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

Kang, Yanming
Hong Kong University of Science and Technology
ykangaf@connect.ust.hk

Glaeser, Tullia
Tulane University
tglaeser@tulane.edu

Mentor: D'Azevedo, Eduardo
Oak Ridge National Laboratory
dazevedoef@ornl.gov

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

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

July 2019

1 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. We extend the algorithm to radix 2 and radix 8.

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[3] template library in our implementation.

The performance of our final implementation is similar to cuFFT, the FFT library provided by Nvidia, for small inputs. Our implementation also maintains high accuracy as the input grows in size.

2 Research Goal

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.

3 Introduction

The Discrete Fourier Transform (DFT) defined by $X(k) = \sum_{n=0}^{N-1} x(n)e^{-2\pi ink/N}$ transforms a sequence of N complex numbers $x[0 : N-1]$ (in time domain) to another sequence of N complex numbers $X[0 : N-1]$ (in frequency domain). The DFT is a linear operation and can be represented in matrix form $X = F_N x$, where F_N is the DFT matrix defined by $F_N(k,l) = e^{-2\pi ikl/N}$. The time complexity of DFT is $\mathcal{O}(N^2)$.

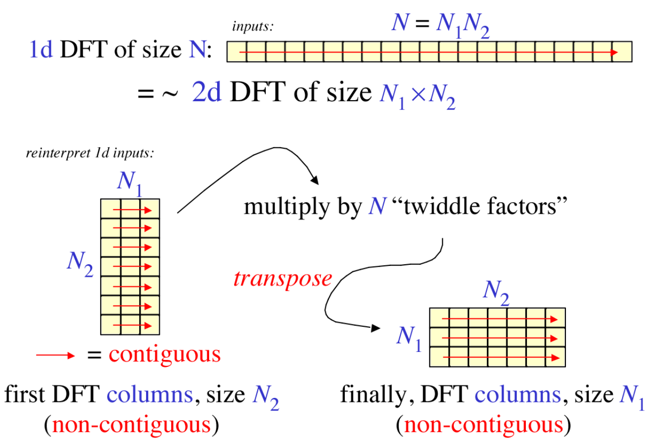
The Cooley-Tukey Fast Fourier Transform[2] computes the DFT with only $\mathcal{O}(N \log N)$ operations. Cooley–Tukey algorithms recursively re-express a DFT of a composite size $N = N_1 N_2$ by doing the following:

1. Perform N_1 DFTs of size N_2.
2. Multiply by complex roots of unity (often called the twiddle factors).

3. Perform N_2 DFTs of size N_1.

Typically, either N_1 or N_2 is a small factor (not necessarily prime), called the radix (which can differ between stages of the recursion). If N_1 is the radix, it is called a decimation in time (DIT) algorithm, whereas if N_2 is the radix, it is decimation in frequency (DIF, also called the Sande–Tukey algorithm).

The basic step of the Cooley–Tukey FFT for general factorizations is shown in Figure 1.



1d DFT of size N:
inputs:
$N = N_1 N_2$
$= \sim$ 2d DFT of size $N_1 \times N_2$
reinterpret 1d inputs:
N_1
N_2
multiply by N "twiddle factors"
transpose
N_2
N_1
= contiguous
first DFT columns, size N_2 (non-contiguous)
finally, DFT columns, size N_1 (non-contiguous)

Figure 1: Cooley-Tukey FFT

As previously stated, we are also using NVIDIA's Tensor Core structure. The Tensor Cores on Nvidia Tesla architecture GPUs 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++.

The Tesla V100 GPU contains 640 Tensor Cores: 8 per SM. Tensor Cores and their associated data paths are custom-crafted to dramatically increase floating-point compute throughput at only modest area and power costs. Each Tensor Core provides a 4 by 4 by 4 matrix processing array which performs the operation $D = A * B + C$, where A, B, C and D are 4 by 4 matrices as Figure 2 shows. The matrix multiply inputs A and B as FP16 matrices, while the accumulation matrices C and D may be either FP16 or FP32.

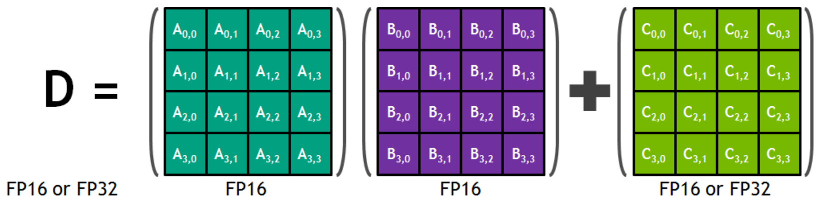


Figure 2: Tensor Core

CUTLASS (CUDA Templates for Linear Algebra Subroutines) is a collection of CUDA C++ template abstractions for implementing high-performance matrix-multiplication (GEMM) at all levels and scales within CUDA. CUTLASS provides support for mixed-precision computations, providing specialized data-movement and multiply-accumulate abstractions for 8-bit integer, half-precision floating point, single-precision floating point, and double-precision floating point types. Furthermore, CUTLASS demonstrates CUDA's WMMA API for targeting the programmable, high-throughput Tensor Cores provided by NVIDIA's Volta architecture and beyond.

4 Algorithm

4.1 FFT Algorithm

We first adopt the radix-4 Cooley-Tukey algorithm because the DFT matrix F_4 can be exactly represented in FP16 (half-precision) without loss of precision; it simply has a real and imaginary part, both composed of 1s, 0s, and -1s. The real and imaginary Fourier matrices are defined as:

$$F_{Nreal}[l,k] = \cos(2\pi kl/N) \tag{1}$$

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

By using the radix-4 decimation-in-time (DIT) algorithm, only allowing $N = 4$, we see the following are the real and imaginary Fourier matrices:

$$F_{4real} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & -1 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix} \tag{3}$$

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

When using the radix-4, the number of data points, N, must be a power of 4 ($N = 4^V$) and the input sequence must be split into four subsequences, $x(4n)$, $x(4n+1)$, $x(4n+2)$, $x(4n+3)$, where $n = 0, 1, \ldots, N/4 - 1$. The radix-4 matrix is also ideal since we are using tensor cores which are built to perform 4x4 matrix-matrix multiply; this ends up fitting perfectly for radix-4. The radix-4 algorithm is shown in Figure 3.

The radix-8 algorithm is another great option; it is similar to the radix-4 algorithm except the base is changed from 4 to 8 where $N = 8M$. Using the same Fourier matrix equations from above with $N = 8$, we can derive the real and imaginary radix-8 Fourier matrices.

Finally, the radix-2 algorithm is also an option that could be further explored. Its DFT matrix F_2 can be exactly represented in FP16 as a real matrix (no complex part).

For all of these algorithms our code mainly changes by changing the division factor of N and multiplication factor of the batch size to the radix number everytime we call the function recursively; we also modify the base case to be called when $N =$ the radix number.

4.2 Splitting Algorithm

Each gemm operation is done in fp16. To maintain high precision, a splitting is done before each gemm and a combining after. Before recursion step of radix-4 1-d FFT, where a multiplication of X with F_4, where X is single precision and has shape $(M, 4, batch)$, is needed, we split X into two $(M, 4, batch)$ input arrays and two $(M, batch)$ scale factor arrays in half precision, as shown in equation (5) and (6). The two scales are determined by the magnitudes of the vector. The splitting algorithm of each 1×4 vector is shown as algorithm 1.

$$X_{32} = scale1 * X_{16hi} + scale2 * X_{16lo} \tag{5}$$

$$X_{32} \cdot F_4 = (scale1 * X_{16hi}) \cdot F_4 + (scale2 * X_{16lo}) \cdot F_4 \tag{6}$$

5 Implementation

5.1 Kernels

The CUDA kernels in last year's base programs (which we based our current code off of) are not efficient in terms of the ratio of computation to memory access. We reduced the number of global memory read/write from 46 to 14 in the splitting kernel, and from 26 to 14 in the accumulating kernel.

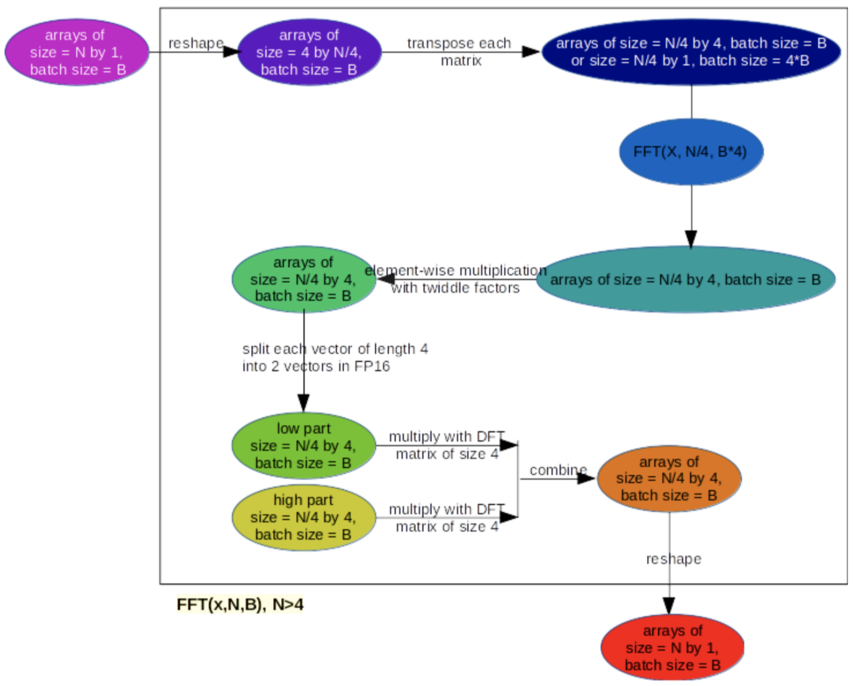

arrays of size = N by 1, batch size = B
reshape
arrays of size = 4 by N/4, batch size = B
transpose each matrix
arrays of size = N/4 by 4, batch size = B or size = N/4 by 1, batch size = 4*B
FFT(X, N/4, B*4)
arrays of size = N/4 by 4, batch size = B
element-wise multiplication with twiddle factors
arrays of size = N/4 by 4, batch size = B
split each vector of length 4 into 2 vectors in FP16
low part size = N/4 by 4, batch size = B
high part size = N/4 by 4, batch size = B
multiply with DFT matrix of size 4
multiply with DFT matrix of size 4
combine
arrays of size = N/4 by 4, batch size = B
reshape
FFT(x,N,B), N>4
arrays of size = N by 1, batch size = B

Figure 3: Radix-4 FFT Flow Chart

Algorithm 1 *Split*(X, n)

```
$scale1 \leftarrow 0.0f$
$scale2 \leftarrow 0.0f$
**for** $i = 0 : n-1$ **do**
    $scale1 \leftarrow (\text{float})\max(scale1, \text{abs}(X[i]))$
**end for**
**for** $i = 0 : n-1$ **do**
    $X_{hi}[i] \leftarrow (\text{half})X[i]/scale1$
**end for**
**for** $i = 0 : n-1$ **do**
    $tmp[i] \leftarrow X[i] - scale1 * (\text{float})X_{hi}[i]$
**end for**
**for** $i = 0 : n-1$ **do**
    $scale2 \leftarrow (\text{float})\max(scale2, \text{abs}(tmp[i]))$
**end for**
**for** $i = 0 : n-1$ **do**
    $X_{lo}[i] \leftarrow (\text{half})tmp[i]/scale2$
**end for**
**return** $X_{hi}, X_{lo}, scale1, scale2$
```

5.2 GEMM

In our recursive algorithm for each invocation of the recursion body we need to compute a batched gemm of dimension M by 4 by 4 (radix-4), and batch size B. If the input has size N, M changes from N/4 to 4 and B changes from 4 to N/4 as the recursion goes deeper.

In the previous implementation the gemms are computed using the cublasGemmStridedBatchedEx API call in the cuBlas library. We found in our profilings that the cuBlas kernel volta_sgemm_fp16_128x64_nn accounts for 50% to 80% of the GPU computation time as the input size changes. The theoretical usage of computation power of the GPU, in terms of FLOPs is lower than 5%. The kernel name suggests it does a large matrix multiplication, which could waste many FLOPs and cause inefficiency.

In our experiments, we also found that the cuBlas API cublasGemmStridedBatchedEx incurs large errors when batch size exceeds 65534. Therefore, a splitting in batches and serial executions of multiple gemms of smaller batch sizes are needed when the input grows in size. However, this type of serial behaviour is extremely slow.

The CUTLASS library provides C++ class templates for using the namespace nvcuda::wmma (warp matrix multiply-accumulate), which is an abstraction of computation on Tensor Cores. Briefly the following steps are performed on each warp.

1. Fill fragments a and b using data in matrices A and B, each 4 by 4, in half precision
2. Fill fragment c using data in matrix C, 4 by 4, in single precision
3. Perform multiply-add and accumulate on c
4. Synchronize and write back to C

A batched strided gemm subroutine then can be built using these templates. However, the batch size limit of 64k also exists here. After examining the CUTLASS source code, we found that the limit is because the maximum grid size in z-axis on V100 is 64k. The size limit in x-axis is 2048M, which is sufficient for inputs that fit in 16GB memory. The batch size limit is eliminated after swapping the x and z dimension.

5.3 2-D and 3-D FFT

The 2-D and 3-D FFT are based on 1-D FFT. The 2-D version does the following, on input of shape $(m, n, batch)$

1. Perform $n * batch$ point-m 1-D FFTs.
2. Permute the first two dimensions, result in shape $(n, m, batch)$.
3. Perform $m * batch$ point-n 1-D FFTs.
4. Permute the first two dimensions, result in shape $(m, n, batch)$.

The 3-D version does the following, on input of shape $(m, n, k, batch)$

1. Perform $n * k * batch$ point-m 1-D FFTs.
2. Permute the first two dimensions, result in shape $(n, m, k, batch)$.
3. Perform $m * k * batch$ point-n 1-D FFTs.
4. Permute the first three dimensions, result in shape $(k, n, m, batch)$.
5. Perform $n * m * batch$ point-k 1-D FFTs.
6. Permute, result in shape $(m, n, k, batch)$.

The permutation operation is modified from a magma[7] batched transpose kernel. The 3-D algorithm is shown in Figure 4.

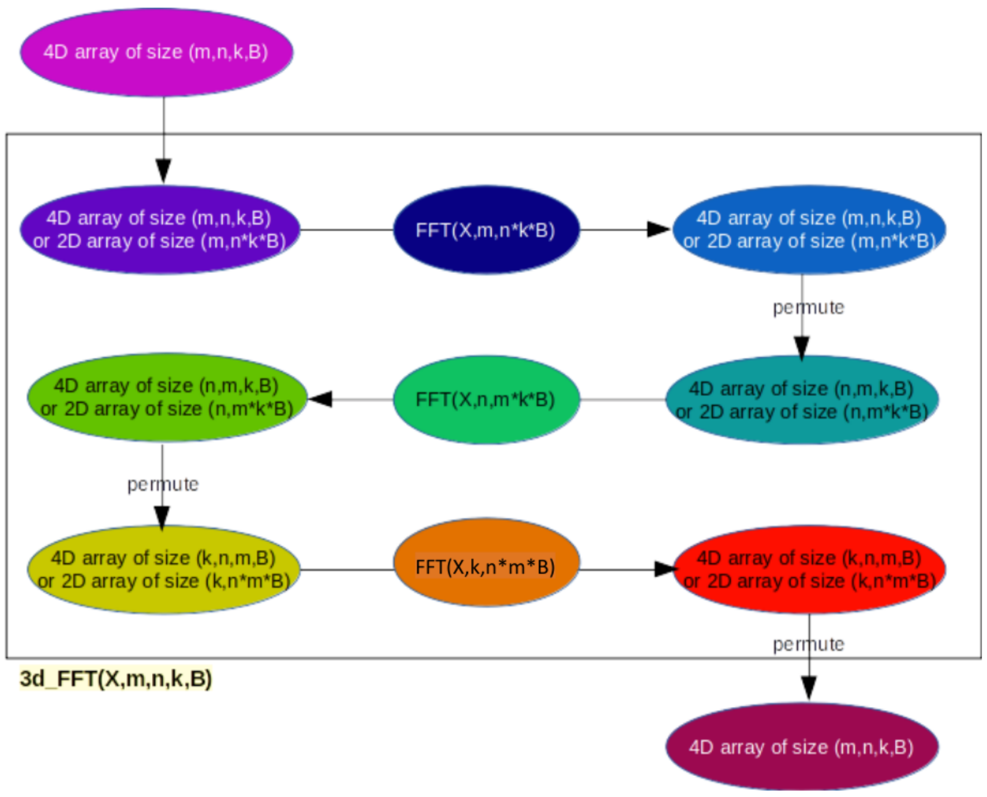

4D array of size (m,n,k,B)
4D array of size (m,n,k,B) or 2D array of size (m,n*k*B)
FFT(X,m,n*k*B)
4D array of size (m,n,k,B) or 2D array of size (m,n*k*B)
permute
4D array of size (n,m,k,B) or 2D array of size (n,m*k*B)
FFT(X,n,m*k*B)
4D array of size (n,m,k,B) or 2D array of size (n,m*k*B)
permute
4D array of size (k,n,m,B) or 2D array of size (k,n*m*B)
FFT(X,k,n*m*B)
4D array of size (k,n,m,B) or 2D array of size (k,n*m*B)
permute
3d_FFT(X,m,n,k,B)
4D array of size (m,n,k,B)

Figure 4: 3-D FFT Flow Chart

5.4 Radix-2

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{7}$$

has no imaginary part and can be represented in half precision without error. However, multiplication of matrices of X (size M-by-2) and F_2 cannot be done directly with nvcuda::wmma. Instead, we construct a 4-by-4 real matrix by putting together two F_2 matrices on the diagonal and combine every two M-2-2 multiplications in one M-4-4 multiplication, as shown below.

$$F_{2diag} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \tag{8}$$

$$\begin{bmatrix} \\ X_1 \\ \\ \end{bmatrix}_{M\times 2} [F_2]; \begin{bmatrix} \\ X_2 \\ \\ \end{bmatrix}_{M\times 2} [F_2] \equiv \begin{bmatrix} \\ X_1|X_2 \\ \\ \end{bmatrix}_{M\times 4} [F_{2diag}] \tag{9}$$

6 Results

N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
1k	2.46	3.23	$5.09 * 10^{-3}$	2.43	$7.75 * 10^{-7}$
4k	2.47	3.25	$5.04 * 10^{-3}$	2.45	$7.84 * 10^{-7}$
16k	2.52	3.27	$5.03 * 10^{-3}$	2.56	$7.83 * 10^{-7}$
32k	2.53	2.43	$5.04 * 10^{-3}$	3.22	$7.81 * 10^{-7}$
64k	2.73	3.40	$5.03 * 10^{-3}$	3.68	$7.80 * 10^{-7}$
256k	4.94	3.71	$5.04 * 10^{-3}$	8.09	$7.82 * 10^{-7}$
1024k	7.71	5.93	$5.04 * 10^{-3}$	14.42	$7.82 * 10^{-7}$
2048k	11.95	7.93	$5.05 * 10^{-3}$	24.90	$7.82 * 10^{-7}$
4096k	19.24	12.47	$5.01 * 10^{-3}$	33.93	$7.81 * 10^{-7}$
16384k	63.93	39.98	$4.55 * 10^{-3}$	111.81	$7.06 * 10^{-7}$
65536k	242.50	151.00	$2.28 * 10^{-3}$	425.91	$3.6 * 10^{-7}$

Table 1: 1-D radix-2 FFT results

N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
1k	3.13	4.27	$5.10 * 10^{-3}$	2.30	$1.04 * 10^{-6}$
4k	3.00	3.40	$5.06 * 10^{-3}$	2.32	$1.05 * 10^{-6}$
16k	3.76	4.60	$1.26 * 10^{-2}$	2.42	$3.36 * 10^{-6}$
64k	2.77	3.43	$1.27 * 10^{-2}$	3.58	$3.36 * 10^{-6}$
256k	5.35	3.96	$2.94 * 10^{-2}$	7.58	$5.99 * 10^{-6}$
1024k	8.98	6.68	$2.95 * 10^{-2}$	14.03	$6.00 * 10^{-6}$
4096k	19.80	12.52	$8.70 * 10^{-2}$	30.81	$1.55 * 10^{-5}$
16384k	63.46	39.83	$9.03 * 10^{-2}$	99.20	$1.85 * 10^{-5}$
65536k	251.84	155.92	$9.03 * 10^{-2}$	381.51	$1.85 * 10^{-5}$

Table 2: 1-D radix-4 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

M*N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
4k	2.44	3.22	$5.71 * 10^{-2}$	2.57	$1.20 * 10^{-5}$
16k	2.94	3.48	$2.79 * 10^{-2}$	3.41	$6.21 * 10^{-6}$
64k	3.12	3.68	$3.75 * 10^{-2}$	4.81	$8.49 * 10^{-6}$
256k	6.02	4.35	$5.70 * 10^{-2}$	10.45	$1.20 * 10^{-5}$
1024k	8.46	6.18	$1.45 * 10^{-1}$	18.70	$3.05 * 10^{-5}$
4096k	19.79	12.63	$2.25 * 10^{-1}$	49.22	$4.52 * 10^{-5}$
16384k	65.39	41.73	$2.97 * 10^{-1}$	177.84	$6.38 * 10^{-5}$
65536k	240.89	156.23	$2.97 * 10^{-1}$	924.24	$6.38 * 10^{-5}$

Table 4: 2-D radix-2 FFT results

M*N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
4k	2.45	3.23	$5.59 * 10^{-2}$	2.74	$1.43 * 10^{-5}$
16k	2.59	2.58	$5.72 * 10^{-2}$	2.98	$1.45 * 10^{-5}$
64k	2.88	3.50	$5.68 * 10^{-2}$	4.24	$1.45 * 10^{-5}$
256k	5.36	3.98	$5.70 * 10^{-2}$	8.13	$1.44 * 10^{-5}$
1024k	8.28	6.28	$1.53 * 10^{-1}$	16.72	$3.53 * 10^{-5}$
4096k	19.85	12.62	$3.27 * 10^{-1}$	42.16	$8.15 * 10^{-5}$
16384k	64.04	40.37	$2.97 * 10^{-1}$	142.80	$7.38 * 10^{-5}$
65536k	257.85	160.44	$3.34 * 10^{-1}$	597.95	$9.90 * 10^{-5}$

Table 5: 2-D radix-4 FFT results

M*N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
4k	2.45	3.29	$5.68 * 10^{-2}$	2.70	$7.57 * 10^{-3}$
32k	2.57	2.48	$5.69 * 10^{-2}$	3.09	$7.62 * 10^{-3}$
256k	5.12	3.77	$5.70 * 10^{-2}$	7.20	$7.60 * 10^{-3}$
2048k	11.67	7.82	$1.92 * 10^{-1}$	18.90	$2.62 * 10^{-2}$
16384k	63.79	39.98	$1.74 * 10^{-1}$	95.17	$2.38 * 10^{-2}$

Table 6: 2-D radix-8 FFT results

K*M*N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
16k	2.79	3.48	$2.07 * 10^{-5}$	6.30	$3.56 * 10^{-9}$
64k	3.00	3.74	$4.04 * 10^{-5}$	8.15	$7.29 * 10^{-9}$
256k	5.40	3.83	$3.87 * 10^{-5}$	10.02	$9.02 * 10^{-9}$
1024k	8.07	6.02	$2.79 * 10^{-5}$	21.08	$6.05 * 10^{-9}$
4096k	19.47	12.63	$1.87 * 10^{-5}$	60.98	$3.92 * 10^{-9}$
16384k	68.51	40.11	$1.41 * 10^{-5}$	230.54	$2.92 * 10^{-9}$
65536k	259.13	148.14	$1.23 * 10^{-5}$	956.44	$2.62 * 10^{-9}$

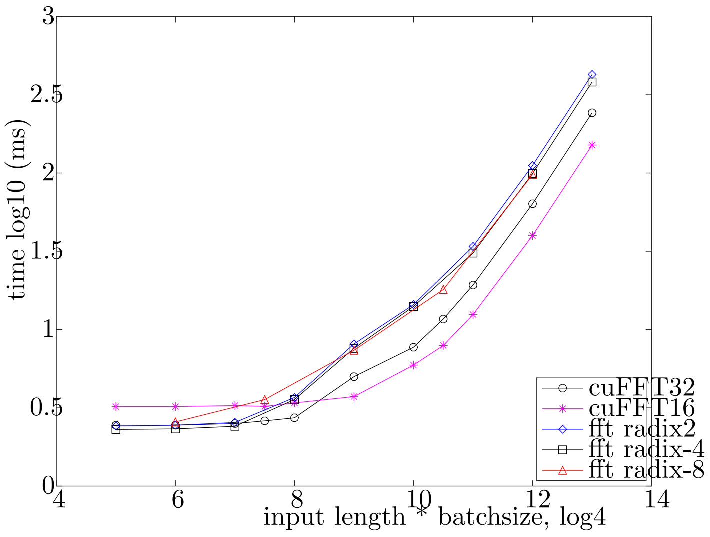
Table 7: 3-D radix-2 FFT results

K*M*N* batchSize	cuFFT32 time	cuFF16 time	cuFFT16 error	accelerated FFT time	accelerated FFT error
16k	2.82	3.51	$2.25 * 10^{-5}$	5.96	$4.08 * 10^{-9}$
64k	2.97	3.57	$4.14 * 10^{-5}$	7.13	$9.65 * 10^{-9}$
256k	5.51	4.02	$9.76 * 10^{-5}$	8.57	$2.47 * 10^{-8}$
1024k	8.46	6.31	$2.62 * 10^{-5}$	18.37	$6.05 * 10^{-9}$
4096k	19.99	12.83	$6.56 * 10^{-6}$	49.12	$1.61 * 10^{-9}$
16384k	69.67	40.56	$1.39 * 10^{-5}$	177.62	$3.56 * 10^{-9}$
65536k	269.77	153.99	$3.69 * 10^{-5}$	727.22	$8.58 * 10^{-9}$

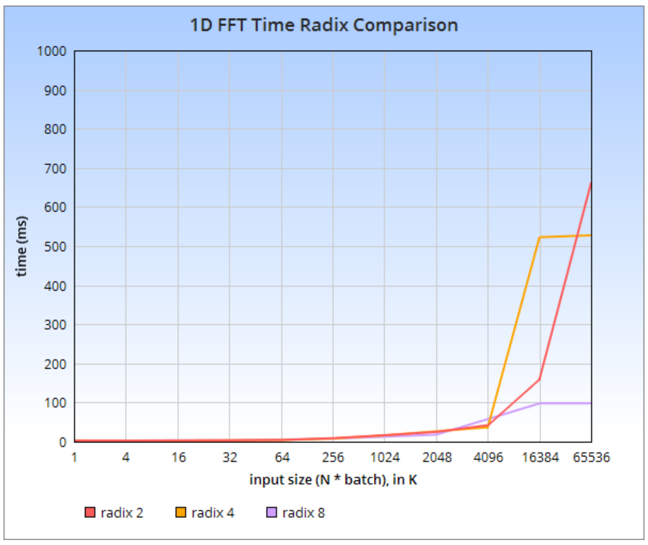
Table 8: 3-D radix-4 FFT results

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	$1.85 * 10^{-6}$

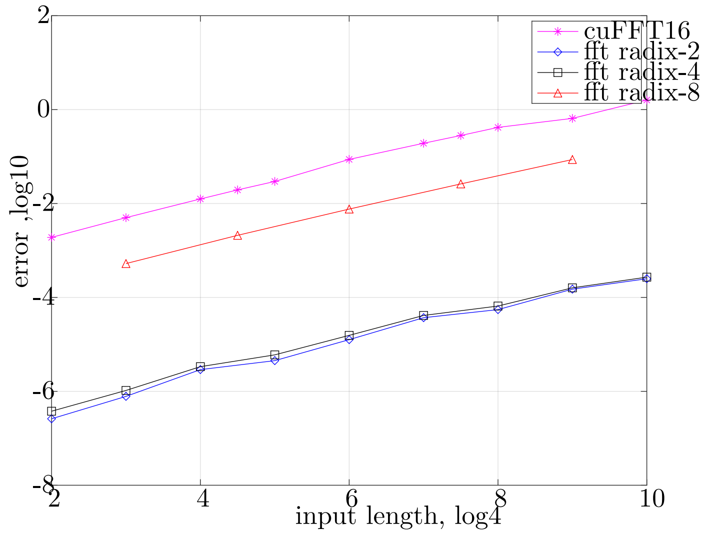
Table 9: 3-D radix-8 FFT results



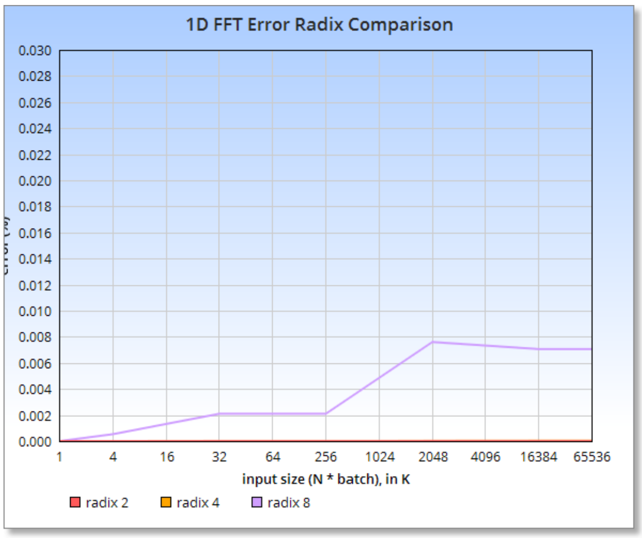
(a) Computation Time Comparison



(b) 1-D FFT Time Radix Comparison

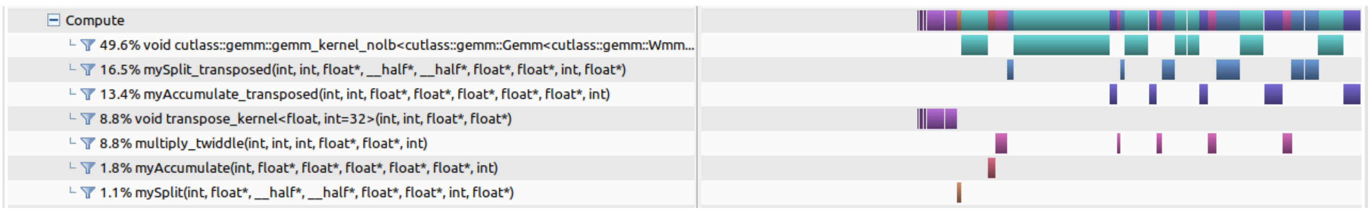


(c) Computation Error Comparison, cuFFT32 is considered accurate

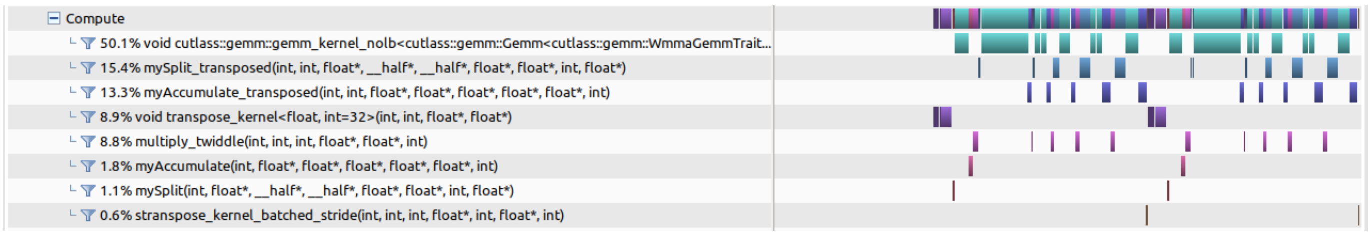


(d) 1-D FFT Error Radix Comparison

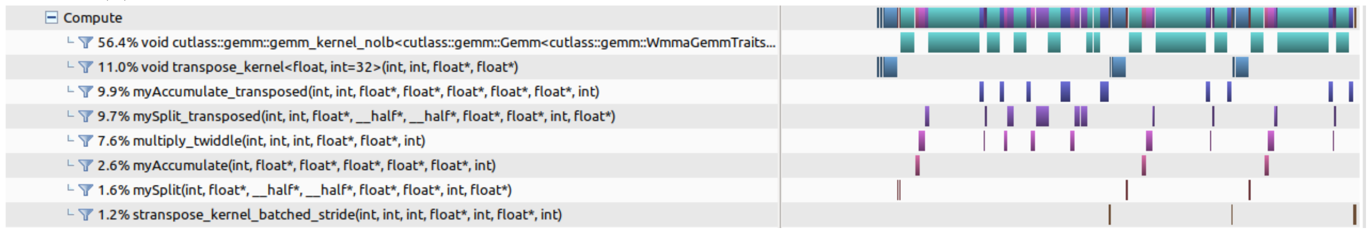
As shown in the 1D line graphs above, the radix 8 algorithm is the fastest but also has the largest error; even so, the error up to input size 65536K does not surpass 0.0262. This is also reflected in the 2D and 3D implementations of radix 8.


Compute
49.6% void cutlass::gemm::gemm_kernel_nolb<cutlass::gemm::Gemm<cutlass::gemm::Wmm...
16.5% mySplit_transposed(int, int, float*, __half*, __half*, float*, float*, int, float*)
13.4% myAccumulate_transposed(int, int, float*, float*, float*, float*, float*, int)
8.8% void transpose_kernel<float, int=32>(int, int, float*, float*)
8.8% multiply_twiddle(int, int, int, float*, float*, int)
1.8% myAccumulate(int, float*, float*, float*, float*, float*, int)
1.1% mySplit(int, float*, __half*, __half*, float*, float*, int, float*)

(a) 1-D Radix-4 FFT Profiling Result


Compute
50.1% void cutlass::gemm::gemm_kernel_nolb<cutlass::gemm::Gemm<cutlass::gemm::WmmaGemmTrait...
15.4% mySplit_transposed(int, int, float*, __half*, __half*, float*, float*, int, float*)
13.3% myAccumulate_transposed(int, int, float*, float*, float*, float*, float*, int)
8.9% void transpose_kernel<float, int=32>(int, int, float*, float*)
8.8% multiply_twiddle(int, int, int, float*, float*, int)
1.8% myAccumulate(int, float*, float*, float*, float*, float*, int)
1.1% mySplit(int, float*, __half*, __half*, float*, float*, int, float*)
0.6% stranspose_kernel_batched_stride(int, int, int, float*, int, float*, int)

(b) 2-D Radix-4 FFT Profiling Result


Compute
56.4% void cutlass::gemm::gemm_kernel_nolb<cutlass::gemm::Gemm<cutlass::gemm::WmmaGemmTraits...
11.0% void transpose_kernel<float, int=32>(int, int, float*, float*)
9.9% myAccumulate_transposed(int, int, float*, float*, float*, float*, float*, int)
9.7% mySplit_transposed(int, int, float*, __half*, __half*, float*, float*, int, float*)
7.6% multiply_twiddle(int, int, int, float*, float*, int)
2.6% myAccumulate(int, float*, float*, float*, float*, float*, int)
1.6% mySplit(int, float*, __half*, __half*, float*, float*, int, float*)
1.2% stranspose_kernel_batched_stride(int, int, int, float*, int, float*, int)

(c) 3-D Radix-4 FFT Profiling Result

Errors are calculated using

$$\textbf{Error} = \sum_{k=0}^{N*B} \text{abs}(XF_{32}(k) - XF(k))/\text{norm} \tag{10}$$

where norm is the bound of random generated X.

$$-\text{norm} \leq X(k) \leq \text{norm}, \forall k < N * B \tag{11}$$

7 Conclusion

We have programmed and implemented a FP32-FP16 mixed precision FFT algorithm, taking advantage of the recent tensor core hardware. Our program effectively emulates the built-in single-precision cuFFT calculation, producing highly accurate results from various inputs. In our code, we provide the radix 2, radix 4, and radix 8 methods, as well as the algorithms for 1D, 2D, and 3D inputs. By replacing last year's cuBLAS library with CUDA's CUTLASS template library, we were able to speed up the program further.

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. As our input size grows, our program also gains an advantage since the tensor cores can be fully utilized and the setup cost compensated for. Our algorithm can be further optimized by implementing customized kernels.

8 Future Work

There are still several interesting directions for further optimizations:

1. Try a split-radix algorithm, combining two or more different radices. For example, one idea is to combine the radix-4 and radix-8 algorithms.

2. Manipulate the code or use different memory allocation tricks to be able to take larger input sizes.

3. Hide memory latency by overlapping FFT and memcpy (H2D, D2H), by splitting batch size and using multiple streams.
4. Provide support to inputs of composite sizes (currently only supporting powers of 2, 4,and 8)

9 Acknowledgement

This research project was sponsored by the National Science Foundation through the Research Experience for Undergraduates (REU) award, with additional support from the Joint Institute of Computational Sciences at the 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] 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.