2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT) | 978-1-6654-4278-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/PACT52795.2021.00032

Accelerating Fourier and Number Theoretic Transforms using Tensor Cores and Warp Shuffles

Sultan Durrani*, Muhammad Saad Chughtai†, Mert Hidayetoglu*, Rashid Tahir‡, Abdul Dakkak*,
Lawrence Rauchwerger*, Fareed Zaffar§, Wen-mei Hwu*
University of Illinois at Urbana-Champaign*,
Georgia Institute of Technology†, University of Prince Mugrin‡,
Lahore University of Management Sciences§
Email: {sultand2, hidayet2, dakkak, rwerger, hwu}@illinois.edu, chughtai@gatech.edu, r.tahir@upm.edu.sa,
fareed.zaffar@lums.edu.pk

***Abstract*—The discrete Fourier transform (DFT) and its specialized case, the number theoretic transform (NTT), are two important mathematical tools having applications in several areas of science and engineering. However, despite their usefulness and utility, their adoption continues to be a challenge as computing the DFT of a signal can be a time-consuming and expensive operation. To speed things up, fast Fourier transform (FFT) algorithms, which are reduced-complexity formulations for computing the DFT of a sequence, have been proposed and implemented for traditional processors and their corresponding instruction sets. With the rise of GPUs, NVIDIA introduced its own FFT computation library called cuFFT, which leverages the power of GPUs to compute the DFT. However, as this paper demonstrates, there is a lot of room for improvement to accelerate the FFT and NTT algorithms on modern GPUs by utilizing specialized operations and architectural advancements. In particular, we present four major types of optimizations that leverage tensor cores and the warp-shuffle instruction. Through extensive evaluations, we show that our approach consistently outperforms existing GPU-based implementations with a speedup of up to $4\times$ for NTT and a speed of up to $1.5\times$ for FFT.**

***Index Terms*—DFT, FFT, NTT, GPU, Tensor Cores, cyphertext, homomorphic encryption**

I. Introduction

Fast Fourier transform (FFT), a reduced computational complexity formulation of the discrete Fourier transform (DFT), has been applied extensively to a wide range of applications ranging from image filtering to differential equation solvers. For example, Digital Signal Processors (DSP) with specialized hardware support for efficient FFT computation have become an essential part of the mobile System-on-Chip (SoC) to support noise suppression, voice enhancement, etc [1].

Similarly, number theoretic transform (NTT), a specialized DFT for integers, is vital in Fully Homomorphic Encryption (FHE). FHE is applicable in a number of applications including private search, confidential information retrieval and secure biometric voting [2]. Indeed, FHE is a game-changer and has started gaining more traction recently with governments and industry [3]. However, most existing FHE-based proposals lack widespread adoption due to significant performance and overhead challenges. Given the diverse spectrum of applications for NTT and FFT algorithms, it is imperative to improve the performance of these functions.

To this end, attempts have been made by both the computer architecture [4] and the software engineering [5] communities to speed up the execution times and reduce runtime overheads. Similarly, the industry has developed its own solutions that leverage the power of parallel computing platforms. For instance, NVIDIA recently introduced the cuFFT library [6], a specialized API to quickly leverage the floating-point power and parallelism of NVIDIA GPUs. However, despite these positive developments, more work is needed to speedup these algorithms further (as evidenced by the slow adoption of FHE) and facilitate widespread usage in everyday applications.

Hence, in this paper, we focus on optimizing both FFT and NTT by utilizing the recently introduced GPU Tensor Cores [7] and Warp-shuffle instruction. The main strength of the Tensor Cores is their very high compute throughput and operand supply bandwidth for modest-sized matrices. Furthermore, since real-world signal data are typically produced by sensors that have limited precision, the low-precision nature of the Tensor Cores can be sufficient for many real-world applications. However, the cores cannot be utilized off-the-shelf for DFT computation as some preprocessing is needed. For instance, one of the major challenges is to transform large-sized DFT operations into a collection of modest-sized DFT operations that can be collectively performed as a small number of modest-sized matrix multiplications on Tensor Cores. Hence, one of the main contributions of this paper is an iterative version of the Cooley-Tukey FFT algorithm for systematically converting DFT operations of a wide range of sizes into collections of modest-sized matrix multiplication operations that fit well into Tensor Cores. Importantly, the proposed conversion process preserves the reduced-complexity nature FFT algorithms. Moreover, our implementation is augmented by the clever use of the Warp-shuffle instruction to exchange data directly between threads without additional copy overhead. In summary, this paper makes the following four primary contributions in optimizing FFT and NTT algorithms in a novel way for efficient execution on GPUs:

- An iterative version of the Cooley-Tukey FFT algorithm, which smartly decomposes large DFT computations into sequences of parallel, small baseline DFT computations

978-1-6654-4278-7/21/$31.00 ©2021 IEEE
DOI 10.1109/PACT52795.2021.00032

that can fit well into Tensor Cores while preserving the low complexity of the FFT algorithm. The proposed decomposition, utilizes the Tensor Cores in an ideal way and prevents both unnecessary spillover operations and capacity wastage.
- Warp-shuffle-XOR (WSX) based FFT butterfly computation that allows the shuffle hardware to complement the Tensor Cores for some of the DFT computation generated by Cooley-Tukey that are too small to make good use of the Tensor Cores. This also plays a key role in accelerating NTT execution times.
- Using constant memory to store bit reversal and twiddle factor lookup tables to drastically reduce the use of registers and improve occupancy of the SMs (Streaming Multiprocessors).
- Efficiently utilizing shared memory (on-chip scratchpad) to completely remove global memory (DRAM) accesses, which can be time-consuming and expensive, within the inner core of the algorithm and alleviate memory bandwidth bottleneck.

These key optimizations allow our approach to consistently outperform existing-GPU based NTT implementations with a speedup of up to $4\times$ and cuFFT with a speed of up to $1.5\times$. We also demonstrate the impact of more routine optimizations such as using CUDA streams and asynchronous data copies to overlap communication with computation. In the following sections we first go over GPU architecture and CUDA Programming Model and show critical Mathematical formulations necessary for understanding the FFT and NTT algorithms.

II. Background

A. An Overview of GPUs

Over the past decade, GPUs have continued to evolve from specialized devices for graphics rendering to accelerators and compute engines for scientific applications and machine learning workloads. For instance, to improve the efficiency of machine learning computations, NVIDIA introduced Tensor Cores in their Volta Architecture to perform very fast matrix multiplication operations for low-precision data formats. In their latest A100 GPUs, the peak single-precision floating-point compute throughput is 19.5 TFLOPS without Tensor Cores compared to 156 TFLOPS for GPUs with Tensor Cores [8]. This is indeed a game changing boost in throughput . Figure 1 shows the internals of the Streaming Multi-processor (SM) of the latest Ampere GPU. Each SM consists of four sub-cores which in turn contain two Tensor Cores (TC) each. The intent of the Tensor Cores has been to accelerate Machine and Deep Learning training and inference but they also hold potential for a wider class of algorithmic applications, a theme that is explored in our work. As our contributions involve low-level optimizations, it is imperative to first understand the execution and memory models of CUDA, Tensor Cores and the Warp-shuffle instruction. Hence we present a brief overview below:

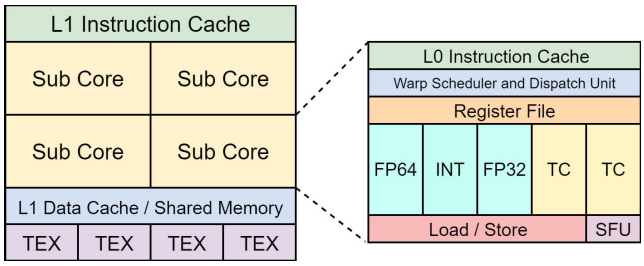

Fig. 1. NVIDIA Ampere architecture Streaming Multiprocessor (SM) and the internals of a sub-core showing Tensor Cores (TC) .

1) CUDA Execution Model: Compute Unified Device Architecture (CUDA) is a parallel programming platform that enables NVIDIA GPUs to execute programs written in C, C++, etc. Although the methods described in this paper are not limited to CUDA, all implementation examples are in CUDA and experiments are conducted in a system with CUDA hardware and software. Here, we will highlight a few CUDA terms that will be important for understanding the rest of the paper.

A CUDA program is divided into a host and device part where the host functions typically run on the CPU and the device functions run on the GPU. The NVCC compiler provided by NVIDIA separates both parts during compilation. There are three types of function declarations, i.e, host, global and device. The global function can be called from the host or the device but runs on the device while the host and device functions are only callable from the host and device respectively (as implied by the name). The global function execution initiation is also known as a CUDA Kernel launch. There is a hierarchy of threads associated with the global function when it is launched on the device. The outermost thread is called a grid which contains one or more blocks which in turn contain threads. A grid is mapped on the whole of the GPU while the block is mapped on a single Streaming Multiprocessor (SM). Threads in a block can access data in a Shared Memory whose contents are local to the block. The SM executes threads in groups of 32 called warps, similar to a Single Instruction Multiple Data (SIMD) execution. In the pre-Volta architectures, every warp would have a single program counter (PC) and call stack (S). Starting with the Volta generation, there is an individual PC and S per thread in order to allow equal concurrency between all threads, regardless of warp. The GPU can launch one or more of these CUDA Kernels asynchronously [9].

2) CUDA Memory Model: The CUDA Memory hierarchy is also a fundamental part of the system. It contains three levels or layers, i.e, *Global*, *Constant* and *Texture* memories, which can be seen by all threads of the grid. The Global memory is the slowest as it is farthest away from the execution engines/cores among the three. The Constant memory is the fastest among the three as it is a read-only memory which is heavily cached close to the execution engines/cores. The Texture memory is also a read-only memory whose memory

traffic is routed through the texture cache, which is optimized for 2D spatial locality. When a thread block is executed on an SM, the SM assigns an on-chip Shared Memory to the thread block, which is private to the thread block and shared among all threads of the block. Finally, every thread has access to its own pool of private registers, which are closest to the ALUs and clearly the fastest.

3) Tensor Cores: The CUDA Tensor Cores were introduced starting with the Volta architecture. Each Streaming Multiprocessor has four sub-cores wherein each sub-core has two Tensor Cores. The primary operation that the Tensor Cores perform is matrix "multiply and accumulate" i.e., ($D = A \cdot B + C$). These Tensor Cores have evolved over the years and can now support multiple low precision and short types including Floating Point (FP) 16 (half-precision) and char as input. The *Warp Matrix Multiply Accumulate* (WMMA) API allows access to Tensor Core programming. Programmers use API functions to load, compute, and store fragments. The following three functions are key to processing matrices on the Tensor Cores.

- *load_matrix_sync* : loads from either global or shared memory to Tensor Core fragments
- *store_matrix_sync* : stores from tensor core fragments to either global or shared memory
- *mma_sync* : performs the matrix multiply and accumulate operation

All three functions call the __syncwarp() function which waits until all warp lanes (threads of a warp) have synchronized and then the whole warp performs the operation. The Tensor Cores support varying matrix tiles including 16x16x16, which serves as a baseline for our DFT computation.

4) Warp Shuffle (SHFL) Instruction: In the Kepler series of GPUs, Warp Shuffle instructions were introduced which enabled a thread to directly read a register from another thread in the same warp. This newly introduced feature, allowed communication between threads without the need to utilize the relatively slower shared memory. A number of Warp Shuffle instructions were introduced which included *shfl, shfl_down, shfl_up and shfl_xor*. In this paper we would be extensively using the *shfl_xor* instruction, which takes three arguments i.e., participating threads, data to exchange and the lane mask. The XOR operation is applied between caller lane_id and the lane mask to determine the lane from which to copy the value.

B. Fourier Transform

Now that the reader is familiar with some of the internals of CUDA relevant to our work, we move to Fourier Transforms, which will complete the background knowledge needed to appreciate the contributions made by this work. The Fourier transform [10] of the function $f(x)$ is the function

$$F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-i\omega x}\, dx \tag{1}$$

where $i = \sqrt{-1}$ and $e^{i\theta} = \cos\theta + i\sin\theta$ which is known as the Euler's Identity.

The integration of $f(x)$ in the Fourier Transform is often represented as $\mathcal{F}(f(x))$ where $\mathcal{F}$ is referred to as the Fourier transform operator. In Signal Processing, $f(x)$ is the input signal and $F(\omega)$ is known as its corresponding *frequency spectrum*. When the signal is discrete and periodic we use the Discrete Fourier Transform (DFT) instead, which is defined by the formula

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} \tag{2}$$

where

$$W_N = e^{-i\frac{2\pi}{N}} \tag{3}$$

The input signal is x_n for $n = 0 \ldots (N-1)$ and W_N^k for $k = 0 \ldots (N-1)$ is known as the *Nth roots of unity* or *twiddle factor*. It takes its name from the fact that $(W_N^k)^N = 1$ for all k if we take complex arithmetic into account. There are two properties associated with the *twiddle factor*, i.e, periodicity and symmetry. The periodicity property states that $W_N^{\alpha+N} = W_N^{\alpha}$ while the symmetry property means that values that are 180 degrees out of phase are negatives of each other. The output X_k is the Discrete Fourier Transform of the input signal x_n. This is a list of N complex numbers. This operation can also be represented in terms of matrices and the DFT algorithm is just a dense matrix vector multiplication as shown below.

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ \vdots \\ X_{N-1} \end{pmatrix} = \begin{pmatrix} W^0 & W^0 & W^0 & \cdots & W^0 \\ W^0 & W^1 & W^2 & \cdots & W^{N-1} \\ W^0 & W^2 & W^4 & \cdots & W^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ W^0 & W^{N-1} & W^{2(N-2)} & \cdots & W^{(N-1)(N-1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{pmatrix}$$

The time complexity of the matrix-vector multiplication formulation of DFT is $O(N^2)$. FFT is just a lower-complexity formulation of the DFT algorithm. FFT is a divide-and-conquer algorithm, which is able to perform this matrix vector multiplication in $O(N \log_2 N)$ steps. Furthermore, it is also possible to compute the inverse Fourier Transform with the same computational complexity. There are multiple representations of the FFT operation. One is via the butterfly algorithm, which we will go over in later sections and another via factorizing the dense Fourier Matrix into a bunch of sparse matrices. In our implementations, we use the *Cooley-Tuckey Recursive Algorithm*, which we will discuss in the section below.

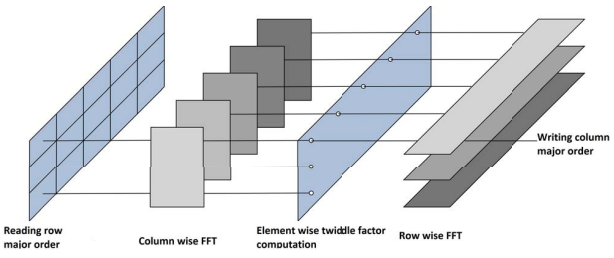

Fig. 2. Cooley–Tukey Recursive Algorithm Steps.

C. Cooley–Tukey Recursive Algorithm

The Cooley–Tukey Recursive Algorithm [11] can be mapped in terms of matrix multiplication and has the ability to utilize the dense matrix multiplication acceleration provided by the Tensor Cores. We will first state the mathematical formulation of the algorithm. The algorithm begins by treating the input length N as a composite $N = N_1 \cdot N_2$. Now, if we apply this to the original DFT equation via index mapping [12].

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} \tag{4}$$

changes to:

$$X_{k_1+N_1k_2} = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x_{n_1N_2+n_2} W_N^{(n_1N_2+n_2)(k_1+N_1k_2)}$$

where,

$$n = n_1N_2 + n_2$$

$$k = k_1 + N_1k_2$$

for $n_1, k_1 = 0, 1, \ldots (N_1-1)$ and $n_2, k_2 = 0, 1, \ldots (N_2-1)$

Now,if we simplify the exponentials we get:

$$\begin{aligned} &W_N^{(n_1N_2+n_2)(k_1+N_1k_2)} \\ &= (W_N^{n_1N_2k_1})(W_N^{n_1N_2N_1k_2})(W_N^{n_2k_1})(W_N^{n_2N_1k_2}) \\ &= W_{N_1}^{n_1k_1} \cdot 1 \cdot W_N^{n_2k_1} W_{N_2}^{n_2k_2} \end{aligned}$$

Since $W_N^{N_1} = W_{N_2}$, $W_N^{N_2} = W_{N_1}$ and $W_N^N = 1$, plugging it back and rearranging would give:

$$X_{k_1+N_1k_2} = \sum_{n_2=0}^{N_2-1} (W_N^{n_2k_1} (\sum_{n_1=0}^{N_1-1} x_{n_1N_2+n_2} W_{N_1}^{n_1k_1})) W_{N_2}^{n_2k_2}$$

The obtained resultant Equation provides a very interesting way of computing a one dimensional DFT by expressing it as a two dimensional matrix. First, the input vector is read row-wise into a $N_1 \times N_2$ matrix. This can simply be a row-major layout interpretation of the original vector into a 2D matrix. Then, N_1 length DFTs are computed over all N_2 columns. In the subsequent step, the result is multiplied element-wise with the twiddle factors. After the element-wise multiplication operation, N_2 length FFTs are computed over all N_1 rows. Finally, the resultant matrix is written out column-wise, i.e., transposed, to get the final vector. Figure 2 shows a graphical depiction of the steps involved. This is a recursive process as when computing the column-wise FFTs we can continue with the first step and decompose it further. A base-case size would be required to end the recursion and to apply the DFT operation directly.

D. Number Theoretic Transform

The Number Theoretic Transform (NTT) [13] is a form of DFT specialized for the finite field of integers. The main difference between NTT and DFT is their twiddle factors: DFT uses the powers of Nth root of unity in Equation 3, whereas NTT uses $\Psi \in \mathbb{Z}_p^n$, the powers of the Nth root of unity modulo a prime number such that $\Psi^N \equiv 1 \pmod p$ for a given input size N and a prime number p where $p = kN+1$. The forward NTT transform is defined as:

$$A_k = \sum_{n=0}^{N-1} a_n \Psi_N^{nk} \pmod p \tag{5}$$

Algorithm 1: Cooley-Tukey NTT

Input: A vector $a = (a[0], a[1], ..., a[N-1]) \in \mathbb{Z}_p^n$ where p is a prime such that $p \equiv 1 \mod 2N$. A precomputed twiddle factor table $\Psi = (\Psi[0], \Psi[1], ..., \Psi[N-1])$.

Output: $a \leftarrow NTT(a)$ is in bit-reversed order.

```
t = N/2
for (m = 1; m < N; m = 2m) do
    for (i = 0; i < m; i = i + 1) do
        for (j = 2i · t; j < 2i · t + t; j = j + 1) do
            X = a[j]
            Y = a[j + t] · Ψ[m + i]
            a[j] = X + Y  mod p
            a[j + t] = X − Y  mod p
    t = t/2;
```

The Cooley-Tukey algorithm [13] can be used to implement a similarly fast version of NTT, as shown in other works [14], [15].

Algorithm 1 shows a radix-2 based implementation of the Cooley-Tukey based NTT, which is adapted from [16]. An input vector of size N is recursively divided into 2 interleaved $N/2$-point NTTs at each stage, with a total of $\log_2 N$ iterations or stages. The twiddle factors Ψ, which are used at each stage are stored in a pre-computed table for efficiency. The output vector produced is in bit-reversed order so the input needs bit reversing to reorder the output. A variant of this is based on the Stockham Algorithm [17] and does not require bit reversal at the output but for simplicity, we will only be considering the Cooley-Tukey based NTT.

Given the value of quickly computing NTTs, a number of prior works have focused on accelerating NTT on GPUs [14], [18]. However, these studies do not take advantage of Tensor Cores or warp-shuffle instructions, which is one of the main contributions of our work.

III. Design and Methodology

A. Tensor Core based DFT Matrix

Our approach is based on the Cooley-Tukey recursive algorithm mentioned earlier. One of the underlying ideas of our work is an efficient representation of a complex data type. As

the Tensor Core inputs cannot be the CUDA standard complex type or a vectorized type like *half2*, we define two half matrices, one for the real part and the other for the imaginary. For example, a 4-point DFT Fourier Matrix is represented as follows:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & -1 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

The real matrix contains the real part of the coefficients while the imaginary matrix contains the imaginary part. The first step of our algorithm is to have a base case because no matter how many times we decompose the vector recursively, it will come to a point where multiplication by the DFT Fourier matrix is necessary.

As one of the available Tensor Core tile sizes is 16×16, it would be ideal to have a 16-point DFT as a base case. Using the Tensor Cores we strive to process batches of 16 DFTs, each of which is a base-case 16-point DFT. Algorithm 2 shows the complex matrix multiplication steps required to do a 16-point DFT using the Tensor Cores.

Algorithm 2: Tensor Core based 16 batch 16-point DFT (unitDFT)

1: $F_{real} \leftarrow$ *Real Part of the Fourier Matrix*
2: $F_{img} \leftarrow$ *Imaginary Part of the Fourier Matrix*
3: $X_{real} \leftarrow$ *Real Part of the Input Vector*
4: $X_{img} \leftarrow$ *Imaginary Part of the Input Vector*
5: $N_{img} \leftarrow -F_{img}$
6: $C \leftarrow \underline{\mathbf{0}}$
7: $T_1 \leftarrow F_{real} \cdot X_{real} + C$
8: $T_2 \leftarrow F_{img} \cdot X_{real} + C$
9: $Y_{real} \leftarrow N_{img} \cdot X_{img} + T_1$
10: $Y_{img} \leftarrow F_{real} \cdot X_{img} + T_2$

B. Warp-shuffle-XOR (WSX) NTT and FFT butterfly computation

One of the key contributions of this paper is to efficiently implement NTT and FFT using the warp shuffle instruction. This is based on the *O(N* $\log_2 N$*)* based butterfly representation. The Warp-shuffle-XOR (WSX) based NTT/FFT computation has two parts, i.e., the bit reversal step and the butterfly exchange step. In the first step, each thread loads an input value based on a bit reversal table. The offset of the bit reversal table entry accessed by each thread is (thread_id % NUM) where NUM is the size of the NTT/FFT. The following is the bit reversal table in a 4-point NTT/FFT:

Index	offset
0	0
1	1
2	-1
3	0

TABLE I
BIT REVERSAL TABLE FOR 4-POINT FFT

Note that the first and the last index yields out 0, which means there will not be a swap of values as bits "00" and "11" remain the same when reversed. The second index yields out 1 while the third one yields out -1, which indicates that a swap between the values of these two threads as the bits "10" become "10" and vice versa. One of the key optimizations applied is pre-generating this bit reversal table and storing it in constant memory instead of creating it in each thread. As constant memory is read-only and gets heavily cached, this provides a good reduction of the overall execution time.

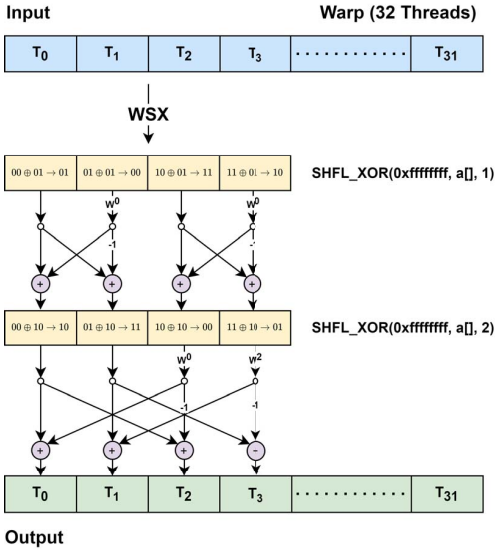

Fig. 3. WSX based 4 point FFT/NTT algorithm.

The second step involves the butterfly exchange. Listing 1 shows the FFT, and Listing 2 shows the NTT CUDA implementation while Figure 3 shows the 4-point computation flow. Note that some of the arrows in Figure 3 show the twiddle factors that need to be multiplied to the data in that lane.

The key part of this implementation is the use of shuffle function. The warp shuffle function enables a thread to directly read a register from another thread in the same warp. The threads in the warp can either collectively exchange or broadcast data. This is a lot faster than using shared memory, which requires a load, a store and an extra register to hold the address. The *__shlf_xor_sync* takes three arguments, i.e., mask, value and and an ID. In our case, the mask is `0xffffffff`, which indicates that all 32 threads in the warp would take part in the exchange. The value is the data that is being exchanged while the (ID XOR lane_id) determines which warp lane to send the data to. Similar to the bit reversal table, we created a twiddle factor table (TF_table) in the constant memory as these twiddle factors remain same throughout all warps in the grid. The twiddle factor values for NTT and FFT are computed differently. Moreover, control divergence is avoided through

Listing 1: Shuffle XOR based FFT butterfly exchange step

```
__inline__ __device__ half2 btfy(half2 a,int num){
    unsigned lane_id,j,x,y,ind;
    asm volatile ("mov.u32 %0, %laneid;" : "=r"(lane_id));
    a = fft_2_pt(a,2);
    if(num==2) return a;
    for(int i=4;i<=num;i*=2){
        j = (i/2); x = (2*!((lane_id%i)/j))-1;
        y = ((lane_id%i)/j); ind = (lane_id%j)+(j-1);
        half2 w1 = __shfl_xor_sync(0xffffffff,a,j);
        half2 b1 = __hadd2(w1,cmul(TF_table[ind],smul(a,x)));
        half2 b2 = __hadd2(a,cmul(TF_table[ind],w1));
        a = __hadd2(smul(b1,y),smul(b2,!y));
    }
    return a;}
```

Listing 2: Shuffle XOR based NTT butterfly exchange step

```
__inline__ __device__ int ntt_btfy(int a,int num){
    unsigned lane_id,j,x,y,ind;
    asm volatile ("mov.u32 %0, %laneid;" : "=r"(lane_id));
    a = nttt_2_pt(a,2);
    if(num==2) return a;
    for(int i=4;i<=num;i*=2){
        j = (i/2); x = (2*!((lane_id%i)/j))-1;
        y = ((lane_id%i)/j); ind = (lane_id%j)+(j-1);
        int w1 = __shfl_xor_sync(0xffffffff,a,j);
        int b1 = w1 + (TF_table[ind] * a * x);
        int b2 = a + (TF_table[ind] * w1);
        a = ((b1 * y) + (b2 * (!y))) % PRIME;
    }
    return a;}
```

the use of logical operators and divisions to eliminate any branches that restricted control flow based on the lane_id. In case of FFT, the function *cmul* is used for computing half precision complex element multiplication while *smul* does a scalar multiplication.

C. Fully Optimized Algorithm

In this section, we present our overall algorithmic approach as depicted by the concept diagram in Figure 4 and Algorithm 3. Our optimized algorithm is divided into the following 7 stages.

1) To start off, data movement and pre-computations take place. Input Data is moved from the host memory to device memory while the constant memory is populated with the read-only bit reversal and the twiddle factor tables. The base Fourier matrices are also generated, which can fit into the Tensor Core tile size.
2) The main CUDA kernel is launched, which computes the column-wise DFT operation using the Tensor Cores by effectively utilizing shared memory for faster accesses. For smaller sizes the *WSX* algorithm is directly used.
3) Now, a per thread element-wise complex number multiplication is done with the resultant output and the twiddle factors. This can also be thought of as a Hadamard product.
4) We now move towards a row-based FFT computation. A decision is made whether to use the Tensor Cores or the *WSX* algorithm. An appropriate element-wise twiddle factor computation is done where required.

Algorithm 3: Optimized FFT Algorithm

Input: A complex vector
$x = (x[0], x[1], ..., x[N-1]) \in \mathbb{C}$
Pre-computed bit reversal and twiddle factor table
$X_m \leftarrow$ batches of input x grouped in $m \times m$ sub-matrix
T $\leftarrow$ Twiddle Factors, $B \leftarrow$ Number of Batches
$N \leftarrow$ Single Batch Length
$\Psi_m \leftarrow$ m point DFT matrix where m $\leq$ Tensor Core tile size
Output: $y \leftarrow FFT(x)$

1 **for** $(i = 0; i < \lceil \frac{B}{m} \rceil; i++)$ **do**
2 $\quad Y = T[i] \odot (\Psi_m \cdot X_m[i])$
3 $\quad N = \frac{N}{m}$
4 $\quad < synchronize >$
5 $\quad$ **if** $N < m$ **then**
6 $\quad\quad WSX(Y[i], N)$
7 $\quad$ **else if** $N == m$ **then**
8 $\quad\quad$ goto line 2
9 $\quad$ **else**
10 $\quad\quad WSX(Y[i], \frac{N}{m})$
11 $\quad\quad$ goto line 2
12 $\quad blockTranspose(Y[i])$

5) The decision is executed.
6) We now do a matrix transpose to dump the final output back to global memory. This can be achieved via the Tensor Cores or a per thread approach.
7) Last, we move towards the next group of batches and continue doing so until all are exhausted.

IV. Experimental Results

In our experiments, we have used NVIDIA's Ampere A100 GPU for comparisons. Due to a lack of any official GPU-based standard implementation for NTT, we implemented a baseline to compare against our WSX-based NTT. It would follow a standard block synchronous approach where threads of a block would collaborate and work on a chunk of input vector concurrently. In the interest of fairness, the number of threads used in our experiments were kept the same for both the implementations (baseline and WSX-based NTT) and the type used was integer.

Figure 5 shows the first set of results. The goal here is to test the baseline NTT implementation against our WSX-based approach. As evident, our WSX-based NTT has delivered a substantial speedup of up to $4\times$, especially in case of larger input sizes where the gap between the performance of the two systems is clearly observable.

The y axis shows the execution time in milliseconds. CUDA events were used to measure the kernel execution time and the copying time between host and device was excluded. We also used the latest NVIDIA profiler i.e., *nsight-compute* to instrument the achieved compute and memory throughput values as a percentage of theoretical maximum. For the same experiment, Figure 6 shows the corresponding GPU utilization

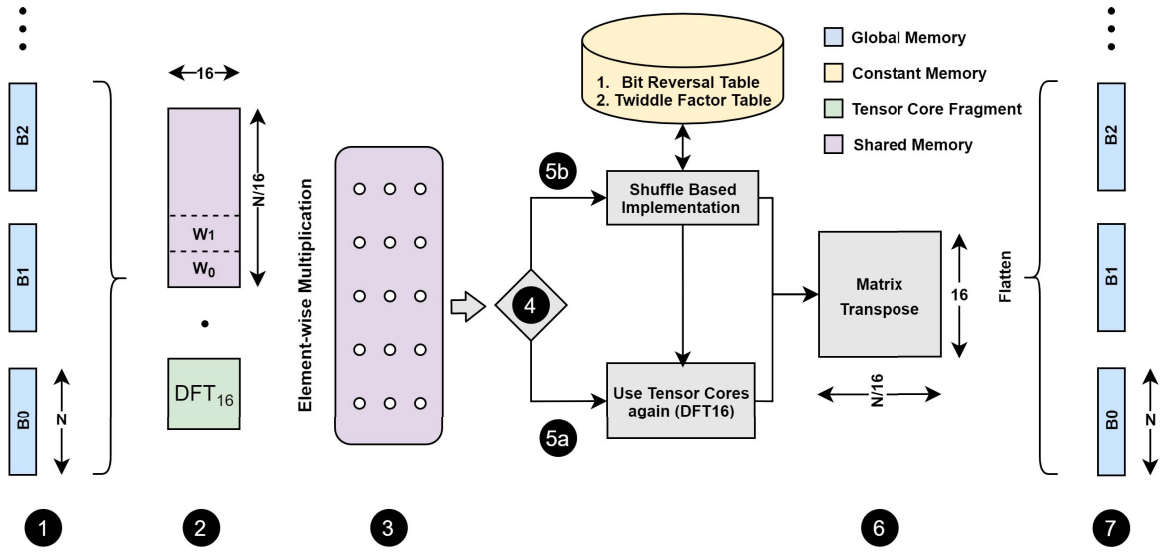

Fig. 4. Concept Diagram for our optimized algorithmic approach

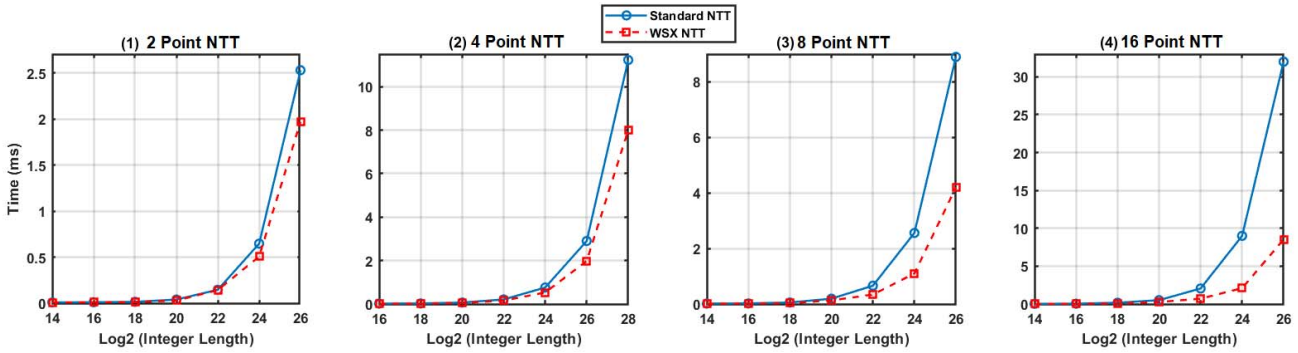

Fig. 5. NTT Execution Time Graphs.

study for NTT. Our WSX approach is clearly utilizing better compute and memory resources and we would achieve up to $6\times$ higher compute and memory throughput in case of the larger 16 radix NTT. It is interesting to note that with increase in input vector length and radix value, the compute throughput trace overtakes the memory throughput trace. We can also safely deduce that our implementation is both memory and compute-bound and would definitely scale well with the increase in the number of Streaming Multiprocessors.

We now move to the second batch of results that focus on FFT. For our FFT benchmarking, we used Fp16 supported cufftXt library for comparisons. The cufftXt library has documented limitations i.e does not support more than 4 billion elements and is restricted to powers of two. Figure 7 shows our fully optimized FFT execution time results against cuFFT. On the x-axis, we have the number of input batches while on the y-axis, we have the execution time in milliseconds. Logarithmic scaling is used on the y-axis to correctly fit all the values in the graph and to improve readability. Note that due to the number of element restriction of cufftXt the number of batches decrease with increase in length of a single batch. In all of our cuFFT experiments, we did not take the time for the first run into account, which included cuFFT's plan creation and specialization overhead. Although it was one time operation, it is still quite expensive taking around 100ms to complete. In the end, we used the *cufftDestroy* function to remove the plan buffers from GPU memory. Similarly, we did not include the time for pre-computing some of our structures i.e., bit Reversal table and twiddle Factor table. The time for this was negligible at less than 1ms.

We clearly boast decent speedup compared to cuFFT. In smaller batch sizes we get about $1.5\times$ speedup while going neck to neck and converging in extreme cases where the library support is exhausted. This is a notable win as cuFFT, which follows FFTW, specializes its algorithm based on the underlying hardware. Furthermore, our implementation did not compromise on precision and it co-related well with cuFFT's results. We again used NVIDIA's *nsight-compute* profiler to

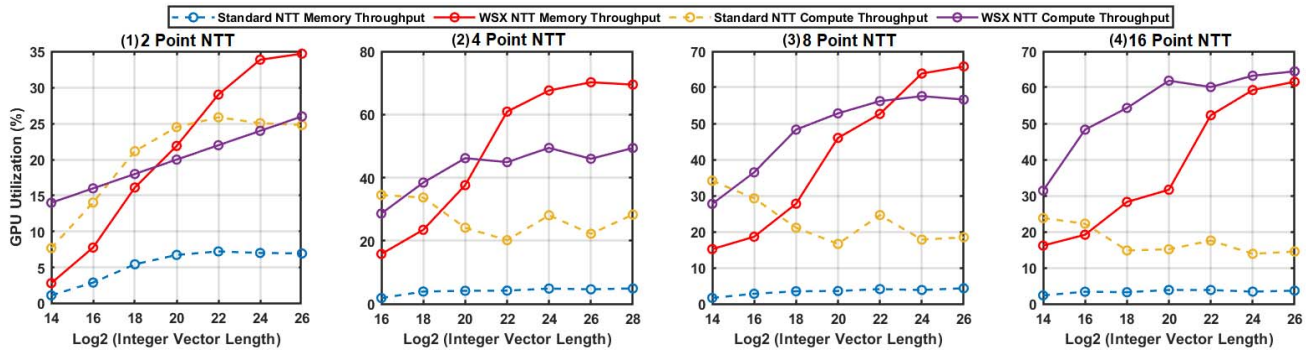

Fig. 6. NTT GPU Utilization Graphs.

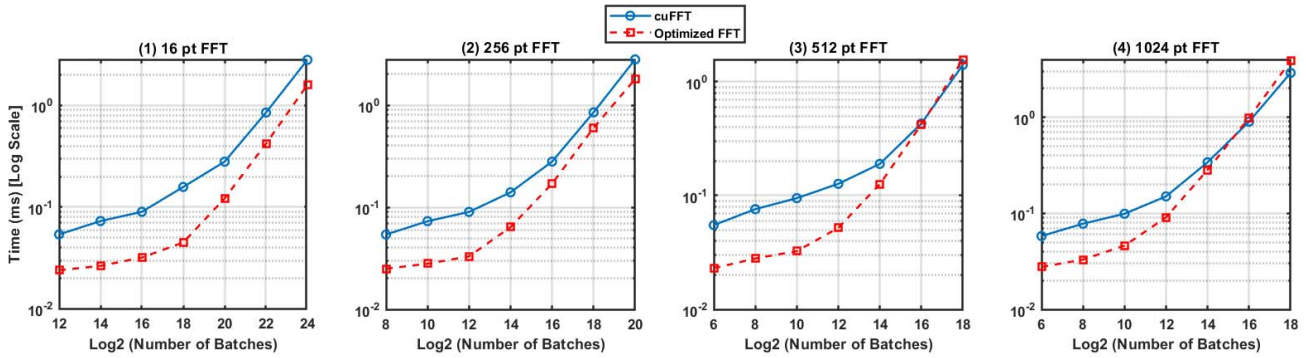

Fig. 7. FFT Execution Time Graphs.

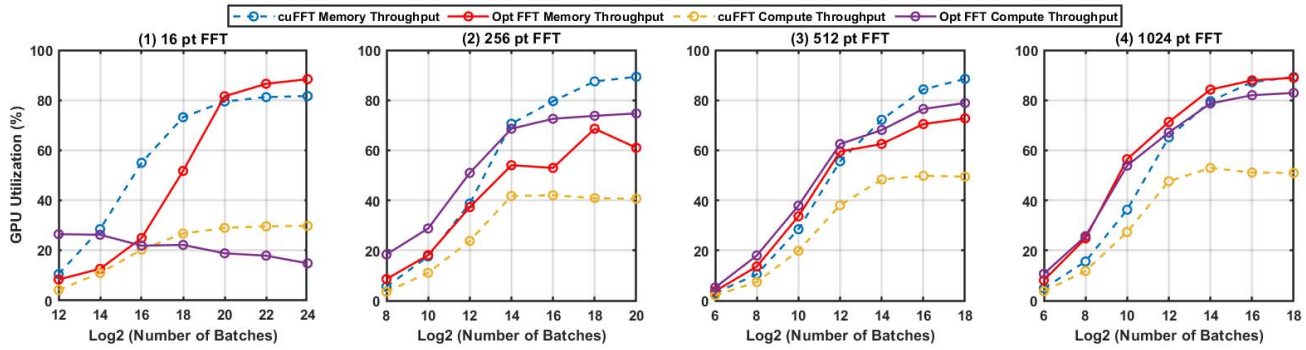

Fig. 8. FFT GPU Utilization Graphs.

get the achieved compute and memory throughput values. Like before, Figure 8 shows the detailed GPU utilization Graphs for FFT results. Here also, it is evident that our optimized algorithm has about 2× higher compute throughput which is due to the utilization of Tensor Cores. In case of memory throughput utilization, we are almost neck to neck with cuFFT implementation and even overtake cuFFT in several instances. One of the key insights that can be drawn from these results is that the cuFFT implementation is indeed memory-bound while our optimized approach is both memory and compute-bound. This means that if the number of Tensor Cores continue to increase with newer GPU releases (which is the current trend), our implementation would surely improve in performance further.

V. Using CUDA Streams

A stream is a sequence of operations that execute in issue-order on the device (GPU). CUDA allows us to initiate multiple streams and run operations concurrently. While operations within a stream are guaranteed to execute in the prescribed order, operations across different streams can be interleaved and or executed concurrently. To take advantage of multiple streams both the GPU kernel execution and the

memory copy must be done asynchronously. GPU kernels are asynchronous by default so they can easily fit in. For memory copy, *cudaMemcpyAsync* is used which has the ability to return the call to the host even before the copying is complete. For this to work the host memory needs to be allocated using *cudaMallocHost*, which is page locked and accessible to the device.

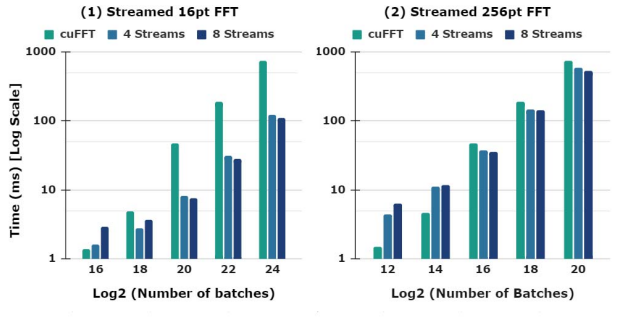

Fig. 9. FFT Complete Execution Time with multiple streams.

Listing 3 shows an example of using CUDA streams. Note that the input array is divided into chunks and launched on different streams. The streams are also bidirectional i.e., while one is copying from host to device another would be copying from device to host. CUDA streams allow for an overlap between communication and computation, which greatly boosts overall execution time. The graphs in Figure 9 show the impact of CUDA streams on 16-point and 256-point Fourier transforms. Note that on the y-axis, we have the complete time which includes the copying as well as the execution time. There is a clear speedup when using streams for higher batch sizes although using twice as many streams does not mean twice as big of a speedup, as seen when using 8 streams instead of 4. In case of low batch sizes, we see that in fact cuFFT is faster. This is due to the overhead associated with creating both, multiple CUDA Streams and launching multiple kernels. This implies that the number of streams launched should depend on the number of input elements and hence, a varied approach should be adopted based on the nature of the batch sizes.

VI. Related Work

Many of the earlier works on GPUs used graphics APIs to compute the FFT. This is generally done to improve the performance of matrix multiplication done during FFT computation [19]. [12] demonstrates how to do this for image processing applications on early GPU hardware and [20] compares the use of FFT and convolution techniques on GPU to perform image filtering.

[21] used OpenGL-based Vertex and Fragment shaders while [22] developed a DirectX based implementation. Similarly, [23] uses the modern CUDA API to implement mixed-radix FFT algorithms while exploiting shared memory. GPU clusters have also been used to optimize large-scale FFT calculations [24].The 3D-FFT, a variation for 3D images, is a very data and compute intensive kernel that also has CUDA-based implementations in [25] and [26]. All these works are commendable and have contributed towards advancing the field. They serve as an inspiration to our own work.

More specifically, attempts have also been made in the past to accelerate NTT on GPUs [27], like our work, however, we introduce a novel way to utilize Tensor Cores and warp-shuffle instruction to optimize performance. Our comparisons with the state of the art cuFFT library depicts the viability and impact of our approach.

In general, Tensor Cores have been used in the past in a variety of applications beyond Deep Learning workloads. They have been used for generating weather forecast models by accelerating the calculation of Legendre transforms [28]. Similarly, authors in [29] aim to accelerate arithmetic reduction using a chained matrix multiply add (MMA) approach by using Tensor Cores. Reduction and scan have been formulated in terms of matrix multiplication and mapped on the tensor cores utilizing high performance in [30]. Iterative Refinement solvers have also been mapped on the Tensor cores with improved accuracy [31]. Finally, Tensor cores have also been utilized to generate Image Processing code by extending the Halide domain specific language (DSL) [32].

Tensor-Core based complex matrix multiplication has been studied in [33] and demonstrates a notable speed up for a variety of cases. The warp-shuffle instruction has also been used earlier for implementing FFT in [34]. Independently [33], [34] provide an important base for this paper. There has been an attempt to optimize the FFT using Mixed Precision on Tensor Cores [35], sacrificing precision for speed. Even though the work is promising, the reported performance of the system has room for improvement due to the inherent trade-off of their approach. [36] is a notable work that expands our idea of using tensor cores for FFT and presents a comprehensive mixed-precision FFT algorithm that offers performance improvements over cuFFT.

Lastly, this work builds on our parallel [37], [38] which discuss a similar methodology of using tensor cores and warp shuffle to outperform cuFFT on Volta GPUs.

VII. Limitations and Future Work

Currently, NVIDIA Tensor Cores do not support full 32-bit integer as input. In the future, with the support for integers and long types, we could further optimize our approach of using Tensor Cores to compute NTT. Moreover, it could also be interesting to see how our approach scales with multiple GPUs.

In terms of the limitations, our work does not take into account input with prime number lengths. Although, zero padding approximation can be applied to compliment our proposed approach, it is currently unclear how specialized prime FFT computation algorithms, like Rader's [39] and Bluestein's [40] algorithms can be mapped onto the Tensor Cores. This is an area we plan to explore in future work.

Listing 3: Launching Kernel with CUDA Streams

```
cudaStream_t cs[NS];
size_t s_size = sizeof(half)*NUM*(BATCH_SIZE/NS);
for(int s=0;s<NS;s++){
                cudaStreamCreate(&cs[s]);
                int off = (s*NUM*(BATCH_SIZE/NS));
                cudaMemcpyAsync(&d_xr[off],&h_xr[off],s_size,cudaMemcpyHostToDevice,cs[s]);
                kernel<T1,T2,NUM,WARP_BLOCK,BATCH_SIZE/NS>
                <<<gridDim,blockDim,0,cs[s]>>>(d_xr+off,d_xi+off,d_yr+off,d_yi+off);
                cudaMemcpyAsync(&h_yr[off],&d_yr[off],s_size,cudaMemcpyDeviceToHost,cs[s]);
        }
for(int i=0;i<NS;i++)
        cudaStreamSynchronize(cs[i]);
for(int i=0;i<NS;i++)
        cudaStreamDestroy(cs[i]);
```

VIII. Conclusion

Graphics Processing Units (GPUs) are evolving at a rapid pace and vendors are coming under increasing pressure to incorporate additional domain-specific architectural features. For example, we might soon see an accelerator for computing the Hadamard product or sparse matrices, which would boost performance for certain types of computations. However, these specialized features and accelerators, though intended for targeted workloads and narrowly-defined operations, can be carefully studied and applied to other unintended domains as well. For instance, Tensor Cores, which were specifically developed to accelerate Machine Learning and Deep Learning workloads can be leveraged in novel ways to get additional performance benefits in other engineering and scientific domains, as shown in this work. In particular, by utilizing recent advancements in GPU architectures and instruction sets, such as Tensor Core and WSX-based operations, we have improved the execution times of FFT and NTT algorithm, outperforming the existing standards. Our results underscore the need to explore additional avenues of utilizing various macro and micro-architectural features and properties, thereby expanding the benefits for the broader community.

References

[1] J. Goldthwaite. The value of digital signal processing. [Online]. Available: https://www.sensear.com/blog/the-value-of-digital-signal-processing

[2] Y. Doröz, E. Öztürk, and B. Sunar, "A million-bit multiplier architecture for fully homomorphic encryption," *Microprocessors and Microsystems*, vol. 38, no. 8, pp. 766–775, 2014.

[3] D. Coldewey. Duality scores $14m darpa contract for hardware-accelerated homomorphic encryption. [Online]. Available: https://techcrunch.com/2021/02/03/duality-scores-14m-darpa-contract-for-hardware-accelerated-homomorphic-encryption/amp/

[4] A. A. Al Sallab, H. Fahmy, and M. Rashwan, "Optimized hardware implementation of fft processor," in *2009 4th International Design and Test Workshop (IDT)*. IEEE, 2009, pp. 1–5.

[5] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the fft," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, vol. 3. IEEE, 1998, pp. 1381–1384.

[6] C. D. Zone. CUDA toolkit documentation -. [Online]. Available: https://docs.nvidia.com/cuda/cufft/index.html

[7] T. NVIDIA, "V100 GPU architecture," 2017.

[8] NVIDIA, "A100 GPU architecture," 2020. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf...

[9] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: A hands-on approach*. Morgan kaufmann, 2016.

[10] M. Rahman, *Applications of Fourier transforms to generalized functions*. WIT Press, 2011.

[11] P. Duhamel and M. Vetterli, "Fast Fourier transforms: A tutorial review and a state of the art," *Signal Processing (Elsevier)*, vol. 19, no. ARTICLE, pp. 259–299, 1990.

[12] C. Burrus, "Index mappings for multidimensional formulation of the dft and convolution," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, no. 3, pp. 239–242, 1977.

[13] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[14] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on GPUs: An implementation using CUDA," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018.

[15] H. Chen, K. Han, Z. Huang, A. Jalali, and K. Laine, "Simple encrypted arithmetic library v2. 3.0," *Microsoft Research, December*, 2017.

[16] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 264–275.

[17] W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, D. E. Nelson, C. M. Rader, and P. D. Welch, "What is the fast fourier transform?" *Proceedings of the IEEE*, vol. 55, no. 10, pp. 1664–1674, 1967.

[18] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[19] X. Cui, Y. Chen, and H. Mei, "Improving performance of matrix multiplication and fft on GPU," in *2009 15th International Conference on Parallel and Distributed Systems*. IEEE, 2009, pp. 42–48.

[20] O. Fialka and M. Cadik, "FFT and convolution performance in image filtering on GPU," in *Tenth International Conference on Information Visualisation (IV'06)*. IEEE, 2006, pp. 609–614.

[21] K. Moreland and E. Angel, "The FFT on a GPU," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2003, pp. 112–119.

[22] J. L. Mitchell, M. Y. Ansari, and E. Hart, "Advanced image processing with directx® 9 pixel shaders," *ShaderX2*, pp. 439–464, 2004.

[23] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete fourier transforms on graphics processors," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Ieee, 2008, pp. 1–12.

[24] Y. Chen, X. Cui, and H. Mei, "Large-scale FFT on GPU clusters," in *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010, pp. 315–324.

[25] S. Keskin, E. Erdil, and T. Kocak, "An efficient parallel implementation

of 3D-FFT on gpu," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2017, pp. 1–4.
[26] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-D FFT kernel for gpus using CUDA," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 2008, pp. 1–11.
[27] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 264–275.
[28] S. Hatfield, M. Chantry, P. Düben, and T. Palmer, "Accelerating high-resolution weather models with deep-learning hardware," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2019, pp. 1–11.
[29] C. A. Navarro, R. Carrasco, R. J. Barrientos, J. A. Riquelme, and R. Vega, "Gpu tensor cores for fast arithmetic reductions," *arXiv preprint arXiv:2001.05585*, 2020.
[30] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu, "Accelerating reduction and scan using tensor core units," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 46–57.
[31] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 603–613.
[32] S. Sioutas, S. Stuijk, T. Basten, L. Somers, and H. Corporaal, "Programming tensor cores from an image processing dsl," in *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, 2020, pp. 36–41.
[33] A. Abdelfattah, S. Tomov, and J. Dongarra, "Matrix multiplication on batches of small matrices in half and half-complex precisions," *Journal of Parallel and Distributed Computing*, vol. 145, pp. 188–201, 2020.
[34] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A gpu performance evaluation," *arXiv preprint arXiv:1412.7580*, 2014.
[35] A. Sorna, X. Cheng, E. D'azevedo, K. Won, and S. Tomov, "Optimizing the fast fourier transform using mixed precision on tensor core hardware," in *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*. IEEE, 2018, pp. 3–7.
[36] B. Li, S. Cheng, and J. Lin, "tcfft: Accelerating half-precision fft through tensor cores," 2021.
[37] S. Durrani, M. S. Chughtai, A. Dakkak, W.-m. Hwu, and L. Rauchwerger, "Fft blitz: the tensor cores strike back," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 488–489.
[38] S. H. K. Durrani, "Utilizing gpu tensor cores for algorithmic acceleration," 2020.
[39] C. M. Rader, "Discrete fourier transforms when the number of data samples is prime," *Proceedings of the IEEE*, vol. 56, no. 6, pp. 1107–1108, 1968.
[40] L. Bluestein, "A linear filtering approach to the computation of discrete fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 18, no. 4, pp. 451–455, 1970.