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[5], 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 radix 2 and radix 8 variations.
The performance of our final implementation is similar to cuFFT, the FFT library provided by Nvidia, for small inputs.

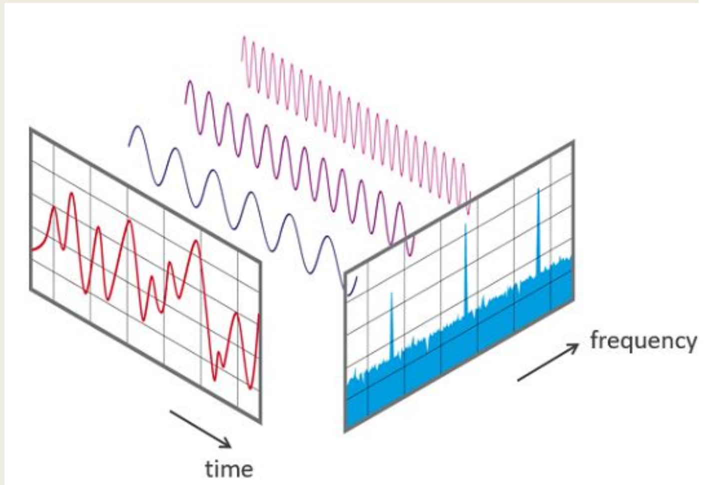
We utilize the Tensor Cores by splitting each single precision matrix into two half precision matrices before matrix-matrix multiplications, and combining them after the multiplications. We use the parallel computing platform CUDA 10.0 and the CUTLASS template library in our implementation.

Background

Discrete Fourier Transform (DFT)
The DFT converts time domain signals to frequency domain signals according to the equation:

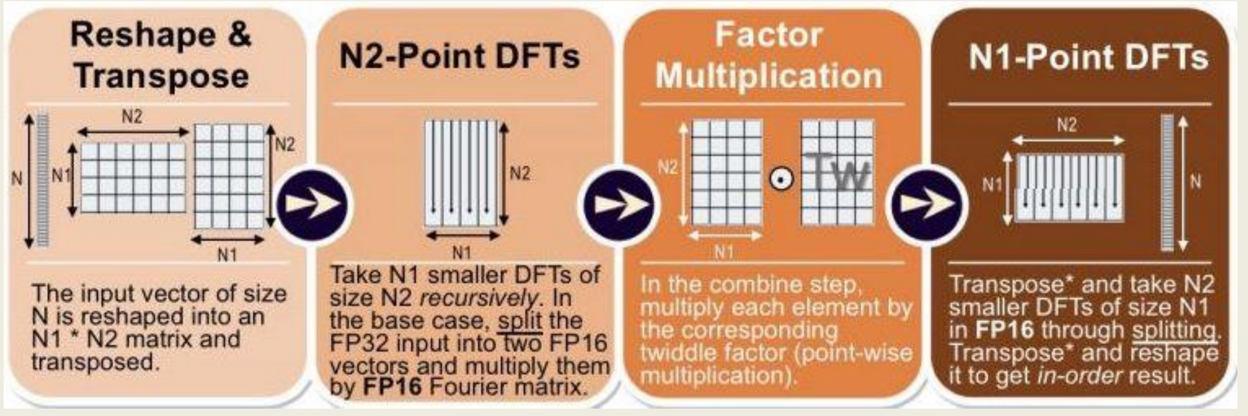
$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-i2\pi kn/N}$$

Applications of DFT:
- Image analysis
- Speech analysis
- Data compression
- Solving PDEs
- Polynomial multiplications

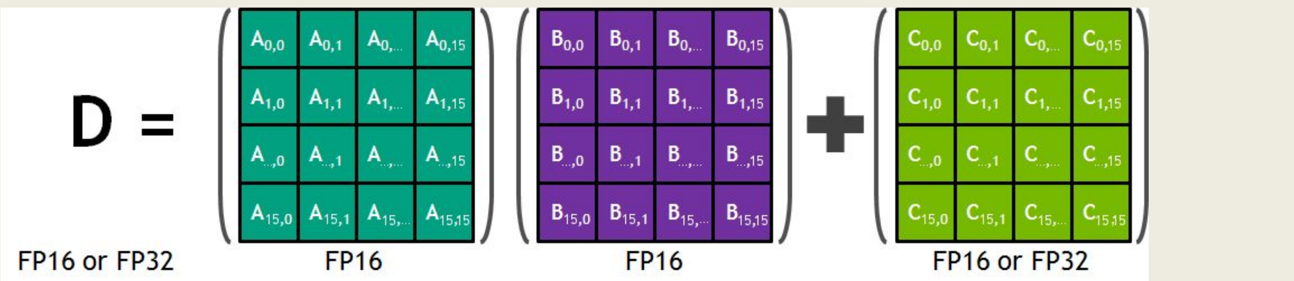


Fast Fourier Transform (FFT)
The FFT reduces the time complexity from O(N²) (DFT) to O(NlogN), which is feasible for large data.

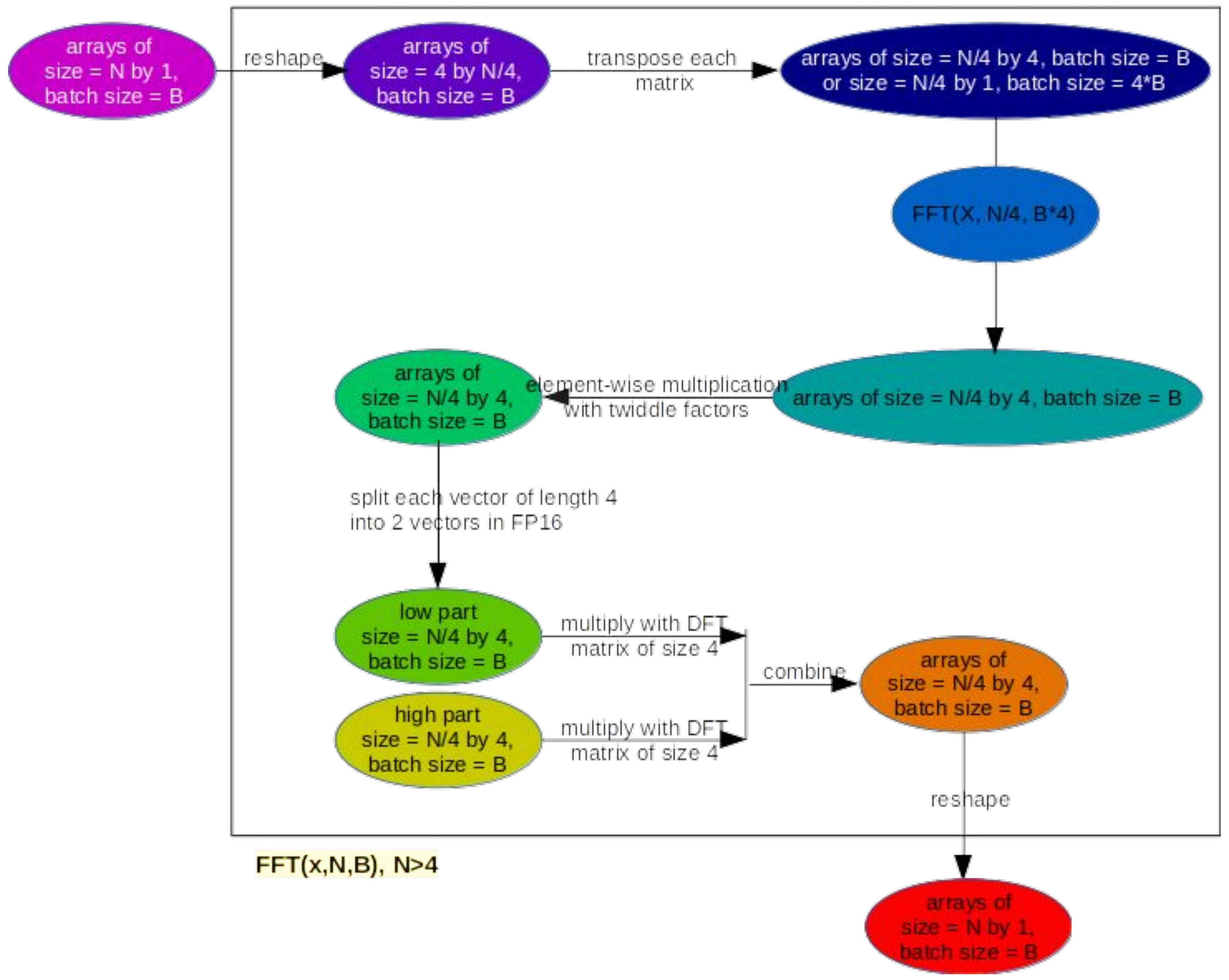
Cooley-Tukey FFT Algorithm
1. Perform N1 DFTs of size N2.
2. Multiply by complex roots of unity (often called the twiddle factors).
3. Perform N2 DFTs of size N1.

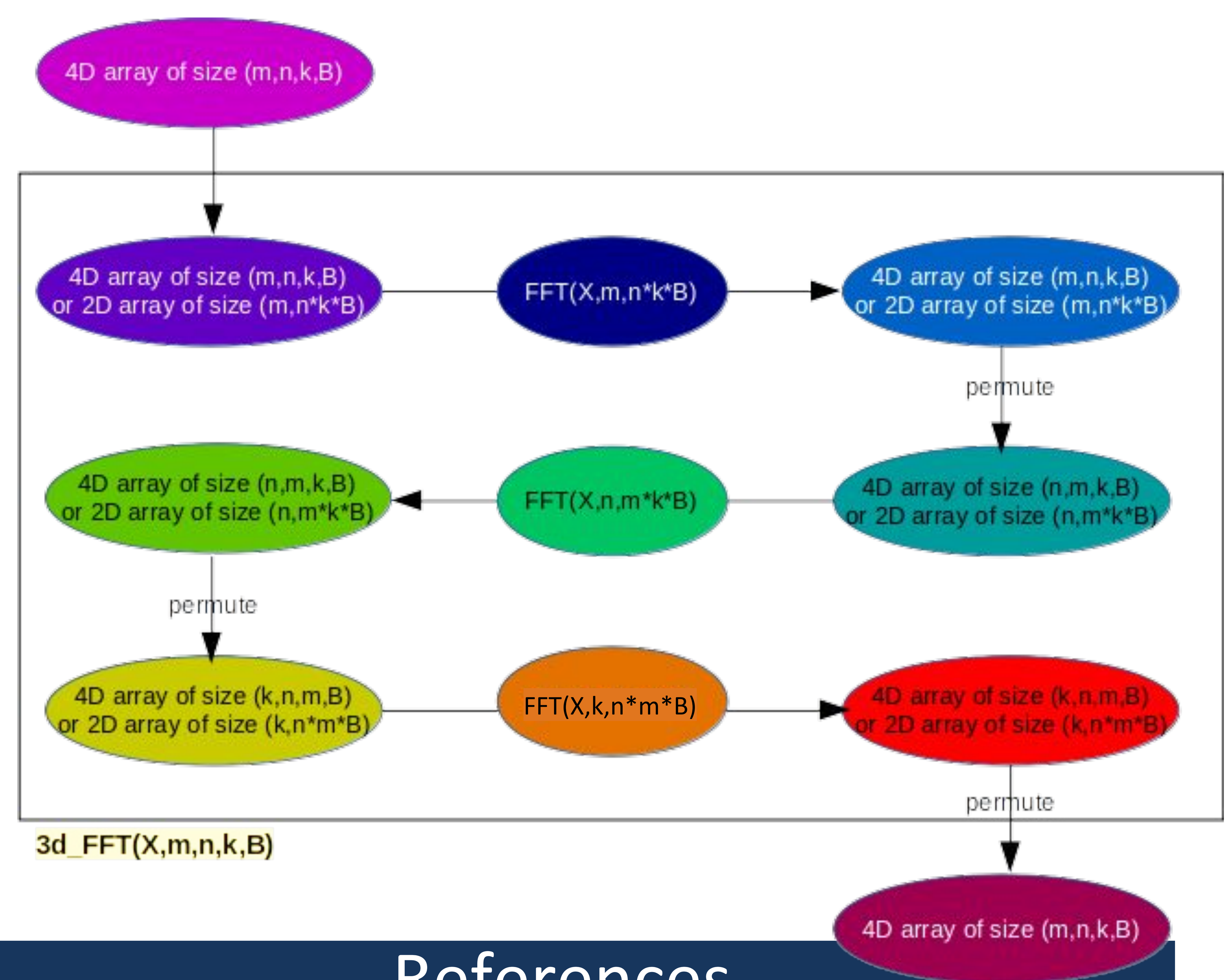


Tensor Cores on Nvidia Volta GPUs
Tensor Cores are matrix-multiply-and-accumulate units that can provide 8 times more throughput doing half precision (FP16) operations than FP32 operations. Tensor Cores are programmable using the cuBlas library and directly using CUDA C++.



Algorithm





References

[1] CMLaboratory CMLAB. "Fast Fourier Transform (FFT).". www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html/. [Online; accessed July-2019].
[2] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation, 1965.
[3] NVIDIA Corporation. CUDA TEMPLATE LIBRARY FOR DENSE LINEAR ALGEBRA AT ALL LEVELS AND SCALES.https://github.com/NVIDIA/cutlass, 2019. [Online; accessed July-2019].
[4] NVIDIA Corporation. CUFFT Library Programming Guide. https://docs.nvidia.com/cuda/cufft/index.html, 2019. [Online; accessed July-2019].
[5] Sorna Anumeena et al. Accelerating the fast fourier transform usingmixed precision on tensor core hardware. www.jics.utk.edu/files/images/recsem-reu/2018/fft/Report.pdf, 2018. [accessed July-2019].
[6] Stefano Markidis et al. NVIDIA Tensor Core Programmability, Performance and Precision. NVIDIA Tensor Core Programmability, Performance & Precision, pages 1–12.
[7] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Computing, 36(5-6):232–240, June 2010.
[8] Jake VanderPlas. "Understanding the FFT Algorithm.". jakevdp.github.io/blog/2013/08/28/understanding-the-fft/, 2013

Results

1D radix 8 FFT Results

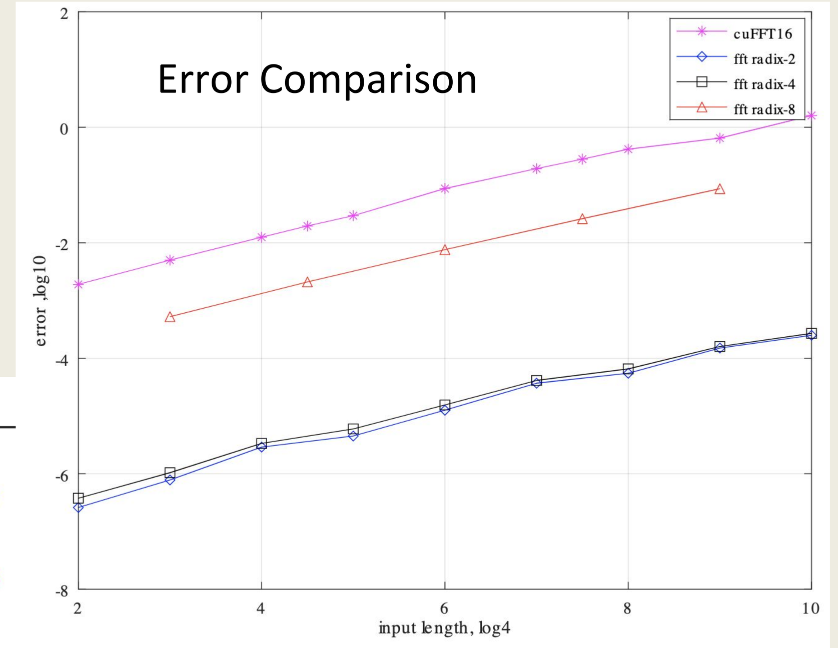
N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
4k	2.43	3.18	$5.02 * 10^{-3}$	2.28	$5.34 * 10^{-4}$
32k	2.53	2.44	$1.93 * 10^{-2}$	3.23	$2.10 * 10^{-3}$
256k	4.90	3.65	$1.94 * 10^{-2}$	6.59	$2.10 * 10^{-3}$
2048k	11.69	7.88	$8.73 * 10^{-2}$	16.72	$7.60 *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

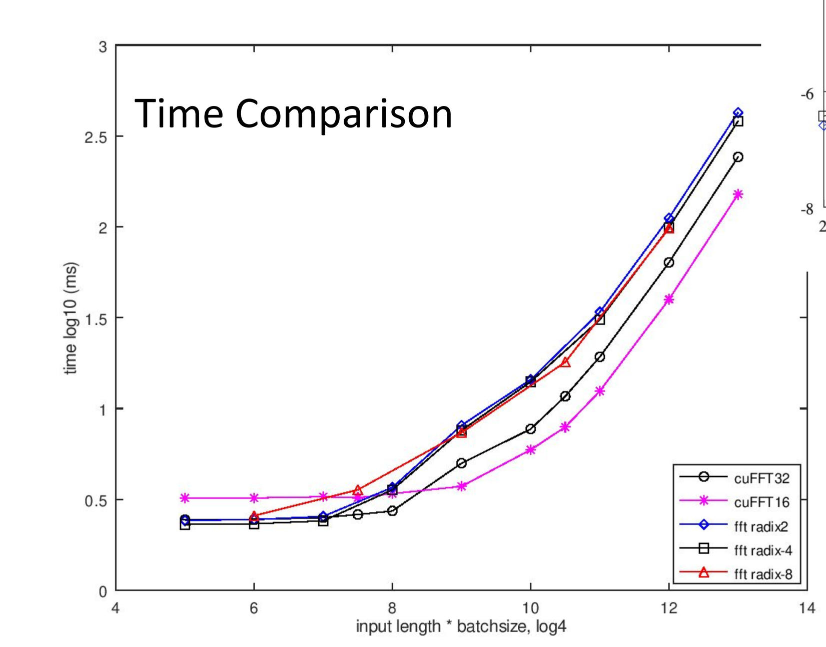
Table 3: 1-D radix-8 FFT results

3D radix 8 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





As shown in these 1D line graphs, the radix 8 algorithm is generally the fastest of our implementations but also has the largest error; even so, the error up to input size 16384K does not surpass 7.06 * 10^{-3}. This is also reflected in the 2D and 3D implementations of radix 8.

Conclusion and Future Work

We successfully completed the 1D, 2D, & 3D Fourier Transforms of an input sequence using radixes 2, 4, & 8; the different radixes can take different input sizes and reshape the input in different manners. While the speed of our implementation is still inferior to CUDA's built-in cuFFT library, we are quickly approaching its efficiency through the radix 8 algorithm and with the help of CUTLASS.

Future:
- Split-radix algorithm, combining 2+ different radices. eg. combine radix 4 and radix 8
- Manipulate the code / use different memory allocation tricks →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).

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.