Optimizing GPU kernels with cuBLAS

Manvi, Nishi, Knud, Sam

What is cuBLAS?

High Level Overview

cuBLAS = CUDA + BLAS

- BLAS = Basic Linear Algebra Subprogram
  - A fundamental linear algebra operation.
  - Includes vector addition, matrix multiplication, dot product, linear combination, etc…
- Three classes of fundamental BLAS operations/routines
  - Level 1: O(n).

$$\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$$

  - Level 2: O(n^2)

$$\mathbf{y} \leftarrow \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$$

$$\mathbf{T}\mathbf{x} = y$$

  - Level 3: O(n^3)

$$C \leftarrow \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$$

$$B \leftarrow \alpha\mathbf{T}^{-1}\mathbf{B}$$

Real numbers: alpha, beta
Vectors: x, y
Matrices: A, B, C, T
T is triangular

BLAS: https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

The cuBLAS API Specification/Organization

- Organized in the 3 levels (4 if you count helpers/utils)
- Each BLAS can be fine-tuned with hyper-parameters

$$C = \alpha \text{op}(A)\text{op}(B) + \beta C$$

This function performs the symmetric rank- $2k$ update

$$C = \alpha(\text{op}(A)\text{op}(B)^T + \text{op}(B)\text{op}(A)^T) + \beta C$$

- Methods support variable-precision:
  - **cublas<t>FUNCTION_NAME()**
    - **cublasSgemm()** : Real Single-precision (float)
    - **cublasDgemm()** : Real Double-precision (double)
    - **cublasCgemm()** : Complex, Single-precision (cuComplex)
    - **cublasZgemm()** : Complex, Double-precision (cuDoubleComplex)

How to Use cuBLAS

API Usage

What does using cuBLAS look like?

Computing C = A*B

Lab3 … Tedious!

```
#define TILE_SIZE (32)
__global__ void MatrixMultiply(float* C, float* A, float* B) {
        __shared__ float A_TILE[TILE_SIZE * TILE_SIZE];
        __shared__ float B_TILE[TILE_SIZE * TILE_SIZE];
        int tx = threadIdx.x; int ty = threadIdx.y;
        int OUT_X = blockIdx.x * blockDim.x + threadIdx.x;
        int OUT_Y = blockIdx.y * blockDim.y + threadIdx.y;
        if (OUT_X + OUT_Y * Cx < Cx * Cy) {
        printf("Out of range! Thread idx: (%d, %d)\n", tx, ty);
        return;
        }
        int NUM_TILES = (Ax + TILE_SIZE - 1) / TILE_SIZE;
        float sum = 0.0f;
        for (int i = 0; i < NUM_TILES; i++) {
        int TILE_LOC = tx + ty * TILE_SIZE;
        int A_GLOB = (tx + i * TILE_SIZE) + ty * Ax; // offset x dim
        int B_GLOB = tx + (ty + i * TILE_SIZE) * Bx; // offset y dim
        if (A_GLOB < Ax * Ay && TILE_LOC < TILE_SIZE * TILE_SIZE)
                        A_TILE[TILE_LOC] = A[A_GLOB];
        if (B_GLOB < Bx * By && TILE_LOC < TILE_SIZE * TILE_SIZE)
                        B_TILE[TILE_LOC] = B[B_GLOB];
          syncthreads();
        for (int j = 0; j < TILE_SIZE; j++) {
                        int c1 = (j + ty * TILE_SIZE < TILE_SIZE * TILE_SIZE);
                        int c2 = (tx + j * TILE_SIZE < TILE_SIZE * TILE_SIZE);
                        if (c1 && c2) {
                        sum += A_TILE[j + ty * TILE_SIZE] * B_TILE[tx + j * TILE_SIZE];
                        }
        }
        __syncthreads();
        }
        C[OUT_X + OUT_Y * Cx] = sum;
        return;
}
```

```
        int NUM_X_TILES = (Cx  + TILE_SIZE - 1) / TILE_SIZE;
        int NUM_Y_TILES = (Cy + TILE_SIZE - 1) / TILE_SIZE;
        dim3 GRID_DIM (NUM_X_TILES, NUM_Y_TILES, 1);
        dim3 BLOCK_DIM (TILE_SIZE, TILE_SIZE, 1);
        MatrixMultiply<<<GRID_DIM, BLOCK_DIM>>>(C, B, A);
        cudaDeviceSynchronize();
```

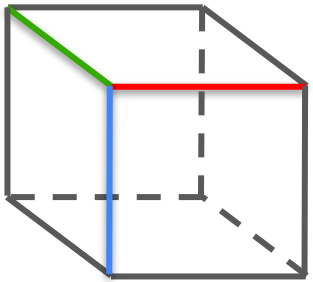
Lab3 using cuBLAS!

```
#define CUBLAS_ARGS CUBLAS_HANDLE, CUBLAS_OP_N,
CUBLAS_OP_N, By, Ax, Ay, &alpha, B, Bx, dA, Ay, &beta,
dC, Bx
      cublasStatus_t CUBLAS_ERR;
      cublasHandle_t CUBLAS_HANDLE;
      CUBLAS_ERR = cublasCreate(&CUBLAS_HANDLE);
      const float alpha = 1.0;
      const float beta = 0.0;

      CUBLAS_ERR = cublasSgemm (CUBLAS_ARGS);

      cublasDestroy(CUBLAS_HANDLE);
```

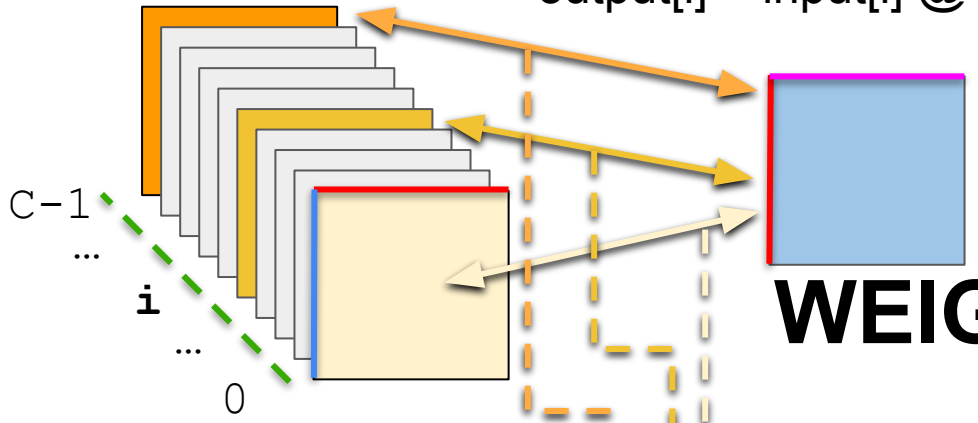
Suppose we want to compute many small matrix mults



INPUT

φ

output[i] = input[i] @ weights



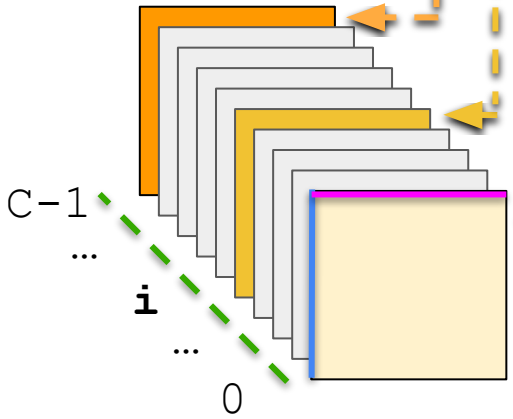
X_DIM = X
Y_DIM = A

WEIGHTS

X_DIM = A
Y_DIM = B
Z_DIM = C

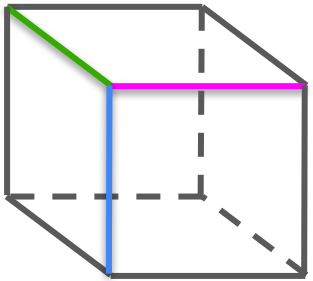
Math: C x B x A matrix

CUDA: dim3 DIM(A, B, C)



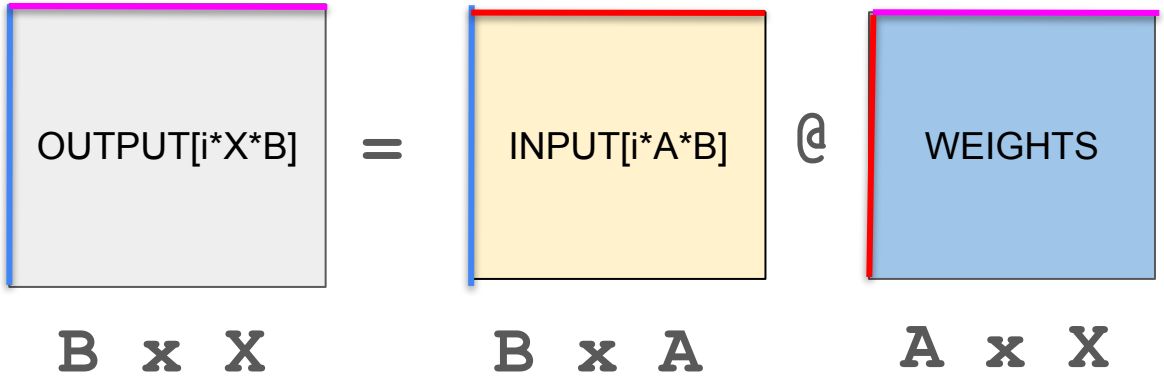
φ'

OUTPUT



I.E.

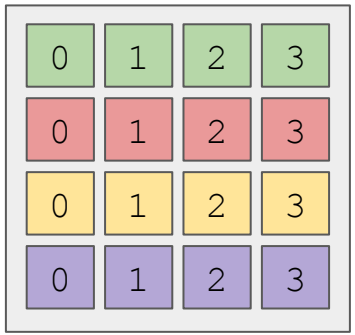
For i=0 to i=BatchSize-1



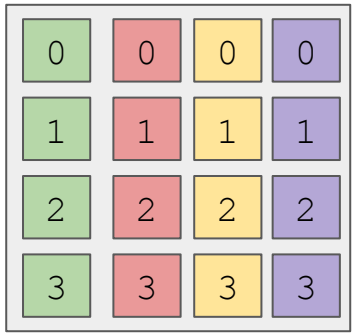
This is the correct logical operation
However, it is not this easy because of how we have laid out these matrices in Host/Device Global Memory.

Serializing Matrices for cuBLAS

- So far in class we have only been dealing with row-major convention.
- cuBLAS assumes its inputs are **column-major**!
  - This is because it wants to support Legacy Fortran API implementations.
- This is equivalent to an "implicit transpose" of the logical dimensions we imagine (and thus store) in row-major.
  - Transpose(Transpose(i, j)) = Transpose(j, i) = (i, j)

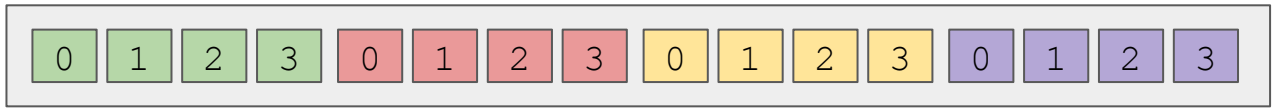


The matrix we store



The logical Matrix cuBLAS interprets from reading our RAM

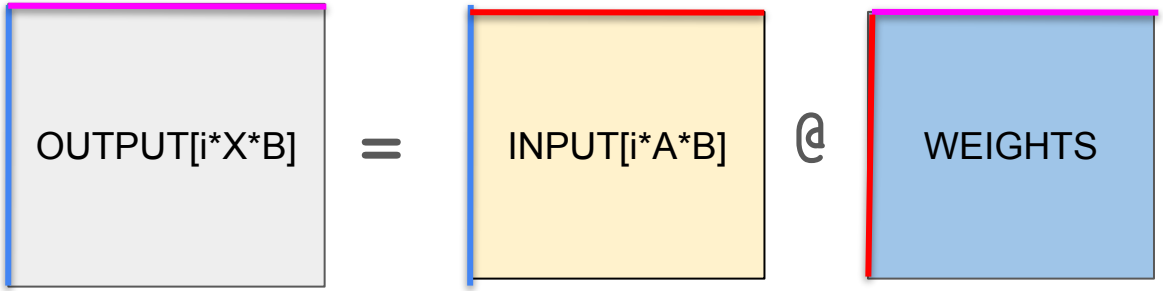
Our RAM



Correcting

For i=0 to i=BatchSize-1

To compute output of dimension (B x X) with row-major convention



(B x X) = (B x A) @ (A x X)

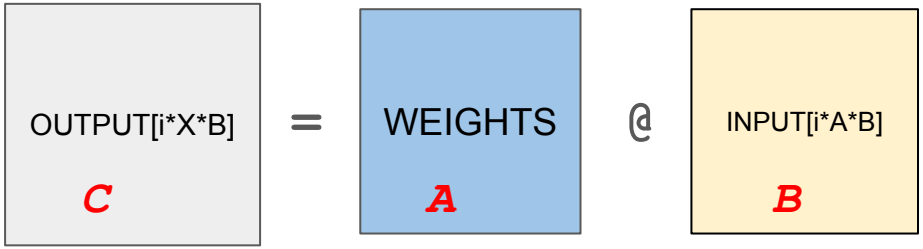
If we output (X x B) matrix in column-major, this is equivalent to a (B x X) matrix in row-major

(X x B) = (? x ?) @ (? x ?)

(X x B) = (X x A) @ (A x B)

Already implicitly transposed when read by cuBLAS!

Using cuBLAS GEMM



(X x B) = (X x A) @ (A x B)
m n lda k ldb n

transa = transb = CUBLAS_OP_N
(already implicitly transposed)

$$C = \alpha \text{op}(A)\text{op}(B) + \beta C$$

where α and β are scalars, and A, B and C are matrices stored in column-major format with dimensions $\text{op}(A)$ $m \times k$, $\text{op}(B)$ $k \times n$ and C $m \times n$, respectively. Also, for matrix A

$$\text{op}(A) = \begin{cases} A & \text{if transa} == \text{CUBLAS\_OP\_N} \\ A^T & \text{if transa} == \text{CUBLAS\_OP\_T} \\ A^H & \text{if transa} == \text{CUBLAS\_OP\_C} \end{cases}$$

Param.	Memory	In/out	Meaning
handle		input	Handle to the cuBLAS library context.
transa		input	Operation op(A) that is non- or (conj.) transpose.
transb		input	Operation op(B) that is non- or (conj.) transpose.
m		input	Number of rows of matrix op(A) and C.
n		input	Number of columns of matrix op(B) and C.
k		input	Number of columns of op(A) and rows of op(B).
alpha	host or device	input	<type> scalar used for multiplication.
A	device	input	<type> array of dimensions lda x k with lda >= max(1, m) if transa == CUBLAS_OP_N and lda x m with lda >= max(1, k) otherwise.
lda		input	Leading dimension of two-dimensional array used to store the matrix A.
B	device	input	<type> array of dimension ldb x n with ldb >= max(1, k) if transb == CUBLAS_OP_N and ldb x k with ldb>=max(1,n) otherwise.
ldb		input	Leading dimension of two-dimensional array used to store matrix B.
beta	host or device	input	<type> scalar used for multiplication. If beta == 0, C does not have to be a valid input.
C	device	in/out	<type> array of dimensions ldc x n with ldc >= max(1, m).
ldc		input	Leading dimension of a two-dimensional array used to store the matrix C.

Level 3 cuBLAS API calls

- **cublasSgemm():** Function which performs single-precision general matrix-matrix multiplication (GEMM)
  - Optimized for large matrix operations which ensures efficient GPU usage and max throughput

$$C[i] = \alpha \text{op}(A[i])\text{op}(B[i]) + \beta C[i], \text{ for i } \in [0, batchCount - 1]$$

- **cublas<T>gemmBatched():** Executes the same GEMM over a uniform batch of matrices in one call

$$C + i * strideC = \alpha \text{op}(A + i * strideA)\text{op}(B + i * strideB) + \beta(C + i * strideC), \text{ for i } \in [0, batchCount - 1]$$

cuBLAS Basic Parameters

- **cublasHandle_t:** Maintains context for all cuBLAS calls, enables multithreading and multi-GPU setups
- **alpha/beta:** Scale matrix vals in GEMM, allows asynchronous execution
- **lda/ldb/ldc:** Leading dims - define memory layout, indicate row stride between elements which is important bc cuBLAS is col-major
- **cudaStream_t:** Launches operations asynchronously, allows overlap w/ other GPU work
- **m,n,k:** Define matrix shapes
- **Pointer Mode:** Controls whether scalars are stored on host/device

cuBLAS Under the Hood

What Makes It Good

cuBLAS Heuristics

Recommender System:

- Chooses fastest implementation at runtime (before launch)
- Guarantees 93% of peak performance when using the kernel it picked (all kernel families)
- Customizable (via *cublasLtMatmulAlgoGetHeuristic*)

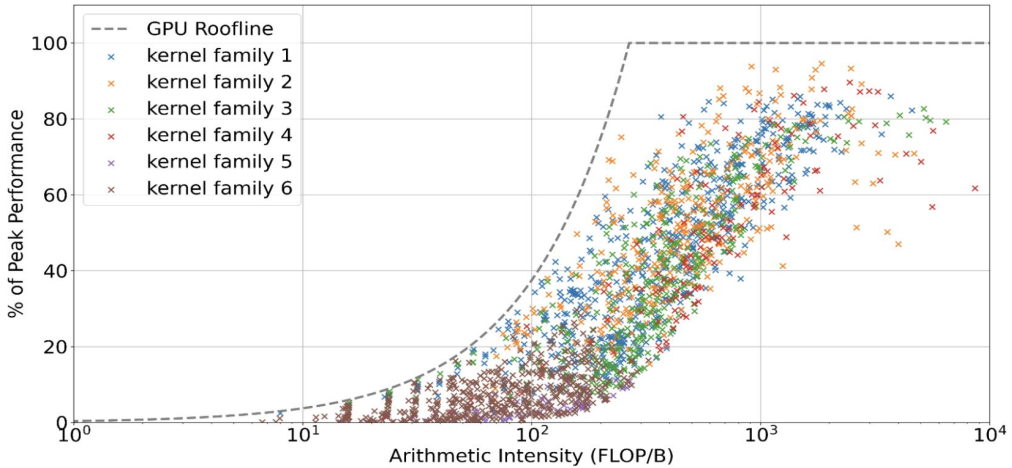


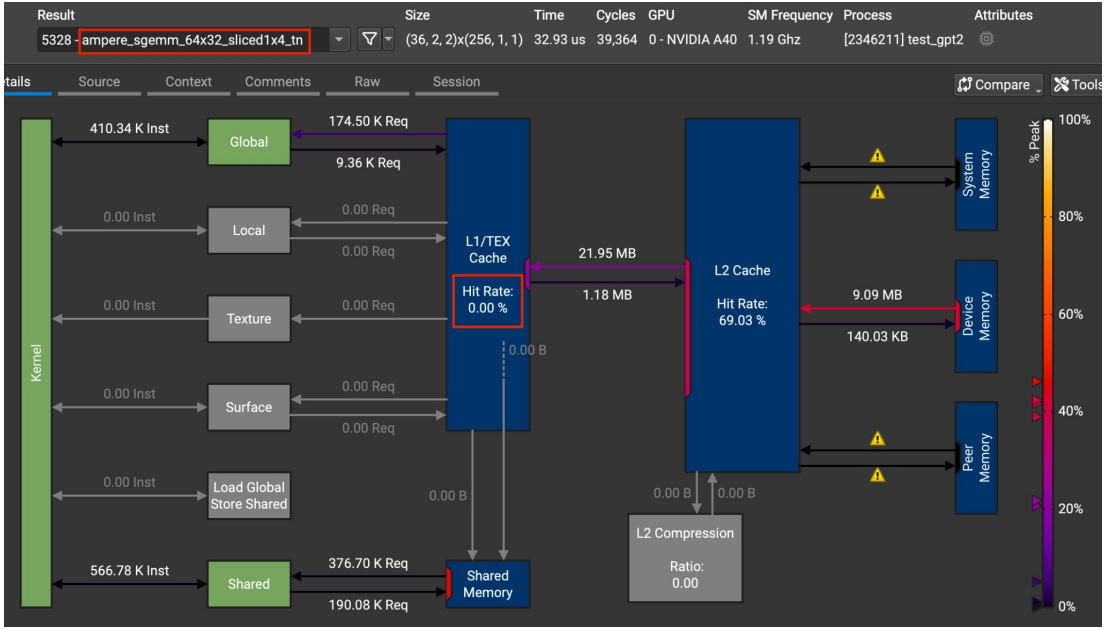
Figure 2. Sampling of various GEMMs using multiple configurations (kernels and launch parameters) in different kernel families available in the cuBLAS library plotted against the GPU roofline at a set frequency (x-axis is log-scale)

Heuristics usage in cuBLAS: https://developer.nvidia.com/blog/introducing-grouped-gemm-apis-in-cublas-and-more-performance-updates/#library_performance_and_benchmarking

cuBLAS Memory Access Patterns Example

(kernel: Ampere_SGemm_64x32_sliced1x4_tn)

- 0% L1 Cache Hit Rate (but high DRAM to L2 activity)
- Bypass L1 Cache (loads through constant cache from DRAM)




```
ff314340  @P3   LDG.E.LTC128B.CONSTANT R72, [R78]
ff314350  @!P3  MOV R72, RZ
ff314360  @P1   IADD3 R78, P2, R78, c[0x0][0x180], RZ
ff314370  @P1   IADD3.X R79, R79, c[0x0][0x184], RZ, P2, !PT
ff314380  @P4   LDG.E.LTC128B.CONSTANT R73, [R78]
ff314390  @!P4  MOV R73, RZ
ff3143a0  @P1   IADD3 R78, P2, R78, c[0x0][0x180], RZ
ff3143b0  @P1   IADD3.X R79, R79, c[0x0][0x184], RZ, P2, !PT
ff3143c0  @P5   LDG.E.LTC128B.CONSTANT R74, [R78]
ff3143d0  @!P5  MOV R74, RZ
ff3143e0  @P1   IADD3 R78, P2, R78, c[0x0][0x180], RZ
ff3143f0  @P1   IADD3.X R79, R79, c[0x0][0x184], RZ, P2, !PT
ff314400  @P6   LDG.E.LTC128B.CONSTANT R75, [R78]
ff314410  @!P6  MOV R75, RZ
```

Forum and Docs on SASS and data movement:

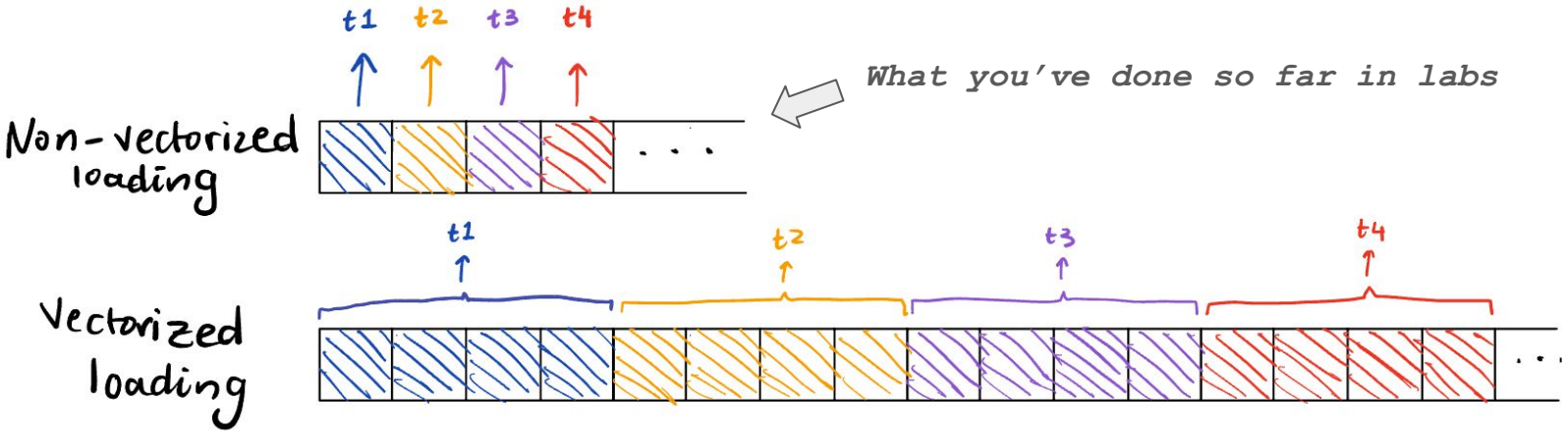
https://forums.developer.nvidia.com/t/whats-different-between-ld-and-ldg-load-from-generic-memory-vs-load-from-global-memory/40856?utm_source=chatgpt.com

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-conversion-instructions-ld%5B/url%5D

cuBLAS's Memory Accesses are NOT Coalesced??

They are, just not the way we expect.

- Each thread loads from multiple adjacent memory locations in ONE load instruction (same clock cycle)
- Memory controller will coalesce these reads too as they can still be served by the same DRAM burst



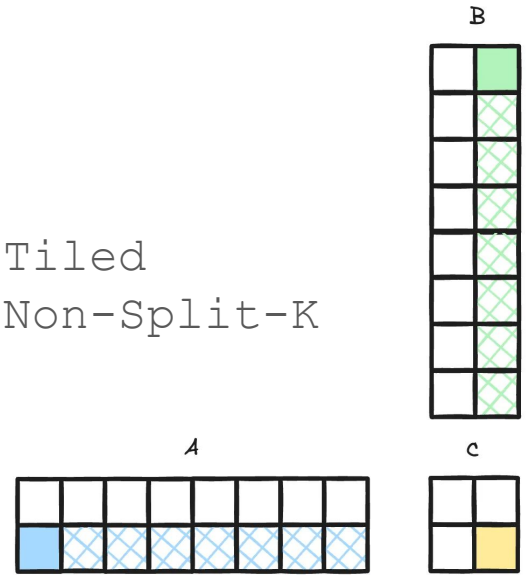
Vectorized Loading: https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/

Split-K Kernel

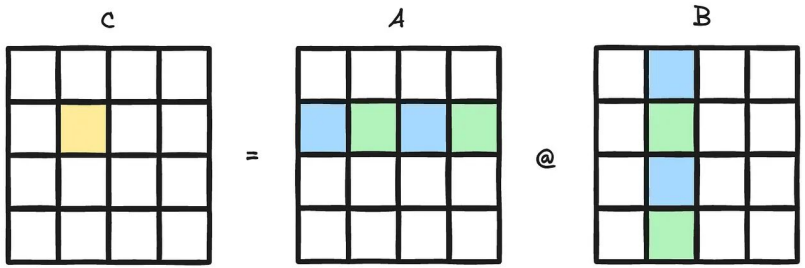
For **large** K values (hidden dimension) in matmul

cuBLAS assigns multiple threads to partially compute each output value, splitting along the K dimension

Tiled
Non-Split-K



Tiled
Split-K



Split-K usage in cuBLAS docs: https://docs.nvidia.com/cuda/cublas/#cublasltmatmulalgocapattributes-t

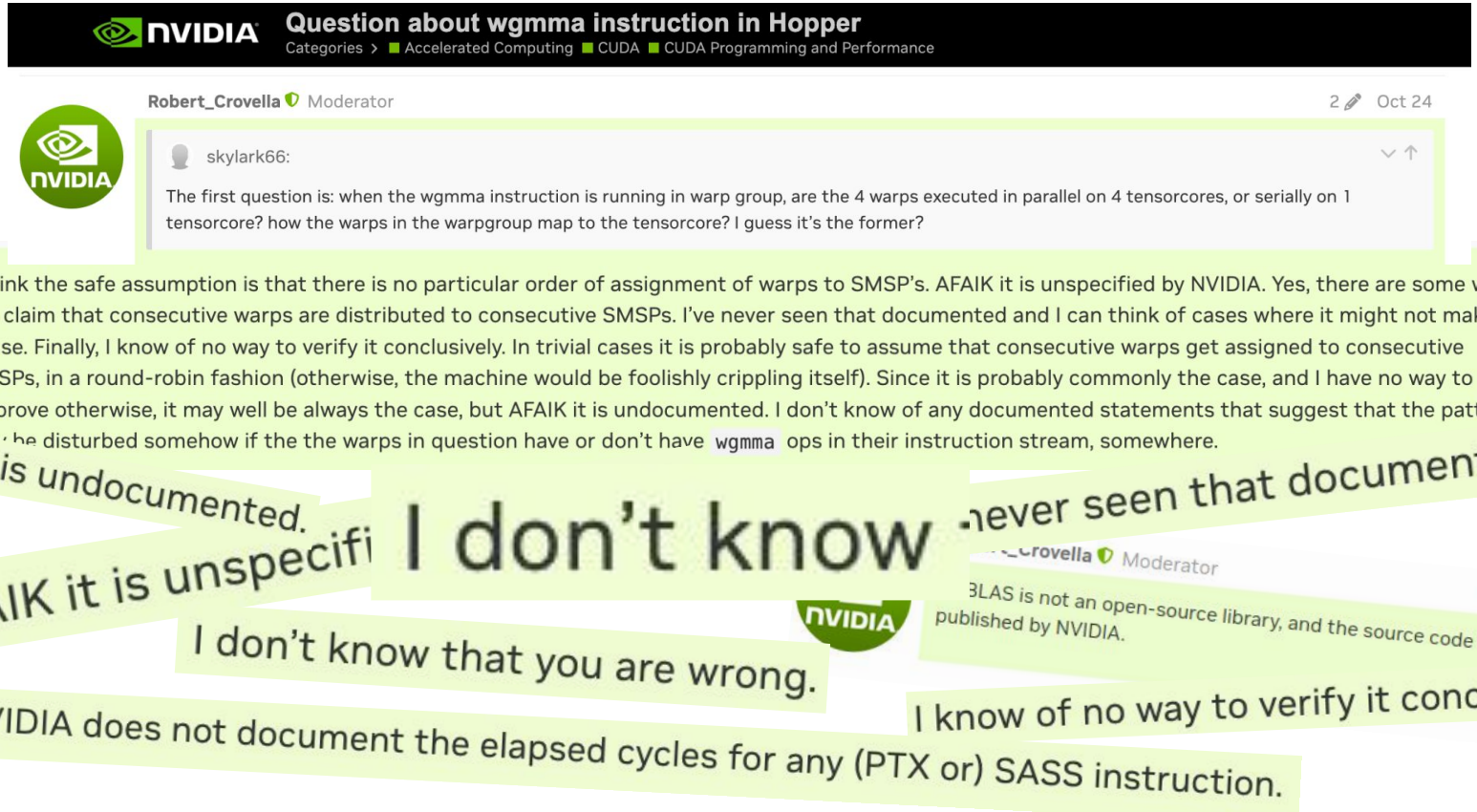
WMMA vs WGMMA in cuBLAS

- WMMA(Warp Matrix Multiply-Accumulate):
  - Volta+ Architectures (available for our A40 GPUs)
  - Used under the hood by *cublasGemmEx* and *cublasLtMatmul*
  - Benefit: No __syncthreads() as there is no inter-warp dependency
    - Threads in a warp run lock-step
  - Issue: warps work on output tiles independently, redundantly load input copies into Shared Memory even if overlapping input is loaded to use by another warp
  - Adds unnecessary DRAM traffic

- WGMMA (Warp-Group Matrix Multiply-Accumulate):
  - Hopper+ Architectures, replaces WMMA in *cublasGemmEx and cublasLtMatmul*
  - Warps cooperate as a group to compute bigger tiles (more reuse)
  - Shared memory storage redundancy is greatly reduced
  - Less DRAM traffic and less stalling(overall less loads compared to WMMA)
  - My Speculation: avoids inter-warp synchronization overheads with warp shuffles (not documented clearly)

Blog Post on WGMMA: https://research.colfax-intl.com/cutlass-tutorial-wgmma-hopper/

Disclaimer: NVIDIA Likes to Keep Secrets



NVIDIA Forum Post Source: https://forums.developer.nvidia.com/t/question-about-wgmma-instruction-in-hopper/311023/2

Disclaimer: NVIDIA Likes to Keep Secrets

NVIDIA Forum Post Source: https://forums.developer.nvidia.com/t/question-about-wgmma-instruction-in-hopper/311023/2

cuBLAS Timeline

We are here

Classic BLAS

cuBLAS 1.0 ⟶ 8.0

Baseline

✗ Tensor Cores

✗ Fusion

✗ Grouped GEMM

TensoreCore Transition 2.0

cuBLAS 10.1

Backend: cuBLASLt

✓ Tensor Cores

✓ Fusion

✗ Grouped GEMM

Current

cuBLAS 12.x+

Backend: cuBLASLt

64-bit/indexed APIs, TF32 & FP8

Fusion + extended epilogues

Tensor cores + TF32/FP32/FP8

✓ Tensor Cores

✓ Fusion

✓ Grouped GEMM

TensoreCore Transition 1.0

cuBLAS 9.0 ⟶ 10.0

Backend: wmma

✓ Tensor Cores

✗ Fusion

✗ Grouped GEMM

TensorCore Transition 3.0

cuBLAS 10.1 ⟶ 11.6

Backend: cuBLASLt

Tensor cores + TF32

✓ Tensor Cores

✓ Fusion

✗ Grouped GEMM

Further Details

cuBLAS Architecture Enabling Speed

- **Multi-kernel Architecture:** 100s of optimized kernels autotuned for specific shapes, data types, and hardware
- Peak efficiency with cuBLAS reaching 245 FLOPS/byte, saturating compute throughput
- **Warp-level tiling** distributes “chunks” of work across schedulers
- cuBLAS internally selects tensor-core-backed kernels if data alignment and math mode permits it
- **Streamed execution** and shared workspaces which allow concurrent kernel launches across streams
- **Heuristics** - cuBLAS can select optimal kernel variants (ex: with tensor cores, split-K, or vectorized loads) based on matrix dimensions, memory alignment, and compute type to maximize occupancy and throughput
- **Register blocking** - each thread computes multiple outputs using thread-local registers which minimizes redundant loads

Pseudocode

```
Initialize cuBLAS handle

For each task:
    Create a separate CUDA stream
    Assign the stream to the cuBLAS handle
    // Optionally set user-owned workspace for determinism
    Set pointer mode to device for async scalar inputs
    Configure math mode (for example, allow Tensor Core use)

    Check data is aligned (ex: 16-byte alignment) for Tensor Core eligibility
    Prepare matrices A, B, and C

    Call GEMM function:
        - cuBLAS selects optimized kernel from hundreds available
        - Applies warp-level tiling to divide matrix into tiles per warp
        - Each thread computes multiple outputs using register blocking
        - Vectorized loads reduce memory transactions (float4 aligned)
        - Heuristics choose algorithm based on shape, datatype, and layout

    //Optionally batch or stream similar operations for concurrency

Synchronize streams if necessary
Destroy cuBLAS handle
```

Useful cublas API Calls

- **cublasSgemmStridedBatched():** for GEMM operations between matrices of the same size
  - “Strided” implies matrices are evenly spaced in memory
- **cublasSgemmGroupedBatched()** for GEMM operations between matrices of differing sizes
  - “Grouped” implies the operation groups matrices of the same size together

Updated Gemm functions

- **cublasGemmEx()** is a newer cublasSgemm()
  - Allows for mixed precision workflows beyond FP32 and FP64 (FP16,BF16,INT32,etc…)
  - Allows you to control tensor cores per-call
  - Autotuning via cublasgemmAlgo_t
  - Introduced with the Pascal architecture series
- **cublasLtMatmul()** is the most involved cublas GEMM for extreme performance tuning:
  - Introduces low precision data types like FP4 and FP8
  - Allows you to fuse epilogue kernels to your operation
  - Handle optimizations like Split-K automatically
  - cublasLtMatmulAlgoGetHeuristic() to autotune

cuBlAS Performance Tuning Procedure

Tuning process is very straightforward

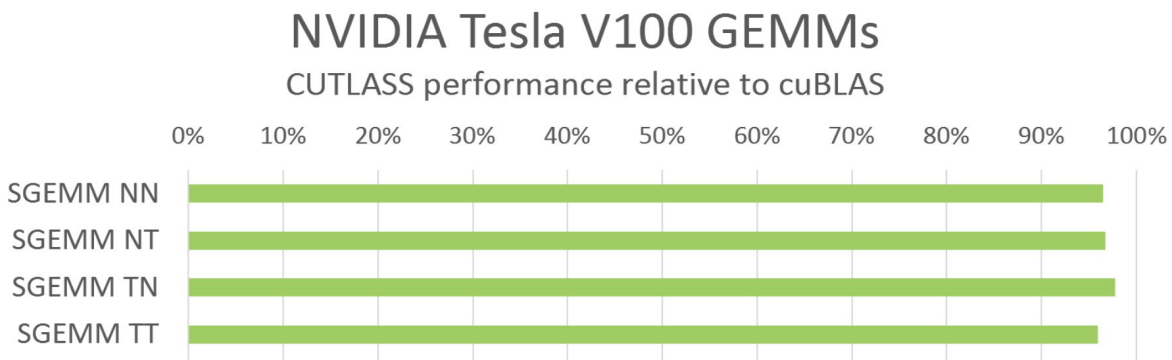
1. Choose the minimum required precision of the operation (FP32, TF32, FP8, etc…) and the appropriate api call
2. Ensure you are utilizing tensor-cores in your call
3. Choose between batching or grouping
4. Exploit stream level overlap
5. Split-K when K is large
6. Allocate a workspace buffer if needed
7. Profile using Nsight and iterate this procedure till satisfied

Thanks for listening!!!

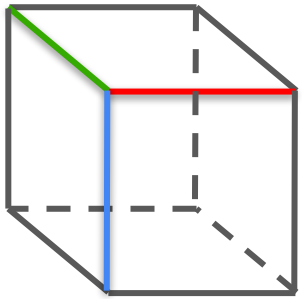
Appendix

Bonus - CUTLASS

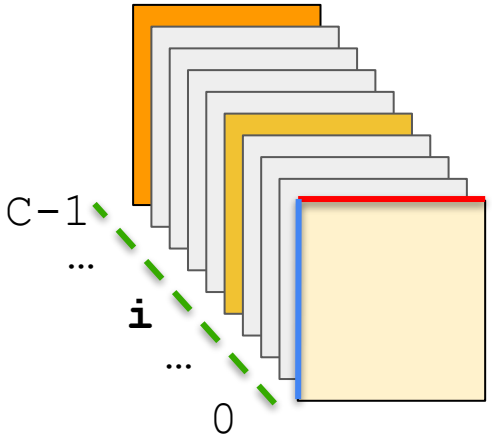
- CUTLASS is the open-source library for GEMM implemented in CUDA C++
- Provides the underlying strategies used in cuBLAS
  - Unlike cuBLAS, CUTLASS decomposes GEMM into modular building blocks
- Uses hierarchical tiling which basically breaks matmul into thread block, warp, and thread tiles
  - This optimizes data reuse at each memory level
- Supports Tensorcore Acceleration with WMMA API
- Performs comparably to cuBLAS for scalar GEMM computations
  - Main advantage: Open source so tile sizes, etc. can be tweaked
  - Drawback: Not production ready and not fully optimized



3D Input Tensor in Memory

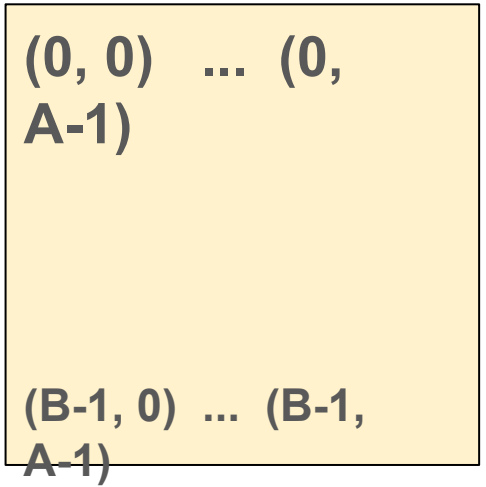


X_DIM = A
Y_DIM = B
Z_DIM = C



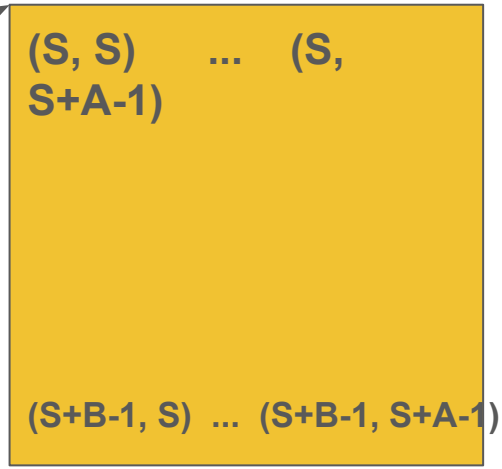
(Row, Col)

(float*)input + 0



int S = A * B * i;

(float*)input + S



- The cuBLAS API provides optimized BLAS methods.
- What are the benefits of using API?
  - You don’t have to worry about bugs in your code. We can trust the cuBLAS maintainers.
  - The cuBLAS kernels are already optimized.
  - The same BLAS kernel can compute many different things, we just have to tweak some parameters.
  - cuBLAS is written by NVIDIA engineers who keep the library updated with the release of new architectures.

Disclaimer: NVIDIA Likes to Keep Secrets

NVIDIA **Question about wgmma instruction in Hopper**
Categories > Accelerated Computing ■ CUDA ■ CUDA Programming and Performance

I don't know that you are wrong.

Robert_Crovella Moderator

skylark66:

The first question is: when the wgmma instruction is running in warp group, are the 4 warps executed in parallel on 4 tensorcores, or serially on 1 tensorcore? how the warps in the warpgroup map to the tensorcore? I guess it's the former?

I think the safe assumption is that there is no particular order of assignment of warps to SMSP's. AFAIK it is unspecified by NVIDIA. Yes, there are some who will claim that consecutive warps are distributed to consecutive SMSPs. I've never seen that documented and I can think of cases where it might not make sense. Finally, I know of no way to verify it conclusively. In trivial cases it is probably safe to assume that consecutive warps get assigned to consecutive SMSPs, in a round-robin fashion (otherwise, the machine would be foolishly crippling itself). Since it is probably commonly the case, and I have no way to disprove otherwise, it may well be always the case, but AFAIK it is undocumented. I don't know of any documented statements that suggest that the pattern may be disturbed somehow if the the warps in question have or don't have `wgmma` ops in their instruction stream, somewhere.

AFAIK it is unspecified by NVIDIA.

I've never seen that documented

I know of no way to verify it conclusively.

AFAIK, NVIDIA does not document the elapsed cycles for any (PTX or) SASS instruction.

Robert_Crovella Moderator May 31 '16

CUBLAS is not an open-source library, and the source code for it is not published by NVIDIA.

AFAIK it is undocumented.

I don't know

NVIDIA Forum Post Source:
https://forums.developer.nvidia.com/t/question-about-wgmma-instruction-in-hopper/311023/2