High-Performance Implementation of the Fast Fourier Transform Using the Cooley–Tukey Algorithm

Sharvari Satish Deshmukh
A59022896

ECE 277 - Fall 2024

Discrete Fourier Transform & Fast Fourier Transform

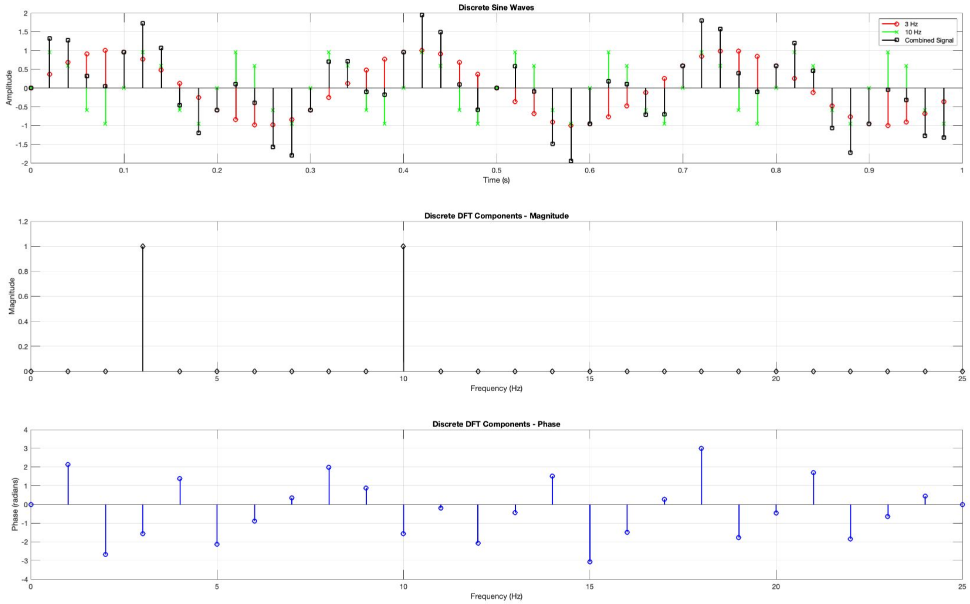
Let $x_0, \ldots, x_{n-1}$ be **complex numbers**. The **DFT** is defined as

$$X_k = \sum_{m=0}^{n-1} x_m e^{-i\frac{2\pi k m}{n}}, \quad k = 0, \ldots, n-1,$$

where $e^{i\frac{2\pi}{n}}$ is a **primitive** nth root of 1.

DFT ----> FFT

$O(n^2)$ -----> $O(n\log_2(n))$



Cooley-Tukey: Recursive vs Iterative Algorithm

```
def FFT_recursive(P):
    # P - [p0, p1, ..., pn-1] coeff representation
    n = len(P)  # n is a power of 2
    if n == 1:
        return P
    ω = e^(2πi/n)
    Pe, Po = [p0, p2, ..., pn-2], [p1, p3, ..., pn-1]
    ye, yo = FFT(Pe), FFT(Po)
    y = [0] * n
    for j in range(n/2):
        y[j] = ye[j] + ω^j * yo[j]
        y[j + n/2] = ye[j] - ω^j * yo[j]
    return y
```

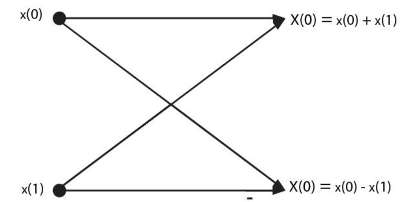
```
def FFT_iterative(P):
    # P - [p0, p1, ..., pn-1] coeff representation
    n = len(P)  # n is a power of 2
    log_n = log2(n)
    # Bit-reversal permutation
    P = bit_reverse_copy(P)
    for s in range(1, log_n + 1):
        m = 2^s
        ω_m = e^(2πi/m)
        for k in range(0, n, m):
            ω = 1
            for j in range(m//2):
                t = ω * P[k + j + m//2]
                u = P[k + j]
                P[k + j] = u + t
                P[k + j + m//2] = u - t
                ω = ω * ω_m
    return P
```

Cooley-Tukey: Iterative (Radix-2)

```
def FFT_iterative(P):
    # P - [p0, p1, ..., pn-1] coeff representation
    n = len(P)  # n is a power of 2
    log_n = log2(n)
    # Bit-reversal permutation
    P = bit_reverse_copy(P)
    for s in range(1, log_n + 1):
        m = 2^s
        ω_m = e^(2πi/m)
        for k in range(0, n, m):
            ω = 1
            for j in range(m//2):
                t = ω * P[k + j + m//2]
                u = P[k + j]
                P[k + j] = u + t
                P[k + j + m//2] = u - t
                ω = ω * ω_m
    return P
```

i=3 (011) --> (110) i=6 => swap(3,6)

twiddle factor



Radix-2 Butterfly

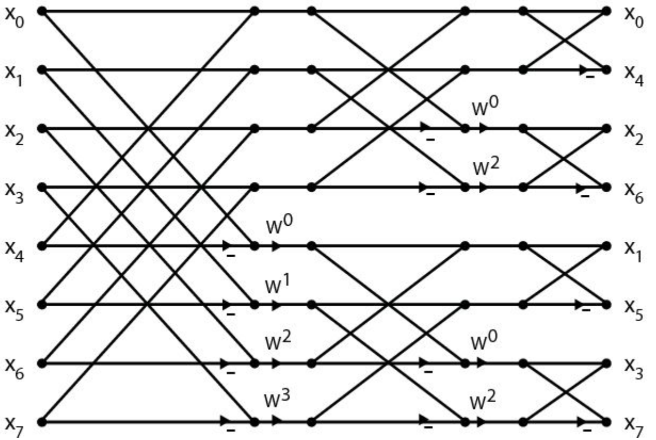
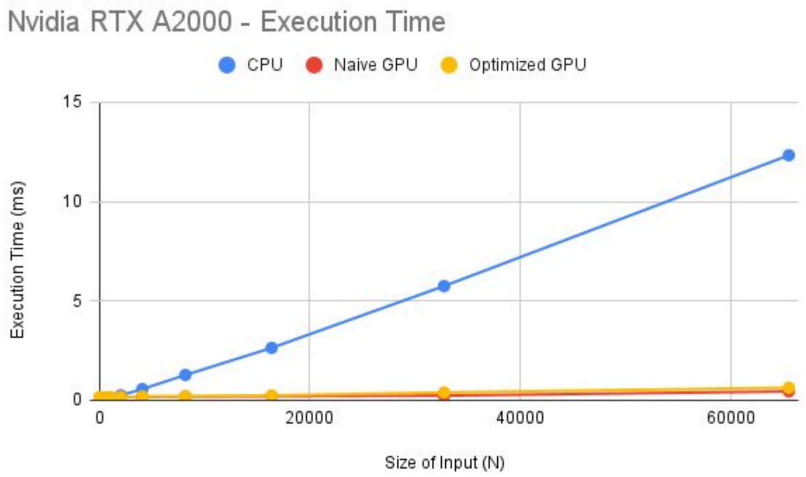


Figure 1: Length-8 Radix-2 FFT Flow Graph

Simple GPU implementation

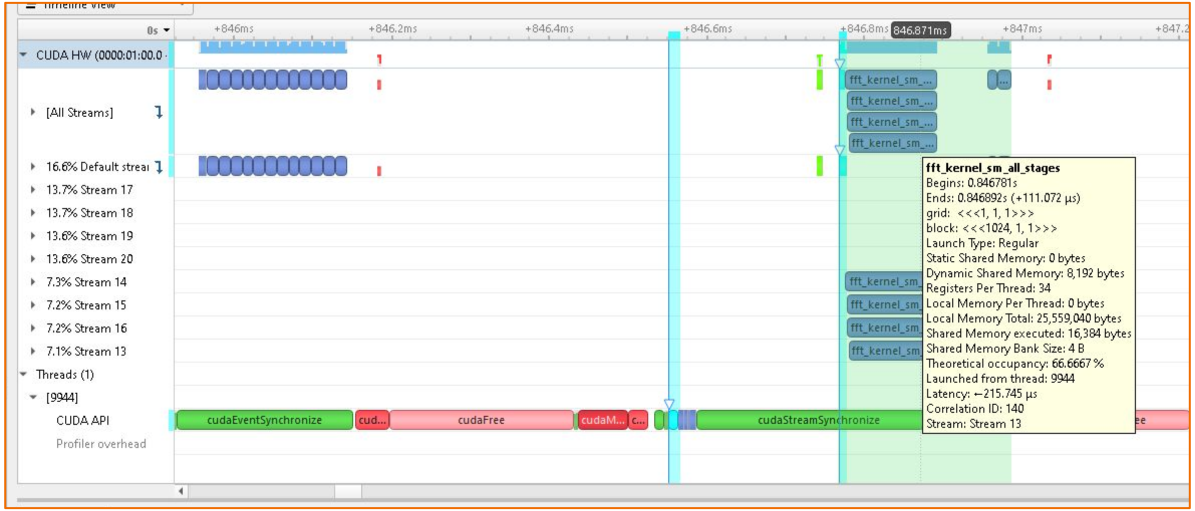

```
__global__ void fft_stage_kernel(float2* d_data, int N, int s)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int halfSize = 1 << (s - 1);  // m2
    int fftSize = 1 << s;         // m

    if (tid < N / 2) {
        int group = tid / halfSize;
        int j = tid % halfSize;
        int k = group * fftSize;

        float angle = -2.0f * (float)M_PI * j / (float)fftSize;
        float2 w = make_float2(cosf(angle), sinf(angle));

        int index1 = k + j;
        int index2 = k + j + halfSize;

        float2 u = d_data[index1];
        float2 t = d_data[index2];

        // Complex multiplication t * w
        float2 temp;
        temp.x = w.x * t.x - w.y * t.y;
        temp.y = w.x * t.y + w.y * t.x;

        // Butterfly
        d_data[index1].x = u.x + temp.x;
        d_data[index1].y = u.y + temp.y;
        d_data[index2].x = u.x - temp.x;
        d_data[index2].y = u.y - temp.y;
    }
}
```

Optimized GPU: Streams & Shared Memory




```
__global__ void fft_kernel_sm_all_stages(float2* d_data, int N, int logN)
{
    extern __shared__ float2 s_data[];
    int tid = threadIdx.x;
    s_data[tid] = d_data[tid];

    for (int s = 1; s <= logN; s++) {
        int m = 1 << s;
        int m2 = m >> 1;

        __syncthreads();

        if (tid < N / 2) {
            int group = tid / m2;
            int j = tid % m2;
            int k = group * m;

            float angle = -2.0f * (float)M_PI * (float)j / (float)m;
            float2 w = make_float2(cosf(angle), sinf(angle));

            int i1 = k + j;
            int i2 = k + j + m2;

            float2 u = s_data[i1];
            float2 t = s_data[i2];

            // Complex multiplication t * w
            float2 temp;
            temp.x = w.x * t.x - w.y * t.y;
            temp.y = w.x * t.y + w.y * t.x;

            // Butterfly
            s_data[i1].x = u.x + temp.x;
            s_data[i1].y = u.y + temp.y;
            s_data[i2].x = u.x - temp.x;
            s_data[i2].y = u.y - temp.y;
        }
    }

    __syncthreads();
    d_data[tid] = s_data[tid];
}
```

Optimized GPU: Streams & Shared Memory



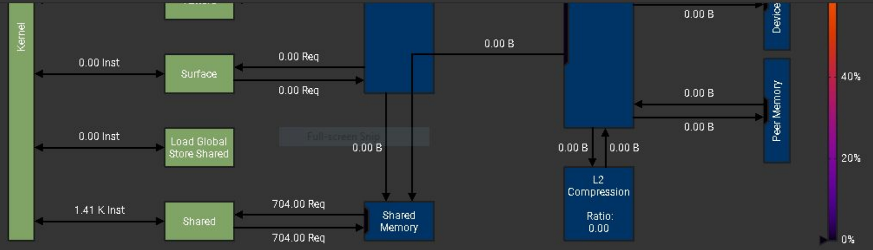
N = 4096

Optimized GPU: Bank Conflicts

debug.txt.ncu-rep | debug_opt_bc.txt.ncu-rep *

Page: Details | Result: 2 - 534 - fft_kernel_sm_all_stages | Add Baseline | Apply Rules | Occupancy Calculator | Copy as Image

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	534 - fft_kernel_sm_all_stages (1, 1, 1)x(1024, 1, 1)	241.09 usecond	133,486	34	0 - NVIDIA RTX A2000	553.68 cycle/usecond	8.6	[12476] fftimpl.exe



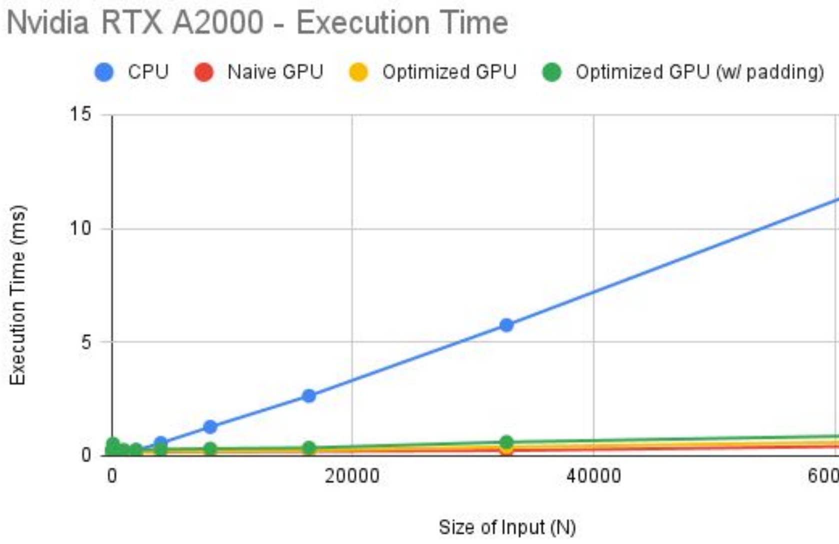
Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	704	704	1,920	0.06	1,216
Shared Load Matrix	0	0	0	0	0
Shared Store	704	704	1,920	0.01	1,216
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	143	0.05	0
Total	1,408	1,408	3,983	0.11	2,432

L1/TEX Cache

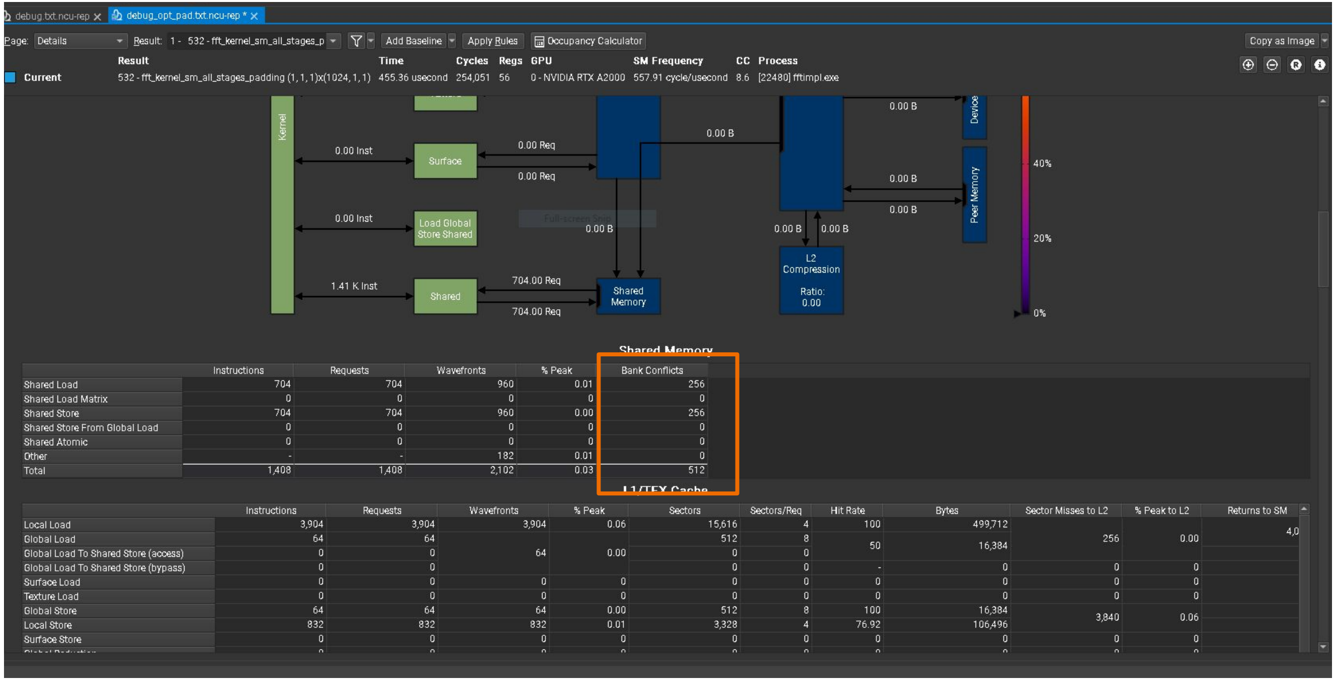
	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req	Hit Rate	Bytes	Sector Misses to L2	% Peak to L2	Returns to SM
Local Load	2,880	2,880	2,880	0.08	11,520	4	100	368,640			2,9
Global Load	64	64			512	8	50	16,384	256	0.01	
Global Load To Shared Store (access)	0	0	64	0.00	0	0					
Global Load To Shared Store (bypass)	0	0			0	0	-	0	0	0	
Surface Load	0	0	0	0	0	0	0	0	0	0	
Texture Load	0	0	0	0	0	0	0	0	0	0	
Global Store	64	64	64	0.00	512	8	100	16,384	6,912	0.20	
Local Store	1,600	1,600	1,600	0.05	6,400	4	90	204,800			
Surface Store	0	0	0	0	0	0	0	0	0	0	

Optimized GPU: Padding




```
__global__ void fft_kernel_sm_all_stages(float2* d_data, int N, int logN)
{
    extern __shared__ float s_mem[];
    const int warpSize = 32;
    const int paddingPerWarp = 4;
    const int numWarps = N / warpSize;
    const int paddedLength = N + numWarps * paddingPerWarp;
    float* s_real = s_mem;
    float* s_imag = s_mem + paddedLength;
    int tid = threadIdx.x;
    auto paddedIndex = [=](int i) {
        int warpId = i / warpSize;
        int offset = warpId * paddingPerWarp;
        return i + offset;
        };
    float2 val = d_data[tid];
    s_real[paddedIndex(tid)] = val.x;
    s_imag[paddedIndex(tid)] = val.y;
    for (int s = 1; s <= logN; s++) {
        int m = 1 << s;
        int m2 = m >> 1;
        __syncthreads();
        if (tid < N / 2) {
            int group = tid / m2;
            int j = tid % m2;
            int k = group * m;
            float angle = -2.0f * (float)M_PI * (float)j / (float)m;
            float2 w = make_float2(cosf(angle), sinf(angle));
            int i1 = k + j;
            int i2 = k + j + m2;
            float ur = s_real[paddedIndex(i1)];
            float ui = s_imag[paddedIndex(i1)];
            float tr = s_real[paddedIndex(i2)];
            float ti = s_imag[paddedIndex(i2)];
            float temp_r = w.x * tr - w.y * ti;
            float temp_i = w.x * ti + w.y * tr;
            s_real[paddedIndex(i1)] = ur + temp_r;
            s_imag[paddedIndex(i1)] = ui + temp_i;
            s_real[paddedIndex(i2)] = ur - temp_r;
            s_imag[paddedIndex(i2)] = ui - temp_i;
        }
    }
    __syncthreads();
    val.x = s_real[paddedIndex(tid)];
    val.y = s_imag[paddedIndex(tid)];
    d_data[tid] = val;
}
```

Optimized GPU: Padding



Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	704	704	960	0.01	256
Shared Load Matrix	0	0	0	0	0
Shared Store	704	704	960	0.00	256
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	182	0.01	0
Total	1,408	1,408	2,102	0.03	512

L1/TEX Cache

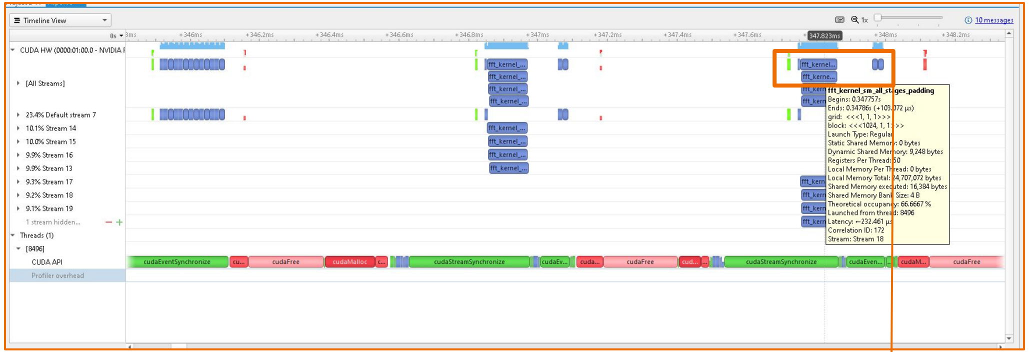
	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req	Hit Rate	Bytes	Sector Misses to L2	% Peak to L2	Returns to SM
Local Load	3,904	3,904	3,904	0.06	15,616	4	100	499,712			4,0
Global Load	64	64			512	8	50	16,384	256	0.00	
Global Load To Shared Store (access)	0	0	64	0.00	0	0					
Global Load To Shared Store (bypass)	0	0			0	0	-	0	0	0	
Surface Load	0	0	0	0	0	0	0	0	0	0	
Texture Load	0	0	0	0	0	0	0	0	0	0	
Global Store	64	64	64	0.00	512	8	100	16,384	3,840	0.06	
Local Store	832	832	832	0.01	3,328	4	76.92	106,496			
Surface Store	0	0	0	0	0	0	0	0	0	0	

Optimized GPU: Instructions Overhead

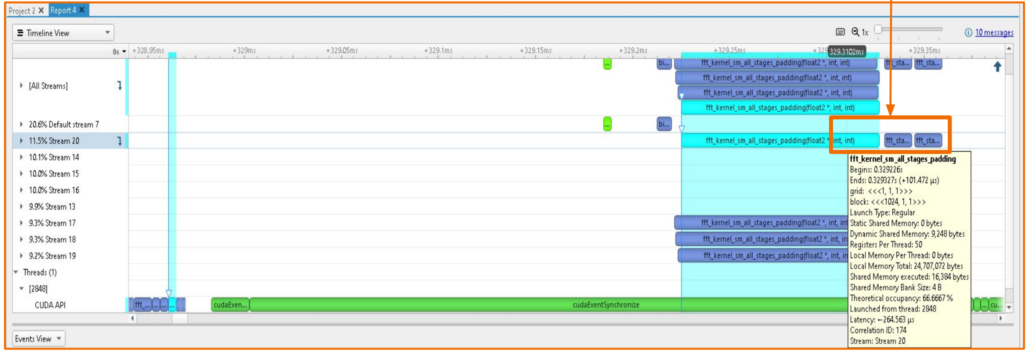
```
__global__ void fft_kernel_sm_all_stages(float2* d_data, int N, int logN)
{
    extern __shared__ float s_mem[];
    const int warpSize = 32;
    const int paddingPerWarp = 4;
    const int numWarps = N / warpSize;
    const int paddedLength = N + numWarps * paddingPerWarp;
    float* s_real = s_mem;
    float* s_imag = s_mem + paddedLength;
    int tid = threadIdx.x;
    auto paddedIndex = [=](int i) {
        int warpId = i / warpSize;
        int offset = warpId * paddingPerWarp;
        return i + offset;
        };
    float2 val = d_data[tid];
    s_real[paddedIndex(tid)] = val.x;
    s_imag[paddedIndex(tid)] = val.y;
    for (int s = 1; s <= logN; s++) {
        int m = 1 << s;
        int m2 = m >> 1;
        __syncthreads();
        if (tid < N / 2) {
            int group = tid / m2;
            int j = tid % m2;
            int k = group * m;
            float angle = -2.0f * (float)M_PI * (float)j / (float)m;
            float2 w = make_float2(cosf(angle), sinf(angle));
            int i1 = k + j;
            int i2 = k + j + m2;
            float ur = s_real[paddedIndex(i1)];
            float ui = s_imag[paddedIndex(i1)];
            float tr = s_real[paddedIndex(i2)];
            float ti = s_imag[paddedIndex(i2)];
            float temp_r = w.x * tr - w.y * ti;
            float temp_i = w.x * ti + w.y * tr;
            s_real[paddedIndex(i1)] = ur + temp_r;
            s_imag[paddedIndex(i1)] = ui + temp_i;
            s_real[paddedIndex(i2)] = ur - temp_r;
            s_imag[paddedIndex(i2)] = ui - temp_i;
        }
    }
    __syncthreads();
    val.x = s_real[paddedIndex(tid)];
    val.y = s_imag[paddedIndex(tid)];
    d_data[tid] = val;
}
```

```
__global__ void fft_kernel_sm_all_stages_padding(float2* d_data, int N, int logN)
{
    extern __shared__ float s_mem[];

    const int tid = threadIdx.x;
    const int warpSize = WARP_SIZE;
    const int paddingPerWarp = PADDING_SIZE;
    const int numWarps = N / warpSize;
    const int paddedLength = N + numWarps * paddingPerWarp;
    float* s_real = s_mem;
    float* s_imag = s_mem + paddedLength;
    int warpId_t = tid / warpSize;
    int pTid = tid + warpId_t * paddingPerWarp;
    float2 val = d_data[tid];
    s_real[pTid] = val.x;
    s_imag[pTid] = val.y;
    for (int s = 1; s <= logN; s++) {
        int m = 1 << s;
        int m2 = m >> 1;
        float angleBase = -2.0f * (float)M_PI / (float)m;
        __syncthreads();
        if (tid < N / 2) {
            int group = tid / m2;
            int j = tid % m2;
            int k = group * m;
            float angle = angleBase * j;
            float2 w = make_float2(__cosf(angle), __sinf(angle));
            int i1 = k + j;
            int i2 = k + j + m2;
            int warpId_i1 = i1 / warpSize;
            int pI1 = i1 + warpId_i1 * paddingPerWarp;
            int warpId_i2 = i2 / warpSize;
            int pI2 = i2 + warpId_i2 * paddingPerWarp;
            float ur = s_real[pI1];
            float ui = s_imag[pI1];
            float tr = s_real[pI2];
            float ti = s_imag[pI2];
            float temp_r = w.x * tr - w.y * ti;
            float temp_i = w.x * ti + w.y * tr;
            s_real[pI1] = ur + temp_r;
            s_imag[pI1] = ui + temp_i;
            s_real[pI2] = ur - temp_r;
            s_imag[pI2] = ui - temp_i;
        }
    }
    __syncthreads();
    val.x = s_real[pTid];
    val.y = s_imag[pTid];
    d_data[tid] = val;
}
```

Optimized GPU: Synchronization Overhead




```
for (int i = 0; i < numSegments; i++)
    cudaStreamSynchronize(streams[i]);

if (N > baseSize) {
    for (int s = logBaseSize + 1; s <= logN; s++) {
        int m = 1 << s;
        int totalPairs = N / 2;
        int gridSize = (totalPairs + blockSize - 1) / blockSize;
        fft_stage_kernel << <gridSize, blockSize >> > (d_data, N, s);
    }
}
```

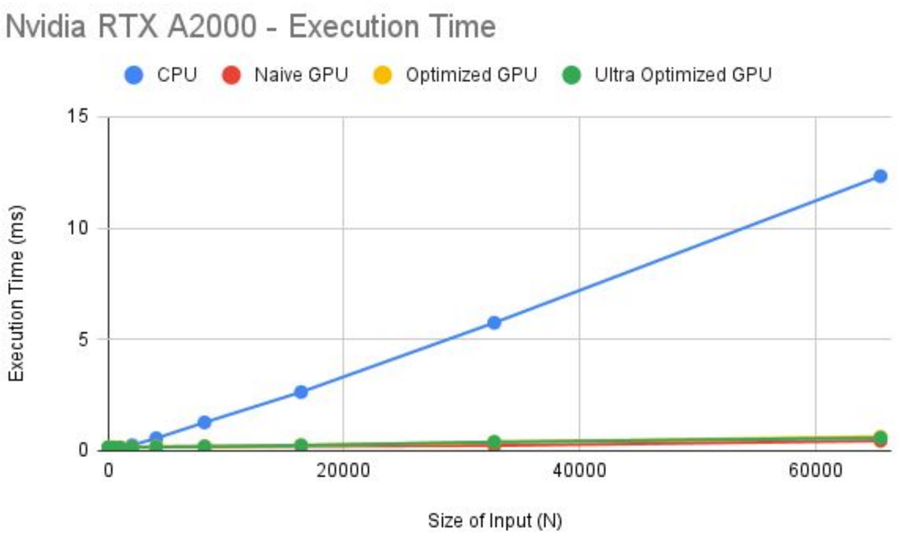


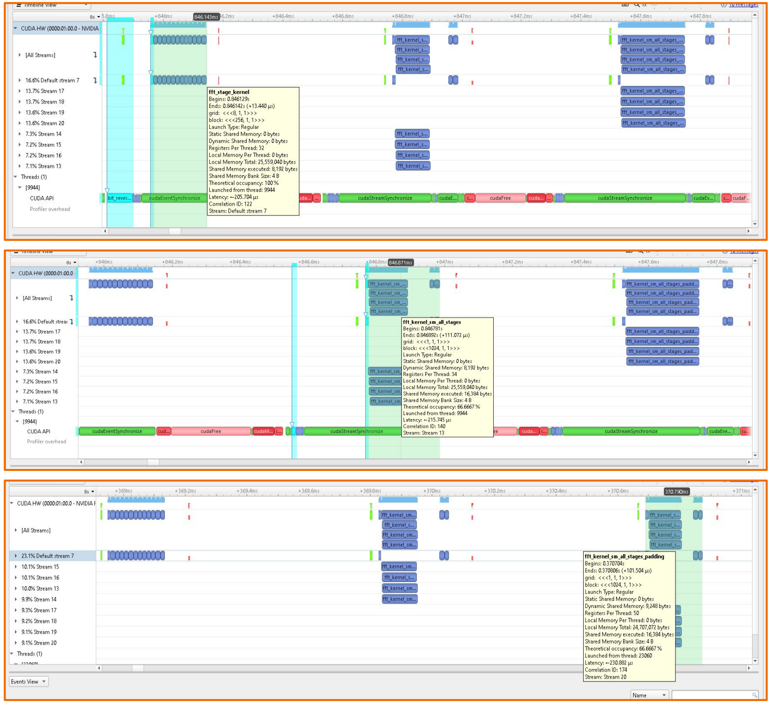

```
// for (int i = 0; i < numSegments; i++)
    // cudaStreamSynchronize(streams[i]);

if (N > baseSize) {
    for (int s = logBaseSize + 1; s <= logN; s++) {
        int m = 1 << s;
        int totalPairs = N / 2;
        int gridSize = (totalPairs + blockSize - 1) / blockSize;
        fft_stage_kernel << <gridSize, blockSize, 0, streams[numSegments - 1] >> > (d_data, N, s);
    }
}
```

N = 4096

DEMO

Results





N = 4096

Limitations & Future Scope

- Algorithmic limitations:
  - Dependence on Input size
  - Sequential dependency
  - Butterfly design
- Shared memory size limitation
- Synchronization overhead
- Floating point data:
  - Precision vs Performance
  - Cascade of error

- Hybrid algorithms, zero-padding (Radix-4/8, Stockham Algorithm)
- Mixed precision (Tensor Cores), error-compensating algorithms
- Batch Processing, Hardware-Aware Optimizations
- State-of-the-art FFT Libraries

References

[1] Burrus, Sidney. “The Cooley-Tukey Fast Fourier Transform Algorithm ∗ C .” (2014).

[2] https://www.wikiwand.com/en/articles/Fast_Fourier_transform

[3] https://www.youtube.com/watch?v=h7apO7q16V0&t=1336s&ab_channel=Reducible

[4] https://github.com/KAdamek/SMFFT

[5] https://forums.developer.nvidia.com/t/does-cufft-show-much-higher-efficiency-than-cpu-fft-routines/17790/4

[6] https://github.com/roguh/cuda-fft/tree/main

[7] https://github.com/anair-eng/CUDA-MPI-pthreads-FFT

[8] https://cs.wmich.edu/gupta/teaching/cs5260/5260Sp15web/studentProjects/tiba&hussein/03278999.pdf

Thank You!

Appendix

Nvidia RTX A2000

N	blockSize	CPU	Naive GPU	Optimized GPU	Ultra Optimized
32	256	0.0089	0.0768	0.155648	0.16384
64	256	0.0118	0.090112	0.149504	0.14336
128	256	0.0178	0.1024	0.151552	0.141312
256	256	0.0301	0.114688	0.149536	0.144384
512	256	0.06	0.129024	0.161888	0.142336
1024	256	0.1254	0.154624	0.150528	0.14336
2048	256	0.2513	0.156672	0.164864	0.156672
4096	256	0.5649	0.172032	0.195584	0.171008
8192	256	1.2737	0.186368	0.202752	0.191488
16384	256	2.6371	0.205824	0.246784	0.2456
32768	256	5.7456	0.235552	0.394368	0.395264
65536	256	12.3248	0.449536	0.63376	0.579584