Developing CUDA Kernels for Accelerated Matrix Multiplication on NVIDIA Hopper Architecture using the CUTLASS Library

GANESH BIKSHANDI[†] and JAY SHAH[†]

We explain how to develop NVIDIA CUDA kernels for optimized general matrix multiplication (GEMM) on NVIDIA Hopper architecture using the template collection CUTLASS and its core library CuTe. Our main contribution is to provide an implementation of a GEMM kernel that uses the Tensor Memory Accelerator (TMA) and Warp Group Matrix-Multiply-Accumulate (WGMMA) operations introduced with NVIDIA Hopper architecture.

1 INTRODUCTION

The problem of devising computationally efficient algorithms for matrix multiplication is of fundamental importance due to the basic role of this algebraic operation in scientific computing and machine learning. Indeed, it is the ability to massively parallelize matrix multiplication on the GPU that has in large part enabled the rise of AI in the modern world (in the form of ever-more sophisticated neural networks).

As a closed-source and broadly scoped solution, NVIDIA already delivers high-performance implementations of general matrix multiplication (GEMM) in the cuBLAS library and convolutions (CONV) in the cuDNN library. However, in view of the ongoing rapid evolution of GPU architectures and powerful new AI models, it is becoming increasingly important for end-users to be able to implement GEMM and CONV algorithms for their specific applications by writing their own CUDA code. Custom code is mostly needed in the *epilogue* of a GEMM kernel, where another operation has to be fused before writing out the result matrix.

For example, Flash Multi-Head Attention (FMHA) is a recent optimized implementation of attention layers in transformer deep learning models such as Large Language Models (LLMs) [1]. As part of that proposed attention algorithm, one wants to fuse the matrix multiplications with the softmax nonlinear operations in order to exploit data locality and thereby achieve better performance, as opposed to a straightforward implementation without fusion. Implementing FMHA using cuBLAS is not possible as cuBLAS does not expose the thread block or thread-level operations, where the fusion would need to happen.[1]

To address these needs, NVIDIA has provided a suite of CUDA C++ template abstractions for implementing GEMM and related computations through their open-source library CUTLASS, integrated with the CuTe core library and backend since version 3.0. CUTLASS eases the development of optimized kernels by bringing to bear modern software engineering practices otherwise lacking in plain CUDA, such as OOPs, C++ templates (generics), abstract data types (e.g., Tensors, Layouts, and Shapes), and re-usable design patterns.

Nonetheless, it can be challenging for the CUDA programmer to fully leverage the capabilities afforded to them by CUTLASS. In this publication and the accompanying code, we explain and implement a CuTe-based GEMM kernel for Hopper architecture that in particular exploits the new Tensor Memory Accelerator (TMA) and Warp Group MMA (WGMMA) operations to improve performance. To make this publication more self-contained, we also include a larger analysis of the GEMM problem; other insightful sources on this include [3] and [5]. In future work, we plan to integrate our GEMM kernel into FMHA and other LLM kernels.

[1]NVIDIA has announced the cuBLASDx API as a version of cuBLAS that will provide custom fusion support [2], but this isn't widely available yet.

[†]Colfax Research. A copy of this paper is available at https://research.colfax-intl.com/nvidia-hopper-gemm-cutlass/.
Date: December 14, 2023. *Email*: research@colfax-intl.com.

Complete working code for the GEMM kernel explained in this paper is available at: https://github.com/ColfaxResearch/cutlass-kernels.

2 INITIAL RUNDOWN ON GEMM PERFORMANCE

The H100 PCIe GPU provides a maximum compute performance of 51 Teraflops for FP32 execution and 378 Teraflops for TF32 execution [6].[2] The goal for a performant GEMM implementation is to achieve as high a percentage of this theoretical maximum as possible. With reference to the functionality exposed by CUTLASS, performance is sensitive to many parameters such as:

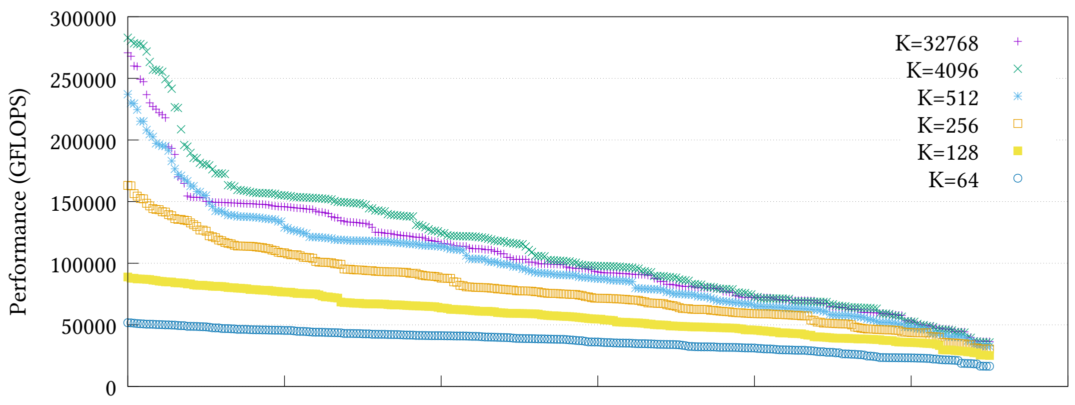

Fig. 1. CUTLASS SGEMM (TF32) Performance on Hopper (H100 PCIe) GPU for M=N=4096 and $K \in \{64,128,256,512,4096,32768\}$ given different tiling shapes, warp counts and stage counts. Sorted in decreasing order of performance.

- Shape of matrices (tall, skinny, or square);
- Layout of matrices (row or column);
- Number of warps per thread block;
- Thread block shape;
- Thread cluster shape;
- Number of pipeline stages in the software pipelining optimization;
- Chosen precision (TF32 vs FP32 vs FP16 vs FP8);
- Usage of special MMA instructions like WGMMA or TMA.

Figure 1 shows the performance variation with different values chosen for TF32 precision.[3] The results were obtained using `cutlass_profiler`, a tool provided by CUTLASS that generates different GEMM kernels off of a base kernel by varying parameters and then measures the resulting running time and GFLOPS (among other metrics).

From our empirical studies, we discovered that the best GEMM kernels try to optimize for maximum Tensor Core utilization and maximum possible GPU occupancy using software pipelining. In Figure 2, we display the

[2]The NVIDIA datasheet lists 756 TFlops for TF32 with sparsity.
[3]Performance is evaluated with respect to the matrix-multiply operation $C = A \cdot B$ for A and B generic $M \times K$ and $K \times N$ matrices, respectively. In general, the variables M, N, and K always refer to these dimensions.

performance of four such versions of GEMM for single-precision TF32 on H100 PCIe GPU. To the best of our knowledge, these kernels have been optimally tuned given the available specifications.

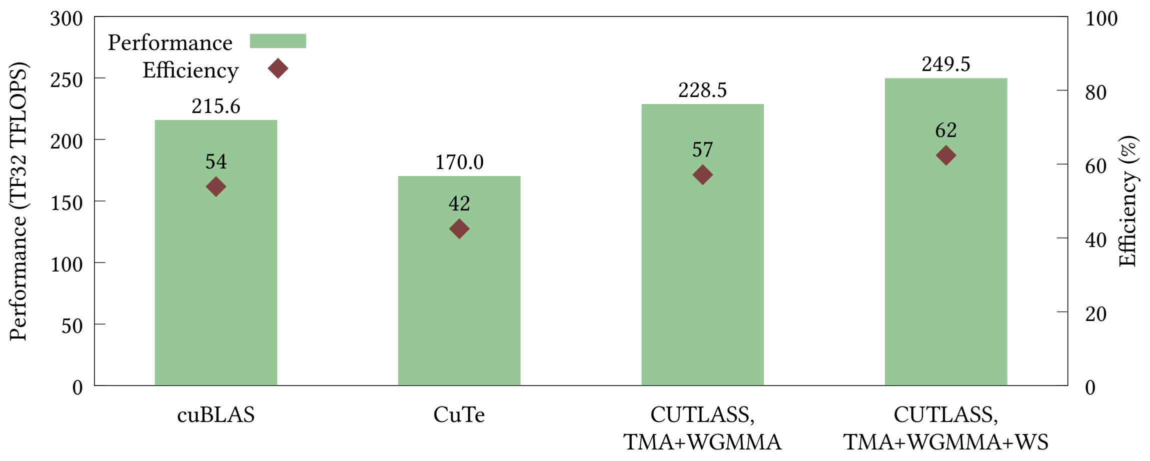

Fig. 2. SGEMM (TF32) Performance for M=N=K=4096.

The four versions listed here are:

(1) cuBLAS: A version from the cuBLAS library in TF32 execution mode (set using `NVIDIA_TF32_OVERRIDE=1`).

(2) CuTe: A hand-implemented version (presented in this paper) using CuTe, a collection of C++ CUDA template abstractions for defining and operating on hierarchically multidimensional layouts of threads and data. This implementation uses TMA for loads and WGMMA for matrix operations. Software pipelining (cf. §5) is not used in this version.

(3) CUTLASS, TMA+WGMMA: A version shipped with the CUTLASS library that uses TMA loads and WGMMA instructions along with software pipelining optimization. This version served as our guide in implementing GEMM kernels using CuTe.

(4) CUTLASS, TMA+WGMMA+WS: An improved version of (3) that also uses Warp Specialization (WS) and Thread Clusters. This version, while still not the best possible, utilizes other key differentiators of Hopper architecture.[4] We have chosen to include this to showcase deeper optimizations available for Hopper.

Discussion on using cuBLAS versus CUTLASS has sometimes been framed as trading off the superior general performance of cuBLAS for the customizability of CUTLASS.[5] However, Figure 2 shows that CUTLASS is now more than competitive with cuBLAS; even our custom version, which implements only a small subset of all possible optimizations, comes close in performance. We also note that the best CUTLASS kernel we studied achieves close to **280 TeraFlops** in performance. This observed discrepancy in performance between cuBLAS and CUTLASS is consistent with NVIDIA's own benchmarking [4].

At any rate, the upshot is that good performance for a custom lightweight GEMM kernel is achievable thanks to CUTLASS given some effort put in by the CUDA programmer, which bodes well for the performance of *fused*

[4]The idea of warp specialization itself is of course not new to Hopper, but it is only implemented in CUTLASS starting with Hopper [11].
[5]See for example https://github.com/NVIDIA/cutlass/issues/109 for public comment.

kernels down the line. Furthermore, in §6 we will see that our CuTe program outperforms both cuBLAS and CUTLASS kernels for *Batched-GEMM* with $K = 64$ and $L = 96$ for `batch_count`.

3 MATRIX MULTIPLICATION ON THE GPU

To orient the reader, we give a rapid review of some of the basic ideas involved in writing a GEMM kernel in CUDA.

3.1 The GPU Memory Hierarchy and CUDA Thread Hierarchy

To begin with, understanding the GPU memory hierarchy is crucial for optimizing GEMM kernels. The GPU has three distinct levels in its memory hierarchy, proceeding from larger and slower to smaller and faster memory:

(1) HBM (High Bandwidth Memory) or Global Memory (GMEM).
(2) Shared memory (on-chip) (SMEM).
(3) Register memory (RMEM).

On the other hand, the CUDA programming model has the thread hierarchy. This is comprised, from coarser to finer groupings, of grids, thread blocks (i.e., cooperative thread arrays or CTAs), and threads.[6] GMEM is available across the entire grid; SMEM is per streaming multiprocessor (SM), to which a thread block is assigned; and individual threads have their own RMEM. In this way, CUDA exposes the GPU memory hierarchy to the programmer [8]. Also note that for the H100 GPU, 32 threads are grouped into one warp for parallel execution on a SM; the size of a warp is fixed by the particular GPU architecture and thus not under the programmer's control.

The matrix multiplication algorithms of interest to us are written to be aware of this hierarchical structure. More precisely, they decompose the top-level matrix multiplication into multiple sub-matrix multiplications (or tiled matrix multiplications). Each decomposition step made in the algorithm corresponds to moving across one of the levels in the CUDA thread hierarchy and GPU memory hierarchy. A typical example of how one might assign tiling shapes to different levels of the hierarchy is given in Figure 3 (taken from [3]).[7]

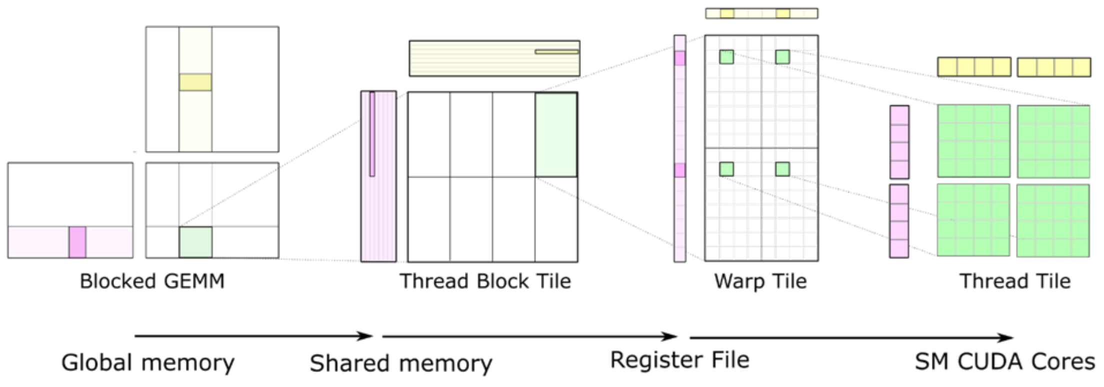

Fig. 3. A typical tiling hierarchy used by a GEMM kernel optimized for NVIDIA GPU.

Figure 3 should be read as follows:

[6]On top of this, Hopper introduces thread block clusters as intermediate between grids and thread blocks. The thread blocks within a cluster, spread over different SMs, can access each other's shared memory (*distributed shared memory*).
[7]In practice, there are many possible choices for how one might tile the matrices (cf. Figure 1).

- At the HBM or global memory level, the A matrix is divided across the rows (M dimension) and the B matrix is divided across the columns (N dimension). The result matrix C is divided along both the M and N dimensions. A row panel of A and a column panel of B are assigned to a thread block, which computes a tile of output C.
- Each thread block further tiles the row panel of A and column panel of B along the K dimension; this ensures that the tiles fit in shared memory. The corresponding tiles of A and B are then multiplied, accumulating the result in the tile of the C matrix assigned to the thread block.
- Further optimization then proceeds recursively by sub-tiling each of the tiles for warp and thread-level computation. Sub-tiles are moved into register memory and the actual element-wise multiplication is always done at the last level of tiling.
- Note that the sub-tile chosen in the warp tile to be assigned to a thread does not have a contiguous shape; rather, we have 32 threads assigned to the sub-tiles in each quadrant of the warp tile, so that each thread sub-tile is spread over all four quadrants. This is done to exploit *memory coalescing* for parallel thread execution.

In the remainder of this section, we go over some (pseudo)code that describes both the naive Matmul and tiled Matmul algorithms.

3.2 Naive Matrix Multiplication

Recall that a naive matrix multiplication can be written using a triply nested loop in C++ as follows:

```
for (int i = 0; i < M; ++i)
   for (int j = 0; j < N; ++j)
      for (int k = 0; k < K; ++k)
         C[i][j] += A[i][k] * B[k][j];
```

However, writing in CUDA means we can parallelize one or two or all the loops. CUDA naturally provides 2D parallelism in the form of thread blocks and grids when a CUDA kernel is launched, which makes it easy to parallelize the i and j loops. The following code snippet illustrates this principle.

```
__global__ void NaiveGemmKernel(int M, int N, int K, float const *A, int lda, float
     const *B, int ldb, float *C, int ldc) {
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   int j = threadIdx.y + blockIdx.y * blockDim.y;
   if (i < M && j < N) {
      for (int k = 0; k < K; ++k)
         C[i + j * ldc] += A[i + k * lda] * B[k + j * ldb];
   }
}
dim3 block(16, 16);
dim3 grid( (M + block.x - 1) / block.x, (N + block.y - 1) / block.y);
NaiveGemmKernel<<< grid, block >>>(M, N, K, A, lda, B, ldb, C, ldc);
```

The k loop is now the main loop of the matrix multiplication, while i and j are not expressed "explicitly" in the CUDA code as they are inferred from the kernel launch parameters given for the thread block and grid dimensions. The k loop is not parallelized, but CUTLASS provides an option to parallelize along that dimension also, via the k-split algorithm. For our discussion, we will suppose that the k loop is not parallelized.

3.3 Outer Product Summation and Tiling

The naive matrix multiplication in §3.2 requires the matrices A and B to be repeatedly brought inside the cache or shared memory. To obviate this issue, the k loop can be permuted to be the outside loop, and this leads to the "outer product" version of matrix multiplication in which the result matrix C is computed as a sum of k many outer products (of *columns* of A and *rows* of B).

```
for (int k = 0; k < K; ++k) // K dimension now outer-most loop
   for (int i = 0; i < M; ++i)
      for (int j = 0; j < N; ++j)
         C[i][j] += A[i][k] * B[k][j];
```

One problem with computing the result matrix C via outer product summation is that this requires C to be live all the time, which for large enough dimensions can quickly exceed all available on-chip memory. A way to circumvent this problem is "tiling" or "blocking". The computation can first be tiled into smaller units that can fit into on-chip memory, after which another set of loops iterates over the tiles:

```
for (int m = 0; m < M; m += MT) // iterate over M dimension
   for (int n = 0; n < N; n += NT) // iterate over N dimension
      for (int k = 0; k < K; ++k)
         for (int i = 0; i < MT; ++i) // compute for one tile
            for (int j = 0; j < NT; ++j) {
               int row = m + i;
               int col = n + j;
               C[row][col] += A[row][k] * B[k][col];
            }
```

Note that we have not yet tiled along the K dimension; this comes next.

3.4 Hierarchical or Recursive Matrix Multiplication

We now refer back to Figure 3 and its multiple levels of tiling. Matrix multiplication optimized for the GPU is implemented in an hierarchical or recursive fashion. At every level of the recursion, the program copies a tile from one memory to another (e.g., global to shared memory). The actual multiply-accumulate-add happens only at the leaf level (the last level), which in Figure 3 is thread-level. At the leaf level, we have a choice to either use the standard multiply-accumulate-add of the SIMT core or specialized Tensor Core instructions. An implementation optimized for Hopper architecture will choose to recurse only till warpgroup-level (instead of thread-level) and use the special WGMMA instructions that use the Tensor Cores more optimally [10, §9.7.14].

Pseudocode illustrating this is displayed below:

```
// MT, NT, KT = dimensions at threadblock level
// MW, NW, KW = dimensions at warp level

// Loop1A: threadblock-level concurrency
Loop1A: for each m, n in M, N with step MT, NT
    Loop1B: for each k in K with step KT
        Move a chunk of A from GMEM to SMEM (As)
        Move a chunk of B from GMEM to SMEM (Bs)
        // Loop2A: warp-level concurrency
        Loop2A: for each mm, nn in MT, NT with step MW, NW
            Loop2B: for each kk in KT with step KW
                Move a chunk of As from SMEM to RMEM (Ar)
                Move a chunk of Bs from SMEM to RMEM (Br)
                // run mma and accumulate in registers
                // further recursion is hidden by mma call
                mma(Ar, Br, accum)
```

Note that the equivalent of the parallelization step handled by launching the CUDA kernel corresponds to the outermost loop in this pseudocode. Since parallelization occurs over chunks of the A and B matrices that now are no longer single rows and columns, converting this pseudocode into working CUDA code is not straightforward. We will see that the CUTLASS API provides a method for doing this via `local_tile`.

3.5 Fusing Operations with Matrix Multiplication

For use in real-life workloads, a GEMM kernel will often involve additional operations to be *fused* with the above Matmul, such as a linear scaling step. The place where that is done (at the end of the outermost loop) is referred to as the *epilogue*. Typically, the accumulator will be in registers in the epilogue. To understand the basic reason for kernel fusion, consider linear scaling: even though this could be done separate to the initial Matmul, fusing in the epilogue helps reduce the total bandwidth requirement, since otherwise the linear scaling would necessitate a re-read of the original C and the resulting D matrix from global memory.

Although the idea is simple, as a software engineering task there are many issues to be aware of when writing a fused kernel. For one, there are multiple elementwise operations that need to be fused with Matmul (e.g., ReLU) in a typical AI-intensive workload. Secondly, the difficulty of fusing various operations can vary greatly with the operation; for example, softmax is harder to handle than linear scaling.

4 DEVELOPING A GEMM KERNEL USING CUTLASS AND CUTE

Developing and maintaining a performant Matmul kernel is difficult mainly because the shape of the tiles at each level must be tuned for different GPU architectures, different precision types of the underlying data, and different matrix sizes. When new levels in the CUDA thread hierarchy are introduced (e.g., the thread cluster level introduced with Hopper architecture), the kernel code must also be changed with additional loops and other code to handle that. The unroll parameters need to be tuned by the compiler too. Apart from all this, there are specialized instructions like WGMMA in Hopper that can boost the performance of the leaf Matmul step. Finally, fusing operations like linear scaling or softmax with Matmul introduces more complexity into the equation.

To handle all of this, one has the CUTLASS library. CUTLASS is a C++ templates-based library that provides a very high-level interface for defining the shape of the tiles at each level of the memory hierarchy. It also provides an interface called *Epilogue* that allows the programmer to fuse operations like linear scaling after the completion of a leaf Matmul. Moreover, CUTLASS defines default leaf level kernels itself and most often selects highly tuned kernels for the GPU architecture based on the tiling parameters supplied during compile time, the specified architecture and the precision.

4.1 CUTLASS API Basics

A basic listing of CUTLASS-based Matmul is described in Listing 1. Apart from CuTe, CUTLASS has the following 3 important APIs for GEMM, each corresponding to a distinct level of the GPU memory hierarchy [12]:

(1) Device API;
(2) Kernel API;
(3) Collective API.

The Collective API embodies a thread block or a cluster of thread blocks (from Hopper architecture onwards). Collective APIs can be used to construct a GEMM as well as the epilogue to be fused with GEMM. The default epilogue simply writes out the accumulator of GEMM from register memory to global memory. CUTLASS defines several other typical operations such as linear scaling and clamping; other device-side function call operators may also be used to perform custom operations.

The Kernel API embodies the entire grid. It thus schedules the collectives and is responsible for tiling the input matrices into row and column panels, loading the references to them and invoking the GEMM and the epilogues. Fusion of epilogues with GEMM happens at the Kernel API level.

The Device API is the highest-level API. It is invoked from the Host (i.e., CPU) and does not have any detail about the specifics of the Matmul implementation. This API is used by host-side .cu code to invoke CUTLASS's GEMM kernels, much like cuBLAS API.

The CUTLASS API relies on C++ templates and meta-programming. Many compile-time values can be set to default (or auto) values, while the optimal values are chosen to be best possible by the CUTLASS implementation. Most of the compile-time parameters described in the listing are self-explanatory. The constructed GEMM kernel can be executed with input and output arguments just like any C++ functor objects.

```
using ElementA = float; // Element type for A matrix operand
using ElementB = float; // Element type for B matrix operand
using ElementC = float; // Element type for C and D matrix operands
using ArchTag = cutlass::arch::Sm90; // Tag indicating the SM
using OperatorClass = cutlass::arch::OpClassTensorOp;  // Operator class tag
using TileShape = Shape<_128,_128,_32>; // Threadblock-level tile size
using ClusterShape = Shape<_1,_2,_1>;   // Shape of the threadblocks in a cluster

                         Collective API

using CollectiveMainloop = typename cutlass::gemm::collective::CollectiveBuilder<
    ArchTag, OperatorClass,
   ElementA, RowMajor, 4,
   ElementB, ColumnMajor, 4,
   ElementAccumulator,
   TileShape, ClusterShape,
   cutlass::gemm::collective::StageCountAuto,
   cutlass::gemm::collective::KernelScheduleAuto
>::CollectiveOp;

using CollectiveEpilogue = typename cutlass::epilogue::collective::
    CollectiveBuilder<
   cutlass::arch::Sm90, cutlass::arch::OpClassTensorOp,
   TileShape, ClusterShape,
   cutlass::epilogue::collective::EpilogueTileAuto,
   ElementC, ElementC,
   ElementC, ColumnMajor, 4,
   ElementC, ColumnMajor, 4,
   cutlass::epilogue::collective::EpilogueScheduleAuto
>::CollectiveOp;
```

Kernel API

```
using GemmKernel = cutlass::gemm::kernel::GemmUniversal<
   Shape<int,int,int>, // Indicates ProblemShape
   CollectiveMainloop,
   CollectiveEpilogue
>;
```

Device API

```
using Gemm = cutlass::gemm::device::GemmUniversalAdapter<GemmKernel>;
```

Listing 1. Constructing a GEMM Kernel for SM90 (Hopper) Architecture

4.2 Tiled Matrix Multiplication Using CuTe

Even though CUTLASS provides APIs for optimized GEMM and fusing operations with GEMM, developing fused kernels like FMHA requires lower level APIs of CUTLASS, as the fusion is not straightforward. Such a custom kernel has been shipped as part of CUTLASS [9]. However, that kernel does not use the new Hopper features and is mostly customized for SM80 architecture. It is somewhat cumbersome to redevelop this for SM90.

CuTe is another API within the CUTLASS API that provides even more flexibility to develop GEMM kernels. It specifically introduces the concept of *Shapes* and *Layouts*, using which programmers can define the different levels of tiling explicitly. Additionally, it provide APIs to:

(a) Convert matrices in to tensors and partition them;
(b) Access the tiles of a tensor that belong to a thread block (`local_tiles`);
(c) Make a local partition of a tensor that belongs to a thread within a thread block (`local_partition`);
(d) Copy between GEMM, SMEM and RMEM (`copy`);
(e) Multiply tensors with special Matmul instructions like WGMMA (`gemm`);

(f) Synchronize between thread clusters;
(g) Make special swizzle layouts for shared memory.

Listing 2 shows a basic CuTe based Matmul CUDA kernel, obtained from CUTLASS repository [14].[8]

```
template <class MShape, class NShape, class KShape,
class TA, class AStride, class ABlockLayout, class AThreadLayout,
class TB, class BStride, class BBlockLayout, class BThreadLayout,
class TC, class CStride, class CBlockLayout, class CThreadLayout,
class Alpha, class Beta>

__global__ static
void
gemm_device(MShape M, NShape N, KShape K,
   TA const* A, AStride dA, ABlockLayout blockA, AThreadLayout tA,
   TB const* B, BStride dB, BBlockLayout blockB, BThreadLayout tB,
   TC      * C, CStride dC, CBlockLayout blockC, CThreadLayout tC,
   Alpha alpha, Beta beta)
{
   using namespace cute;
   using X = Underscore;

   // Shared memory buffers.
   __shared__ TA smemA[cosize_v<ABlockLayout>];
   __shared__ TB smemB[cosize_v<BBlockLayout>];

   auto sA = make_tensor(make_smem_ptr(smemA), blockA);
   auto sB = make_tensor(make_smem_ptr(smemB), blockB);

   // Represent the full tensors.
   auto mA = make_tensor(make_gmem_ptr(A), make_shape(M,K), dA);
   auto mB = make_tensor(make_gmem_ptr(B), make_shape(N,K), dB);
   auto mC = make_tensor(make_gmem_ptr(C), make_shape(M,N), dC);

   // Get the appropriate blocks for this thread block.
   auto MT = size<0>(sA);
   auto NT = size<0>(sB);
   auto KT = size<1>(sB);

   auto gA = local_tile(mA, make_shape(MT, KT), make_coord(blockIdx.x, _));
   auto gB = local_tile(mB, make_shape(NT, KT), make_coord(blockIdx.y, _));
   auto gC = local_tile(mC, make_shape(MT, NT), make_coord(blockIdx.x, blockIdx.y);

   // Define partitioned views of GMEM and SMEM for COPY
```

[8]The code has been adjusted in places for clarity. We also display the kernel only, not the main program that launches the kernel.

```
auto tAgA = local_partition(gA, tA, threadIdx.x);
auto tAsA = local_partition(sA, tA, threadIdx.x);
auto tBgB = local_partition(gB, tB, threadIdx.x);
auto tBsB = local_partition(sB, tB, threadIdx.x);

// Define partitioned views of SMEM for GEMM.
// Partition sA (M,K) by the rows of tC.
auto tCsA = local_partition(sA, tC, threadIdx.x, Step<_1, X>{});
// Partition sB (N,K) by the cols of tC.
auto tCsB = local_partition(sB, tC, threadIdx.x, Step< X,_1>{});
// Partition gC (M,N) by the tile of tC.
auto tCgC = local_partition(gC, tC, threadIdx.x, Step<_1,_1>{});

// Allocate the accumulators (RMEM).
auto tCrC = make_fragment_like(tCgC);
// Clear the accumulators
clear(tCrC);
```

Matmul Begin

```
// Data is copied from GMEM to SMEM using the COPY views.
// gemm(.) operates on the GEMM views.
auto k_max = size<2>(tAgA);
for (int k = 0; k < k_max; ++k) {
   // Copy GMEM to SMEM.
   copy(tAgA(_,_,k), tAsA);
   copy(tBgB(_,_,k), tBsB);

   cp_async_fence();
   cp_async_wait<0>();
   __syncthreads();

 // Compute GEMM on SMEM.
 // Accumulate to registers.
   gemm(tCsA, tCsB, tCrC);
   __syncthreads();
 }
```

Matmul End

```
// Epilogue fusion goes here.
 for (int i = 0; i < size(tCgC); ++i)
 {
     tCgC(i) = tCrC(i);
 }
```

```
}
```

Listing 2. Basic GEMM using CuTe

The key part of the Matmul computation is listed in the loop. The kernel computes the matrix multiplication of A and B^T resulting in C. The matrices A and B are tiled as shown in Figure 4. The tiling of C was already discussed earlier in the naive Matmul implementation. The key differences between naive Matmul and this Matmul are:

(1) Matrix elements are brought from GMEM to SMEM first using an async `copy` operation;
(2) The result matrix C is stored in RMEM and finally written back to GMEM in the epilogue;
(3) The computation is tiled along the K dimension also. This enables step (1), as A and B tiles are small enough to fit in shared memory compared to the full row or column panel.

The two key APIs of CuTe to be emphasized in Listing 2 are:

(1) `local_tile`: extracts the tiles local to a thread block into a tensor.
(2) `local_partition`: extracts the elements local to a thread in a thread block into a tensor.

Once local tiles are obtained using the CuTe API, the corresponding tiles of A and B are multiplied using the GEMM API and the result is accumulated in the C matrix (in registers). After the last tile along the K dimension is processed, the result C is then written to GMEM.

An important feature in CuTe is the *view* of a tensor (in the sense of the C++ concept). During the `copy` operation, where the data is read from global to shared memory, a view based on `AThreadLayoutA` (`tA`) and `BThreadLayout` (`tB`) is used for input tensors. Such a view is created to improve coalescing for global memory loads, for example. However, during the `gemm` operation, a view based on `CThreadLayout` (`tC`) is used. Such a thread-to-data mapping improves the performance of matrix-multiply computation but may not result in coalesced stores to global memory. The original shared memory can be read and written using the different views. Thus, the thread layouts of the `copy` and `gemm` operations are decoupled so that the best choice for each operation can be chosen by the user.

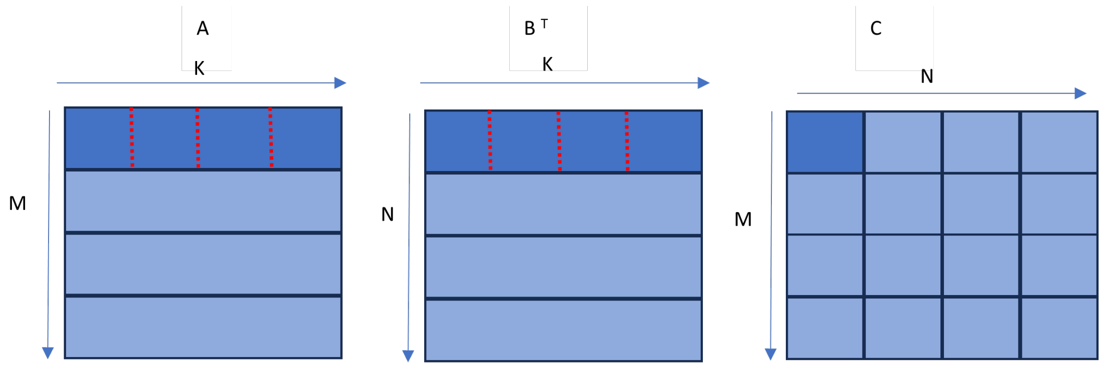

Fig. 4. Graphical representation of Tiled CuTe GEMM kernel with SIMT Core. The optimized implementation in §4.3 will instead use TMA for loading tiles of A and B from GMEM to SMEM, and WGMMA for computing the tiles of C.

Note that we have used no-transpose for A and transpose for B (commonly referred to as NT layout), as that is easy to implement using *SIMT mul-add*. In contrast, the versions shown in Figure 2 all use TN layout. Nevertheless, this version will serve as a basis for the version of Matmul that we present next.

4.3 Incorporating TMA and WGMMA instructions from NVIDIA Hopper Architecture

The listing 2 in the preceding section uses a plain *SIMT mul-add* operation as the atom to compute the product of two tiles. On the other hand, modern NVIDIA GPUs such as the H100 provide Tensor Cores that accelerate mixed-precision computation several-fold. Additionally, Hopper architecture also introduces the Tensor Memory Accelerator (TMA), which can transfer large blocks of data efficiently between global memory and shared memory. To utilize TMA and Tensor Cores, two important changes are required:

- The copy API call should be changed to include the TMA copy atom;
- The gemm API call should be changed to include the MMA atom – for Hopper, we choose WGMMA.

The WGMMA atom schedules data of size 64×8 (other sizes are possible too) split across 128 threads atomically as one operation on a tensor core. The mapping of rows and columns of the resultant $MT \times NT$ tile of C to threads is more complex than those listed in the preceeding sections. CuTe provides utilities that define the mapping, so users need not understand the details of the complex thread layout. At the same time, while implementing complex fused kernels in the *epilogue*, programmers can use CuTe APIs to unravel the thread layout so as to obtain the correct row and column of individual elements of the tile.

Typically, while using WGMMAs the loads of the A and B tiles are executed using TMA. TMA loads offer better performance than *cp.async*. WGMMA operations require that the shared memory allocated for the tiles of the A and B matrices be in a certain “swizzled” format [10], which is supported by TMA. TMA loads are *asynchronous* and are invoked from one thread (typically thread 0), while the other threads wait on a `cuda::barrier` for the operation to be completed. Thus, TMA loads require producer-consumer style synchronization between the threads of a warp.

Listing 3 shows the relevant part of the new GEMM kernel that uses TMA and WGMMA. The copy and gemm views are not shown in the new listing for brevity.[9]

```
....
for (int k = 0; k < size<1>(tAgA); ++k)
{
  .....

  //copy A and B from GMEM to SMEM using COPY views.
  if (threadIdx.x == 0)
  {
    /// Initialize shared memory barrier
    ....
    copy(tma_copy_a, tAgA(_,k), tAsA);
    copy(tma_copy_b, tBgB(_,k), tBsB);
  }
  __syncthreads();

  /// Wait on the shared memory barrier.
  ....
  __syncthreads();
```

[9]CuTe provides APIs to get the correct views for TMA copy and WGMMA gemm operations.

```
    warpgroup_fence_operand(tCrC);

    cute::gemm(wmma_atom, tCrA, tCrB, tCrC);

    warpgroup_commit_batch();
    warpgroup_wait<1>();
    __syncthreads();
  }
   .....
```

Listing 3. GEMM using TMA+WGMMA

As is shown in Figure 5, the TMA+WGMMA version provides almost a sevenfold improvement compared to the CuTe basic version, since it uses the Tensor Cores. However, there is clear room for improvement in terms of Tensor Core utilization.

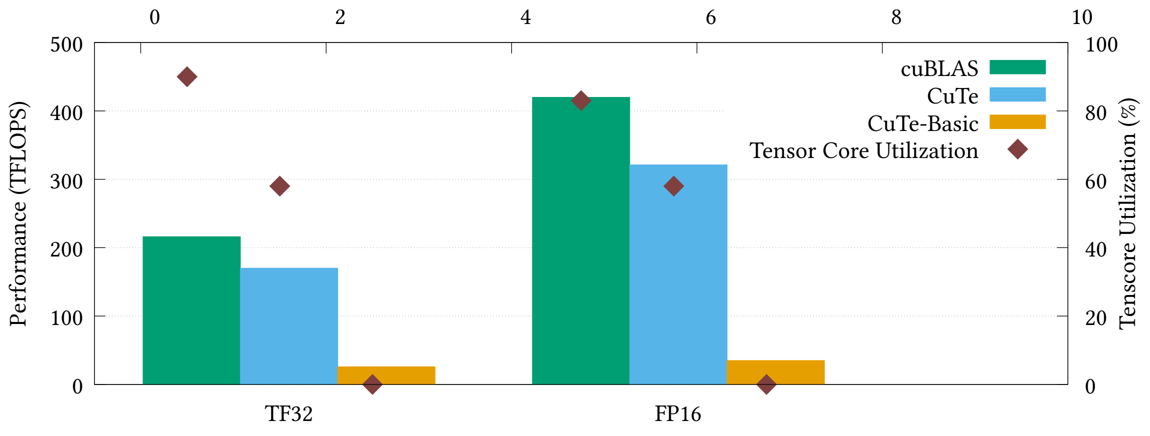

Fig. 5. cuBLAS vs CuTe vs CuTe-Basic (TF32 and FP16 precision).

5 ADDITIONAL OPTIMIZATIONS

Recall from §2 that the best CUTLASS kernel for SGEMM delivers around 280 TFLOPS, while cuBLAS delivers around 215 TFLOPS. CUTLASS implements many more optimizations to achieve this superior level of performance. To list a few from the documentation [11], we have:

(1) **Software Pipelining** – Software pipelining is a technique to hide memory latency where memory accesses and math instructions are executed concurrently, while always accounting for the dependencies between these steps. The CUTLASS implementation uses multiple buffers at both the thread block and warp level.

(2) **Warp Specialization** – With optimizations like software pipelining, different threads or groups of threads naturally have distinct roles. Some are *producers* that load data, while others are *consumers* that run the MMA instructions. The idea of warp specialization is to spatially partition the warps in a thread block into two groupings of producers and consumers.

(3) **Persistent Kernels** – Persistent kernels is a CUDA design pattern that aims to avoid kernel launch and configuration overhead by keeping the kernel persistent on the GPU across multiple calls. In CUTLASS, this involves having persistent thread blocks compute multiple output tiles over their lifetime.

(4) **Two co-operative consumer warp groups** – WGMMA allows the operand A tile to be in register memory too instead of shared memory. However, that restricts the tile size of A due to limited register space. Splitting the tile size across the M dimension into two and assigning to two different consumer warp groups allows for larger tile sizes and eases register pressure.

(5) **Warp-Specialized Persistent Ping-Pong kernel** – The two consumer warp groups from (4) are each assigned to a different output tile. This allows for the epilogue of one consumer warp group to be overlapped with the math operations of the other consumer warp group, thus maximizing tensor core utilization. There is also synchronization on the side of the producer warp groups.

From our empirical studies, point (5) in particular is largely responsible for the gap between the fourth column in Figure 2 and the indicated 280 TFLOPS number for the best measured CUTLASS kernel.

6 BATCHED-GEMM

The AI workflow that we are targeting does not involve multiplying large square matrices. Instead, it involves large square matrices decomposed as products of matrices with small K (e.g., 64 or 128), and with batch count $L > 1$ (e.g., 64 or 96); cf. [1, §2.2]. Such a scheme is popularly known as *Batched-GEMM*. Our CuTe program can be extended to handle Batched-GEMM by simply setting the third dimension of the grid to be L. We then use `blockIdx.z` when using the `local_tile` operation inside the CUDA kernel, as shown in listing 4.

```
 ....
auto gA = local_tile(mA, make_shape(MT, KT), make_coord(blockIdx.x, _, blockIdx.
 z));
auto gB = local_tile(mB, make_shape(NT, KT), make_coord(blockIdx.y, _, blockIdx.
 z));
auto gC = local_tile(mC, make_shape(MT, NT), make_coord(blockIdx.x, blockIdx.y,
 blockIdx.z);
 ....
```

Listing 4. Batched-GEMM kernel using CuTe

Performance of such a Batched-GEMM using CuTe is shown in Figure 6. Surprisingly, the CuTe program outperforms both cuBLAS and CUTLASS, even though it does not use any of the additional optimizations that CUTLASS uses as listed in §5.

At the same time, all of the programs deliver sub-optimal performance for Batched-GEMM. From Figure 1, we can infer that, for small K (64 or 128), the performance of GEMM is sub-optimal in general. In particular, it appears that the optimizations listed in §5 need to be adapted to the case of smaller matrices and Batched-GEMM, or else different methods should be brought to bear.

7 CONCLUSION AND FUTURE WORK

To summarize, we discussed how to develop a CuTe-based GEMM kernel for NVIDIA Hopper architecture that uses the Tensor Memory Accelerator (TMA) and Warp Group MMA (WGMMA) operations. Our CuTe program

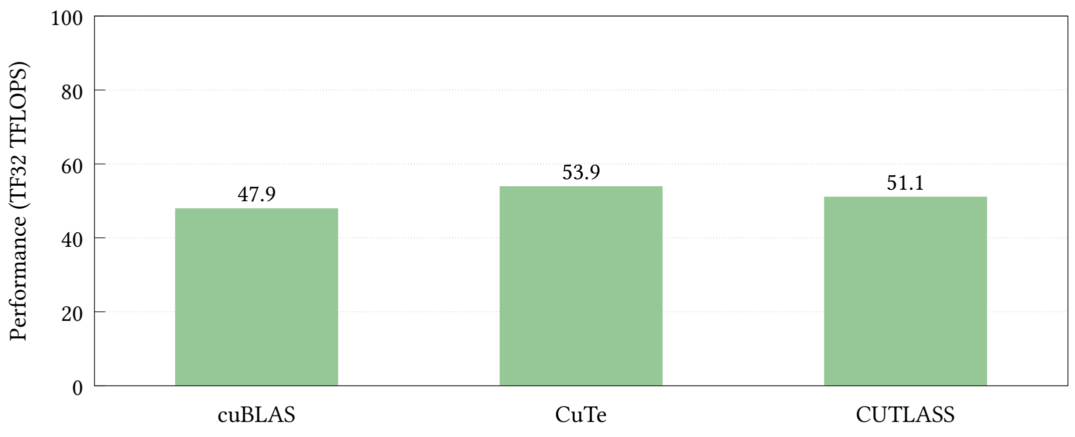

Fig. 6. Batched-SGEMM with M=N=4096, K=64 and L=96.

achieved close to 80% of the performance of the standard cuBLAS GEMM kernel with this one single optimization. For *Batched-GEMM*, our CuTe program outperformed both cuBLAS and CUTLASS.

Currently, we are working to integrate our GEMM kernel with Flash Multi-Head Attention (FMHA) [1], a popular attention layer used in Large Language Models (LLMs). Some important challenges to be addressed are:

- FMHA necessitates changes to the basic GEMM kernel described in this paper. The most significant change is that all panels of the B matrix are assigned to the same thread block, nullifying any parallelism along that dimension in the grid. On the other hand, LLMs use batched GEMM, introducing parallelism along the third dimension L of the grid.
- The online-softmax [13] computation has to be fused with the result of the GEMM (`tCrC`) in registers, which involves atomic operations across threads for computing the maximum and sum of each row.
- FMHA uses a smaller K dimension (64 or 128) in comparison to M and N. From Figure 1, we know that GEMM performance for small K values tends to be very sub-optimal. We plan to adapt the optimizations listed in §5 to the setting of matrices with small K and *Batched-GEMM*.

REFERENCES

[1] *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*. Tri Dao. July 17, 2023. https://arxiv.org/abs/2307.08691.
[2] *New cuBLAS 12.0 Features and Matrix Multiplication Performance on NVIDIA Hopper GPUs*. Roman Dubtsov, Evarist Fomenko and Babak Hejazi. February 1, 2023. https://developer.nvidia.com/blog/new-cublas-12-0-features-and-matrix-multiplication-performance-on-nvidia-hopper-gpus/
[3] *CUTLASS: Fast Linear Algebra in CUDA C++*. Andrew Kerr, Duane Merrill, Julien Demouth and John Tran. December 5, 2017. https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/.
[4] *CUTLASS 3.2 – Performance*. https://github.com/NVIDIA/cutlass#performance.
[5] *CuTe dense matrix-matrix multiply tutorial*. https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/0x_gemm_tutorial.md.
[6] *NVIDIA H100 Tensor Core GPU Datasheet*. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet.
[7] *NVIDIA Hopper Architecture In-Depth*. Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito and Sridhar Ramaswamy. March 22, 2022. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[8] *CUDA Refresher: The CUDA Programming Model*. Pradeep Gupta. https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

[9] *xFormers: A modular and hackable Transformer modelling library*. Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza. 2022. https://github.com/facebookresearch/xformers.

[10] *Parallel Thread Execution ISA Version 8.2*. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[11] *Efficient GEMM in CUDA*. https://github.com/NVIDIA/cutlass/blob/main/media/docs/efficient_gemm.md.

[12] *CUTLASS 3.0 GEMM API*. https://github.com/NVIDIA/cutlass/blob/main/media/docs/gemm_api_3x.md

[13] *Online normalizer calculation for softmax*. Maxim Milakov and Natalia Gimelshein. 2018. https://doi.org/10.48550/arXiv.1805.02867.

[14] https://github.com/NVIDIA/cutlass/blob/main/examples/cute/tutorial/sgemm_nt_1.cu.