Metal Performance Primitives (MPP) Programming Guide

Version 1

Developer

2026-03-16 | Copyright © 2026 Apple Inc. | All Rights Reserved.

Contents

Tables and Figures

2026-03-16 | Copyright © 2026 Apple Inc. | All Rights Reserved.

1 Introduction

1.1 Overview

Metal 4 introduces the tensor resource and the Metal Performance Primitives (MPP) framework for authoring machine learning kernels that leverage GPU neural accelerators in the Apple M5 chip. If you're new to GPU programming, this guide contains a basic introduction to get you started using the tensor APIs for matrix operations. For more experienced users, it contains detailed information about key architectural principles to help you maximize performance on Apple silicon GPUs with simple, clean implementations.

The `mpp::tensor_ops` APIs contain a library of compute primitives you can use inside a kernel to invoke threadgroup- or simdgroup-scoped operations on tensors. To illustrate best practices, the following sections cover the design and implementation of an optimized matrix multiplication kernel, as well as advanced parts of the API, which you can use to implement novel fused-kernel designs.

1.2 Metal Tensor APIs

The Metal tensor type encapsulates a base storage pointer, along with additional metadata for the rank, data type, dimensions, and strides. You can create a tensor inside a compute kernel as follows:

```
// Given data ptr_a, dimensions dim_x, dim_y, leading dimension stride ld_a
auto mA = tensor(ptr_a, dextents<int, 2>{dim_x, dim_y}, array<int, 2>{1, ld_a});
```

You can also bind a tensor on the host as part of a compute encoder using the `MTLTensor` API. See the Metal 4 documentation for full details.

You can view a tensor at a particular offset with the `slice` operation.

```
// Given tensor mA with offset_x, offset_y
auto tA = mA.slice(offset_x, offset_y);
```

Internally, the tensor type uses the dimensions of the original tensor and the offset values to compute the dimensions of the new slice. For best performance, you can track the size of the slice statically, as this can help simplify tensor operations by eliminating bounds checking. The static `slice` API supports this usage.

```
// Given tensor mA with offset_x, offset_y, and static dimensions DIM_X, DIM_Y
auto tA = mA.slice<DIM_X, DIM_Y>(offset_x, offset_y);
```

The tensor types may reference memory from different address spaces, such as `device` or `threadgroup`.

2026-03-16 | Copyright © 2026 Apple Inc. | All Rights Reserved.

2 Generalized Matrix Multiplication (GEMM)

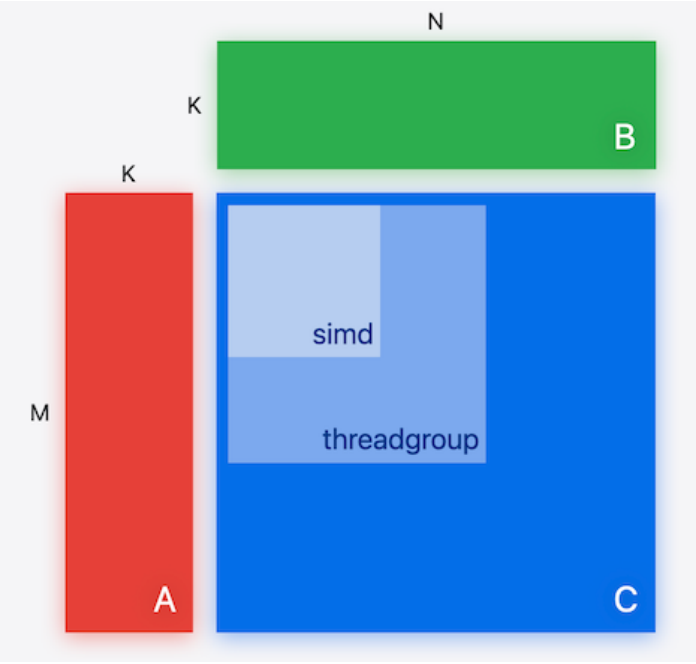
2.1 Background

A GEMM kernel computes the matrix product `D = A @ B`, where `A` is an `M x K` matrix, `B` is a `K x N` matrix, and `D` is an `M x N` matrix. This is a classic performance optimization problem because the data size scales quadratically and the number of arithmetic operations scales cubically. Given a large enough problem dimension, there's enough arithmetic intensity to get peak performance, but achieving this requires maximizing data reuse at every level of the memory hierarchy.

To maximize performance, you can decompose the problem into a hierarchy of output tiles, matching the hierarchy of parallelism available on the GPU:

- Simdgroup: Each simdgroup owns a tile of the output matrix
- Threadgroup: A group of adjacent simdgroup tiles combine to form a threadgroup tile
- Grid: Many threadgroups launch to traverse the entire output matrix

Figure 1. GEMM tiling



For additional background on GEMM performance concepts, see the appendix for theoretical derivations of arithmetic intensity and a more detailed explanation of tiled GEMM algorithms.

Following this hierarchy, the `mpp::tensor_ops` APIs allow you to run a matmul with the output tile size defined at either the simdgroup scope with `execution_simdgroup` or the threadgroup scope with `execution_simdgroups<N>`, where `N` defines the total number of simdgroups in the threadgroup. For a given execution scope, you can program the tile size and other options in the `matmul2d_descriptor`. Note that all threads you specify in the scope need to participate in the tensor operation; otherwise, the behavior is undefined. The following example shows how to

2026-03-16 | Copyright © 2026 Apple Inc. | All Rights Reserved.

compute an output tile of size SM x SN on a single simdgroup, accumulating over the full K dimension of the input matrices:

```
constexpr auto desc = matmul2d_descriptor(SM, SN);
matmul2d<desc, execution_simdgroup> op;
auto mA = tensor(ptr_a, dextents<int, 2>{K, SM}, array<int, 2>{1, ld_a});
auto mB = tensor(ptr_b, dextents<int, 2>{SN, K}, array<int, 2>{1, ld_b});
auto mD = tensor(ptr_d, dextents<int, 2>{SN, SM}, array<int, 2>{1, ld_d});
op.run(mA, mB, mD);
```

Surrounding this code, there are some important optimizations that you can apply to get the best performance out of the GPU. These are described in the following sections.

2.2 Key Distinctions of Apple Silicon GPUs

If you're already familiar with GPU kernel authoring for machine learning, there are some important distinctions about the Apple silicon GPUs that greatly simplify the process of writing high-performing code.

- Theadgroup memory staging isn't necessary for the most optimized GEMM kernels. The on-chip memory hierarchy works best when you directly access device memory and allow the different levels of caching to work as designed.
- Kernels don't require any explicit software pipelining to overlap memory and compute operations. Metal achieves this execution overlap naturally with sufficient thread occupancy on each core.

In combination, these properties lead to optimal code that's much simpler than existing platforms. For absolute peak performance, some additional tuning may be required to optimize cache utilization and execution pipelining.

Putting these principles into practice, the following section highlights the most important optimizations you can apply in GEMM kernels to get the best performance.

2.3 Key Optimizations for GEMM Kernels

2.3.1 Threadgroup Tile Size

The threadgroup tile size needs to be large enough to promote sufficient data reuse between simdgroups. Increasing the threadgroup size too much may reduce performance on smaller shapes due to lower occupancy or dimension quantization effects. These dimension quantization effects come from two main sources:

- If the overall GEMM shape isn't divisible by the tile size, partial tiles along the edges may perform wasted compute. You can mitigate this performance loss by considering divisibility constraints when picking tile sizes.
- If the total number of tiles isn't divisible by the number of GPU cores, some cores may not have threadgroups available to run at the end of kernel execution.

Tile size also affects occupancy because larger tiles result in fewer threadgroups needed to cover a shape. For important workloads, you can maximize performance by tuning over a wide range of tile sizes. Performance tuning with benchmarks provides the best possible performance because it accounts for all machine-specific factors. Because tuning every

2026-03-16 | Copyright © 2026 Apple Inc. | All Rights Reserved.

workload variation isn't feasible, start with generic heuristics for a good baseline tile size selection, and augment with tuning as needed.

On the M5 chip, a `2 x 2` tile of simdgroups per threadgroup is a good starting point for 16-bit floating-point operands, increasing as needed. Smaller data types may require larger tile sizes to reach peak performance. The amount of work that each simdgroup owns also plays a big role in performance, which is covered below.

2.3.2 Simdgroup Tile Size

The simdgroup tile size needs to be large enough to promote sufficient data reuse among threads within a simdgroup. The same occupancy and quantization effects relevant to threadgroup tile size also apply here. On the M5 chip, `SM == SN == 32` is a good starting point for 16-bit floating-point operands, increasing `SM` and `SN` as needed. Smaller data types may require larger tile sizes to reach peak performance. Beyond a certain threshold, increasing simdgroup tile size may reduce performance if operands are too large to fit in fast thread-local memory.

2.3.3 Threadgroup Walk Order

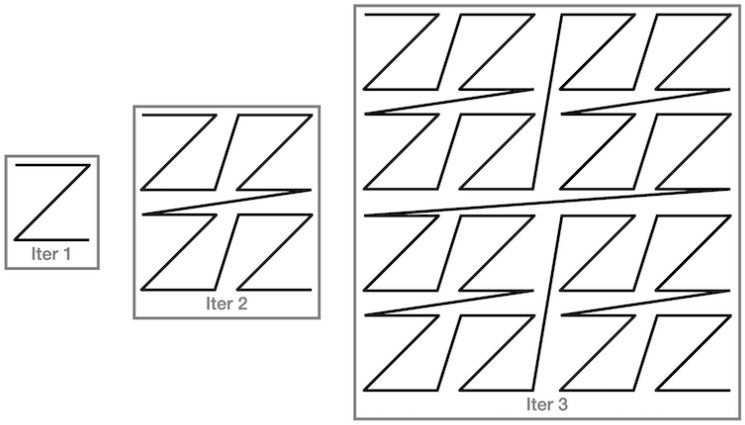
The order that the GPU runs the threadgroup tiles plays an integral role in determining the last-level-cache hit rate of a GEMM kernel. Ideally, actively running threadgroups cover a square region of output tiles in the result matrix so that different cores maximally reuse shards of both input matrices during computation. This reduces the overall memory bandwidth required to reach peak performance.

To get the best cache locality, follow these steps when assigning output tiles to threadgroups in your kernel:

1. Linearize threadgroup size to one dimension `MTLSize(threadsPerThreadgroup, 1, 1)`
2. Linearize grid size to one dimension `MTLSize(threadgroupsPerGrid, 1, 1)`
3. At the start of the kernel, map the `threadgroup_position_in_grid` to two dimensions using a locality-preserving transformation, such as the Morton ordering

The Morton ordering is a common example of a space-filling two-dimensional curve that preserves locality, with a visualization of the curve shown below:

Figure 2. Morton order curve



2026-03-16 | Copyright © 2026 Apple Inc. | All Rights Reserved.

Many similar alternatives exist, such as the Hilbert curve and others. You can experiment with other walk-order options during performance tuning, but Morton ordering is generally sufficient for good performance on the M5 chip. The process of transforming a one-dimensional index into two-dimensional coordinates is often called deinterleaving. One advantage of the Morton ordering is that you can implement deinterleaving with fast bitwise operations, resulting in minimal overhead at kernel startup.

2.3.4 Accumulation Loop Synchronization

During execution, each simdgroup traverses along the K dimension of matrices A and B to accumulate the final result. Within a threadgroup, each simdgroup is in a different position along the K dimension, depending on the dynamics of simdgroup scheduling. Ideally, the simdgroups are relatively close together to maximize data reuse and minimize the overall cache-working-set size. For shapes with a very large K dimension, kernel execution can sometimes synchronize poorly and lead to cache thrashing.

To prevent this, you can multiply accumulate partial results and periodically place threadgroup barriers to synchronize execution. Extending the previous example, you can split the K dimension into tiles of size `BK` and implement this optimization as follows:

```
// Given input tensors mA, mB
constexpr auto mmul_mode = matmul2d_descriptor::mode::multiply_accumulate;
constexpr auto desc = matmul2d_descriptor(SM, SN, BK, false, false, false, mmul_mode);
matmul2d<desc, execution_simdgroup> op;
auto rD = op.get_destination_cooperative_tensor<decltype(mA), decltype(mB), float>();
const int num_k = K / BK;
for (int k = 0; k < num_k; ++k) {
   threadgroup_barrier(mem_flags::mem_none);
   // Slice input tiles tA, tB
   op.run(tA, tB, rD);
}
// Store result rD to device memory
```

Note that the matrix partial products accumulate in a destination cooperative tensor, which distributes elements of the tensor across local memory of threads participating in the operation. For a complete description of the cooperative tensor type, see the section on advanced kernels below.

Tuning the optimal barrier frequency requires a careful balance between synchronization overhead and cache-working-set size. Each barrier reduces the pool of instructions available to execute, so it's recommended that you benchmark important workloads and tune appropriately. On the M5 chip, `BK == 128` is a good starting point, increasing as needed.

2.3.5 Static Tensor Extents

Internally, the `mpp::tensor_ops` implementation needs to perform bounds checking on all loads when the input tensor extents are smaller than the dimensions of the `matmul2d_descriptor` associated with the operation. For best performance, use static tensor extents when the tile size is aligned to avoid the performance penalty from bounds checking. An optimized GEMM implementation contains the following:

- Alignment function constants to identify whether the full problem dimension is a multiple of the threadgroup tile size

2026-03-16 | Copyright © 2026 Apple Inc. | All Rights Reserved.

- Bounds checks to determine if the threadgroup is a full or partial tile, which may be constantly evaluated in the aligned case
- Kernel implementations that use static tensor extents for full tile sizes, and dynamic extents for partial tile sizes

When bounds checks are required for an axis, you can use the `dynamic_extent` constant in the static slice API.

2026-03-16 | Copyright © 2026 Apple Inc. | All Rights Reserved.

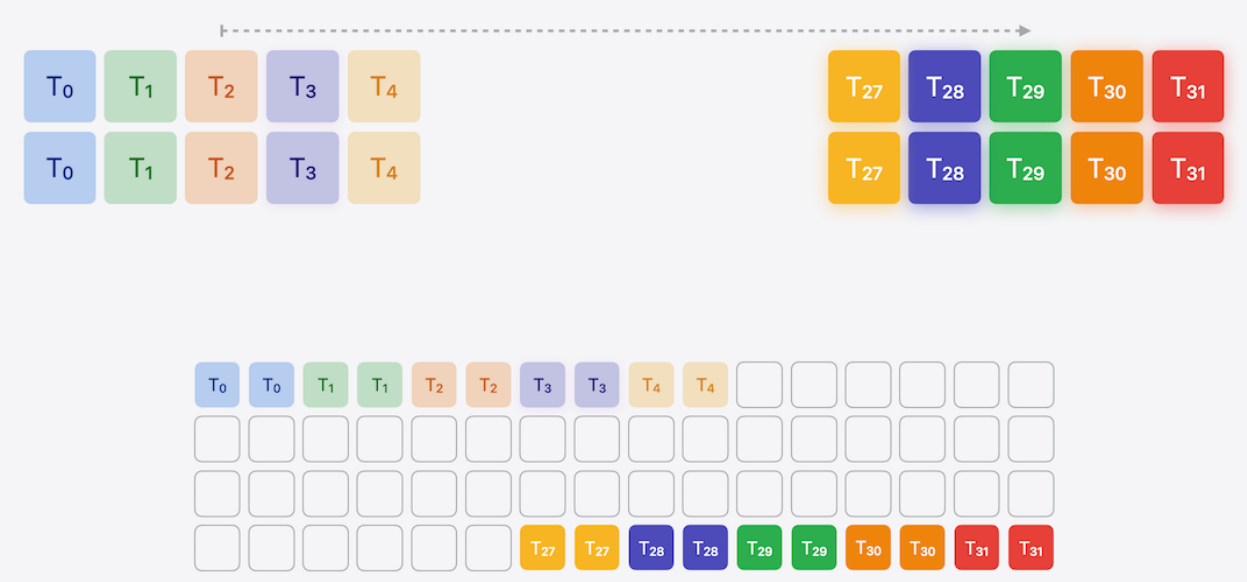
3 Applications in Advanced Kernels

Many machine learning applications implement advanced fused operations to reduce overall memory traffic and achieve peak compute performance at a full network level. The `mpp::tensor_ops` APIs are designed with these use cases in mind, allowing you to embed an accelerated matmul operation into part of a larger kernel design. Both source and destination tensors can reference memory from either a `device` or `threadgroup` address space. Additionally, the source and destination operands may each be a `cooperative_tensor`, enabling fusion in fast thread-local memory.

3.1 Cooperative Tensors

As mentioned earlier, a `cooperative_tensor` distributes elements over threads in the execution scope of a given tensor operation. When you create a tensor operation with a cooperative source or destination, the operation internally determines the correct assignment of element coordinates to threads based on hardware requirements. For example, the diagram below shows how threads of a simdgroup, each represented with unique colors, might distribute ownership of elements of a matrix:

Figure 3. Cooperative tensor storage



Fusing operations with cooperative tensors often yields the best performance by minimizing overall memory traffic and eliminating synchronization overhead associated with threadgroup memory usage. However, cooperative tensors can sometimes be less flexible because the tensor operation determines the assignment of elements to threads. If your application requires a custom thread assignment, you can still fuse tensor operations with threadgroup memory tensors.

3.2 Postfix Fusion

One of the most common fusion opportunities is postfix fusion, which applies elementwise functions to the output of a matrix product directly in thread local memory. This is usually much

2026-03-16 | Copyright © 2026 Apple Inc. | All Rights Reserved.

faster than a round trip through threadgroup or device memory. Postfix fused operations frequently involve a bias addition or activation function.

For example, many high-level tensor libraries define an addmm primitive, which performs a fused scaled addition on the result of the matrix product. Suppose you want to compute D = alpha * (A @ B) + beta * C. Extending the previous optimized example, the final matrix product resides in rD, which is a cooperative_tensor destination. You can then fuse the scaled addition operation as follows:

```
// Given input tensors mA, mB with cooperative destination rD
auto oD = op.get_destination_cooperative_tensor<decltype(mA), decltype(mB), half>();
auto tC = tensor(ptr_c, dextents<int, 2>{SN, SM}, array<int, 2>{1, ld_c});
auto rC = op.get_destination_cooperative_tensor<decltype(mA), decltype(mB), half>();
rC.load(tC);
for (int i = 0; i < rD.get_capacity(); ++i) {
   oD[i] = half((rD[i] * alpha) + (rC[i] * beta));
}
auto tD = tensor(ptr_d, dextents<int, 2>{SN, SM}, array<int, 2>{1, ld_d});
oD.store(tD);
```

Note that the code above uses the cooperative tensor load and store functions for the best vectorization, which is generally preferred over scalar operations on the individual elements.

2026-03-16 | Copyright © 2026 Apple Inc. | All Rights Reserved.

4 Appendix

4.1 Theoretical Analysis of GEMM Performance

Given a GEMM with dimensions `M`, `N`, and `K` as defined above, you can derive expected performance projections from the ratio of compute operations to memory operations, which is a concept known as arithmetic intensity. GEMMs with low arithmetic intensity are memory bound workloads, and GEMMs with high arithmetic intensity are compute bound workloads, forming the basis of a roofline model for kernel performance.

GEMMs have `M * N * K` multiply add operations with `(M * K) + (N * K)` input elements and `M * N` output elements. Hence, the arithmetic intensity is given by:

```
(M * N * K) / ((M * K) + (N * K)) == (M * N) / (M + N) ops per input element
```

and

```
(M * N * K) / (M * N) == K ops per output element
```

with a total arithmetic intensity of:

```
(M * N * K) / ((M * K) + (N * K) + (M * N)) ops per element
```

Therefore, you can make the number of compute operations per load arbitrarily large by increasing both `M` and `N`, achieving the best-case scenario when `M == N`. Similarly, you can make the number of compute operations per store arbitrarily large by increasing `K`.

Using these quantities, you can determine cutoff points where the arithmetic intensity of a workload is high enough to be compute bound. In general, assume that shapes with square outputs and larger dimensions are more likely to be compute bound. The precise cutoff point depends on the peak floating-point operations per second (FLOPs) and the peak memory bandwidth that your GPU supports.

For simplicity, this type of roofline model assumes perfect input reuse in the kernel implementation where each input element loads from device memory only once, and all other accesses hit in the cache. In reality, achieving good cache hit rates requires a carefully tiled implementation, as described in the optimization sections of this document.

In GEMM kernels, a tile refers to some section of the output matrix assigned to a compute execution resource. Optimized GEMM implementations on GPUs use a multilevel tiling hierarchy that assigns tiles to simdgroups and threadgroups, covering the whole output matrix with the launch grid. The optimization sections of this document contain detailed information about how to tune GEMM tile size parameters to achieve peak performance.

2026-03-16 | Copyright © 2026 Apple Inc. | All Rights Reserved.

Apple Inc.
Copyright © 2026 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer or device for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-branded products.

Apple Inc.
One Apple Park Way
Cupertino, CA 95014
408-996-1010

Apple is a trademark of Apple Inc., registered in the U.S. and other countries.

APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT, ERROR OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

Some jurisdictions do not allow the exclusion of implied warranties or liability, so the above exclusion may not apply to you.