Accelerating 3D FFT with Half-Precision Floating Point Hardware on GPU

Students: Yanming Kang (HKUST) and Tullia Glaeser (Tulane)
Mentors: Ed D’Azevedo (ORNL), Stan Tomov (ICL, UTK), & Kwai Wong (UTK, JICS)

Research Goal

- Previous project: 1D & 2D FFT using radix 4
- **OUR** goal --
  - Accelerate
  - Larger inputs
  - 3D algorithm
  - radix 2 & radix 8
- * Using CUBLAS 10.0 and CUTLASS template library

Mixed Precision & Tensor Cores

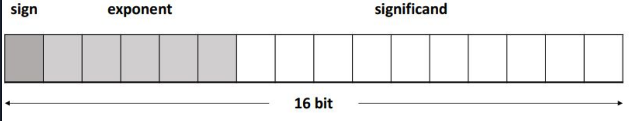


Fig. 4: Half precision floating-point number representation.

- Tensor: “a mathematical object analogous to but more general than a vector, represented by an array of components that are functions of the coordinates of a space” -- large dense matrix
- NVIDIA Volta microarchitecture ft. specialized computing units, *Tensor Cores*
- tensore core support → mixed precision -- matrix multiplication operations done w/ half-precision input data (FP16)-- the rest FFT done on single precision data (FP32)
- FP16 arithmetic enables Volta Tensor Cores which offer 125 TFlops of computational throughput on generalized matrix-matrix multiplications (GEMMs) and convolutions, an 8X increase over FP32
- Matrix entries multiplied in neural networks are small w/ respect to value of prev. Iter. → can use half precision, result is still small in val. → result accumulated to other much larger val., in single precision to avoid precision loss
- Deep neural network training = tolerant to precision loss up to certain degree

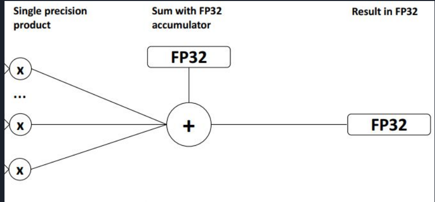


Fig. 3: FMAs in NVIDIA Tensor Cores.

Discrete Fourier Transform (DFT) & Fast Fourier Transform (FFT)

$$X(k) = \sum_{n=0}^{N-1} x(n)\, e^{-\left(i\frac{2\pi nk}{N}\right)}$$

- DFT [$O(N^2)$]: for num. computations in digital signal processing (incl fast convolution, spectrum analysis)
  - N discrete time series signals →(into) N discrete frequency components (amplitude + phase)
  - In matrix form: $X(k) = F_N x$, $F_N = e^{-2\pi ikl/N}$
- FFT [$O(N\log N)$]: Fast algorithm for DFT
  - widely used num. algorithm
  - plays vital role in many scientific and engineering applications
    1. image processing
    2. speech recognition
    3. data analysis
    4. large scale simulations
  - Maj. time in large comp. apps
  - To keep improving performance/time -- implement it on GPU

The FFT (DIT, radix-n1)

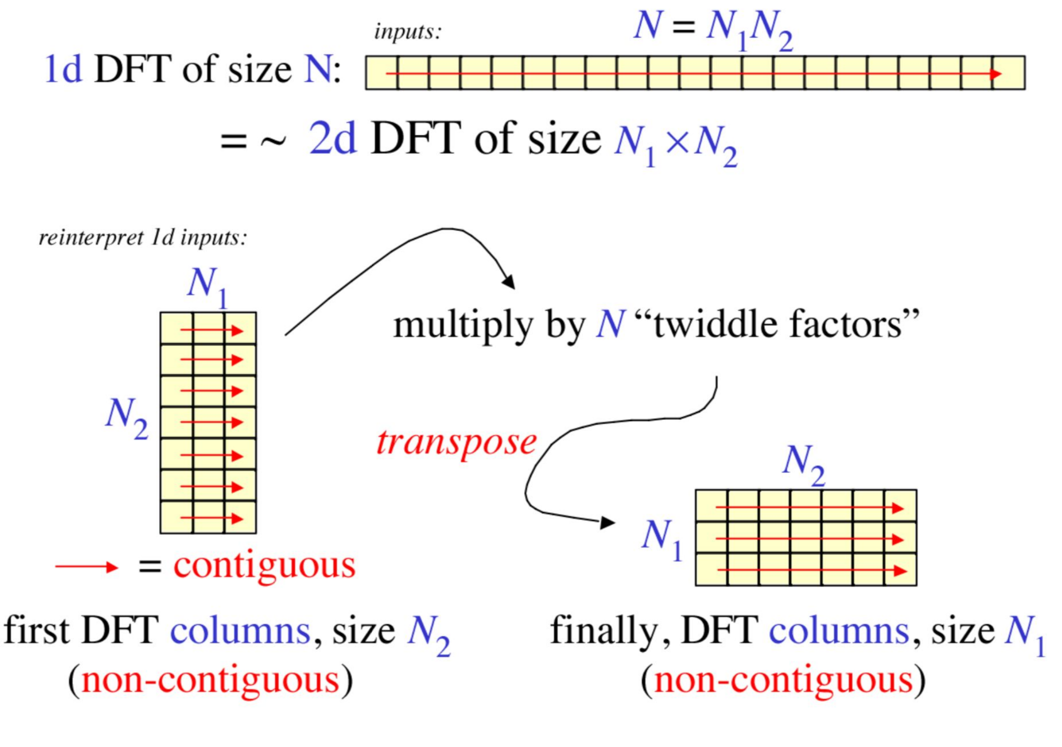


Figure 1: Cooley-Tukey FFT

The **Cooley-Tukey** Fast Fourier Transform computes the DFT with only O(N log N) operations. Cooley–Tukey algorithms *recursively re-express* a DFT of a composite size $N = N_1N_2$ by doing the following:

1. Perform N_1 DFTs of size N_2.

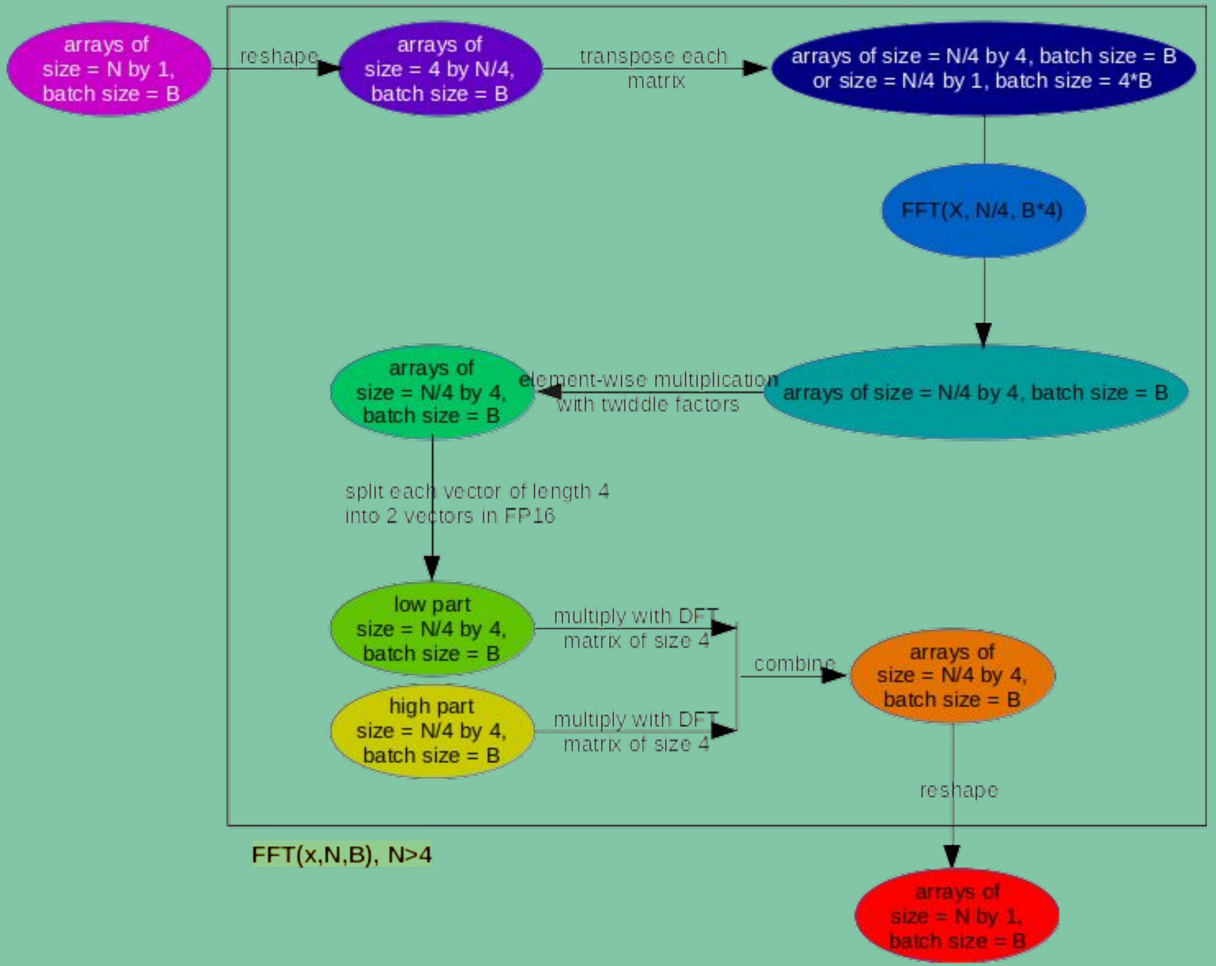
2. Multiply by complex roots of unity (often called the twiddle factors) $\{W_N[k,l] = e^{-2\pi ikl/N}\}$.

3. Perform N_2 DFTs of size N_1.

Different radixes/algorithms

- N_1 = radix → decimation in time (DIT, Cooley-Tukey)
- N_2 = radix → decimation in frequency (DIF, Sande-Tukey)
- **Radix 4** -- $N=4^v$, input sequence=x(4n), x(4n+1), x(4n+2), x(4n+3), n=0,1,...,N/4-1
  - DFT matrix F_4 = exactly representable in FP16, w/o loss of precision
    - $F_{Nreal}[l,k] = \cos(2\pi kl/N)$ N=4
    - $F_{Nimag}[l,k] = -\sin(2\pi kl/N)$
    - $F_{4real} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & -1 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix}$
    - $F_{4imag} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$
  - Ideal -- tensor cores built to perform 4x4 matrix-matrix-mult

Radix 4



arrays of size = N by 1, batch size = B
reshape
arrays of size = 4 by N/4, batch size = B
transpose each matrix
arrays of size = N/4 by 4, batch size = B or size = N/4 by 1, batch size = 4*B
FFT(X, N/4, B*4)
arrays of size = N/4 by 4, batch size = B
element-wise multiplication with twiddle factors
arrays of size = N/4 by 4, batch size = B
split each vector of length 4 into 2 vectors in FP16
low part size = N/4 by 4, batch size = B
high part size = N/4 by 4, batch size = B
multiply with DFT matrix of size 4
multiply with DFT matrix of size 4
combine
arrays of size = N/4 by 4, batch size = B
reshape
arrays of size = N by 1, batch size = B
FFT(x,N,B), N>4

Different radixes/algorithms

- **Radix 8** -- $N=8^v$
  - DFT matrix F_8 -- use previous equations w/ N=8
    - $F_{Nreal}[l,k] = \cos(2\pi kl/N)$
    - $F_{Nimag}[l,k] = -\sin(2\pi kl/N)$
    - → have rads (rad(2)/2, etc) → not exactly representable in FP16, larger error
    - Good for tensor cores too (4x4 matrix-matrix-mult)

- **Radix 2** -- $N=2^v$
  - DFT matrix F_2 = exactly representable in FP16 (w/ no complex part)

  - $F_2[l,k] = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$

  - Problem w/ tensor cores (4x4 matrix-matrix-mult) -- trick

Radix-2 Implementation (trick)

Multiplication with $F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ cannot be done directly

due to the restriction of nvcuda::wmma API.

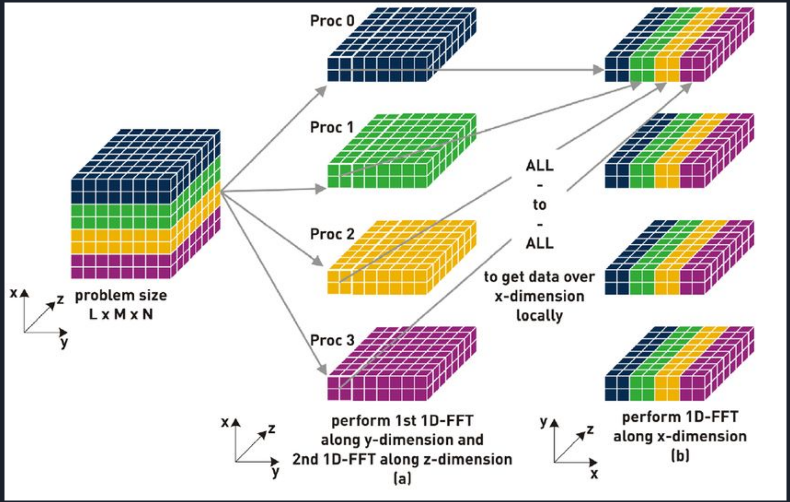
Must construct 4-by-4 matrix to use tensor cores.

$$F_{2diag} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

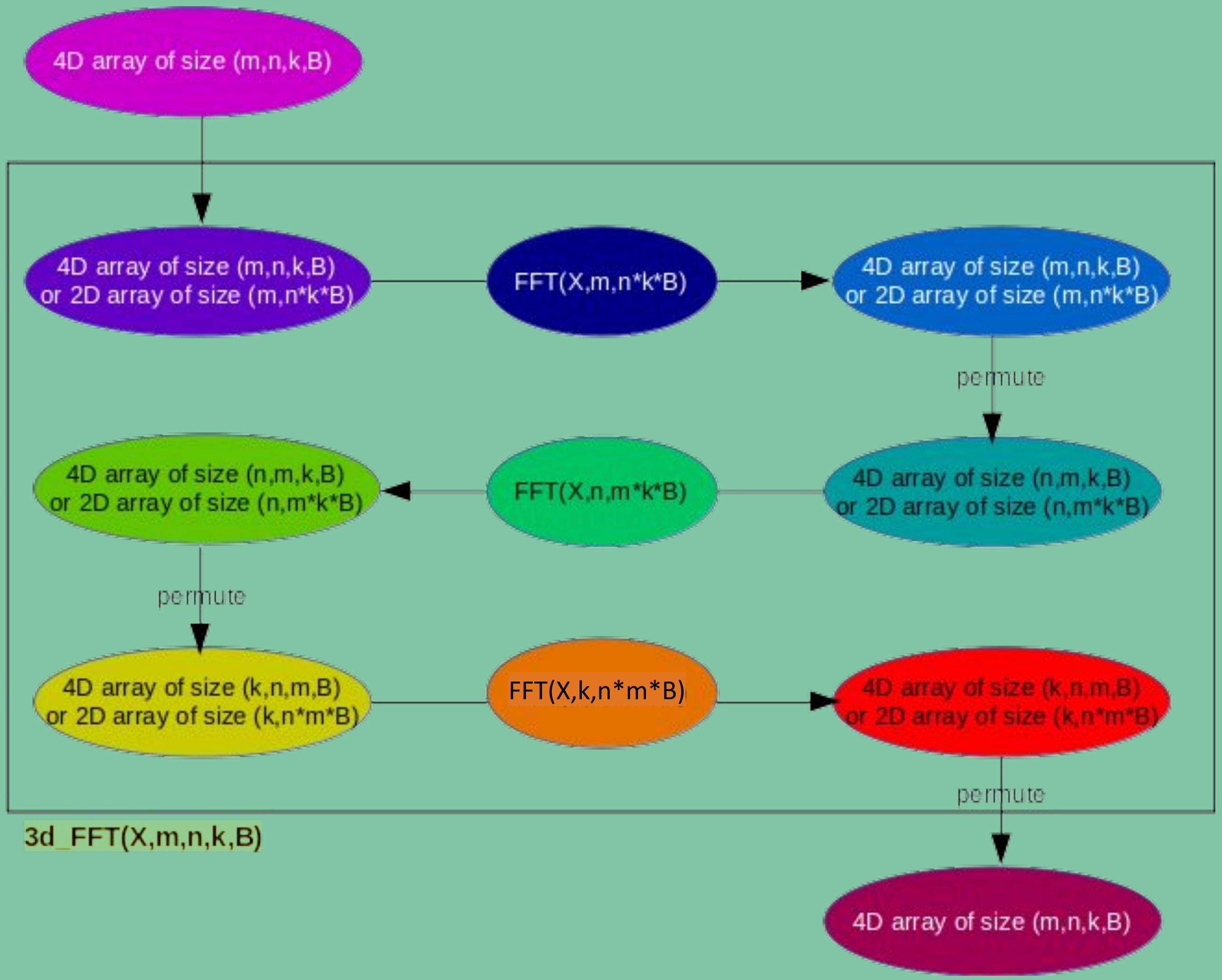
$$\begin{bmatrix} \\ X_1 \\ \\ \end{bmatrix}_{M\times 2} [F_2]; \begin{bmatrix} \\ X_2 \\ \\ \end{bmatrix}_{M\times 2} [F_2] \equiv \begin{bmatrix} \\ X_1 | X_2 \\ \\ \end{bmatrix}_{M\times 4} [F_{2diag}]$$

Implementing 1D, 2D, & 3D FFT (in radix 4)

- 1D FFT of x (described previously):
  a. x = 1D array (size = n * batch), B (4 x N/4) matrices or 1 (4 x N/4 x B) tensor (B = # of batches)
  b. Find DFT of each of those matrices
  c. Multiply by twiddle factor ($W = e^{-2\pi ikn/N}$)
- 2D FFT:
  a. x = (m x n x batch)
  b. Reshape x to be 1D array [m*n*batch, 1, 1]
  c. Call 1D FFT on it
  d. Transpose & do 1D FFT in other direction
- 3D (breakdown shown in pic):
  a. Take 1D FFT in each direction OR
  b. Take 2D FFT in 2 directions & 1D in last dir.

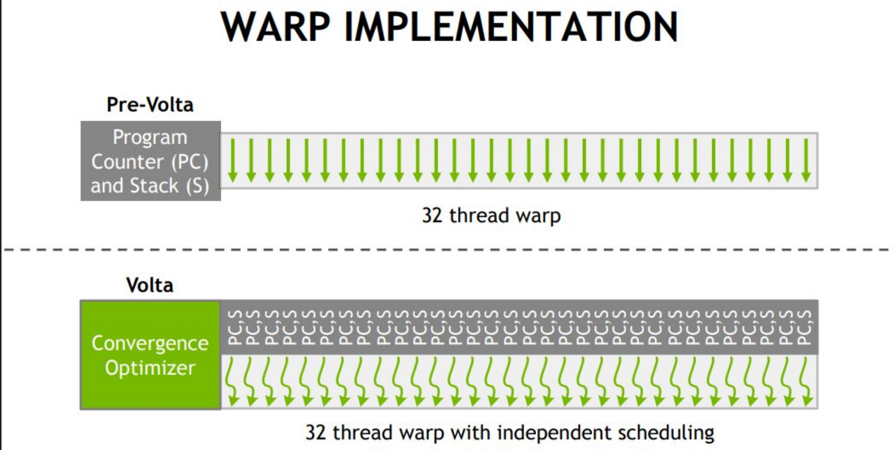


Radix 4 3D



4D array of size (m,n,k,B)
4D array of size (m,n,k,B) or 2D array of size (m,n*k*B)
FFT(X,m,n*k*B)
4D array of size (m,n,k,B) or 2D array of size (m,n*k*B)
permute
4D array of size (n,m,k,B) or 2D array of size (n,m*k*B)
FFT(X,n,m*k*B)
4D array of size (n,m,k,B) or 2D array of size (n,m*k*B)
permute
4D array of size (k,n,m,B) or 2D array of size (k,n*m*B)
FFT(X,k,n*m*B)
4D array of size (k,n,m,B) or 2D array of size (k,n*m*B)
permute
3d_FFT(X,m,n,k,B)
4D array of size (m,n,k,B)

CUDA background



There are multiple layers of abstraction:

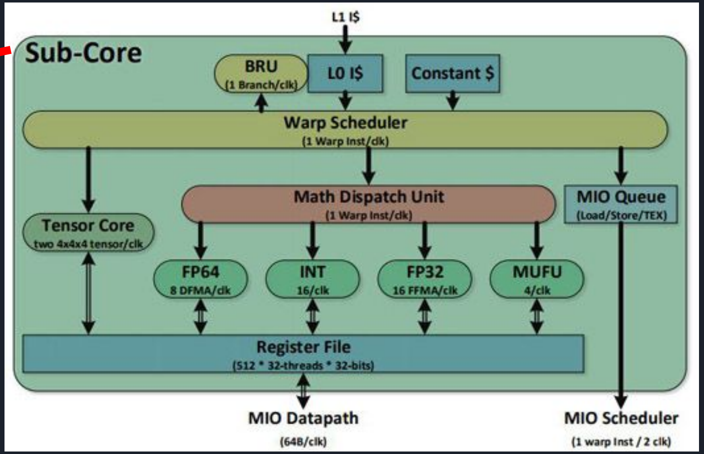
- Divides work into multiple threads (a kernel)
- threads are organized in thread blocks
- A thread block is executed by a Streaming Multiprocessor (SM)

Inside the SM, threads are launched in groups of 32 called warps.

Tensor Cores on V100

Tesla V100 with 84 SMs

Tensor Cores on V100

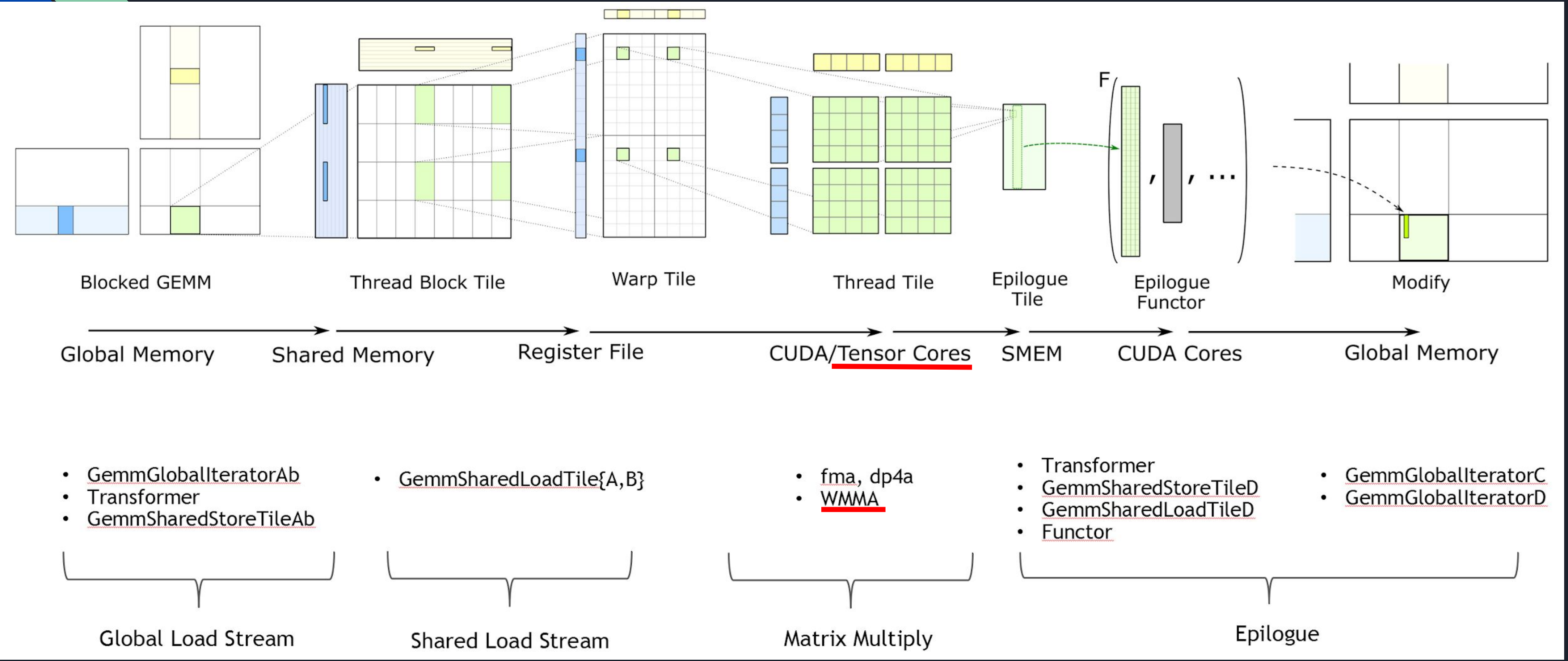


Each Tensor Core can do two half precision 4-by-4-by-4 matrix multiplications per clock cycle. -- 8x throughput than single precision

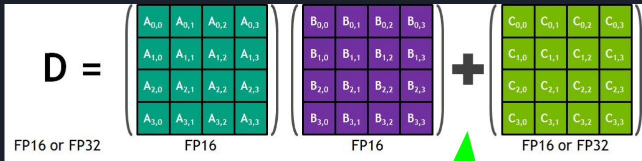
Programmers can access Tensor Cores via the Warp-Level Matrix Multiply-Accumulate (nvcuda::wmma) API

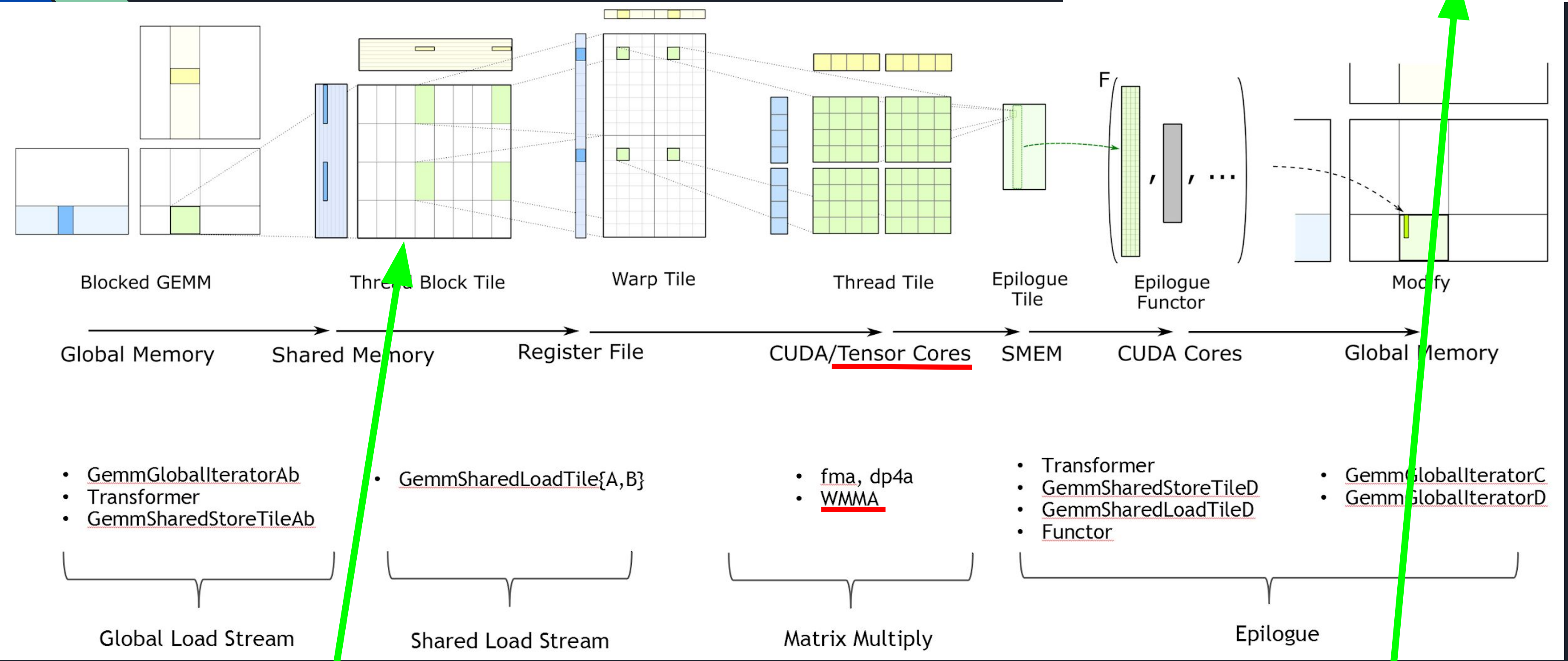
```
// Load the inputs
wmma::load_matrix_sync(a_frag, a + aRow + aCol * lda, lda);
wmma::load_matrix_sync(b_frag, b + bRow + bCol * ldb, ldb);
// Perform the matrix multiplication
wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
```

CUTLASS (CUDA Templates for Linear Algebra Subroutines)



CUTLASS (CUDA Templates for Linear Algebra Subroutines)





The threadblock's OutputTile is partitioned among the warps, and each computes a warp-level matrix product.

Why use templates

- Generic programming -- larger design space.
- Collect compile-time constants (e.g. matrix dimensions, precision) to speedup kernels.
    - Static array allocation
    - Loop unrolling
    - Function inlining
    - Constant folding
- Faster than cuBlas

Dynamic Splitting Algorithm (radix-4)

$$X_{32} = scale1 * X_{16hi} + scale2 * X_{16lo} \quad (5)$$

$$X_{32} \cdot F_4 = (scale1 * X_{16hi}) \cdot F_4 + (scale2 * X_{16lo}) \cdot F_4 \quad (6)$$

nvcuda::wmma requires A and B to be half precision when doing gemm C += A * B.
We need to split the input.

Scales are computed dynamically:

- scale1 = max(abs(X))
- X_16hi = (half) X_32 / scale1
- tmp = scale1 * (flaot)(half) X_16hi
- scale2 = max(abs(tmp))
- X_16lo = (half) tmp / scale2

1D results

Radix 2

N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
1k	2.46	3.23	$5.09 * 10^{-3}$	2.43	$7.75 * 10^{-7}$
4k	2.47	3.25	$5.04 * 10^{-3}$	2.45	$7.84 * 10^{-7}$
16k	2.52	3.27	$5.03 * 10^{-3}$	2.56	$7.83 * 10^{-7}$
32k	2.53	2.43	$5.04 * 10^{-3}$	3.22	$7.81 * 10^{-7}$
64k	2.73	3.40	$5.03 * 10^{-3}$	3.68	$7.80 * 10^{-7}$
256k	4.94	3.71	$5.04 * 10^{-3}$	8.09	$7.82 * 10^{-7}$
1024k	7.71	5.93	$5.04 * 10^{-3}$	14.42	$7.82 * 10^{-7}$
2048k	11.95	7.93	$5.05 * 10^{-3}$	24.90	$7.82 * 10^{-7}$
4096k	19.24	12.47	$5.01 * 10^{-3}$	33.93	$7.81 * 10^{-7}$
16384k	63.93	39.98	$4.55 * 10^{-3}$	111.81	$7.06 * 10^{-7}$
65536k	242.50	151.00	$2.28 * 10^{-3}$	425.91	$3.6 * 10^{-7}$

Radix 4

N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
1k	3.13	4.27	$5.10 * 10^{-3}$	2.30	$1.04 * 10^{-6}$
4k	3.00	3.40	$5.06 * 10^{-3}$	2.32	$1.05 * 10^{-6}$
16k	3.76	4.60	$1.26 * 10^{-2}$	2.42	$3.36 * 10^{-6}$
64k	2.77	3.43	$1.27 * 10^{-2}$	3.58	$3.36 * 10^{-6}$
256k	5.35	3.96	$2.94 * 10^{-2}$	7.58	$5.99 * 10^{-6}$
1024k	8.98	6.68	$2.95 * 10^{-2}$	14.03	$6.00 * 10^{-6}$
4096k	19.80	12.52	$8.70 * 10^{-2}$	30.81	$1.55 * 10^{-5}$
16384k	63.46	39.83	$9.03 * 10^{-2}$	99.20	$1.85 * 10^{-5}$
65536k	251.84	155.92	$9.03 * 10^{-2}$	381.51	$1.85 * 10^{-5}$

Radix 8

N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
4k	2.43	3.18	$5.02 * 10^{-3}$	2.28	$5.34 * 10^{-4}$
32k	2.53	2.44	$1.93 * 10^{-2}$	3.23	$2.10 * 10^{-3}$
256k	4.90	3.65	$1.94 * 10^{-2}$	6.59	$2.10 * 10^{-3}$
2048k	11.69	7.88	$8.73 * 10^{-2}$	16.72	$7.60 * 10^{-3}$
16384k	62.92	39.54	$7.89 * 10^{-2}$	79.13	$7.06 * 10^{-3}$

2D Results

Radix 4

M*N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
4k	2.45	3.23	$5.59 * 10^{-2}$	2.74	$1.43 * 10^{-5}$
16k	2.59	2.58	$5.72 * 10^{-2}$	2.98	$1.45 * 10^{-5}$
64k	2.88	3.50	$5.68 * 10^{-2}$	4.24	$1.45 * 10^{-5}$
256k	5.36	3.98	$5.70 * 10^{-2}$	8.13	$1.44 * 10^{-5}$
1024k	8.28	6.28	$1.53 * 10^{-1}$	16.72	$3.53 * 10^{-5}$
4096k	19.85	12.62	$3.27 * 10^{-1}$	42.16	$8.15 * 10^{-5}$
16384k	64.04	40.37	$2.97 * 10^{-1}$	142.80	$7.38 * 10^{-5}$
65536k	257.85	160.44	$3.34 * 10^{-1}$	597.95	$9.90 * 10^{-5}$

Radix 2

M*N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
4k	2.44	3.22	$5.71 * 10^{-2}$	2.57	$1.20 * 10^{-5}$
16k	2.94	3.48	$2.79 * 10^{-2}$	3.41	$6.21 * 10^{-6}$
64k	3.12	3.68	$3.75 * 10^{-2}$	4.81	$8.49 * 10^{-6}$
256k	6.02	4.35	$5.70 * 10^{-2}$	10.45	$1.20 * 10^{-5}$
1024k	8.46	6.18	$1.45 * 10^{-1}$	18.70	$3.05 * 10^{-5}$
4096k	19.79	12.63	$2.25 * 10^{-1}$	49.22	$4.52 * 10^{-5}$
16384k	65.39	41.73	$2.97 * 10^{-1}$	177.84	$6.38 * 10^{-5}$
65536k	240.89	156.23	$2.97 * 10^{-1}$	924.24	$6.38 * 10^{-5}$

Radix 8

M*N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
4k	2.45	3.29	$5.68 * 10^{-2}$	2.70	$7.57 * 10^{-3}$
32k	2.57	2.48	$5.69 * 10^{-2}$	3.09	$7.62 * 10^{-3}$
256k	5.12	3.77	$5.70 * 10^{-2}$	7.20	$7.60 * 10^{-3}$
2048k	11.67	7.82	$1.92 * 10^{-1}$	18.90	$2.62 * 10^{-2}$
16384k	63.79	39.98	$1.74 * 10^{-1}$	95.17	$2.38 * 10^{-2}$

3D Results

the radix 8 algorithm is the fastest but also has the largest error.

The reason is that the DFT matrix F_8 cannot be represented in fp16 with no error. The deeper the recursion goes, the larger the total error will be.

K*M*N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
16k	2.79	3.48	$2.07 * 10^{-5}$	6.30	$3.56 * 10^{-9}$
64k	3.00	3.74	$4.04 * 10^{-5}$	8.15	$7.29 * 10^{-9}$
256k	5.40	3.83	$3.87 * 10^{-5}$	10.02	$9.02 * 10^{-9}$
1024k	8.07	6.02	$2.79 * 10^{-5}$	21.08	$6.05 * 10^{-9}$
4096k	19.47	12.63	$1.87 * 10^{-5}$	60.98	$3.92 * 10^{-9}$
16384k	68.51	40.11	$1.41 * 10^{-5}$	230.54	$2.92 * 10^{-9}$
65536k	259.13	148.14	$1.23 * 10^{-5}$	956.44	$2.62 * 10^{-9}$

Table 7: 3-D radix-2 FFT results

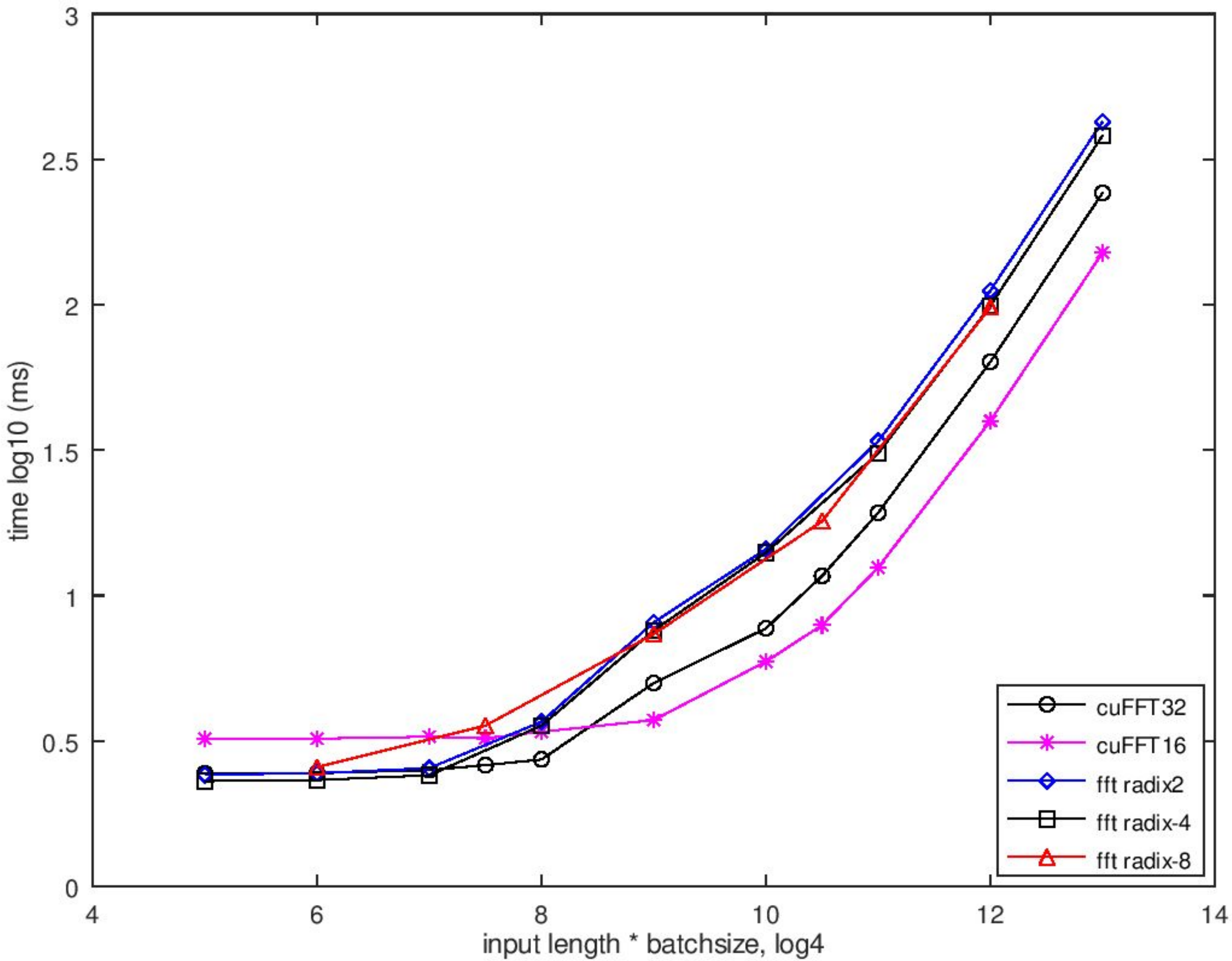
K*M*N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
16k	2.82	3.51	$2.25 * 10^{-5}$	5.96	$4.08 * 10^{-9}$
64k	2.97	3.57	$4.14 * 10^{-5}$	7.13	$9.65 * 10^{-9}$
256k	5.51	4.02	$9.76 * 10^{-5}$	8.57	$2.47 * 10^{-8}$
1024k	8.46	6.31	$2.62 * 10^{-5}$	18.37	$6.05 * 10^{-9}$
4096k	19.99	12.83	$6.56 * 10^{-6}$	49.12	$1.61 * 10^{-9}$
16384k	69.67	40.56	$1.39 * 10^{-5}$	177.62	$3.56 * 10^{-9}$
65536k	269.77	153.99	$3.69 * 10^{-5}$	727.22	$8.58 * 10^{-9}$

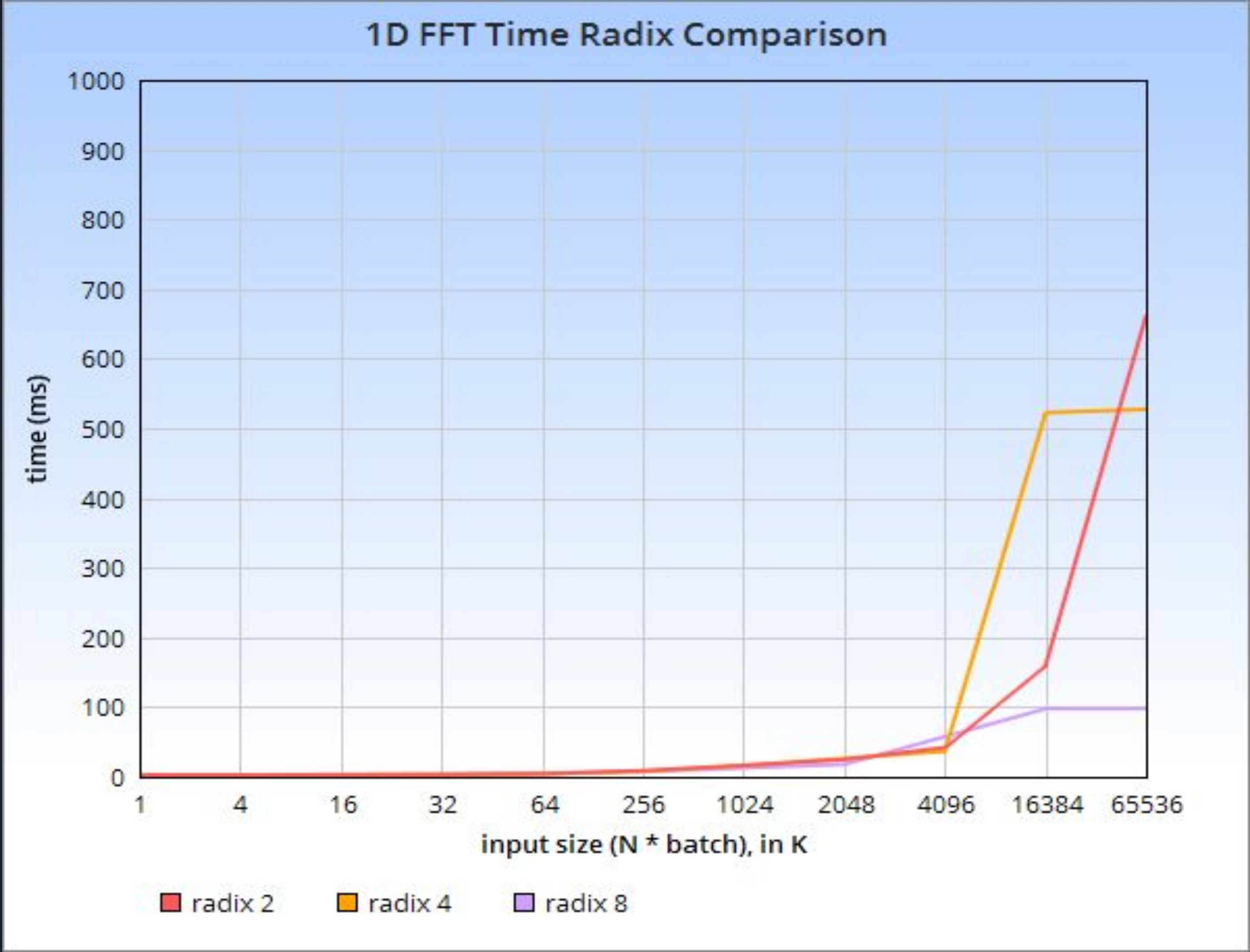
Table 8: 3-D radix-4 FFT results

K*M*N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
32k	2.77	2.66	$2.60 * 10^{-4}$	4.10	$3.41 * 10^{-5}$
256k	5.34	3.88	$3.23 * 10^{-5}$	7.48	$2.82 * 10^{-6}$
2048k	11.95	8.11	$4.62 * 10^{-6}$	20.43	$3.38 * 10^{-7}$
16384k	67.82	39.75	$1.39 * 10^{-5}$	113.30	$1.85 * 10^{-6}$

Table 9: 3-D radix-8 FFT results

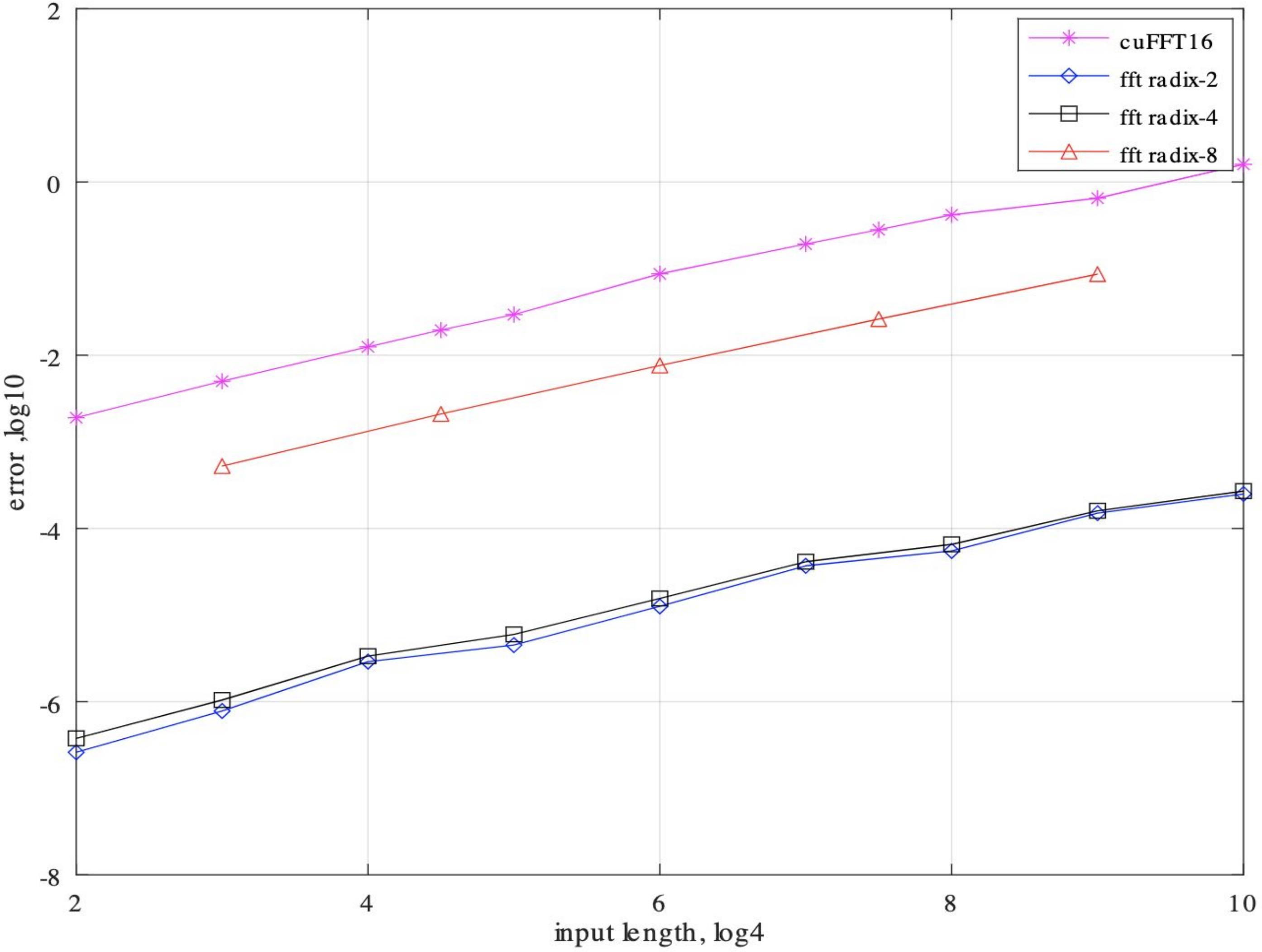
Time comparison

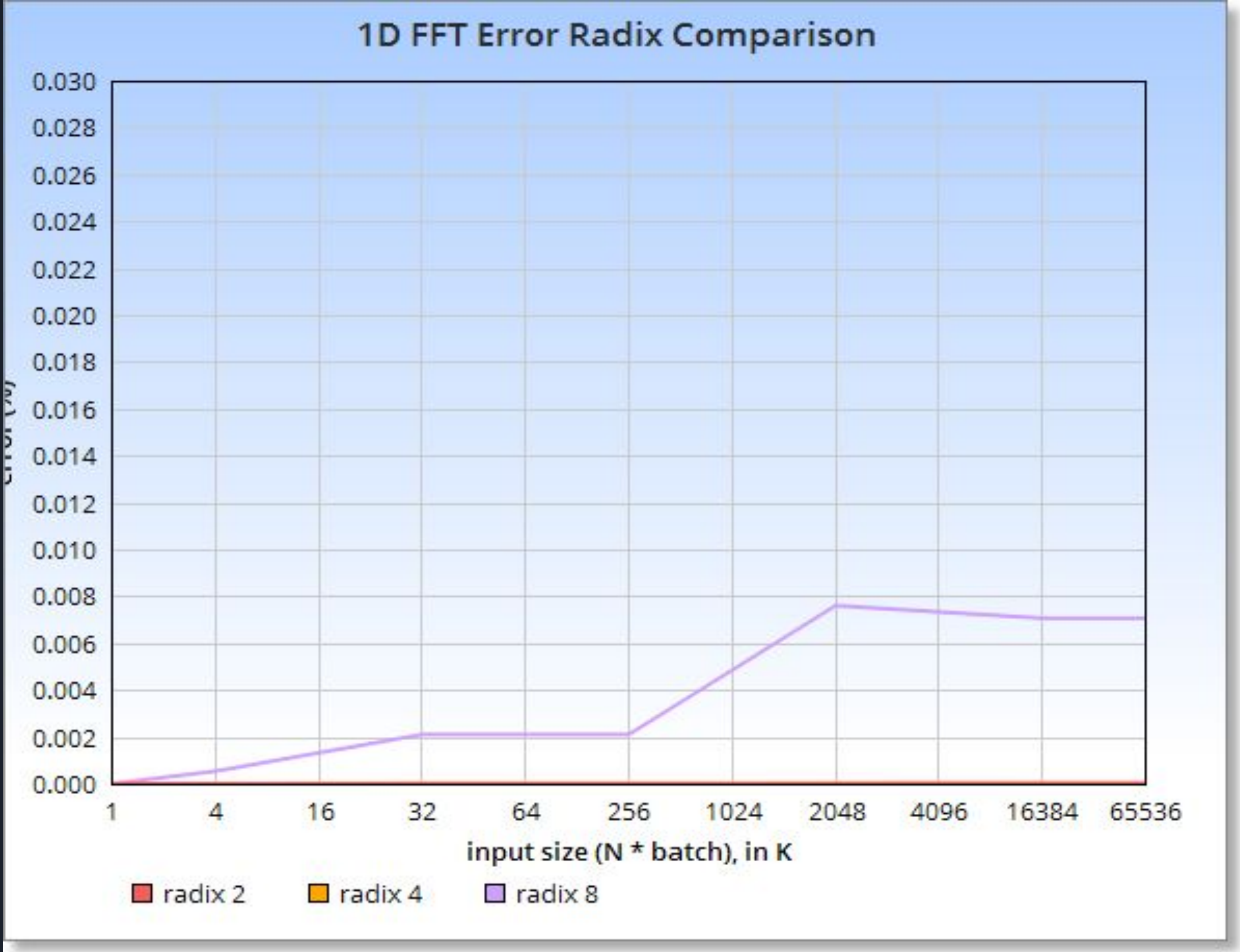




radix 8 = fastest

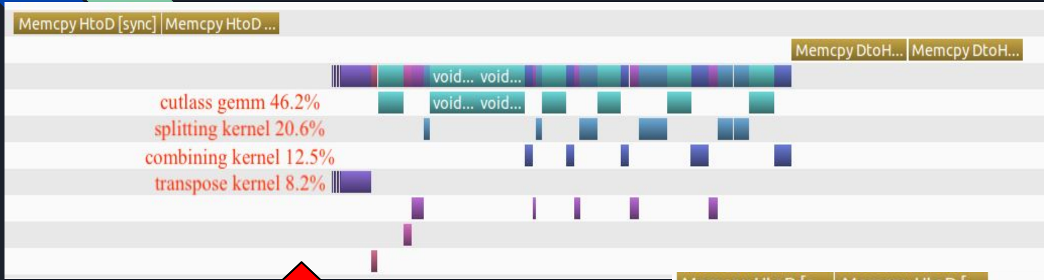
Error comparison





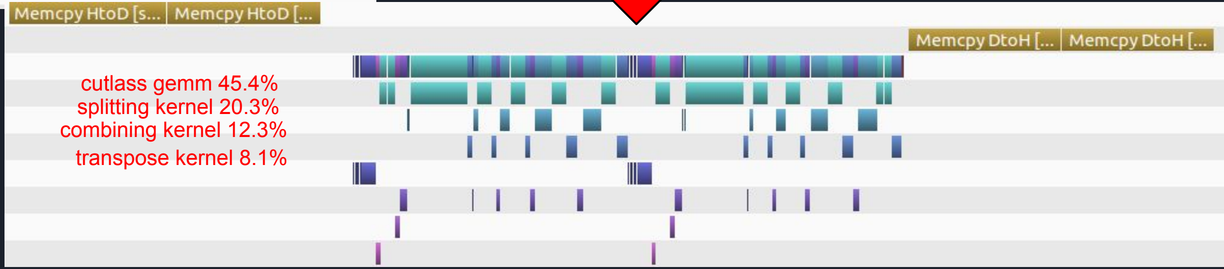
radix 8 = largest
error -- still small

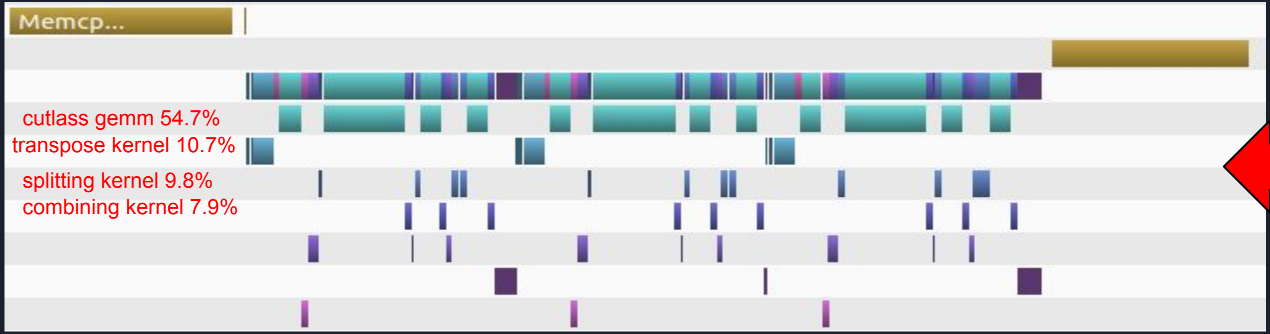
NVIDIA Visual Profiler Analysis of Radix-4



1D, N*batch = 1,048,576

2D, N*M*batch = 67,108,864





3D, K*N*M*batch = 67,108,864

In the Future

- Split-radix algorithm, combining 2+ different radices. eg. combine radix-4 and radix-8 algorithms
- Manipulate the code / use different memory allocation tricks → to take larger input sizes
- Hide memory latency by overlapping FFT and memcpy (H2D, D2H), by splitting batch size and using multiple streams.
- Provide support to inputs of composite sizes (now only powers of 2, 4, 8).
- Integer approximation of F_8

References

- “Fast Fourier Transform (FFT).” *CMLAB*, CMLaboratory, www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html
- Markidis, Stefano, et al. *NVIDIA Tensor Core Programmability, Performance & Precision*. pp. 1–12, *NVIDIA Tensor Core Programmability, Performance & Precision*
- Sorna, Anumeena, et al. *Accelerating the Fast Fourier Transform Using Mixed Precision on Tensor Core Hardware*. National Science Foundation (NSF), 2018, *Accelerating the Fast Fourier Transform Using Mixed Precision on Tensor Core Hardware*, www.jics.utk.edu/files/images/recsem-reu/2018/fft/Report.pdf
- VanderPlas, Jake. “Understanding the FFT Algorithm.” *Pythonic Perambulations*, 28 Aug. 2013, jakevdp.github.io/blog/2013/08/28/understanding-the-fft/