A Case Study in CUDA Kernel Fusion: Implementing FlashAttention-2 on NVIDIA Hopper Architecture using the CUTLASS Library

GANESH BIKSHANDI[†] and JAY SHAH[†]

We provide an optimized implementation of the forward pass of FlashAttention-2, a popular memory-aware scaled dot-product attention algorithm, as a custom fused CUDA kernel targeting NVIDIA Hopper architecture and written using the open-source CUTLASS library. In doing so, we explain the challenges and techniques involved in fusing online-softmax with back-to-back GEMM kernels, utilizing the Hopper-specific Tensor Memory Accelerator (TMA) and Warpgroup Matrix-Multiply-Accumulate (WGMMA) instructions, defining and transforming CUTLASS Layouts and Tensors, overlapping copy and GEMM operations, and choosing optimal tile sizes for the Q, K and V attention matrices while balancing the register pressure and shared memory utilization. In head-to-head benchmarks on a single H100 PCIe GPU for some common choices of hyperparameters, we observe 20-50% higher FLOPs/s over a version of FlashAttention-2 optimized for last-generation NVIDIA Ampere architecture.

1 INTRODUCTION

For parallel programming on the GPU, one of the most powerful and complex techniques the programmer has at their disposal is *kernel fusion*, which simply refers to the task of combining multiple individual kernels into a new single kernel. The main benefit from kernel fusion comes from reducing the number of reads from and writes to global memory, which is the largest and slowest level of the GPU memory hierarchy. Since contemporary applications are often memory-bound (as opposed to compute-bound) due to growth in GPU compute power outpacing that of memory bandwidth, kernel fusion's importance to breaking through the so-called "memory wall" in intensive workloads has only risen with time.[1]

Among the most challenging such workloads today involve the training and inference of *large language models* (LLMs). Contemporary LLMs are transformer deep learning models that contain an enormous number of learnable parameters; for example, GPT-3 has about 175 billion parameters [2]. At the heart of the transformer architecture is the attention mechanism [3]. Attention involves two matrix multiplications and a row-wise softmax operation and is recalled in §2. Given the centrality of attention to the transformer model, this would appear to be a natural candidate for kernel fusion. It is perhaps surprising then that to our knowledge, the first published attempt to write attention as a fused kernel was only presented in 2022 in the form of the FlashAttention algorithm by Dao et al. [4], which they present as a "memory-aware" version of attention. Subsequently, Dao reworked the algorithm in [5], as well as recoding it from the ground up using NVIDIA's open-source CUTLASS library for high-performance linear algebra [9, 10].[2] FlashAttention has since seen widespread adoption and is regarded as the current state-of-the-art [7, 8].

In this paper and the accompanying code, we will take FlashAttention-2 from [5] as a model example of kernel fusion and consider the engineering challenges involved in implementing it as a CUDA kernel. As in [5], we will heavily rely on tools from the CUTLASS library, which greatly simplifies the development of CUDA kernels through the systematic use of abstractions such as Layouts and Tensors. As our interest is primarily didactic, we will constrain ourselves to the forward pass of attention for our study.[3] Our goal is twofold:

(1) To give an implementation specifically targeting Hopper (SM90) architecture through using Hopper-specific features such as the Tensor Memory Accelerator (TMA) for copying and Warpgroup Matrix-Multiply-Accumulate (WGMMA) instructions for GEMM. In contrast, the implementation in [5] targets Ampere

[1] See [24] for a polemical take on this, though in a sense NVIDIA's CUTLASS library stands as a rebuttal to that article's main claim.
[2] CUTLASS itself was completely rewritten in 2023 for the release of version 3. In particular, the backend core library CuTe is new to version 3.
[3] By contrast, the backward pass needed for the backpropagation step during training has a different problem profile as an exercise in kernel fusion. In particular, it involves heavier pressure on the shared memory [5, §2.3.2].

[†] Colfax Research. A copy of this paper is available at https://research.colfax-intl.com/nvidia-hopper-flashattention-2/.
Date: December 18, 2023. *Email*: research@colfax-intl.com.

(SM80) architecture.[4] When benchmarked against FLASH-2 from the Dao AI Lab [6] on a single H100 PCIe GPU, we find 20-50% higher FLOPs/s in certain representative cases (cf. Figure 3 in §7).

(2) To document and explain some of the challenges and techniques involved that might be generally applicable to problems in kernel fusion, such as the importance of CUTLASS Layouts and transformations thereof. The aim is to complement the discussion in [4, 5] by highlighting some implementation-level details that those papers gloss over.

The accompanying code can be found at https://github.com/ColfaxResearch/cutlass-kernels/tree/master/src/fmha. We highly encourage the reader to run and examine it alongside reading this paper.

2 STANDARD ATTENTION AND FLASH (MEMORY-AWARE) ATTENTION

In this section, we give a rapid review of attention in a transformer model and the FlashAttention-2 algorithm. The input to a transformer model is a batch of tokens of shape $(L = batch\_size, N = seqlen)$. The embedding layer converts this input into a tensor M of shape $(L, N, D = embedding\_dim)$. We then obtain the three Q, K and V tensors by multiplying M with three separate trainable weight matrices of square dimension D (as a batched matmul). Q, K, V are then divided along the D mode into h many "heads" of head dimension $d = D/h$, so we get these three tensors to be of shape

$$(L = batch\_size, N = seqlen, h = \#heads, d = headdim).$$

Each head is trained and inferred independently. The choices of h and d depend on the model, but generally $N \gg d$. For example, the *distilbert-base-uncased* model [20] uses $d = 64$, $h = 12$, and $N = 512$ by default.

Abusing notation, let Q, K, and V also denote the $N \times d$ matrices associated to a given head. Then the attention output is given by the formula[5]

$$O = \text{softmax}\left(\frac{1}{\sqrt{d}} QK^T\right) V = \text{softmax}\left(\frac{1}{\sqrt{d}} S\right) V = PV$$

where $S = \frac{1}{\sqrt{d}} QK^T$ and $P = \text{softmax}(S)$ are standard variable names for the intermediate expressions. In practice, we replace S by $S - \text{rowmax}(S)$ before taking softmax to avoid overflow with the exponential function; this doesn't change the output of softmax.[6] Concatenating over all heads and batches yields the output tensor to be fed into subsequent layers of the model. Observe that the computation of O is independent over the different heads and batches and thus can be executed in parallel. GEMM is also naturally parallelizable along both rows and columns [1]. With this in mind, we have the following naïve (or standard) implementation of attention on the GPU:

Algorithm 1 Standard Attention

1: Load Q and K by blocks from HBM.
2: Compute $S = (1/\sqrt{d})QK^T$ (GEMM-I).
3: Write S to HBM.
4: Read S from HBM.
5: Compute $S = S - \text{rowmax}(S)$.
6: Compute $P = \text{softmax}(S)$.
7: Write P to HBM.
8: Load P and V by blocks from HBM.
9: Compute $O = PV$ (GEMM-II).
10: Write O to HBM.

[4] Dao has advertised a comprehensive Hopper-specific implementation as work-in-progress [5, p. 12].
[5] One also has optional masking of S and dropout for P.
[6] The expression $S - \text{rowmax}(S)$ means we subtract each entry in S by the maximum entry in its respective row. In general, translating a vector by a common value doesn't change the softmax of the vector.

Materializing the matrices S and P to HBM (i.e., gmem) adversely impacts the overall runtime as well as the memory requirement. Indeed, the size of S scales quadratically with the sequence length N, which is large (e.g., on the order of 4K or even 32K for state-of-the-art LLM models). Instead, one wants to fuse the individual steps of the attention operation into a single CUDA kernel, thereby bypassing intermediate writes to gmem. Fusing GEMM with element-wise operations is very straightforward. By comparison, fusing softmax with GEMM is not straightforward as *a priori* softmax involves computing the global maximum and sum along the rows of S, while fusion should occur at the threadblock (i.e., CTA) level.

The Fused Multi-Head Attention (FMHA) algorithm in [5], taking inspiration from the online-softmax algorithm [19], restructures the attention computation to overcome these difficulties and successfully accomplish the fusion.[7] Specifically, the FMHA algorithm tiles the matrices Q and K and computes a "partial" or "local" softmax on the output of GEMM-I, storing the result in local memory (smem or rmem). During GEMM-II with tiles of V, the partial results are read from local memory, re-scaled with the missing scaling factor and summed back to the result.

The FMHA algorithm is recalled as Algorithm 2 and illustrated in Figure 1. We have chosen tile sizes for Q (bM = QBLK) and K, V (bN = KBLK) such that Q, K, V are split into tiles along the row dimension (i.e., M or N dimension), keeping the K-dimension un-tiled ($bK = d$).[8] We have also chosen to display the variant of the algorithm where the first operand for the second GEMM is stored in rmem as opposed to smem.

Algorithm 2 FlashAttention-2 (FMHA)

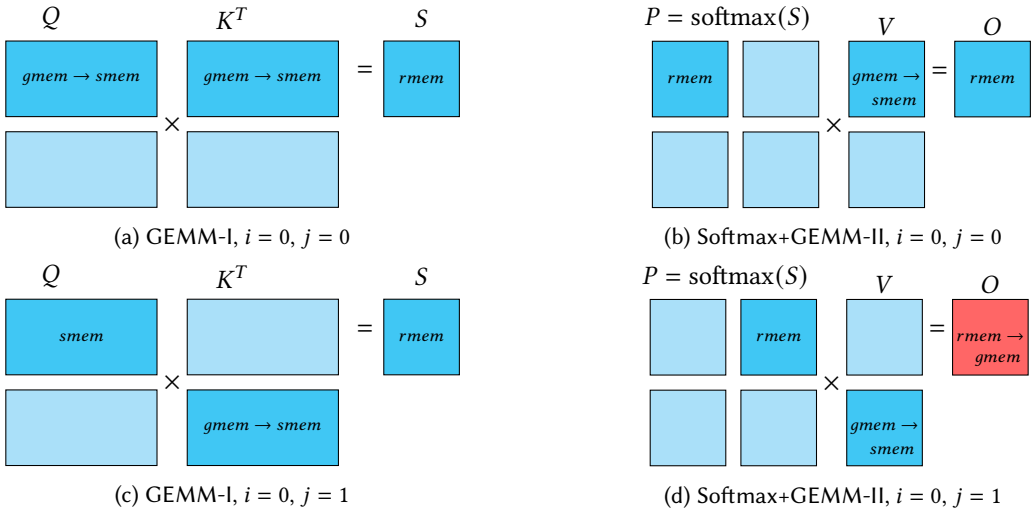
for i **in** range(tiles of Q) **do**
  Load $bM \times d$ tile Q_i from HBM to SMEM.
  Initialize $bM \times d$ accumulator $O_i = (0)$.
  Initialize $bM \times 2$ rowmax $m_i = (-\infty)$ and $bM \times 1$ rowsum $\Sigma_i = (0)$.
  for j **in** range(tiles of K) **do**
    Load $bN \times d$ tile K_j from HBM to SMEM.
    Compute $S_{ij} = (1/\sqrt{d})(Q_i K_j^T)$ (SS-GEMM-I).
    Update rowmax $m_i = (m_i^{\text{new}}, m_i^{\text{old}})$, tracking rowmax at steps j and $j-1$.
    Compute $\widetilde{P}_{ij} = \exp(S_{ij} - m_i^{\text{new}})$.
    Update rowsum $\Sigma_i = \exp(m_i^{\text{old}} - m_i^{\text{new}})\Sigma_i + \text{rowsum}(\widetilde{P}_{ij})$.
    Load $bN \times d$ tile V_j from HBM to SMEM.
    Compute $O_i = \exp(m_i^{\text{old}} - m_i^{\text{new}})O_i + \widetilde{P}_{ij}V_j$ (RS-GEMM-II).
  end for
  Compute $O_i = (1/\Sigma_i)O_i$.
  Write O_i to HBM.
end for

Accumulators O_i, S_{ij}, m_i, and Σ_i are implicitly stored in rmem. Note that over the outer loop, the algorithm is parallel over threadblocks, while within the outer loop, the algorithm executes within a single threadblock. Therefore, in code the inner loop will appear as the *mainloop* of the computation.

3 CUTLASS/CUTE FUNDAMENTALS, TMA, AND WGMMA

NVIDIA's open-source library CUTLASS and its backend core library CuTe allow one to efficiently write a fused CUDA kernel customized for Hopper (SM90) architecture. In this section, we describe the abstractions and methods from CUTLASS/CuTe that we need to implement Algorithm 2 as a CUDA kernel, including asynchronous copy and gemm via Hopper-specific TMA and WGMMA instructions.

[7] Strictly speaking, FlashAttention-2 is an example of an FMHA algorithm, but we will also refer to it as FMHA in this paper for brevity.
[8] K as the inner dimension for GEMM should not be confused with the matrix K.



(a) GEMM-I, $i = 0$, $j = 0$

(b) Softmax+GEMM-II, $i = 0$, $j = 0$

(c) GEMM-I, $i = 0$, $j = 1$

(d) Softmax+GEMM-II, $i = 0$, $j = 1$

Fig. 1. Steps of the FMHA algorithm. `smem` and `rmem` are used between different stages of computation. Output is written to `gmem` in the last stage. Not shown in the figure are the scalings applied to S and O across iterations.

3.1 Layouts and Tensors

The core abstraction of CuTe is the Layout [11]. Mathematically, a layout $L = \mathbf{n} : \mathbf{d}$ is an object comprised of two integer tuples of common length, the shape $\mathbf{n} = (n_1, ..., n_s)$ and the stride $\mathbf{d} = (d_1, ..., d_s)$,[9] which determine a multi-linear function

$$g : [0, n_1) \times \cdots \times [0, n_s) \to \mathbb{N}, \quad (a_1, ..., a_s) \mapsto \Sigma_{i=1}^{s} a_i d_i$$

or equivalently a function of a single variable

$$f = g \circ \iota : [0, \Pi_{i=1}^{s} n_i) \cong [0, n_1) \times \cdots \times [0, n_s) \to \mathbb{N},$$

where the isomorphism ι is given by the "column-major" traversal. For example, the "column-major" layout $L = (4, 4) : (1, 4)$ specifies the identity inclusion

$$f : [0, 16) \to \mathbb{N}, \quad i \mapsto i,$$

whereas the "row-major" layout $L' = (4, 4) : (4, 1)$ specifies the function sending $[0, 16)$ in order to

$$\{0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15\}.$$

The use of Layouts is to specify mappings from a logical coordinate (or tuple of coordinates) to a physical coordinate, like an address in HBM or shared memory, or alternatively another logical coordinate, like a mapping of a (thread, value) coordinate to a matrix coordinate for an MMA instruction. For example, the WGMMA instruction with 64×64 accumulator C has the Layout (in `cute/atom/mma_traits_sm90_gmma.hpp`):

```
using CLayout_64x64  = Layout<Shape <Shape <  _4,_8, _4>,Shape < _2,_2,  _8>>,
                              Stride<Stride<_128,_1,_16>,Stride<_64,_8,_512>>>;
```

Listing 1. The 64×64 accumulator Layout for WGMMA.

This describes the $(T, V) \mapsto (M, N)$ mapping and corresponds to Figure 2 [18, Figure 118 in §9.7.14].[10]

[9]These tuples can be also nested but should match in extent, e.g. one could have $L = ((2, 2), 8) : ((1, 2), 4)$. The associated layout function is insensitive to parenthesization, but layout operations like reduction along a mode can depend on such.

[10]WGMMA is per warpgroup, so involves 128 threads. To match against $64 * 64$ entries, one thus has 32 values per thread. Note how the function associated to `CLayout_64x64` restricts to an isomorphism $[0, 64 * 64) \xrightarrow{\cong} [0, 64 * 64)$, and we implicitly have the "column-major" isomorphism $[0, 64 * 64) \cong [0, 64) \times [0, 64)$ for matching the one-dimensional codomain to the two-dimensional logical (M, N) coordinate.

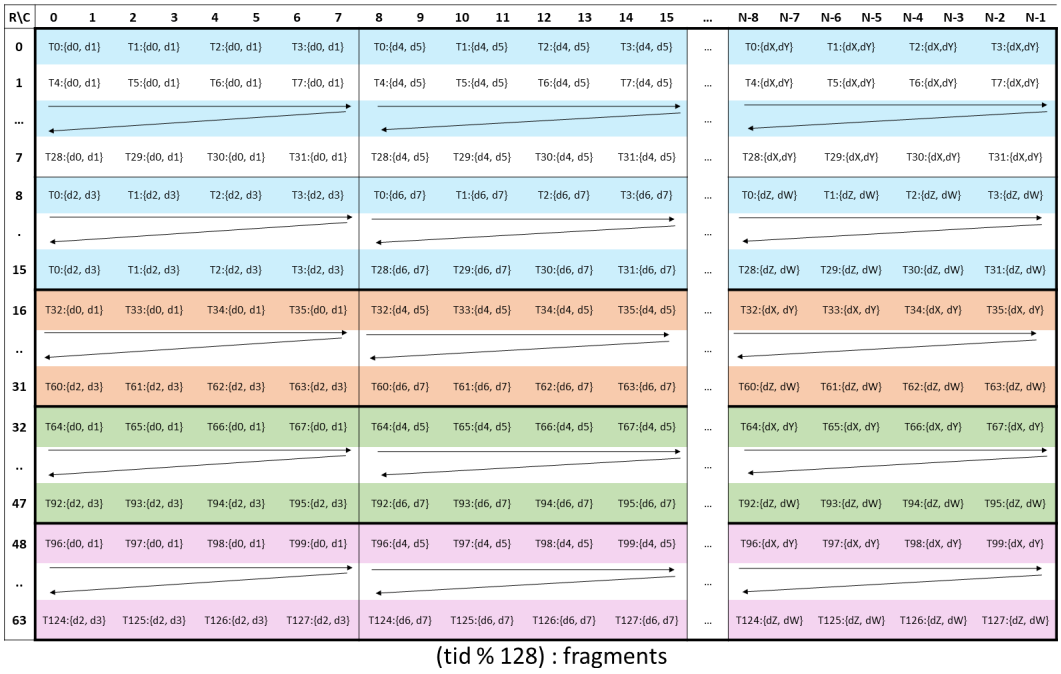


Fig. 2. Structure of WGMMA accumulator. Take N to be a multiple of 16 for specific choices of tile size, e.g. $N = 64$.

CuTe Tensors then are constructed from Layouts and pointers into memory, or can be derived from other Tensors (e.g., slicing a Tensor to get a thread-level view). Tensors can be "owning" (e.g., in registers) or "non-owning" (e.g., a view, in the C++20 sense, of global or shared memory) [12].

Though these abstractions may seem complicated at first glance, a working understanding of Layouts and Tensors is essential for developing a fused kernel with CUTLASS. For example, we will see the necessity of "reshaping" an accumulator layout (from GEMM-I) to an operand layout (for GEMM-II).

3.2 TMA Copy

Hopper introduces the dedicated Tensor Memory Accelerator (TMA) unit for asynchronous copying from `gmem` to `smem`. TMA is exposed by CUTLASS via `make_tma_copy`, which is constructed using the full `gmem` Tensor and the target `smem` Layout:

```
auto tileShapeQ = make_shape(bM{}, bK{});
auto smemLayoutQ = tile_to_shape(GMMA::Layout_K_SW128_Atom<MmaA>{}, tileShapeQ);
Layout gmemLayoutQ = make_layout(make_shape(M, K, H, B), make_stride(K * H, 1, K, H
     * M * K));
Tensor gQ = make_tensor(ptrQ, gmemLayoutQ);
auto tmaQ = make_tma_copy(SM90_TMA_LOAD{}, gQ, smemLayoutQ, tileShapeQ, Int<1>{});
```

Listing 2. Constructing the TMA copy object for Q

For optimal performance, we choose the K-major (i.e., row-major) 128-byte swizzling format for `smemLayoutQ`; since the atom is premade for us in CuTe, we just need to invoke `tile_to_shape` to fit this to the given tile shape.[11] Swizzling in the shared memory is a standard CUDA optimization technique that serves to mitigate

[11]Technically, this produces `smemLayoutQ` as a ComposedLayout object in CuTe, where one postcomposes the function represented by the Layout with the swizzle function.

bank conflicts [22]. On device, we can then construct thread-level views of gmem and smem (in the code, given by the Tensors tQgQ and tQsQ) and execute the copy operation using tmaQ:[12]

```
cfk::copy(tQgQ(_, 0), tQsQ(_, 0), tmaLoadQ, tma_load_mbar[0]);
```

Listing 3. Executing the TMA copy.

In Listing 3, our custom cfk method wraps cute::copy with some synchronization/barrier logic. The other instances of copying from gmem to smem are handled similarly.

3.3 TiledMMA and GEMM

For optimal performance with Hopper, we want to execute asynchronous WGMMA instructions for matrix multiplication. To facilitate this, CuTe has the MMA atom [13] and the TiledMMA object wrapping it:

```
// USE SS version of GMMA for GEMM-I.
using TiledMma0 = decltype(cute::make_tiled_mma(
      cute::GMMA::ss_op_selector<MmaA, MmaB, MmaC, Shape<bM, bN, bK>>(),
      MmaTileShape{}));
// USE RS version of GMMA for GEMM-II (Default).
using TiledMma1 = decltype(cute::make_tiled_mma(
      cute::GMMA::rs_op_selector<MmaA, MmaB, MmaC, Shape<bM, bK, bN>,
                                 GMMA::Major::K, GMMA::Major::MN>(),
      MmaTileShape{}));
```

Listing 4. TiledMMAs for GEMM-I and GEMM-II

Here, ss_op_selector and rs_op_selector are CuTe helper functions[13] for selecting appropriate SM90 MMA atoms given the target tile sizes, precision formats (the operand types MmaA and MmaB are FP16 while the accumulator type MmaC is FP32), and choice of whether to put the first operand in smem (as for GEMM-I) or rmem (as for GEMM-II). The TiledMMA objects are then used to construct the Tensors that will be arguments for the gemm call. For example, we have for GEMM-I:

```
TiledMma0 tiledMma0;
auto threadMma0 = tiledMma0.get_thread_slice(threadIdx.x);
Tensor tSrQ = threadMma0.partition_fragment_A(sQ);
Tensor tSrK = threadMma0.partition_fragment_B(sK);
Tensor tSrS = partition_fragment_C(tiledMma0, tileShapeS);
// ...
cfk::gemm_bar_wait(tiledMma0, tSrQ, tSrK, tSrS, tma_load_mbar[0]);
```

Listing 5. Code for GEMM-I. The gemm call happens within the mainloop.

In Listing 5, our custom cfk method wraps cute::gemm with some synchronization/barrier logic. Similarly, we have for GEMM-II:

```
cfk::gemm_bar_wait(tiledMma1, convert_type<PrecType, AccumType>(tOrP), tOrV,
                       tOrO, tma_load_mbar[1]);
```

Listing 6. The gemm call for GEMM-II, within the mainloop.

[12]Note though that the TMA programming model is single-threaded - one thread is elected to carry out the copy.
[13]Source code in cute/arch/mma_sm90.hpp.

4 LAYOUT TRANSFORMATIONS

Correctly defining the Layouts for the Tensors `tOrP` and `tOrV` featuring as operands for GEMM-II in Listing 6 involves effecting certain transformations of prior Layouts, which we discuss in this section.

4.1 Taking a Transposed Layout

By default, a GEMM call in CuTe is in the BLAS NT format, so `cute::gemm` takes an $M \times K$-matrix A and a $N \times K$-matrix B, and computes $C = AB^T$. Recall that GEMM-I concerns QK^T while GEMM-II concerns PV. Thus, we need to alter the Layout for V (once in `smem`) so that it can be accepted as the second operand for GEMM-II. Given the `smem` Layout for V with shape (bN, bK), we can get to the transposed Layout by a precomposition trick:

```
auto tileShapeV = make_shape(bN{}, bK{});
auto smemLayoutV = tile_to_shape(GMMA::Layout_K_SW128_Atom<MmaB>{}, tileShapeV);
// Layout for Vtranspose. For use in GEMM-II.
auto tileShapeVt = make_shape(bK{}, bN{});
auto smemLayoutVt = composition(smemLayoutV, make_layout(tileShapeVt, GenRowMajor
    {}));
```

Mathematically, given two layouts L and L' with associated functions[14] $f, f' : \mathbb{N} \to \mathbb{N}$, one can form the composition $f \circ f' : \mathbb{N} \to \mathbb{N}$ and ask whether there is a layout L'' such that its associated function f'' equals $f \circ f'$; if so, we declare the composition $L \circ L'$ to be given by L''. In code, CuTe's `composition` function deduces L'' for the programmer.[15] In the case at hand, we are precomposing `smemLayoutV` by the layout (bK, bN) : (bN, 1) to take the transposed layout.[16] Note also that `smemLayoutV` involves postcomposing a layout function with a swizzle function, and precomposition by any layout leaves this postcomposition in place.

Given that the `gmem` to `smem` copy was done with reference to `smemLayoutV`, we can then make the transposed Tensor `sVt` accessing `smem` and correctly define `tOrV`:

```
Tensor sVt = make_tensor(make_smem_ptr(shared_storage.smem_v.data()),smemLayoutVt);
// ...
Tensor tOrV = threadMma1.partition_fragment_B(sVt);
```

4.2 Reshaping Accumulator to Operand Layout

The definition of the Tensor `tOrP` featuring in GEMM-II involves a custom layout transformation method `ReshapeTStoTP`. This method takes in the accumulator Tensor `tSrS` and the Tensor `tOrS` derived from the TiledMMA object created for GEMM-II, and produces a 'reshaped' Layout suitable for defining `tOrP`:

```
Tensor tOrS = threadMma1.partition_fragment_A(sS);
auto tOrPLayout = ReshapeTStoTP()(tSrS, tOrS);
auto tOrP = make_tensor(tSrS.data(), tOrPLayout);
```

The idea is that we need to traverse the accumulator of GEMM-I, held in registers, according to the operand `ALayout` selected for TiledMma1. Since we explicitly choose this MMA atom such that operand A is in registers, the relevant `ALayout` will look like this (in `cute/atom/mma_traits_sm90_gmma.hpp`):

[14] One can always canonically extend the domain of a layout function to all of $\mathbb{N}$ by allowing the last dimension to go to ∞.
[15] One should be careful as to whether two given layouts L and L' can actually be composed, which requires satisfying certain divisibility conditions. The `composition` function has some static assert checks to rule out impermissible cases, but these aren't comprehensive.
[16] By contrast, if we used `GenColMajor` instead of `GenRowMajor`, then we would precompose with the identity function and thus do nothing.

```
// Register source layout for 16-bit value types
using ALayout_64x16 = CLayout_64x16;
// ...
using CLayout_64x16 = Layout<Shape <Shape <  _4,_8, _4>,Shape < _2,_2,  _2>>,
                             Stride<Stride<_128,_1,_16>,Stride<_64,_8,_512>>>;
```

Note in particular that the operand `ALayout` is also described by Figure 2 as with the accumulator `CLayout`, but now the operand dimensions are fixed to be 64×16, so each of the 128 threads has 8 values associated to it. To explain further, let's consider the example where tile sizes are all 128 for bM, bN, and bK. Then GEMM-I has accumulator `CLayout_64x128`. In this case, printing to console with thread 0 yields:

```
tSrS: ptr[32b](0x7f25e7fff9e0) o ((_2,_2,_16),_2,_1):((_1,_2,_4),_64,_0)
tOrS: ptr[16b](0x7f25e7fffbe0) o ((_2,_2,_2),_2,_8):((_1,_2,_4),_8,_16)
tOrPLayout: ((_2,_2,_2),_2,_8):((_1,_2,_4),_64,_8)
```

Listing 7. Tensors and Layouts for tile sizes (bM,bN,bK)=(128,128,128).

For the shapes, the first inner tuple is the value tuple, while the other two coordinates are the column and row coordinates we get from tiling the 128×128 matrix with the relevant atom shape (either 64×128 for accumulator or 64×16 for operand). The Layouts describe a logical to physical mapping where the physical addresses are in `rmem`. Observe that for `tSrS` and `tOrS`, the functions associated to these Layouts, as functions of one variable, are in fact the identity functions.[17] Indeed, we are placing the values for a given thread contiguously in `rmem`, and then tiling-to-shape in *column-major* order. On the other hand, the difference between the dimensions 64×16 and 64×128 lies in the row dimension. Therefore, we have to traverse `tSrS.data()` in a different order than that prescribed by the Layout of `tOrS` when defining `tOrP`, and thus we change the strides as indicated for `tOrPLayout`. By contrast, if we instead had bM = 64, then we wouldn't need to reshape:

```
tSrS: ptr[32b](0x7fed48fffb60) o ((_2,_2,_16),_1,_1):((_1,_2,_4),_0,_0)
tOrS: ptr[16b](0x7fed48fffc60) o ((_2,_2,_2),_1,_8):((_1,_2,_4),_0,_8)
tOrPLayout: ((_2,_2,_2),_1,_8):((_1,_2,_4),_0,_8)
```

Listing 8. Tensors and Layouts for tile sizes (bM,bN,bK)=(64,128,128).

Finally, note that the reshaping action is decoupled from downcasting the precision format; for the GEMM-II call in Listing 6, the `convert_type` call on `tOrP` will return a new Tensor with `tOrPLayout` as its Layout.

5 ONLINE SOFTMAX AND SHUFFLE REDUCTION

The online-softmax part of the inner loop in Algorithm 2 lying between GEMM-I and GEMM-II involves the row-wise computation of max and sum and altering the S matrix in place. Moreover, we need to successively rescale the matrix O after the first iteration of the loop. In code, we have:

```
if (blockIdxY == 0) { // Compute Online Softmax and NO Output Rescaling.
    onlineSoftmaxAndRescale<true, AccumType>(rowMax, rowSum, tSrS, tOrO, scale); }
else { // Compute Online Softmax and Output Rescaling.
    onlineSoftmaxAndRescale<false, AccumType>(rowMax, rowSum, tSrS, tOrO, scale); }
```

Listing 9. online-softmax in the mainloop.

[17] However, the extra semantic information encoded by the Layouts (as opposed to their associated functions) is of course important for the reshaping method.

After GEMM-I, entries of the matrix S reside in each thread's registers as per Figure 2. In particular, values per thread occur over two rows and are traversed in a replicated 'Z' pattern; for example, see Listings 7 and 8. In code, this means we need to maintain two values of max when traversing `rmem`:

```
for (int k = 0; k < NT * size<2>(VT); ++k) {
    data[n] = FragValType(AccumType(data[n]) * scaleFactor);
    max0 = cutlass::fast_max(max0, AccumType(data[n])); n++;

    data[n] = FragValType(AccumType(data[n]) * scaleFactor);
    max0 = cutlass::fast_max(max0, AccumType(data[n])); n++;

    data[n] = FragValType(AccumType(data[n]) * scaleFactor);
    max1 = cutlass::fast_max(max1, AccumType(data[n])); n++;

    data[n] = FragValType(AccumType(data[n]) * scaleFactor);
    max1 = cutlass::fast_max(max1, AccumType(data[n])); n++;
}
```

Listing 10. Scaling and computing the threadwise rowmax. `data` is `tSrS.data()`.

From Figure 2 again, we see that a row is partitioned among 4 threads (a *quad*). To assemble these threadwise rowmaxs into the actual rowmax, we could invoke atomic max operations. However, NVIDIA has also provided shuffle instructions to exchange data among threads (from Kepler architecture onwards [23]). We can use these to avoid any atomic operations and thereby avoid the memory access latency inherent to such:

```
auto max_quad_0 = ShflReduce<4>::run(max0, maxOp);
auto max_quad_1 = ShflReduce<4>::run(max1, maxOp);
mi(rowId) = max_quad_0;
mi(rowId + 1) = max_quad_1;
```

Listing 11. Two shuffle reductions for computing the max of two rows.

The `ShflReduce` method is a textbook implementation, but we include it here for completeness:

```
template <typename T> struct MaxOp {
  __device__ inline T operator()(T const &x, T const &y) {
    return x > y ? x : y;
  }
};

template <int THREADS> struct ShflReduce {
  static_assert(THREADS == 32 || THREADS == 16 || THREADS == 8 || THREADS == 4);
  template <typename T, typename Operator>
  static __device__ inline T run(T x, Operator &op) {
    constexpr int OFFSET = THREADS / 2;
    x = op(x, __shfl_xor_sync(uint32_t(-1), x, OFFSET));
    return ShflReduce<OFFSET>::run(x, op);
  }
};

template <> struct ShflReduce<2> {
```

```
  template <typename T, typename Operator>
  static __device__ inline T run(T x, Operator &op) {
    x = op(x, __shfl_xor_sync(uint32_t(-1), x, 1));
    return x;
  }
};
```

Listing 12. Computing global rowmax using `shfl` instructions. `ShflReduce<4>` computes the global max of a *quad*.

The rowsum computation proceeds similarly.

6 OVERLAPPING COPY AND GEMM

One important feature of programming on Hopper GPUs is the ability to overlap asynchronous TMA copy with asynchronous WGMMA instructions in order to hide memory latency and maximize GPU throughput. To accomplish this, CUTLASS recommends the use of software pipelining with multiple buffers for each stage of the pipeline [14].

Implementing such a scheme is costly from a software engineering point of view, as it would entail large-scale alterations to the structure of the code. Moreover, this would also increase the shared memory requirement by a significant factor. We can instead exploit the structure of the FMHA algorithm, which has two GEMMs occurring within its inner loop. Rather than pipelining for a single GEMM, we issue the loads for GEMM-II with the GEMM-I call and vice-versa. The resulting code is given in Listing 13.

```
// Copy first tile of K from GMEM to SMEM.
cfk::copy_nobar(tKgK(_, 0), tKsK(_, 0), tmaLoadK, tma_load_mbar[0]);

for (uint64_t blockIdxY = 0; blockIdxY < nTilesOfK; ++blockIdxY) {
    .....
    // Copy current tile of V from GMEM to SMEM.
    cfk::copy_nobar(tVgV(_, 0), tVsV(_, 0), tmaLoadV, tma_load_mbar[1]);
    clear(tSrS);

    // Issue GEMM-I.
    cfk::gemm_bar_wait(tiledMma0, tSrQ, tSrK, tSrS, tma_load_mbar[0]);
    .....

    // Copy next tile of K from GMEM to SMEM.
    if (blockIdxY != (nTilesOfK - 1)) {
      .....
      cfk::copy_nobar(tKgK(_, 0), tKsK(_, 0), tmaLoadK, tma_load_mbar[0]);
    }
    .....

    // ISSUE GEMM-II with Operand A from RMEM.
    cfk::gemm_bar_wait(tiledMma1, convert_type<PrecType, AccumType>(tOrP), tOrV,
                       tOrO, tma_load_mbar[1]);
}
```

Listing 13. COPY-GEMM overlapping using TMA+WGMMA

7 RESULTS

We benchmark our implementation of the forward pass of FMHA against two other versions: one shipped as part of CUTLASS 3.3 [10] and FLASH-2 from the Dao AI Lab [6]. Both of those versions uses SM80 ISA for COPY and GEMM, while we use SM90 ISA (TMA and WGMMA). All of our experiments were conducted on an H100 GPU with PCIe. We summarize the results in Figure 3. To interpret these results correctly, the reader should be aware of the following points:

- Our COLFAX kernel was compiled with CUTLASS 3.3 and CUDA 12.2.
- The CUTLASS FMHA kernel is given as example 41 in their codebase and was upstreamed from xFormers [17].[18] We used the FLASH-2 kernel that was part of release 2.3.2.
- For our program, given the head dimension we experimented with different sizes for QBLK and KBLK in the range $(64, 128) \times (64, 128)$ and chose the best performing one. See Table 1.
- The input matrices had randomly generated values drawn from the Gaussian distribution with mean 0 and variance 1, as might be produced after layer normalization.
- As in [5, §4.1], the number of floating point operations was computed in terms of the dominant contributions from the two matmuls, ignoring lower order factors like softmax.[19]
- The `-use_fast_math` NVCC compiler flag was used with all three CUDA kernels.
- Operand types were FP16 and accumulator types were FP32.[20]
- We executed the different kernels each with a large number of iterations (iterations=1000). Moreover, when rerunning the benchmarking, we observed variations of up to 1 TFLOPs/s.

HEADDIM	(64×64)	(64×128)	(128×64)	(128×128)
64	230.1	259.5	247.9	251.4
128	292.6	289.3	295.7	208.7
256	308.1	276.1	39.3	36.7

Table 1. Performance with different tile shapes for QBLK × KBLK.

We observe that our version achieves speedup close to a factor of 2.5 to 3 over CUTLASS, but only a 20%-50% improvement over FLASH-2. From our earlier experiments with GEMM [1], we expected the 128×128 tile size to deliver the best performance. However, 128×128 suffers from performance degradation due to register pressure. We observed register spills with 128×128 tile size as reported by NVCC. Additionally, GEMM-II uses both operand A and accumulator C in `rmem`. Sufficient register space is not available to keep them in `rmem` at the same time, due to which the issue of GEMM-II being serialized occurs (as reported by NVCC).

The best performing version uses one warpgroup (128 threads) per CTA, leaving the register space allowed for another warpgroup wasted.[21] In the course of conducting this research, we extended our implementation to use two warpgroups (256 threads) per every tile of operand A of WGMMA. Even though this increases the register space per CTA, the end performance was worse and we chose not to report it in this paper. A better implementation with two warpgroups is work-in-progress.

8 FUTURE WORK

The present work originated as part of a larger effort to study CUDA optimization techniques, with a focus on new capabilities afforded by the migration to Hopper architecture. From NVIDIA's H100 datasheet [15], we see that the theoretical maximum TFLOPs/s is 756 for the FP16 Tensor Cores.[22] As such, we believe that the

[18] See https://github.com/NVIDIA/cutlass/pull/992. Note that this dates from before the release of the FlashAttention-2 paper.
[19] In contrast, the CUTLASS benchmarking code sums up lower order factors as well when reporting its FLOPs/s computation. We changed this for the common benchmark.
[20] We chose these precision formats for the purposes of reporting benchmarks against the SM80 kernels without changing accuracy across kernels, but we plan to move to lower precision formats in our follow-up work.
[21] The H100 GPU has 64K registers per SM, each of them being 32-bit [16, §1.4.1.1]. The maximum number of registers per thread is 255. Using 128 threads utilizes approximately 32K registers.
[22] NVIDIA reports 1513 TFLOPs/s for FP16 with sparsity.

reservoir of applicable optimizations for the attention problem is far from exhausted. We plan to study at least the following optimizations in future work:

- Using two warpgroups (256 threads) per CTA and using a proper warp specialization (WS) scheme;
- Introducing more pipelining stages into the COPY-GEMM overlapping;
- Leveraging threadblock clusters and the new distributed shared memory for the K and V matrix COPY.

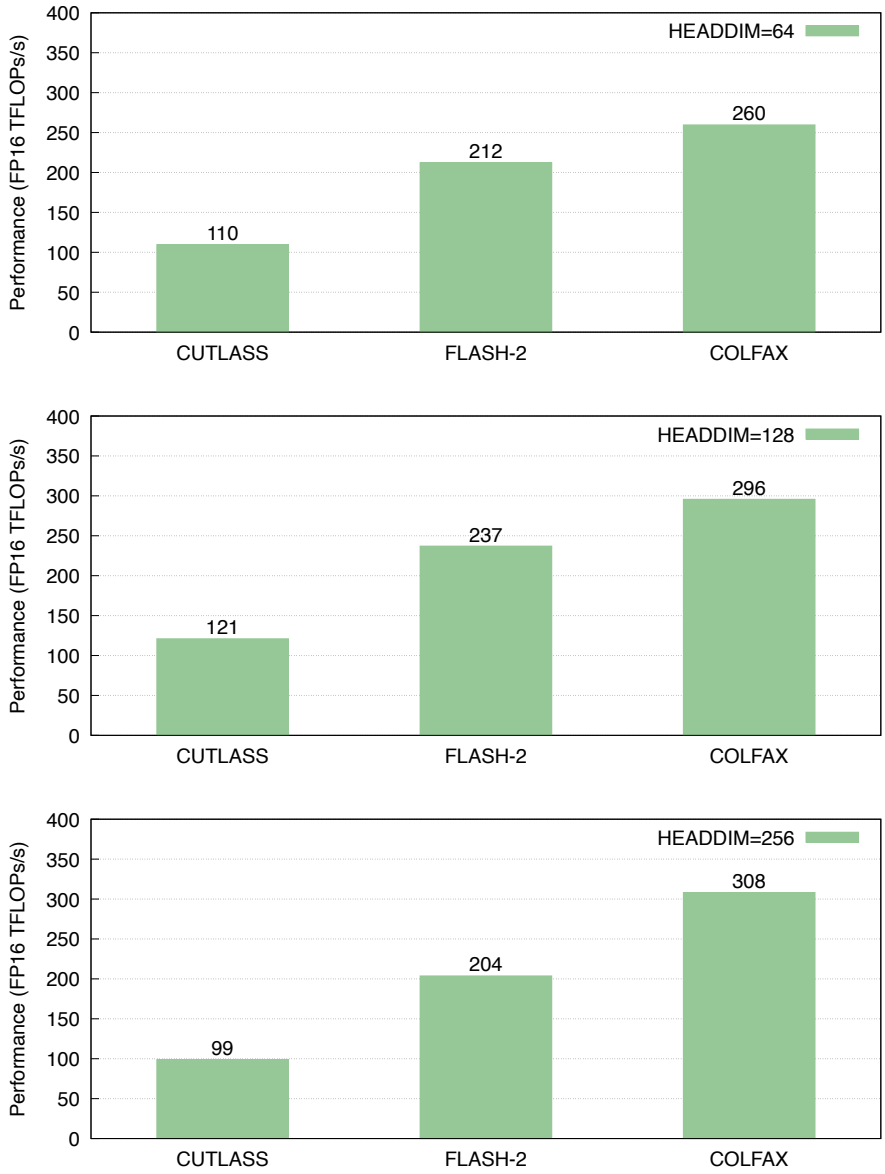


Fig. 3. Performance of FMHA (*forward* pass) on Hopper (H100 PCIe) GPU for *SEQLEN*=*KEYLEN*=4096, *head_dim*=[64, 128, 256], *num_heads*=[32, 16, 8], *batch_size*=4 and FP16 precision with FP32 accumulator. CUTLASS and FLASH-2 versions use SM80 ISA for GEMM and COPY; COLFAX version uses SM90 ISA.

We also anticipate more implementations of FMHA on next-generation GPU hardware to appear in the near future, and plan to study their methodologies when possible. In particular, we emphasize that SM90 implementations of FMHA (both forward and backward pass) have already appeared as part of LLM libraries used in production. For example, NVIDIA has provided SM90 FMHA kernels as part of its TensorRT-LLM library [21] that use TMA and warp specialization (WS), though there the kernel source code is not publically available, and OpenAI's Triton also includes an SM90 version in the latest nightly build.[23][24] Finally, innovations and advances in GPU architecture beyond Hopper should also provide fruitful ground for revisiting and improving upon the design of FMHA kernels.

REFERENCES

[1] *Developing CUDA Kernels for Accelerated Matrix Multiplication on NVIDIA Hopper Architecture using the CUTLASS Library.* Colfax Research. 2023. https://research.colfax-intl.com/nvidia-hopper-gemm-cutlass/

[2] *Language Models are Few-Shot Learners.* Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, Dario Amodei. May 28, 2020. https://arxiv.org/abs/2005.14165.

[3] *Attention Is All You Need.* Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. August 2, 2023. https://arxiv.org/abs/1706.03762.

[4] *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness.* Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, Christopher Ré. June 23, 2022. https://arxiv.org/abs/2205.14135.

[5] *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning.* Tri Dao. July 17, 2023. https://arxiv.org/abs/2307.08691.

[6] *FlashAttention — Fast and memory-efficient exact attention.* https://github.com/Dao-AILab/flash-attention

[7] *FlashAttention adoption.* https://github.com/Dao-AILab/flash-attention/blob/main/usage.md.

[8] Setting New Records at Data Center Scale Using NVIDIA H100 GPUs and NVIDIA Quantum-2 InfiniBand. Ashraf Eassa and Sukru Burc Eryilmaz. November 8, 2023. https://developer.nvidia.com/blog/setting-new-records-at-data-center-scale-using-nvidia-h100-gpus-and-quantum-2-infiniband/.

[9] *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/.

[10] *CUTLASS — CUDA Templates for Linear Algebra Subroutines.* https://github.com/NVIDIA/cutlass.

[11] *CuTe Layouts.* https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/01_layout.md.

[12] *CuTe Tensors.* https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/03_tensor.md.

[13] *CuTe's support for Matrix Multiply-Accumulate instructions.* https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/0t_mma_atom.md.

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

[15] *NVIDIA H100 Tensor Core GPU Datasheet.* https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet.

[16] *NVIDIA Hopper Tuning Guide.* https://docs.nvidia.com/cuda/hopper-tuning-guide/index.html

[17] *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.

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

[19] *Online normalizer calculation for softmax.* Maxim Milakov and Natalia Gimelshein. July 28, 2018. https://arxiv.org/abs/1805.02867.

[20] *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter.* Victor Sanh, Lysandre Debut, Julien Chaumond, Thomas Wolf. March 1, 2020. https://arxiv.org/abs/1910.01108.

[21] TensorRT-LLM 0.5.0. https://github.com/NVIDIA/TensorRT-LLM/tree/release/0.5.0.

[22] *Using Shared Memory in CUDA C/C++.* Mark Harris. January 28, 2013. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[23] *Faster Parallel Reductions on Kepler.* Justin Luitjens. February 13, 2014. https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/.

[24] *How Nvidia's CUDA Monopoly In Machine Learning Is Breaking - OpenAI Triton And PyTorch 2.0.* Dylan Patel. January 16, 2023. https://www.semianalysis.com/p/nvidiaopenaitritonpytorch.

[23] See https://github.com/openai/triton/pull/2544.

[24] We thank Harun Bayraktar and Tri Dao for alerting us to TensorRT-LLM's and Triton's respective SM90 implementations. Since our benchmarking results were conceived in the spirit of an ablation study on the impact of the TMA and WGMMA instructions on performance, we elect to defer a proper comparison with these and other SM90 kernels to future work.