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

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

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

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

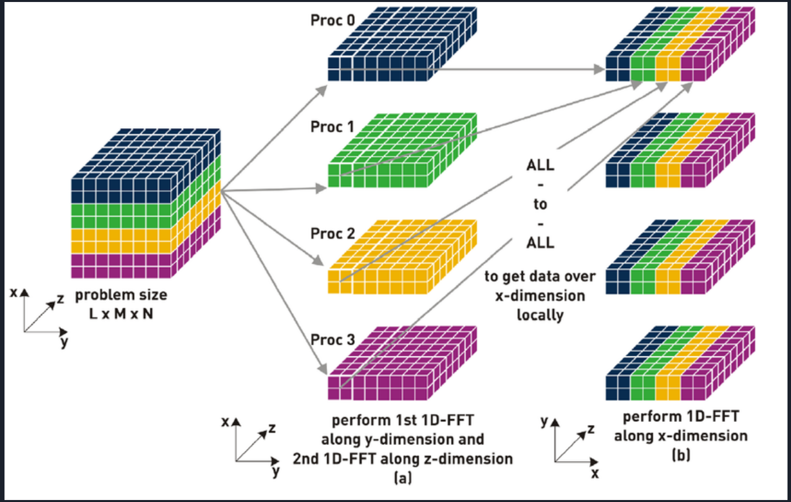
- DFT [O(N^2)]: for num. computations in digital signal processing (incl fast convolution, spectrum analysis)
  - N discrete time series signals →(into) N discrete frequency components (amplitude + phase)
- FFT [O(NlogN)]: Fast algorithms for DFT -- widely used num. Algorithm -- plays vital role in many scientific and engineering applications (image processing, speech recog., data analysis, large scale simulations
  - Maj. time in large comp. apps
  - Cooley + Tukey Algorithm:
    1. Symmetry of DFT: $X_{N+k} = X_k =$
    2. Divide DFT alg. into odd + even p
       - → halved the computations to be O(2M) where M is half of N → O(N)
    1. Keep doing this recursively → halves computation cost every time → O(NlogN)
  - To keep improving performance/time -- implement it on GPU

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

$$= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i\,2\pi\,k\,m\,/\,(N/2)} + e^{-i\,2\pi\,k\,/\,N} \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i\,2\pi\,k\,m\,/\,(N/2)}$$

Implementing 1D, 2D, & 3D FFT

- 1D FFT of x:
  a. x = 1D array, 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 = $e^{-i2\square kn/N}$)
- 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.
- MATLAB + **CUDA**
  a. Currently use CUBLAS/CUTLASS and Radix-4


problem size
L x M x N
Proc 0
Proc 1
Proc 2
Proc 3
ALL - to - ALL
to get data over x-dimension locally
perform 1st 1D-FFT along y-dimension and 2nd 1D-FFT along z-dimension
(a)
perform 1D-FFT along x-dimension
(b)
x
y
z

This image cannot currently be displayed.

Mixed Precision & Tensor Cores

- 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

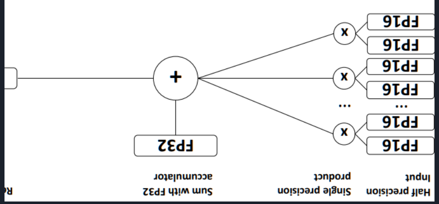
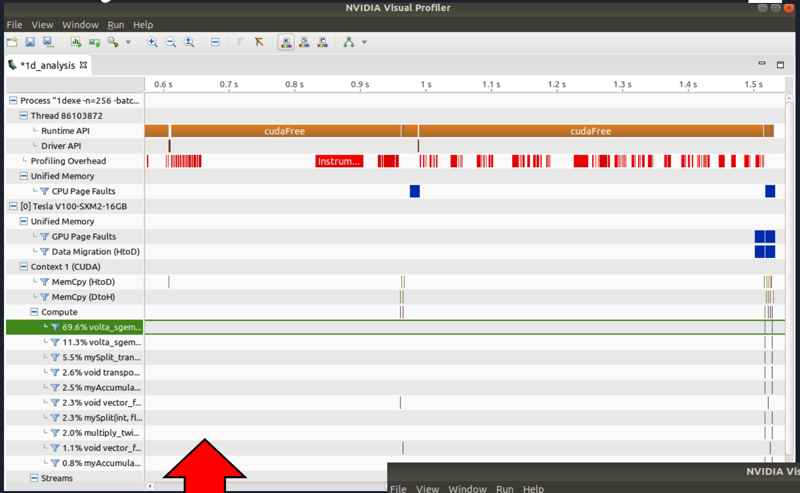
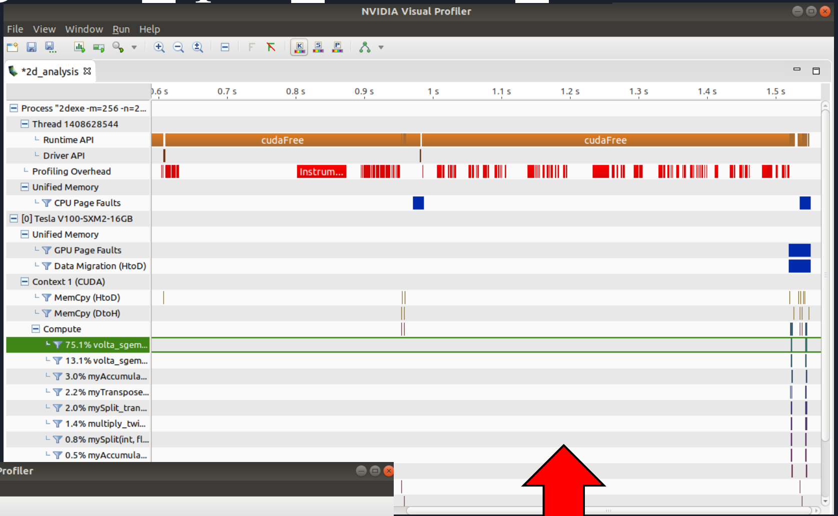


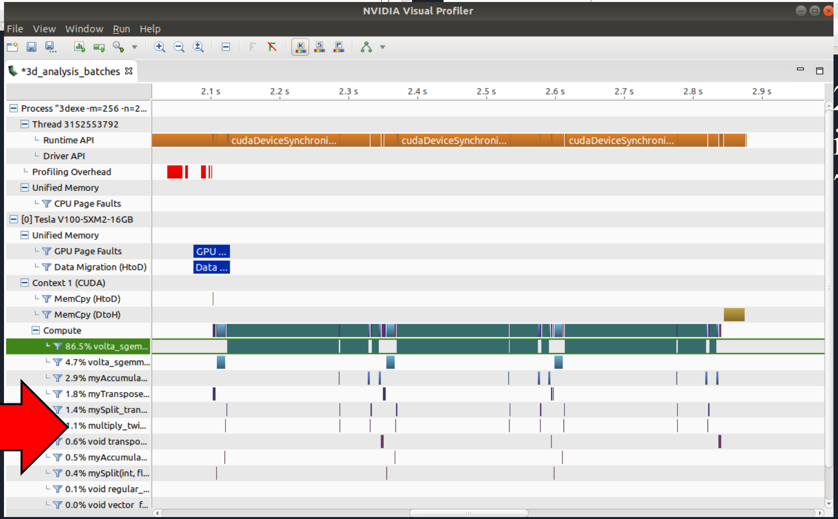
Fig. 3: FMAs in NVIDIA Tensor Cores.

Inefficiency with Transform -- volta_sgemm_fp16_128x64_nn





1D, n=256, batch=1, iter=1
69.6%

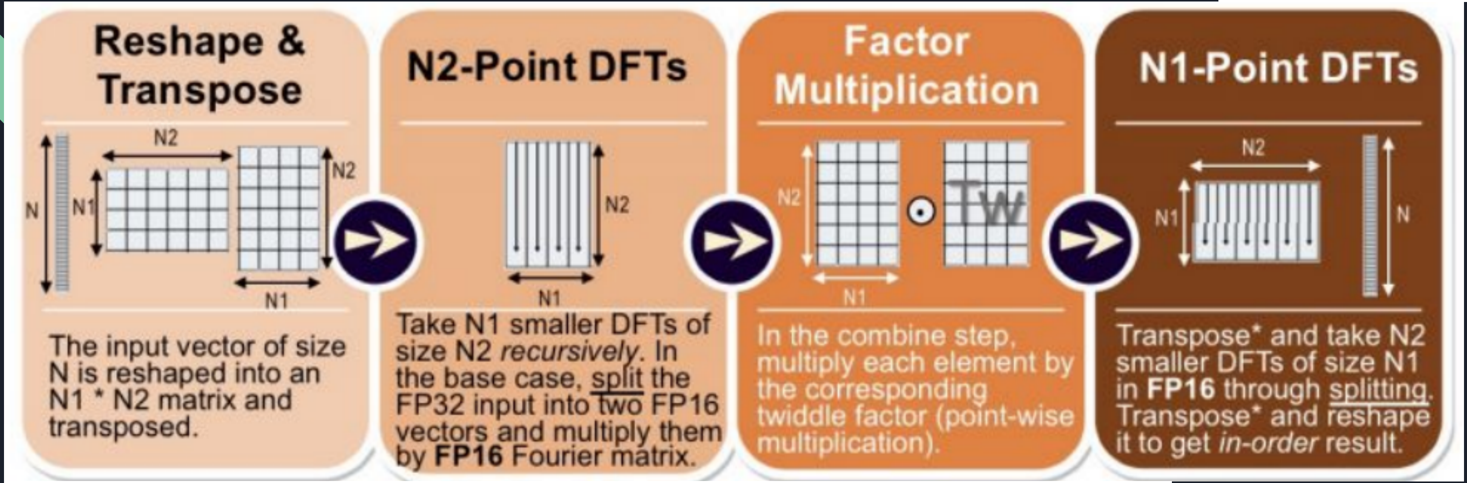


2D, m=256, n=256, batch=1, iter=1
75.1%

3D, m=256, n=256, k=256, batch=1, iter=1
86.5%

The FFT (radix-n1) in matrix form

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



Reshape & Transpose

The input vector of size N is reshaped into an N1 * N2 matrix and transposed.

N2-Point DFTs

Take N1 smaller DFTs of size N2 *recursively*. In the base case, split the FP32 input into two FP16 vectors and multiply them by **FP16** Fourier matrix.

Factor Multiplication

In the combine step, multiply each element by the corresponding twiddle factor (point-wise multiplication).

N1-Point DFTs

Transpose* and take N2 smaller DFTs of size N1 in **FP16** through splitting. Transpose* and reshape it to get *in-order* result.

$$Y = F_N X = (W_N \times F_{n2} \cdot \mathrm{reshape}(X, n1, n2)^T) F_{n1}$$

$$\text{where } (W_N)_{k,l} = e^{-\frac{2\pi ikl}{N}}$$

We use n1 = 4 since F4 =[1 1 1 1 can be stored in fp16 with no error.

1 1 1 1

1 i -1 -i

1 -1 1 -1

The algorithm

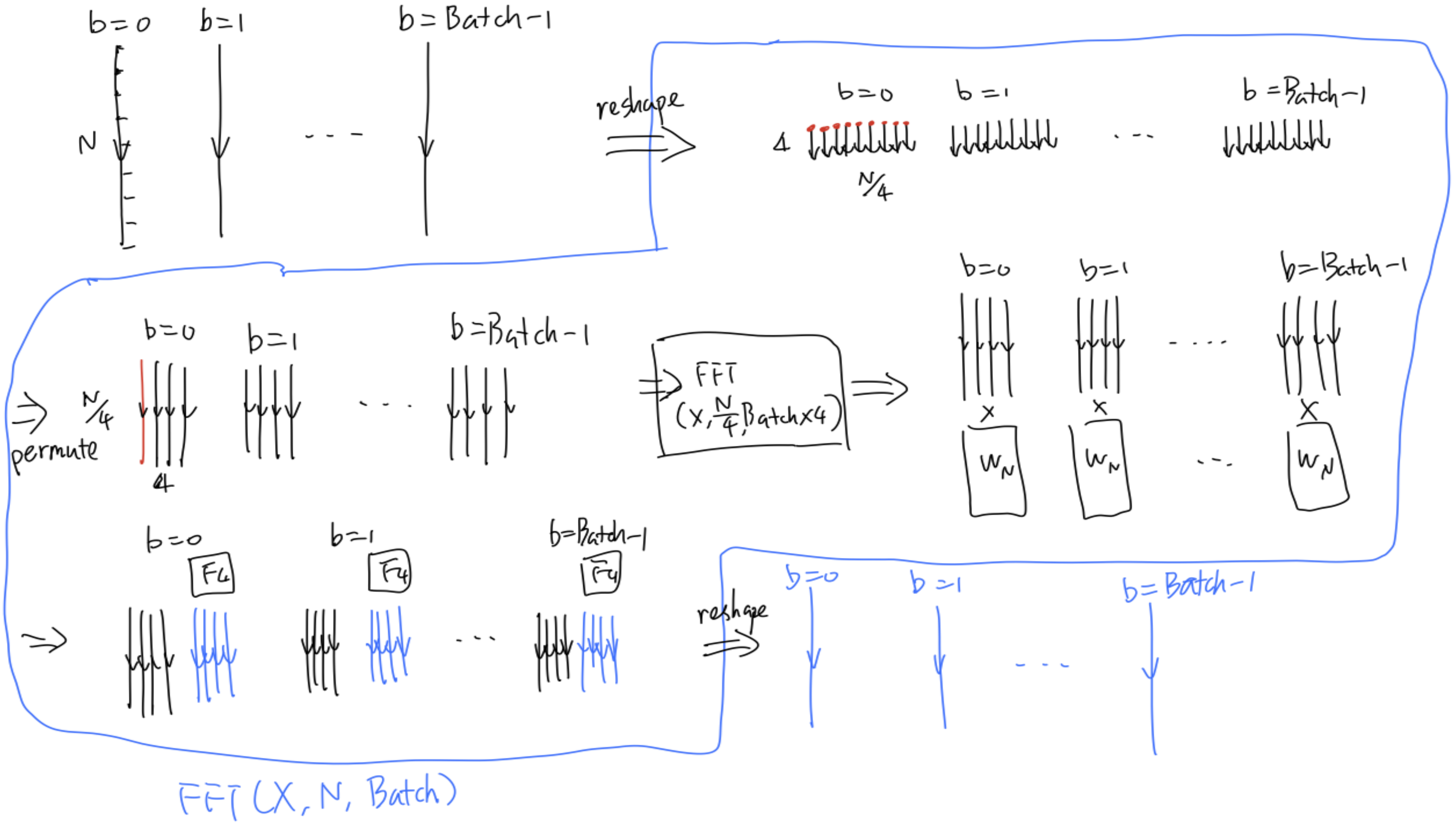
```
//Batched 1d FFT of length N
Radix_4_FFT_recursion(X, N, Batch):
    If N=4 then
        Return F4 * X
                                        (batched gemm)
    //See X as a (4 by N/4 by Batch) array
    permute(X, [2,1,3])
    //X as a (N/4 by 4 by Batch) array
    Y <- Radix_4_FFT_recursion(X, N/4, Batch*4)
    Multiply elementwise Y with W_N
    Return Y * F4
                                        (batched gemm)
End
```

Splitting is done before gemm

Combining is done after gemm

x(32) = s1(32) * x_hi(16) + s2(32) * x_lo(16),

Gemm is done to x_hi, x_lo


b=0
b=1
b=Batch-1
N
reshape
b=0
b=1
b=Batch-1
4
N/4
b=0
b=1
b=Batch-1
permute
N/4
4
FFT (X, N/4, Batch×4)
b=0
b=1
b=Batch-1
×
W_N
b=0
b=1
b=Batch-1
F_4
reshape
b=0
b=1
b=Batch-1
FFT (X, N, Batch)

CUTLASS (CUDA Templates for Linear Algebra Subroutines)

The most expensive step in the recursion: the second batched gemm

Result1 = X * F4_re; Result2 = X * F4_im
where

F4_re, F4_im: 4 by 4, fp16
X=[X_re_hi, X_re_lo, X_im_hi, X_im_lo]: m by 4 by Batch*4, fp16
Result1, Result2: m by 4 by Batch*4, fp32

For batch size = B, length = N input, will do gemms for:

m = N,	Batch = B
m = N/4,	Batch = 4B
…	
m = 4,	Batch = NB/4

cuBlas is not optimized for slender matrix multiplication (volta_sgemm_fp16_128x64_nn)

CUTLASS vs cuBlas
m by 4 * 4 by 4 matrix multiplication

m	Batch size	cuBlas(ms)	cutlass(ms)	Mean error
64	1048576	40.7779	13.4457	1.23754e-12
256	65536	5.10469	3.07621	1.27887e-12
256	262144	20.4031	12.2688	1.24481e-12
1024	16384	5.07802	3.00108	1.23879e-12
1024	65536	20.2993	11.9628	1.24625e-12
4096	4096	5.08486	3.00046	1.26754e-12
4096	16384	20.2965	11.882	1.22616e-12
16384	4096	44.524	11.8838	1.23812e-12

In the Near Future: Radix-2 vs. Radix-4

- Radix-2 algorithms: 2^v data points
  - a. *decimation-in-time* (DIT): Simplest + most common form of Cooley-Tukey alg
    - i. DFTs of even- & odd-indexed inputs, repeat recursively (O(NlogN))
  - b. *Decimation-in-frequency* (DIF): (O(NlogN)) -- divide + conquer
    - i. split DFT into 2 summations [(0 $\rightarrow$ N/2) +(N/2 $\rightarrow$ N)]
    - ii. Split those split summations into even & odd
    - iii. Repeat recursively
- Currently using radix-4 (4^v data pts)
- Why radix-2?
  - a. DFT of identity [2,2] matrix = real matrix (not complex) & exactly representable in FP16
  - b. Use tensor cores to implement it
  - c. ALTHOUGH radix-4 = more efficient when $N = 2^v$

Citations

- https://www.jics.utk.edu/files/images/recsem-reu/2018/fft/Report.pdf
- https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/
- https://arxiv.org/pdf/1803.04014.pdf
- http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html