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

Students: Yanming Kang (HKUST) and Tullia Glaeser (Tulane University)
Mentors: E. D'Azevedo (ORNL) and S. Tomov (UTK)

Abstract

We present a Fast Fourier Transform implementation utilizing the Tensor Core structure on Nvidia Volta GPUs. We base our work on an existing project, optimizing it to support inputs of larger sizes and higher dimensions.

The previous project completed the 1D and 2D FFT using radix-4 and our **objective** is to accelerate these programs, allow for larger inputs, implement the 3D algorithm, and provide a radix-2 variation.
The performance of our final implementation is similar to cuFFT, the FFT library provided by Nvidia, for small inputs.

We utilize the Tensor Cores by splitting each single precision matrix into two half precision matrices before matrix-matrix multiplications, and combining them after the multiplications. We use the parallel computing platform CUDA 10.0 and the CUTLASS template library in our implementation.

Background

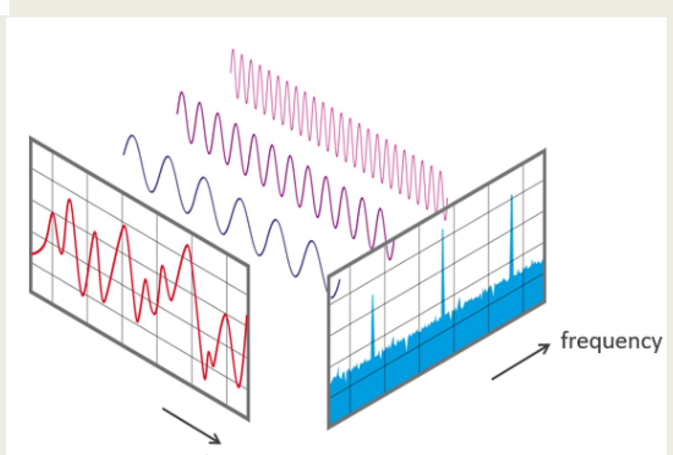
Discrete Fourier Transform (DFT)

The DFT converts time domain signals to frequency domain signals according to the equation:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-i2\pi kn/N}$$

Applications of DFT:

- Image analysis
- Speech analysis
- Data compression
- Solving PDEs
- Polynomial multiplications

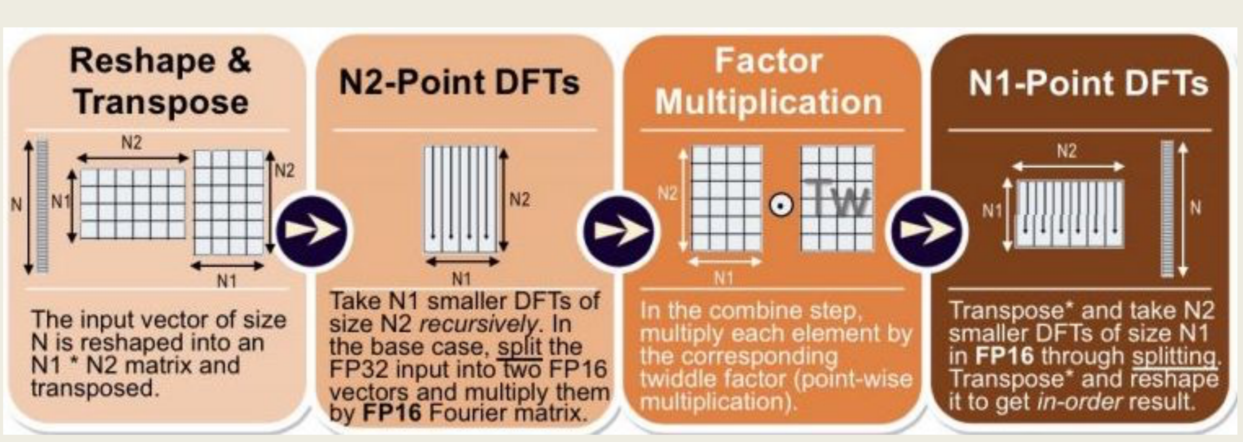


Fast Fourier Transform (FFT)

The FFT reduces the time complexity from $O(N^2)$ (DFT) to $O(N\log N)$, which is feasible for large data.

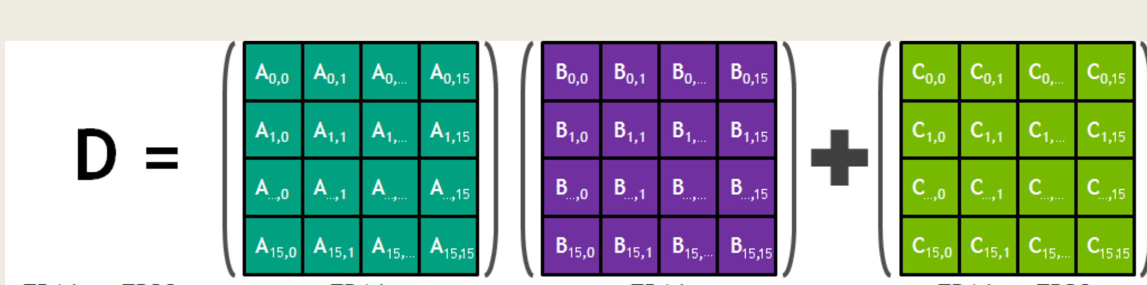
Cooley-Tukey FFT Algorithm

1. Perform N1 DFTs of size N2.
2. Multiply by complex roots of unity (often called the twiddle factors).
3. Perform N2 DFTs of size N1.

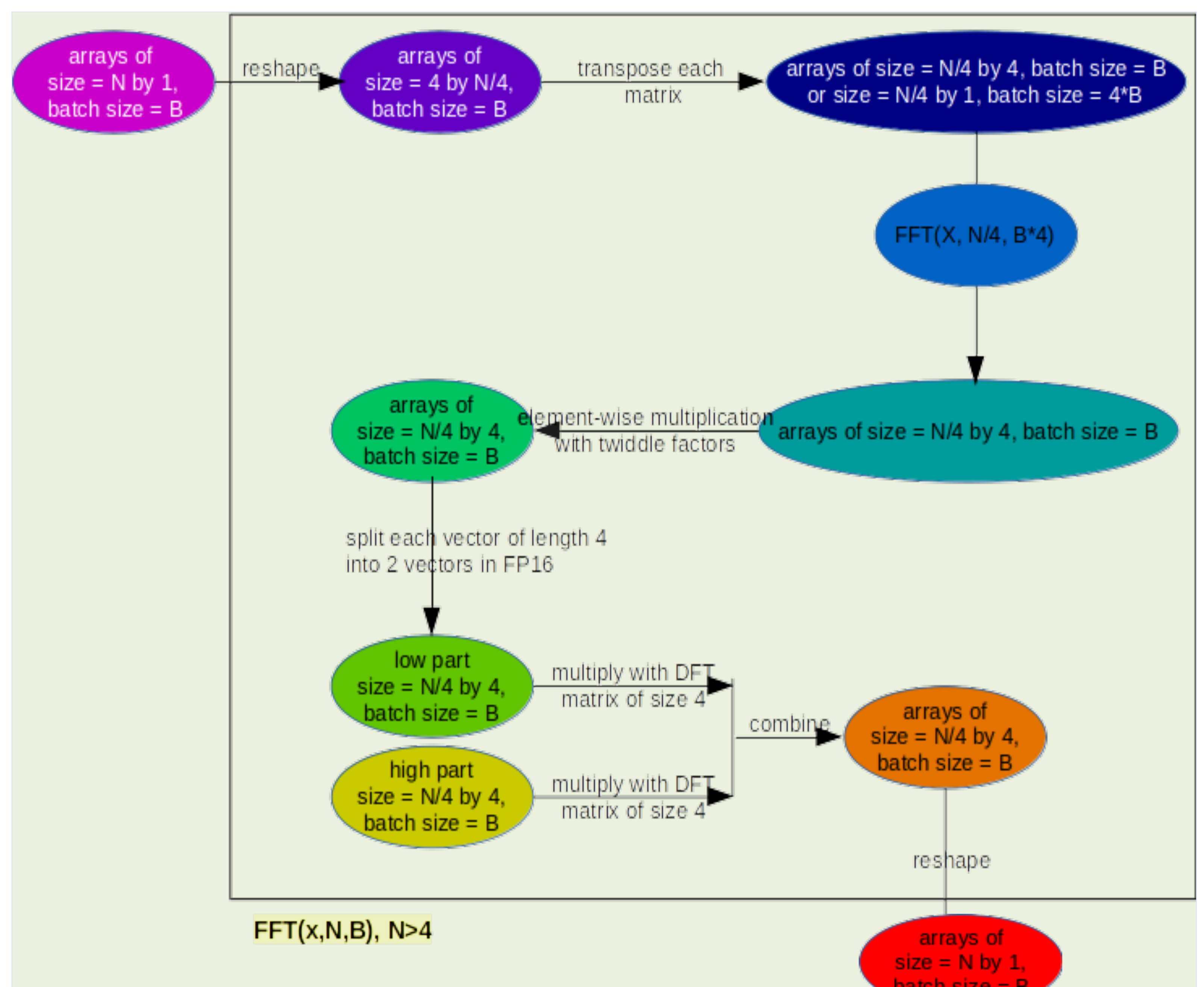


Tensor Cores on Nvidia Volta GPUs

Tensor Cores are matrix-multiply-and-accumulate units that can provide 8 times more throughput doing half precision (FP16) operations than FP32 operations.
Tensor Cores are programmable using the cuBlas library and directly using CUDA C++.



Algorithm




```
Split(X,n):
    scale1 = 0.0f
    scale2 = 0.0f
    for i = 0:n-1
        scale1 = (float)max(scale1,abs(X[i]))
    for i = 0:n-1
        X_hi[i] = (half)X[i]/scale1
    for i = 0:n-1
        tmp[i] = X[i] - scale1[i] * (float)X_hi[i]
    for i = 0:n-1
        scale2 = (float)max(scale2,abs(tmp[i]))
    for i = 0:n-1
        X_lo[i] = (half)tmp[i]/scale2
    return X_hi,X_lo,scale1,scale2
end
```

Results

1D-FFT Results

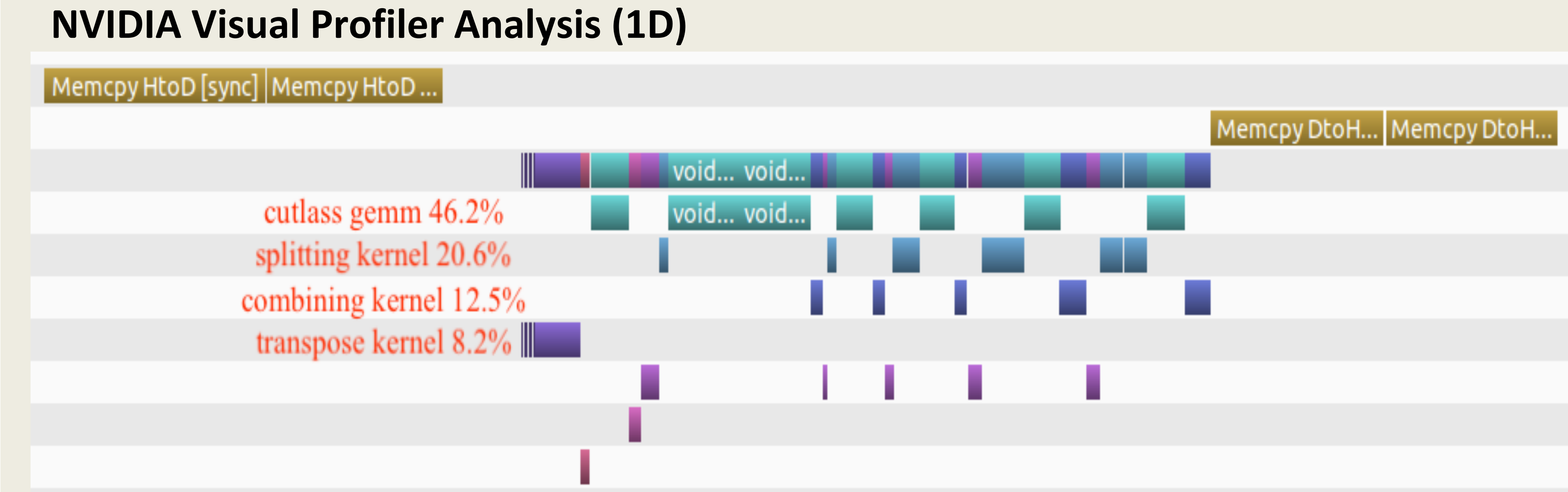
N * batch size	cuFFT 32 time (ms)	cuFFT 16 time	cuFFT 16 error[1]	accelerated FFT time	accelerated FFT error[2]	base code FFT time
1k	2.672344	3.329465	0.00191806897	2.275482	0.000000240801	4.343482
4k	2.618013	3.395736	0.0050298227	2.318682	0.000000687060322	3.688474
16k	2.721780	3.402526	0.012709721	2.414895	0.00000284706289	3.938617
64k	3.031922	3.757677	0.0295635611	3.46227	0.00000411905148	6.452106
256k	5.714272	4.05492	0.0875284001	7.000023	0.0000119805045	12.714869
1024k	9.584766	6.833481	0.192052826	13.442294	0.0000361031998	Not supported
4096k	19.234568	12.505309	0.417603344	29.521494	0.0000513304258	Not supported
16384k	65.876312	41.075443	0.588486552	98.621094	0.000133268419	Not supported
65536k	242.107086	151.377029	0.890920997	397.807739	0.000124264188	Not supported

3D-FFT Results (inputs are M by N by K)

M*N*K*batch size	cuFFT 32 time (ms)	cuFFT 16 time	cuFFT 16 error[1]	accelerated FFT time	accelerated FFT error[2]
1k	2.809283	3.367596	0.3687504530	5.071026	0.0000681395
4k	2.718959	3.594859	0.8239015937	4.131129	0.0001604883
16k	2.884303	3.796732	7.4552483559	4.910933	0.0015084511
64k	2.906378	3.492988	16.5170917511	6.833582	0.0034472588
256k	5.220939	4.153818	35.7382469177	12.123748	0.0076717613
1024k	8.608585	6.271199	354.0160522461	25.708443	0.0714902207
4096k	20.165436	13.420002	828.1459960938	64.373634	0.1603385806
16384k	64.665436	40.914425	1595.7766113281	225.811356	0.3202181458
65536k	243.238541	157.071777	35710.8671875000	903.765625	0.4209230542

1,2: Errors are calculated when batch size = 64 and inputs are random complexes in [-1,1], cuFFT 32 results considered exact.

NVIDIA Visual Profiler Analysis (1D)



Conclusion and Future Work

We successfully completed the 3D Fourier Transform of a N-length input sequence using radix-4.
We are also working on completing the FFT using radix-2, which divides the algorithm into 2 even and odd parts instead of the 4 of radix-4; this seems to be a faster algorithm and if we don't complete it on time it will be good future work.
Future:

- Allow for larger inputs -- possibly using an updated version of CUDA
- Speed up our FFT algorithm to match cuFFT
- Efficient memory allocation to minimize data transmission between host (CPU) and device (GPU)

Acknowledgements

This project was sponsored by the National Science Foundation through Research Experience for Undergraduates (REU) award, with additional support from the Joint Institute of Computational Sciences at University of Tennessee Knoxville. This project used allocations from the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by the National Science Foundation. In addition, the computing work was also performed on technical workstations donated by the BP High Performance Computing Team.
This material is based upon work supported by the U.S. DOE, Office of Science, BES, ASCR, SciDAC program.