NSF REU - 2018: Project Report

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

Anumeena Sorna
Electronics and Communciation Engineering
National Institute of Technology, Tiruchirappalli
108115011@nitt.edu

Xiaohe Cheng
Hong Kong University of Science and Technology
xchengaj@connect.ust.hk

Mentor: Eduardo DAzevedo
Oak Ridge National Laboratory
dazevedoef@ornl.gov

Mentor: Kwai Wong
University of Tennessee, Knoxville
kwong@utk.edu

Mentor: Stanimire Tomov
University of Tennessee, Knoxville
tomov@icl.utk.edu

ABSTRACT

We present a fast and accurate parallel algorithm for computing the Fast Fourier Transform on the Volta Graphical Processing Unit. This paper focuses on utilizing the speedup due to using the half precision multiplication capability of latest graphical processing units' tensor core hardware without significantly degrading on the precision of the Fourier Transform result. In this paper, an algorithm is developed that dynamically splits the input single precision data set into two half precision sets at the lowest level for half precision multiplication and recombines the result at a later step. The Fast Fourier Transform algorithm is widely used in many applications and we hope to further optimize the algorithm for the domain specific computational needs.

Keywords: Fast Fourier Transform; GPU Tensorcores; CUDA; Mixed Precision

RESEARCH GOAL

In this research experience, we developed and implemented a method that allows faster computation of the FFT by exploiting tensor core hardware on V100 GPUs but, at the same, time preserve as much accuracy as possible using a mixed precision method. We tested this algorithm using MATLAB for verification of results and proceeded to implementation on the GPUs using CUDA and cuBLAS API.

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 alloted to computing the FFT. 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.

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 Nvdia Volta GPU Hardware, a large speed up, up to 12x, in half precision matrix multiplications has been introduced.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 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. DFT and FFT ALGORITHM

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

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

The inverse is given by:

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

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

The DFT can also clearly be represented in the matrix form:

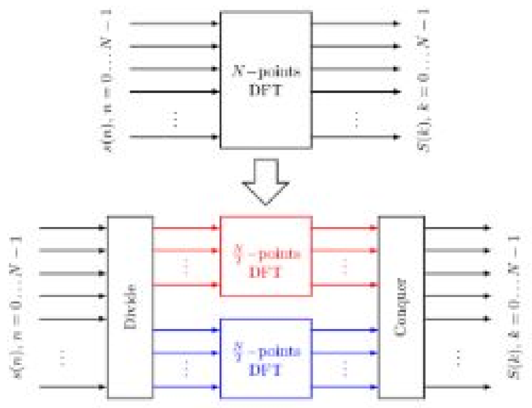
$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{kn}$$

$$\begin{bmatrix} X[0] \\ X[1] \\ X[2] \\ \vdots \\ X[N-1] \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N^1 & W_N^2 & \cdots & W_N^{(N-1)} \\ 1 & W_N^2 & W_N^4 & \cdots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{(N-1)} & W_N^{2(N-1)} & \cdots & W_N^{(N-1)^2} \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ \vdots \\ x[N-1] \end{bmatrix}$$

The high number of matrix multiplications of the DFT enable it to be a highly parallel algorithm that can benefit from the advanced matrix multiplication capability of GPUs. It is also important to note that the DFT is clearly a linear operation.

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 × n2 in order compute of the DFT with a lower number of computations. By dividing a problem of size N to two (or x) problems of size N/2 (or N/x), it attains time complexity O(NlogN).

The FFT is a divide and conquer algorithm that uses the symmetry of the DFT equation to reduce the number of computations

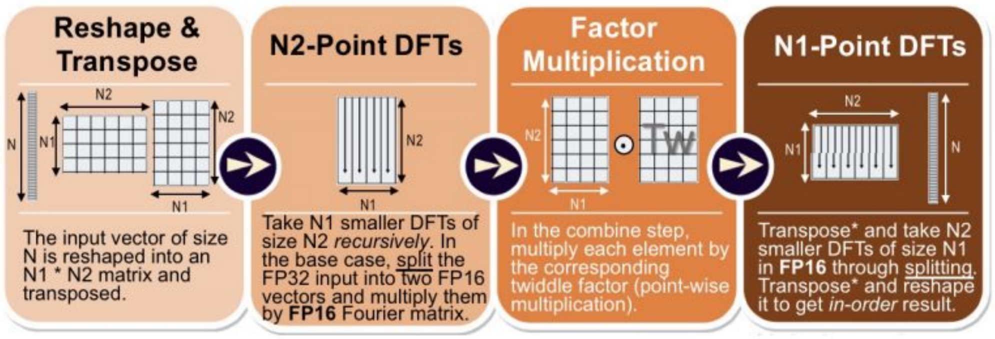


Source: DSPlib http://en.dsplib.org/content/fft_introduction/fft_introduction.html

There have been many FFT algorithm proposed. We follow the simple and elegant orginal FFT algorithm proposed by Gentlemen and Sande in their paper "Fun for Profit." The algorithm can be succinctly stated as follows:

1) Represent the length N sequence as an n1 × n2 matrix.
2) Transpose the matrix, resulting with an n2 × 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 × 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 3 matrix transposition operations.


Reshape & Transpose
N2
N
N1
N2
N1
The input vector of size N is reshaped into an N1 * N2 matrix and transposed.
N2-Point DFTs
N2
N1
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
N2
N1
Tw
In the combine step, multiply each element by the corresponding twiddle factor (point-wise multiplication).
N1-Point DFTs
N2
N1
N
Transpose* and take N2 smaller DFTs of size N1 in **FP16** through splitting. Transpose* and reshape it to get *in-order* result.

This can be represented as:

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

Where xn1×n2 is the vector x reshaped as a matrix of n1×n2 and FN is the Fourier matrix defined by FN [k, l] = e −2j π kl/N and W is twiddle matrix given by WN [k, l] = e −2j π kl/N.

A. Adapting the algorithm

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

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

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.

$$\begin{aligned} FFT(x) &= (F_4 \cdot (W_N \times [F_{N/4} \cdot x^T_{4\times N/4}])^T)^T \\ &= (W_N \times [F_{N/4} \cdot x^T_{4\times N/4}])^{T^T} \cdot F_4^T \\ &= (W_N \times [F_{N/4} \cdot X^T_{4\times N/4}]) \cdot F_4^T \end{aligned}$$

This can be further simplified by observing that the real and imaginary Fourier matrices of a length 4 sequence are symmetrical. Therefore, FT4 = F4.

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

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.

B. Adapted FFT Algorithm

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

1) Represent the length N sequence as an 4 × N/4 matrix.
2) Transpose the matrix, resulting with an N/4 × 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.

III. 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 data set but the rest of the FFT algorithm operates on the single precision data set.

At the lowest level of the FFT algorithm, the single precision data sequence is converted into two half precision data sets. 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 data sets and recombined.This splitting operation will be 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.

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

In order to retain as much accuracy as possible, a dynamic splitting algorithm is employed. Scaling vectors, s1 and s2 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 s1 and divide the respective column by the scaling factor

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

Step 2: Convert the input FP32 matrix xf p32 into FP16 matrix x1f p16

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

Step 3: Calculate the residual error caused by conversion and store as x2fp32

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

Step 4: Find the absolute norm of each column of the residual matrix to decide s2 and divide the respective column by the scaling factor.

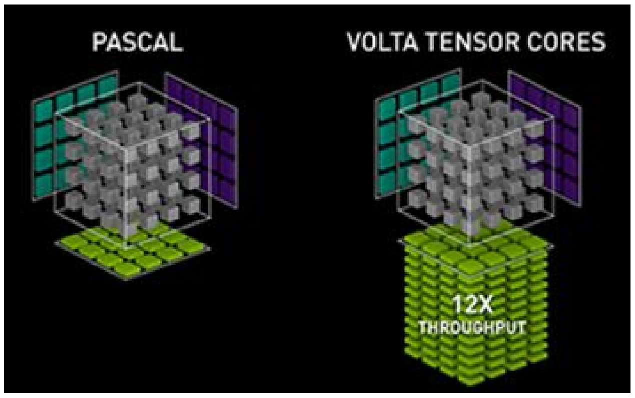
$$s_2(j) = \max_{i=1}^{m} \|x2_{fp32}(i,j)\| \quad : i = 1, ..., n1 \quad : j = 1, ..., n2$$
$$x2_{\vec{fp32}}(i,j) = \frac{x2_{\vec{fp32}(}i,j)}{s_2(j)}$$

Step 5: Convert the residual FP32 matrix x2f p32 into FP16 matrix x2f p16

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

IV. IMPLEMENTATION

Our experimental platform is a heterogeneous processor consisting of a CPU and a GPU. Our CPU card is and the GPU is NVIDIA Tesla V100 GPU. For reference another platform consisting of CPU and NVIDIA Pascal GPU was used.



Source: https://www.nvidia.com/en-us/data-center/tensorcore/

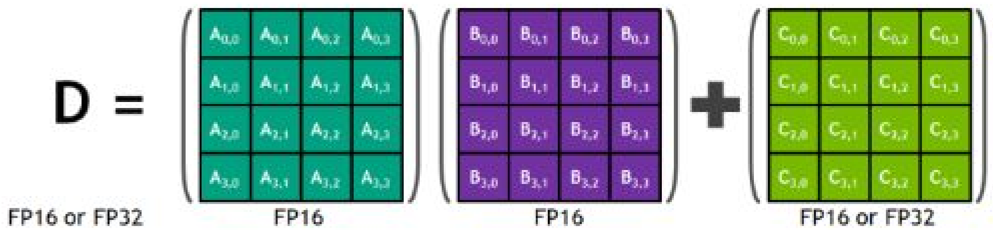


Figure 1. Tensor core 4*4*4 matrix multiply and accumulate. Source: https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/

Tensor cores give a 8x increase in throughput using half precision input. This has been utilized by cuBLAS API to accelerate matrix multiplication.

The following cuBLAS API kernels were used:

- cuBLAS <t> gemmEx
- cublasGemmStridedBatchedEx()

For further details on code, refer to bitbucket.

V. EXPERIMENTAL RESULTS

We evaluate our implementation by testing its performance on one NVIDIA®Tesla®V100 GPU. It has 640 tensor cores and 5120 CUDA cores, with 16 GB GPU memory and 900G GB/sec memory bandwidth.

We use CUDA events and the nvprof profiler to measure the execution time and analyze the splitting overhead. We also calculate its deviation from the FP32 cuFFT results and compare the accuracy with FP16 cuFFT.

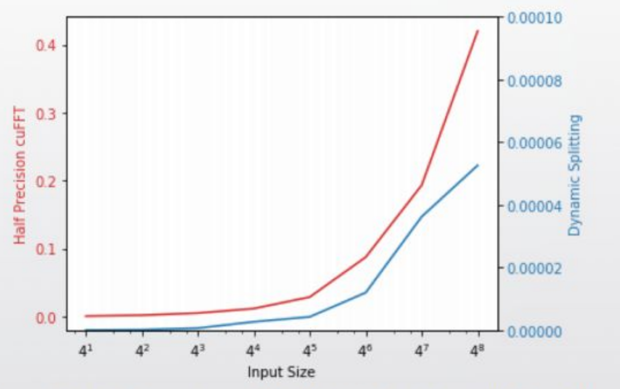
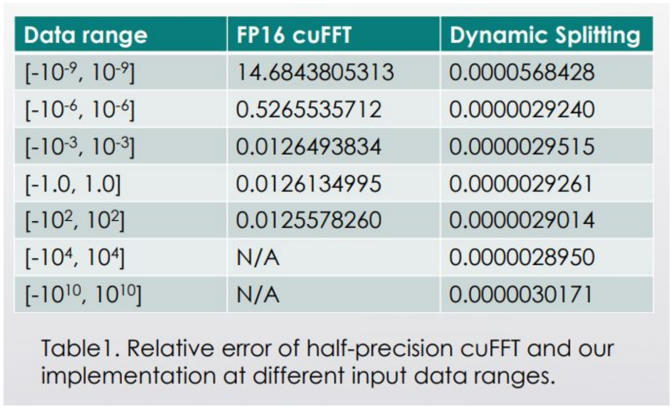


Figure1. Accuracy of half-precision cuFFT and our implementation. The average error of FP16 cuFFT is 7322x greater than our method.

The error statistics indicate that our implementation preserves high accuracy, even with growing input sizes



Data range	FP16 cuFFT	Dynamic Splitting
$[-10^{-9}, 10^{-9}]$	14.6843805313	0.0000568428
$[-10^{-6}, 10^{-6}]$	0.5265535712	0.0000029240
$[-10^{-3}, 10^{-3}]$	0.0126493834	0.0000029515
[-1.0, 1.0]	0.0126134995	0.0000029261
$[-10^{2}, 10^{2}]$	0.0125578260	0.0000029014
$[-10^{4}, 10^{4}]$	N/A	0.0000028950
$[-10^{10}, 10^{10}]$	N/A	0.0000030171

Table1. Relative error of half-precision cuFFT and our implementation at different input data ranges.

The dynamic splitting method preserves high accuracy over a wide range of inputs.

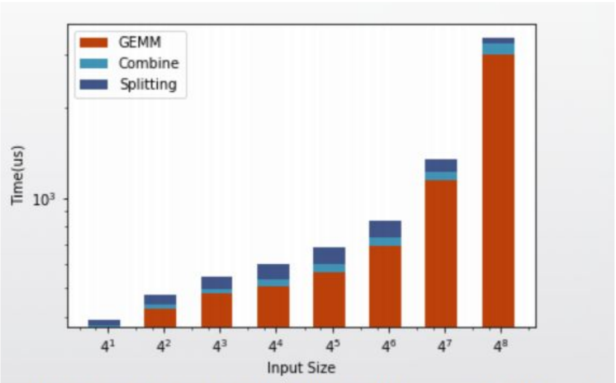


Figure1. Execution time breakdown at different input sizes. The matrix multiplication (by calling cublasGemmStridedBatchedEx) consumes around 90% of total time.

Compared with matrix multiplication, the time spent on splitting and combine is not significant.

VI. CONCLUSION

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 advantage with larger input size as the tensor core can be fully utilized and the setup cost can be amortized.

The speed of our implementation is inferior to cuFFT library. It can be optimized by implementing customized kernels. We also expect it to gain advantage with large input sizes, as tensor cores can be fully utilized and setup cost can be amortized.

Current implementation can handle a large variety of inputs. The relative error exceeds 0.1% or the program throws error when: Input data range ≥ 3 * 1010; or • Input data range ≤ 5 * 10-11 (Close to dynamic range of single precision number). The range may be further enlarged by pre-scaling all input numbers

VII. FUTURE WORK

There are several interesting directions for further optimizations:

1. 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. This may be improved by implementing transpose and GEMM kernels.

2. Another direction is to design an auto-tuning splitting algorithm that supports ill-conditioned inputs, and further optimizes the splitting overhead.
3. Also, our implementation of matrix transpose kernel has yet to take advantage of the shared memory. It can be accelerated by applying the "tiled" design.
4. A more sophisticated splitting algorithm may be designed. This could be done using bit manipulations.
5. The cuLASS library can be used, which shows higher performance than cuBLAS

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

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.
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.

V. Kumar, A. Grama, A. Gupta, and G. Karypis, Introduction to parallel computing: design and analysis of algorithms. Benjamin/Cummings Redwood City, 1994, vol. 400.

NVIDIA. (2018, Jul.) Cuda fast fourier transform library. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

J. Appleyard and S. Yokim. (2017, Oct.) Programming tensor cores in cuda 9. [Online]. Available: https://devblogs.nvidia.com/programmingtensor-cores-cuda-9

NVIDIA. (2017, Jul.) Nvidia tesla v100 gpu accelerator. [Online]. Available: http://www.nvidia.com/content/PDF/Volta-Datasheet.pdf

NVIDIA. (2017, Jul.) Nvidia tesla v100 gpu accelerator. [Online]. Available: http://www.nvidia.com/content/PDF/Volta-Datasheet.pdf