underlying buffer operations.

With these foundational concepts established, the documentation delves into the *Coordinate Systems - The Mathematical Foundation* that powers tile distribution. This engine implements the mathematical framework that have been introduced, providing compile-time transformations between P-space, Y-space, X-space, and D-space. Understanding these transformations at a deep level enables developers to reason about performance implications and design custom distribution strategies for novel algorithms. The *Individual Transform Operations* and *Tensor Adaptors - Chaining Transformations* provide the building blocks for these transformations.

The high-level *Tile Distribution - The Core API* APIs represent the culmination of these lower-level abstractions. These APIs provide an accessible interface for common patterns while exposing enough flexibility for advanced optimizations. Through concrete examples and detailed explanations, the documentation will demonstrate how to leverage these APIs to achieve near-optimal performance across a variety of computational patterns. The *Tile Window - Data Access Gateway* abstraction provides the gateway for efficient data access.

The exploration of coordinate systems goes beyond the basic P, Y, X, D framework to encompass advanced topics such as multi-level tiling, replication strategies, and specialized coordinate systems for specific algorithm classes. The *Encoding Internals* reveals the mathematical foundations, while *Thread Mapping - Connecting to Hardware* shows how these abstractions map to hardware. This comprehensive treatment ensures that developers can handle not just common cases but also novel algorithms that require custom distribution strategies.

The implementation details reveal the template metaprogramming techniques that enable CK Tile's zero-overhead abstractions. Topics like *Tensor Descriptors - Complete Tensor Specifications*, *LoadStoreTraits - Memory Access Optimization Engine*, and *Static Distributed Tensor* show how these abstractions achieve zero overhead. By understanding

these implementation strategies, advanced developers can extend the framework, contribute optimizations, and debug performance issues at the deepest level.

The connection between abstract coordinate transformations and concrete hardware thread mapping represents a critical piece of the puzzle. The documentation will examine how logical thread organizations map to physical GPU resources, how to avoid common pitfalls like bank conflicts (see *Understanding AMD GPU LDS and Bank Conflicts* and *Load Data Share Index Swapping*) and divergent execution, and how to structure computations for maximum hardware utilization. The *CK Tile Hardware Documentation* section provides deep dives into architecture-specific optimizations.

Finally, the advanced topics section explores cutting-edge optimization techniques, including *Space-Filling Curves - Optimal Memory Traversal* for optimal memory traversal, *Sweep Tile* for clean iteration patterns, and practical examples like *Convolution Implementation with CK Tile* and *A Block GEMM on MI300*. These topics prepare developers to push the boundaries of GPU performance and contribute to the ongoing evolution of high-performance computing.

6.2.6 Summary

The journey through this introduction has revealed tile distribution as a fundamental paradigm shift in how GPU programming is approached. By establishing a mathematical framework for coordinate transformation, tile distribution bridges the gap between algorithmic elegance and hardware efficiency.

The significance of this approach extends beyond mere performance optimization. Tile distribution enables developers to express algorithms in their natural mathematical form while achieving performance that approaches the theoretical limits of the hardware. This reconciliation of abstraction and efficiency has been a goal of high-performance computing, and tile distribution provides a step towards this goal.

The structured, predictable mappings between logical and physical coordinates that tile distribution provides yield multiple benefits. Efficient memory access emerges naturally from the framework, with coalesced access patterns and cache-friendly layouts arising from the mathematical structure rather than manual optimization. Thread cooperation becomes an inherent property of the system, with the distribution encoding automatically organizing threads into efficient collaborative patterns. The scalability across different hardware architectures demonstrates the power of abstraction—the same high-level code achieves near-optimal performance whether running on a small mobile GPU or a massive datacenter accelerator.

Perhaps most importantly, tile distribution provides a predictable optimization framework grounded in mathematical principles. Performance characteristics can be analyzed and predicted based on the transformation structure, enabling systematic optimization rather than trial-and-error tuning. This predictability transforms GPU optimization from an art practiced by a few experts into a science accessible to a broader community of developers.

The systematic mapping through P-space, Y-space, X-space, and D-space provides a mental model that clarifies the entire GPU computation process. This model enables developers to reason about their code at multiple levels of abstraction simultaneously, understanding both the high-level algorithmic behavior and the low-level hardware execution patterns.

As the documentation dives deeper into the implementation details, starting with the foundational BufferView abstraction, it is important to remember that each component serves the larger purpose of enabling efficient, scalable GPU computation. The journey from raw memory to advanced tile distributions mirrors the evolution of GPU programming itself, from low-level, hardware-specific optimizations to high-level, portable abstractions that preserve efficiency.

By providing a framework for achieving optimal memory access patterns, tile distribution enables developers to take advantage of the computing power of GPUs without having to know the details of the underlying architecture.

6.2.7 Next Steps

Continue to *Buffer Views - Raw Memory Access* to start building your understanding from the ground up.

6.3 Buffer Views - Raw Memory Access

6.3.1 Overview

At the foundation of the CK Tile system lies BufferView, a compile-time abstraction that provides structured access to raw memory regions within GPU kernels. This serves as the bridge between the hardware's physical memory model and the higher-level abstractions that enable efficient GPU programming. BufferView encapsulates the complexity of GPU memory hierarchies while exposing a unified interface that works seamlessly across different memory address spaces including global memory shared across the entire device, local data share (LDS) memory shared within a workgroup, or the ultra-fast register files private to each thread.

BufferView serves as the foundation for *Tensor Views - Multi-Dimensional Structure*, which add multi-dimensional structure on top of raw memory access. Understanding BufferView is essential before moving on to more complex abstractions like *Tile Distribution - The Core API* and *Tile Window - Data Access Gateway*.

By providing compile-time knowledge of buffer properties through template metaprogramming, BufferView enables the compiler to generate optimal machine code for each specific use case. This zero-overhead abstraction ensures that the convenience of a high-level interface comes with no runtime performance penalty.

One of BufferView's most important features is its advanced handling of out-of-bounds memory access. Unlike CPU programming where such accesses typically result in segmentation faults or undefined behavior, GPU programming must gracefully handle cases where threads attempt to access memory beyond allocated boundaries. BufferView provides configurable strategies for these scenarios, where developers can choose between returning either numerical zero values or custom sentinel values for invalid accesses. This flexibility is important for algorithms that naturally extend beyond data boundaries, such as convolutions with padding or matrix operations with non-aligned dimensions.

The abstraction extends beyond simple memory access to encompass both scalar and vector data types. GPUs achieve their highest efficiency when loading or storing multiple data elements in a single instruction. BufferView seamlessly supports these vectorized operations, automatically selecting the appropriate hardware instructions based on the data type and access pattern. This capability transforms what would be multiple memory transactions into single, efficient operations that fully utilize the available memory bandwidth.

BufferView also incorporates AMD GPU-specific optimizations that leverage unique hardware features. The AMD buffer addressing mode, for instance, provides hardware-accelerated bounds checking that ensures memory safety without the performance overhead of software-based checks. Similarly, BufferView exposes atomic operations that are crucial for parallel algorithms requiring thread-safe updates to shared data structures. These hardware-specific optimizations are abstracted behind a portable interface, ensuring that code remains maintainable while achieving optimal performance.

Memory coherence and caching policies represent another layer of complexity that BufferView manages transparently. Different GPU memory spaces have different coherence guarantees and caching behaviors. Global memory accesses can be cached in L1 and L2 caches with various coherence protocols, while LDS memory provides workgroup-level coherence with specialized banking structures (see *Understanding AMD GPU LDS and Bank Conflicts* for details on avoiding bank conflicts). BufferView encapsulates these details, automatically applying the appropriate memory ordering constraints and cache control directives based on the target address space and operation type.

6.3.2 Address Space Usage Patterns

6.3.3 C++ Implementation

File: `include/ck_tile/core/tensor/buffer_view.hpp`

Basic Creation

By encoding critical properties such as buffer size and address space as template parameters, BufferView transforms what would traditionally be runtime decisions into compile-time constants. This design philosophy enables the compiler to perform aggressive optimizations, including constant propagation, loop unrolling, and instruction selection, that would be impossible with runtime parameters.

The use of compile-time constants extends beyond mere optimization. When the buffer size is encoded in the type system using constructs like number<8>{}, the compiler can statically verify that array accesses are within bounds, eliminate unnecessary bounds checks, and even restructure algorithms to better match the known data dimensions. This compile-time knowledge propagates through the entire computation, enabling optimizations at every level of the abstraction hierarchy.

The address space template parameter represents another crucial design decision. By making the memory space part of the type system, BufferView ensures that operations appropriate for one memory space cannot be accidentally applied to another. This type safety prevents common errors such as attempting atomic operations on register memory or using global memory synchronization primitives on local memory. The compiler enforces these constraints at compile time, transforming potential runtime errors into compile-time diagnostics.

```
#include <ck_tile/core/tensor/buffer_view.hpp>
#include <ck_tile/core/numeric/integral_constant.hpp>

// Create buffer view in C++
__device__ void example_buffer_creation()
{
    // Static array in global memory
    float data[8] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f};
    constexpr index_t buffer_size = 8;

// Create buffer view for global memory
// Template parameters: <AddressSpace>
auto buffer_view = make_buffer_view<address_space_enum::global>(
    data,        // pointer to data
    buffer_size  // number of elements
);

// Implementation detail: The actual C++ template is:
// template <address_space_enum BufferAddressSpace,
//           typename T,
//           typename BufferSizeType,
//           bool InvalidElementUseNumericalZeroValue = true,
//           amd_buffer_coherence_enum Coherence = amd_buffer_coherence_enum::coherence_
↪default>
// struct buffer_view

    // Alternative: Create with explicit type
    using buffer_t = buffer_view<float*, address_space_enum::global>;
    buffer_t explicit_buffer{data, number<buffer_size>{}};

    // Access properties at compile time
    constexpr auto size = buffer_view.get_buffer_size();
    constexpr auto space = buffer_view.get_address_space();

    // The buffer_view type encodes:
    // - Data type (float)
    // - Address space (global memory)
    // - Size (known at compile time for optimization)
    static_assert(size == 8, "Buffer size should be 8");
    static_assert(space == address_space_enum::global, "Should be global memory");
}
```

Out-of-Bounds Handling

Traditional approaches to bounds checking often involve conditional branches that can severely impact performance on GPU architectures, where divergent execution paths within a warp lead to serialization. BufferView's approach sidesteps this problem through two carefully designed modes that maintain performance while providing predictable behavior.

The Zero Value Mode leverages the mathematical property that zero often serves as a neutral element in computations. When an access falls outside the valid buffer range, this mode returns numerical zero without branching. This approach proves particularly effective for algorithms like convolution, where out-of-bounds accesses naturally correspond to zero-padding. The branchless implementation ensures that all threads in a warp follow the same execution path, maintaining the SIMD efficiency that is crucial for GPU performance.

The Custom Value Mode extends this concept by letting developers specify arbitrary sentinel values for invalid accesses. This flexibility accommodates algorithms that require specific values for boundary conditions, such as using negative infinity for maximum operations or special markers for missing data. The implementation maintains the same branchless characteristics, using conditional move instructions or predicated execution to avoid divergent control flow.

```
// Basic buffer view creation with automatic zero for invalid elements
void basic_creation_example() {
    // Create data array
    constexpr size_t buffer_size = 8;
    float data[buffer_size] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f};

    // Create global memory buffer view
    auto buffer_view = make_buffer_view<address_space_enum::global>(data, buffer_size);
}

// Custom invalid value mode
void custom_invalid_value_example() {
    constexpr size_t buffer_size = 8;
    float data[buffer_size] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f};
    float custom_invalid = 13.0f;

    // Create buffer view with custom invalid value
    auto buffer_view = make_buffer_view<address_space_enum::global>(
        data, buffer_size, custom_invalid);
}
```

6.3.4 Get Operations

Scalar Access

The get operations in BufferView form the cornerstone of memory access patterns in CK Tile. These operations embody a advanced understanding of GPU memory systems and the patterns that lead to optimal performance. The scalar access interface incorporates multiple layers of optimization and safety mechanisms that work together to provide both performance and correctness.

The parameter structure of scalar access operations reflects careful design choices aimed at maximizing flexibility while maintaining efficiency. The base index parameter `i` represents the primary offset into the buffer, expressed in terms of elements of type T rather than raw bytes. This type-aware indexing prevents common errors related to pointer arithmetic and ensures that vector types are handled correctly. The additional `linear_offset` parameter provides fine-grained control over the final access location, enabling complex access patterns without requiring expensive index calculations in the kernel code.

The `is_valid_element` parameter provides a solution to conditional memory access. Rather than using traditional if-statements that would cause warp divergence, this boolean parameter enables predicated execution where the memory

access occurs unconditionally but the result is conditionally used. This approach maintains uniform control flow across all threads in a warp, preserving the SIMD execution model that is fundamental to GPU performance.

The invalid value modes provide a mechanism for handling the boundary conditions that arise in parallel algorithms. When `InvalidElementUseNumericalZeroValue` is set to true, the system returns zero for any invalid access, whether due to the `is_valid_element` flag or out-of-bounds indexing. This mode is important for algorithms where zero serves as a natural extension value, such as in image processing with zero-padding or sparse matrix operations where missing elements are implicitly zero.

The custom invalid value mode, activated when `InvalidElementUseNumericalZeroValue` is false, offers additional flexibility for algorithms with specific boundary requirements. This mode returns a user-specified value for invalid accesses, accommodating use cases such as sentinel values in sorting algorithms, infinity values in optimization problems, or special markers in data processing pipelines. The implementation ensures that this flexibility comes without performance penalty, using the same branchless execution strategies as the zero mode.

Out-of-bounds handling leverages AMD GPU hardware capabilities to provide safety with minimal impact to performance. When AMD buffer addressing is enabled, the hardware automatically clamps memory accesses to valid ranges, preventing the segmentation faults that would occur on CPU systems. This hardware-assisted bounds checking operates at wire speed, adding no overhead to the memory access path while ensuring that kernels cannot corrupt memory outside their allocated regions.

Vector Access

Vector memory operations represent one of the most critical optimizations available in modern GPU programming, and BufferView's vector access interface exposes this capability. By using template parameters to specify vector types through constructs like `ext_vector_t<float, N>`, the interface enables compile-time selection of optimal load and store instructions that can transfer multiple data elements in a single memory transaction. This vectorization is crucial for *LoadStoreTraits - Memory Access Optimization Engine*, which automatically selects optimal access patterns.

The significance of vector operations extends beyond bandwidth improvements. GPUs are designed with wide memory buses that can transfer 128, 256, or even 512 bits per transaction. When scalar operations access only 32 bits at a time, they utilize only a fraction of this available bandwidth. Vector operations align with these wide buses, enabling full bandwidth utilization and reducing the total number of memory transactions required.

The implementation of vector access maintains the same parameter structure as scalar operations, providing consistency across the API while automatically handling the complexities of multi-element transfers. The system manages alignment requirements, ensures that vector loads and stores use the optimal hardware instructions, and handles cases where vector operations extend beyond buffer boundaries. This transparent handling of edge cases allows developers to use vector operations confidently without manual boundary checks or special-case code for partial vectors.

Scalar vs Vectorized Memory Access

Understanding BufferView Indexing

C++ Get Operations

```
__device__ void example_get_operations()
{
    // Create buffer view
    float data[8] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f};
    auto buffer_view = make_buffer_view<address_space_enum::global>(data, 8);

    // Simple get - compile-time bounds checking when possible
    auto value_buf = buffer_view.template get<float>(0,1,true); //get the buffer from␣
↪the buffer view
    float value = value_buf.get(0); //get the value from the buffer
```

(continues on next page)

(continued from previous page)

```
    // Get with valid flag - branchless conditional access
    bool valid_flag = false;
    value_buf = buffer_view.template get<float>(0,1,valid_flag);
    value = value_buf.get(0);
    // Returns 0 valid_flag is false

    // vectorized get
    using float2 = ext_vector_t<float, 2>;
    auto vector_buf = buffer_view.template get<float2>(0, 0, true);
    // Loads 2 floats in a single instruction
    float val1 = vector_buf.get(0);
    float val2 = vector_buf.get(1);
}
```

Custom Value Return Mode for OOB & Invalid Access

```
void scalar_get_operations_example() {

    // Create data array
    constexpr size_t buffer_size = 8;
    float data[buffer_size] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f};
    float custom_invalid = 13.0f;

    // Create global memory buffer view with zero invalid value mode (default)
    auto buffer_view = make_buffer_view<address_space_enum::global>(data, buffer_size,␣
↪custom_invalid);

    // Invalid element access with is_valid_element=false
    // Returns custom_invalid due to custom invalid value mode
    auto invalid_value = buffer_view.template get<float>(0, 0, false);
    printf("Invalid element: %.1f\n", invalid_value.get(0));

    // Out of bounds access - AMD buffer addressing handles bounds checking
    // Will return custom_invalid when accessing beyond buffer_size
    auto oob_value = buffer_view.template get<float>(0, 100, true);
    printf("Out of bounds: %.1f\n", oob_value.get(0));
}
```

> **Note**
>
> Partial Out Of Bound (OOB) access during vector reads will return 'junk' values for the OOB access. Zero or custom invalid value is only returned for complete invalid/OOB access, in other words, it is only returned when the first address of the vector is invalid.

6.3.5 Update Operations

```
void scalar_set_operations_example() {

    // Create data array
    constexpr size_t buffer_size = 8;
```

(continues on next page)

(continued from previous page)

```
    float data[buffer_size] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f};

    // Create global memory buffer view
    auto buffer_view = make_buffer_view<address_space_enum::global>(data, buffer_size);

    // Basic set: set<T>(i, linear_offset, is_valid_element, value)
    // Sets element at position i + linear_offset = 0 + 2 = 2
    buffer_view.template set<float>(0, 2, true, 99.0f);

    // Invalid write with is_valid_element=false (ignored)
    buffer_view.template set<float>(0, 3, false, 777.0f);

    // Out of bounds write - handled safely by AMD buffer addressing
    buffer_view.template set<float>(0, 100, true, 555.0f);

    // Vector set
    using float2 = ext_vector_t<float, 2>;
    float2 pair_values{100.0f, 200.0f};
    buffer_view.template set<float2>(0, 5, true, pair_values);
}
```

6.3.6 Atomic Operations

Atomic vs Non-Atomic Operations

C++ Atomic Operations

```
__device__ void example_atomic_operations()
{
    // Shared memory for workgroup-level reductions
    __shared__ float shared_sum[256];
    auto shared_buffer_view = make_buffer_view<address_space_enum::lds>(
        shared_sum, 256
    );

    // Initialize shared memory
    if (threadIdx.x < 256) {
        shared_buffer_view.template set<float>(threadIdx.x, 0.0f, true);
    }
    __syncthreads();

    // Each thread atomically adds to shared memory
    auto my_value = static_cast<float>(threadIdx.x);
    shared_buffer_view.template update<memory_operation_enum::atomic_add, float>(0,0,
↪true,my_value);

    // Atomic max for finding maximum value
    shared_buffer_view.template update<memory_operation_enum::atomic_max, float>(0,1,
↪true,my_value);

    __syncthreads();
}
```

6.3.7 Summary

BufferView abstracts GPU memory hierarchies behind a concise interface. The approach is intended to keep overhead small while enabling optimizations that are otherwise awkward in low-level code.

BufferView offers a unified interface across global, shared, and register memory. Using the same API for each space can lower cognitive overhead, reduce certain classes of mistakes, and support code reuse via template parameters.

Address spaces are encoded in types so that common errors are reported at compile time. Consistent with CK Tile's zero-overhead design aim, compile-time checks are favored over runtime guards. The C++ type system enforces memory-space constraints and can make valid cases more amenable to compiler optimization.

BufferView supports configurable handling of invalid values, optional runtime bounds checks, and conditional access patterns. It also provides atomic operations for thread-safe updates. These features are intended to cover common edge cases without adding unnecessary overhead.

By hiding the complexity of different memory spaces while exposing the operations needed for high-performance GPU computing, BufferView establishes a pattern that the rest of CK Tile follows: compile-time abstractions that enhance rather than compromise performance. The *Tensor Views - Multi-Dimensional Structure* and *Tile Distribution - The Core API* add capability while maintaining the efficiency established at the base. For hardware-specific details about memory hierarchies, see *Intro to AMD CDNA Architecture*.

6.3.8 Next Steps

Continue to *Tensor Views - Multi-Dimensional Structure* to learn how to build structured tensor views on top of buffer views.

6.4 Tensor Views - Multi-Dimensional Structure

6.4.1 Overview

While *BufferView* provides the foundation for raw memory access, TensorView adds multi-dimensional structure to flat memory regions. This abstraction bridges the gap between how developers conceptualize data and how that data is physically stored in linear memory. TensorView enables coordinate-based access patterns that match the natural structure of algorithms while maintaining the performance characteristics necessary for efficient GPU computation.

TensorView presents different logical views of the same underlying memory without copying data. A single memory region can be viewed as a row-major matrix, a column-major matrix, or a transposed matrix, using different TensorView configurations. This zero-copy abstraction enables flexible transformations and access patterns while maintaining optimal memory bandwidth utilization.

6.4.2 TensorView Architecture

6.4.3 The Foundation: BufferView and TensorDescriptor

TensorView builds upon two fundamental components that work in concert to provide structured access to memory. The *BufferView* component handles the low-level memory access, providing type-safe operations with address space awareness. The *TensorDescriptor* component encodes the multi-dimensional structure, including shape information and stride patterns that determine how coordinates map to memory offsets.

This separation of concerns enables optimizations. The BufferView can optimize for the specific memory space while the TensorDescriptor can encode complex access patterns without concern for the underlying memory type. Together, they provide a complete abstraction for multi-dimensional data access.

6.4.4 C++ Implementation

File: include/ck_tile/core/tensor/tensor_view.hpp

Creating TensorViews

The creation of a TensorView involves combining a BufferView with a TensorDescriptor. This process can be done explicitly for maximum control or through convenience functions for common patterns:

```
#include <ck_tile/core/tensor/tensor_view.hpp>
#include <ck_tile/core/tensor/tensor_descriptor.hpp>
#include <ck_tile/core/numeric/tuple.hpp>

// The actual C++ template signature from tensor_view.hpp:
// template <typename BufferView_,
//           typename TensorDesc_,
//           memory_operation_enum DstInMemOp_ = memory_operation_enum::set>
// struct tensor_view

__device__ void example_tensor_creation()
{
    // Create a 3x4 matrix in global memory
    float data[12] = {0,1,2,3,4,5,6,7,8,9,10,11};

    // Method 1: Create buffer and descriptor separately
    auto buffer = make_buffer_view<address_space_enum::global>(data, 12);
    auto desc = make_tensor_descriptor(
        make_tuple(3, 4),    // shape: 3 rows, 4 columns
        make_tuple(4, 1)     // strides: row stride=4, col stride=1
    );

    // Create tensor view
    auto tensor = make_tensor_view<address_space_enum::global>(buffer, desc);

    // Method 2: Use convenience function for packed layout
    auto tensor2 = make_naive_tensor_view_packed<address_space_enum::global>(
        data,                // pointer
        make_tuple(3, 4)     // shape (strides calculated automatically)
    );

    // Access element at (1, 2)
    float value = tensor(make_tuple(1, 2));  // Returns 6

    // Update element
    tensor(make_tuple(2, 1)) = 99.0f;
}
```

Coordinate-Based Access

The fundamental operation of TensorView is translating multi-dimensional coordinates into memory accesses. This translation happens through an advanced pipeline that maintains efficiency while providing flexibility:

6.4.5 Memory Layouts and Strides

A key feature of TensorView is its ability to represent different memory layouts through stride manipulation. This capability enables zero-copy transformations that would otherwise require expensive memory operations:

Row-Major vs Column-Major Layouts

The choice of memory layout has profound implications for performance. Row-major layout, where consecutive elements in a row are stored contiguously, optimizes for row-wise traversal. Column-major layout optimizes for column-wise traversal. CK's TensorView abstraction allows algorithms to work with their natural access patterns regardless of the underlying storage:

```
__device__ void example_memory_layouts()
{
    float data[12] = {0,1,2,3,4,5,6,7,8,9,10,11};

    // Row-major layout (default)
    auto row_major = make_naive_tensor_view_packed<address_space_enum::global>(
        data, make_tuple(3, 4)
    );
    // Strides: (4, 1) - moving one row advances by 4 elements

    // Column-major layout through custom strides
    auto col_major = make_tensor_view<address_space_enum::global>(
        make_buffer_view<address_space_enum::global>(data, 12),
        make_tensor_descriptor(
            make_tuple(3, 4),    // shape
            make_tuple(1, 3)     // strides: row stride=1, col stride=3
        )
    );

    // Transposed view (no data copy!)
    auto transposed = make_tensor_view<address_space_enum::global>(
        make_buffer_view<address_space_enum::global>(data, 12),
        make_tensor_descriptor(
            make_tuple(4, 3),    // transposed shape
            make_tuple(1, 4)     // transposed strides
        )
    );

    // All three views access the same memory, just differently
    // row_major(1,2) == col_major(2,1) == transposed(2,1)
}
```

6.4.6 Advanced Operations

Slicing and Subviews

TensorView supports advanced slicing operations that create new views of subsets of the data. These operations are essential for algorithms that process data in blocks or tiles. See *Tile Window - Data Access Gateway* for production use.

```
__device__ void example_slicing_operations()
{
```

(continues on next page)

(continued from previous page)

```
    // Create a larger tensor
    float data[100];
    auto tensor = make_naive_tensor_view_packed<address_space_enum::global>(
        data, make_tuple(10, 10)
    );

    // Create a subview using transforms
    // This would typically be done with tile_window in production code
    auto subview = make_tensor_view<address_space_enum::global>(
        tensor.get_buffer_view(),
        transform_tensor_descriptor(
            tensor.get_tensor_descriptor(),
            make_tuple(
                make_pass_through_transform(number<5>{}),  // 5 rows
                make_pass_through_transform(number<5>{})   // 5 columns
            ),
            make_tuple(number<2>{}, number<3>{})  // offset (2,3)
        )
    );

    // subview now represents a 5x5 region starting at (2,3)
}
```

Vectorized Access

GPUs achieve maximum memory bandwidth through vectorized operations. TensorView provides native support for vector loads and stores. See *LoadStoreTraits - Memory Access Optimization Engine* for more details.

```
__device__ void example_vectorized_access()
{
    float data[256];
    auto tensor = make_naive_tensor_view_packed<address_space_enum::global>(
        data, make_tuple(16, 16)
    );

    // Create coordinate for vectorized access
    auto coord = make_tensor_coordinate(
        tensor.get_tensor_descriptor(),
        make_tuple(4, 0)  // row 4, starting at column 0
    );

    // Load 4 consecutive elements as float4
    using float4 = vector_type<float, 4>::type;
    auto vec4 = tensor.get_vectorized_elements<float4>(coord, 0);

    // Process vector data
    vec4.x *= 2.0f;
    vec4.y *= 2.0f;
    vec4.z *= 2.0f;
    vec4.w *= 2.0f;

    // Store back
```

(continues on next page)

(continued from previous page)

```
    tensor.set_vectorized_elements<float4>(coord, 0, vec4);
}
```

6.4.7 Performance Considerations

Memory Access Patterns

The efficiency of TensorView operations depends on memory access patterns. Understanding these patterns is important for achieving optimal performance. See *Intro to AMD CDNA Architecture* for hardware considerations.

Compile-Time Optimization

CK's TensorView leverages compile-time optimization to achieve zero-overhead abstraction. When tensor dimensions and strides are known at compile time, the entire coordinate-to-offset calculation can be resolved during compilation:

```
// Compile-time known dimensions enable optimization
constexpr auto shape = make_tuple(number<256>{}, number<256>{});
constexpr auto strides = make_tuple(number<256>{}, number<1>{});

auto tensor = make_tensor_view<address_space_enum::global>(
    buffer,
    make_tensor_descriptor(shape, strides)
);

// This access compiles to a single memory instruction
constexpr auto coord = make_tuple(number<5>{}, number<10>{});
auto value = tensor(coord);  // Offset calculated at compile time
```

6.4.8 TensorView vs BufferView

Understanding when to use TensorView versus BufferView is crucial for writing efficient code:

BufferView excels at raw memory operations where linear access is natural or where the overhead of coordinate calculation would be prohibitive. TensorView is best suited for algorithms that operate in terms of multi-dimensional coordinates, such as matrix operations, image processing, or tensor contractions.

6.4.9 Integration with Tile Distribution

TensorView serves as the foundation for *tile distribution's* higher-level abstractions. When combined with *tile windows* and distribution patterns, TensorView enables the automatic generation of efficient access patterns:

```
// TensorView provides the base abstraction
auto tensor_view = make_naive_tensor_view_packed<address_space_enum::global>(
    global_memory, make_tuple(M, N)
);

// Tile window builds on TensorView for distributed access
auto tile_window = make_tile_window(
    tensor_view,
    tile_shape,
    origin,
    distribution
);
```

(continues on next page)

(continued from previous page)

```
// The distribution automatically generates optimal access patterns
auto distributed_tensor = tile_window.load();
```

6.4.10 Summary

TensorView bridges the gap between logical multi-dimensional data structures and physical memory layout. Through its advanced design, TensorView provides:

Multi-dimensional Indexing: Natural coordinate-based access to data, matching how algorithms conceptualize their operations. This abstraction eliminates error-prone manual index calculations while maintaining performance.

Flexible Memory Layouts: Support for row-major, column-major, and custom stride patterns enables algorithms to work with data in its most natural form. Zero-copy transformations like transposition become stride manipulations.

Zero-Copy Views: The ability to create different logical views of the same physical memory enables flexible transformations without the overhead of data movement. This capability is essential for efficient GPU programming where memory bandwidth is often the limiting factor.

Type Safety: Dimensions and memory spaces are encoded in the type system, catching errors at compile time rather than runtime. This safety comes without performance overhead thanks to template metaprogramming.

Seamless Integration: TensorView works harmoniously with *BufferView* for low-level access and serves as the foundation for higher-level abstractions like *tile windows* and *distributed tensors*.

The abstraction enables writing dimension-agnostic algorithms while maintaining high performance through compile-time optimizations.

6.4.11 Next Steps

Continue to *Coordinate Systems - The Mathematical Foundation* to understand the mathematical foundation of coordinate transformations in CK Tile.

6.5 Tile Distribution - The Core API

6.5.1 Overview

At the heart of Composable Kernel's approach to efficient GPU computation lies TileDistribution, a compile-time abstraction that transforms how developers approach parallel programming on GPUs. Rather than requiring programmers to manually manage thread coordination, memory access patterns, and data distribution, TileDistribution provides an mathematical framework that automatically maps logical computational coordinates to physical execution resources.

The architectural foundation of tile distribution in CK rests upon the *coordinate transformation system* that bridges multiple abstract spaces. This system manages the interaction between four primary coordinate dimensions, each serving a distinct purpose in the overall computation model. The X dimensions represent the physical tensor coordinates, capturing the actual layout of data in memory. The Y dimensions encode the tile access patterns, defining how threads traverse their assigned data. The P dimensions map to processing elements, representing the hierarchical organization of threads, warps, and blocks in the *GPU's execution model*. Additionally, the optional R dimensions enable replication strategies for algorithms that benefit from redundant computation to reduce communication overhead.

This multi-dimensional mapping framework enables CK to express arbitrarily complex data access patterns through a mathematically formalism. The power of this approach becomes evident when considering how traditional GPU programming requires developers to manually calculate memory addresses, ensure coalescing constraints, *avoid bank conflicts*, and manage thread cooperation. TileDistribution handles all these concerns within a unified abstraction that can be analyzed, optimized, and verified at compile time.

The `tile_distribution` template class integrates three essential components that work together to deliver optimal performance. The `PsYs2XsAdaptor` component performs *coordinate transformations* from processing and pattern dimensions to physical tensor coordinates, implementing the mathematical mappings that ensure efficient memory access. The `Ys2DDescriptor` component handles the linearization of Y dimensions, transforming multi-dimensional tile patterns into register allocation schemes that maximize register reuse and minimize register pressure. The `StaticTileDistributionEncoding` captures the hierarchical decomposition of work across the GPU's compute resources, encoding decisions about how work is partitioned across thread blocks, warps, and individual threads.

This design adapts to diverse computational scenarios without manual intervention. The same high-level code can execute on GPUs with different numbers of streaming multiprocessors, varying warp sizes, or distinct memory hierarchies. The compile-time nature of the abstraction ensures that all coordination logic is resolved during compilation, resulting in machine code that is comparable hand-optimized implementations. This adaptability enables a single implementation to achieve improved performance across a wide range of tensor sizes, shapes, and computational patterns without the combinatorial explosion of specialized kernels.

6.5.2 Complete Tile Distribution System Overview

6.5.3 Coordinate System Architecture

6.5.4 What is Tile Distribution?

In GPU programming, distributing work across thousands of parallel threads is an important challenge. Consider a 256×256 matrix multiplication operation and 64 GPU threads organized in warps. The question becomes how to divide this computational work in a way that maximizes memory bandwidth utilization, minimizes bank conflicts, and ensures coalesced memory accesses.

The traditional approach without a tile distribution framework requires programmers to manually calculate global memory addresses for each thread, implement complex index arithmetic that accounts for thread hierarchy (threads within warps, warps within blocks), handle edge cases for non-divisible matrix dimensions, and create different implementations for various matrix sizes. This manual approach is not only error-prone but also fails to adapt to different GPU architectures and their specific memory access patterns.

TileDistribution solves these challenges through a systematic approach to work distribution. It automatically assigns work to threads based on a hierarchical decomposition of the problem space, generates memory access patterns that respect GPU hardware constraints, provides a uniform interface that works across different tensor sizes and shapes, and ensures optimal thread cooperation by automatically managing data movement to thread-local registers.

TileDistribution abstracts the mapping between logical problem coordinates and physical execution resources. Given a thread's position in the GPU's execution hierarchy (specified by warp ID and lane ID within the warp), TileDistribution computes two critical pieces of information: the global memory addresses that this thread should access, and the specific access pattern that ensures efficient memory transactions. This abstraction is implemented in C++ through the following core structure:

```
template <typename PsYs2XsAdaptor_,
          typename Ys2DDescriptor_,
          typename StaticTileDistributionEncoding_,
          typename TileDistributionDetail_>
struct tile_distribution
{
    // Core functionality: map thread coordinates to data
    CK_TILE_HOST_DEVICE static auto _get_partition_index()
    {
        if constexpr(NDimP == 1)
            return array<index_t, 1>{get_lane_id()};
        else if constexpr(NDimP == 2)
            return array<index_t, 2>{get_warp_id(), get_lane_id()};
```

(continues on next page)

(continued from previous page)

```
    }

    // Calculate which tensor elements this thread accesses
    template <typename PartitionIndex>
    CK_TILE_HOST_DEVICE static auto calculate_tile_Ys_index(const PartitionIndex& ps_idx)
    {
        return detail::calculate_tile_Ys_index(
            StaticTileDistributionEncoding{}, ps_idx);
    }
};
```

6.5.5 Problem Space Mapping

6.5.6 Creating a TileDistribution

Creating and using a TileDistribution:

```
// SPDX-License-Identifier: MIT
// Copyright (c) Advanced Micro Devices, Inc. All rights reserved.

#include "ck_tile/host.hpp"
#include "ck_tile/core.hpp"
#include <cstring>
#include <iostream>
#include <vector>

namespace ck_tile {

struct TileDistributionExample
{
    CK_TILE_DEVICE void operator()(float* global_data,
                                   ck_tile::index_t global_shape_0,
                                   ck_tile::index_t global_shape_1) const
    {
        if(threadIdx.x == 0 && blockIdx.x == 0) {
            printf("\n=== Tile Distribution Example (Device Kernel) ===\n");
        }
        block_sync_lds();

        // Create a tile distribution encoding
        // This defines how a tensor is distributed across threads
        auto encoding = tile_distribution_encoding<
            sequence<>,                     // rs_lengths=[] - No replication dimensions
            tuple<
                sequence<2, 2>,             // hs_lengthss=[[2, 2], [2, 2]] -␣
↪Hierarchical lengths for each X dimension
                sequence<2, 2>>,
            tuple<sequence<1>, sequence<2>>, // ps_to_rhss_major=[[1], [2]] - P to RH␣
↪major mappings
            tuple<sequence<0>, sequence<0>>, // ps_to_rhss_minor=[[0], [0]] - P to RH␣
↪minor mappings
            sequence<1, 2>,                  // ys_to_rhs_major=[1, 2] - Y to RH major␣
```

(continues on next page)

(continued from previous page)

```
↪mappings
            sequence<1, 1>>{};                // ys_to_rhs_minor=[1, 1] - Y to RH minor␣
↪mappings

        // Create the tile distribution from the encoding
        auto distribution = make_static_tile_distribution(encoding);

        // Calculate sizes from the distribution encoding
        // x0_size = np.prod(distribution.encoding.hs_lengthss[0])
        constexpr auto hs_lengths_0 = encoding.hs_lengthss_[number<0>{}];  // sequence<2,
↪ 2>
        constexpr auto hs_lengths_1 = encoding.hs_lengthss_[number<1>{}];  // sequence<2,
↪ 2>

        constexpr index_t x0_size = reduce_on_sequence(hs_lengths_0, multiplies{}, number
↪<1>{});
        constexpr index_t x1_size = reduce_on_sequence(hs_lengths_1, multiplies{}, number
↪<1>{});

        // Print distribution info (only from thread 0)
        if(threadIdx.x == 0 && blockIdx.x == 0) {
            printf("\n- Tile distribution created:\n");
            printf("  X dimensions: %d\n", distribution.get_num_of_dimension_x());
            printf("  Y dimensions: %d\n", distribution.get_num_of_dimension_y());
            printf("  P dimensions: %d\n", distribution.get_num_of_dimension_p());
            printf("  X lengths: [%d, %d]\n", x0_size, x1_size);
        }
        block_sync_lds();

    // Create packed tensor view (contiguous row-major) using helper
    auto global_view = make_naive_tensor_view_packed<address_space_enum::global>(
        global_data,
        make_tuple(global_shape_0, global_shape_1));

        // Window configuration
        auto window_lengths = make_tuple(x0_size, x1_size);

        // Get current thread's warp and thread indices
        index_t warp_id = threadIdx.x / get_warp_size();
        index_t thread_id = threadIdx.x % get_warp_size();

        // Window origin - small offset from origin
        auto window_origin = make_tuple(1, 3);  // Small offset from origin

    // Create tile window
    auto tile_window = make_tile_window(
        global_view,
        window_lengths,
        {1, 3},  // Window origin as initializer list
        distribution
    );
```

(continues on next page)

(continued from previous page)

```
    // Load distributed tensor
    auto distributed_tensor = tile_window.load();

        // Collect values by sweeping through the distributed tensor
        constexpr index_t max_elements = x0_size*x1_size;
        float collected_values[max_elements];
        index_t value_count = 0;

        // Sweep through the distributed tensor and collect values using sweep_tile API
        sweep_tile(distributed_tensor, [&](auto idx) {
            if(value_count<max_elements) {
                collected_values[value_count] = distributed_tensor(idx);
                value_count++;
            }
        });

        // Serialize printing in a fixed order for selected threads only.
        static constexpr int print_thread_ids[] = {0, 1, 64, 65};
        for(int sel : print_thread_ids) {
            block_sync_lds();
            if(static_cast<int>(threadIdx.x) == sel) {
                printf("Partition index: (warp=%d, thread=%d)\n", static_cast<int>(warp_
↪id), static_cast<int>(thread_id));
                printf("Collected values: ");
                for(index_t i = 0; i < value_count; i++) {
                    printf("%.0f", collected_values[i]);
                    if(i < value_count - 1) printf(", ");
                }
                printf("\n\n");
            }
            block_sync_lds();
        }
    }
};
}

int main()
{
    // Host-side allocation & initialization of pattern data
    // Reproduce the compile-time sizes used in the kernel: hs_lengths = [2,2] => x␣
↪sizes=4; global = 4+5 = 9
    constexpr ck_tile::index_t global_shape_0 = 9; // x0_size(4) + 5
    constexpr ck_tile::index_t global_shape_1 = 9; // x1_size(4) + 5
    constexpr ck_tile::index_t total_elems    = global_shape_0 * global_shape_1; // 81

    std::vector<float> h_global_data(total_elems);
    for(ck_tile::index_t i = 0; i < global_shape_0; ++i) {
        for(ck_tile::index_t j = 0; j < global_shape_1; ++j) {
            h_global_data[i * global_shape_1 + j] = static_cast<float>(i * 100 + j);
        }
    }
```

(continues on next page)

(continued from previous page)

```
    ck_tile::DeviceMem d_global_data(sizeof(float) * total_elems);
    d_global_data.ToDevice(h_global_data.data());

    std::cout << "\nGlobal data (host print, to be used by device) shape=("
              << static_cast<int>(global_shape_0) << "," << static_cast<int>(global_
↪shape_1) << ")\n\n";
    for(ck_tile::index_t i = 0; i < global_shape_0; ++i) {
        for(ck_tile::index_t j = 0; j < global_shape_1; ++j) {
            std::cout << h_global_data[i * global_shape_1 + j];
            if(j + 1 < global_shape_1) std::cout << "\t";
        }
        std::cout << '\n';
    }
    std::cout << '\n';

    constexpr ck_tile::index_t kBlockSize  = 128;
    constexpr ck_tile::index_t kBlockPerCu = 1;
    constexpr ck_tile::index_t kGridSize = 1;

    using Kernel = ck_tile::TileDistributionExample;
    float ave_time = launch_kernel(ck_tile::stream_config{nullptr, true, 0, 0, 1},
                                   ck_tile::make_kernel<kBlockSize, kBlockPerCu>(
                                       Kernel{},
                                       kGridSize,
                                       kBlockSize,
                                       0,
                                       static_cast<float*>(d_global_data.
↪GetDeviceBuffer()),
                                       global_shape_0,
                                       global_shape_1));

    std::cout << "Kernel execution completed. Average time: " << ave_time << " ms" <<␣
↪std::endl;

    return 0;
}
```

6.5.7 Hierarchical Decomposition

6.5.8 Advanced Example: Matrix Multiplication Distribution

```
// Real GEMM kernel pattern using TileDistribution
template<typename AType, typename BType, typename CType>
__global__ void gemm_kernel(
    const AType* __restrict__ a_ptr,
    const BType* __restrict__ b_ptr,
    CType* __restrict__ c_ptr,
    index_t M, index_t N, index_t K)
{
    // Define the tile distribution encoding at compile time
    using Encoding = tile_distribution_encoding<
        sequence<>,                                     // R: no replication
```

(continues on next page)

(continued from previous page)

```
        tuple<sequence<4, 2, 8, 4>,                    // H for M dimension
              sequence<4, 2, 8, 4>>,                   // H for N dimension
        tuple<sequence<1, 2>, sequence<1, 2>>,         // P to RH major
        tuple<sequence<1, 1>, sequence<2, 2>>,         // P to RH minor
        sequence<1, 1, 2, 2>,                          // Y to RH major
        sequence<0, 3, 0, 3>                           // Y to RH minor
    >;

    // Create the distribution
    constexpr auto distribution = make_static_tile_distribution(Encoding{});

    // Create tensor views
    auto a_view = make_tensor_view<const AType>(
        a_ptr,
        make_naive_tensor_descriptor_packed(make_tuple(M, K)));

    // Create tile window for this thread block
    auto a_window = make_tile_window(
        a_view,
        make_tuple(number<256>{}, number<64>{}),  // window size
        {blockIdx.x * 256, 0},                    // origin
        distribution);

    // Load data to distributed tensor (registers)
    auto a_reg = make_static_distributed_tensor<AType>(distribution);

    a_window.load(a_reg);

    // Computation happens in registers
    // Results written back through another window
}
```

6.5.9 Work Distribution Pattern

6.5.10 Memory Access Patterns

One of the key benefits of TileDistribution is generating optimal memory access patterns. The encoding parameters control how threads access memory:

- **H-dimensions**: Define hierarchical decomposition (Repeat, WarpPerBlock, ThreadPerWarp, Vector)
- **P-to-RH mappings**: Control how thread IDs map to the hierarchy
- **Y-to-RH mappings**: Define the access pattern within each thread's tile

6.5.11 Transformation Pipeline

6.5.12 Performance Comparison

6.5.13 Summary

The automatic work distribution capabilities of TileDistribution eliminate one of the most error-prone aspects of GPU programming. TileDistribution's mathematical framework ensures that every thread knows which data elements it should process and automatically handles complex index arithmetic.

Memory access pattern optimization is a performance benefit of the TileDistribution approach. GPUs achieve their computational throughput only when memory accesses follow specific patterns that enable hardware optimizations such as coalescing and broadcast. TileDistribution automatically generates these patterns, such that threads within a warp access contiguous memory locations, that bank conflicts in shared memory are reduced, and that the memory subsystem operates efficiently. This optimization happens transparently, without manual memory pattern analysis.

By encoding the natural hierarchy of threads, warps, and blocks directly into the distribution strategy, the framework ensures that each level of the hierarchy operates optimally. This hierarchical approach enables tiling strategies that would be impractical to implement manually, such as multi-level tiling that simultaneously optimizes for L1 cache, L2 cache, and register file usage.

The zero-overhead nature of TileDistribution, achieved through use of C++ template metaprogramming and compile-time computation, ensures that the abstraction's benefits come without runtime cost. Every aspect of the distribution strategy is resolved at compile time, resulting in machine code that is comparable to hand-written implementations. The compiler's ability to see through the abstraction enables optimizations that aren't typically available to runtime-based approaches.

The same source code can execute efficiently on GPUs with different warp sizes, different numbers of registers per thread, or different shared memory capacities. This portability includes performance portability, with the framework adapting its strategies to match the characteristics of the target architecture.

TileDistribution provides a solid foundation for the CK ecosystem. This abstraction provides a programming model that insulates developers from the complexity of the underlying hardware while enabling them to use hardware capabilities.

6.5.14 Next Steps

See *Terminology Reference - Key Concepts and Definitions* for a glossary of key concepts and terminology used in CK Tile.

6.6 Coordinate Systems - The Mathematical Foundation

6.6.1 Overview

At the heart of the Composable Kernel framework lies a mathematical foundation based on coordinate transformations. This foundation enables the automatic generation of optimal memory access patterns while maintaining a clear separation between algorithmic intent and hardware implementation details. The coordinate system framework transforms the task of GPU work distribution into a series of well-defined mathematical transformations.

These coordinate systems provide the mathematical machinery that maps abstract thread identities to concrete memory addresses, ensuring that every memory access is optimized for the underlying hardware. This systematic approach eliminates the error-prone manual calculations that plague traditional GPU programming while enabling optimizations that would be impractical to implement by hand.

6.6.2 The Five Coordinate Spaces

The CK framework employs five interconnected coordinate spaces, each serving a specific purpose in the journey from thread identification to memory access. These spaces work together to solve the fundamental challenge of GPU programming: efficiently distributing work across thousands of parallel threads while maintaining optimal memory access patterns.

The Challenge and Solution

Consider a fundamental scenario: an 8×8 matrix and 4 GPU threads. Each thread needs to answer several critical questions:

1. **Which thread am I?** (Thread identification)
2. **What work should I do?** (Work assignment)

3. **Where is my data in the tensor?** (Physical location)
4. **How do I share data with other threads?** (Cooperation)
5. **What's the memory address?** (Hardware access)

The coordinate system framework provides a systematic solution through five specialized spaces that transform from logical concepts to physical reality. Each space captures a different aspect of the computation, and the transformations between them encode the distribution strategy.

6.6.3 Thread Identification

Partition Space (P-space) represents the foundation of the coordinate system hierarchy. This space captures the identity of each processing element within the GPU's execution model, providing a structured way to identify threads across the complex hierarchy of warps, blocks, and grids.

GPU Thread Hierarchy

The structure of P-space directly reflects the *hardware organization* of GPUs. Each thread receives a unique P-coordinate that encodes its position within the execution hierarchy. For simple distributions, P-space might be one-dimensional, containing only a thread ID. For complex hierarchical distributions, P-space can have multiple dimensions representing different levels of the GPU's thread organization.

C++ Implementation

File: `include/ck_tile/core/container/multi_index.hpp`

```cpp
#include <ck_tile/core/container/multi_index.hpp>
#include <ck_tile/core/utility/thread_id.hpp>

template <typename TileDistribution>
__device__ void example_p_space_calculation()
{
    // Get P-coordinates from hardware thread IDs
    const index_t thread_id = get_thread_local_1d_id();
    const index_t warp_id = get_warp_local_1d_id();
    const index_t lane_id = get_lane_id();

    // Convert to multi-dimensional P-coordinates
    auto p_coord_2d = make_multi_index(warp_id, lane_id);

    // Using tile distribution (preferred method)
    constexpr auto tile_distribution = TileDistribution{};
    const auto p_coord = tile_distribution.calculate_p_coord();

    // P-coordinates determine:
    // 1. Work distribution - which data this thread processes
    // 2. Memory coalescing - ensuring optimal access patterns
    // 3. Thread cooperation - coordinating shared memory usage
}
```

The P-space abstraction enables CK to handle different GPU architectures transparently. Whether running on GPUs with 32-thread warps or 64-thread wavefronts, the P-space coordinates provide a consistent interface while the underlying implementation adapts to the hardware.

6.6.4 Logical Work Organization

Yield Space (Y-space) represents the logical organization of work within each thread's assigned tile. While P-space identifies which thread is executing, Y-space defines what that thread does with its assigned work. This abstraction enables the expression of complex access patterns in a hardware-independent manner.

Work Assignment Structure

The power of Y-space lies in its ability to express different iteration patterns without changing the underlying distribution logic. A thread might traverse its Y-space in row-major order for one algorithm, column-major for another, or even use *space-filling curves* for optimal cache utilization. This flexibility enables algorithm-specific optimizations while maintaining a consistent framework.

Hierarchical Y-Space

For complex kernels, Y-space can have a hierarchical structure that mirrors the hierarchical nature of GPU architectures:

```
// Hierarchical Y-space for complex kernels
template <typename TileDistribution>
__device__ void example_hierarchical_y_space()
{
    constexpr auto tile_distribution = TileDistribution{};

    // 4D Y-space: [repeat, warp, thread, vector]
    constexpr auto y_hierarchical = make_tuple(
        number<4>{},   // Repeat dimension
        number<2>{},   // Warp dimension
        number<8>{},   // Thread dimension
        number<4>{}    // Vector dimension
    );

    // Each dimension serves different purpose:
    // - Repeat: Algorithm repetition (e.g., attention heads)
    // - Warp: Inter-warp cooperation patterns
    // - Thread: Per-thread work items
    // - Vector: SIMD vectorization

    // Sweep through Y-space with compile-time unrolling
    sweep_tile(distributed_tensor, [&](auto y_coord) {
        // y_coord is compile-time multi_index
        // All iterations unrolled at compile time
        auto value = distributed_tensor(y_coord);
        // Process value...
    });
}
```

6.6.5 Physical Tensor Coordinates

X-space represents the ground truth of data organization: the actual coordinates within the global tensor. This space directly corresponds to how users conceptualize their data: row and column indices for matrices, spatial coordinates for images, or multi-dimensional indices for general tensors.

Memory Layout Mapping

The relationship between X-space and physical memory involves considerations of data layout, padding, and alignment:

```
template <typename TensorDescriptor>
__device__ void example_x_space_operations()
{
    constexpr auto tensor_desc = TensorDescriptor{};

    // X-space properties
    constexpr auto x_lengths = tensor_desc.get_lengths();
    constexpr auto x_strides = tensor_desc.get_strides();

    // Direct X-coordinate specification
    constexpr auto x_coord = make_multi_index(number<3>{}, number<4>{});

    // Convert to linear offset
    constexpr auto linear_offset = tensor_desc.calculate_offset(x_coord);

    // X-coordinates from P+Y transformation
    const auto x_from_py = tile_dist.calculate_index(p_coord, y_coord);

    // Bounds checking
    const bool valid = is_valid_x_coord(x_coord, x_lengths);
}
```

6.6.6 The Core Transformation: P + Y → X

The transformation from P and Y coordinates to X coordinates represents the heart of tile distribution. This transformation encodes the entire distribution strategy, determining how logical thread work maps to physical tensor locations.

Transformation Pipeline

Mathematical Foundation

The P+Y→X transformation can be expressed mathematically as a composition of functions:

$$X = f(P, Y) = BasePosition(P) + LocalOffset(Y)$$

Where: - BasePosition(P) determines where in the tensor this thread's tile begins - LocalOffset(Y) specifies the offset within the tile

This transformation is highly configurable through the distribution encoding, enabling different strategies for different algorithms while maintaining the same mathematical framework.

6.6.7 Replication and Cooperation

Replication Space (R-space) introduces a mechanism for expressing data sharing and cooperation patterns between threads. Unlike the other coordinate spaces which map to unique data elements, R-space enables multiple processing elements to work on the same data, facilitating communication and reduction operations.

Replication Patterns

```
template <typename TileDistribution>
__device__ void example_r_space_operations()
```

(continues on next page)

(continued from previous page)

```
{
    constexpr auto tile_distribution = TileDistribution{};
    constexpr auto r_lengths = tile_distribution.get_r_lengths();

    // Broadcasting with R-space
    template <typename DataType>
    __device__ auto broadcast_across_r_space(DataType value)
    {
        const auto r_coord = tile_distribution.calculate_r_coord();
        __shared__ DataType shared_value;

        if (r_coord == make_multi_index(0, 0)) {
            shared_value = value;  // Source thread
        }
        __syncthreads();

        return shared_value;  // All threads get the value
    }

    // Reduction across R-space
    template <typename DataType>
    __device__ auto reduce_across_r_space(DataType local_value)
    {
        // Use hardware-accelerated reduction
        return block_reduce_sum(local_value);
    }
}
```

R-space enables cooperation patterns that would be difficult to express otherwise. By providing a systematic way to identify which threads share data, it enables automatic generation of communication patterns.

6.6.8 Memory Linearization

D-space represents the final transformation in the coordinate pipeline: converting multi-dimensional coordinates to linear memory addresses. This transformation incorporates all the low-level details of memory layout, including stride patterns, padding, and alignment requirements.

Linearization Strategies

The linearization process must consider multiple factors:

```
template <typename TensorDescriptor>
__device__ void example_d_space_linearization()
{
    // Standard linearization
    template <typename XCoord>
    __device__ constexpr auto calculate_linear_offset(const XCoord& x_coord)
    {
        index_t offset = 0;
        static_for<0, ndim, 1>{}([&](auto dim) {
            offset += x_coord.at(dim) * strides.at(dim);
        });
        return offset;
```

(continues on next page)

(continued from previous page)

```
    }

    // Specialized patterns for optimization
    // Row-major: offset = x0 * N + x1
    // Column-major: offset = x1 * M + x0
    // Blocked: Complex pattern for cache efficiency
}
```

6.6.9 Complete Pipeline Example

The following is a complete example showing how all coordinate spaces work together:

6.6.10 Real-World Example: Matrix Multiplication

matrix multiplication demonstrates how coordinate systems work in practice/

```
template<typename AType, typename BType, typename CType>
__global__ void gemm_kernel_with_coordinates(
    const AType* a_ptr, const BType* b_ptr, CType* c_ptr,
    index_t M, index_t N, index_t K)
{
    // Define distribution encoding
    using Encoding = tile_distribution_encoding<
        sequence<>,                                 // R: no replication
        tuple<sequence<4, 2, 8, 4>,                // H for M dimension
              sequence<4, 2, 8, 4>>,               // H for N dimension
        tuple<sequence<1, 2>, sequence<1, 2>>,     // P mappings
        tuple<sequence<1, 1>, sequence<2, 2>>,     // P minor
        sequence<1, 1, 2, 2>,                      // Y major
        sequence<0, 3, 0, 3>                       // Y minor
    >;

    constexpr auto distribution = make_static_tile_distribution(Encoding{});

    // Step 1: Get P-coordinates (thread identity)
    const auto p_coord = distribution.calculate_p_coord();

    // Step 2: Iterate through Y-space (work assignment)
    sweep_tile(c_tile, [&](auto y_coord) {
        // Step 3: P+Y→X transformation
        const auto x_coord = distribution.calculate_index(p_coord, y_coord);

        // Step 4: X→D transformation (handled by tensor view)
        // Step 5: Actual computation at these coordinates
        c_tile(y_coord) = compute_element(x_coord);
    });
}
```

6.6.11 Performance Implications

The coordinate system framework enables several critical optimizations:

Memory Coalescing: By carefully structuring the P+Y→X transformation, consecutive threads access consecutive memory locations, achieving optimal memory bandwidth utilization.

Cache Efficiency: The Y-space traversal order can be designed to maximize cache reuse, keeping frequently accessed data in fast memory.

Register Optimization: The Y→D transformation enables optimal register allocation, minimizing register pressure while maximizing reuse.

Vectorization: The coordinate transformations naturally align with vector operations, enabling efficient use of SIMD instructions.

6.6.12 Summary

The coordinate system framework represents the mathematical foundation that enables CK's high performance and productivity benefits. Through the systematic transformation from thread identity (P-space) through logical work organization (Y-space) to physical tensor coordinates (X-space) and finally to linear memory addresses (D-space), this framework solves the fundamental challenges of GPU programming.

Key insights from the coordinate system framework:

Separation of Concerns: Each coordinate space captures a different aspect of the computation, enabling independent optimization of each aspect while maintaining a coherent whole.

Mathematical Rigor: The transformations between coordinate spaces are well-defined mathematical functions, enabling formal analysis and verification of distribution strategies.

Hardware Abstraction: The framework abstracts hardware details while enabling hardware-specific optimizations, achieving both portability and performance.

Automatic Optimization: By encoding distribution strategies as coordinate transformations, the framework enables automatic generation of optimal access patterns that would be impractical to implement manually.

Composability: Different distribution strategies can be expressed by composing different transformations, enabling rapid experimentation and optimization.

These coordinate systems provide the conceptual framework for reasoning about GPU computation and the practical tools for achieving optimal performance. As GPU architectures continue to evolve, this mathematical foundation ensures that CK programs can adapt and continue to achieve high performance.

6.6.13 Next Steps

With a solid understanding of the coordinate system framework, the next sections explore how these concepts are applied in practice. Return to *CK Tile conceptual documentation table of contents* to see the structure of the complete CK Tile documentation.

6.7 Terminology Reference - Key Concepts and Definitions

6.7.1 Overview

The Composable Kernel framework introduces concepts and abstractions that form the foundation of its approach to high-performance GPU computing. This terminology reference serves as a comprehensive guide to the language of CK, providing detailed explanations of each term along with practical examples of their usage in C++ code.

The terminology of CK reflects its layered architecture, with concepts building upon one another in a logical progression. From the fundamental notion of tiles and distributions to the compile-time coordinate transformation systems,

each term represents a carefully designed abstraction that serves a specific purpose in the overall framework. This reference is organized to mirror this conceptual hierarchy, starting with core concepts and progressing through increasingly specialized terminology.

As you explore this reference, you'll notice that many terms are interconnected, reflecting the holistic nature of the CK design. A tile is not just a block of data but a fundamental unit of work distribution. A distribution is not merely a pattern but a mathematical framework for optimal resource utilization. These interconnections are intentional and understanding them is crucial for effective use of the framework.

6.7.2 Core Concepts

Tile

The concept of a tile represents the fundamental unit of data organization in the CK framework. A tile is a contiguous block of data that is processed as a cohesive unit by a coordinated group of threads. This abstraction serves multiple critical purposes in achieving high performance on GPU architectures. By organizing data into tiles, the framework ensures that memory accesses exhibit spatial locality, enabling efficient use of cache hierarchies. The tile size is chosen to balance several competing factors: it must be large enough to amortize the overhead of memory transactions, yet small enough to fit within the limited on-chip memory resources. Furthermore, tiles are designed to align with the *GPU's execution model*, ensuring that threads within a warp access contiguous memory locations for optimal bandwidth utilization.

C++ Usage: `using TileShape = sequence<256, 256>;`

Distribution

The distribution pattern represents one of the most compile-time abstractions in the CK framework, defining the precise mapping between logical data elements and the physical processing resources that will operate on them. A distribution is far more than an assignment scheme—it embodies a strategy for achieving optimal performance on GPU hardware. The distribution determines which threads access which data elements, how those accesses are ordered to maximize memory bandwidth, and how intermediate results are shared between cooperating threads. By encoding these decisions at compile time, distributions enable the generation of highly optimized code that respects hardware constraints while maintaining algorithmic clarity. For a detailed exploration of distribution concepts, see *Tile Distribution - The Core API*.

C++ Type: `tile_distribution<...>`

Encoding

An encoding in CK represents a compile-time specification that captures the strategy for distributing tensor data across GPU processing elements. This specification is not merely a configuration but a mathematical description of the transformation between coordinate spaces. The encoding defines the hierarchical decomposition of work, the mapping between thread indices and data elements, and the patterns by which threads cooperate to process their assigned data. By expressing these concepts as compile-time constants, encodings enable aggressive compiler optimizations while ensuring that distribution strategies can be verified for correctness before execution.

C++ Type: `tile_distribution_encoding<...>`

6.7.3 Coordinate Spaces

For a comprehensive mathematical treatment of coordinate systems, see *Coordinate Systems - The Mathematical Foundation*.

P-Space (Partition Space)

The Partition Space, or P-space, represents the fundamental abstraction for identifying processing elements within the GPU's execution hierarchy. This coordinate space captures the multi-level organization of GPU computation, from individual threads to warps to thread blocks. P-space typically manifests as either a one-dimensional space containing only lane identifiers for simple distributions, or a two-dimensional space incorporating both warp and lane identifiers for more complex hierarchical distributions. The significance of P-space extends beyond mere thread identification—it forms the foundation for all work distribution decisions, determining which processing elements will collaborate on specific data tiles and how they will coordinate their efforts.

The dimensions of P-space directly reflect the hardware's execution model. In a one-dimensional P-space, threads are identified solely by their lane ID within a warp, suitable for algorithms where inter-warp coordination is minimal. Two-dimensional P-space adds warp-level coordination, enabling advanced tiling strategies that leverage both intra-warp and inter-warp parallelism. The values in P-space are always hardware thread indices, providing a direct mapping to the physical execution resources.

C++ Example:

```
// Get current thread's P coordinates
auto p_idx = Distribution::_get_partition_index();
```

Y-Space (Yield Space)

The Yield Space, or Y-space, embodies the logical structure of computation within each tile, representing the pattern by which threads traverse their assigned data. Unlike P-space which identifies threads, Y-space defines what each thread does with its assigned work. This abstraction enables the expression of complex access patterns—from simple linear traversals to advanced space-filling curves—in a hardware-independent manner. The dimensionality of Y-space varies with the algorithm's requirements, typically ranging from two dimensions for matrix operations to four or more for complex tensor contractions.

Y-space serves as the primary iteration space for computational kernels. When a thread processes its assigned tile, it iterates through Y-space coordinates, with each coordinate mapping to specific data elements within the tile. This abstraction enables critical optimizations: the Y-space traversal order can be designed to maximize data reuse, minimize register pressure, or optimize for specific hardware characteristics, all without changing the fundamental algorithm.

C++ Example:

```
// Iterate over Y-space
sweep_tile(tensor, [](auto y_idx) { /*...*/ });
```

X-Space (Physical Tensor Space)

The Physical Tensor Space, or X-space, represents the ground truth of data organization—the actual coordinates within the global tensor. This space directly corresponds to how data is laid out in memory, with dimensions matching those of the tensor being processed. For a matrix, X-space is two-dimensional with row and column coordinates. For a 4D convolution tensor, X-space encompasses batch, channel, height, and width dimensions. X-space serves as the target of the coordinate transformation pipeline, where abstract thread and pattern coordinates are converted into concrete memory addresses.

The relationship between X-space and physical memory is direct but not necessarily trivial. While X-space coordinates identify logical positions within a tensor, the actual memory layout may involve padding, striding, or other transformations for alignment and performance. The CK framework handles these low-level details transparently, allowing algorithms to work with logical X-space coordinates while ensuring efficient physical memory access.

C++ Example:

```
// Calculate X coordinates from P+Y
auto x_idx = distribution.calculate_index(p_idx);
```

R-Space (Replication Space)

The Replication Space, or R-space, introduces a advanced mechanism for expressing redundant computation patterns that enhance performance through data sharing. Unlike the other coordinate spaces which map to unique data elements, R-space enables multiple processing elements to compute the same values, facilitating efficient communication patterns. This replication serves multiple purposes: it can reduce global memory traffic by computing values locally rather than loading them, enable efficient reduction operations by providing private workspace for each thread group, and facilitate complex data exchange patterns that would otherwise require expensive synchronization.

R-space dimensions are optional and algorithm-specific. A matrix multiplication might use R-space to replicate portions of the input matrices across thread groups, enabling each group to compute partial products independently. The framework automatically manages the complexities of replication, including the allocation of private storage and the coordination of replicated computations.

C++ Example:

```
// R-dimensions in encoding
using Encoding = tile_distribution_encoding<
    sequence<2>,  // rs_lengths: 2-way replication
    /*...*/
>;
```

D-Space (Data Space)

The Data Space, or D-space, represents the final stage of the coordinate transformation pipeline—the linearization of multi-dimensional tile data for efficient storage in thread-local registers. This one-dimensional space serves a critical role in managing the GPU's most precious resource: register files. By transforming the potentially complex Y-space coordinates into a linear D-space index, the framework enables efficient register allocation and access patterns that minimize register bank conflicts and maximize instruction-level parallelism.

The transformation from Y-space to D-space is more than a simple flattening operation. It incorporates optimized strategies for register layout that consider the GPU's register file organization, the kernel's register pressure, and the access patterns of the computation. This transformation ensures that frequently accessed elements are kept in registers, that register bank conflicts are minimized, and that the compiler can generate efficient code for register access.

C++ Example:

```
// Y-to-D descriptor linearizes storage
auto d_idx = ys_to_d_descriptor.calculate_offset(y_idx);
```

6.7.4 Dimension Types

H-Dimensions (Hierarchical Dimensions)

The concept of Hierarchical Dimensions, or H-dimensions, represents one of the most key aspects of the CK framework's approach to work distribution. These dimensions encode a multi-level decomposition strategy that mirrors the hierarchical nature of GPU hardware, from individual vector operations up through threads, warps, and thread blocks. Each H-dimension group captures how a single tensor dimension is partitioned across these hardware levels, enabling fine-grained control over data access patterns and computational efficiency.

The structure of H-dimensions follows a specific pattern that reflects the GPU's execution hierarchy. Each H-dimension is expressed as a sequence of factors, where each factor corresponds to a specific level of the hierarchy. Consider the example `sequence<4, 2, 8, 4>`. This seemingly simple sequence encodes a advanced distribution strategy: the

rightmost factor (4) represents vector width, indicating that each memory operation processes 4 elements simultaneously. Moving left, the factor 8 indicates that 8 threads within a warp collaborate on the data. The factor 2 specifies that 2 warps within a block work together. Finally, the leftmost factor 4 indicates that each thread performs 4 iterations, enabling instruction-level parallelism and register reuse.

This hierarchical decomposition enables critical optimizations. By explicitly encoding the distribution strategy at compile time, the framework can generate code that perfectly matches the hardware's capabilities. The vector width aligns with the GPU's memory transaction size. The thread count per warp matches the hardware's SIMD width. The warp count per block balances parallelism with resource constraints. The repetition factor enables loop unrolling and software pipelining. Together, these factors create a distribution strategy that achieves near-optimal performance.

C++ Example:

```
using HsLengthss = tuple<
    sequence<4, 2, 8, 4>,  // H0: M dimension
    sequence<4, 2, 8, 4>   // H1: N dimension
>;
```

RH-Dimensions (R + H Dimensions Combined)

The RH-dimensions represent the unified coordinate space that combines both replication (R) and hierarchical (H) dimensions into a single, coherent framework. This combined space serves as the internal representation used by the coordinate transformation machinery, enabling seamless handling of both replicated and non-replicated data patterns. The unification of these dimensions simplifies the mathematical framework while maintaining the flexibility to express complex distribution strategies.

Within the RH-dimension framework, coordinates are identified by two components: major and minor indices. The major index identifies which dimension group a coordinate belongs to, with 0 reserved for R-dimensions and subsequent values (1, 2, ...) identifying H-dimension groups. The minor index specifies the position within the identified group. This two-level addressing scheme enables efficient navigation through the combined coordinate space while maintaining clear separation between replication and hierarchical decomposition strategies.

The power of RH-dimensions becomes apparent when considering complex algorithms that require both data replication and hierarchical distribution. By providing a unified coordinate system, the framework can express transformations that simultaneously handle replicated data sharing and hierarchical work distribution, all within a single mathematical formalism. This unification is key to achieving both expressiveness and efficiency in the CK framework.

6.7.5 Transformations

Adaptor

An adaptor in the CK framework represents a advanced chain of coordinate transformations that bridges different coordinate spaces. Rather than simple one-to-one mappings, adaptors embody complex mathematical transformations that can involve permutations, embeddings, projections, and non-linear mappings. These transformations are composed at compile time, enabling the generation of highly optimized code that performs the complete transformation in a single step without intermediate representations. For detailed information about adaptors and their implementation, see *Tensor Adaptors - Chaining Transformations*.

The framework provides several specialized adaptor types, each serving a specific role in the coordinate transformation pipeline. The `ps_ys_to_xs_adaptor` performs the critical transformation from processing element and yield space coordinates to physical tensor coordinates, implementing the core logic of tile distribution. This adaptor encodes decisions about how threads are assigned to data, how data is traversed within each thread's assignment, and how these patterns map to the global tensor layout. Similarly, the `ys_to_d_adaptor` handles the transformation from multi-dimensional yield space to linearized data space, optimizing the layout of data in thread-local registers.

The power of adaptors lies in their composability. Complex transformations can be built by chaining simpler adaptors, with the framework automatically optimizing the composition. This design enables the expression of advanced access

patterns—such as transposed access, strided access, or space-filling curves—through the composition of elementary transformations. The compile-time nature of this composition ensures zero runtime overhead while maintaining mathematical clarity.

C++ Type: `tensor_adaptor<...>`

Descriptor

A descriptor in CK provides a complete specification of tensor layout, encompassing not just the logical structure of the data but also all transformations and physical memory layout details. This comprehensive specification serves as the contract between different components of the system, ensuring that all parts of a kernel have a consistent view of how data is organized and accessed. Descriptors combine multiple aspects of tensor representation: the logical shape and dimensions, the physical memory layout including padding and alignment, the coordinate transformations for different access patterns, and optimization hints for the compiler. For comprehensive coverage of descriptors, see *Tensor Descriptors - Complete Tensor Specifications*.

The sophistication of descriptors enables them to represent complex data layouts that arise in real-world applications. A descriptor might specify that a logically 4D tensor is physically stored with padding for alignment, uses a custom stride pattern for the channel dimension, and should be accessed using a space-filling curve for optimal cache utilization. All these details are encoded in the descriptor's type, enabling compile-time verification and optimization.

Descriptors play a crucial role in achieving performance portability. By abstracting the details of data layout behind a well-defined interface, descriptors enable algorithms to be written once and automatically adapted to different data layouts. This abstraction is particularly valuable when dealing with different hardware architectures that may have different alignment requirements, cache line sizes, or memory access patterns.

C++ Type: `tensor_descriptor<...>`

6.7.6 Operations

Load Tile

The load tile operation represents a fundamental building block of GPU kernel design in the CK framework, orchestrating the complex process of transferring data from global memory to thread-local registers. This operation is far more advanced than a simple memory copy—it implements the complete distribution strategy encoded in the tile distribution, ensuring that each thread loads exactly the data it needs for its portion of the computation. The load operation automatically handles memory coalescing to maximize bandwidth utilization, coordinates between threads to avoid redundant loads, manages boundary conditions for tiles that extend beyond tensor bounds, and optimizes the access pattern based on the specific distribution strategy.

The efficiency of the load tile operation stems from its deep integration with the distribution framework. By knowing at compile time exactly which threads will access which data elements, the operation can generate optimal memory access patterns that fully utilize the GPU's memory subsystem. For matrix multiplication, this might mean loading data in a pattern that ensures perfect coalescing. For convolution, it might involve complex patterns that minimize the number of redundant loads while respecting the GPU's cache hierarchy.

C++ Function: `tile_window.load()`

Store Tile

The store tile operation provides the complementary functionality to load tile, transferring computed results from thread-local registers back to global memory. Like its counterpart, the store operation implements optimized strategies that go beyond simple memory writes. It ensures that writes are coalesced for maximum bandwidth efficiency, coordinates between threads to handle overlapping write regions correctly, manages atomic operations when multiple threads write to the same location, and optimizes write patterns to minimize memory traffic.

The store operation must handle additional complexities compared to loads. While loads can often ignore synchronization issues (reading stale data is usually harmless), stores must ensure correctness when multiple threads write to overlapping regions. The framework provides different store modes for different scenarios: exclusive stores where

each element is written by exactly one thread, atomic stores where multiple threads may update the same element, and reduction stores where partial results are accumulated. The choice of store mode is encoded in the distribution strategy and verified at compile time.

C++ Function: `tile_window.store(tile)`

Sweep Tile

The sweep tile operation embodies a key programming paradigm for distributed tensor computation, providing a high-level iteration abstraction over the complex distribution patterns. Rather than requiring manual index calculations and nested loops, sweep tile automatically visits each element in a distributed tensor exactly once, invoking a user-provided function with the appropriate coordinates. This abstraction hides the complexity of the distribution while enabling advanced optimizations such as automatic loop unrolling, software pipelining, and register rotation.

The implementation of sweep tile leverages the compile-time knowledge of the distribution pattern to generate highly optimized iteration code. For simple distributions, this might result in a single unrolled loop. For complex hierarchical distributions, it might generate nested loops with carefully chosen iteration orders that maximize data reuse and minimize register pressure. The beauty of the abstraction is that these optimizations happen transparently—the user simply provides the computation to perform on each element, and the framework handles the rest.

C++ Function: `sweep_tile(tensor, lambda)`

Shuffle Tile

The shuffle tile operation provides efficient intra-warp communication, enabling threads within a warp to exchange data without going through shared memory. This operation leverages the GPU's hardware shuffle instructions, which allow any thread in a warp to read registers from any other thread in the same warp. Shuffle operations are particularly valuable for reduction operations, transpose operations within a warp, and collaborative loading patterns where threads cooperate to load contiguous data and then redistribute it according to the computation pattern.

The framework provides various shuffle patterns optimized for different use cases. Butterfly shuffles enable efficient reductions and FFT-like operations. Broadcast shuffles allow one thread to share data with all others in the warp. Rotation shuffles enable cyclic data exchange patterns. The shuffle tile operation automatically selects the appropriate hardware instructions based on the data type and shuffle pattern, ensuring optimal performance while maintaining portability across different GPU architectures.

C++ Function: `shuffle_tile(tensor, shuffle_pattern)`

6.7.7 Memory Concepts

Coalescing

The property where adjacent threads access adjacent memory locations, maximizing memory bandwidth utilization.

Bank Conflict

A performance degradation that occurs when multiple threads in a warp access different addresses in the same memory bank. For detailed information about bank conflicts and mitigation strategies, see *Understanding AMD GPU LDS and Bank Conflicts*.

Vectorization

The technique of loading/storing multiple elements in a single memory transaction.

C++ Example:

```
// Vector load of 4 elements
using float4 = vector_type<float, 4>::type;
float4 data = tensor_view.template get_vectorized_elements<4>(x_idx);
```

6.7.8 Distribution Components

Window

A view into a subset of a tensor that respects the distribution pattern. For detailed information about tile windows and their usage, see *Tile Window - Data Access Gateway*.

C++ Type: `tile_window<...>`

Static Distributed Tensor

A thread-local tensor stored in registers, distributed according to a tile distribution. For in-depth coverage of static distributed tensors, see *Static Distributed Tensor*.

C++ Type: `static_distributed_tensor<...>`

Spans

Iteration ranges over distributed dimensions, used by sweep operations.

C++ Type: `tile_distributed_span<...>`

6.7.9 GPU Hardware Terms

Warp

A group of threads (32 on AMD GPUs) that execute in lockstep.

Lane

An individual thread within a warp (0-31).

Block

A group of warps that can cooperate through shared memory.

Grid

The complete set of blocks launched for a kernel.

6.7.10 Template Parameters

sequence<...>

A compile-time integer sequence used to specify dimensions and lengths.

Example: `sequence<256, 256>` for a 256×256 tile

tuple<...>

A heterogeneous collection of types, often used for grouping sequences.

Example: `tuple<sequence<4,4>, sequence<4,4>>`

number<N>

A compile-time integer constant.

Example: `number<16>` represents the value 16

6.7.11 Optimization Terms

Register Spilling

When a kernel uses more registers than available, causing data to spill to slower memory.

Occupancy

The ratio of active warps to maximum possible warps on a GPU multiprocessor.

Memory Bandwidth Utilization

The percentage of theoretical memory bandwidth achieved by a kernel.

Instruction-Level Parallelism (ILP)

The ability to execute multiple independent instructions simultaneously.

6.7.12 Common Patterns

GEMM (General Matrix Multiplication)

A fundamental operation where C = A×B + C. For a complete optimization case study, see *A Block GEMM on MI300*.

Reduction

An operation that combines multiple values into a single result (e.g., sum, max).

Broadcast

An operation that replicates a value across multiple processing elements.

Transpose

An operation that swaps dimensions of a tensor.

6.7.13 Performance Metrics

FLOPS (Floating-Point Operations Per Second)

Measure of computational throughput.

Bandwidth

Rate of data transfer, typically measured in GB/s.

Latency

Time delay between issuing an operation and its completion.

Throughput

Rate of operation completion, often measured in operations per second.

6.7.14 Usage Examples

Creating a Distribution

```
// Define encoding
using MyEncoding = tile_distribution_encoding<
    sequence<>,                          // No replication
    tuple<sequence<4,2,8,4>,            // M dimension
          sequence<4,2,8,4>>,           // N dimension
    tuple<sequence<1,2>, sequence<1,2>>, // P mappings
    tuple<sequence<1,1>, sequence<2,2>>, // P minor
    sequence<1,1,2,2>,                   // Y major
    sequence<0,3,0,3>                    // Y minor
>;

// Create distribution
auto distribution = make_static_tile_distribution(MyEncoding{});
```

Using Tile Window

```
// Create window
auto window = make_tile_window(
    tensor_view,
    TileShape{},
    origin,
    distribution
);

// Load-compute-store pattern
auto tile = window.load();
sweep_tile(tile, compute_func);
window.store(tile);
```

6.7.15 Related Documentation

- *Introduction and Motivation - Why Tile Distribution Matters* - Introduction and motivation
- *Buffer Views - Raw Memory Access* - Raw memory access
- *Tile Distribution - The Core API* - Core distribution concepts

6.8 Tensor Adaptors - Chaining Transformations

6.8.1 Overview

While individual *transforms* are effective, TensorAdaptors enable the chaining of multiple transforms together to create complex coordinate transformations. Adaptors can be thought of as transformation pipelines that can reshape, reorder, and restructure tensors in advanced ways.

TensorAdaptors serve as the bridge between individual transforms and the high-level tensor operations used in applications. They provide a composable abstraction that allows developers to build complex data access patterns from simple building blocks.

6.8.2 TensorAdaptor Basics

A TensorAdaptor encapsulates a sequence of *coordinate transformations*, managing the flow of coordinates through multiple transform stages:

Core Components

Each TensorAdaptor contains:

- **transforms**: List of individual *transforms* to apply
- **lower_dimension_hidden_idss**: Mappings between transform stages
- **upper_dimension_hidden_idss**: Hidden dimension mappings for internal stages
- **bottom_dimension_hidden_ids**: Input dimension identifiers
- **top_dimension_hidden_ids**: Output dimension identifiers

The most important method of a TensorAdaptor is `calculate_bottom_index`, which calculates the lower index from the upper index by applying transforms in reverse order.

6.8.3 Transpose Adaptor: Dimension Reordering

The transpose adaptor reorders tensor dimensions according to a permutation pattern. This operation forms the basis for many tensor manipulations in GPU kernels.

```
// Create transpose adaptor: [0, 1, 2] → [2, 0, 1]
auto transpose_adaptor = make_identity_tensor_adaptor<3>();  // Start with identity

// Apply transpose using transform_tensor_adaptor
auto transposed_desc = transform_tensor_descriptor(
    original_desc,
    make_tuple(make_pass_through_transform(original_desc.get_length(2)),
               make_pass_through_transform(original_desc.get_length(0)),
               make_pass_through_transform(original_desc.get_length(1))),
    make_tuple(sequence<2>{}, sequence<0>{}, sequence<1>{}),  // old dims
    make_tuple(sequence<0>{}, sequence<1>{}, sequence<2>{})   // new dims
);

// Alternative: Direct coordinate transformation
multi_index<3> top_coord{0, 1, 2};
// After transpose [2, 0, 1]: coord becomes [2, 0, 1]
```

6.8.4 Single-Stage Adaptors: Custom Transform Chains

Custom adaptors can be created by specifying which transforms to use and how they connect. This provides fine-grained control over the transformation pipeline:

```
// Create a descriptor that merges 2x3 dimensions into single dimension
auto base_desc = make_naive_tensor_descriptor_packed(make_tuple(2, 3));

// Apply merge transform
auto merged_desc = transform_tensor_descriptor(
    base_desc,
    make_tuple(make_merge_transform(make_tuple(2, 3))),
    make_tuple(sequence<0, 1>{}),  // merge dims 0,1
```

(continues on next page)

(continued from previous page)

```
    make_tuple(sequence<0>{})      // to single dim 0
);

// The adaptor is embedded in the descriptor
// To use it:
multi_index<1> top_coord{5};  // 1D coordinate
// This internally calculates: row = 5/3 = 1, col = 5%3 = 2
```

6.8.5 Chaining Adaptors: Building Complex Transformations

The real power of adaptors comes from chaining multiple transformations together to create advanced data access patterns:

```
// Start with a 2D descriptor
auto desc1 = make_naive_tensor_descriptor_packed(make_tuple(2, 3));

// First transformation: merge 2D to 1D
auto merged_desc = transform_tensor_descriptor(
    desc1,
    make_tuple(make_merge_transform(make_tuple(2, 3))),
    make_tuple(sequence<0, 1>{}),  // merge dims 0,1
    make_tuple(sequence<0>{})      // to dim 0
);

// Second transformation: unmerge 1D back to 2D
auto final_desc = transform_tensor_descriptor(
    merged_desc,
    make_tuple(make_unmerge_transform(make_tuple(2, 3))),
    make_tuple(sequence<0>{}),     // from dim 0
    make_tuple(sequence<0, 1>{})   // to dims 0,1
);

// The chained transformation is embedded in final_desc
// Result should be identity transformation
```

6.8.6 Transform Addition: Extending Existing Adaptors

Existing adaptors can be extended with new transforms using `transform_tensor_adaptor`. This pattern is useful for adding padding or other modifications to existing transformation pipelines:

```
// Start with transposed descriptor
auto base_desc = make_naive_tensor_descriptor(
    make_tuple(3, 4),
    make_tuple(1, 3)   // transposed strides
);

// Add padding to both dimensions
auto padded_desc = transform_tensor_descriptor(
    base_desc,
    make_tuple(make_pad_transform(3, 1, 1),   // pad dim 0: 3 → 5
               make_pad_transform(4, 0, 0)),   // keep dim 1: 4 → 4
    make_tuple(sequence<0>{}, sequence<1>{}),  // input dims
```

(continues on next page)

(continued from previous page)

```
    make_tuple(sequence<0>{}, sequence<1>{})   // output dims (keep 2D)
);

// Access pattern
multi_index<2> padded_coord{1, 2};  // In padded space
// Internally calculates: unpadded = [1-1, 2] = [0, 2]
// Then applies transpose strides
```

6.8.7 Advanced Patterns

Complex Nested Transforms

CK Tile supports complex nested transform patterns that enable advanced data layouts:

```
// Example: 4D tensor with complex transformations
// Shape: [A, B, C, D] with various transforms

// 1. Create base descriptor
auto base_desc = make_naive_tensor_descriptor_packed(
    make_tuple(A, B, C, D)
);

// 2. Apply multiple transformations
// First: merge first 3 dimensions
auto step1_desc = transform_tensor_descriptor(
    base_desc,
    make_tuple(make_merge_transform(make_tuple(A, B, C)),
               make_pass_through_transform(D)),
    make_tuple(sequence<0, 1, 2>{}, sequence<3>{}),  // input mapping
    make_tuple(sequence<0>{}, sequence<1>{})         // output: 2D
);

// 3. Then unmerge back but with different grouping
auto step2_desc = transform_tensor_descriptor(
    step1_desc,
    make_tuple(make_unmerge_transform(make_tuple(A*B, C)),
               make_pass_through_transform(D)),
    make_tuple(sequence<0>{}, sequence<1>{}),        // from 2D
    make_tuple(sequence<0, 1>{}, sequence<2>{})      // to 3D
);

// The adaptor chain is embedded in the descriptors
// CK optimizes these at compile time
```

GPU Memory Layout Example

A practical example showing how adaptors create efficient *GPU memory access patterns*:

```
// Create descriptor for thread block tile: 64x64
// With 8x8 vector loads per thread
constexpr auto BlockM = 64;
constexpr auto BlockN = 64;
```

(continues on next page)

(continued from previous page)

```
constexpr auto VectorM = 8;
constexpr auto VectorN = 8;

// Thread arrangement: 8x8 threads
constexpr auto ThreadM = BlockM / VectorM;  // 8
constexpr auto ThreadN = BlockN / VectorN;  // 8

// Create block descriptor with proper layout
auto block_desc = transform_tensor_descriptor(
    make_naive_tensor_descriptor_packed(
        make_tuple(number<BlockM>{}, number<BlockN>{})
    ),
    make_tuple(
        make_unmerge_transform(make_tuple(
            number<ThreadM>{}, number<VectorM>{}
        )),
        make_unmerge_transform(make_tuple(
            number<ThreadN>{}, number<VectorN>{}
        ))
    ),
    make_tuple(sequence<0>{}, sequence<1>{}),           // from 2D
    make_tuple(sequence<0, 2>{}, sequence<1, 3>{})     // to 4D: [TM,TN,VM,VN]
);

// This creates the layout:
// - Dimension 0,1: Thread indices
// - Dimension 2,3: Vector indices within thread
// Enables coalesced memory access on GPU
```

6.8.8 Common Transform Chains

CK Tile provides several common transform chain patterns used throughout GPU kernels:

Padding for Convolution

```
auto padded = transform_tensor_descriptor(
    input,
    make_tuple(make_pad_transform(H, pad_h, pad_h),
               make_pad_transform(W, pad_w, pad_w)),
    make_tuple(sequence<0>{}, sequence<1>{}),
    make_tuple(sequence<0>{}, sequence<1>{})
);
```

Dimension Merging for GEMM

```
auto merged = transform_tensor_descriptor(
    input,
    make_tuple(make_merge_transform(make_tuple(M, K))),
    make_tuple(sequence<0, 1>{}),
    make_tuple(sequence<0>{})
);
```

For complete GEMM optimization strategies, see *A Block GEMM on MI300*.

Broadcasting for Elementwise Operations

```
auto broadcast = transform_tensor_descriptor(
    scalar,
    make_tuple(make_replicate_transform(make_tuple(M, N))),
    make_tuple(sequence<>{}),
    make_tuple(sequence<0, 1>{})
);
```

6.8.9 Key Concepts Summary

TensorAdaptors are the coordination layer that makes complex tensor operations possible:

- **Identity Adaptor**: Starting point for building transformations
- **Transpose Adaptor**: Dimension reordering with permutation patterns
- **Single-Stage Adaptors**: Custom transform chains with precise control
- **Chained Adaptors**: Complex multi-stage transformation pipelines
- **Transform Addition**: Extending existing adaptors with new transforms

Core concepts to remember:

- **Bottom/Top Dimensions**: Input and output coordinate spaces
- **Hidden Dimensions**: Internal coordinate mappings between transforms
- **Transform Chains**: Sequential application of multiple transforms
- **Coordinate Transformation**: Bidirectional mapping between coordinate spaces
- **Nested Transforms**: Complex multi-level transformation hierarchies

6.8.10 Key C++ Patterns in Composable Kernel

1. **Descriptor-Based Adaptors**: In CK, adaptors are typically embedded within *tensor descriptors* rather than created separately
2. **Compile-Time Optimization**: All transformations are resolved at compile time for zero overhead
3. **Type Safety**: Template metaprogramming ensures coordinate transformations are type-safe
4. **GPU Optimization**: Transform chains are designed for efficient GPU memory access patterns. See *Understanding AMD GPU LDS and Bank Conflicts* for LDS optimization.

TensorAdaptors bridge the gap between low-level transforms and high-level tensor operations, providing the flexibility to create advanced data layouts and access patterns that are essential for efficient GPU computing. They build upon the foundation of *BufferViews* and *TensorViews* to provide complex transformation capabilities.

6.8.11 Next Steps

- *Tensor Descriptors - Complete Tensor Specifications* - How adaptors combine with element space to form complete tensor descriptors
- *Individual Transform Operations* - Individual transform types and their properties
- *Tile Window - Data Access Gateway* - How adaptors enable efficient data loading patterns
- *Space-Filling Curves - Optimal Memory Traversal* - Advanced coordinate mapping techniques for cache optimization
- *Static Distributed Tensor* - How adaptors help manage distributed tensor storage

6.9 Individual Transform Operations

The transformation engine is built from individual transform types that each handle specific coordinate conversions.

6.9.1 What Are Transforms?

Transform operations convert coordinates between different dimensional spaces. Each transform operates between two *coordinate spaces*:

- **Lower Dimension Space**: The source coordinate system
- **Upper Dimension Space**: The target coordinate system

Transform Direction

Transforms work bidirectionally:

- **Forward Transform**: Converts coordinates from the lower dimension to the upper dimension
- **Inverse Transform**: Converts coordinates back from the upper dimension to the lower dimension

Zero-Copy Logical Operations

Critical Understanding: All transform operations happen in **logical coordinate space** only. This is a zero-copy system and there is **no data copying or movement** involved.

- **Data Storage**: The actual tensor data remains stored in memory in linear fashion, exactly as specified by the original tensor shape and strides at creation time. See *Buffer Views - Raw Memory Access* for more information about raw memory access.
- **Logical Mapping**: Transforms create different logical views of the same underlying data and only change how access coordinates are interpreted. See *Tensor Views - Multi-Dimensional Structure* for more information about tensor views.

Index Calculation Operations

The transform system provides two operations for coordinate conversion:

- **calculate_lower_index()**: Takes a coordinate from the **upper dimension space** and transforms it to get the corresponding index or coordinate in the **lower dimension space**. This calculates where to find the actual tensor element using the transformed coordinate system.
- **calculate_upper_index()**: Takes a coordinate from the **lower dimension space** and transforms it back to get the corresponding coordinate in the **upper dimension space**. This performs the inverse transformation to recover the original coordinate representation.

These operations enable bidirectional navigation between different coordinate representations of the same underlying tensor data.

Transform System Architecture

6.9.2 MergeTransform

MergeTransform collapses multiple dimensions from the lower coordinate space into a single dimension in the upper coordinate space, effectively reducing the dimensionality of the tensor representation while preserving data relationships. This transform is fundamental to the *tile distribution system*.

C++ Implementation:

```
using namespace ck_tile;

// Create MergeTransform for 4x5 tensor (20 elements total)
auto transform = make_merge_transform(make_tuple(4, 5));

// Forward: Lower (2D) → Upper (1D) - Manual calculation
int row = 2, col = 3;
int linear_index = row * 5 + col;  // Result: 13
printf("2D coord [%d, %d] → Linear index %d\n", row, col, linear_index);
printf("Calculation: %d×5 + %d = %d\n", row, col, linear_index);

// Inverse: Upper (1D) → Lower (2D) - Using transform
multi_index<1> upper_coord;
upper_coord[number<0>{}] = 13;

multi_index<2> lower_coord;
transform.calculate_lower_index(lower_coord, upper_coord);

printf("Linear index %d → 2D coord [%d, %d]\n",
       static_cast<int>(upper_coord[number<0>{}]),
       static_cast<int>(lower_coord[number<0>{}]),
       static_cast<int>(lower_coord[number<1>{}]));
printf("Calculation: 13 ÷ 5 = %d remainder %d\n",
       static_cast<int>(lower_coord[number<0>{}]),
       static_cast<int>(lower_coord[number<1>{}]));
```

6.9.3 UnmergeTransform

UnmergeTransform expands coordinates from a single dimension in the lower coordinate space into multiple dimensions in the upper coordinate space, effectively increasing the dimensionality of the tensor representation while preserving all data relationships.

C++ Implementation:

```
using namespace ck_tile;

// Create UnmergeTransform for 3x4x2 tensor (24 elements total)
auto transform = make_unmerge_transform(make_tuple(3, 4, 2));

// Forward: Lower (1D) → Upper (3D) - Manual calculation
int linear_index = 14;
int dim0 = linear_index / (4 * 2);      // 14 / 8 = 1
int remainder = linear_index % (4 * 2);  // 14 % 8 = 6
int dim1 = remainder / 2;               // 6 / 2 = 3
int dim2 = remainder % 2;               // 6 % 2 = 0

printf("Linear index %d → 3D coord [%d, %d, %d]\n",
       linear_index, dim0, dim1, dim2);
printf("Calculation: 14 = %d×8 + %d×2 + %d\n", dim0, dim1, dim2);

// Inverse: Upper (3D) → Lower (1D) - Using transform
multi_index<3> upper_coord;
upper_coord[number<0>{}] = 1;
```

(continues on next page)

(continued from previous page)

```
upper_coord[number<1>{}] = 3;
upper_coord[number<2>{}] = 0;

multi_index<1> lower_coord;
transform.calculate_lower_index(lower_coord, upper_coord);

printf("3D coord [%d, %d, %d] → Linear index %d\n",
       static_cast<int>(upper_coord[number<0>{}]),
       static_cast<int>(upper_coord[number<1>{}]),
       static_cast<int>(upper_coord[number<2>{}]),
       static_cast<int>(lower_coord[number<0>{}]));
printf("Calculation: %d×8 + %d×2 + %d = %d\n",
       static_cast<int>(upper_coord[number<0>{}]),
       static_cast<int>(upper_coord[number<1>{}]),
       static_cast<int>(upper_coord[number<2>{}]),
       static_cast<int>(lower_coord[number<0>{}]));
```

6.9.4 EmbedTransform

EmbedTransform expands linear indices from the lower coordinate space into multi-dimensional coordinates in the upper coordinate space using configurable strides, enabling flexible strided tensor layouts and sub-tensor views within larger buffers.

C++ Implementation:

```
using namespace ck_tile;

// Create embed transform for 2x3 tensor with strides [12, 1]
// This is commonly used in descriptors
auto transform = make_embed_transform(make_tuple(2, 3), make_tuple(12, 1));

// Forward: Linear → 2D (Manual calculation)
int linear_idx = 14;
int row = linear_idx / 12;        // 14 / 12 = 1
int remainder = linear_idx % 12;  // 14 % 12 = 2
int col = remainder / 1;          // 2 / 1 = 2
printf("Linear index %d → 2D coord [%d, %d]\n", linear_idx, row, col);

// Inverse: 2D → Linear (Using transform)
multi_index<2> upper_coord;
upper_coord[number<0>{}] = 1;
upper_coord[number<1>{}] = 2;

multi_index<1> lower_coord;
transform.calculate_lower_index(lower_coord, upper_coord);
printf("2D coord [%d, %d] → Linear index %d\n",
       static_cast<int>(upper_coord[number<0>{}]),
       static_cast<int>(upper_coord[number<1>{}]),
       static_cast<int>(lower_coord[number<0>{}]));
```

6.9.5 ReplicateTransform

ReplicateTransform creates a higher-dimensional tensor by replicating (broadcasting) a lower-dimensional tensor. It's essentially a broadcasting operation that takes a tensor with fewer dimensions and logically replicates it across new dimensions without data duplication. An example is taking a scalar (0-dimensional) input and broadcasting it across multiple dimensions, enabling efficient broadcasting patterns where a single value appears at every position in a multi-dimensional coordinate space.

C++ Implementation:

```
using namespace ck_tile;

// Create replicate transform for 3x4 broadcasting
auto transform = make_replicate_transform(make_tuple(3, 4));

// Inverse: Upper (2D) → Lower (0D) - Using transform
// Any 2D coordinate maps to empty scalar coordinate
multi_index<2> upper_coord;
upper_coord[number<0>{}] = 1;
upper_coord[number<1>{}] = 2;

multi_index<0> lower_coord;  // Empty coordinate (0 dimensions)
transform.calculate_lower_index(lower_coord, upper_coord);
printf("2D [%d, %d] → Empty scalar [] (always empty)\n",
       static_cast<int>(upper_coord[number<0>{}]),
       static_cast<int>(upper_coord[number<1>{}]));

// Forward: Scalar → 2D (Conceptual - no coordinate calculation needed)
// Broadcasting: Single scalar value appears at ALL positions
printf("Broadcasting: Scalar value appears at every [i,j] where 0i<3, 0j<4\n");
printf("Total positions: 3×4 = 12 positions, all contain same scalar value\n");

// Test multiple coordinates - all map to empty scalar
int test_coords[][2] = {{0, 0}, {1, 2}, {2, 3}};
for(int i = 0; i < 3; i++)
{
    multi_index<2> test_upper;
    test_upper[number<0>{}] = test_coords[i][0];
    test_upper[number<1>{}] = test_coords[i][1];

    multi_index<0> test_lower;
    transform.calculate_lower_index(test_lower, test_upper);
    printf("2D [%d, %d] → Empty scalar []\n",
           test_coords[i][0], test_coords[i][1]);
}
```

6.9.6 OffsetTransform

OffsetTransform shifts coordinates by a fixed offset, creating a translated view of the coordinate space. It performs translation operations where each coordinate in the upper space is mapped to a coordinate in the lower space by adding a constant offset.

C++ Implementation:

```
using namespace ck_tile;

// Create offset transform for coordinate translation
// CK Tile formula: lower = upper + offset
auto transform = make_offset_transform(48, 16);

// Using Transform: Original → Translated (adds offset)
multi_index<1> upper_coord;
upper_coord[number<0>{}] = 5;  // Original index 5

multi_index<1> lower_coord;
transform.calculate_lower_index(lower_coord, upper_coord);
printf("Original index %d → Translated index %d\n",
       static_cast<int>(upper_coord[number<0>{}]),
       static_cast<int>(lower_coord[number<0>{}]));
printf("Calculation: %d + 16 = %d\n",
       static_cast<int>(upper_coord[number<0>{}]),
       static_cast<int>(lower_coord[number<0>{}]));

// Manual Reverse: Translated → Original (subtract offset)
int translated_idx = 21;
int original_idx = translated_idx - 16;
printf("Translated index %d → Original index %d\n", translated_idx, original_idx);

// Test multiple coordinates
int test_indices[] = {0, 10, 20, 47};
for(int i = 0; i < 4; i++)
{
    multi_index<1> test_upper;
    test_upper[number<0>{}] = test_indices[i];

    multi_index<1> test_lower;
    transform.calculate_lower_index(test_lower, test_upper);
    printf("Original %d → Translated %d\n",
           test_indices[i], static_cast<int>(test_lower[number<0>{}]));
}
```

6.9.7 PassThroughTransform - Identity

No-op transform that passes coordinates unchanged. The PassThrough transform is the simplest coordinate transformation in CK Tile, implementing a perfect identity mapping where input coordinates are passed through unchanged to the output. This transform is essential as a placeholder in transformation chains and for dimensions that require no modification.

C++ Implementation:

```
using namespace ck_tile;

// Identity transform - no change
int length = 60;

auto transform = make_pass_through_transform(length);
```

(continues on next page)

(continued from previous page)

```
printf("Length: %d\n", length);

// Forward: Upper → Lower (identity)
multi_index<1> upper_coord;
upper_coord[number<0>{}] = 25;

multi_index<1> lower_coord;
transform.calculate_lower_index(lower_coord, upper_coord);

printf("\nForward: [%d] → [%d] (unchanged)\n",
       static_cast<int>(upper_coord[number<0>{}]),
       static_cast<int>(lower_coord[number<0>{}]));

// Reverse: Lower → Upper (identity)
multi_index<1> lower_input;
lower_input[number<0>{}] = 42;

multi_index<1> upper_result;
// Note: PassThrough is bidirectional identity, so we can use same function
transform.calculate_lower_index(upper_result, lower_input);

printf("Reverse: [%d] → [%d] (unchanged)\n",
       static_cast<int>(lower_input[number<0>{}]),
       static_cast<int>(upper_result[number<0>{}]));
```

6.9.8 PadTransform

PadTransform adds padding to tensor dimensions, mapping coordinates from upper dimension space (with padding) to lower dimension space (original data).

C++ Implementation:

```
using namespace ck_tile;

// PadTransform for coordinate padding
int low_length = 3;   // Original dimension length
int left_pad = 1;     // Padding on left
int right_pad = 1;    // Padding on right

auto transform = make_pad_transform(low_length, left_pad, right_pad);

printf("Low length: %d\n", low_length);
printf("Left pad: %d\n", left_pad);
printf("Right pad: %d\n", right_pad);
printf("Upper length: %d (total with padding)\n", low_length + left_pad + right_pad);

// Test coordinate mapping
int test_coords[] = {0, 1, 2, 3, 4};
for(int i = 0; i < 5; i++)
{
    multi_index<1> upper;
    upper[number<0>{}] = test_coords[i];
```

(continues on next page)

(continued from previous page)

```
    multi_index<1> lower;
    transform.calculate_lower_index(lower, upper);

    int adjusted_idx = static_cast<int>(lower[number<0>{}]);
    bool is_valid = (adjusted_idx >= 0 && adjusted_idx < low_length);

    printf("Upper %d → Lower %d (%s)\n",
           test_coords[i], adjusted_idx,
           is_valid ? "valid" : "padding");
}
```

6.9.9 Additional Transform Types

XorTransform

XorTransform applies a 2D XOR mapping for specialized memory access patterns. It performs XOR operations on coordinates to create transformed memory layouts for specific algorithmic optimizations, particularly useful for avoiding *LDS bank conflicts*.

SliceTransform

SliceTransform extracts a sub-region from a tensor dimension.

ModuloTransform

ModuloTransform applies cyclic wrapping to coordinates using modulo operations.

6.9.10 Summary

Individual transforms provide:

- **Modularity**: Each transform does one thing
- **Composability**: Chain transforms for complex mappings (see *Tensor Adaptors - Chaining Transformations*)
- **Efficiency**: Compile-time optimization in C++
- **Flexibility**: Handle any coordinate conversion need

These transforms enable you to:

1. Create custom tensor views
2. Implement efficient data access patterns
3. Handle padding and boundaries correctly
4. Optimize memory layouts for *GPU access*

The C++ implementations in Composable Kernel provide:

- Zero-overhead abstractions through templates
- Compile-time composition and optimization
- Support for complex coordinate transformations
- Integration with GPU kernel generation
- Foundation for *tile windows* and *load/store traits*

6.9.11 Next Steps

- *Tensor Adaptors - Chaining Transformations* - How to chain transforms together
- *Tensor Descriptors - Complete Tensor Specifications* - Complete tensor descriptions with transforms
- *Tile Window - Data Access Gateway* - Using transforms for efficient data loading
- *Thread Mapping - Connecting to Hardware* - How transforms enable thread-to-data mapping
- *A Block GEMM on MI300* - Practical application in GEMM kernels

6.10 Tensor Descriptors - Complete Tensor Specifications

6.10.1 Overview

A TensorDescriptor is the complete blueprint for a tensor. It combines a shape, stride information, and a series of *transformations* into a single object that defines exactly how a tensor's data is laid out in memory. This specification enables CK Tile to create complex tensor views without any data movement.

In CK Tile, TensorDescriptors serve as the foundation for all tensor operations, providing:

- **Memory Layout Specification**: How data is arranged in physical memory
- **Logical View Definition**: How the tensor appears to the programmer
- **Transformation Pipeline**: A series of *coordinate transformations*
- **Zero-Copy Views**: Different logical representations of the same data, building on *BufferViews* and *TensorViews*

6.10.2 Creating Basic Tensor Layouts

CK Tile provides several ways to create tensor descriptors for common memory layouts.

Custom Strides

The most fundamental way to define a tensor is with custom strides. This provides full control over how many elements to "jump" in memory to move to the next item along each dimension. This is particularly useful for creating padded layouts required by GPU algorithms.

```
using namespace ck_tile;

// Create a 3x4 tensor, but make each row take up 8 elements in memory
// (4 for data, 4 for padding)
constexpr auto M = 3;
constexpr auto N = 4;
constexpr auto RowStride = 8;  // Padded stride

auto descriptor = make_naive_tensor_descriptor(
    make_tuple(M, N),           // Shape: [3, 4]
    make_tuple(RowStride, 1)    // Strides: [8, 1]
);

// The total memory needed is 3 rows * 8 elements/row = 24
constexpr auto element_space_size = M * RowStride;

// Calculate offset of the element at [row=1, col=2]
multi_index<2> coord{1, 2};
```

(continues on next page)

(continued from previous page)

```
auto offset = descriptor.calculate_offset(coord);
// offset = 1*8 + 2*1 = 10
```

Packed Row-Major Layout

For most cases, a tightly packed, row-major layout is sufficient. The strides are calculated automatically, leaving no unused space between elements.

```
using namespace ck_tile;

// Create a packed 3x4 tensor
auto descriptor_packed = make_naive_tensor_descriptor_packed(
    make_tuple(3, 4)
);

// Total memory is 3 * 4 = 12 elements
// Strides are automatically [4, 1] for row-major layout

// Calculate offset of the element at [row=1, col=2]
multi_index<2> coord{1, 2};
auto offset = descriptor_packed.calculate_offset(coord);
// offset = 1*4 + 2*1 = 6
```

Aligned Layout

For GPU performance, memory layouts often need to be aligned. This function creates a row-major layout but ensures that each row's starting address is a multiple of a given alignment value, adding padding if necessary.

```
using namespace ck_tile;

// Create a 4x5 tensor with 8-element alignment
constexpr auto align = 8;  // Align each row to 8-element boundary

auto descriptor_aligned = make_naive_tensor_descriptor_aligned(
    make_tuple(4, 5),
    align
);

// Without alignment, size would be 4*5=20
// With alignment, the row stride becomes 8 (smallest multiple of 8 >= 5)
// Total size = 4 rows * 8 elements/row = 32
```

6.10.3 The Pipeline Concept

Every TensorDescriptor in CK Tile can be thought of as a **transformation pipeline**. The functions above create the *first stage* of this pipeline, defining the initial *transformation* that takes a simple, one-dimensional block of memory and presents it as a logical, multi-dimensional tensor view.

The Initial Pipeline Stage

A simple packed descriptor sets up a pipeline with a single transform:

- **Input**: The raw, one-dimensional memory buffer (hidden dimension ID 0)
- **Output**: The logical dimensions that you interact with (hidden dimension IDs 1, 2, ...)

This initial stage converts linear memory addresses into multi-dimensional coordinates. See *Tensor Adaptors - Chaining Transformations* for how transforms chain together.

6.10.4 Advanced Layouts: Step-by-Step Transformation

The `transform_tensor_descriptor` function adds new stages to an existing descriptor's pipeline using *transforms*.

Transform a [2, 6] Tensor into a [2, 2, 3] View

This example reinterprets a 2D tensor with shape [2, 6] as a 3D tensor with shape [2, 2, 3], without changing the underlying 12-element memory buffer.

Step 1: Define the Base Descriptor

```
using namespace ck_tile;

// Create the [2, 6] base descriptor
auto base_descriptor = make_naive_tensor_descriptor_packed(
    make_tuple(2, 6)
);

// This creates an initial pipeline stage that:
// - Takes the raw buffer (hidden ID 0) as input
// - Produces two outputs (hidden IDs 1 and 2)
// - These outputs become logical dimensions 0 and 1
```

Step 2: Define the New Transformation Stage

To get from [2, 6] to [2, 2, 3], we need:

- **For logical dimension 0 (length 2)**: Preserve it with PassThroughTransform
- **For logical dimension 1 (length 6)**: Split it with UnmergeTransform([2, 3])

Step 3: Apply Transformation

```
// Create the transformed descriptor
auto transformed_descriptor = transform_tensor_descriptor(
    base_descriptor,
    make_tuple(
        make_pass_through_transform(2),      // For dim 0
        make_unmerge_transform(make_tuple(2, 3))  // For dim 1
    ),
    make_tuple(sequence<0>{}, sequence<1>{}),     // Input mapping
    make_tuple(sequence<0>{}, sequence<1, 2>{})   // Output mapping
);

// Result: A [2, 2, 3] view of the same data
```

Analysis of the Final Pipeline

The pipeline now has three stages:

1. **Base UnmergeTransform**: Converts raw buffer to [2, 6] layout
2. **PassThroughTransform**: Preserves the first dimension
3. **UnmergeTransform**: Splits the second dimension into [2, 3]

6.10.5 5D to 3D Block Transformation

These concepts are critical in *GPU programming*. This example transforms a 5D tensor representing a GPU thread block's workload into a simpler 3D view using MergeTransform. See *Thread Mapping - Connecting to Hardware* for thread distribution details.

```
using namespace ck_tile;

// Define parameters (typical for a GPU block)
constexpr auto Block_M = 256;
constexpr auto NumWarps = 8;
constexpr auto WarpSize = 64;
constexpr auto KVector = 4;
constexpr auto wavesPerK = 2;
constexpr auto wavesPerM = NumWarps / wavesPerK;
constexpr auto NumIssues = Block_M / wavesPerM;

// Create the base 5D descriptor
auto base_descriptor = make_naive_tensor_descriptor_packed(
    make_tuple(NumIssues, wavesPerM, wavesPerK, WarpSize, KVector)
);

// Transform to 3D by merging dimensions
auto transformed_descriptor = transform_tensor_descriptor(
    base_descriptor,
    make_tuple(
        make_pass_through_transform(NumIssues),
        make_merge_transform(make_tuple(wavesPerM, wavesPerK)),
        make_merge_transform(make_tuple(WarpSize, KVector))
    ),
    make_tuple(sequence<0>{}, sequence<1, 2>{}, sequence<3, 4>{}),
    make_tuple(sequence<0>{}, sequence<1>{}, sequence<2>{})
);

// Result: [NumIssues, wavesPerM*wavesPerK, WarpSize*KVector]
// This simplifies thread block management while preserving data layout
```

6.10.6 Common Descriptor Patterns

Matrix Transposition

```
// Create a transposed view of a matrix
auto transposed = transform_tensor_descriptor(
    original_matrix,
    make_tuple(
```

(continues on next page)

(continued from previous page)

```
        make_pass_through_transform(N),
        make_pass_through_transform(M)
    ),
    make_tuple(sequence<1>{}, sequence<0>{}),  // Swap dimensions
    make_tuple(sequence<0>{}, sequence<1>{})
);
```

Padding for Convolution

```
// Add padding to spatial dimensions
auto padded = transform_tensor_descriptor(
    input_tensor,
    make_tuple(
        make_pass_through_transform(N),    // Batch
        make_pass_through_transform(C),    // Channel
        make_pad_transform(H, pad_h, pad_h),  // Height
        make_pad_transform(W, pad_w, pad_w)   // Width
        ),
    make_tuple(sequence<0>{}, sequence<1>{}, sequence<2>{}, sequence<3>{}),
    make_tuple(sequence<0>{}, sequence<1>{}, sequence<2>{}, sequence<3>{})
    );
```

For a complete convolution example, see *Convolution Implementation with CK Tile*.

Tensor Slicing

```
// Extract a sub-tensor
auto slice = transform_tensor_descriptor(
    full_tensor,
    make_tuple(
        make_slice_transform(M, start_m, end_m),
        make_slice_transform(N, start_n, end_n)
    ),
    make_tuple(sequence<0>{}, sequence<1>{}),
    make_tuple(sequence<0>{}, sequence<1>{})
);
```

6.10.7 Key Concepts Summary

TensorDescriptors provide a key abstraction for tensor manipulation:

- **Pipeline Architecture**: Each descriptor is a transformation pipeline
- **Zero-Copy Views**: All transformations are logical, no data movement
- **Composability**: Complex layouts built from simple transforms
- **GPU Optimization**: Designed for efficient GPU memory access patterns

Important principles:

1. **Always Handle All Dimensions**: When transforming, provide a transform for each input dimension
2. **Hidden Dimension IDs**: Track the flow of data through the pipeline
3. **Compile-Time Resolution**: All transformations resolved at compile time

4. **Type Safety**: Template metaprogramming ensures correctness

6.10.8 Performance Considerations

When designing tensor descriptors for GPU kernels:

1. **Memory Coalescing**: Ensure contiguous threads access contiguous memory
2. **Bank Conflicts**: Avoid patterns that cause *shared memory conflicts*
3. **Alignment**: Use aligned layouts for better memory throughput
4. **Padding**: Strategic padding can improve access patterns. Ssee *Load Data Share Index Swapping* for advanced techniques.

6.10.9 Next Steps

- *Tile Window - Data Access Gateway* - Using descriptors for efficient data loading
- *Tile Distribution - The Core API* - How descriptors enable automatic work distribution
- *Convolution Implementation with CK Tile* - Real-world application of complex descriptors
- *Static Distributed Tensor* - Managing distributed tensors with descriptors
- *A Block GEMM on MI300* - GEMM kernels using descriptor transformations

6.11 Tile Window - Data Access Gateway

6.11.1 Overview

While *TileDistribution* determines the mapping between threads and tensor coordinates, TileWindow provides the mechanism for loading and storing data with memory access patterns. This abstraction encapsulates coalesced memory accesses, vectorization, and boundary handling into an interface.

TileWindow implements a distribution-aware windowing mechanism that views a subset of a larger tensor through the lens of a tile distribution. This windowing is a distribution-aware view that automatically generates memory access patterns for the underlying hardware. The system combines knowledge of the *tensor's layout*, the distribution pattern, and the *GPU's memory subsystem* characteristics to generate optimized load and store operations.

6.11.2 TileWindow Architecture

6.11.3 What is a TileWindow?

The challenge in GPU programming lies in the gap between logical tensor operations and the physical realities of memory access. While *TileDistribution* solves the problem of work assignment by mapping threads to *tensor coordinates*, it does not address how threads access the data at those coordinates. TileWindow serves as the critical bridge between logical work assignment and physical memory operations.

TileWindow implements a distribution-aware windowing mechanism that transforms abstract coordinate mappings into concrete memory access patterns. The abstraction takes into account the data elements each thread needs and also how to access them in a way that maximizes memory bandwidth utilization. This involves optimized techniques such as memory coalescing, where adjacent threads access adjacent memory locations, and vectorization, where multiple elements are loaded or stored in a single transaction.

C++ Implementation Overview:

```cpp
// From ck_tile/core/tensor/tile_window.hpp
#include <ck_tile/core/tensor/tile_window.hpp>
#include <ck_tile/core/tensor/static_distributed_tensor.hpp>
#include <ck_tile/core/algorithm/coordinate_transform.hpp>

template <typename TensorView_,
          typename WindowLengths_,
          typename TileDistribution_>
struct tile_window_with_static_distribution
{
    using TensorView = remove_cvref_t<TensorView_>;
    using Distribution = remove_cvref_t<TileDistribution_>;
    using DataType = typename TensorView::DataType;

    // Core components that define the window
    TensorView tensor_view_;      // View into the underlying tensor
    Distribution distribution_;    // How to distribute data across threads
    array<index_t, TensorView::get_num_of_dimension()> origin_;

    // Window-specific information
    static constexpr auto window_lengths = WindowLengths{};
    static constexpr index_t num_of_dimension = TensorView::get_num_of_dimension();

    // Constructor
    CK_TILE_HOST_DEVICE constexpr tile_window_with_static_distribution(
        const TensorView& tensor_view,
        const WindowLengths& /*window_lengths*/,
        const array<index_t, num_of_dimension>& origin,
        const Distribution& distribution)
        : tensor_view_{tensor_view},
          distribution_{distribution},
          origin_{origin}
    {}

    // Load operation with automatic coalescing
    template <typename DistributedTensor>
    CK_TILE_DEVICE void load(DistributedTensor& dst_tensor) const
    {
        // Sophisticated load implementation that:
        // 1. Calculates optimal access pattern
        // 2. Handles vectorization automatically
        // 3. Ensures coalesced memory access
        // 4. Manages boundary conditions
    }
};
```

6.11.4 LoadStoreTraits - The Access Pattern Engine

Behind every TileWindow operation lies *LoadStoreTraits*, a compile-time analysis engine that determines an optimized way to access memory. This component bridges the gap between the logical distribution pattern and the physical memory subsystem, analyzing the distribution to find opportunities for vectorization and coalescing.

LoadStoreTraits performs several analyses:

- **Vector dimension identification**: Finds which Y dimension has stride 1 for optimal vectorization
- **Access pattern calculation**: Determines the number and order of memory operations
- **Space-filling curve construction**: Creates an optimal traversal order for cache efficiency

C++ LoadStoreTraits Analysis:

```
// LoadStoreTraits analyzes the distribution pattern
template <typename Distribution>
struct load_store_traits
{
    static constexpr index_t ndim_y = Distribution::ndim_y;

    // Analyze which Y dimension has stride 1 (best for vectorization)
    static constexpr index_t vector_dim_y = []() {
        // Complex compile-time analysis to find optimal dimension
        return find_vector_dimension<Distribution>();
    }();

    // Calculate vectorization potential
    static constexpr index_t scalar_per_vector = []() {
        // Determine how many elements can be loaded in one instruction
        return calculate_vector_size<Distribution, DataType>();
    }();

    // Space-filling curve for optimal traversal
    using sfc_type = space_filling_curve<ndim_y>;
    static constexpr sfc_type sfc_ys = make_space_filling_curve<Distribution>();

    // Get Y indices for a given access
    CK_TILE_DEVICE constexpr auto get_y_indices(index_t i_access) const
    {
        return sfc_ys.get_index(i_access);
    }
};
```

6.11.5 Space-Filling Curves for Memory Access

TileWindow uses *space-filling curves* to determine the order in which memory is accessed. Space-filling curves provide cache-friendly traversal patterns that help maximize hardware utilization. The "snake" pattern minimizes the distance between consecutive accesses, keeping data in cache longer.

C++ Space-Filling Curve Implementation:

```
// Space-filling curve for optimal memory traversal
template <index_t NDim>
struct space_filling_curve
{
    array<index_t, NDim> tensor_lengths;
    array<index_t, NDim> dim_access_order;
    bool snake_curved;

    // Get coordinates for the i-th access
    CK_TILE_DEVICE constexpr auto get_index(index_t i_access) const
```

(continues on next page)

(continued from previous page)

```
    {
        array<index_t, NDim> indices;

        // Snake pattern logic for cache-friendly access
        if (snake_curved) {
            // Implement snake curve traversal
            // Minimizes distance between consecutive accesses
        }

        return indices;
    }
};
```

6.11.6 TileWindow Data Flow

6.11.7 Creating and Using TileWindow

```
using namespace ck_tile;

// Create a tensor view for input data
auto tensor_view = make_naive_tensor_view(
    data_ptr,
    make_tuple(256, 256),    // Shape
    make_tuple(256, 1)       // Strides
);

// Define window parameters
constexpr auto window_size = make_tuple(32, 32);
auto window_origin = make_tuple(64, 64);

// Create distribution for the window
auto distribution = make_static_tile_distribution<
    tile_distribution_encoding<
        sequence<>,                                 // No replication
        tuple<sequence<4, 2>, sequence<4, 2>>,  // 8x8 threads
        tuple<sequence<1>, sequence<1>>,        // Thread mapping
        tuple<sequence<0>, sequence<0>>,        // Minor indices
        sequence<1, 1>,                         // Y-space: 2x2 per thread
        sequence<1, 1>                          // Y-space minor
    >
>{};

// Create the tile window
auto window = make_tile_window(
    tensor_view,
    window_size,
    window_origin,
    distribution
);

// Load data into distributed tensor
auto distributed_data = make_static_distributed_tensor<float>(distribution);
```

(continues on next page)

(continued from previous page)

```
window.load(distributed_data);
```

6.11.8 The Load Operation Deep Dive

Calls to `window.load()` trigger the following sequence of operations:

1. **Distributed tensor creation**: Automatically creates a *distributed tensor* sized for the distribution
2. **Coordinate calculation**: Uses precomputed coordinates for efficiency
3. **Vectorized access**: Groups elements for vector loads based on *LoadStoreTraits* analysis
4. **Memory coalescing**: Ensures adjacent threads access adjacent memory
5. **Boundary handling**: Manages edge cases automatically

C++ Load Implementation Details:

```
template <typename DistributedTensor>
CK_TILE_DEVICE void load(DistributedTensor& dst_tensor) const
{
    // Get LoadStoreTraits for optimal access pattern
    using Traits = load_store_traits<Distribution>;

    // Iterate through all accesses determined by space-filling curve
    static_for<0, Traits::num_access, 1>{}([&](auto i_access) {
        // Get Y-space indices for this access
        const auto y_indices = Traits::get_y_indices(i_access);

        // Calculate global coordinates
        const auto x_indices = distribution_.calculate_x_from_y(y_indices);
        const auto global_indices = add_arrays(origin_, x_indices);

        // Perform vectorized load if possible
        if constexpr (Traits::scalar_per_vector > 1) {
            // Vector load path
            using VectorType = vector_type_t<DataType, Traits::scalar_per_vector>;
            const auto vector_data = tensor_view_.template get_vectorized_elements
↪<VectorType>(
                global_indices, Traits::vector_dim_y);
            dst_tensor.template set_vectorized_elements(y_indices, vector_data);
        } else {
            // Scalar load path
            const auto scalar_data = tensor_view_.get_element(global_indices);
            dst_tensor.set_element(y_indices, scalar_data);
        }
    });
}
```

6.11.9 Load Operation Architecture

6.11.10 Memory Access Patterns

One of TileWindow's key features is generating optimal memory access patterns. The system analyzes the distribution to ensure:

- **Coalescing**: Adjacent threads access adjacent memory locations
- **Vectorization**: Multiple elements loaded in single instructions
- **Bank conflict avoidance**: Shared memory accesses avoid *conflicts*
- **Cache optimization**: Access patterns maximize cache reuse

C++ Memory Pattern Analysis:

```
// Analyze memory access pattern for a distribution
template <typename Distribution>
struct memory_access_analyzer
{
    static constexpr bool is_coalesced()
    {
        // Check if threads in a warp access consecutive memory
        return Distribution::check_coalescing_pattern();
    }

    static constexpr index_t vector_size()
    {
        // Determine optimal vector size (1, 2, 4, 8)
        return Distribution::calculate_vector_size();
    }

    static constexpr bool has_bank_conflicts()
    {
        // Analyze shared memory access pattern
        return Distribution::detect_bank_conflicts();
    }
};
```

6.11.11 Window Movement and Updates

TileWindow supports efficient window movement for sliding window algorithms. The precomputed coordinate system makes updates more efficient:

```
// Sliding window pattern
for (index_t row = 0; row < tensor_height; row += stride) {
    for (index_t col = 0; col < tensor_width; col += stride) {
        // Update window position - O(1) operation
        window.set_window_origin(make_tuple(row, col));

        // Load from new position
        window.load(distributed_data);

        // Process data
        process_tile(distributed_data);

        // Store results if needed
        output_window.store(distributed_data);
    }
}
```

6.11.12 Store Operations with Vectorization

Store operations use the same compile-time analysis as loads. The *LoadStoreTraits* helps make stores as efficient as loads, with similar vectorization and coalescing benefits:

```
template <typename DistributedTensor>
CK_TILE_DEVICE void store(const DistributedTensor& src_tensor) const
{
    using Traits = load_store_traits<Distribution>;

    // Same optimized pattern as load, but in reverse
    static_for<0, Traits::num_access, 1>{}([&](auto i_access) {
        const auto y_indices = Traits::get_y_indices(i_access);
        const auto x_indices = distribution_.calculate_x_from_y(y_indices);
        const auto global_indices = add_arrays(origin_, x_indices);

        if constexpr (Traits::scalar_per_vector > 1) {
            // Vectorized store
            const auto vector_data = src_tensor.template get_vectorized_elements(
                y_indices, Traits::vector_dim_y);
            tensor_view_.template set_vectorized_elements(
                global_indices, vector_data, Traits::vector_dim_y);
        } else {
            // Scalar store
            const auto scalar_data = src_tensor.get_element(y_indices);
            tensor_view_.set_element(global_indices, scalar_data);
        }
    });
}
```

6.11.13 Complete Load-Compute-Store Pipeline

```
template<typename AType, typename BType, typename CType>
__global__ void gemm_kernel_with_windows(
    const AType* __restrict__ a_ptr,
    const BType* __restrict__ b_ptr,
    CType* __restrict__ c_ptr,
    index_t M, index_t N, index_t K)
{
    // Create tensor views
    auto a_tensor = make_naive_tensor_view(
        a_ptr, make_tuple(M, K), make_tuple(K, 1));
    auto b_tensor = make_naive_tensor_view(
        b_ptr, make_tuple(K, N), make_tuple(N, 1));
    auto c_tensor = make_naive_tensor_view(
        c_ptr, make_tuple(M, N), make_tuple(N, 1));

    // Define tile sizes
    constexpr index_t tile_m = 32;
    constexpr index_t tile_n = 32;
    constexpr index_t tile_k = 8;

    // Create distributions
```

(continues on next page)

(continued from previous page)

```
    auto a_dist = make_static_tile_distribution<...>();
    auto b_dist = make_static_tile_distribution<...>();
    auto c_dist = make_static_tile_distribution<...>();

    // Calculate tile position
    const index_t block_m = blockIdx.y * tile_m;
    const index_t block_n = blockIdx.x * tile_n;

    // Create tile windows
    auto a_window = make_tile_window(
        a_tensor,
        make_tuple(tile_m, tile_k),
        make_tuple(block_m, 0),
        a_dist);

    auto b_window = make_tile_window(
        b_tensor,
        make_tuple(tile_k, tile_n),
        make_tuple(0, block_n),
        b_dist);

    auto c_window = make_tile_window(
        c_tensor,
        make_tuple(tile_m, tile_n),
        make_tuple(block_m, block_n),
        c_dist);

    // Create distributed tensors for register storage
    auto a_reg = make_static_distributed_tensor<AType>(a_dist);
    auto b_reg = make_static_distributed_tensor<BType>(b_dist);
    auto c_reg = make_static_distributed_tensor<CType>(c_dist);

    // Initialize accumulator
    c_reg.clear();

    // Main GEMM loop
    for(index_t k = 0; k < K; k += tile_k) {
        // Update window positions
        a_window.set_window_origin(make_tuple(block_m, k));
        b_window.set_window_origin(make_tuple(k, block_n));

        // Load tiles - LoadStoreTraits ensures optimal pattern
        a_window.load(a_reg);
        b_window.load(b_reg);

        // Compute
        gemm(a_reg, b_reg, c_reg);
    }

    // Store result - using same optimized pattern
    c_window.store(c_reg);
}
```

6.11.14 Performance Characteristics

6.11.15 Best Practices

Window Size Selection

Choose window sizes that balance register usage with data reuse:

```
// Optimal window size calculation
template <typename DataType, index_t RegistersPerThread>
constexpr auto calculate_optimal_window_size()
{
    // Consider register constraints
    constexpr index_t elements_per_thread = RegistersPerThread / sizeof(DataType);

    // Common tile sizes that work well
    constexpr array<index_t, 5> common_sizes = {8, 16, 32, 64, 128};

    // Find largest size that fits in registers
    for (auto size : common_sizes) {
        if (size * size <= elements_per_thread) {
            return size;
        }
    }
    return 8;  // Minimum reasonable size
}
```

Access Pattern Optimization

Design distributions for optimal memory access:

```
// Create distribution optimized for coalescing
// This example shows a 32x32 tile distributed across threads
using OptimalDistribution = tile_distribution_encoding<
    sequence<>,                                    // RsLengths: No replication
    tuple<sequence<4, 8>, sequence<4, 8>>,         // HsLengthss: Hierarchical␣
↪decomposition
    tuple<sequence<1, 2>, sequence<1, 2>>,         // Ps2RHssMajor: P to RH major mapping
    tuple<sequence<0, 0>, sequence<1, 0>>,         // Ps2RHssMinor: P to RH minor mapping
    sequence<1, 1, 2, 2>,                          // Ys2RHsMajor: Y to RH major mapping
    sequence<0, 1, 0, 1>                           // Ys2RHsMinor: Y to RH minor mapping
>;
```

6.11.16 Summary

TileWindow provides:

- **Automatic optimization**: Generates optimal memory access patterns through *LoadStoreTraits*
- **Distribution awareness**: Works seamlessly with *TileDistribution*
- **Space-filling curves**: Optimizes traversal order for cache efficiency (see *Space-Filling Curves - Optimal Memory Traversal*)
- **Vectorization**: Automatic multi-element operations
- **Precomputation**: Zero-overhead *coordinate transformations*

- **Flexible windowing**: Supports various access patterns and window configurations
- **Safety**: Automatic boundary handling

Key benefits:

1. **Performance**: Improves memory bandwidth through coalescing and vectorization
2. **Productivity**: Reduces reliance manual memory management code
3. **Correctness**: Automatic boundary checking and handling
4. **Composability**: Integrates with other CK abstractions
5. **Intelligence**: LoadStoreTraits analyzes and optimizes access

The TileWindow abstraction helps build high-performance GPU kernels, providing an interface for complex memory access patterns while helping maintain performance gains. The compile-time analysis performed by LoadStoreTraits ensures that memory operations are as efficient as possible, while the space-filling curve traversal maximizes cache utilization.

6.11.17 Next Steps

- *LoadStoreTraits - Memory Access Optimization Engine* - Deep dive into access pattern optimization
- *Space-Filling Curves - Optimal Memory Traversal* - Advanced traversal patterns
- *Static Distributed Tensor* - Register-based tensor storage
- *Load Data Share Index Swapping* - Advanced shared memory optimization
- *Sweep Tile* - Efficient tile-based algorithms

6.12 LoadStoreTraits - Memory Access Optimization Engine

6.12.1 Overview

LoadStoreTraits is a critical optimization component that analyzes *tile distributions* to determine the most efficient memory access patterns. It serves as the engine behind *TileWindow's* high-performance data movement, automatically identifying the best dimension for vectorization and creating optimized access sequences using *space-filling curves*.

At compile time, LoadStoreTraits performs compile-time analysis of the distribution pattern to extract key information about memory access opportunities. This analysis determines how many elements can be loaded or stored in a single instruction, which dimension provides the best vectorization opportunity, and what traversal order maximizes cache utilization. The result is a set of compile-time constants and methods that guide the runtime execution of load and store operations.

6.12.2 Key Concepts

Vectorization Selection

LoadStoreTraits analyzes tensor dimensions to find the optimal one for vectorized loads and stores, prioritizing:

- **Contiguous memory access** (stride = 1)
- **Maximum vector length** based on data type and *hardware capabilities*
- **Alignment requirements** for efficient memory transactions

Space-Filling Curve Integration

The system automatically creates a *space-filling curve* that maximizes cache utilization while respecting vectorization constraints. This ensures that consecutive memory accesses are spatially close, reducing cache misses and improving memory bandwidth utilization.

Access Pattern Optimization

LoadStoreTraits manages the trade-off between vector size and number of memory accesses, finding a solution that minimizes total memory transactions while maximizing bandwidth utilization.

6.12.3 C++ Implementation

The LoadStoreTraits class analyzes distribution patterns at compile time:

```
template <typename Distribution, typename DataType>
struct load_store_traits
{
    // Compile-time analysis results
    static constexpr index_t ndim_y = Distribution::ndim_y;
    static constexpr index_t ndim_x = Distribution::ndim_x;

    // Find which Y dimension has stride 1 (best for vectorization)
    static constexpr index_t vector_dim_y = []() {
        // Complex compile-time analysis to find optimal dimension
        const auto strides = Distribution::calculate_y_strides();
        for (index_t i = 0; i < ndim_y; ++i) {
            if (strides[i] == 1) return i;
        }
        return ndim_y - 1;  // Default to last dimension
    }();

    // Calculate how many scalars fit in a vector
    static constexpr index_t scalar_per_vector = []() {
        // Determine based on data type and hardware capabilities
        if constexpr (sizeof(DataType) == 4) {  // float32
            return min(Distribution::get_y_length(vector_dim_y), 4);
        } else if constexpr (sizeof(DataType) == 2) {  // float16
            return min(Distribution::get_y_length(vector_dim_y), 8);
        }
        return 1;
    }();

    // Total scalars accessed per memory operation
    static constexpr index_t scalars_per_access = scalar_per_vector;

    // Space-filling curve for optimal traversal
    using sfc_type = space_filling_curve<ndim_y>;
    static constexpr sfc_type sfc_ys = make_space_filling_curve<Distribution>();

    // Total number of accesses needed
    static constexpr index_t num_access =
        Distribution::get_num_of_element_y() / scalars_per_access;
```

(continues on next page)

(continued from previous page)

```
    // Get Y indices for a given access
    CK_TILE_DEVICE constexpr auto get_y_indices(index_t i_access) const
    {
        return sfc_ys.get_index(i_access);
    }

    // Get detailed vectorized access information
    CK_TILE_DEVICE constexpr auto get_vectorized_access_info(index_t i_access) const
    {
        const auto base_indices = get_y_indices(i_access);
        // Return structure with base indices, vector dimension, and size
        return vectorized_access_info{
            base_indices,
            vector_dim_y,
            scalar_per_vector
        };
    }
};
```

6.12.4 Vectorization Selection Algorithm

LoadStoreTraits employs an advanced algorithm to select the best dimension for vectorization:

Example: Comparing Different Memory Layouts

```
// Row-major layout [4×16]
using RowMajorDist = tile_distribution_encoding<
    sequence<>,                                 // No replication
    tuple<sequence<2, 2>, sequence<4, 4>>,  // 4x16 total
    tuple<sequence<1>, sequence<1>>,        // Thread mapping
    tuple<sequence<0>, sequence<0>>,        // Minor indices
    sequence<2, 4>,                         // Y-space per thread
    sequence<1, 1>                          // Y-space minor
>;

// Column-major layout [16×4]
using ColMajorDist = tile_distribution_encoding<
    sequence<>,                                 // No replication
    tuple<sequence<4, 4>, sequence<2, 2>>,  // 16x4 total
    tuple<sequence<1>, sequence<1>>,        // Thread mapping
    tuple<sequence<0>, sequence<0>>,        // Minor indices
    sequence<4, 2>,                         // Y-space per thread
    sequence<1, 1>                          // Y-space minor
>;

// LoadStoreTraits analysis
using RowTraits = load_store_traits<RowMajorDist, float>;
using ColTraits = load_store_traits<ColMajorDist, float>;

// Row-major: vectorizes dimension 1 (4 elements)
static_assert(RowTraits::vector_dim_y == 1);
static_assert(RowTraits::scalar_per_vector == 4);
```

(continues on next page)

(continued from previous page)

```
// Column-major: vectorizes dimension 1 (2 elements)
static_assert(ColTraits::vector_dim_y == 1);
static_assert(ColTraits::scalar_per_vector == 2);
```

6.12.5 Memory Access Patterns

LoadStoreTraits creates efficient access patterns using space-filling curves:

C++ Access Pattern Example:

```
// Create a 6x8 tile distribution
using TileDist = tile_distribution_encoding<
    sequence<>,
    tuple<sequence<2, 3>, sequence<2, 4>>,  // 6x8 tile
    tuple<sequence<1>, sequence<1>>,
    tuple<sequence<0>, sequence<0>>,
    sequence<3, 4>,                         // 3x4 per thread
    sequence<1, 1>
>;

using Traits = load_store_traits<TileDist, float>;

// Access pattern visualization
template <typename Traits>
CK_TILE_DEVICE void visualize_access_pattern()
{
    printf("Tile: %dx%d\n", TileDist::get_tile_m(), TileDist::get_tile_n());
    printf("Vector dimension: %d\n", Traits::vector_dim_y);
    printf("Scalars per access: %d\n", Traits::scalars_per_access);
    printf("\nAccess sequence:\n");

    // Show first few accesses
    static_for<0, min(6, Traits::num_access), 1>{}([](auto i) {
        const auto indices = Traits::get_y_indices(i);
        const auto info = Traits::get_vectorized_access_info(i);

        printf("Access %d: Base=[%d,%d], Vector size=%d\n",
               i, indices[0], indices[1], info.vector_size);
    });
}
```

6.12.6 Performance Analysis

Memory Access Efficiency

LoadStoreTraits optimizes for several performance metrics:

```
template <typename Distribution>
struct memory_access_analyzer
{
    using Traits = load_store_traits<Distribution, float>;
```

(continues on next page)

(continued from previous page)

```
    // Calculate memory bandwidth utilization
    static constexpr float bandwidth_utilization()
    {
        constexpr index_t bytes_per_access = Traits::scalar_per_vector * sizeof(float);
        constexpr index_t cache_line_size = 64;  // bytes
        return static_cast<float>(bytes_per_access) / cache_line_size * 100.0f;
    }

    // Calculate total memory transactions
    static constexpr index_t total_transactions()
    {
        return Traits::num_access;
    }

    // Check coalescing efficiency
    static constexpr bool is_perfectly_coalesced()
    {
        // Perfect coalescing when adjacent threads access adjacent memory
        return Traits::vector_dim_y == Distribution::ndim_y - 1 &&
               Traits::scalar_per_vector >= 4;
    }
};
```

Comparing Different Configurations

```
// Configuration 1: Simple 8x8 tile
using Simple8x8 = tile_distribution_encoding<
    sequence<>,
    tuple<sequence<2, 4>, sequence<2, 4>>,
    tuple<sequence<1>, sequence<1>>,
    tuple<sequence<0>, sequence<0>>,
    sequence<4, 4>,
    sequence<1, 1>
>;

// Configuration 2: Optimized for vectorization
using OptimizedVector = tile_distribution_encoding<
    sequence<>,
    tuple<sequence<4, 2>, sequence<2, 8>>,
    tuple<sequence<1>, sequence<1>>,
    tuple<sequence<0>, sequence<0>>,
    sequence<2, 8>,  // 2x8 per thread for better vectorization
    sequence<1, 1>
>;

// Analysis
using SimpleAnalyzer = memory_access_analyzer<Simple8x8>;
using OptimizedAnalyzer = memory_access_analyzer<OptimizedVector>;

static_assert(SimpleAnalyzer::bandwidth_utilization() == 25.0f);  // 4*4/64
static_assert(OptimizedAnalyzer::bandwidth_utilization() == 50.0f);  // 8*4/64
```

(continues on next page)

(continued from previous page)

```
// Better bandwidth utilization leads to improved performance
```

6.12.7 Integration with Space-Filling Curves

LoadStoreTraits automatically configures space-filling curves for optimal access:

```
template <typename Distribution>
struct space_filling_curve_optimizer
{
    using Traits = load_store_traits<Distribution, float>;

    static constexpr auto create_optimized_curve()
    {
        // Move vector dimension to end of access order
        array<index_t, Distribution::ndim_y> dim_order;

        // Fill non-vector dimensions first
        index_t pos = 0;
        for (index_t i = 0; i < Distribution::ndim_y; ++i) {
            if (i != Traits::vector_dim_y) {
                dim_order[pos++] = i;
            }
        }

        // Vector dimension last for contiguous access
        dim_order[pos] = Traits::vector_dim_y;

        // Create space-filling curve with optimized order
        return space_filling_curve<Distribution::ndim_y>{
            Distribution::get_y_lengths(),
            dim_order,
            Traits::scalar_per_vector,
            true  // Enable snake pattern
        };
    }
};
```

6.12.8 Advanced Optimizations

Multi-Level Vectorization

For complex *distributions*, LoadStoreTraits can identify multiple levels of vectorization:

```
template <typename Distribution>
struct multi_level_vectorization
{
    // Primary vector dimension (innermost, stride 1)
    static constexpr index_t primary_vector_dim =
        load_store_traits<Distribution, float>::vector_dim_y;

    // Secondary vector dimension (next best option)
```

(continues on next page)

(continued from previous page)

```
    static constexpr index_t secondary_vector_dim = []() {
        const auto strides = Distribution::calculate_y_strides();
        for (index_t i = 0; i < Distribution::ndim_y; ++i) {
            if (i != primary_vector_dim &&
                strides[i] <= 4) {  // Small stride
                return i;
            }
        }
        return -1;
    }();

    // Can use 2D vectorization?
    static constexpr bool supports_2d_vector = secondary_vector_dim >= 0;
};
```

Adaptive Vector Size Selection

LoadStoreTraits adapts vector size based on multiple factors:

```
template <typename Distribution, typename DataType>
struct adaptive_vector_size
{
    static constexpr index_t calculate_optimal_vector_size()
    {
        constexpr index_t dim_length =
            Distribution::get_y_length(load_store_traits<Distribution, DataType>::vector_
↪dim_y);

        // Hardware-specific vector sizes
        constexpr array<index_t, 4> valid_sizes = {8, 4, 2, 1};

        // Find largest valid size that divides dimension length
        for (auto size : valid_sizes) {
            if (dim_length % size == 0 &&
                size * sizeof(DataType) <= 32) {  // Max vector register size
                return size;
            }
        }
        return 1;
    }
};
```

6.12.9 Best Practices

1. **Design Distributions for Vectorization**

```
// Good: Inner dimension is power of 2
using GoodDist = tile_distribution_encoding<
    sequence<>,
    tuple<sequence<4, 2>, sequence<2, 8>>,  // Inner dim = 16
    tuple<sequence<1>, sequence<1>>,
    tuple<sequence<0>, sequence<0>>,
```

(continues on next page)

(continued from previous page)

```
    sequence<2, 8>,  // 8 elements for vectorization
    sequence<1, 1>
>;
```

2. **Consider Data Type Size**

```
// Adjust distribution based on data type
template <typename DataType>
using AdaptiveDist = std::conditional_t<
    sizeof(DataType) == 2,  // FP16
    tile_distribution_encoding<...>,  // 8-wide vectors
    tile_distribution_encoding<...>   // 4-wide vectors for FP32
>;
```

3. **Align for Cache Lines**

```
// Ensure tile dimensions align with cache lines
static_assert(TileDist::get_tile_n() * sizeof(float) % 64 == 0,
              "Tile width should align to cache lines");
```

For more optimization techniques, see *Understanding AMD GPU LDS and Bank Conflicts* and *Load Data Share Index Swapping*.

6.12.10 Summary

LoadStoreTraits provides:

- **Automatic vectorization analysis**: Identifies optimal dimensions and vector sizes
- **Space-filling curve optimization**: Creates cache-friendly access patterns. See *Space-Filling Curves - Optimal Memory Traversal* for more information.
- **Compile-time optimization**: All analysis done at compile time for zero overhead
- **Hardware adaptation**: Adjusts to different data types and *architectures*
- **Performance transparency**: Clear metrics for memory efficiency

The compile-time analysis performed by LoadStoreTraits ensures that every memory operation in CK Tile achieves near-optimal performance, making it a critical component in the high-performance computing stack.

6.12.11 Next Steps

- *Space-Filling Curves - Optimal Memory Traversal* - Deep dive into traversal patterns
- *Tile Window - Data Access Gateway* - How LoadStoreTraits enables efficient data access
- *Static Distributed Tensor* - The target of optimized loads/stores
- *Coordinate Systems - The Mathematical Foundation* - Understanding the coordinate transformations
- *A Block GEMM on MI300* - Real-world application of LoadStoreTraits

6.13 Space-Filling Curves - Optimal Memory Traversal

6.13.1 Overview

The SpaceFillingCurve (SFC) class provides a systematic way to traverse multi-dimensional tensors, supporting both scalar and vectorized access patterns. This is particularly important for optimizing memory access patterns in *GPU kernels*, where the order of memory accesses can dramatically impact performance through cache utilization, memory coalescing, and prefetching effectiveness.

A space-filling curve is a continuous curve that visits every point in a multi-dimensional space exactly once. In the context of CK Tile, it defines a mapping from a 1D access index to multi-dimensional *tensor coordinates*, enabling efficient traversal patterns that maximize hardware utilization.

6.13.2 Key Concepts

Tensor Traversal

The space-filling curve defines a mapping from a 1D access index to multi-dimensional tensor coordinates. This abstraction allows complex multi-dimensional access patterns to be expressed as simple linear iterations, while maintaining optimal memory access characteristics.

Vectorized Access

GPUs support vector load and store instructions that can access multiple consecutive elements in a single operation. SpaceFillingCurve supports this by allowing specification of how many elements to access per dimension (`scalars_per_access`), enabling efficient utilization of these hardware features.

Dimension Ordering

The order in which dimensions are traversed impacts memory access patterns. Row-major vs column-major ordering, for example, can mean the difference between the preferred sequential memory access and strided access which can potentially cause cache thrashing.

Snake Patterns

Snake, or serpentine, patterns reverse the traversal direction on alternate rows and planes, keeping consecutive accesses spatially close. This pattern is particularly effective for maintaining cache locality when moving between rows or higher-dimensional boundaries.

Usage

SFC mainly uses Tile Transpose, Tile shuffling iteration, and CShuffle to access the tile data in the discrete way the application requires and have the best cache memory coherent hit.

6.13.3 C++ Implementation

The C++ template class provides compile-time optimization of traversal patterns:

```
template<index_t NDimSFC,
         typename SFCLengths,
         typename DimAccessOrder,
         typename ScalarsPerAccess,
         bool IsSnakeCurved = false>
struct space_filling_curve
{
    static constexpr index_t ndim = NDimSFC;
```

(continues on next page)

(continued from previous page)

```
    static constexpr auto tensor_lengths = SFCLengths{};
    static constexpr auto dim_access_order = DimAccessOrder{};
    static constexpr auto scalars_per_access = ScalarsPerAccess{};
    static constexpr bool snake_curved = IsSnakeCurved;

    // Calculate access dimensions (with ceiling division)
    static constexpr auto access_lengths = []() {
        array<index_t, ndim> lengths;
        for (index_t i = 0; i < ndim; ++i) {
            lengths[i] = (tensor_lengths[i] + scalars_per_access[i] - 1)
                       / scalars_per_access[i];
        }
        return lengths;
    }();

    // Total number of accesses needed
    static constexpr index_t get_num_of_access()
    {
        index_t total = 1;
        for (index_t i = 0; i < ndim; ++i) {
            total *= access_lengths[i];
        }
        return total;
    }

    // Convert 1D access index to N-D coordinates
    CK_TILE_DEVICE constexpr auto get_index(index_t i_access) const
    {
        array<index_t, ndim> indices;

        // Calculate indices in access space
        index_t remaining = i_access;
        for (index_t i = ndim - 1; i >= 0; --i) {
            const index_t dim = dim_access_order[i];
            indices[dim] = remaining % access_lengths[dim];
            remaining /= access_lengths[dim];
        }

        // Apply snake pattern if enabled
        if constexpr (snake_curved) {
            apply_snake_pattern(indices);
        }

        // Scale by scalars_per_access
        for (index_t i = 0; i < ndim; ++i) {
            indices[i] *= scalars_per_access[i];
        }

        return indices;
    }

    // Calculate step between two accesses
```

(continues on next page)

(continued from previous page)

```
    CK_TILE_DEVICE constexpr auto get_step_between(
        index_t start, index_t end) const
    {
        const auto start_idx = get_index(start);
        const auto end_idx = get_index(end);

        array<index_t, ndim> step;
        for (index_t i = 0; i < ndim; ++i) {
            step[i] = end_idx[i] - start_idx[i];
        }
        return step;
    }
};
```

6.13.4 Basic Usage Examples

Scalar Access Patterns

```
// Row-major traversal of 4x6 matrix
using RowMajorCurve = space_filling_curve<
    2,                    // 2D
    sequence<4, 6>,       // Shape: 4x6
    sequence<0, 1>,       // Dimension order: row then column
    sequence<1, 1>,       // Scalar access
    false                 // No snake pattern
>;

// Total accesses needed
constexpr index_t num_access = RowMajorCurve::get_num_of_access();  // 24

// Access pattern (first 10)
static_for<0, 10, 1>{}([](auto i) {
    constexpr auto indices = RowMajorCurve{}.get_index(i);
    printf("Access %d: [%d, %d]\n", i, indices[0], indices[1]);
});
// Output: [0,0], [0,1], [0,2], [0,3], [0,4], [0,5], [1,0], [1,1], ...
```

Vectorized Access Patterns

```
// Vector-4 access on dimension 1
using VectorizedCurve = space_filling_curve<
    2,                    // 2D
    sequence<4, 8>,       // Shape: 4x8
    sequence<0, 1>,       // Row-major
    sequence<1, 4>,       // Vector-4 on dimension 1
    false
>;

// Access pattern visualization
static_for<0, VectorizedCurve::get_num_of_access(), 1>{}([](auto i) {
    constexpr auto indices = VectorizedCurve{}.get_index(i);
```

(continues on next page)

(continued from previous page)

```
    printf("Access %d: row %d, cols [%d:%d]\n",
           i, indices[0], indices[1], indices[1] + 3);
});
// Output: row 0, cols [0:3], row 0, cols [4:7], row 1, cols [0:3], ...
```

Column-Major vs Row-Major

```
// Compare access patterns
using RowMajor = space_filling_curve<2, sequence<4, 6>,
                                     sequence<0, 1>, sequence<1, 1>, false>;
using ColMajor = space_filling_curve<2, sequence<4, 6>,
                                     sequence<1, 0>, sequence<1, 1>, false>;

// Row-major: [0,0], [0,1], [0,2], ... (traverse rows)
// Col-major: [0,0], [1,0], [2,0], ... (traverse columns)
```

6.13.5 Advanced Patterns

Snake Pattern for Cache Optimization

The snake pattern reverses traversal direction on alternate rows, minimizing the distance between consecutive accesses:

```
using SnakeCurve = space_filling_curve<
    2,
    sequence<4, 8>,
    sequence<0, 1>,
    sequence<1, 1>,
    true  // Enable snake pattern
>;

// Access pattern with snake:
// Row 0: [0,0], [0,1], [0,2], ..., [0,7]
// Row 1: [1,7], [1,6], [1,5], ..., [1,0]  (reversed!)
// Row 2: [2,0], [2,1], [2,2], ..., [2,7]
// Row 3: [3,7], [3,6], [3,5], ..., [3,0]  (reversed!)
```

GEMM Tile Access Pattern

For *matrix multiplication*, optimal access patterns are crucial:

```
// GEMM tile: 16x32 with vector-8 loads
// Column-major for coalesced access in GEMM
using GemmTileCurve = space_filling_curve<
    2,
    sequence<16, 32>,    // Tile size
    sequence<1, 0>,      // Column-major
    sequence<1, 8>,      // Vector-8 loads
    false
>;

// This creates a pattern where:
// - Each access loads 8 consecutive elements
```

(continues on next page)

(continued from previous page)

```
// - Accesses proceed down columns (coalesced for column-major storage)
// - Total accesses: 16 * (32/8) = 64
```

3D Tensor Patterns

```
// 3D tensor with mixed vectorization
using Tensor3D = space_filling_curve<
    3,
    sequence<4, 8, 16>,  // 4x8x16 tensor
    sequence<0, 1, 2>,   // Access order
    sequence<1, 2, 4>,   // Different vector sizes per dimension
    false
>;

// Access pattern:
// - Dimension 0: scalar access
// - Dimension 1: vector-2 access
// - Dimension 2: vector-4 access
// Total accesses: 4 * (8/2) * (16/4) = 64
```

6.13.6 Performance Analysis

Step Analysis for Memory Patterns

Understanding step patterns between accesses is crucial for performance:

```
template <typename SFC>
struct access_pattern_analyzer
{
    static constexpr void analyze_locality()
    {
        index_t sequential_steps = 0;
        index_t cache_line_jumps = 0;
        index_t large_jumps = 0;

        constexpr SFC sfc{};

        for (index_t i = 0; i < SFC::get_num_of_access() - 1; ++i) {
            const auto step = sfc.get_step_between(i, i + 1);

            // Calculate Manhattan distance
            index_t distance = 0;
            for (index_t d = 0; d < SFC::ndim; ++d) {
                distance += abs(step[d]);
            }

            if (distance <= 1) {
                sequential_steps++;
            } else if (distance <= 16) {  // Within cache line
                cache_line_jumps++;
            } else {
                large_jumps++;
```

(continues on next page)

(continued from previous page)

```
            }
        }

        // Report statistics...
    }
};
```

Optimizing for Hardware

```
// Optimize for GPU memory coalescing
template <typename DataType, index_t WarpSize = 32>
struct coalesced_access_pattern
{
    // For coalescing, adjacent threads should access adjacent memory
    static constexpr index_t vector_size = sizeof(float4) / sizeof(DataType);

    using OptimalPattern = space_filling_curve<
        2,
        sequence<BlockM, BlockN>,
        sequence<1, 0>,              // Column-major for coalescing
        sequence<1, vector_size>,    // Vectorized on fast-changing dimension
        false
    >;
};
```

6.13.7 Handling Edge Cases

Non-Divisible Dimensions

When tensor dimensions aren't evenly divisible by vector size:

```
// 5x7 tensor with 2x3 access pattern
using EdgeCaseCurve = space_filling_curve<
    2,
    sequence<5, 7>,
    sequence<0, 1>,
    sequence<2, 3>,
    false
>;

// Access lengths use ceiling division: ceil(5/2) x ceil(7/3) = 3x3
static_assert(EdgeCaseCurve::access_lengths[0] == 3);
static_assert(EdgeCaseCurve::access_lengths[1] == 3);

// Boundary handling needed for accesses that exceed tensor bounds
template <typename SFC>
CK_TILE_DEVICE void safe_access(index_t i_access)
{
    const auto indices = SFC{}.get_index(i_access);

    // Check bounds for each dimension
    bool in_bounds = true;
```

(continues on next page)

(continued from previous page)

```
    for (index_t d = 0; d < SFC::ndim; ++d) {
        if (indices[d] + SFC::scalars_per_access[d] > SFC::tensor_lengths[d]) {
            in_bounds = false;
            break;
        }
    }

    if (in_bounds) {
        // Full vector access
    } else {
        // Partial access with masking
    }
}
```

6.13.8 Integration with CK Tile

LoadStoreTraits Integration

LoadStoreTraits uses space-filling curves to optimize memory access:

```
template <typename Distribution>
struct load_store_traits
{
    // Create optimized space-filling curve
    using sfc_type = space_filling_curve<
        Distribution::ndim_y,
        typename Distribution::y_lengths,
        optimized_dim_order<Distribution>,     // Computed order
        optimized_scalars_per_access<Distribution>,
        true  // Enable snake for cache optimization
    >;

    static constexpr sfc_type sfc_ys{};
};
```

TileWindow Usage

TileWindow leverages space-filling curves for systematic tile traversal:

```
template <typename TileWindow>
CK_TILE_DEVICE void process_tile(const TileWindow& window)
{
    using Traits = typename TileWindow::traits_type;
    constexpr auto sfc = Traits::sfc_ys;

    // Traverse tile using space-filling curve
    static_for<0, sfc.get_num_of_access(), 1>{}([&](auto i) {
        const auto indices = sfc.get_index(i);
        // Process element at indices...
    });
}
```

6.13.9 Best Practices

1. **Choose Appropriate Dimension Order**

```
// For row-major storage, use row-major traversal
using RowMajorSFC = space_filling_curve<2, Shape, sequence<0, 1>, ...>;

// For column-major storage, use column-major traversal
using ColMajorSFC = space_filling_curve<2, Shape, sequence<1, 0>, ...>;
```

2. **Optimize Vector Size**

```
// Match vector size to cache line for optimal bandwidth
constexpr index_t optimal_vector = min(
    tensor_length_fast_dim,
    cache_line_size / sizeof(DataType)
);
```

3. **Enable Snake Pattern for Large Tensors**

```
// Snake pattern helps when jumping between rows/planes
using CacheFriendlySFC = space_filling_curve<
    NDim, Lengths, Order, Scalars,
    true  // Enable snake
>;
```

4. **Consider Memory Layout**

```
// Align access patterns with physical memory layout
static_assert(
    SFC::scalars_per_access[fastest_dim] * sizeof(DataType)
    % cache_line_size == 0,
    "Vector access should align with cache lines"
);
```

6.13.10 Summary

Space-filling curves provide:

- **Systematic traversal**: Convert N-D access to 1D iteration
- **Vectorization support**: Efficient use of vector load and store instructions
- **Cache optimization**: Snake patterns and dimension ordering for locality
- **Flexibility**: Adaptable to different *tensor shapes* and access patterns
- **Performance**: Compile-time optimization with zero runtime overhead

The advanced traversal patterns enabled by space-filling curves are fundamental to achieving high performance in GPU kernels, ensuring that memory access patterns align with *hardware capabilities*.

6.13.11 Next Steps

- *LoadStoreTraits - Memory Access Optimization Engine* - How curves optimize memory access
- *Sweep Tile* - Traversing distributed tensors
- *Static Distributed Tensor* - The data structures being traversed

- *Tile Window - Data Access Gateway* - Integration with data loading
- *A Block GEMM on MI300* - Real-world application example

6.14 Static Distributed Tensor

6.14.1 Overview

Static distributed tensors represent the thread-local data containers in CK Tile's programming model. Unlike traditional GPU programming where developers manually manage thread-local arrays and coordinate access patterns, static distributed tensors provide a high-level abstraction that automatically handles data distribution across threads while maintaining the performance characteristics of register-based storage.

Each thread in a workgroup owns a portion of the overall tensor data, stored in its registers or local memory. The distribution pattern follows the *tile distribution* rules, ensuring that collective operations across all threads reconstruct the complete logical tensor while individual threads operate only on their local portions.

This design enables three critical optimizations:

- It maximizes register utilization by keeping frequently accessed data in the fastest memory hierarchy.
- It eliminates redundant memory accesses since each thread maintains its own working set.
- It provides a clean abstraction for complex algorithms like matrix multiplication where each thread accumulates partial results that eventually combine into the final output.

6.14.2 Thread-Local Storage Model

The static distributed tensor implements an advanced storage model that maps multi-dimensional tensor data to thread-local arrays:

```
template<typename DataType,
         typename TileDistribution,
         typename... Lengths>
struct StaticDistributedTensor {
    // Each thread stores its portion of the tensor
    static constexpr index_t kNumElements =
        TileDistribution::GetNumElementsPerThread();

    // Thread-local storage - typically maps to registers
    DataType data_[kNumElements];

    // Access using Y-space coordinates (see :ref:`ck_tile_coordinate_systems`)
    __device__ DataType& operator()(const YIndex& idx) {
        // Convert Y coordinate to local buffer offset
        index_t offset = TileDistribution::YToLocalOffset(idx);
        return data_[offset];
    }
};
```

The storage layout follows these principles:

1. **Contiguous Storage**: Each thread's data is stored in a contiguous array, optimizing register allocation and enabling vectorized operations.
2. **Deterministic Mapping**: The Y-coordinate to buffer offset mapping is computed at compile time, eliminating runtime overhead.

3. **Alignment Guarantees**: The storage layout respects hardware alignment requirements for efficient memory operations.

6.14.3 Memory Layout and Access Patterns

Understanding how static distributed tensors organize memory is important for performance optimization. Consider a 2D tensor distributed across a 2D thread block:

```
// Define a 64x64 tensor distributed across 16x16 threads
using TileDist = TileDistribution<
    Sequence<64, 64>,      // Tensor dimensions
    Sequence<16, 16>       // Thread block dimensions
>;

// Each thread owns a 4x4 subtensor
using MyTensor = StaticDistributedTensor<float, TileDist>;

__device__ void example_kernel() {
    MyTensor accumulator;

    // Initialize thread-local data
    for(index_t i = 0; i < 4; ++i) {
        for(index_t j = 0; j < 4; ++j) {
            // Y-space coordinates for this thread's elements
            YIndex y_idx = make_tuple(
                threadIdx.y * 4 + i,
                threadIdx.x * 4 + j
            );
            accumulator(y_idx) = 0.0f;
        }
    }
}
```

The memory layout follows a hierarchical pattern:

6.14.4 Element Access and Indexing

Static distributed tensors provide multiple indexing modes to support different access patterns:

```
template<typename DataType, typename TileDistribution>
class StaticDistributedTensor {
public:
    // Y-space indexing (most common) - see :ref:`ck_tile_coordinate_systems`
    __device__ DataType& operator()(const YIndex& y_idx) {
        return data_[YToOffset(y_idx)];
    }

    // Direct buffer indexing (for vectorized operations)
    __device__ DataType& operator[](index_t offset) {
        return data_[offset];
    }

    // Structured access for multi-dimensional patterns
    template<typename... Coords>
```

(continues on next page)

(continued from previous page)

```
    __device__ DataType& at(Coords... coords) {
        YIndex y_idx = make_tuple(coords...);
        return (*this)(y_idx);
    }

    // Vectorized access for performance
    template<index_t VectorSize>
    __device__ auto get_vector(index_t offset) {
        using VectorType = vector_type_t<DataType, VectorSize>;
        return *reinterpret_cast<VectorType*>(&data_[offset]);
    }
};
```

The indexing system supports several optimization strategies:

1. **Compile-Time Resolution**: When indices are known at compile time, the compiler can optimize away all indexing calculations.
2. **Vectorized Access**: Accessing multiple elements as vectors enables efficient register-to-register transfers.
3. **Boundary Checking**: Debug builds include automatic boundary checking to catch indexing errors early.

6.14.5 Thread Coordination and Synchronization

Static distributed tensors excel at patterns where threads cooperate to process larger data structures:

```
// Matrix multiplication accumulator pattern
// See :ref:`ck_tile_gemm_optimization` for complete example
template<typename AType, typename BType, typename CType>
__device__ void gemm_accumulate(
    const TileWindow<AType>& a_window,
    const TileWindow<BType>& b_window,
    StaticDistributedTensor<CType>& c_accumulator)
{
    // Each thread accumulates its portion
    constexpr index_t kInnerTiles = 8;

    for(index_t k = 0; k < kInnerTiles; ++k) {
        // Load tiles from global memory
        auto a_tile = a_window.load(k);
        auto b_tile = b_window.load(k);

        // Synchronize to ensure all loads complete
        __syncthreads();

        // Local accumulation (no synchronization needed)
        for(index_t i = 0; i < 4; ++i) {
            for(index_t j = 0; j < 4; ++j) {
                CType sum = 0;
                for(index_t kk = 0; kk < 4; ++kk) {
                    sum += a_tile(i, kk) * b_tile(kk, j);
                }
                c_accumulator.at(i, j) += sum;
            }
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

Key coordination patterns include:

1. **Accumulation**: Each thread maintains partial results that combine to form the final answer.
2. **Scatter/Gather**: Threads can efficiently reorganize data through coordinated read/write patterns.
3. **Reduction**: Tree-based reduction algorithms naturally map to the distributed storage model.

6.14.6 Practical Usage Patterns

Static distributed tensors are useful in many common GPU programming patterns:

1. Register Blocking for Matrix Operations

```
// Optimize register usage for small matrix tiles
template<index_t M, index_t N>
struct RegisterTile {
    using Distribution = TileDistribution<
        Sequence<M, N>,
        Sequence<1, 1>  // Single thread owns entire tile
    >;
    using Tensor = StaticDistributedTensor<float, Distribution>;

    __device__ void compute() {
        Tensor tile;
        // All M*N elements in registers of one thread
        // Enables aggressive unrolling and scheduling
    }
};
```

2. Warp-Level Primitives

```
// Distribute across warp for collaborative operations
template<typename T>
struct WarpDistributedVector {
    using Distribution = TileDistribution<
        Sequence<32>,    // 32 elements
        Sequence<32>     // 32 threads in warp
    >;
    using Tensor = StaticDistributedTensor<T, Distribution>;

    __device__ T warp_reduce_sum() {
        Tensor data;
        // Each thread has one element
        // Use warp shuffle for reduction
        T value = data[0];
        for(int offset = 16; offset > 0; offset /= 2) {
            value += __shfl_down_sync(0xffffffff, value, offset);
        }
        return value;
```

(continues on next page)

(continued from previous page)

```
    }
};
```

3. Shared Memory Staging

```
// Combine with shared memory for complex patterns
// See :ref:`ck_tile_lds_bank_conflicts` for LDS optimization
template<typename T>
struct StagedComputation {
    using RegTensor = StaticDistributedTensor<T, RegDistribution>;
    using LdsTensor = StaticDistributedTensor<T, LdsDistribution>;

    __device__ void process() {
        RegTensor reg_data;
        __shared__ T shared_buffer[1024];

        // Stage 1: Compute in registers
        compute_local(reg_data);

        // Stage 2: Exchange through shared memory
        store_to_lds(reg_data, shared_buffer);
        __syncthreads();

        // Stage 3: Load different pattern
        LdsTensor lds_data;
        load_from_lds(shared_buffer, lds_data);
    }
};
```

6.14.7 Performance Considerations

Optimizing static distributed tensor usage requires understanding several *performance factors*:

Register Pressure: Each thread's local storage typically maps to registers. Excessive storage requirements can cause register spilling:

```
// Monitor register usage
template<typename T, index_t Size>
struct RegisterAnalysis {
    static constexpr index_t kRegistersPerElement = sizeof(T) / 4;
    static constexpr index_t kTotalRegisters = Size * kRegistersPerElement;

    static_assert(kTotalRegisters <= 64,
                  "Exceeds typical register budget");
};
```

Memory Coalescing: When loading/storing distributed tensors, ensure access patterns promote coalescing. See *Intro to AMD CDNA Architecture* for more information about coalescing.

```
// Good: Coalesced access pattern
template<typename Tensor>
__device__ void coalesced_store(Tensor& tensor, float* global_ptr) {
    index_t tid = threadIdx.x + blockIdx.x * blockDim.x;
```

(continues on next page)

(continued from previous page)

```
    #pragma unroll
    for(index_t i = 0; i < Tensor::kNumElements; ++i) {
        global_ptr[tid + i * gridDim.x * blockDim.x] = tensor[i];
    }
}
```

Instruction Scheduling: Organize operations to maximize instruction-level parallelism:

```
// Interleave independent operations
template<typename Tensor>
__device__ void optimized_accumulate(Tensor& acc,
                                     const Tensor& a,
                                     const Tensor& b) {
    #pragma unroll
    for(index_t i = 0; i < Tensor::kNumElements; i += 4) {
        // Group independent operations
        float tmp0 = a[i+0] * b[i+0];
        float tmp1 = a[i+1] * b[i+1];
        float tmp2 = a[i+2] * b[i+2];
        float tmp3 = a[i+3] * b[i+3];

        // Accumulate after multiplies complete
        acc[i+0] += tmp0;
        acc[i+1] += tmp1;
        acc[i+2] += tmp2;
        acc[i+3] += tmp3;
    }
}
```

6.14.8 Integration with CK Tile Ecosystem

Static distributed tensors integrate seamlessly with other CK Tile components:

```
// Complete example: Distributed GEMM kernel
template<typename ALayout, typename BLayout, typename CLayout>
__global__ void distributed_gemm_kernel(
    const float* __restrict__ a_ptr,
    const float* __restrict__ b_ptr,
    float* __restrict__ c_ptr,
    index_t M, index_t N, index_t K)
{
    // Define distributions
    constexpr index_t kTileM = 128;
    constexpr index_t kTileN = 128;
    constexpr index_t kTileK = 32;

    using ATileDist = TileDistribution<
        Sequence<kTileM, kTileK>,
        Sequence<32, 8>
    >;
    using BTileDist = TileDistribution<
        Sequence<kTileK, kTileN>,
```

(continues on next page)

(continued from previous page)

```
        Sequence<8, 32>
    >;
    using CTileDist = TileDistribution<
        Sequence<kTileM, kTileN>,
        Sequence<32, 32>
    >;

    // Create distributed accumulator
    StaticDistributedTensor<float, CTileDist> c_accumulator;

    // Initialize to zero
    #pragma unroll
    for(index_t i = 0; i < c_accumulator.kNumElements; ++i) {
        c_accumulator[i] = 0.0f;
    }

    // Main GEMM loop
    for(index_t k_tile = 0; k_tile < K; k_tile += kTileK) {
    // Create tile windows for this iteration
        auto a_window = make_tile_window(
        a_ptr, ALayout{M, K},
        ATileDist{},
        {blockIdx.y * kTileM, k_tile}
    );

        auto b_window = make_tile_window(
            b_ptr, BLayout{K, N},
            BTileDist{},
            {k_tile, blockIdx.x * kTileN}
        );

        // Load tiles to distributed tensors
        auto a_tile = a_window.load();
        auto b_tile = b_window.load();

        // Distributed matrix multiply
        distributed_gemm_accumulate(a_tile, b_tile, c_accumulator);
    }

    // Store results
    auto c_window = make_tile_window(
        c_ptr, CLayout{M, N},
        CTileDist{},
        {blockIdx.y * kTileM, blockIdx.x * kTileN}
    );
    c_window.store(c_accumulator);
}
```

6.14.9 Summary

Static distributed tensors provide the foundation for high-performance thread-local computation in CK Tile. By abstracting the complexities of register allocation, thread coordination, and memory access patterns, they enable developers to write clear, maintainable code that achieves hardware efficiency. The key benefits include:

- **Automatic Distribution**: The *tile distribution* system handles all thread-to-data mapping
- **Register Efficiency**: Thread-local storage maps directly to registers when possible
- **Zero-Overhead Abstraction**: All distribution logic resolves at compile time
- **Seamless Integration**: Works naturally with *tile windows*, *descriptors*, and other CK Tile components
- **Performance Transparency**: The storage model makes performance characteristics clear and predictable

When combined with the broader CK Tile ecosystem, static distributed tensors enable the construction of complex GPU kernels that match hand-tuned assembly performance while maintaining the clarity of high-level mathematical expressions.

6.15 Convolution Implementation with CK Tile

6.15.1 Overview

This section covers how CK Tile's *tensor descriptor* system enables efficient convolution implementations on GPUs. Convolution operations are fundamental in deep learning, and understanding their optimization reveals how high-performance libraries achieve their efficiency. This section progresses from a naive implementation to an optimized approach using tensor descriptors, showing how they enable efficient memory access patterns for GPU acceleration.

The key insight is that convolution can be transformed from a complex nested loop operation into a highly parallel matrix multiplication through the image to column (im2col) transformation. CK Tile's tensor descriptors provide the perfect abstraction for implementing this transformation efficiently without data duplication.

6.15.2 Understanding Sliding Windows

Before diving into convolution, it's crucial to understand how sliding windows work. In convolution, overlapping patches need to be extracted from the input image. Traditional approaches would copy these patches, but CK Tile uses *tensor descriptors* to create efficient *views* without data duplication.

Simple Tiling Example

Non-overlapping tiles:

```
// Create a 6x6 matrix tiled into 2x2 blocks
template<typename DataType>
struct SimpleTiling {
    static constexpr index_t kMatrixSize = 6;
    static constexpr index_t kTileSize = 2;
    static constexpr index_t kNumTiles = kMatrixSize / kTileSize;

    // Original matrix: shape=(6, 6), strides=(6, 1)
    // Tiled view: shape=(3, 3, 2, 2), strides=(12, 2, 6, 1)
    using TileDescriptor = TensorDescriptor<
        Sequence<kNumTiles, kNumTiles, kTileSize, kTileSize>,
        Sequence<12, 2, 6, 1>
    >;
```

(continues on next page)

(continued from previous page)

```
    __device__ void demonstrate() {
        // To move to next tile row: skip 2 matrix rows = 6 × 2 = 12
        // To move to next tile col: skip 2 matrix cols = 1 × 2 = 2
        // Within tile: use original strides (6, 1)
    }
};
```

The key insight is understanding **strides**, the number of elements to skip to move to the next element in each dimension. For non-overlapping tiles, we skip by `tile_size` in the outer dimensions.

Overlapping Windows for Convolution

For convolution, overlapping windows that slide by one element are needed:

```
// Extract 3x3 overlapping windows from a 6x6 image
template<typename DataType>
struct ConvolutionWindows {
    static constexpr index_t H = 6;      // Image height
    static constexpr index_t W = 6;      // Image width
    static constexpr index_t K = 3;      // Kernel size
    static constexpr index_t OutH = H - K + 1;  // Output height = 4
    static constexpr index_t OutW = W - K + 1;  // Output width = 4

    // Windows descriptor: shape=(4, 4, 3, 3), strides=(6, 1, 6, 1)
    using WindowDescriptor = TensorDescriptor<
        Sequence<OutH, OutW, K, K>,
        Sequence<W, 1, W, 1>  // Key: stride by 1 for overlap!
    >;

    __device__ DataType extract_window(const DataType* image,
                                       index_t out_i, index_t out_j,
                                       index_t k_i, index_t k_j) {
        WindowDescriptor desc;
        index_t offset = desc.calculate_offset({out_i, out_j, k_i, k_j});
        return image[offset];
    }
};
```

The stride pattern `[W, 1, W, 1]` creates sliding windows:

- Moving one step in output row: jump `W` elements (one image row)
- Moving one step in output col: jump 1 element (one image column)
- Within each window: same strides to access the 3×3 patch

6.15.3 Naive Convolution Implementation

A straightforward implementation for reference:

```
template<typename DataType>
__global__ void naive_convolution_kernel(
    const DataType* __restrict__ input,
    const DataType* __restrict__ kernel,
```

(continues on next page)

(continued from previous page)

```
    DataType* __restrict__ output,
    index_t H, index_t W, index_t K)
{
    index_t out_h = H - K + 1;
    index_t out_w = W - K + 1;

    // Each thread computes one output element
    index_t out_i = blockIdx.y * blockDim.y + threadIdx.y;
    index_t out_j = blockIdx.x * blockDim.x + threadIdx.x;

    if (out_i < out_h && out_j < out_w) {
        DataType sum = 0;

        // Extract window and apply convolution
        for (index_t ki = 0; ki < K; ++ki) {
            for (index_t kj = 0; kj < K; ++kj) {
                index_t in_i = out_i + ki;
                index_t in_j = out_j + kj;
                sum += input[in_i * W + in_j] * kernel[ki * K + kj];
            }
        }

        output[out_i * out_w + out_j] = sum;
    }
}
```

This implementation directly follows the mathematical definition but has poor memory access patterns and limited parallelism within each output computation.

6.15.4 Window Extraction with Tensor Descriptors

CK Tile's tensor descriptors provide an clean way to extract convolution windows:

```
template<typename DataType, index_t H, index_t W, index_t K>
struct ConvolutionWindowExtractor {
    static constexpr index_t OutH = H - K + 1;
    static constexpr index_t OutW = W - K + 1;

    // Create tensor descriptor for all windows
    using WindowsDescriptor = TensorDescriptor<
        Sequence<OutH, OutW, K, K>,
        Sequence<W, 1, W, 1>
    >;

    __device__ void extract_all_windows(
        const DataType* input,
        DataType* windows_buffer)
    {
        WindowsDescriptor desc;

        // Extract all windows in parallel
        index_t tid = threadIdx.x + blockIdx.x * blockDim.x;
```

(continues on next page)

(continued from previous page)

```
        index_t total_elements = OutH * OutW * K * K;

        for (index_t i = tid; i < total_elements; i += gridDim.x * blockDim.x) {
            // Convert linear index to 4D coordinates
            index_t tmp = i;
            index_t kj = tmp % K; tmp /= K;
            index_t ki = tmp % K; tmp /= K;
            index_t out_j = tmp % OutW; tmp /= OutW;
            index_t out_i = tmp;

            // Calculate source offset using descriptor
            index_t src_offset = desc.calculate_offset({out_i, out_j, ki, kj});
            windows_buffer[i] = input[src_offset];
        }
    }
};
```

The tensor descriptor automatically handles the complex indexing required for overlapping windows, making the code cleaner and less error-prone.

6.15.5 Im2col Transformation

The im2col transformation converts the 4D windows tensor into a 2D matrix suitable for matrix multiplication. This is where CK Tile's *transformation system* shines:

```
template<typename DataType, index_t OutH, index_t OutW, index_t K>
struct Im2colTransformer {
    static constexpr index_t NumWindows = OutH * OutW;
    static constexpr index_t PatchSize = K * K;

    // Step 1: Create 4D windows descriptor
    using WindowsDescriptor = TensorDescriptor<
        Sequence<OutH, OutW, K, K>,
        Sequence<W, 1, W, 1>
    >;

    // Step 2: Apply merge transforms to create 2D im2col layout
    using Im2colDescriptor = decltype(
        transform_tensor_descriptor(
            WindowsDescriptor{},
            make_tuple(
                make_merge_transform(Sequence<OutH, OutW>{}),  // Merge spatial dims
                make_merge_transform(Sequence<K, K>{})         // Merge kernel dims
            ),
            Sequence<0, 1>{},  // Merge dimensions 0,1
            Sequence<2, 3>{}   // Merge dimensions 2,3
        )
    );

    __device__ void create_im2col_matrix(
        const DataType* input,
        DataType* im2col_matrix)
```

(continues on next page)

(continued from previous page)

```
    {
        Im2colDescriptor desc;

        // Each thread handles multiple elements
        index_t tid = threadIdx.x + blockIdx.x * blockDim.x;
        index_t total_elements = NumWindows * PatchSize;

        for (index_t i = tid; i < total_elements; i += gridDim.x * blockDim.x) {
            index_t window_idx = i / PatchSize;
            index_t patch_idx = i % PatchSize;

            // Calculate source offset using merged descriptor
            index_t src_offset = desc.calculate_offset({window_idx, patch_idx});
            im2col_matrix[i] = input[src_offset];
        }
    }
};
```

The transformation pipeline: 1. Start with 4D tensor `[OutH, OutW, K, K]` 2. Merge spatial dimensions: `[OutH, OutW]` → `NumWindows` 3. Merge kernel dimensions: `[K, K]` → `PatchSize` 4. Result: 2D matrix `[NumWindows, PatchSize]`

6.15.6 Optimized Convolution Kernel

Combining all components into an optimized convolution implementation:

```
template<typename DataType,
         index_t TileM, index_t TileN, index_t TileK,
         index_t BlockM, index_t BlockN>
__global__ void optimized_convolution_kernel(
    const DataType* __restrict__ input,
    const DataType* __restrict__ kernel,
    DataType* __restrict__ output,
    index_t H, index_t W, index_t K)
{
    constexpr index_t WarpSize = 32;
    const index_t OutH = H - K + 1;
    const index_t OutW = W - K + 1;
    const index_t NumWindows = OutH * OutW;
    const index_t PatchSize = K * K;

    // Create im2col descriptor for this image size
    using Im2colDesc = TensorDescriptor<
        Sequence<NumWindows, PatchSize>,
        DynamicStrides  // Computed based on H, W, K
    >;

    // Tile distribution for matrix multiplication
    using ATileDist = TileDistribution<
        Sequence<TileM, TileK>,
        Sequence<BlockM, 1>
    >;
```

(continues on next page)

(continued from previous page)

```
    using BTileDist = TileDistribution<
        Sequence<TileK, TileN>,
        Sequence<1, BlockN>
    >;
    using CTileDist = TileDistribution<
        Sequence<TileM, TileN>,
        Sequence<BlockM, BlockN>
    >;

    // Thread-local accumulator
    StaticDistributedTensor<DataType, CTileDist> c_accumulator;

    // Initialize accumulator
    #pragma unroll
    for (index_t i = 0; i < c_accumulator.size(); ++i) {
        c_accumulator[i] = 0;
    }

    // Main GEMM loop over K dimension
    for (index_t k_tile = 0; k_tile < PatchSize; k_tile += TileK) {
        // Create tile windows for im2col matrix and kernel
        auto a_window = make_tile_window<ATileDist>(
            input, Im2colDesc{H, W, K},
            {blockIdx.y * TileM, k_tile}
        );

        auto b_window = make_tile_window<BTileDist>(
            kernel, TensorDescriptor<Sequence<PatchSize, 1>>{},
            {k_tile, 0}
        );

        // Load tiles
        auto a_tile = a_window.load();
        auto b_tile = b_window.load();

        // Synchronize after loads
        __syncthreads();

        // Local matrix multiplication
        #pragma unroll
        for (index_t m = 0; m < TileM/BlockM; ++m) {
            #pragma unroll
            for (index_t n = 0; n < TileN/BlockN; ++n) {
                #pragma unroll
                for (index_t k = 0; k < TileK; ++k) {
                    c_accumulator.at(m, n) +=
                        a_tile.at(m, k) * b_tile.at(k, n);
                }
            }
        }
    }
```

(continues on next page)

(continued from previous page)

```
    // Store results back to global memory
    auto c_window = make_tile_window<CTileDist>(
        output, TensorDescriptor<Sequence<OutH, OutW>>{OutW, 1},
        {blockIdx.y * TileM, blockIdx.x * TileN}
    );
    c_window.store(c_accumulator);
}
```

6.15.7 Multi-Channel Convolution

Real-world convolutions involve multiple input and output channels. CK Tile handles this cleanly:

```
template<typename DataType,
         index_t H, index_t W,
         index_t CIn, index_t COut,
         index_t K>
struct MultiChannelConvolution {
    static constexpr index_t OutH = H - K + 1;
    static constexpr index_t OutW = W - K + 1;
    static constexpr index_t NumWindows = OutH * OutW;
    static constexpr index_t PatchSize = K * K * CIn;

    // 5D windows descriptor [OutH, OutW, K, K, CIn]
    using Windows5D = TensorDescriptor<
        Sequence<OutH, OutW, K, K, CIn>,
        Sequence<W*CIn, CIn, W*CIn, CIn, 1>
    >;

    // Im2col: [NumWindows, PatchSize]
    using Im2colDesc = decltype(
        transform_tensor_descriptor(
            Windows5D{},
            make_tuple(
                make_merge_transform(Sequence<OutH, OutW>{}),
                make_merge_transform(Sequence<K, K, CIn>{})
            ),
            Sequence<0, 1>{},
            Sequence<2, 3, 4>{}
        )
    );

    // Filter layout: [K*K*CIn, COut]
    using FilterDesc = TensorDescriptor<
        Sequence<PatchSize, COut>,
        Sequence<COut, 1>
    >;

    __device__ void compute(
        const DataType* input,    // [H, W, CIn]
        const DataType* filters,  // [K, K, CIn, COut]
        DataType* output)         // [OutH, OutW, COut]
    {
```

(continues on next page)

(continued from previous page)

```
        // The convolution becomes a matrix multiplication:
        // [NumWindows, PatchSize] @ [PatchSize, COut] = [NumWindows, COut]
        // Then reshape to [OutH, OutW, COut]
    }
};
```

The multi-channel extension naturally follows from the single-channel case:

- Input: `[H, W, CIn]`
- Filters: `[K, K, CIn, COut]`
- Im2col matrix: `[NumWindows, K×K×CIn]`
- Output: `[OutH, OutW, COut]`

6.15.8 Performance Optimizations

CK Tile enables several optimizations for convolution:

1. Memory Coalescing

```
// Coalesced access pattern for im2col
template<index_t VectorSize>
__device__ void load_im2col_vectorized(
    const float* input,
    float* im2col_tile,
    const Im2colDescriptor& desc)
{
    using VectorType = vector_type_t<float, VectorSize>;

    // Load multiple elements per thread
    index_t tid = threadIdx.x;
    index_t stride = blockDim.x;

    for (index_t i = tid; i < NumElements; i += stride * VectorSize) {
        VectorType vec = *reinterpret_cast<const VectorType*>(&input[i]);
        *reinterpret_cast<VectorType*>(&im2col_tile[i]) = vec;
    }
}
```

2. Shared Memory Tiling

```
// Use shared memory for frequently accessed data
__shared__ float smem_a[TileM][TileK];
__shared__ float smem_b[TileK][TileN];

// Collaborative loading with proper bank conflict avoidance
auto load_tile_to_smem = [&](auto& window, float smem[][TileK]) {
    #pragma unroll
    for (index_t i = threadIdx.y; i < TileM; i += blockDim.y) {
        #pragma unroll
        for (index_t j = threadIdx.x; j < TileK; j += blockDim.x) {
            smem[i][j] = window.at(i, j);
        }
```

(continues on next page)

(continued from previous page)

```
    }
};
```

3. Register Blocking

```
// Each thread computes multiple output elements
template<index_t RegM, index_t RegN>
struct RegisterBlock {
    float c_reg[RegM][RegN];

    __device__ void compute(const float* a_smem, const float* b_smem) {
        #pragma unroll
        for (index_t k = 0; k < TileK; ++k) {
            #pragma unroll
            for (index_t m = 0; m < RegM; ++m) {
                #pragma unroll
                for (index_t n = 0; n < RegN; ++n) {
                    c_reg[m][n] += a_smem[m] * b_smem[n];
                }
            }
        }
    }
};
```

6.15.9 Performance Characteristics

The tensor descriptor approach provides optimal performance characteristics:

Table 1: Method Comparison

Method	Memory Usage	Parallelization	GPU Efficiency	Flexibility
Naive loops	Low	Poor	Poor	High
Direct im2col copy	High	Excellent	Good	Medium
Tensor descriptors	Medium	Excellent	Excellent	High
CK Tile optimized	Low	Excellent	Excellent	High

Key advantages of the CK Tile approach:

1. **Zero-copy views**: Tensor descriptors create logical views without data duplication
2. **Compile-time optimization**: All indexing calculations resolve at compile time
3. **Hardware-aware**: Automatic alignment and vectorization based on *architecture*
4. **Composability**: Complex access patterns built from simple *transformations*
5. **Performance portability**: Same code optimizes differently for different GPUs

6.15.10 Summary

This example demonstrates how CK Tile transforms convolution from a memory-bound operation with poor parallelism into a compute-bound operation that utilizes GPU resources. The key insights are:

- **Sliding windows** can be efficiently represented using tensor descriptors with appropriate strides
- **Im2col transformation** converts convolution to matrix multiplication without data copies

- **Tile distribution** enables optimal work distribution across GPU threads (see *Tile Distribution - The Core API*)
- **Multi-channel support** extends naturally through higher-dimensional descriptors
- **Performance optimizations** like vectorization and shared memory are seamlessly integrated (see *A Block GEMM on MI300* for similar techniques)

The tensor descriptor system provides a unified framework for these transformations, enabling automatic generation of efficient kernels for various convolution configurations and hardware architectures. This approach forms the foundation for production deep learning frameworks' convolution implementations.

6.16 Advanced Coordinate Movement

6.16.1 Overview

Advanced coordinate operations form the bridge between mathematical transformations and practical tensor manipulation in CK Tile. These operations enable efficient navigation through complex tensor layouts without recalculating entire transformation chains. Understanding coordinate movement is essential for implementing high-performance GPU kernels that traverse multi-dimensional data structures.

The coordinate movement system provides two key abstractions: TensorCoordinate for descriptor-aware navigation and TensorAdaptorCoordinate for tracking positions through transformation chains. Together with movement functions, they enable advanced access patterns while maintaining optimal performance through incremental updates rather than full recalculation.

For the mathematical foundations of coordinate systems, see *Coordinate Systems - The Mathematical Foundation*. For simpler coordinate concepts, see *Tensor Coordinates*.

6.16.2 TensorCoordinate: Descriptor-Aware Navigation

TensorCoordinate combines a multi-dimensional position with descriptor context to provide efficient offset calculation and validation. It caches transformation results to avoid redundant computations during navigation. This builds on the *Tensor Descriptors - Complete Tensor Specifications* concepts for tensor specifications.

Basic Structure

```
template<typename TensorDescriptor>
class TensorCoordinate {
private:
    MultiIndex top_index_;      // Position in top dimensions
    MultiIndex hidden_index_;   // Cached transformation results
    index_t offset_;           // Cached linear offset

public:
    // Create coordinate from descriptor and position
    __host__ __device__ TensorCoordinate(
        const TensorDescriptor& desc,
        const MultiIndex& top_index)
    {
        top_index_ = top_index;
        // Apply descriptor transforms to compute hidden indices
        hidden_index_ = desc.calculate_bottom_index(top_index);
        offset_ = desc.calculate_offset(top_index);
    }
```

(continues on next page)

(continued from previous page)

```
    // Access methods
    __host__ __device__ const MultiIndex& get_index() const {
        return top_index_;
    }

    __host__ __device__ index_t get_offset() const {
        return offset_;
    }

    __host__ __device__ index_t ndim_hidden() const {
        return hidden_index_.size();
    }
};
```

Creating and Using TensorCoordinate

```
// Example: Navigate a 4x3 matrix with custom strides
template<typename DataType>
__device__ void demonstrate_tensor_coordinate() {
    // Create descriptor for 4x3 matrix, row-major layout
    using Desc = TensorDescriptor<
        Sequence<4, 3>,    // Shape
        Sequence<3, 1>     // Strides
    >;
    Desc desc;

    // Create coordinate at position [2, 1]
    auto coord = make_tensor_coordinate(desc, make_multi_index(2, 1));

    // Access coordinate information
    auto position = coord.get_index();        // [2, 1]
    auto offset = coord.get_offset();         // 2*3 + 1 = 7
    auto hidden_dims = coord.ndim_hidden();  // 0 (no hidden dims)

    // Use offset for memory access
    DataType* tensor_data = ...;
    DataType value = tensor_data[offset];
}
```

Key Benefits

1. **Context Preservation**: The coordinate maintains descriptor context for validation
2. **Cached Calculations**: Transformation results are cached for efficiency
3. **Type Safety**: Compile-time checking ensures coordinate-descriptor compatibility
4. **Zero Overhead**: All operations resolve at compile time when possible

6.16.3 TensorAdaptorCoordinate: Transform-Aware Tracking

TensorAdaptorCoordinate extends the concept to track coordinates through transformation chains, maintaining both input (top) and output (bottom) positions. This leverages *Tensor Adaptors - Chaining Transformations* and *Individual Transform Operations* for complex coordinate mappings.

Structure and Implementation

```
template<typename TensorAdaptor>
class TensorAdaptorCoordinate {
private:
    MultiIndex top_index_;      // Input position
    MultiIndex bottom_index_;   // Output after transformations
    MultiIndex hidden_index_;   // Intermediate results

public:
    // Create from adaptor and position
    __host__ __device__ TensorAdaptorCoordinate(
        const TensorAdaptor& adaptor,
        const MultiIndex& top_index)
    {
        top_index_ = top_index;
        // Apply adaptor transforms
        bottom_index_ = adaptor.calculate_bottom_index(top_index);
        // Cache intermediate results
        hidden_index_ = adaptor.get_hidden_index(top_index);
    }

    // Access transformed coordinates
    __host__ __device__ const MultiIndex& get_top_index() const {
        return top_index_;
    }

    __host__ __device__ const MultiIndex& get_bottom_index() const {
        return bottom_index_;
    }
};
```

Tracking Through Transformations

```
// Example: Track coordinates through transpose
template<typename DataType>
__device__ void demonstrate_adaptor_coordinate() {
    // Create transpose adaptor (swap dimensions)
    auto adaptor = make_transpose_adaptor<2>(Sequence<1, 0>{});

    // Create coordinate at [2, 3]
    auto coord = make_tensor_adaptor_coordinate(
        adaptor,
        make_multi_index(2, 3)
    );

    // Track transformation
```

(continues on next page)

(continued from previous page)

```
    auto input_pos = coord.get_top_index();     // [2, 3]
    auto output_pos = coord.get_bottom_index();  // [3, 2] (swapped)

    // Use for complex access patterns
    DataType* src_data = ...;
    DataType* dst_data = ...;

    // Read from transposed position
    index_t src_offset = calculate_offset(output_pos);
    DataType value = src_data[src_offset];
}
```

6.16.4 Efficient Coordinate Movement

The `move_tensor_coordinate` function provides efficient navigation by updating coordinates incrementally rather than recreating them.

Basic Movement Operations

```
// Move tensor coordinate through descriptor
template<typename TensorDescriptor>
__host__ __device__ void move_tensor_coordinate(
    const TensorDescriptor& desc,
    TensorCoordinate<TensorDescriptor>& coord,
    const MultiIndex& step)
{
    // Update top index
    coord.top_index_ += step;

    // Incrementally update cached values
    // Only recalculate affected transformations
    if (transformation_affects_movement(desc, step)) {
        coord.hidden_index_ = desc.calculate_bottom_index(coord.top_index_);
        coord.offset_ = desc.calculate_offset(coord.top_index_);
    } else {
        // Fast path: simple offset update
        coord.offset_ += calculate_step_offset(desc, step);
    }
}
```

Practical Movement Patterns

```
// Example: Efficient matrix traversal
template<typename DataType>
__global__ void matrix_traversal_kernel(
    const DataType* input,
    DataType* output,
    index_t rows, index_t cols)
{
    // Create descriptor for matrix
    using Desc = TensorDescriptor<DynamicSequence, DynamicSequence>;
```

(continues on next page)

(continued from previous page)

```
    Desc desc(make_tuple(rows, cols), make_tuple(cols, 1));

    // Start at thread's assigned position
    index_t start_row = blockIdx.y * blockDim.y + threadIdx.y;
    index_t start_col = blockIdx.x * blockDim.x + threadIdx.x;

    auto coord = make_tensor_coordinate(
        desc,
        make_multi_index(start_row, start_col)
    );

    // Row-wise traversal pattern
    for (index_t i = 0; i < 4; ++i) {
        if (coord.get_index()[0] < rows) {
            // Process current position
            output[coord.get_offset()] =
                process_value(input[coord.get_offset()]);

            // Move to next column
            move_tensor_coordinate(desc, coord, make_multi_index(0, 1));

            // Wrap to next row if needed
            if (coord.get_index()[1] >= cols) {
                move_tensor_coordinate(
                    desc, coord,
                    make_multi_index(1, -cols)
                );
            }
        }
    }
}
```

Movement Through Adaptors

```
// Move through adaptor transformations
template<typename TensorAdaptor>
__host__ __device__ MultiIndex move_tensor_adaptor_coordinate(
    const TensorAdaptor& adaptor,
    TensorAdaptorCoordinate<TensorAdaptor>& coord,
    const MultiIndex& step)
{
    // Update top index
    MultiIndex old_top = coord.top_index_;
    coord.top_index_ += step;

    // Calculate new bottom index
    MultiIndex old_bottom = coord.bottom_index_;
    coord.bottom_index_ = adaptor.calculate_bottom_index(coord.top_index_);

    // Return the change in bottom coordinates
    return coord.bottom_index_ - old_bottom;
}
```

6.16.5 Advanced Movement Patterns

Real-world applications use advanced movement patterns for optimal memory access. These patterns often relate to *Tile Window - Data Access Gateway* operations and *Tile Distribution - The Core API* concepts:

Tiled Access Pattern

```
template<index_t TileM, index_t TileN>
__device__ void tiled_movement_pattern(
    const float* input,
    float* output,
    index_t M, index_t N)
{
    // Descriptor for full matrix
    using MatrixDesc = TensorDescriptor<
        DynamicSequence,
        DynamicSequence
    >;
    MatrixDesc desc(make_tuple(M, N), make_tuple(N, 1));

    // Start at tile corner
    index_t tile_row = blockIdx.y * TileM;
    index_t tile_col = blockIdx.x * TileN;

    auto coord = make_tensor_coordinate(
        desc,
        make_multi_index(tile_row, tile_col)
    );

    // Process tile with efficient movement
    #pragma unroll
    for (index_t i = 0; i < TileM; ++i) {
        #pragma unroll
        for (index_t j = 0; j < TileN; ++j) {
            if (i == 0 && j == 0) {
                // First element - already positioned
            } else if (j == 0) {
                // New row - move down and back to start column
                move_tensor_coordinate(
                    desc, coord,
                    make_multi_index(1, -(TileN-1))
                );
            } else {
                // Same row - move right
                move_tensor_coordinate(
                    desc, coord,
                    make_multi_index(0, 1)
                );
            }

            // Process element
            output[coord.get_offset()] =
                compute_value(input[coord.get_offset()]);
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Space-Filling Curve Movement

For more details on space-filling curves and their benefits, see *Space-Filling Curves - Optimal Memory Traversal*.

```
// Snake pattern for optimal cache usage
template<index_t BlockSize>
__device__ void snake_pattern_movement(
    const float* input,
    float* output,
    index_t M, index_t N)
{
    using Desc = TensorDescriptor<DynamicSequence, DynamicSequence>;
    Desc desc(make_tuple(M, N), make_tuple(N, 1));

    auto coord = make_tensor_coordinate(
        desc,
        make_multi_index(threadIdx.y, threadIdx.x)
    );

    // Snake through block
    for (index_t row = 0; row < BlockSize; ++row) {
        for (index_t col = 0; col < BlockSize; ++col) {
            // Process current position
            process_element(input, output, coord.get_offset());

            // Snake movement pattern
            if (row % 2 == 0) {
                // Even rows: move right
                if (col < BlockSize - 1) {
                    move_tensor_coordinate(
                        desc, coord, make_multi_index(0, 1)
                    );
                }
            } else {
                // Odd rows: move left
                if (col < BlockSize - 1) {
                    move_tensor_coordinate(
                        desc, coord, make_multi_index(0, -1)
                    );
                }
            }
        }

        // Move to next row
        if (row < BlockSize - 1) {
            move_tensor_coordinate(
                desc, coord, make_multi_index(1, 0)
            );
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

6.16.6 Performance Considerations

Efficient coordinate movement is critical for GPU performance. See *Intro to AMD CDNA Architecture* for hardware details.

1. Incremental Updates

```
// Inefficient: recreate coordinate
for (index_t i = 0; i < N; ++i) {
    auto coord = make_tensor_coordinate(desc, make_multi_index(i, j));
    process(data[coord.get_offset()]);
}

// Efficient: incremental movement
auto coord = make_tensor_coordinate(desc, make_multi_index(0, j));
for (index_t i = 0; i < N; ++i) {
    process(data[coord.get_offset()]);
    move_tensor_coordinate(desc, coord, make_multi_index(1, 0));
}
```

2. Movement Caching

```
// Cache frequently used movements
template<typename Desc>
struct MovementCache {
    MultiIndex row_step = make_multi_index(1, 0);
    MultiIndex col_step = make_multi_index(0, 1);
    MultiIndex diag_step = make_multi_index(1, 1);

    __device__ void move_row(auto& coord) {
        move_tensor_coordinate(Desc{}, coord, row_step);
    }
};
```

3. Vectorized Movement

```
// Move multiple coordinates simultaneously
template<index_t NumCoords>
__device__ void vectorized_movement(
    TensorCoordinate<Desc> coords[NumCoords],
    const MultiIndex& step)
{
    #pragma unroll
    for (index_t i = 0; i < NumCoords; ++i) {
        move_tensor_coordinate(Desc{}, coords[i], step);
    }
}
```

6.16.7 Integration with CK Tile Components

Coordinate movement integrates seamlessly with other CK Tile components:

```
// Example: Tile window with coordinate movement
template<typename TileWindow>
__device__ void process_tile_with_movement(
    TileWindow& window,
    index_t tile_size)
{
    // Create coordinate for tile traversal
    auto coord = window.get_tile_coordinate();

    // Process tile elements with movement
    for (index_t i = 0; i < tile_size; ++i) {
        for (index_t j = 0; j < tile_size; ++j) {
            // Load using coordinate
            auto value = window.load_at(coord);

            // Process value
            auto result = compute(value);

            // Store result
            window.store_at(coord, result);

            // Move to next element
            window.move_coordinate(coord, {0, 1});
        }
        // Move to next row
        window.move_coordinate(coord, {1, -tile_size});
    }
}
```

Advanced coordinate operations provide the foundation for efficient tensor navigation in CK Tile:

- **TensorCoordinate**: Combines position with descriptor context for validated navigation
- **TensorAdaptorCoordinate**: Tracks coordinates through transformation chains
- **move_tensor_coordinate**: Enables efficient incremental updates without recalculation
- **Movement Patterns**: Support advanced access patterns like tiling and space-filling curves
- **Performance**: Incremental updates are orders of magnitude faster than coordinate recreation
- **Integration**: Seamlessly works with tile windows, distributions, and other CK Tile components

These operations are essential for implementing high-performance GPU kernels that can navigate complex tensor layouts efficiently. By understanding and utilizing coordinate movement, kernels can be created that achieve optimal memory access patterns while maintaining code clarity and correctness.

6.17 Load Data Share Index Swapping

6.17.1 Overview

Local Data Share (LDS) index swapping, also known as XOR preshuffle, is a critical optimization technique in CK Tile for resolving bank conflicts in shared memory. Bank conflicts occur when multiple threads in a warp attempt to access different addresses within the same memory bank simultaneously, causing serialization and performance

degradation. CK Tile generalizes the XOR preshuffle technique through a compile-time coordinate transformation system that automatically handles complex access patterns.

The key insight is that transforming the logical 2D coordinates used to access LDS into a different 2D coordinate space ensures that threads accessing data simultaneously access different memory banks. This transformation is implemented through CK Tile's composable transform system, making it both flexible and efficient. See *Individual Transform Operations* and *Coordinate Systems - The Mathematical Foundation* for more information about the composable transform system.

6.17.2 Coordinate Transformation Pipeline

CK Tile performs coordinate transformations to bring LDS access from the original 2D position (M, K dimensions) into transformed (M', K') coordinates:

Step 1: XOR Transform

The original K coordinate is split into K0 and K1, where K1 represents the thread vector size along the K dimension (KPack) and K0 is KPerBlock/KPack.

The XOR transformation updates the K0 coordinate using the formula:

$$K0' = K0^{(M\%(KPerBlock/KPack*MLdsLayer))}$$

This XOR operation redistributes accesses across memory banks by mixing bits from the M and K dimensions.

Step 2: Unmerge Transform

The transformed K0' is split into L and K0" components, creating an intermediate 4D coordinate space. This is necessary when MLdsLayer > 1, allowing multiple rows to share the same set of memory banks for better utilization with smaller tile sizes.

The unmerge operation:

$$L = K0'/(KPerBlock/KPack)K0'' = K0'\%(KPerBlock/KPack)$$

When MLdsLayer == 1, this simplifies to L=0 and K0"=K0'.

Step 3: Merge Transform

The final step merges the 4D coordinates back into 2D transformed coordinates (M', K').

6.17.3 C++ Implementation

Here's how the complete transformation chain is implemented in CK Tile using *Tensor Descriptors - Complete Tensor Specifications* and transforms:

```
template<index_t KPerBlock,
         index_t KPack,
         index_t MLdsLayer,
         index_t MPerBlock>
struct LdsIndexSwapping {
    static constexpr index_t KPerBlock_over_KPack = KPerBlock / KPack;
    static constexpr index_t MPerBlock_over_MLdsLayer = MPerBlock / MLdsLayer;

    // Step 1: Create base descriptor
    using BaseLengths = Sequence<
        KPerBlock_over_KPack * MLdsLayer,
```

(continues on next page)

(continued from previous page)

```
        MPerBlock_over_MLdsLayer,
        KPack
    >;
    using BaseStrides = Sequence<
        KPack,
        KPerBlock * MLdsLayer,
        1
    >;

    using BaseDescriptor = TensorDescriptor<BaseLengths, BaseStrides>;

    // Step 2: Apply XOR transform
    using PermutedDescriptor = decltype(
        transform_tensor_descriptor(
            BaseDescriptor{},
            make_tuple(
                make_xor_transform(
                    Sequence<MPerBlock_over_MLdsLayer,
                             KPerBlock_over_KPack * MLdsLayer>{}
                ),
                make_pass_through_transform(Number<KPack>{})
            ),
            Sequence<1, 0>{},  // XOR on dims [1,0]
            Sequence<2>{}      // Pass through dim 2
        )
    );

    // Step 3: Apply unmerge and final transforms
    using FinalDescriptor = decltype(
        transform_tensor_descriptor(
            PermutedDescriptor{},
            make_tuple(
                make_unmerge_transform(
                    Sequence<MLdsLayer, KPerBlock_over_KPack>{}
                ),
                make_pass_through_transform(Number<MPerBlock_over_MLdsLayer>{}),
                make_pass_through_transform(Number<KPack>{})
            ),
            Sequence<0>{},      // Unmerge dim 0
            Sequence<1>{},      // Pass through dim 1
            Sequence<2>{},      // Pass through dim 2
            Sequence<0, 2>{},   // Output dims from unmerge
            Sequence<1>{},      // Output dim 1
            Sequence<3>{}       // Output dim 3
        )
    );
};
```

6.17.4 Practical Usage in GEMM

Here's how LDS index swapping is used in a real GEMM kernel. See *A Block GEMM on MI300* for more information about GEMM optimization.

```
template<typename DataType,
         index_t BlockM, index_t BlockN, index_t BlockK,
         index_t KPack>
__global__ void gemm_kernel_with_lds_swapping(
    const DataType* __restrict__ a_global,
    const DataType* __restrict__ b_global,
    DataType* __restrict__ c_global,
    index_t M, index_t N, index_t K)
{
    // Shared memory allocation
    __shared__ DataType a_lds[BlockM * BlockK];
    __shared__ DataType b_lds[BlockK * BlockN];

    // Create LDS descriptor with index swapping
    constexpr index_t MLdsLayer = 2;  // Typical value for bank conflict avoidance

    using ALdsDesc = typename LdsIndexSwapping<
        BlockK, KPack, MLdsLayer, BlockM
    >::FinalDescriptor;

    // Load from global to LDS with swapped indices
    auto load_a_to_lds = [&](index_t k_offset) {
        // Each thread loads its portion
        index_t tid = threadIdx.x;
        constexpr index_t NumThreads = blockDim.x;
        constexpr index_t ElementsPerThread = (BlockM * BlockK) / NumThreads;

        #pragma unroll
        for (index_t i = 0; i < ElementsPerThread; ++i) {
            index_t linear_idx = tid * ElementsPerThread + i;

            // Convert linear index to 2D coordinates
            index_t m_idx = linear_idx / BlockK;
            index_t k_idx = linear_idx % BlockK;

            // Load from global memory
            DataType value = a_global[
                (blockIdx.y * BlockM + m_idx) * K + k_offset + k_idx
            ];

            // Store to LDS using swapped coordinates
            ALdsDesc desc;
            index_t lds_offset = desc.calculate_offset({
                0,                  // L component (for this example)
                m_idx / MLdsLayer,  // M component
                k_idx / KPack,      // K0 component
                k_idx % KPack       // K1 component
            });
```

(continues on next page)

(continued from previous page)

```
            a_lds[lds_offset] = value;
        }
    };

    // Main GEMM computation loop
    for (index_t k = 0; k < K; k += BlockK) {
        // Load tiles to LDS with index swapping
        load_a_to_lds(k);
        __syncthreads();

        // Compute using swapped LDS layout
        // ... (matrix multiplication using transformed coordinates)
    }
}
```

6.17.5 Bank Conflict Analysis

The effectiveness of index swapping can be analyzed by examining access patterns:

```
template<index_t WarpSize = 32>
struct BankConflictAnalyzer {
    static constexpr index_t NumBanks = 32;
    static constexpr index_t BankWidth = 4;  // 4 bytes per bank

    template<typename LdsDescriptor>
    static void analyze_access_pattern() {
        // Simulate warp access pattern
        index_t bank_access[NumBanks] = {0};

        // Each thread in warp accesses one element
        for (index_t tid = 0; tid < WarpSize; ++tid) {
            // Calculate coordinates for this thread
            index_t m_coord = tid / 8;  // Example mapping
            index_t k_coord = tid % 8;

            // Get LDS offset using descriptor
            LdsDescriptor desc;
            index_t offset = desc.calculate_offset({m_coord, k_coord});

            // Determine bank
            index_t bank = (offset * sizeof(float) / BankWidth) % NumBanks;
            bank_access[bank]++;
        }

        // Check for conflicts
        index_t max_conflict = 0;
        for (index_t bank = 0; bank < NumBanks; ++bank) {
            max_conflict = max(max_conflict, bank_access[bank]);
        }

        printf("Max bank conflict: %d-way\n", max_conflict);
    }
```

(continues on next page)

(continued from previous page)

```
};
```

6.17.6 Performance Benefits

LDS index swapping provides several key benefits:

1. **Conflict-Free Access**: Eliminates or significantly reduces bank conflicts
2. **Higher Throughput**: Enables full memory bandwidth utilization
3. **Automatic Optimization**: Transformation parameters can be tuned per architecture
4. **Composability**: Integrates seamlessly with other CK Tile transformations

6.17.7 Advanced Configurations

Different configurations can be used based on tile sizes and data types:

```
// Configuration for different scenarios
template<typename DataType, index_t TileSize>
struct LdsSwappingConfig {
    // Smaller tiles may need different MLdsLayer
    static constexpr index_t MLdsLayer =
        (TileSize <= 32) ? 1 :
        (TileSize <= 64) ? 2 : 4;

    // Adjust KPack based on data type
    static constexpr index_t KPack =
        sizeof(DataType) == 2 ? 8 :    // FP16/BF16
        sizeof(DataType) == 4 ? 4 : 2;  // FP32

    // Validate configuration
    static_assert(TileSize % (MLdsLayer * KPack) == 0,
                  "Tile size must be divisible by MLdsLayer * KPack");
};
```

6.17.8 Integration with Tile Distribution

LDS index swapping works seamlessly with CK Tile's distribution system. See *Tile Distribution - The Core API* for more information about CK Tile's distribution system.

```
template<typename TileDistribution>
struct DistributedLdsAccess {
    using LdsDesc = typename LdsIndexSwapping<...>::FinalDescriptor;

    __device__ void load_from_lds(
        const float* lds_ptr,
        StaticDistributedTensor<float, TileDistribution>& reg_tensor)
    {
        // Each thread loads its distributed portion
        auto coord = make_tensor_coordinate(LdsDesc{}, {0, 0, 0, 0});

        #pragma unroll
        for (index_t i = 0; i < reg_tensor.size(); ++i) {
```

(continues on next page)

(continued from previous page)

```
            // Calculate swapped LDS coordinates for this element
            auto [m, k] = TileDistribution::get_local_tile_index(i);

            // Move to correct position
            move_tensor_coordinate(LdsDesc{}, coord, {0, m, k/4, k%4});

            // Load with transformed coordinates
            reg_tensor[i] = lds_ptr[coord.get_offset()];
        }
    }
};
```

6.17.9 Summary

LDS index swapping in CK Tile provides a effective and flexible solution to the bank conflict problem:

- **Generalized XOR Preshuffle**: Extends the basic XOR technique through composable transforms
- **Multi-Step Pipeline**: Coordinates flow through XOR → Unmerge → Merge transformations
- **Automatic Optimization**: Parameters like MLdsLayer adapt to tile sizes and data types
- **Zero Overhead**: All transformations resolve at compile time
- **Seamless Integration**: Works naturally with other CK Tile components

By understanding and utilizing LDS index swapping, kernels can achieve maximum shared memory bandwidth, which is often the limiting factor in GPU kernel performance. The transformation-based approach makes it easy to experiment with different swapping strategies while maintaining code clarity.

For practical examples of how index swapping is used in complete kernels, see *Memory Swizzling with Morton Ordering*. For more on coordinate operations used here, see *Advanced Coordinate Movement* and *Tensor Coordinates*.

6.18 Memory Swizzling with Morton Ordering

6.18.1 Overview

This chapter demonstrates a practical application of tensor descriptors for implementing memory swizzling patterns, specifically Morton ordering (Z-order curve) within tiles. Memory swizzling is used to optimize GPU memory access patterns and reduce *bank conflicts*. Morton ordering provides a space-filling curve that maintains spatial locality while enabling efficient parallel access. See *Space-Filling Curves - Optimal Memory Traversal* for more information about parallel access.

Morton ordering is widely used in:

- **GPU Texture Memory**: Optimizing cache efficiency for 2D texture access
- **Matrix Operations**: Reducing memory bank conflicts in shared memory
- **Image Processing**: Improving locality for block-based algorithms
- **Scientific Computing**: Enhancing data access patterns for stencil operations

6.18.2 Understanding Morton Ordering

Morton ordering interleaves the bits of 2D coordinates to create a 1D ordering that preserves spatial locality. For a 2D coordinate (y, x), we split each coordinate into its binary bits and interleave them:

- y = $y_1 y_0$ (2 bits)
- x = $x_1 x_0$ (2 bits)
- Morton index = $y_1 x_1 y_0 x_0$ (4 bits)

This creates a Z-shaped traversal pattern within each tile:

```
// Morton encoding for 2D coordinates
template<index_t NumBits = 2>
__host__ __device__ index_t morton_encode_2d(index_t y, index_t x) {
    index_t result = 0;
    for (index_t i = 0; i < NumBits; ++i) {
        index_t bit_y = (y >> i) & 1;
        index_t bit_x = (x >> i) & 1;
        result |= (bit_y << (2*i + 1)) | (bit_x << (2*i));
    }
    return result;
}

// Morton decoding back to 2D coordinates
template<index_t NumBits = 2>
__host__ __device__ void morton_decode_2d(
    index_t morton_idx,
    index_t& y,
    index_t& x)
{
    y = 0;
    x = 0;
    for (index_t i = 0; i < NumBits; ++i) {
        y |= ((morton_idx >> (2*i + 1)) & 1) << i;
        x |= ((morton_idx >> (2*i)) & 1) << i;
    }
}
```

Morton Pattern Analysis

The Morton index layout in a 4×4 tile follows this pattern:

```
Morton Index Layout:
 0  1  4  5
 2  3  6  7
 8  9 12 13
10 11 14 15
```

Bit pattern breakdown:

```
(0,0) = (00, 00) →  0 = 0000
(0,1) = (00, 01) →  1 = 0001
(0,2) = (00, 10) →  4 = 0100
(0,3) = (00, 11) →  5 = 0101
```

(continues on next page)

(continued from previous page)

```
(1,0) = (01, 00) →  2 = 0010
(1,1) = (01, 01) →  3 = 0011
(1,2) = (01, 10) →  6 = 0110
(1,3) = (01, 11) →  7 = 0111
```

6.18.3 Stage 1: Tiling with UnmergeTransform

First, we split our texture into tiles using tensor descriptors (see *Tensor Descriptors - Complete Tensor Specifications* and *Individual Transform Operations*). This creates a hierarchical structure: (Y_blk, y_in, X_blk, x_in).

```
template<index_t H, index_t W, index_t TileSize>
struct TiledTextureDescriptor {
    static constexpr index_t NumTilesY = H / TileSize;
    static constexpr index_t NumTilesX = W / TileSize;

    // Original descriptor for H×W texture
    using BaseDesc = TensorDescriptor<
        Sequence<H, W>,
        Sequence<W, 1>  // Row-major layout
    >;

    // Stage 1: Split into tiles
    // Transform: [H, W] → [NumTilesY, TileSize, NumTilesX, TileSize]
    using TiledDesc = decltype(
        transform_tensor_descriptor(
            BaseDesc{},
            make_tuple(
                make_unmerge_transform(Sequence<NumTilesY, TileSize>{}),
                make_unmerge_transform(Sequence<NumTilesX, TileSize>{})
            ),
            Sequence<0>{},     // Y dimension
            Sequence<1>{},     // X dimension
            Sequence<0, 1>{},  // Y → (Y_blk, y_in)
            Sequence<2, 3>{}   // X → (X_blk, x_in)
        )
    );
};
```

Example usage for an 8×8 texture with 4×4 tiles:

```
// Create tiled descriptor
using TiledDesc8x8 = TiledTextureDescriptor<8, 8, 4>::TiledDesc;

// Access pattern: iterate tile by tile
template<typename DataType>
__device__ void process_tiled_texture(const DataType* texture) {
    TiledDesc8x8 desc;

    // Process each tile
    for (index_t y_blk = 0; y_blk < 2; ++y_blk) {
        for (index_t x_blk = 0; x_blk < 2; ++x_blk) {
            // Process elements within tile
```

(continues on next page)

(continued from previous page)

```
            for (index_t y_in = 0; y_in < 4; ++y_in) {
                for (index_t x_in = 0; x_in < 4; ++x_in) {
                    // Calculate offset using descriptor
                    index_t offset = desc.calculate_offset({
                        y_blk, y_in, x_blk, x_in
                    });

                    DataType value = texture[offset];
                    // Process value...
                }
            }
        }
    }
}
```

6.18.4 Stage 2: Morton Ordering with MergeTransform

The key insight is that MergeTransform enables Morton ordering by reordering and merging coordinate bits. The transformation involves:

1. Split coordinates into individual bits using UnmergeTransform
2. Reorder and merge bits using MergeTransform to create the Morton index

This leverages the coordinate transformation system described in *Coordinate Systems - The Mathematical Foundation*.

Mathematical Foundation

```
template<index_t TileSize = 4>
struct MortonTransform {
    static_assert(TileSize == 4, "This example assumes 4x4 tiles");

    // Split 4 → (2, 2) for bit extraction
    using SplitTransform = UnmergeTransform<Sequence<2, 2>>;

    // Merge bits in Morton order: (y₀, x₀, y₁, x₁) → Morton
    using MortonMergeTransform = MergeTransform<Sequence<2, 2, 2, 2>>;

    // The merge operation computes:
    // morton_idx = y₁×8 + x₁×4 + y₀×2 + x₀
    // This matches the bit interleaving pattern!
};
```

Complete Morton Implementation

Here's a complete implementation combining both stages:

```
template<typename DataType, index_t H, index_t W, index_t TileSize = 4>
struct MortonSwizzledTexture {
    static constexpr index_t NumTilesY = H / TileSize;
    static constexpr index_t NumTilesX = W / TileSize;

    // Manual Morton implementation for reliability
```

(continues on next page)

(continued from previous page)

```
    __device__ static void apply_morton_swizzling(
        const DataType* input,
        DataType* output)
    {
        // Process each tile
        for (index_t tile_y = 0; tile_y < NumTilesY; ++tile_y) {
            for (index_t tile_x = 0; tile_x < NumTilesX; ++tile_x) {
                // Apply Morton ordering within tile
                for (index_t morton_idx = 0; morton_idx < TileSize * TileSize; ++morton_
↪idx) {
                    // Decode Morton index to tile coordinates
                    index_t y_in, x_in;
                    morton_decode_2d<2>(morton_idx, y_in, x_in);

                    // Calculate global coordinates
                    index_t global_y = tile_y * TileSize + y_in;
                    index_t global_x = tile_x * TileSize + x_in;

                    // Calculate linear indices
                    index_t src_idx = global_y * W + global_x;
                    index_t dst_idx = (tile_y * NumTilesX + tile_x) * TileSize *␣
↪TileSize + morton_idx;

                    output[dst_idx] = input[src_idx];
                }
            }
        }
    }
};
```

6.18.5 Memory Access Pattern Analysis

An analysis of the benefits of Morton ordering for different access patterns:

```
template<index_t TileSize = 4>
struct AccessPatternAnalyzer {
    // Analyze spatial locality
    __host__ static void analyze_morton_locality() {
        printf("Morton Order Spatial Locality Analysis:\n");
        printf("Adjacent indices and their 2D distance:\n");

        for (index_t i = 0; i < TileSize * TileSize - 1; ++i) {
            index_t y1, x1, y2, x2;
            morton_decode_2d<2>(i, y1, x1);
            morton_decode_2d<2>(i + 1, y2, x2);

            index_t manhattan_dist = abs(y2 - y1) + abs(x2 - x1);
            printf("Morton %2d→%2d: (%d,%d)→(%d,%d), distance: %d\n",
                   i, i+1, y1, x1, y2, x2, manhattan_dist);
        }
    }
```

(continues on next page)

(continued from previous page)

```
    // Compare cache line usage
    __host__ static void analyze_cache_efficiency() {
        constexpr index_t CacheLineSize = 128;  // bytes
        constexpr index_t ElementSize = sizeof(float);
        constexpr index_t ElementsPerCacheLine = CacheLineSize / ElementSize;

        printf("\nCache Efficiency Analysis:\n");
        printf("Cache line size: %d bytes (%d floats)\n",
               CacheLineSize, ElementsPerCacheLine);

        // Row-major access
        index_t row_major_lines = 0;
        for (index_t y = 0; y < TileSize; ++y) {
            for (index_t x = 0; x < TileSize; x += ElementsPerCacheLine) {
                row_major_lines++;
            }
        }

        // Morton access
        index_t morton_lines = 0;
        index_t current_line = -1;
        for (index_t i = 0; i < TileSize * TileSize; ++i) {
            index_t y, x;
            morton_decode_2d<2>(i, y, x);
            index_t linear_idx = y * TileSize + x;
            index_t cache_line = linear_idx / ElementsPerCacheLine;

            if (cache_line != current_line) {
                morton_lines++;
                current_line = cache_line;
            }
        }

        printf("Row-major: %d cache lines\n", row_major_lines);
        printf("Morton: %d cache lines\n", morton_lines);
    }
};
```

6.18.6 GPU Kernel Implementation

A complete GPU kernel using Morton ordering for optimized memory access:

```
template<typename DataType,
         index_t BlockSize = 16,
         index_t TileSize = 4>
__global__ void morton_optimized_kernel(
    const DataType* __restrict__ input,
    DataType* __restrict__ output,
    index_t H, index_t W)
{
    // Shared memory with Morton layout
    __shared__ DataType smem[BlockSize * BlockSize];
```

(continues on next page)

(continued from previous page)

```
    // Thread and block indices
    const index_t tid_x = threadIdx.x;
    const index_t tid_y = threadIdx.y;
    const index_t bid_x = blockIdx.x;
    const index_t bid_y = blockIdx.y;

    // Global position
    const index_t global_x = bid_x * BlockSize + tid_x;
    const index_t global_y = bid_y * BlockSize + tid_y;

    // Load to shared memory with coalescing
    if (global_x < W && global_y < H) {
        smem[tid_y * BlockSize + tid_x] = input[global_y * W + global_x];
    }
    __syncthreads();

    // Process tiles with Morton ordering
    constexpr index_t TilesPerBlock = BlockSize / TileSize;

    // Each thread processes one element in Morton order
    const index_t tile_id = (tid_y / TileSize) * TilesPerBlock + (tid_x / TileSize);
    const index_t morton_in_tile = (tid_y % TileSize) * TileSize + (tid_x % TileSize);

    // Decode Morton index
    index_t y_in_tile, x_in_tile;
    morton_decode_2d<2>(morton_in_tile, y_in_tile, x_in_tile);

    // Calculate position in shared memory
    const index_t tile_y = tile_id / TilesPerBlock;
    const index_t tile_x = tile_id % TilesPerBlock;
    const index_t smem_y = tile_y * TileSize + y_in_tile;
    const index_t smem_x = tile_x * TileSize + x_in_tile;

    // Process with Morton access pattern
    DataType value = smem[smem_y * BlockSize + smem_x];

    // Apply computation...
    value = compute_function(value);

    // Store result
    if (global_x < W && global_y < H) {
        output[global_y * W + global_x] = value;
    }
}
```

6.18.7 Bank Conflict Reduction

Morton ordering is particularly effective for reducing shared memory bank conflicts (complementing the XOR preshuffle technique described in *Load Data Share Index Swapping*):

```
template<index_t WarpSize = 32>
struct BankConflictAnalysis {
    static constexpr index_t NumBanks = 32;
    static constexpr index_t BankWidth = 4;  // bytes

    template<typename AccessPattern>
    __host__ static void analyze_bank_conflicts(
        const char* pattern_name,
        AccessPattern access_func)
    {
        index_t bank_access[NumBanks] = {0};

        // Simulate warp access
        for (index_t tid = 0; tid < WarpSize; ++tid) {
            index_t offset = access_func(tid);
            index_t bank = (offset * sizeof(float) / BankWidth) % NumBanks;
            bank_access[bank]++;
        }

        // Find maximum conflict
        index_t max_conflict = 0;
        for (index_t bank = 0; bank < NumBanks; ++bank) {
            max_conflict = max(max_conflict, bank_access[bank]);
        }

        printf("%s: %d-way bank conflict\n", pattern_name, max_conflict);
    }

    __host__ static void compare_access_patterns() {
        printf("Bank Conflict Analysis for 4x4 Tile Access:\n");

        // Row-major access
        analyze_bank_conflicts("Row-major", [](index_t tid) {
            return (tid / 4) * 4 + (tid % 4);
        });

        // Morton access
        analyze_bank_conflicts("Morton", [](index_t tid) {
            index_t y, x;
            morton_decode_2d<2>(tid % 16, y, x);
            return y * 4 + x;
        });
    }
};
```

6.18.8 Practical Applications

Real-world usage of Morton ordering in CK Tile:

1. Texture Cache Optimization

```
template<typename DataType>
struct TextureCacheOptimized {
```

(continues on next page)

(continued from previous page)

```
    static constexpr index_t TextureTileSize = 8;

    __device__ static DataType sample_2d_morton(
        const DataType* texture,
        float u, float v,
        index_t width, index_t height)
    {
        // Convert normalized coordinates to texel coordinates
        index_t x = u * width;
        index_t y = v * height;

        // Determine tile
        index_t tile_x = x / TextureTileSize;
        index_t tile_y = y / TextureTileSize;

        // Position within tile
        index_t x_in_tile = x % TextureTileSize;
        index_t y_in_tile = y % TextureTileSize;

        // Convert to Morton index
        index_t morton_idx = morton_encode_2d<3>(y_in_tile, x_in_tile);

        // Calculate final offset
        index_t tile_offset = (tile_y * (width / TextureTileSize) + tile_x)
                              * TextureTileSize * TextureTileSize;

        return texture[tile_offset + morton_idx];
    }
};
```

2. Matrix Multiplication with Swizzled Tiles

For complete GEMM optimization techniques, see *A Block GEMM on MI300*.

```
template<typename DataType, index_t TileM, index_t TileN, index_t TileK>
struct SwizzledGEMM {
    __device__ static void load_tile_morton(
        const DataType* matrix,
        DataType* tile,
        index_t row_offset,
        index_t col_offset,
        index_t ld)
    {
        // Load tile with Morton ordering for better LDS bank utilization
        #pragma unroll
        for (index_t i = 0; i < TileM * TileN; ++i) {
            index_t row_in_tile, col_in_tile;
            morton_decode_2d<3>(i, row_in_tile, col_in_tile);

            if (row_in_tile < TileM && col_in_tile < TileN) {
                index_t global_row = row_offset + row_in_tile;
                index_t global_col = col_offset + col_in_tile;
                tile[i] = matrix[global_row * ld + global_col];
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }
};
```

6.18.9 Summary

Morton ordering with CK Tile provides memory optimization capabilities:

- **Spatial Locality**: Z-order curve maintains 2D locality in 1D memory layout
- **Bank Conflict Reduction**: Distributed access patterns across memory banks
- **Cache Efficiency**: Better utilization of cache lines for 2D access patterns
- **Mathematical Framework**: Tensor descriptors express swizzling cleanly
- **Practical Implementation**: Bit manipulation provides reliable results

Key implementation insights:

1. **MergeTransform** is essential for expressing Morton bit interleaving
2. **Manual bit manipulation** provides reliable and efficient implementation
3. **Tiling + Morton** combines hierarchical locality with local optimization
4. **GPU-specific tuning** adapts patterns to hardware characteristics

The tensor descriptor approach provides the mathematical framework for expressing these complex memory patterns, while practical implementations often use direct bit manipulation for efficiency and reliability.

For more examples of practical CK Tile usage, see *Convolution Implementation with CK Tile*. For the underlying buffer and tensor abstractions, see *Buffer Views - Raw Memory Access* and *Tensor Views - Multi-Dimensional Structure*.

6.19 Tensor Coordinates

6.19.1 Overview

Before diving into transforms and adaptors (see *Individual Transform Operations* and *Tensor Adaptors - Chaining Transformations*), it's essential to understand the basic coordinate system in CK Tile. MultiIndex is a container that extends the C++ array with additional operations for multi-dimensional indexing. It is the fundamental building block used throughout the system.

MultiIndex serves as the common currency between different coordinate spaces (see *Coordinate Systems - The Mathematical Foundation*), enabling seamless transformation and navigation through complex tensor layouts. Every transform, adaptor, and descriptor in CK Tile operates on these coordinate containers.

6.19.2 MultiIndex Implementation

The C++ implementation provides both compile-time and runtime flexibility:

```
// Basic MultiIndex structure
template<index_t NDim>
struct MultiIndex {
    static constexpr index_t kNDim = NDim;

    // Storage for coordinate values
```

(continues on next page)

(continued from previous page)

```
    array<index_t, NDim> data_;

    // Constructors
    __host__ __device__ constexpr MultiIndex() : data_{} {}

    __host__ __device__ constexpr MultiIndex(
        const array<index_t, NDim>& values) : data_(values) {}

    // Element access
    __host__ __device__ constexpr index_t& operator[](index_t i) {
        return data_[i];
    }

    __host__ __device__ constexpr const index_t& operator[](index_t i) const {
        return data_[i];
    }

    // Size query
    __host__ __device__ static constexpr index_t size() {
        return NDim;
    }
};
```

6.19.3 Creating and Using MultiIndex

CK Tile provides convenient factory functions for creating MultiIndex objects:

```
#include <ck_tile/core/container/multi_index.hpp>

__device__ void example_multiindex_usage() {
    // Create 3D coordinate with runtime values
    auto coord = make_multi_index(1, 2, 3);

    // Access dimensions
    auto x = coord[0];  // Returns 1
    auto y = coord[1];  // Returns 2
    auto z = coord[2];  // Returns 3

    // For compile-time coordinates, use number<>
    auto coord_static = make_multi_index(
        number<1>{}, number<2>{}, number<3>{}
    );

    // Create from tuple
    auto shape = make_tuple(128, 256, 64);
    auto coord2 = to_multi_index(shape);

    // Modify coordinate
    auto new_coord = coord;
    new_coord[0] = 5;  // Set X to 5

    // Use in tensor access
```

(continues on next page)

(continued from previous page)

```
    auto tensor = make_naive_tensor_view<address_space_enum::global>(
        data_ptr, shape, strides
    );

    // Create tensor coordinate for access
    auto tensor_coord = make_tensor_coordinate(
        tensor.get_tensor_descriptor(), coord
    );
}
```

For more advanced coordinate operations and movement patterns, see *Advanced Coordinate Movement*.

Compile-Time Optimization

CK Tile leverages C++ templates for zero-overhead abstractions:

```
// Compile-time MultiIndex operations
template<index_t... Is>
__host__ __device__ constexpr auto make_static_multi_index() {
    return MultiIndex<sizeof...(Is)>{array{Is...}};
}

// Example: Matrix access pattern
template<index_t M, index_t N>
__device__ void optimized_matrix_access(float* matrix) {
    // Compile-time coordinates
    constexpr auto origin = make_static_multi_index<0, 0>();
    constexpr auto corner = make_static_multi_index<M-1, N-1>();

    // Loop unrolling with compile-time indices
    #pragma unroll
    for (index_t i = 0; i < M; ++i) {
        #pragma unroll
        for (index_t j = 0; j < N; ++j) {
            auto coord = make_multi_index(i, j);
            // Compiler can optimize based on known bounds
            process_element(matrix[i * N + j]);
        }
    }
}
```

6.19.4 MultiIndex in Coordinate Flow

MultiIndex serves as the interface between user code and the transformation pipeline:

6.19.5 Common Usage Patterns

Pattern 1: Tensor Iteration

```
template<typename DataType, index_t M, index_t N>
__device__ void iterate_2d_tensor(DataType* tensor) {
    // Iterate through tensor using MultiIndex
```

(continues on next page)

(continued from previous page)

```
    for (index_t i = 0; i < M; ++i) {
        for (index_t j = 0; j < N; ++j) {
            auto coord = make_multi_index(i, j);

            // Use coordinate for structured access
            DataType& element = tensor[coord[0] * N + coord[1]];

            // Process element
            element = process_value(element);
        }
    }
}
```

Pattern 2: Boundary Checking

```
template<index_t NDim>
__device__ bool is_valid_coordinate(
    const MultiIndex<NDim>& coord,
    const MultiIndex<NDim>& shape)
{
    for (index_t i = 0; i < NDim; ++i) {
        if (coord[i] < 0 || coord[i] >= shape[i]) {
            return false;
        }
    }
    return true;
}

// Usage in kernel
__global__ void safe_tensor_kernel(float* tensor, index_t H, index_t W) {
    auto coord = make_multi_index(
        blockIdx.y * blockDim.y + threadIdx.y,
        blockIdx.x * blockDim.x + threadIdx.x
    );

    auto shape = make_multi_index(H, W);

    if (is_valid_coordinate(coord, shape)) {
        tensor[coord[0] * W + coord[1]] = compute_value(coord);
    }
}
```

Pattern 3: Transform Chaining

```
// Apply multiple transformations to coordinates
template<typename Transform1, typename Transform2>
__device__ auto apply_transform_chain(
    const MultiIndex<2>& input_coord,
    const Transform1& t1,
    const Transform2& t2)
{
```

(continues on next page)

(continued from previous page)

```
    // First transformation
    auto intermediate = t1.calculate_bottom_index(input_coord);

    // Second transformation
    auto final = t2.calculate_bottom_index(intermediate);

    return final;
}
```

6.19.6 Advanced MultiIndex Operations

Arithmetic Operations

```
template<index_t NDim>
struct MultiIndexOps {
    // Element-wise addition
    __device__ static MultiIndex<NDim> add(
        const MultiIndex<NDim>& a,
        const MultiIndex<NDim>& b)
    {
        MultiIndex<NDim> result;
        #pragma unroll
        for (index_t i = 0; i < NDim; ++i) {
            result[i] = a[i] + b[i];
        }
        return result;
    }

    // Scalar multiplication
    __device__ static MultiIndex<NDim> scale(
        const MultiIndex<NDim>& coord,
        index_t factor)
    {
        MultiIndex<NDim> result;
        #pragma unroll
        for (index_t i = 0; i < NDim; ++i) {
            result[i] = coord[i] * factor;
        }
        return result;
    }

    // Dot product (for linear indexing)
    __device__ static index_t dot(
        const MultiIndex<NDim>& coord,
        const MultiIndex<NDim>& strides)
    {
        index_t result = 0;
        #pragma unroll
        for (index_t i = 0; i < NDim; ++i) {
            result += coord[i] * strides[i];
        }
        return result;
```

(continues on next page)

(continued from previous page)

```
    }
};
```

Specialized Coordinates

```
// Thread coordinate helper
struct ThreadCoordinate {
    __device__ static auto get_thread_coord_1d() {
        return make_multi_index(
            blockIdx.x * blockDim.x + threadIdx.x
        );
    }

    __device__ static auto get_thread_coord_2d() {
        return make_multi_index(
            blockIdx.y * blockDim.y + threadIdx.y,
            blockIdx.x * blockDim.x + threadIdx.x
        );
    }

    __device__ static auto get_thread_coord_3d() {
        return make_multi_index(
            blockIdx.z * blockDim.z + threadIdx.z,
            blockIdx.y * blockDim.y + threadIdx.y,
            blockIdx.x * blockDim.x + threadIdx.x
        );
    }
};
```

6.19.7 Integration with Tensor Operations

MultiIndex is the foundation for all tensor operations in CK Tile (see *Tensor Views - Multi-Dimensional Structure* and *Buffer Views - Raw Memory Access* for tensor abstractions):

```
template<typename TensorView>
__device__ void tensor_operation_example(TensorView& tensor) {
    // Get tensor shape as MultiIndex
    auto shape = tensor.get_tensor_descriptor().get_lengths();

    // Create coordinate for center element
    MultiIndex<TensorView::kNDim> center;
    #pragma unroll
    for (index_t i = 0; i < TensorView::kNDim; ++i) {
        center[i] = shape[i] / 2;
    }

    // Access center element
    auto center_value = tensor(center);

    // Create stencil pattern using MultiIndex
    constexpr auto offsets = make_tuple(
```

(continues on next page)

(continued from previous page)

```
        make_multi_index(-1,  0),  // North
        make_multi_index( 1,  0),  // South
        make_multi_index( 0, -1),  // West
        make_multi_index( 0,  1)   // East
    );

    // Apply stencil
    auto sum = center_value;
    static_for<0, 4, 1>{}([&](auto i) {
        auto neighbor = MultiIndexOps<2>::add(center, get<i>(offsets));
        if (is_valid_coordinate(neighbor, shape)) {
            sum += tensor(neighbor);
        }
    });
}
```

6.19.8 Performance Considerations

MultiIndex is designed for zero-overhead abstraction (see *Intro to AMD CDNA Architecture* for GPU performance fundamentals):

1. **Compile-Time Resolution**: When dimensions are known at compile time, all operations are inlined
2. **Register Allocation**: Small fixed-size arrays typically stay in registers
3. **Vectorization**: Compiler can vectorize operations on MultiIndex arrays
4. **Memory Layout**: Contiguous storage enables efficient cache usage

```
// Performance-optimized coordinate operations
template<index_t NDim>
struct OptimizedCoordOps {
    // Fused multiply-add for linear indexing
    __device__ __forceinline__ static index_t
    compute_offset(const MultiIndex<NDim>& coord,
                   const MultiIndex<NDim>& strides)
    {
        index_t offset = 0;

        // Unroll for small dimensions
        if constexpr (NDim <= 4) {
            #pragma unroll
            for (index_t i = 0; i < NDim; ++i) {
                offset = __fma_rn(coord[i], strides[i], offset);
            }
        } else {
            // Partial unrolling for larger dimensions
            #pragma unroll 4
            for (index_t i = 0; i < NDim; ++i) {
                offset += coord[i] * strides[i];
            }
        }

        return offset;
```

(continues on next page)

(continued from previous page)

```
    }
};
```

6.19.9 Summary

MultiIndex is the foundation of CK Tile's coordinate system:

- **Simple Abstraction**: Container for N integers representing position
- **Universal Usage**: Every transform and adaptor operates on MultiIndex
- **Type-Safe**: Compile-time size and bounds checking in C++
- **Zero-Overhead**: Template metaprogramming ensures no runtime cost
- **Flexible**: Supports both compile-time and runtime coordinates

Understanding MultiIndex is crucial before moving to transforms and adaptors, as they all build upon this fundamental coordinate representation. MultiIndex is the common language that allows all CK Tile components to work together seamlessly.

For the complete picture of how MultiIndex fits into the CK Tile coordinate system, see *Coordinate Systems - The Mathematical Foundation*. For practical usage in tile distribution, see *Tile Distribution - The Core API*.

6.20 Sweep Tile

6.20.1 Overview

Sweep operations are the clean way to iterate over distributed data in CK Tile. They complete the tile distribution workflow by providing clean, efficient iteration patterns that automatically handle all the complex indexing details. Sweep operations are similar to `forEach()` operation. Sweep operations call a function for every data element.

Sweep operations use the "load once, use many times" pattern. Load X data once into registers, then sweep through Y positions while keeping X in fast memory. This maximizes data reuse and minimizes memory bandwidth requirements.

6.20.2 The Complete GPU Workflow

Sweep operations are the final piece of the distributed computing puzzle:

1. **TileDistribution**: "Here's how to divide work"
2. **TileWindow**: "Here's the data, loaded efficiently"
3. **Sweep Operations**: "Here's how to process every element"
4. **User code**: "Thanks! *does computation*"

Without sweep operations, manual nested loops and complex index calculations are required, increasing the risk of missing elements or double-processing. Sweep operations provide lambda-based iteration with automatic handling of all elements.

See *Coordinate Systems - The Mathematical Foundation* for more information about coordinate systems.

6.20.3 Basic Sweep Implementation

The fundamental sweep pattern in C++:

```
template<typename DistributedTensor, typename Func>
__device__ void sweep_tile(
    const DistributedTensor& tensor,
    Func&& func)
{
    // Get Y-space dimensions
    constexpr auto y_lengths = tensor.get_tile_distribution()
                                     .get_y_vector_lengths();

    // Generate nested loops at compile time
    static_for<0, y_lengths.size(), 1>{}([&](auto i) {
        sweep_tile_impl<i.value>(tensor, func, make_tuple());
    });
}

// Recursive implementation for arbitrary dimensions
template<index_t Dim, typename DistributedTensor, typename Func, typename... Indices>
__device__ void sweep_tile_impl(
    const DistributedTensor& tensor,
    Func&& func,
    tuple<Indices...> indices)
{
    constexpr auto y_lengths = tensor.get_tile_distribution()
                                     .get_y_vector_lengths();

    if constexpr (Dim == y_lengths.size()) {
        // Base case: call function with complete indices
        func(make_multi_index(indices...));
    } else {
        // Recursive case: iterate this dimension
        static_for<0, y_lengths[Dim], 1>{}([&](auto i) {
            sweep_tile_impl<Dim + 1>(
                tensor, func,
                tuple_cat(indices, make_tuple(i))
            );
        });
    }
}
```

6.20.4 Memory Efficiency Pattern

The sweep pattern provides significant memory efficiency benefits. This is particularly important for GPU architectures (see *Intro to AMD CDNA Architecture*) where memory bandwidth is often the limiting factor:

6.20.5 Practical Sweep Patterns

Pattern 1: Simple Element Processing

This pattern demonstrates the basic usage with *Static Distributed Tensor*:

```
template<typename DataType>
__device__ void simple_sweep_example(
    StaticDistributedTensor<DataType, Distribution>& input,
```

(continues on next page)

(continued from previous page)

```
    StaticDistributedTensor<DataType, Distribution>& output)
{
    // Process each element
    sweep_tile(input, [&](auto y_indices) {
        DataType value = input.get_element(y_indices);
        DataType result = compute_function(value);
        output.set_element(y_indices, result);
    });
}
```

Pattern 2: Accumulation

```
template<typename DataType, typename Distribution>
__device__ DataType sweep_accumulate(
    const StaticDistributedTensor<DataType, Distribution>& tensor)
{
    DataType sum = 0;

    sweep_tile(tensor, [&](auto y_indices) {
        sum += tensor.get_element(y_indices);
    });

    return sum;
}
```

Pattern 3: Conditional Processing

```
template<typename DataType, typename Distribution>
__device__ void conditional_sweep(
    StaticDistributedTensor<DataType, Distribution>& tensor,
    DataType threshold)
{
    sweep_tile(tensor, [&](auto y_indices) {
        DataType value = tensor.get_element(y_indices);
        if (value > threshold) {
            // Process only values above threshold
            tensor.set_element(y_indices, process_large_value(value));
        }
    });
}
```

6.20.6 GEMM Sweep Pattern

The sweep pattern is fundamental to high-performance matrix multiplication. See *A Block GEMM on MI300* for more information about GEMM optimization details.

```
template<typename ADataType, typename BDataType, typename CDataType,
         typename ADistribution, typename BDistribution, typename CDistribution>
__device__ void gemm_sweep_tile(
    const TileWindow<ADataType>& a_window,
    const TileWindow<BDataType>& b_window,
```

(continues on next page)

(continued from previous page)

```
    TileWindow<CDataType>& c_window)
{
    // Phase 1: Load A tile into registers (X dimension)
    auto a_tile = make_static_distributed_tensor<ADataType, ADistribution>();
    a_window.load(a_tile);  // Load once, reuse many times

    // Phase 2: Create C accumulator
    auto c_accumulator = make_static_distributed_tensor<CDataType, CDistribution>();

    // Initialize accumulator
    sweep_tile(c_accumulator, [&](auto y_indices) {
        c_accumulator.set_element(y_indices, 0);
    });

    // Phase 3: Sweep through B positions (Y dimension)
    constexpr index_t k_per_block = BDistribution::get_lengths()[1];

    for (index_t k = 0; k < k_per_block; ++k) {
        // Load current B slice
        auto b_slice = make_static_distributed_tensor<BDataType, BDistribution>();
        b_window.load_slice(b_slice, k);

        // Compute C += A * B for this slice
        sweep_tile(c_accumulator, [&](auto c_indices) {
            CDataType sum = c_accumulator.get_element(c_indices);

            // Inner product for this C element
            constexpr index_t inner_dim = ADistribution::get_lengths()[1];
            for (index_t i = 0; i < inner_dim; ++i) {
                auto a_indices = make_multi_index(c_indices[0], i);
                auto b_indices = make_multi_index(i, c_indices[1]);

                sum += a_tile.get_element(a_indices) *
                       b_slice.get_element(b_indices);
            }

            c_accumulator.set_element(c_indices, sum);
        });
    }

    // Phase 4: Store result
    c_window.store(c_accumulator);
}
```

6.20.7 Advanced Sweep Patterns

Multi-Dimensional Sweep

```
template<typename DataType, index_t D0, index_t D1, index_t D2>
__device__ void tensor_3d_sweep(
    StaticDistributedTensor<DataType, Distribution3D>& tensor)
{
```

(continues on next page)

(continued from previous page)

```
    // Sweep through 3D tensor with nested structure
    sweep_tile(tensor, [&](auto indices) {
        // indices is MultiIndex<3> with [d0, d1, d2]
        index_t d0 = indices[0];
        index_t d1 = indices[1];
        index_t d2 = indices[2];

        // Process based on 3D position
        DataType value = tensor.get_element(indices);

        // Example: Different processing for different planes
        if (d2 == 0) {
            // First plane: special processing
            value = special_process(value);
        } else {
            // Other planes: normal processing
            value = normal_process(value);
        }

        tensor.set_element(indices, value);
    });
}
```

Strided Sweep

```
template<typename DistributedTensor, typename Func, index_t Stride>
__device__ void strided_sweep(
    const DistributedTensor& tensor,
    Func&& func)
{
    constexpr auto y_lengths = tensor.get_tile_distribution()
                                     .get_y_vector_lengths();

    // Sweep with stride in first dimension
    static_for<0, y_lengths[0], Stride>{}([&](auto i) {
        // Create indices for this strided position
        auto indices = make_multi_index(i);

        // Complete remaining dimensions normally
        sweep_remaining_dims<1>(tensor, func, indices);
    });
}
```

Block Sweep for Cache Optimization

This pattern leverages shared memory to avoid *Understanding AMD GPU LDS and Bank Conflicts*:

```
template<typename DataType, typename Distribution, index_t BlockSize>
__device__ void block_sweep_pattern(
    StaticDistributedTensor<DataType, Distribution>& tensor)
{
    constexpr auto y_lengths = tensor.get_tile_distribution()
```

(continues on next page)

(continued from previous page)

```
                                    .get_y_vector_lengths();
    constexpr index_t num_blocks = (y_lengths[0] + BlockSize - 1) / BlockSize;

    // Process in blocks for better cache utilization
    static_for<0, num_blocks, 1>{}([&](auto block_id) {
        constexpr index_t block_start = block_id * BlockSize;
        constexpr index_t block_end = min(block_start + BlockSize, y_lengths[0]);

        // Load block data into shared memory
        __shared__ DataType block_cache[BlockSize][y_lengths[1]];

        // Cooperative load
        static_for<block_start, block_end, 1>{}([&](auto i) {
            static_for<0, y_lengths[1], 1>{}([&](auto j) {
                auto indices = make_multi_index(i, j);
                block_cache[i - block_start][j] = tensor.get_element(indices);
            });
        });

        __syncthreads();

        // Process from cache
        static_for<0, block_end - block_start, 1>{}([&](auto i) {
            static_for<0, y_lengths[1], 1>{}([&](auto j) {
                DataType value = block_cache[i][j];
                value = complex_process(value);

                auto indices = make_multi_index(block_start + i, j);
                tensor.set_element(indices, value);
            });
        });
    });
}
```

6.20.8 Performance Characteristics

Sweep operations provide several performance benefits:

Compiler Optimizations

Using *LoadStoreTraits - Memory Access Optimization Engine* and *Space-Filling Curves - Optimal Memory Traversal* enables optimal memory access patterns:

```
// The compiler can optimize sweep patterns effectively
template<typename DataType>
__device__ void optimized_sweep_example(
    StaticDistributedTensor<DataType, Distribution>& tensor)
{
    // This sweep pattern:
    sweep_tile(tensor, [&](auto indices) {
        tensor.set_element(indices, tensor.get_element(indices) * 2.0f);
    });
```

(continues on next page)

(continued from previous page)

```
    // Compiles to something like:
    // #pragma unroll
    // for (index_t i = 0; i < tensor.size(); ++i) {
    //     tensor[i] *= 2.0f;
    // }

    // With:
    // - Complete unrolling for small tensors
    // - Vectorized loads/stores
    // - No function call overhead
    // - Perfect instruction scheduling
}
```

6.20.9 Integration with CK Tile Components

Complete workflow example:

```
template<typename DataType>
__global__ void complete_tile_kernel(
    const DataType* input,
    DataType* output,
    index_t M, index_t N)
{
    // 1. Define distribution
    constexpr index_t BlockM = 64;
    constexpr index_t BlockN = 64;

    using Distribution = TileDistribution<
        Sequence<BlockM, BlockN>,
        Sequence<16, 16>
    >;

    // 2. Create tile windows
    auto input_window = make_tile_window(
        input, make_tuple(M, N),
        make_tuple(blockIdx.y * BlockM, blockIdx.x * BlockN),
        Distribution{}
    );

    auto output_window = make_tile_window(
        output, make_tuple(M, N),
        make_tuple(blockIdx.y * BlockM, blockIdx.x * BlockN),
        Distribution{}
    );

    // 3. Load input tile
    auto input_tile = make_static_distributed_tensor<DataType, Distribution>();
    input_window.load(input_tile);

    // 4. Create output tile
    auto output_tile = make_static_distributed_tensor<DataType, Distribution>();
```

(continues on next page)

(continued from previous page)

```
    // 5. Process with sweep
    sweep_tile(input_tile, [&](auto indices) {
        DataType value = input_tile.get_element(indices);
        DataType result = complex_computation(value);
        output_tile.set_element(indices, result);
    });

    // 6. Store results
    output_window.store(output_tile);
}
```

6.20.10 Summary

SweepTile provides clean and efficient iteration over distributed data:

- **Efficiency**: Load once, use many times pattern
- **Simplicity**: Clean lambda-based iteration abstraction
- **Performance**: Zero overhead with perfect access patterns
- **Flexibility**: Various sweep patterns for different algorithms

Key benefits:

1. **Memory Bandwidth**: Optimal reuse of loaded data
2. **Register Pressure**: Keep hot data in fastest memory
3. **Code Clarity**: Express algorithms naturally
4. **Compiler Optimization**: Enable aggressive optimizations

The sweep pattern is fundamental to high-performance GPU kernels, turning complex iteration patterns into simple, efficient operations. Combined with TileDistribution and TileWindow, sweep operations complete the toolkit for clean and performant GPU computing.

6.21 Encoding Internals

6.21.1 Overview

The tile distribution encoding system represents the core mathematical framework that transforms high-level tensor distribution specifications into concrete, optimized GPU kernel implementations. This advanced compile-time machinery bridges the gap between abstract mathematical descriptions and executable coordinate transformations, enabling the Composable Kernel framework to generate highly efficient code for complex tensor operations.

At its heart, the encoding system defines how multi-dimensional tensor data is distributed across GPU processing elements through a hierarchical decomposition scheme. By specifying relationships between different coordinate spaces of replication (R), hierarchical (H), partition (P), and yield (Y) dimension, the encoding provides a complete blueprint for data layout and access patterns that can be resolved entirely at compile time. This is the internal mechanism behind *Tile Distribution - The Core API*. See *Coordinate Systems - The Mathematical Foundation* for more information about coordinate spaces.

6.21.2 Encoding Structure

The tile distribution encoding employs a template-based type system that captures the complete specification of tensor distribution patterns at compile time:

```
template <typename RsLengths_,      // Replication dimension lengths
          typename HsLengthss_,     // Hierarchical dimension lengths
          typename Ps2RHssMajor_,   // P to RH mapping (major)
          typename Ps2RHssMinor_,   // P to RH mapping (minor)
          typename Ys2RHsMajor_,    // Y to RH mapping (major)
          typename Ys2RHsMinor_>    // Y to RH mapping (minor)
struct tile_distribution_encoding
{
    // All computations resolved at compile time
    static constexpr index_t NDimX = HsLengthss::size();
    static constexpr index_t NDimP = Ps2RHssMajor::size();
    static constexpr index_t NDimY = Ys2RHsMajor::size();
    static constexpr index_t NDimR = RsLengths::size();

    // Static member functions for compile-time access
    __host__ __device__ static constexpr auto get_rs_lengths() {
        return RsLengths_{};
    }

    __host__ __device__ static constexpr auto get_hs_lengthss() {
        return HsLengthss_{};
    }

    // Nested detail struct performs complex compile-time calculations
    struct detail {
        // Precomputed mappings and transformations
        static constexpr auto get_h_dim_lengths_prefix_sum();
        static constexpr auto get_uniformed_idx_y_to_h();
        // ... compile-time computation ...
    };
};
```

Key Template Features

1. **Template Metaprogramming**: All parameters are types, not values, enabling compile-time optimization
2. **Constexpr Functions**: Everything is computed at compile time
3. **Type Aliases**: Clean access to template parameters
4. **Static Member Functions**: No runtime overhead

6.21.3 Parameter Breakdown

R-Dimensions: Replication Specification

The `RsLengths` parameter defines dimensions that are replicated across processing units, enabling data sharing patterns essential for many tensor operations:

```
// Example: GEMM with warp-level replication
using RsLengths = Sequence<NWarpPerBlock, MWarpPerBlock>;
```

(continues on next page)

(continued from previous page)

```
// This creates replication pattern:
// - NWarpPerBlock warps share the same A data
// - MWarpPerBlock warps share the same B data
```

Replication serves several purposes:

- **Data Reuse**: Same input data needed by multiple output computations
- **Reduction Operations**: Multiple threads collaborate on single result
- **Memory Efficiency**: Reduces global memory bandwidth requirements

H-Dimensions: Hierarchical Decomposition

The `HsLengthss` parameter represents hierarchical decomposition of tensor dimensions:

```
// Example: Block-level GEMM decomposition
using HsLengthss = Tuple<
    Sequence<MRepeat, MWarp, MThread, MVec>,  // M-dimension
    Sequence<NRepeat, NWarp, NThread, NVec>   // N-dimension
>;

// This creates hierarchy:
// - MRepeat: iterations per thread in M
// - MWarp: warps assigned to M
// - MThread: threads per warp for M
// - MVec: vector size for M
```

The decomposition enables:

- **Memory Coalescing**: Aligning with warp/thread organization
- **Register Blocking**: Tile sizes that fit in register file
- **Shared Memory Utilization**: Tiles that exploit data reuse

P-Dimensions: Partition Mapping

The `Ps2RHssMajor` and `Ps2RHssMinor` parameters define work assignment:

```
// Example: 2D thread block mapping
// P0 = warp_id, P1 = lane_id
using Ps2RHssMajor = Tuple<
    Sequence<1>,  // P0 maps to H1 (warp dimension)
    Sequence<2>   // P1 maps to H2 (thread dimension)
>;
using Ps2RHssMinor = Tuple<
    Sequence<1>,  // Use second component of H1
    Sequence<2>   // Use third component of H2
>;
```

The mapping mechanism:

- **Major Index**: Which RH-dimension group (0=R, 1-N=H)
- **Minor Index**: Component within that group

Y-Dimensions: Logical View Mapping

The `Ys2RHsMajor` and `Ys2RHsMinor` define the user-facing interface:

```
// Example: 2D tile access pattern
using Ys2RHsMajor = Sequence<1, 1, 2, 2>;  // Y→H mapping
using Ys2RHsMinor = Sequence<0, 1, 0, 1>;  // Component selection

// Creates 2x2 logical view:
// Y[0,0] → H1[0], H2[0]
// Y[0,1] → H1[1], H2[0]
// Y[1,0] → H1[0], H2[1]
// Y[1,1] → H1[1], H2[1]
```

6.21.4 Transformation Pipeline

The encoding generates a transformation pipeline that converts coordinates using the concepts from *Individual Transform Operations* and *Tensor Adaptors - Chaining Transformations*:

Building the Transformation Chain

```
template <typename Encoding>
__host__ __device__ auto make_ps_ys_to_xs_adaptor(const Encoding& encoding)
{
    // Step 1: Create individual transforms
    constexpr auto replicate_transform = make_replicate_transform(
        encoding.get_rs_lengths());

    constexpr auto unmerge_transform = make_unmerge_transform(
        encoding.get_hs_lengthss());

    constexpr auto merge_transform = make_merge_transform(
        encoding.get_rhs_to_xs_mapping());

    // Step 2: Chain transforms together
    constexpr auto transform_chain = chain_transforms(
        replicate_transform,
        unmerge_transform,
        merge_transform);

    // Step 3: Create adaptor with the chain
    return make_tile_adaptor(
        transform_chain,
        encoding.get_lower_dimension_hidden_idss());
}
```

Transform Implementation Example

```
// Replicate transform implementation
template <typename Lengths>
struct replicate_transform
{
    static constexpr index_t num_of_upper_dimension = size(Lengths{});
```

(continues on next page)

(continued from previous page)

```
    static constexpr index_t num_of_lower_dimension = 2 * num_of_upper_dimension;

    template <typename UpperIndex>
    __host__ __device__ constexpr auto
    calculate_lower_index(const UpperIndex& idx_upper) const
    {
        // Replicate each coordinate: [a,b] -> [a,b,0,0]
        auto idx_lower = make_zero_multi_index<num_of_lower_dimension>();

        static_for<0, num_of_upper_dimension, 1>{}([&](auto i) {
            idx_lower(i) = idx_upper[i];
            idx_lower(i + num_of_upper_dimension) = 0;
        });

        return idx_lower;
    }
};
```

6.21.5 Y to D Linearization

The Y→D descriptor handles memory layout within each thread, building on *Tensor Descriptors - Complete Tensor Specifications* concepts:

```
template <typename YLengths, typename YStrides>
struct ys_to_d_descriptor
{
    static constexpr index_t num_of_dimension = size(YLengths{});

    // Calculate linear offset from Y coordinates
    template <typename YIndex>
    __host__ __device__ constexpr index_t
    calculate_offset(const YIndex& idx_y) const
    {
        index_t offset = 0;

        static_for<0, num_of_dimension, 1>{}([&](auto i) {
            offset += idx_y[i] * YStrides{}[i];
        });

        return offset;
    }

    // Get element space size (total elements per thread)
    __host__ __device__ static constexpr index_t
    get_element_space_size()
    {
        return reduce_on_sequence(
            YLengths{},
            multiplies{},
            number<1>{});
    }
};
```

Memory Layout Optimization

```
// Optimized layout for vector operations
template <index_t M, index_t N, index_t VectorSize>
struct make_ys_to_d_descriptor_for_gemm
{
    // Layout: [M/VectorSize][N][VectorSize]
    // This ensures vector loads are contiguous in memory
    using type = tile_descriptor<
        Sequence<M/VectorSize, N, VectorSize>,
        Sequence<N * VectorSize, VectorSize, 1>>;
};
```

Integration in Distributed Tensor

This shows how the encoding integrates with *Static Distributed Tensor*:

```
template <typename TileDistribution>
struct static_distributed_tensor
{
    using ys_to_d_descriptor = typename TileDistribution::ys_to_d_descriptor;

    // Thread-local storage
    static constexpr index_t thread_buffer_size =
        ys_to_d_descriptor::get_element_space_size();

    DataType thread_buffer_[thread_buffer_size];

    // Access element at Y coordinate
    template <typename YIndex>
    __host__ __device__ DataType& at(const YIndex& idx_y)
    {
        const index_t offset = ys_to_d_descriptor{}.calculate_offset(idx_y);
        return thread_buffer_[offset];
    }
};
```

6.21.6 Practical Examples

Example 1: Simple 2x2 Distribution

```
// No replication, simple hierarchy
using SimpleEncoding = tile_distribution_encoding<
    Sequence<>,           // rs_lengths: no replication
    Tuple<                // hs_lengthss: 2x2 hierarchy
        Sequence<2>,
        Sequence<2>
    >,
    Tuple<Sequence<>, Sequence<>>,  // ps_to_rhss_major
    Tuple<Sequence<>, Sequence<>>,  // ps_to_rhss_minor
    Sequence<1, 2>,                 // ys_to_rhs_major
    Sequence<0, 0>                  // ys_to_rhs_minor
>;
```

Example 2: GEMM Distribution

```
// Complex GEMM distribution with replication
template<index_t MPerBlock, index_t NPerBlock, index_t KPerBlock,
         index_t MPerWarp, index_t NPerWarp,
         index_t MRepeat, index_t NRepeat>
using GemmBlockEncoding = tile_distribution_encoding<
    Sequence<>,  // No block-level replication
    Tuple<       // Hierarchical decomposition
        Sequence<MRepeat, MPerBlock/MPerWarp/MRepeat>,  // M
        Sequence<NRepeat, NPerBlock/NPerWarp/NRepeat>   // N
    >,
    Tuple<       // Warp assignment
        Sequence<1, 2>,  // [warp_m, warp_n]
        Sequence<>
    >,
    Tuple<
        Sequence<1, 0>,  // Major indices
        Sequence<>
    >,
    Sequence<1, 1, 2, 2>,  // Y mapping
    Sequence<0, 1, 0, 1>   // Y components
>;
```

6.21.7 Performance Implications

The encoding system is designed for maximum GPU performance. See *Intro to AMD CDNA Architecture* for hardware fundamentals.

Memory Access Patterns

- **Coalescing**: Hierarchical decomposition ensures adjacent threads access adjacent memory
- **Bank Conflicts**: Careful dimension ordering prevents shared memory conflicts. See *Understanding AMD GPU LDS and Bank Conflicts* for more information.
- **Vectorization**: Natural support for vector loads and stores. See *LoadStoreTraits - Memory Access Optimization Engine* for more information.

Register Efficiency

- **Optimal Allocation**: Y→D linearization minimizes register usage
- **Spill Avoidance**: Compile-time sizing prevents register spills
- **Reuse Patterns**: Encoding enables efficient register reuse

Compile-Time Optimization

```
// All encoding operations resolve at compile time
template<typename Encoding>
struct encoding_optimizer {
    // Compute all derived values at compile time
    static constexpr auto total_elements = /* computed */;
    static constexpr auto access_pattern = /* computed */;
    static constexpr auto memory_layout = /* computed */;
```

(continues on next page)

(continued from previous page)

```
    // Generate optimized code paths
    template<typename Func>
    __device__ void apply_optimized(Func&& f) {
        if constexpr (is_simple_pattern) {
            // Direct access path
        } else if constexpr (is_strided_pattern) {
            // Strided access path
        } else {
            // General access path
        }
    }
};
```

6.21.8 Summary

The tile distribution encoding system demonstrates compile-time computation:

- **Mathematical Foundation**: Complete specification through dimensional relationships
- **Zero Overhead**: All computations resolve at compile time
- **Composable Design**: Individual transforms compose into complex mappings
- **Hardware Alignment**: Natural mapping to GPU execution hierarchy
- **Performance Focus**: Every design decision optimizes for GPU efficiency

The encoding internals show how CK Tile achieves practical performance. By leveraging C++ template metaprogramming and careful architectural design, the framework generates code that rivals hand-optimized implementations while maintaining clarity and composability.

For practical examples of how the encoding system is used, see *Thread Mapping - Connecting to Hardware*. For coordinate operations that build on these encodings, see *Advanced Coordinate Movement*.

6.22 Thread Mapping - Connecting to Hardware

This section explains how threads get their unique IDs and how those map to specific data, and connecting mathematical abstractions to physical hardware.

Thread mapping is the bridge between the mathematical abstraction and the physical hardware that executes the code. Thread mapping works closely with *Tile Distribution - The Core API* to ensure optimal performance.

6.22.1 Thread Identification and Partition Indices

Before threads can process data, they need to know who they are and what work they're responsible for.

Hardware Thread Identification

In GPU hardware, threads are organized hierarchically:

```
// CUDA/HIP thread identification
__device__ void get_thread_coordinates()
{
    // Grid-level coordinates (which block)
    int block_x = blockIdx.x;
```

(continues on next page)

(continued from previous page)

```
    int block_y = blockIdx.y;
    int block_z = blockIdx.z;

    // Block-level coordinates (which thread in block)
    int thread_x = threadIdx.x;
    int thread_y = threadIdx.y;
    int thread_z = threadIdx.z;

    // Warp identification
    int warp_id = threadIdx.x / 32;  // 32 threads per warp
    int lane_id = threadIdx.x % 32;  // Position within warp

    // Global thread ID calculation
    int global_thread_id = blockIdx.x * blockDim.x + threadIdx.x;
}
```

C++ Thread Mapping in CK

Composable Kernel abstracts thread identification into partition indices, building on the *Coordinate Systems - The Mathematical Foundation* foundation:

```
// From tile_partition.hpp
template <typename ThreadLayout>
struct tile_partition
{
    CK_TILE_DEVICE static constexpr index_t get_thread_idx()
    {
        return threadIdx.x;
    }

    CK_TILE_DEVICE static constexpr index_t get_block_idx()
    {
        return blockIdx.x;
    }

    // Convert to multi-dimensional partition index
    template <index_t NumDim>
    CK_TILE_DEVICE static constexpr auto get_partition_index()
    {
        constexpr auto thread_layout = ThreadLayout{};

        // Convert linear thread ID to multi-dimensional index
        return thread_layout.template get_index<NumDim>(get_thread_idx());
    }
};
```

Thread Hierarchy Structure

The hardware organizes threads in a specific hierarchy. See *Intro to AMD CDNA Architecture* for hardware details.

Block Level: Groups of warps working together

- Warps per block defined by encoding, for example, 2×2 warps

- Shared memory and synchronization scope
- Block-level coordination possible

Warp Level: Groups of threads executing in lockstep

- Threads per warp defined by encoding, for example, 8×8 threads
- SIMD execution (all threads execute same instruction)
- Warp-level primitives (shuffle, vote, etc.)

Thread Level: Individual execution units

- Vector size per thread, for example, 4×4 elements
- Independent register space
- Vector operations on multiple elements

Thread ID Mapping

Each thread gets a unique ID that maps to its position in the hierarchy. For example, in an RMSNorm configuration:

- **Repeat (M, N)**: (4, 4) - Number of iterations
- **Warps per block (M, N)**: (2, 2) - 4 warps total
- **Threads per warp (M, N)**: (8, 8) - 64 threads per warp
- **Vector size (M, N)**: (4, 4) - 16 elements per thread

This gives us:

- **Threads per block**: 256 (4 warps × 64 threads/warp)
- **Elements per thread**: 16 (4×4 vector)
- **Total elements**: 4096 per block

6.22.2 Thread-to-Data Mapping

Once threads know their IDs, they need to map those IDs to specific data elements.

Data Distribution Pattern

The RMSNorm operation distributes tensor data across threads in a structured pattern:

Hierarchical Data Distribution:

- **Block Level**: Multiple iterations (repeat factor)
- **Warp Level**: Warps process different regions
- **Thread Level**: Threads within warp handle adjacent data
- **Vector Level**: Each thread processes multiple elements

Thread Work Assignment

Each thread is assigned a specific rectangular region of the tensor. For example:

- Thread in Warp[0,0] Thread[0,0] might process:
  - Data region (M): [0:4)
  - Data region (N): [0:4)

  - Total elements: 16

- Thread in Warp[0,0] Thread[0,1] might process:
  - Data region (M): [0:4)
  - Data region (N): [4:8)
  - Total elements: 16

This pattern ensures adjacent threads access adjacent memory for optimal coalescing. The *LoadStoreTraits - Memory Access Optimization Engine* system further optimizes these access patterns.

6.22.3 Thread Cooperation Patterns

Threads don't work in isolation. Threads cooperate at different levels to achieve optimal performance.

Warp-Level Cooperation

Threads within a warp execute in lockstep (SIMD):

- **Synchronization**: Automatic SIMD execution
- **Data sharing**: Warp shuffle instructions
- **Collective ops**: Warp-level reductions
- **Memory access**: Coalesced patterns

Block-Level Cooperation

Threads within a block can share data and synchronize:

- **Shared memory**: All threads in block can access (see *Understanding AMD GPU LDS and Bank Conflicts* for optimization)
- **Synchronization**: `__syncthreads()` barriers
- **Data exchange**: Through shared memory
- **Collective operations**: Block-wide reductions

Vector-Level Processing

Each thread processes multiple elements:

- **Register efficiency**: Multiple elements in registers
- **Memory coalescing**: Vectorized loads/stores
- **Instruction efficiency**: SIMD operations on vectors
- **Bandwidth utilization**: Maximum memory throughput

6.22.4 Memory Access Patterns

The thread mapping directly affects memory access.

C++ Implementation of Memory Access

Here's how CK implements memory access patterns:

```
// Coalesced memory access pattern
template <typename DataType, index_t VectorSize>
__device__ void coalesced_load(const DataType* __restrict__ src,
                               DataType* __restrict__ dst,
                               index_t tid)
{
    // Each thread loads VectorSize elements
    // Adjacent threads access adjacent memory
    constexpr index_t stride = blockDim.x;

    // Vectorized load for efficiency
    using vector_t = vector_type_t<DataType, VectorSize>;

    // Calculate aligned address
    const vector_t* src_vec = reinterpret_cast<const vector_t*>(
        src + tid * VectorSize);

    // Single vectorized load instruction
    vector_t data = *src_vec;

    // Store to registers
    reinterpret_cast<vector_t*>(dst)[0] = data;
}

// CK's distributed tensor load implementation
template <typename DistributedTensor>
__device__ void load_tile_window(DistributedTensor& dist_tensor,
                                 const auto& tile_window)
{
    // Get thread's partition index
    constexpr auto partition = tile_partition::get_partition_index();

    // Each thread loads its assigned data
    tile_window.load(dist_tensor, partition);

    // Hardware automatically coalesces adjacent thread accesses
}
```

Memory Access Optimization Techniques

CK uses several techniques to optimize memory access:

```
// 1. Vector loads for maximum bandwidth
template <index_t N>
using vector_load_t = conditional_t<N == 1, float,
                      conditional_t<N == 2, float2,
                      conditional_t<N == 4, float4,
                                            float>>>;

// 2. Swizzling to avoid bank conflicts
template <index_t BankSize = 32>
__device__ index_t swizzle_offset(index_t tid, index_t offset)
{
```

(continues on next page)

(continued from previous page)

```
    // Rotate access pattern to avoid conflicts
    return (offset + (tid / BankSize)) % BankSize;
}

// 3. Prefetching for latency hiding
__device__ void prefetch_next_tile(const float* src, index_t offset)
{
    // Prefetch to L2 cache
    __builtin_prefetch(src + offset, 0, 3);
}
```

Memory Efficiency Benefits

The structured thread mapping provides several memory efficiency benefits:

Memory Coalescing Benefits:

- **Adjacent access**: Threads in same warp access adjacent memory locations
- **Cache efficiency**: Related data loaded together into cache lines
- **Bandwidth utilization**: Maximum memory bandwidth achieved
- **Reduced latency**: Fewer memory transactions needed

Performance Characteristics:

- **Predictable patterns**: Access patterns known at compile time
- **Vectorization**: Hardware can optimize vector operations
- **Reduced overhead**: No complex address calculations at runtime
- **Scalability**: Pattern scales efficiently with thread count

6.22.5 Practical Thread Mapping Example

Complete C++ Kernel Example

The following example shows how thread mapping works in a CK kernel:

```
// RMSNorm kernel using CK's thread mapping
template <typename DataType,
          typename ComputeType,
          index_t BlockSize,
          index_t VectorSize>
__global__ void rmsnorm_kernel(const DataType* __restrict__ x,
                               DataType* __restrict__ y,
                               const DataType* __restrict__ weight,
                               ComputeType epsilon,
                               index_t hidden_size)
{
    // 1. Thread identification
    const index_t tid = threadIdx.x;
    const index_t bid = blockIdx.x;

    // 2. Create tile distribution encoding
```

(continues on next page)

(continued from previous page)

```
    // This would be defined based on your specific RMSNorm pattern
    using Encoding = tile_distribution_encoding<
        sequence<>,                          // No replication
        tuple<sequence<4, 2>, sequence<4, 2>>, // H dimensions
        tuple<sequence<1>, sequence<2>>,     // P to RH major
        tuple<sequence<0>, sequence<0>>,     // P to RH minor
        sequence<1, 2>,                      // Y to RH major
        sequence<0, 0>                       // Y to RH minor
    >;
    constexpr auto tile_dist = make_static_tile_distribution(Encoding{});

    // 3. Get thread's partition index from distribution
    const auto partition_idx = tile_dist._get_partition_index();

    // 4. Shared memory for reduction
    __shared__ ComputeType shared_sum[BlockSize];

    // 5. Create tensor view and tile window
    auto x_view = make_naive_tensor_view<address_space_enum::global>(
        x + bid * hidden_size,
        make_tuple(hidden_size),
        make_tuple(number<1>{})
    );

    auto x_window = make_tile_window(
        x_view,
        make_tuple(hidden_size),
        make_tuple(number<0>{}),
        tile_dist);

    // 6. Each thread processes its assigned elements
    ComputeType thread_sum = 0;
    static_for<0, VectorSize, 1>{}([&](auto i) {
        // Access pattern would depend on your tile window setup
        // This is conceptual - actual implementation varies
        thread_sum += val * val;
    });

    // 7. Warp-level reduction
    thread_sum = warp_reduce_sum<WarpSize>(thread_sum);

    // 8. Block-level reduction
    if (tid % WarpSize == 0) {
        shared_sum[tid / WarpSize] = thread_sum;
    }
    __syncthreads();

    // 9. Final reduction by first warp
    if (tid < BlockSize / WarpSize) {
        thread_sum = shared_sum[tid];
        thread_sum = warp_reduce_sum<BlockSize / WarpSize>(thread_sum);
    }
```

(continues on next page)

(continued from previous page)

```
    // 10. Compute RMS and normalize
    if (tid == 0) {
        shared_sum[0] = rsqrt(thread_sum / hidden_size + epsilon);
    }
    __syncthreads();

    const ComputeType rms_recip = shared_sum[0];

    // 11. Write normalized output
    auto y_window = make_tile_window(
        make_tensor_view<address_space_enum::global>(y + bid * hidden_size),
        tile_dist);

    static_for<0, VectorSize, 1>{}([&](auto i) {
        auto idx = tile_dist.get_tensor_coordinate(partition_idx, i);
        ComputeType val = static_cast<ComputeType>(x_window.get(idx));
        ComputeType w = static_cast<ComputeType>(weight[idx[1]]);
        y_window.set(idx, static_cast<DataType>(val * rms_recip * w));
    });
}
```

Key Thread Mapping Concepts in Action

1. **Thread-to-Data Assignment**: Each thread gets a unique `partition_idx`
2. **Vectorized Access**: Each thread processes `VectorSize` elements
3. **Warp Cooperation**: Threads within a warp perform reductions
4. **Block Synchronization**: All threads synchronize for final result
5. **Coalesced Memory**: Adjacent threads access adjacent memory

6.22.6 Key Takeaways

Thread mapping is the bridge between mathematical abstractions and physical hardware execution:

Thread Identification:

1. **Hierarchical Organization**: Threads organized in blocks → warps → threads → vectors
   - Each level has specific cooperation capabilities
   - Hardware provides efficient primitives at each level
   - Thread IDs map directly to data regions
   - Predictable and efficient execution patterns
2. **Data Assignment**: Each thread gets a specific rectangular region
   - Work distributed evenly across threads
   - Memory access patterns optimized for coalescing
   - Vector operations maximize throughput
   - Scalable across different hardware configurations
3. **Cooperation Patterns**: Threads cooperate at multiple levels

- Warp-level SIMD execution for efficiency
- Block-level shared memory and synchronization
- Vector-level processing for maximum throughput
- Hierarchical coordination for complex operations

Performance Benefits:

- **Memory Coalescing**: Adjacent threads access adjacent memory for optimal bandwidth
- **Cache Efficiency**: Related data loaded together, reducing memory latency
- **Vectorization**: Hardware can optimize multiple operations per thread
- **Predictable Patterns**: Compile-time optimization of access patterns

Why This Matters:

Thread mapping connects encodings, transformations, and distributions to hardware execution.

The RMSNorm example shows how a real operation uses these concepts to achieve optimal performance on GPU hardware. Every thread knows exactly what data to process, how to access it efficiently, and how to cooperate with other threads.

Related Topics

- *Tensor Descriptors - Complete Tensor Specifications* - Complete tensor specifications that thread mapping uses
- *Advanced Coordinate Movement* - Advanced coordinate operations for thread navigation
- *Sweep Tile* - How threads iterate over distributed data
- *A Block GEMM on MI300* - Real-world application of thread mapping in GEMM kernels
- *Space-Filling Curves - Optimal Memory Traversal* - Optimal traversal patterns for thread access

6.23 CK Tile Hardware Documentation

This section provides in-depth coverage of hardware-specific concepts and optimizations for CK Tile on AMD GPUs.

6.23.1 Overview

Understanding the underlying hardware architecture is crucial for achieving optimal performance with CK Tile. This documentation covers:

- AMD CDNA architecture fundamentals
- Memory hierarchy and optimization techniques
- Practical examples of high-performance kernels

6.23.2 Documentation Structure

Intro to AMD CDNA Architecture

The AMD CDNA architecture is a specialized GPU design for high-performance computing (HPC) and AI workloads. Unlike the RDNA architecture used in gaming GPUs, CDNA is optimized for data center tasks, prioritizing compute density, memory bandwidth, and scalability. This is achieved through several key architectural features.

For more information about the AMD GPU architecture, see the GPU architecture documentation.

Implications for CK Tile

Understanding the CDNA architecture is crucial for effective use of CK Tile:

1. **Thread Organization**: CK Tile's hierarchical *Thread Mapping - Connecting to Hardware* (blocks → warps → threads) directly maps to CDNA's hardware organization.
2. **Memory Hierarchy**: CK Tile's *Buffer Views - Raw Memory Access* and *Tile Window - Data Access Gateway* are designed to efficiently utilize the L2, Infinity Cache, and LDS hierarchy.
3. **Register Pressure**: CK Tile's compile-time optimizations help minimize VGPR usage, preventing spills to slower memory.
4. **Warp Execution**: CK Tile's *Tile Distribution - The Core API* ensures that threads within a warp access contiguous memory for optimal SIMD execution.
5. **LDS Utilization**: CK Tile's *Static Distributed Tensor* and *Tile Window - Data Access Gateway* make effective use of the 64KB LDS per CU.

By understanding these architectural features, developers can better appreciate how CK Tile's abstractions map to hardware capabilities and why certain design decisions were made in the framework.

Related Topics

- *Thread Mapping - Connecting to Hardware* - How threads are organized and mapped to hardware
- *Coordinate Systems - The Mathematical Foundation* - Mathematical foundation for data distribution
- *Understanding AMD GPU LDS and Bank Conflicts* - Optimizing shared memory access patterns
- *LoadStoreTraits - Memory Access Optimization Engine* - Memory access optimization strategies
- *A Block GEMM on MI300* - Practical application of architecture knowledge

Understanding AMD GPU LDS and Bank Conflicts

Introduction

Local Data Share (**LDS**) is AMD's shared memory within a compute unit (see *Intro to AMD CDNA Architecture* for architecture details). It is organized into **32 or 64 banks** depending on the hardware architecture, each bank has a 4 bytes width. Understanding how memory addresses map to banks is key to avoiding **bank conflicts**.

Bank Mapping

For AMD GCN architecture, the LDS bank mapping is typically:

$$\text{bank} = \left(\frac{\text{address in bytes}}{4} \right) \bmod 32$$

This means:

- Addresses that differ by multiples of `bank numbers * 4 bytes` map to the same bank.
- Conflicts occur when multiple threads in the same wave access the same bank **in the same cycle**.

Not all the lanes can produce bank conflicts. HW divides access to LDS from wavefront into phases. Which lanes would be considered in each phase depends on the width of the instruction. Let us consider `ds_write_b128` as an example as it is the instruction that has the largest granularity write with the highest performance. Here access will be divided into 8 phases for 64 lane wavefront. If in 1 phase there will not be two thread access the same bank, there will bot be bank conflict:

- lane0~lane7
- lane8~lane15

- lane16~lane23
- lane24~lane31
- lane32~lane39
- lane40~lane47
- lane48~lane55
- lane56~lane63

If within each group of lanes there is no conflict it is an LDS bank conflict free write access.

Bank Access Patterns

LDS bank access can be simulated for a given set of thread addresses. With a 32 bank LDS with 4 bytes per bank, each thread will be writing 8 2-byte elements (16 bytes total), consuming 4 banks in LDS. fp16 or bf16 are the common formats GPU kernels have to deal with. With the phase access pattern like above by default it is a bank conflict free LDS write access.

Write Access Pattern

For LDS write instructions like `ds_write_b128`, the hardware provides conflict-free access when threads write to consecutive addresses. Each phase of 8 lanes writes to different banks, avoiding conflicts.

Read Access Pattern

Similarly for LDS read instruction `ds_read_b128`, when there is no bank conflict in these 8 lane groups:

- 0:3+20:23
- 4:7+16:19
- 8:11+28:31
- 12:15+24:27
- 32:35+52:55
- 36:39+48:51
- 40:43+60:63
- 44:47+56:59

then it's bank conflict-free for LDS reading.

The reason for accessing the data vertically is because in most LDS access the MFMA instruction in the next step and the MFMA are requirde to access the data vertically like above.

The LDS read access pattern illustrated below is typical for LDS usage in machine learning workloads. The read pattern can generate 4-way bank conflicts in every phase of access. You can experiment with `row_padding` (padding in a number of banks) to see if the problem can be solved this way, but also remember that in practice this will require additional LDS storage. The bigger the padding, the more additional storage is necessary.

XOR Preshuffle: An Alternative to Padding

Another technique to reduce LDS bank conflicts is **XOR preshuffling** (see *Load Data Share Index Swapping* for detailed implementation). Instead of adding padding between rows, we can permute the column indices for each row using XOR. This method can help to avoid bank conflicts without allocating extra storage in LDS.

For a wavefront of 64 threads, if each thread writes a vector of 8 fp16 elements (16 bytes), and the row size is 64 elements, the column index for each element in a row is adjusted as follows:

- `KTypeSize = 2`
- `KPerBlock = 64` // 64 elements per row
- `KPack = 8` // 8 elements per thread

The adjusted column position for element `(x, y)` is:

$$x' = \left(y \bmod \frac{\text{KPerBlock}}{\text{KPack}} \right) \oplus x$$

where $\oplus$ is the bitwise XOR, and x, y are the original positions of a vector element with respect to the LDS banks.

C++ Implementation

Here's how CK implements XOR preshuffling:

```
// XOR-based column index adjustment
template <index_t KPerBlock, index_t KPack>
__device__ constexpr index_t xor_preshuffle(index_t row, index_t col)
{
    constexpr index_t num_cols = KPerBlock / KPack;
    return (row % num_cols) ^ col;
}

// LDS write with XOR preshuffle
template <typename DataType, index_t RowStride>
__device__ void lds_write_with_xor(DataType* lds_ptr,
                                   const DataType* src,
                                   index_t row,
                                   index_t col)
{
    // Apply XOR preshuffle to column index
    index_t col_xor = xor_preshuffle<64, 8>(row, col);

    // Write to LDS with adjusted column
    index_t offset = row * RowStride + col_xor * 8;

    // Vectorized write (assuming 128-bit write)
    *reinterpret_cast<float4*>(lds_ptr + offset) =
        *reinterpret_cast<const float4*>(src);
}

// LDS read with XOR preshuffle
template <typename DataType, index_t RowStride>
__device__ void lds_read_with_xor(DataType* dst,
                                  const DataType* lds_ptr,
                                  index_t row,
                                  index_t col)
{
    // Apply same XOR preshuffle for read
    index_t col_xor = xor_preshuffle<64, 8>(row, col);
```

(continues on next page)

(continued from previous page)

```
    // Read from LDS with adjusted column
    index_t offset = row * RowStride + col_xor * 8;

    // Vectorized read
    *reinterpret_cast<float4*>(dst) =
        *reinterpret_cast<const float4*>(lds_ptr + offset);
}
```

Integration with CK Tile

CK Tile handles LDS bank conflict avoidance through its abstractions:

1. **TileWindow** (*Tile Window - Data Access Gateway*): Automatically applies XOR preshuffling when loading/storing to LDS
2. **StaticDistributedTensor** (*Static Distributed Tensor*): Manages LDS allocation with proper alignment
3. **LoadStoreTraits** (*LoadStoreTraits - Memory Access Optimization Engine*): Selects optimal access patterns to minimize conflicts

Example usage in CK Tile:

```
// CK Tile automatically handles bank conflict avoidance
template <typename TileDistribution>
__device__ void gemm_kernel()
{
    // Create tile window with automatic XOR preshuffle
    auto a_window = make_tile_window(
        a_tensor_view,
        tile_size,
        origin,
        tile_distribution);

    // Load to LDS - XOR preshuffle applied automatically
    auto a_lds_tensor = make_static_distributed_tensor<
        element_type,
        decltype(tile_distribution)>();

    a_window.load(a_lds_tensor);

    // Subsequent reads from LDS are conflict-free
    sweep_tile(a_lds_tensor, [](auto idx, auto& val) {
        // Process data...
    });
}
```

Performance Impact

Proper LDS bank conflict avoidance can have significant performance impact:

- **4-way conflicts**: Can reduce effective LDS bandwidth by 75%
- **XOR preshuffle**: Restores full bandwidth with zero storage overhead
- **Padding**: Also effective but requires 12.5-25% more LDS storage

Best Practices

1. **Use CK Tile abstractions**: They automatically handle bank conflict avoidance
2. **Prefer XOR preshuffle**: No storage overhead compared to padding
3. **Verify with profiling**: Use rocprof to check for LDS bank conflicts
4. **Consider access patterns**: Design algorithms with bank-friendly patterns

By understanding LDS bank conflicts and using CK Tile's automatic conflict avoidance mechanisms, developers can achieve optimal shared memory performance without manual optimization.

Related Topics

- *Load Data Share Index Swapping* - Detailed XOR preshuffle implementation
- *Memory Swizzling with Morton Ordering* - Morton ordering for memory swizzling
- *Intro to AMD CDNA Architecture* - Understanding AMD GPU architecture
- *Tile Window - Data Access Gateway* - Automatic conflict avoidance in data access
- *Static Distributed Tensor* - LDS memory management
- *A Block GEMM on MI300* - Practical application in GEMM kernels
- *Individual Transform Operations* - Coordinate transformations for conflict avoidance

A Block GEMM on MI300

Introduction to GEMMs

This document illustrates key concepts of implementing a block GEMM (General Matrix Multiplication) kernel on AMD's MI300 GPU. GEMM is a fundamental building block for many machine learning workloads, including attention mechanisms and Mixture of Experts (MoE) models.

The problem addressed here is the standard matrix multiplication: $C = A \cdot B$, where matrix A has dimensions **M x K** and matrix B has dimensions **K x N**. The resulting matrix C will have dimensions **M x N**. For simplicity and a better memory access pattern, it will be assumed that matrix B is in a column-major format, which means its shape is logically represented as **N x K**.

Format and Dimensions

The first step in designing the kernel is to select the data format and dimensions.

Data Format: bf16

While `float32` is a common choice, its high precision is computationally expensive and can be unnecessary for model convergence. A more suitable alternative is a half-precision floating-point format. We will use **bfloat16 (bf16)**.

Bfloat16 is a 16-bit format that uses the same 8-bit exponent as `float32`. This allows it to have the same dynamic range, which is critical for avoiding overflow and underflow during training. The key difference is that `bf16` uses only 7 bits for the mantissa (versus 23 bits in `float32`), which makes it functionally equivalent to a simple right bit-shift of a 32-bit float: `(float32 >> 16)`.

Dimensions: M=4864, N=4096

To maximize hardware utilization, dimensions are used that utilize the GPU's resources well. For this example, **M = 4864** and **N = 4096** are used. The rationale behind these particular values will be explained later.

Input data

The input will be uniformly distributed random data on the interval [-1, 1]:

```
initializeMatrix(A.data(), M, K, -1.0, 1.0);
initializeMatrix(B.data(), N, K, -1.0, 1.0);
```

Simple Matmul

On the AMD **MI300** GPU series (see *Intro to AMD CDNA Architecture*), each Compute Unit (CU) contains **four SIMD units**. Each SIMD unit can execute a single **wavefront** of 64 threads in parallel. Since there are four wavefronts per CU, a CU can therefore sustain the execution of up to **256 concurrent threads**.

These 256 threads then can be logically grouped into a **thread block**, which is responsible for computing a **sub-block (tile)** of the output matrix `C`. A block of 256 threads can be arranged as a **16×16 thread block**, where each thread computes one element of a **16×16 tile** of the result matrix `C`. Multiple thread blocks are then organized into a **grid**, such that the collection of blocks covers the entire output matrix.

Consider a baseline matrix multiplication kernel where **each thread computes one output element** of `C`. The HIP launch configuration can be defined as:

```
dim3 blockSizeRef(16, 16);
dim3 gridSizeRef((N + blockSizeRef.x - 1) / blockSizeRef.x,
                 (M + blockSizeRef.y - 1) / blockSizeRef.y);

matrixMulHIP<<<gridSizeRef, blockSizeRef, 0, 0>>>(d_A, d_B, d_C);
```

And the GPU Kernel:

```
__global__ void matrixMulHIP(s_type * __restrict__ A,
                             s_type* __restrict__ B,
                             float* __restrict__ C)
{
    // Calculate global thread coordinates in output matrix C
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Boundary check for valid threads
    if (row < N && col < N) {
        float value = 0.0f;
        // Perform the dot product of row from A and column from B
        for (int k = 0; k < K; ++k) {
            value += A[row * K + k] * B[col * K + k];
        }
        // Store computed value in output matrix
        C[row * N + col] = value;
    }
}
```

This kernel has a very low compute throughput according to `rocprofv3` profiler output. It is stalling on global memory read transactions effectively starving the rest of the pipeline that needs that data to proceed.

Memory Bandwidth Analysis

In a naïve implementation of matrix multiplication, **pressure on global memory loads** quickly becomes the bottleneck. To understand why, it is necessary to look at how a single **16×16 block** of the destination matrix `C` is computed by one block of threads within a compute unit.

Each thread in the block is responsible for computing a single element of `C`. To do so, it loops over the `K` dimension and, in every iteration, fetches **two values** from global memory:

- one from a row of `A`
- one from a column of `B`

This means:

- Number of threads in a 16×16 block is 256.
- Each thread performs 2K global loads
- **Total global loads** = 256 × 2K = 512K
- **Total global stores** = 256 (one per output element in `C`)

To reuse each element of `A` and `B` perfectly (loading each only once), the unique data required would be:

- Unique `A` elements: 16 × K = 16K
- Unique `B` elements: 16 × K = 16K
- **Total unique loads** = 16K + 16K = 32K
- **Total stores** = 256
- **Naïve kernel**: 512K global loads + 256 stores
- **Ideal reuse**: 32K global loads + 256 stores

This illustrates a **16× difference in memory traffic** for the same computation on a small, 16x16 block.

What is Tiling?

Cooperative Loading with LDS

In the naïve implementation, threads within the same compute unit (CU) do not cooperate with each other at all. Each thread independently and greedily loads the row elements of `A` and the column elements of `B` that it needs in order to compute its corresponding value in `C`.

Each CU on the MI300 has **64 KB of Local Data Share (LDS)** (see *Understanding AMD GPU LDS and Bank Conflicts* for optimization techniques) that acts as a shared memory space accessible by all threads in that CU. This opens the possibility of **cooperative loading**.

Instead of having every thread repeatedly fetch its own data directly from global memory, threads can **collaboratively preload** a block of data into LDS. Once in LDS, this data can be reused by many threads, reducing redundant global memory fetches.

Entire rows or columns of `A` and `B` can't be preloaded into LDS, since they might be very large and LDS has a fixed capacity. The solution is to load **small blocks (tiles)** of data at a time. For example:

- Load a **16×16 tile** from `A` and `B` into LDS
- Allow all threads in the CU to reuse the data from that tile to compute their portion of the result

- Once done, move the tile window forward along the `K` dimension
- Repeat until the entire **16×16 output block** of `C` is computed

This technique of **tiling with cooperative loading** reduces global memory traffic and improves GPU efficiency by leveraging fast, on-chip LDS as in LDS has a better speed and reuse of the data.

Tiling Mathematics

How many elements of matrices A and B need to be loaded with the tiling approach?

For a thread block computing a `TILE_M` × `TILE_N` output tile with K-blocking:

- Elements of **A** loaded per block:

$$\text{A_loads} = \text{TILE_M} \cdot K$$

- Elements of **B** loaded per block:

$$\text{B_loads} = \text{TILE_N} \cdot K$$

- Total outputs produced per block:

$$\text{outputs} = \text{TILE_M} \cdot \text{TILE_N}$$

The **average loads per output element** (ignoring C traffic) are:

$$\text{loads per output} = \frac{\text{TILE_M} \cdot K + \text{TILE_N} \cdot K}{\text{TILE_M} \cdot \text{TILE_N}} = K \left(\frac{1}{\text{TILE_M}} + \frac{1}{\text{TILE_N}} \right)$$

To simplify the formula, consider a square tile of size T, to compute one value in C:

- Naïve (no tiling) = 2K loads per output.
- With tiling = 2K/T.
- **Reduction factor = T**.

Example: T=16

$$\text{loads per output} = \frac{2K}{16} = \frac{K}{8}$$

Compared to the naïve 2K, this gives a **16× reduction** in global memory traffic per output element.

LDS Usage and Tiling Efficiency

How much space in LDS would this tiling use? Matrices **A** and **B** store data in **bf16** format. For a small 16×16 tile:

- Each matrix contains 16 × 16 = 256 elements.
- At 2 bytes per element, each matrix occupies 256 × 2 = 512 bytes.
- Total for A and B: 512 × 2 = 1 KB.

There is much more space in LDS, so why not try a bigger tile size? 32 KB for each matrix can be used, which allows the tile size to be increased to **256×64**. With this tile size, each compute unit (CU) will output a **256×256 block in C**. With this approach, the number of global memory reads will be **256 times smaller per element in C** compared to a brute-force approach.

Variation of the GEMM in Inference

When implementing GEMM in inference, because B matrix is the weight which is static, the B matrix will be preshuffled to the warp GEMM MFMA shape to have a faster access for registers to do the MFMA operations. In this strategy there are the following optimizations:

- Shared Memory bypass of the B Matrix.
- Loop over the A Matrix stored in the shared memory and let B stays in the registers.
- Ping Pong buffering for the GEMM Pipeline

Utilization Considerations

This section explains why the input dimensions **M = 4864** and **N = 4096** are convenient choices.

The MI300 has **304 compute units (CUs)**. If a tile size of **256×64** is chosen, where the **K dimension** is iterated over, then the output grid size is:

```
M / 256 × N / 256 = 4864 / 256 × 4096 / 256 = 19 × 16 = 304
```

This matches the total number of compute units on the GPU. That means every CU can be fully occupied with one tile of work, and imbalance or underutilization is not as much of a concern.

Advanced Optimizations

Matrix Fused Multiply-Add

Because compute-to-memory-access ratio can be a bottleneck, optimizing for bandwidth only isn't enough.

GPUs offer dedicated **matrix (or tensor) cores** for multiplication tasks. These cores are specifically designed to accelerate matrix operations.

To take full advantage of these specialized cores, intrinsic instructions can be used. Intrinsic instructions are hardware-specific functions that allow for direct access to the matrix core pipelines. For this example, `__builtin_amdgcn_mfma_f32_16x16x16f16`, has a low latency of only 16 cycles, will be used.

16x16 matrices will be used as input, and 16x16 matrices will be used as output. These instructions work as *accumulate add*, what they effectively do is: `D = A*B + C`. This is useful in this example since results will be accumulated over multiple tiles over K dimension.

Optimizing Data Flow with Pipelining

To maximize performance, the flow for this kernel uses a **pipeline** or **double buffering** to keep the compute units continuously fed with data, reducing idle time. This pipeline consists of a series of stages that process data concurrently:

- **Stage 1: Global Memory to Registers:** The first stage involves pre-loading data directly from **global memory** into Vector General Purpose Registers (VGPR). This is the slowest part of the pipeline. Because of this, this operation is performed as early as possible.
- **Stage 2: Registers to LDS (Shared Memory):** As data is being loaded from global memory, the next stage of the pipeline moves the data from the VGPRs into **LDS (Local Data Share)**, or shared memory. This is an intermediate step that makes the data accessible to all threads within the workgroup at very low latency.
- **Stage 3: LDS to Registers:** With the data now in LDS, the data is transferred from LDS back into a different set of VGPR registers, which will serve as the direct input for the compute operations.
- **Stage 4: Computation with MFMA:** The Matrix-FMA (MFMA) intrinsic uses the data from the VGPRs to perform the actual matrix multiplication and accumulation.

By using this pipelined approach, the different stages of data movement and computation happen in parallel. While the current VGPRs are being consumed by the MFMA operation, the next set of data is already being moved from LDS to another set of VGPRs, and the next tile of data is being loaded from global memory into a third set of VGPRs. This overlapping of operations is key to keeping the GPU's compute units fully utilized.

CK Tile Implementation

Here's how CK Tile implements an optimized GEMM kernel:

```
template <typename ADataType,
          typename BDataType,
          typename CDataType,
          index_t BlockSize,
          index_t MPerBlock,
          index_t NPerBlock,
          index_t KPerBlock>
__global__ void ck_tile_gemm_kernel(const ADataType* __restrict__ a_global,
                                    const BDataType* __restrict__ b_global,
                                    CDataType* __restrict__ c_global,
                                    index_t M,
                                    index_t N,
                                    index_t K)
{
    // Define tile distribution encoding
    using Encoding = tile_distribution_encoding<
        sequence<>,                                 // No replication
        tuple<sequence<4, 2, 8, 4>,                // M dimension hierarchy
              sequence<4, 2, 8, 4>>,               // N dimension hierarchy
        tuple<sequence<1, 2>, sequence<1, 2>>,     // Thread mapping
        tuple<sequence<1, 1>, sequence<2, 2>>,     // Minor indices
        sequence<1, 1, 2, 2>,                      // Y-space mapping
        sequence<0, 3, 0, 3>                       // Y-space minor
    >;

    constexpr auto tile_dist = make_static_tile_distribution(Encoding{});

    // Create tensor views for global memory
    auto a_global_view = make_naive_tensor_view<address_space_enum::global>(
        a_global, make_tuple(M, K), make_tuple(K, 1));
    auto b_global_view = make_naive_tensor_view<address_space_enum::global>(
        b_global, make_tuple(N, K), make_tuple(K, 1));
    auto c_global_view = make_naive_tensor_view<address_space_enum::global>(
        c_global, make_tuple(M, N), make_tuple(N, 1));

    // Calculate block offset
    const index_t block_m_id = blockIdx.y;
    const index_t block_n_id = blockIdx.x;

    // Create tile windows for loading
    auto a_window = make_tile_window(
        a_global_view,
        make_tuple(number<MPerBlock>{}, number<KPerBlock>{}),
        make_tuple(block_m_id * MPerBlock, 0),
```

(continues on next page)

(continued from previous page)

```
        tile_dist);

    auto b_window = make_tile_window(
        b_global_view,
        make_tuple(number<NPerBlock>{}, number<KPerBlock>{}),
        make_tuple(block_n_id * NPerBlock, 0),
        tile_dist);

    // Allocate LDS storage
    auto a_lds = make_static_distributed_tensor<ADataType,
                                                decltype(tile_dist)>();
    auto b_lds = make_static_distributed_tensor<BDataType,
                                                decltype(tile_dist)>();

    // Initialize accumulator
    auto c_reg = make_static_distributed_tensor<CDataType,
                                                decltype(tile_dist)>();
    sweep_tile(c_reg, [](auto idx, auto& val) { val = 0; });

    // Main GEMM loop with pipelining
    constexpr index_t num_k_tiles = K / KPerBlock;

    // Preload first tile
    a_window.load(a_lds);
    b_window.load(b_lds);
    __syncthreads();

    // Pipeline loop
    for(index_t k_tile = 0; k_tile < num_k_tiles - 1; ++k_tile) {
        // Move windows for next iteration
        a_window.move_slice_window(make_tuple(0, KPerBlock));
        b_window.move_slice_window(make_tuple(0, KPerBlock));

        // Prefetch next tile while computing current
        auto a_lds_next = make_static_distributed_tensor<ADataType,
                                                         decltype(tile_dist)>();
        auto b_lds_next = make_static_distributed_tensor<BDataType,
                                                         decltype(tile_dist)>();

        a_window.load_async(a_lds_next);
        b_window.load_async(b_lds_next);

        // Compute with current tile
        gemm_tile(a_lds, b_lds, c_reg);

        // Wait for prefetch and swap buffers
        __syncthreads();
        a_lds = a_lds_next;
        b_lds = b_lds_next;
    }

    // Last tile computation
```

(continues on next page)

(continued from previous page)

```
    gemm_tile(a_lds, b_lds, c_reg);

    // Store result
    auto c_window = make_tile_window(
        c_global_view,
        make_tuple(number<MPerBlock>{}, number<NPerBlock>{}),
        make_tuple(block_m_id * MPerBlock, block_n_id * NPerBlock),
        tile_dist);

    c_window.store(c_reg);
}
```

Key Takeaways

1. **Tiling is essential**: Reduces memory traffic by orders of magnitude
2. **Use specialized hardware**: MFMA instructions provide massive speedup
3. **Pipeline operations**: Hide memory latency with computation
4. **CK Tile abstractions**: Automatically handle complex optimizations
5. **Hardware-aware dimensions**: Choose problem sizes that map well to CU count

By understanding these optimization techniques and using CK Tile's high-level abstractions, developers can improve performance onGPUs without manual low-level optimization.

Related Topics

- *Tile Distribution - The Core API* - Core distribution mechanism used in GEMM
- *Tile Window - Data Access Gateway* - Window-based data access patterns
- *Static Distributed Tensor* - LDS memory management for tiles
- *Understanding AMD GPU LDS and Bank Conflicts* - Avoiding bank conflicts in GEMM
- *Thread Mapping - Connecting to Hardware* - How threads map to GEMM computation
- *LoadStoreTraits - Memory Access Optimization Engine* - Optimized memory access patterns
- *Space-Filling Curves - Optimal Memory Traversal* - Advanced traversal patterns
- *Sweep Tile* - Iterating over distributed data
- *Intro to AMD CDNA Architecture* - Understanding the hardware
- *Coordinate Systems - The Mathematical Foundation* - Mathematical foundation

GPU Architecture Basics

Intro to AMD CDNA Architecture provides an introduction to AMD CDNA architecture.

LDS and Bank Conflicts

Understanding AMD GPU LDS and Bank Conflicts explains Local Data Share (LDS) optimization.

GEMM Optimization Case Study

A Block GEMM on MI300 demonstrates a complete optimization journey.

6.23.3 Key Hardware Considerations

Memory Hierarchy

1. **Global Memory**: High latency, high bandwidth
   - Optimize with coalesced access patterns
   - Use tile windows for automatic optimization
2. **L2/Infinity Cache**: Intermediate storage
   - Benefits from spatial and temporal locality
   - CK Tile's tiling naturally improves cache hit rates
3. **LDS**: Low latency, shared within CU
   - 64KB per CU, organized in 32 banks
   - CK Tile handles bank conflict avoidance
4. **Registers**: Lowest latency, per-thread storage
   - 512 VGPRs available per wavefront
   - CK Tile's compile-time optimization minimizes usage

Compute Resources

1. **Wavefront Execution**: 64 threads in lockstep
   - CK Tile ensures coalesced memory access
   - Automatic warp-level synchronization
2. **Matrix Units**: Specialized MFMA instructions
   - 16x16x16 operations in 16 cycles
   - CK Tile can leverage these automatically
3. **Occupancy**: Balancing threads vs resources
   - Register pressure affects occupancy
   - CK Tile helps through efficient register use

6.23.4 Performance Guidelines

To achieve optimal performance with CK Tile:

1. **Choose appropriate tile sizes**:
   - Match hardware capabilities (e.g., 256x256 for GEMM)
   - Consider LDS capacity and register pressure
2. **Align problem dimensions**:
   - Match CU count when possible (304 for MI300)
   - Use padding for non-aligned sizes

3. **Enable pipelining**:
    - Use double buffering for latency hiding
    - CK Tile supports async operations
4. **Profile and verify**:
    - Use rocprof to check for bottlenecks
    - Verify bank conflict avoidance
    - Monitor occupancy and register usage

6.23.5 Next Steps

- Review *Intro to AMD CDNA Architecture* for architecture fundamentals
- Study *Understanding AMD GPU LDS and Bank Conflicts* for shared memory optimization
- Explore *A Block GEMM on MI300* for a complete optimization example

For practical implementation, refer back to the main *CK Tile Conceptual Documentation* documentation to see how these hardware concepts integrate with CK Tile's abstractions.

CHAPTER

SEVEN

COMPOSABLE KERNEL EXAMPLES AND TESTS

After *building and installing Composable Kernel*, the examples and tests will be moved to `/opt/rocm/bin/`.

All tests have the prefix `test` and all examples have the prefix `example`.

Use `ctest` with no arguments to run all examples and tests, or use `ctest -R` to run a single test. For example:

```
ctest -R test_gemm_fp16
```

Examples can be run individually as well. For example:

```
./bin/example_gemm_xdl_fp16 1 1 1
```

For instructions on how to run individual examples and tests, see their README files in the `example` and `test` GitHub folders.

To run smoke tests, use `make smoke`.

To run regression tests, use `make regression`.

In general, tests that run for under thirty seconds are included in the smoke tests and tests that run for over thirty seconds are included in the regression tests.

CHAPTER

EIGHT

COMPOSABLE KERNEL PRECISION SUPPORT

This topic lists the data type support for the Composable Kernel library on AMD GPUs.

This page lists the data types supported by the library itself and does not indicate hardware support. A type listed here is only usable if the GPU architecture also supports it; otherwise it is unsupported. For data type support across the other ROCm libraries and by GPU architecture, see the Data types and precision support page.

8.1 Supported data types overview

The following table summarizes the input and output data types supported by Composable Kernel.

Icon	Definition
	Fully supported as both an input and output type.
	Partially supported as an input or output type.

Data types not listed in the table below are not supported.

Data type	Support
int8	
int32	
float4 (E2M1)	
float6 (E2M3)	
float6 (E3M2)	
float8 (E4M3)	
float8 (E5M2)	
float16	
bfloat16	
float32	
float64	

8.2 Related references

For details about the underlying C++ types used by Composable Kernel, see the following pages:

- *Supported scalar data types* — the C++ scalar types, their bit widths, and bit-layout descriptions.
- *Custom types* — the custom types Composable Kernel defines for specialized operations.

CHAPTER

NINE

COMPOSABLE KERNEL SUPPORTED SCALAR DATA TYPES

The Composable Kernel library provides support for the following scalar data types:

Type	Bit Width	Description
`double`	64-bit	Standard IEEE 754 double precision floating point
`float`	32-bit	Standard IEEE 754 single precision floating point
`int32_t`	32-bit	Standard signed 32-bit integer
`int8_t`	8-bit	Standard signed 8-bit integer
`uint8_t`	8-bit	Standard unsigned 8-bit integer
`bool`	1-bit	Boolean type
`ck::half_t`	16-bit	IEEE 754 half precision floating point with 5 exponent bits, 10 mantissa bits, and 1 sign bit
`ck::bhalf_t`	16-bit	Brain floating point with 8 exponent bits, 7 mantissa bits, and 1 sign bit
`ck::f8_t`	8-bit	8-bit floating point (E4M3 format) with 4 exponent bits, 3 mantissa bits, and 1 sign bit
`ck::bf8_t`	8-bit	8-bit brain floating point (E5M2 format) with 5 exponent bits, 2 mantissa bits, and 1 sign bit
`ck::f4_t`	4-bit	4-bit floating point format (E2M1 format) with 2 exponent bits, 1 mantissa bit, and 1 sign bit
`ck::f6_t`	6-bit	6-bit floating point format (E2M3 format) with 2 exponent bits, 3 mantissa bits, and 1 sign bit
`ck::bf6_t`	6-bit	6-bit brain floating point format (E3M2 format) with 3 exponent bits, 2 mantissa bits, and 1 sign bit

CHAPTER

TEN

COMPOSABLE KERNEL CUSTOM DATA TYPES

Composable Kernel supports the use of custom types that provide a way to implement specialized numerical formats.

To use custom types, a C++ type that implements the necessary operations for tensor computations needs to be created. These should include:

- Constructors and initialization methods
- Arithmetic operators if the type will be used in computational operations
- Any conversion functions needed to interface with other parts of an application

For example, to create a complex half-precision type:

```
struct complex_half_t
{
    half_t real;
    half_t img;
};

struct complex_half_t
{
    using type = half_t;
    type real;
    type img;

    complex_half_t() : real{type{}}, img{type{}} {}
    complex_half_t(type real_init, type img_init) : real{real_init}, img{img_init} {}
};
```

Custom types can be particularly useful for specialized applications such as complex number arithmetic, custom quantization schemes, or domain-specific number representations.

CHAPTER

ELEVEN

COMPOSABLE KERNEL VECTOR TEMPLATE UTILITIES

Composable Kernel includes template utilities for creating vector types with customizable widths. These template utilities also flatten nested vector types into a single, wider vector, preventing the creation of vectors of vectors.

Vectors composed of supported scalar and custom types can be created with the `ck::vector_type` template.

For example, `ck::vector_type<float, 4>` creates a vector composed of four floats and `ck::vector_type<ck::half_t, 8>` creates a vector composed of eight half-precision scalars.

For vector operations to be valid, the underlying types must be either a *supported scalar type* or *a custom type* that implements the required operations.

CHAPTER

TWELVE

COMPOSABLE KERNEL WRAPPER

The Composable Kernel library provides a lightweight wrapper to simplify the more complex operations.

Example:

```
const auto shape_4x2x4         = ck::make_tuple(4, ck::make_tuple(2, 4));
const auto strides_s2x1x8      = ck::make_tuple(2, ck::make_tuple(1, 8));
const auto layout = ck::wrapper::make_layout(shape_4x2x4, strides_s2x1x8);

std::array<ck::index_t, 32> data;
auto tensor = ck::wrapper::make_tensor<ck::wrapper::MemoryTypeEnum::Generic>(&data[0],␣
↪layout);

for(ck::index_t w = 0; w < size(tensor); w++) {
    tensor(w) = w;
}

// slice() == slice(0, -1) (whole dimension)
auto tensor_slice = tensor(ck::wrapper::slice(1, 3), ck::make_tuple(ck::wrapper::slice(),
↪ ck::wrapper::slice()));
std::cout << "dims:2,(2,4) strides:2,(1,8)" << std::endl;
for(ck::index_t h = 0; h < ck::wrapper::size<0>(tensor_slice); h++)
{
    for(ck::index_t w = 0; w < ck::wrapper::size<1>(tensor_slice); w++)
    {
        std::cout << tensor_slice(h, w) << " ";
    }
    std::cout << std::endl;
}
```

Output:

```
dims:2,(2,4) strides:2,(1,8)
1 5 9 13 17 21 25 29
2 6 10 14 18 22 26 30
```

Tutorials:

- GEMM tutorial

Advanced examples:

- Image to column
- Basic gemm

- Optimized gemm

CHAPTER

THIRTEEN

COMPOSABLE KERNEL GLOSSARY

Add+Multiply
See *fused add multiply*.

alignment
Alignment is a memory management strategy where data structures are stored at addresses that are multiples of a specific value.

arithmetic logic unit
The arithmetic logic unit (ALU) is the GPU component responsible for arithmetic and logic operations.

bank conflict
A bank conflict occurs when multiple *work-items* in a *wavefront* access different addresses that map to the same shared memory bank.

batched GEMM
A *kernel* that calls *VGEMMs* with different batches of data. All the data batches have the same *problem shape*.

block Size
The block size is the number of *work-items* in a *compute unit*.

block tile
A block tile is a memory *tile* processed by a *work group*.

Col2Im
Col2Im is a data transformation technique that converts column data to image format.

compute unit
The compute unit (CU) is the parallel vector processor in an AMD GPU with multiple *ALUs*. Each compute unit will run all the *wavefronts* in a *work group*. A compute unit is equivalent to NVIDIA's streaming multiprocessor.

coordinate transformation primitives
Coordinate transformation primitives are Composable Kernel utilities for converting between different coordinate systems.

dense tensor
A dense tensor is a tensor where most of its elements are non-zero. Dense tensors are typically stored in a contiguous block of memory.

descriptor
Metadata structure that defines *tile* properties, memory layouts, and coordinate transformations for Composable Kernel *operations*.

device
Device refers to the GPU hardware that runs parallel kernels. The device contains the *compute units*, memory hierarchy, and specialized accelerators.

dilation
Dilation is the spacing between *kernel* elements in convolution *operations*, allowing the receptive field to grow without increasing kernel size.

elementwise
An elementwise *operation* is an operation applied to each tensor element independently.

epilogue
The epilogue is the final stage of a kernel. Activation functions, bias, and other post-processing steps are applied in the epilogue.

fast changing dimension
The fast changing dimension is the innermost dimension in memory layout.

fused add multiply
A common fused *operation* in machine language and linear algebra, where an *elementwise* addition is immediately followed by a multiplication. Fused add multiply is often used for bias and scaling in neural network layers.

GEMM
See *general matrix multiply*.

GEMV
See *general matrix vector multiplication*

general matrix multiply
A general matrix multiply (GEMM) is a Core matrix *operation* in linear algebra and deep learning. A GEMM is defined as $C = \alpha AB + \beta C$, where A, B, and C are matrices, and α and β are scalars.

general matrix vector multiplication
General matrix vector multiplication (GEMV) is an *operation* where a matrix is multiplied by a vector, producing another vector.

GGEMM
See *grouped GEMM*.

global memory
The main device memory accessible by all threads, offering high capacity but higher latency than shared memory.

grid
A grid is a collection of *work groups* that run a kernel. Each work group within the grid operates independently and can be scheduled on a different *compute unit*. A grid can be organized into one, two, or three dimensions. A grid is equivalent to an NVIDIA thread block.

grouped GEMM
A *kernel* that calls multiple *VGEMMs*. Each call can have a different *problem shape*.

host
Host refers to the CPU and the main memory system that manages GPU execution. The host is responsible for launching kernels, transferring data, and coordinating overall computation.

host-device transfer
A host-device transfer is the process of moving data between *host* and *device* memory.

Im2Col
Im2Col is a data transformation technique that converts image data to column format.

inner dimension
The inner dimension is the faster-changing dimension in memory layout.

input
See *problem shape*.

kernel
A kernel is a function that runs an *operation* or a collection of operations. A kernel will run in parallel on several *work-items* across the GPU. In Composable Kernel, kernels require *pipelines*.

launch parameters
Launch parameters are the configuration values, such as *grid* and *block size*, that determine how a *kernel* is mapped to hardware resources.

LDS
See *local data share*.

LDS banks
LDS banks are a type of memory organization where consecutive addresses are distributed across multiple memory banks for parallel access. LDS banks are used to prevent memory access conflicts and improve bandwidth when LDS is used.

load tile
Load tile is an operation that transfers data from *global memory* or the *local data share* to *vector general purpose registers*.

local data share
Local data share (LDS) is high-bandwidth, low-latency on-chip memory accessible to all the *work-items* in a *work group*. LDS is equivalent to NVIDIA's shared memory.

matrix core
A matrix core is a specialized GPU unit that accelerate matrix operations for AI and deep learning tasks. A GPU contains multiple matrix cores.

matrix fused multiply-add
Matrix fused multiply-add (MFMA) is a *matrix core* instruction for GEMM *operations*.

memory coalescing
Memory coalescing is an optimization strategy where consecutive *work-items* access consecutive memory addresses in such a way that a single memory transaction serves multiple work-items.

MFMA
See *matrix fused multiply-add*.

naive GEMM
The naive GEMM, sometimes referred to as a vanilla GEMM or VGEMM, is the simplest form of *GEMM* in Composable Kernel. The naive GEMM is defined as $C = AB$, where A, B, and C are matrices. The naive GEMM is the baseline GEMM that all other GEMM *operations* build on.

occupancy
The ratio of active *wavefronts* to the maximum possible number of wavefronts.

operation
An operation is a computation on input data.

outer dimension
The outer dimension is the slower-changing dimension in memory layout.

padding
Padding is the addition of extra elements, often zeros, to tensor edges in order to control output size in convolution and pooling, or to align data for memory access.

permute
Permute is an *operation* that rearranges the order of tensor axes, often for the purposes of matching *kernel* input formats or optimize memory access patterns.

pinned memory
Pinned memory is *host* memory that is page-locked to accelerate transfers between the CPU and GPU.

pipeline
: A Composable Kernel pipeline schedules the sequence of operations for a *kernel*, such as the data loading, computation, and storage phases. A pipeline consists of a *problem* and a *policy*.

policy
: The policy is the part of the *pipeline* that defines memory access patterns and hardware-specific optimizations.

problem
: The problem is the part of the *pipeline* that defines input and output shapes, data types, and mathematical *operations*.

problem shape
: The problem shape defines the dimensions and data types of input tensors that define the *problem*.

reference kernel
: A reference *kernel* is a baseline kernel implementation used to verify correctness and performance. Composable Kernel makes two reference kernels, one for CPU and one for GPU, available.

register
: Registers are the fastest tier of memory. They're used for storing temporary values during computations and are private to the *work-items* that use them.

scalar general purpose register
: A scalar general purpose register (SGPR) is a *register* shared by all the *work-items* in a *wave*. SGPRs are used for constants, addresses, and control flow common across the entire wave.

SGPR
: See *scalar general purpose register*.

SIMD
: See *single-instruction, multi-data*

SIMT
: See *single-instruction, multi-thread*

single-instruction, multi-data
: Single-instruction, multi-data (SIMD) is a parallel computing model where the same instruction is run with different data simultaneously.

single-instruction, multi-thread
: Single-instruction, multi-thread (SIMT) is a parallel computing model where all the *work-items* within a *wavefront* run the same instruction on different data.

sparse tensor
: A sparse tensor is a tensor where most of its elements are zero. Typically only the non-zero elements of a sparse tensor and their indices are stored.

Split-K GEMM
: Split-K GEMM is a parallelization strategy that partitions the reduction dimension (K) of a *GEMM* across multiple *compute units*, increasing parallelism for large matrix multiplications.

store tile
: Store tile is an operation that transfers data from *vector general purpose registers* to *global memory* or the *local data share*.

stride
: A stride is the step size to move from one element to the next in a specific dimension of a tensor or matrix. In convolution and pooling, the stride determines how far the *kernel* moves at each step.

tile
: A tile is a sub-region of a tensor or matrix that is processed by a *work group* or *work-item*. Rectangular data blocks are the unit of computation and memory transfer in Composable Kernel, and are the basis for tiled algorithms.

tile distribution
The tile distribution is the hierarchical data mapping from *work-items* to data in memory.

tile partitioner
The tile partitioner defines the mapping between the *problem* dimensions and GPU hierarchy. It specifies *work group*-level *tile* sizes and determines *grid* dimensions by dividing the problem size by the tile sizes.

tile programming API
The *tile* programming API is Composable Kernel's high-level interface for defining tile-based computations with predefined hardware mappings for data loading and storing.

tile window
Viewport into a larger tensor that defines the current tile's position and boundaries for computation.

transpose
Transpose is an *operation* that rearranges the order of tensor axes, often for the purposes of matching *kernel* input formats or optimize memory access patterns.

user customized tile pipeline
A customized *tile pipeline* that combines custom *problem* and *policy* components for specialized computations.

user customized tile pipeline optimization
The process of tuning the *tile* size, memory access pattern, and hardware utilization for specific workloads.

vanilla GEMM
See *naive GEMM*.

vector
The vector is the smallest data unit processed by an individual *work-item*. A vectors is typically four to sixteen elements, depending on data type and hardware.

vector general purpose register
A vector general purpose register (VGPR) is a *register* that stores individual thread data. Each thread in a *wave* has its own set of VGPRs for private variables and calculations.

VGEMM
See *naive GEMM*.

VGPR
See *vector general purpose register*.

wave tile
A wave *tile* is a sub-tile processed by a single *wavefront* within a *work group*. The wave tile is the base level granularity of a *single-instruction, multi-thread (SIMD)* model.

wavefront
Also referred to as a wave, a wavefront is a group of *work-items* that run the same instruction. A wavefront is equivalent to an NVIDIA warp.

work group
A work group is a collection of *work-items* that can synchronize and share memory. A work group is equivalent to NVIDIA's thread block.

work-item
A work-item is the smallest unit of parallel execution. A work-item runs a single independent instruction stream on a single data element. A work-item is equivalent to an NVIDIA thread.

CHAPTER

FOURTEEN

CONTRIBUTING TO COMPOSABLE KERNEL

Review the Composable Kernel documentation before contributing to the Composable Kernel project. This documentation provides information about core concepts and configurations, as well as providing *steps for building Composable Kernel*. Some of this information is also available in the Composable Kernel README.

Consult the AMD Developer Central portal for more information about AMD products.

14.1 Reporting issues

Use Github issues to log and track issues and enhancement requests.

If you encounter an issue with the Composable Kernel library, search the existing GitHub issues to determine whether the problem has already been reported. If it hasn't, submit a new issue that includes:

- A description of the problem, including what you observed, what you were expecting, and why this was an issue.
- Your configuration details, including the GPU, OS, and ROCm version, and any Docker image you used.
- The steps to reproduce the issue, including any CMake command you used to build the library, as well as the frequency of the issue.
- Any workarounds you've found and what you expect in a resolution.

14.2 Contributing to the codebase

All external contributors to the Composable Kernel codebase must follow these guidelines:

- Use the correct branch: Use your own branch for your changes. Create your branch from the develop branch.
- Describe your changes: Provide the motivation for the changes and a general description of all code changes.
- Add design documents for major changes: Major architectural changes must be accompanied by comprehensive design documents uploaded with your pull request.
- Add inline documentation: Include relevant documentation and inline comments with your code changes.
- Link your pull request to related issues: Add links to any issues resolved by your changes in your pull request description.
- Verify and test the changes: Run all relevant existing tests and write new tests for any new functionality that isn't covered by existing tests.
- Provide performance numbers: Include documentation showing before and after performance numbers for any changes that potentially impact build times or run times.
- Keep your branch up to date: Regularly rebase or merge the develop branch back into your feature branch. This should be done both prior to creating your pull request and during the review process.

- Ensure a manageable pull request size: Pull requests should be limited to approximately one thousand lines. If your changes significantly exceed one thousand lines, break them into smaller pull requests that can be reviewed independently.
- Use pre-commit hooks to adhere to the coding style: Composable Kernel's coding style is defined in .clang-format. Use the provided pre-commit hooks to run clang formatting and linting. Instructions on installing pre-commit hooks are available in the README file.

Forks require an approver from AMD to trigger continuous integration (CI) testing. This approval process is necessary for security and resource management.

Depending on the complexity of your changes, an AMD developer might need to pull your changes and perform additional fixes or modifications before merging. This collaborative approach ensures compatibility with internal systems and standards.

You can see a complete list of pull requests on the Composable Kernel GitHub page.

CHAPTER

FIFTEEN

LICENSE

Copyright (c) 2018- , Advanced Micro Devices, Inc. (Chao Liu, Jing Zhang) Copyright (c) 2019- , Advanced Micro Devices, Inc. (Letao Qin, Qianfeng Zhang, Liang Huang, Shaojie Wang) Copyright (c) 2022- , Advanced Micro Devices, Inc. (Anthony Chang, Chunyu Lai, Illia Silin, Adam Osewski, Poyen Chen, Jehandad Khan) Copyright (c) 2019-2021, Advanced Micro Devices, Inc. (Hanwen Chang) Copyright (c) 2019-2020, Advanced Micro Devices, Inc. (Tejash Shah) Copyright (c) 2020 , Advanced Micro Devices, Inc. (Xiaoyan Zhou) Copyright (c) 2021-2022, Advanced Micro Devices, Inc. (Jianfeng Yan)

SPDX-License-Identifier: MIT Copyright (c) 2018-2025, Advanced Micro Devices, Inc. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

INDEX

A

Add+Multiply, 195
alignment, 195
arithmetic logic unit, 195

B

bank conflict, 195
batched GEMM, 195
block Size, 195
block tile, 195

C

Col2Im, 195
compute unit, 195
coordinate transformation primitives, 195

D

dense tensor, 195
descriptor, 195
device, 195
dilation, 196

E

elementwise, 196
epilogue, 196

F

fast changing dimension, 196
fused add multiply, 196

G

GEMM, 196
GEMV, 196
general matrix multiply, 196
general matrix vector multiplication, 196
GGEMM, 196
global memory, 196
grid, 196
grouped GEMM, 196

H

host, 196
host-device transfer, 196

I

Im2Col, 196
inner dimension, 196
input, 196

K

kernel, 197

L

launch parameters, 197
LDS, 197
LDS banks, 197
load tile, 197
local data share, 197

M

matrix core, 197
matrix fused multiply-add, 197
memory coalescing, 197
MFMA, 197

N

naive GEMM, 197

O

occupancy, 197
operation, 197
outer dimension, 197

P

padding, 197
permute, 197
pinned memory, 197
pipeline, 198
policy, 198
problem, 198
problem shape, 198

R

reference kernel, 198

register, 198

S

scalar general purpose register, 198
SGPR, 198
SIMD, 198
SIMT, 198
single-instruction, multi-data, 198
single-instruction, multi-thread, 198
sparse tensor, 198
Split-K GEMM, 198
store tile, 198
stride, 198

T

tile, 198
tile distribution, 199
tile partitioner, 199
tile programming API, 199
tile window, 199
transpose, 199

U

user customized tile pipeline, 199
user customized tile pipeline optimization, 199

V

vanilla GEMM, 199
vector, 199
vector general purpose register, 199
VGEMM, 199
VGPR, 199

W

wave tile, 199
wavefront, 199
work group, 199
work-item, 199