Optimizing the Fast Fourier Transform using Mixed Precision on Tensor Core Hardware

Anumeena Sorna*, Xiaohe Cheng†, Eduardo D'Azevedo‡, Kwai Wong§, Stanimire Tomov§
*National Institute of Technology, Tiruchirappalli
108115011@nitt.edu
†Hong Kong University of Science and Technology
xchengaj@connect.ust.hk
‡Oak Ridge National Laboratory
dazevedoef@ornl.gov
§University of Tennessee, Knoxville
kwong@utk.edu
tomov@icl.utk.edu

***Abstract*—The Fast Fourier Transform is a fundamental tool in scientific and technical computation. The highly parallelizable nature of the algorithm makes it a suitable candidate for GPU acceleration. This paper focuses on exploiting the speedup due to using the half precision multiplication capability of the latest GPUs' tensor core hardware without significantly degrading the precision of the Fourier Transform result. We develop an algorithm that dynamically splits the input single precision dataset into two half precision sets at the lowest level, uses half precision multiplication, and recombines the result at a later step. This work paves the way for using tensor cores for high precision inputs.**

***Index Terms*—Fast Fourier Transform; GPU Tensorcores; CUDA; Mixed Precision**

I. Introduction

The Fast Fourier Transform (FFT) is a widely used numerical algorithm that plays a vital role in many scientific and engineering applications. In large computational applications, including image processing, speech recognition, and large scale simulations, a majority of execution time is allotted to computing the FFT [1] [2] [3]. In order to improve performance of the FFT, many investigations have been made on implementing the FFT on the computationally superior Graphical Processing Unit (GPU) platform [4].

Recently, half precision floating point arithmetic (FP16) is gaining popularity with its faster speed and energy saving ability. With the introduction of the tensor cores on the NVIDIA Volta GPU Hardware, a large speed up, up to 12x, in half precision matrix multiplications, has been introduced [5]. The FFT can benefit greatly from the advantages offered by tensor cores, as it is a matrix multiplication intensive algorithm.

Unfortunately, this half precision hardware cannot be exploited in scientific FFT applications where single precision (FP32) is required. In order to satisfy the accuracy requirement while utilizing the advanced half precision hardware, a mixed precision method utilizing dynamic splitting is developed. This method efficiently uses the computational capability of tensor cores without a significant drop in precision.

II. Background and Related Work

In numerical computing, there have been many attempts to utilize the fast operations on low precision numbers to emulate high precision computation [6] [7] [8]. The more compact numerical representation better utilizes the memory space and eliminates memory communication costs between memory hierarchy. Besides memory advantage, low precision calculation tends to outperform high precision calculation for more than two times in terms of GFLOPS. Therefore, it is desirable to emulate the accuracy of a high precision number with two low-precision number. Previous works have presented various methods to mix double precision and single precision computation to optimize the performance, but there has been little discussion on how to exploit the high-speed half precision hardware.

III. FFT Algorithm

The Discrete Fourier Transform (DFT) converts a finite discrete signal in the time domain to a one in the frequency domain according to the following equation:

$$X[k] = \sum_{n=0}^{N-1} x(n) * e^{-2j\pi nk/N} \tag{1}$$

The inverse is given by:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] * e^{2j\pi nk/N} \tag{2}$$

Where $x(n)$ is the discrete signal in the time domain, $X[k]$ is the discrete signal in the frequency domain and N is the entire length of the sequence. The DFT can clearly be rewritten as a matrix multiplication with the number of computations required of the order $O(N^2)$.

The class of algorithms that efficiently calculate the DFT with a lower number of computations is known as FFT. Gentleman and Sande developed the first FFT algorithm that rewrote the length N sequence as $N = n1 \times n2$ in order compute of the DFT with a lower number of computations

[9]. By dividing a problem of size N into two (or x) problems of size $N/2$ (or N/x), it attains time complexity $O(NlogN)$ [10].

The algorithm can be succinctly stated as follows:

1) Represent the length N sequence as an $n1 \times n2$ matrix.
2) Transpose the matrix, resulting with an $n2 \times n1$ matrix.
3) Take $n1$ individual $n2$-point-FFTs down the columns of the matrix.
4) Perform element wise multiplication with the resultant matrix and the twiddle factor matrix.
5) Transpose the matrix, resulting with an $n1 \times n2$ matrix.
6) Take $n2$ individual $n1$-point-FFTs down the columns of the matrix.
7) Transpose the resultant matrix

This method requires two multicolumn FFTs as well as three matrix transposition operations.

This can be represented as:

$$\text{FFT}(x) = F_N \cdot x = (F_{n1} \cdot (W_N \times [F_{n2} \cdot x^T_{n1 \times n2}])^T)^T \quad (3)$$

Where $x_{n1 \times n2}$ is the vector x reshaped as a matrix of $n1 \times n2$ and F_N is the Fourier matrix defined by $F_N[k,l] = \exp(-2j\pi kl/N)$ and W is twiddle matrix given by $W_N[k,l] = \exp(-2j\pi kl/N)$.

For convenience, the $\times$ operation is used to denote scalar multiplication and the $\cdot$ operation is used to denote matrix multiplication.

A. Adapting the Algorithm

1) Choosing the radix: The real and imaginary Fourier matrices are defined as:

$$F_{Nreal}[k,l] = \cos(2\pi kl/N) \quad (4)$$
$$F_{Nimag}[k,l] = -\sin(2\pi kl/N) \quad (5)$$

By choosing a radix of 4, or only allowing $N = 4$, we can observe that the elements of the real and imaginary Fourier matrix is either 1, 0, or -1. This is exactly representable in FP16 without loss of precision.

$$F_{4real} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & -1 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix}$$

$$F_{4imag} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

2) Elimination of a Few Transposes: Large matrix transpositions are bulky operations limited by communication and memory bandwidth. To reduce the number of transpositions required, we may employ common matrix properties to simplify the FFT equation.

$$\text{FFT}(x) = (F_4 \cdot (W_N \times [F_{N/4} \cdot x^T_{4 \times N/4}])^T)^T \quad (6)$$
$$= (W_N \times [F_{N/4} \cdot x^T_{4 \times N/4}])^{T^T} \cdot F_4^T \quad (7)$$
$$= (W_N \times [F_{N/4} \cdot X^T_{4 \times N/4}]) \cdot F_4^T \quad (8)$$

This can be further simplified by observing that the real and imaginary Fourier matrices of a length 4 sequence are symmetrical. Therefore, $F_4^T = F_4$.

$$\text{FFT}(x) = (W_N \times [F_{N/4} \cdot x^T_{4 \times N/4}]) \cdot F_4 \quad (9)$$

This simplification reduces the number of transpositions required from 3 to 1. But, this introduces a complexity in the code; the FFTs computed in step 3 and 6 cannot be calculated in an identical fashion. The order of matrix multiplication is interchanged in the two steps. However, this complication is preferable to computing extra transpositions.

B. Adapted FFT Algorithm

Keeping the previous adaptations in mind, the implemented FFT algorithm is as follows:

1) Represent the length N sequence as a $4 \times N/4$ matrix.
2) Transpose the matrix, resulting with a $N/4 \times 4$ matrix.
3) Take FFTs down the columns of the matrix recursively until the size of the FFT transform does not exceed 4.
4) Perform element wise multiplication with the resultant matrix and the twiddle factor matrix.
5) Take length-4 FFTs down the columns of the matrix.

IV. Dynamic Splitting

In order to exploit the throughput of the tensor cores, a mixed precision approach is developed. This method ensures that only the matrix multiplication operations are done on the half precision input dataset but the rest of the FFT algorithm operates on the single precision dataset.

At the lowest level of the FFT algorithm, the single precision data sequence is converted into two half precision datasets. Every FP32 number is expressed as a scaled sum of two FP16 numbers. As the FFT is a linear algorithm, Length-4 FFTs are applied separately to the half precision datasets and recombined. This splitting operation is called twice, right before the FFT matrix multiplication in step 3 of the adapted FFT algorithm and before the FFT matrix multiplication in step 5 of this algorithm.

$$x_{fp32}[:] = s1_{fp32} \times x1_{fp16}[:] + s2_{fp32} \times x2_{fp16}[:] \quad (10)$$

$$F_N \cdot x_{fp32}[:] = s1_{fp32} \times (F_N \cdot x1_{fp16}[:]) + s2_{fp32} \times (F_N \cdot x2_{fp16}[:]) \quad (11)$$

In order to retain as much accuracy as possible, a dynamic splitting algorithm is employed. Scaling vectors, s_1 and s_2

are utilized to minimize the error caused by the FP32 to FP16 conversion. These scaling factors are determined for each column of the input matrix and are single precision numbers.

A. Dynamic Splitting Algorithm:

Step 1: Find the absolute norm of each column of the input matrix to decide s_1 and divide the respective column by the scaling factor

$$s_1(j) = \max_{i=1}^{m} \|x_{fp32}(i,j)\| \quad : i = 1, ..., n1 \\ : j = 1, ..., n2$$

$$x_{fp32}(i,j) = \frac{x_{fp32}(i,j)}{s_1(j)}$$

Step 2: Convert the input FP32 matrix x_fp32 into FP16 matrix $x1_fp16$

$$x1_{fp16}(i,j) \triangleright x_{fp32}(i,j) \quad : i = 1, ..., n1 \\ : j = 1, ..., n2$$

Step 3: Calculate the residual error caused by conversion and store as $x2_{fp32}$

$$x2_{fp32}(i,j) = x_{fp32}(i,j) - \vec{x1_{fp16}}(i,j) \times s_1(j) \quad : i = 1..n1 \\ : j = 1..n2$$

$$x_{fp32}(j) = x_{fp32}(i,j) \times s_1(j)$$

Step 4: Find the absolute norm of each column of the residual matrix to decide s_2 and divide the respective column by the scaling factor

$$s_2(j) = \max_{i=1}^{m} \|x2_{fp32}(i,j)\| \quad : i = 1, ..., n1 \\ : j = 1, ..., n2$$

$$\vec{x2_{fp32}}(i,j) = \frac{\vec{x2_{fp32}}(i,j)}{s_2(j)}$$

Step 5: Convert the residual FP32 matrix $x2_fp32$ into FP16 matrix $x2_fp16$

$$x2_{fp16}(i,j) \triangleright x2_{fp32}(i,j) \quad : i = 1, ..., n1 \\ : j = 1, ..., n2$$

In order to avoid mathematical round off error, the scaling factors, s_1 and s_2, are chosen to be powers of 2. In this case then multiplication or division may be done by simply shifting the exponent bit pattern, leaving the mantissa unmodified.

V. Implementation

Our experimental platform is a heterogeneous processor consisting of a CPU and a GPU. We implemented the FFT algorithm as described in sections 3 and 4 on an NVIDIA Volta V100 graphics card.

Specification	V100
Peak FP32 TFlops	15.7
Peak FP16 TFlops	125
Tensor Cores	640
CUDA Cores	51200
GPU Memory	16 GB
Memory bandwidth	900GB/sec

Implementing the FFT on the graphics card is a relatively straightforward process simplified by utilizing the commonly used cuBLAS library API.

The algorithm consists of 3 major arithmetic operations: splitting FP32 numbers into two FP16 numbers, transposing matrices, and multiplying matrices. Customized kernels are written for the splitting operation and the transpose operation. The matrix multiplication is computed using the CublasGemmEx and CublasGemmStridedBatch functions.

Of the three operations, only the matrix multiplication operation utilizes the tensor core hardware. We expect the overhead due to splitting the input is not greater than the speedup offered by computing the matrix multiplication on the computationally superior tensor cores.

VI. Experimental Results

In this section, we evaluate our FFT implementation by comparing it with cuFFT library APIs. We run experiments to test their performance under various configurations. The problem size, i.e. the number of elements in a single vector, increases from 4 to 65536[1]. We also test the multi-vector (batched) cases and the batch size is from 1 to 1024. The input data are randomly generated and are restricted to a given range.

The major performance metrics in the evaluation are speed and accuracy. We choose not to compare the FLOPS as the algorithm used in cuFFT APIs are automatically determined, and the number of operations varies among different algorithms. We make use of CUDA Event to mark the beginning and completion of computation to measure the execution time. The error of our implementation and the half-precision cuFFT is assessed based on the results of single-precision cuFFT. We normalize the error by the input range:

$$Error = \frac{\sum |Result - BaselineResult|}{InputRange}$$

Figure 1 compares the accuracy of two half-precision FFT: our implementation and cuFFT that takes 16-bit inputs. The error statistics indicate that our implementation reduces the computation error significantly. It preserves high accuracy even with growing input sizes, which may qualify it for many scientific applications.

An important aspect to consider is the execution time of the matrix multiplication as compared to the splitting and subsequent recombining of the FP32 numbers. If the overhead due to splitting is larger than the speed up due to the tensor core multiplication, then the mixed precision method would not achieve and benefit over the normal method. Figure 2 shows that this is not the case. As the input size increases, the execution time of the splitting and recombining operations stays relatively low as compared to the GEMM matrix multiplication. This implies utilizing the tensor cores would provide

[1]We intended to test the performance under larger problem sizes, but a limit is imposed by the cuBLAS library. The batched matrix multiplication API is called during our implementation, and the total size cannot exceed 65536 [11].

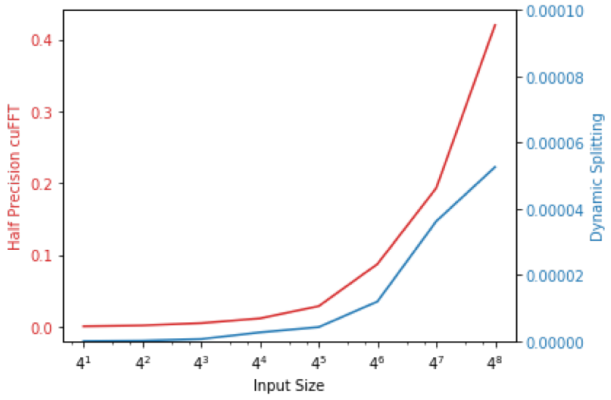

Fig. 1. Accuracy of half-precision cuFFT and our implementation.

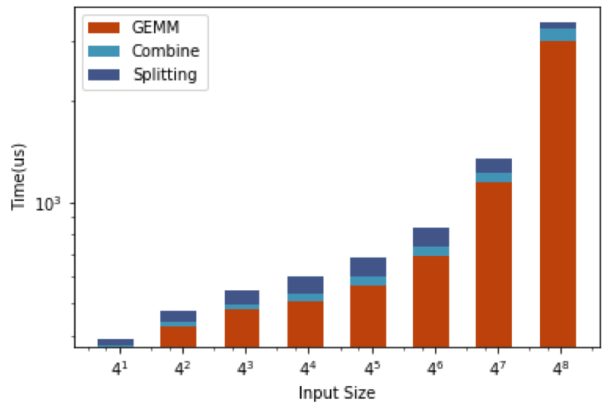

Fig. 2. Execution time breakdown at different input sizes.

speed up to the FFT algorithm despite the overhead incurred due to mixed precision.

Lastly, the execution time of the entire FFT program is compared to that of the cuFFT of FP32 and FP16. It is seen that our implementation is currently inferior to the highly optimized cuFFT library. However, from the previous results, we infer using this mixed precision method with optimized kernels would achieve lower execution time.

VII. Conclusion and Future Work

We have designed and implemented a FP32-FP16 mixed precision FFT that takes advantage of the recent tensor core hardware. The dynamic splitting method effectively emulates single-precision calculation and produces highly accurate results from a variety of inputs. The speed of current cuBLAS-based implementation is inferior to cuFFT APIs, but we expect it to gain an advantage with larger input size as the tensor core can be fully utilized and the setup cost can be amortized.

The current implementation can handle a large variety of inputs. The relative error exceeds 0.1 or the program throws an error when: input data range greater than `3 * 1E10`; or input data range less than `5 * 1E-11` (close to the dynamic range of single precision number). The range may be further enlarged by pre-scaling all input numbers.

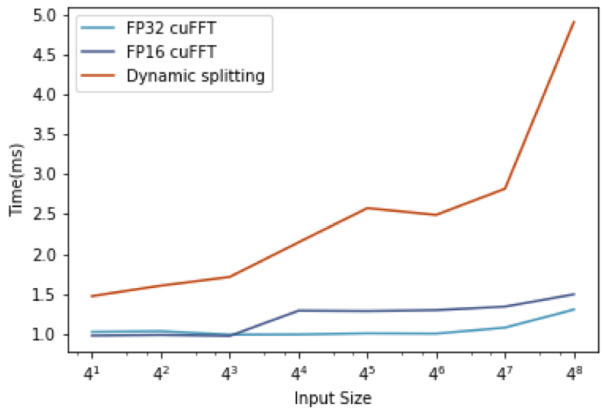

Fig. 3. Average execution time of three different FFTs with growing input sizes.

Our GPU-based mixed precision FFT implementation can easily be applied to many areas of scientific computing where increased accuracy is desired.

There are several interesting directions for further optimizations. Many operations can be tuned specifically for the problem size involved in the FFT calculation. The time spent on 16-bit GEMM grows quickly when input size exceeds 16384. This may due to the inefficient cuBLAS implementation for the problem set. This may be improved by implementing transpose and GEMM kernels. Another direction is to design an auto-tuning splitting algorithm that supports ill-conditioned inputs, and further optimizes the splitting overhead. A more sophisticated splitting algorithm may be designed. This could be done using bit manipulations. Lastly, our implementation of matrix transpose kernel has yet to take advantage of the shared memory. It can be accelerated by applying the "tiled" design.

Acknowledgment

This research 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. This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725.

REFERENCES

[1] B.-Y. T. Shing-Tai Pan, Chih-Chin Lai, "The implementation of speech recognition systems on fpga-based embedded systems with soc architecture," *IJICIC*, vol. 7, no. 10, 2011.

[2] M. F. H. Buijs, A. Pomerleau, "Implementation of a fast fourier transform (fft) for image processing applications," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 22, pp. 420–424, 1974.

[3] J. Kong and S. Yu, "Fourier transform infrared spectroscopic analysis of protein secondary structures," *Acta biochimica et biophysica Sinica*, vol. 39, no. 8, pp. 549–559, 2007.

[4] X. L. Shuo Chen, "A hybrid gpu/cpu fft library for large fft problems," *IEEE 32nd International Performance Computing and Communications Conference*, 2014.

[5] S. C. Stefano Markidis, I. P. Erwin Laure, and J. Vetter, "Nvidia tensor core programmability, performance and precision," *Eighth International Workshop on Accelerators and Hybrid Exascale Systems*, 2018.

[6] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak, "Mixed precision iterative refinement techniques for the solution of dense linear systems," *The International Journal of High Performance Computing Applications*, vol. 21, no. 4, pp. 457–466, 2007.

[7] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, no. 12, pp. 2526–2533, 2009.

[8] S. Le Grand, A. W. Götz, and R. C. Walker, "Spfp: Speed without compromise—a mixed precision model for gpu accelerated molecular dynamics simulations," *Computer Physics Communications*, vol. 184, no. 2, pp. 374–380, 2013.

[9] W. M. Gentleman and G. Sande, "Fast fourier transforms: for fun and profit," in *Proceedings of the November 7-10, 1966, fall joint computer conference*. ACM, 1966, pp. 563–578.

[10] D. H. Bailey, "Ffts in external or hierarchical memory," *The Journal Of Supercomputing*, vol. 4, p. 23–35, 1989.

[11] (2018, Jul.) cublas batched gemm throw not supported error with large batch size. [Online]. Available: https://stackoverflow.com/questions/51500189/cublas-batched-gemm-throw-not-supported-error-with-large-batch-size