Implementations of DFT and FFT

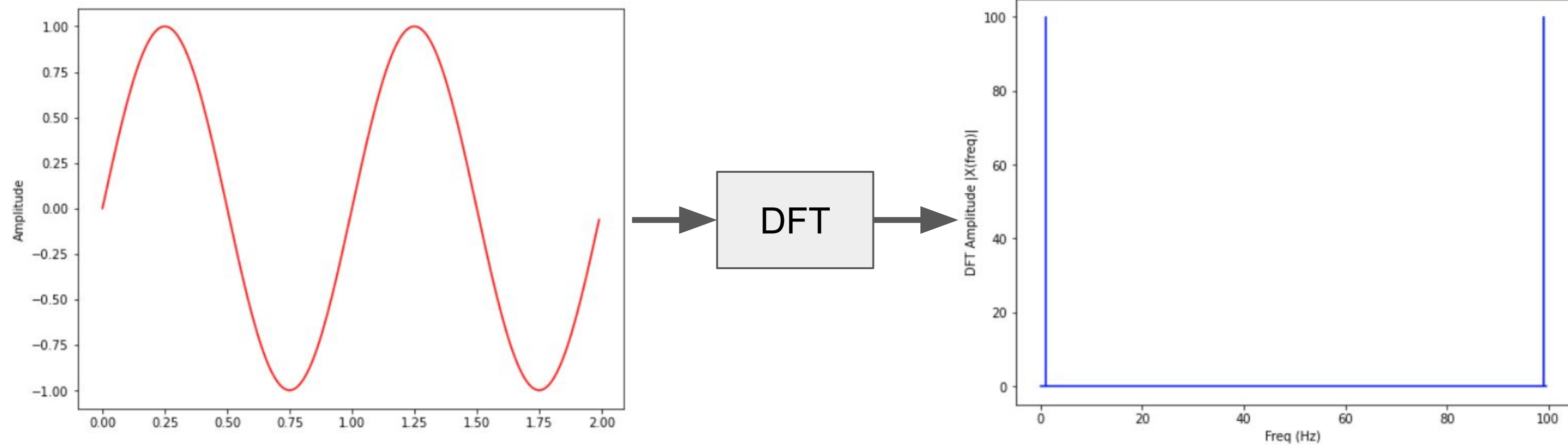
Speaker: Jia-Ming Lin

Outline

- Discrete Fourier Transform(DFT) and its Application
- Matrix-Vector Multiplication Optimization
- HLS implementation of DFT and Optimization
- HLS implementation of FFT and Optimization
- Labs
  - Matrix-Vector Multiplication and Optimization
  - DFT and Optimization
  - FFT and Optimization

Discrete Fourier Transform(DFT) and its Application

- Change a discrete signal in time domain to the same signal in frequency domain.



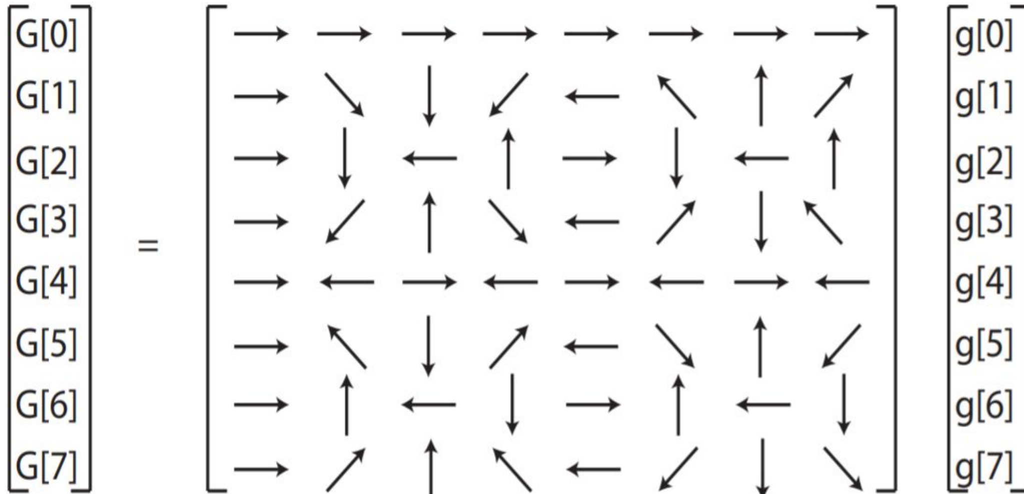
Discrete Fourier Transform(DFT) and its Application

- N point DFT can be determined through $N \times N$ matrix multiplied by a vector of size N
- $g[]$: real valued discrete function
  S : matrix of DFT coefficients
  $G[]$: converted function in frequency domain

$$G = S \cdot g \text{ where } S = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & s & s^2 & \cdots & s^{N-1} \\ 1 & s^2 & s^4 & \cdots & s^{2(N-1)} \\ 1 & s^3 & s^6 & \cdots & s^{3(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & s^{N-1} & s^{2(N-1)} & \cdots & s^{(N-1)(N-1)} \end{bmatrix} \text{ and } s = e^{\frac{-j2\pi}{N}}$$

Discrete Fourier Transform(DFT) and its Application

- Example: visualization of the DFT coefficients for an **8** point DFT operation



- Multiply by Row 0:
  Rotate by 0 degree
- Multiply by Row **1**:
  Rotate by
  1*(**360/8**)*j = 1*(45)*j degree
- Multiply by Row **2**:
  Rotate by
  2*(**360/8**)*j = 2*(45)*j degree
- ...
- Multiply by Row **N-1**:
  Rotate by
  (N-1)*(**360/8**)*j = (N-1)*(45)*j degree

Discrete Fourier Transform(DFT) and its Application

- Application: MP3 Audio Compression

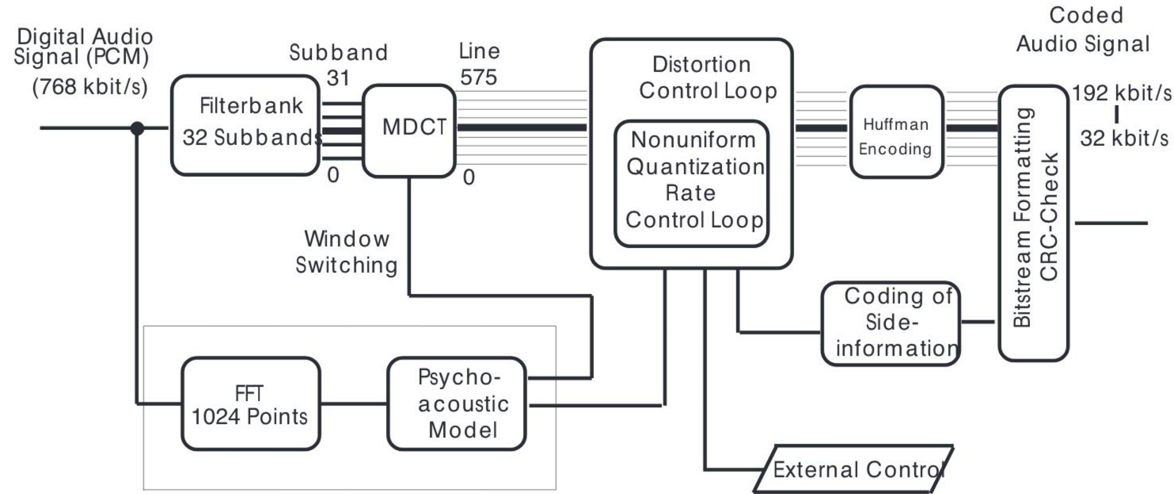


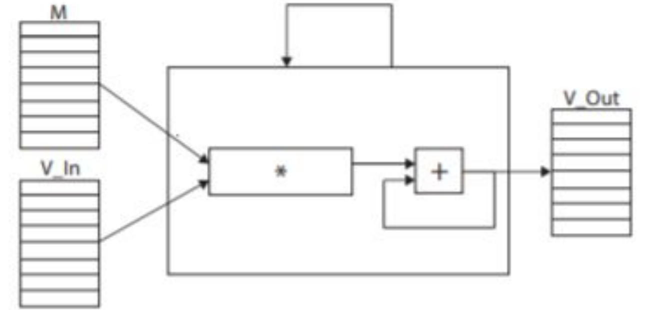
Figure 2: Block diagram of an MPEG-1 Layer-3 encoder.

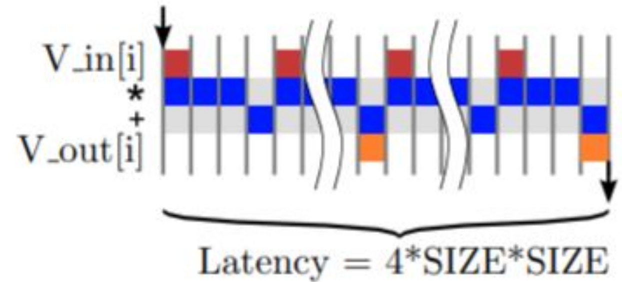
https://www.projectrhea.org/rhea/index.php/Use_of_Fourier_Transforms_in_MP3_Audio_Compression

Matrix-Vector Multiplication Optimization

Baseline

```
#include"top.h"

#define SIZE 8
typedef ap_int<32> BaseType;

void matrix_vector(
        BaseType M[SIZE][SIZE],
        BaseType v_in[SIZE],
        BaseType v_out[SIZE]){

    int i, j;
    DATA_LOOP: for(i = 0; i < SIZE; i++){
        BaseType sum = 0;
        PRODUCT_LOOP: for(j = 0; j < SIZE; j++){
            sum += M[i][j] * v_in[j];
        }
        v_out[i] = sum;
    }
}
```





- **Advantage**: Minimal resource consumption
- **Disadvantage**: Long task latency and task interval

Pipelining and Parallelism

- 32-bit x 32-bit integer multiplier
  - Chained 3 DSPs, takes 3 cycles to complete
- Pipelined Multiplier ⇒ at most 3 concurrent multiplications.

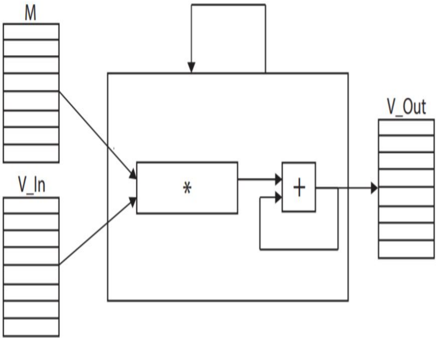
DSP_1						
DSP_2						
DSP_3						
ADD						

```
PRODUCT_LOOP: for(j = 0; j < SIZE; j++){
#pragma HLS PIPELINE
    sum += M[i][j] * v_in[j];
}
```

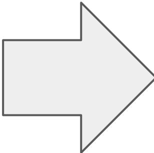
PRODUCT_ LOOP	Task Interval	Task Latency	Total Latency
Without Pipeline	4	4	4*8=32
With Pipeline	1	4	8+(4-1)=11

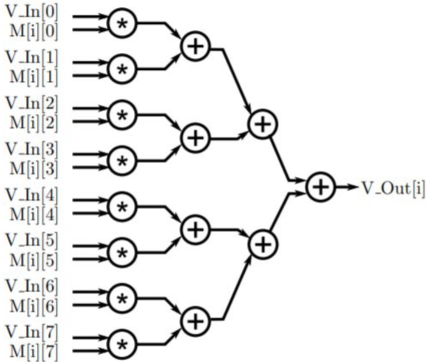
Pipelining and Parallelism

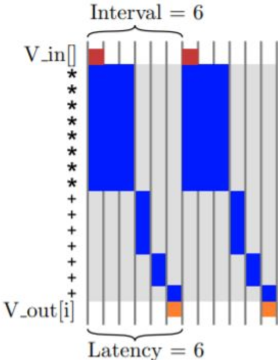
- Parallelize multiplications in vectors' product



Fully Unroll + Array Partition

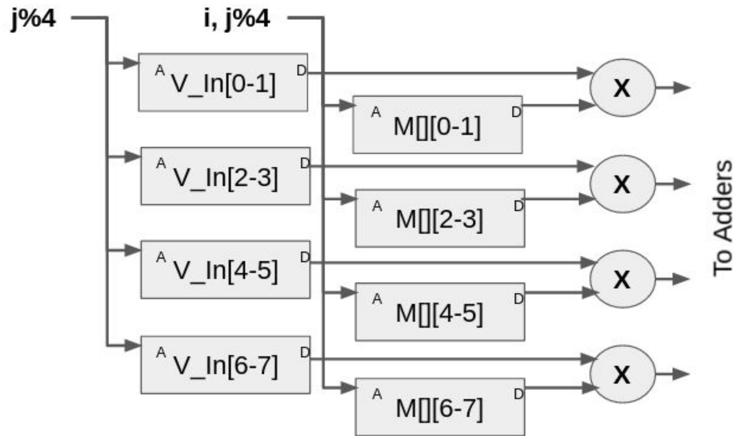






Pipelining and Parallelism

- Resources and Performance Tradeoff
  - When SIZE = 1024, it is not possible to fully unroll due to resource constraints.
    - We consider this case in Lab
- Partially unroll with specifying factor
  - Example: **(1)**SIZE = 8, **(2)** four parallel 32-bit x 32-bit multipliers




```
#pragma HLS RESOURCE variable=M core=RAM_1P_BRAM
#pragma HLS RESOURCE variable=v_in core=RAM_1P_BRAM

#pragma HLS ARRAY_PARTITION variable=M cyclic factor=4 dim=2
#pragma HLS ARRAY_PARTITION variable=v_in cyclic factor=4 dim=1

    int i, j;
    DATA_LOOP: for(i = 0; i < SIZE; i++){
        BaseType sum = 0;

        PRODUCT_LOOP: for(j = 0; j < SIZE; j++){
#pragma HLS UNROLL factor=4
            sum += M[i][j] * v_in[j];
        }

        v_out[i] = sum;
    }
```

HLS implementation of DFT and Optimization

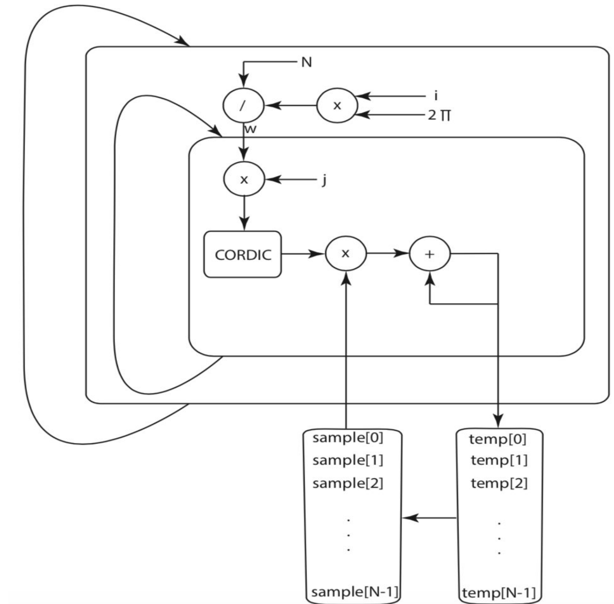
Baseline

- In MP3 audio compression, the size of DFT is 1024.
- Implementing DFT by directly using matrix-vector multiplication is not practical.
  - Since DFT coefficient array is sized 1024x1024
  - Computing the coefficients of DFT in each iteration of the inner loop.

S

G[0] G[1] G[2] G[3] G[4] G[5] G[6] G[7] = S g[0] g[1] g[2] g[3] g[4] g[5] g[6] g[7]

- g[j] is a complex number
- Multiplying by S[i][j] is said to perform rotation



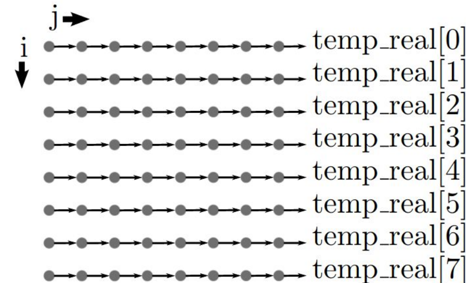
Optimization: loop interchange

- Pipeline on loop "j"

```
for (i = 0; i < SIZE; i += 1) {      <- Loop i
    temp_real[i] = 0;
    temp_imag[i] = 0;

    w = (2.0 * 3.141592653589  / SIZE) * (TEMP_TYPE)i;

    for (j = 0; j < SIZE; j ++) {    <- Loop j
    #pragma HLS PIPELINE II=1
        c = cos(j * w);
        s = -sin(j * w);

        temp_real[i] += (sample_real[j] * c - sample_imag[j] * s);
        temp_imag[i] += (sample_real[j] * s + sample_imag[j] * c);
    }
}
```

- Since data type is float, the critical path is on float adder, takes 5 cycles.
- **temp_real[i]** value depends on result from previous iteration.
- Loop "j" can not be pipelined with II=1

- Interchanging loop "i" and "j"
  - Iterating through "i" before "j"
  - Then pipelining in loop "i"



HLS implementation of FFT and Optimization

Background

- The DFT requires O(n^2) multiply and add operations.
  - Matrix-vector multiplication
- The FFT requires only O(n*log(n))
  - **Divide-and-conquer approach** based on the symmetry of DFT coefficient matrix
  - Refer to this video for basic concept.

Evaluation

Given $f(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$, evaluate on $[1, 2, \ldots, n]$

$$f(1) = a_0 + a_1 \cdot 1 + \cdots + a_{n-1} \cdot 1^{n-1}$$
$$f(2) = a_0 + a_1 \cdot 2 + \cdots + a_{n-1} \cdot 2^{n-1}$$
$$f(3) = a_0 + a_1 \cdot 3 + \cdots + a_{n-1} \cdot 3^{n-1}$$
$$\cdots$$
$$f(n) = a_0 + a_1 \cdot n + \cdots + a_{n-1} \cdot n^{n-1}$$

$$\Leftrightarrow \begin{bmatrix} 1 & 1 & \ldots & 1 \\ 1 & 2 & \ldots & 2^{n-1} \\ \ldots & \ldots & \ldots & \ldots \\ 1 & n & \ldots & n^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \ldots \\ a_{n-1} \end{bmatrix}$$

Complexity = $O(n^2)$

Can we do better?

Note: denote $f(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$ as $[a_0, a_1, \ldots, a_{n-1}]$

Evaluation

Even Function:

$$f_e(x) = f_e(-x)$$

Example:

$$f_e(x) = 1 + x^2$$

Odd Function:

$$f_o(-x) = -f_o(x)$$

Example:

$$f_o(x) = x + 2x^3$$

Represent as even and odd:

$$f(x) = 3 + x + x^2 + 4x^3$$

$$= 3 + x^2 + (x + 4x^3)$$

Choose evaluation points

$$[\pm x_1, \pm x_2, \dots, \pm x_{\frac{n}{2}}]$$

Evaluation

Problem:
Evaluate $f(x)$: $[a_0, a_1, \dots, a_{n-1}]$ **on** $[\pm x_1, \pm x_2, \dots, \pm x_{\frac{n}{2}}]$

$$f(x) = f_e(x^2) + x f_o(x^2)$$

Sub-problem 1
Evaluate $f_e(x^2)$: $[a_0, a_2, \dots, a_{n-2}]$
on $[x_1^2, x_2^2, \dots, x_{\frac{n}{2}}^2]$

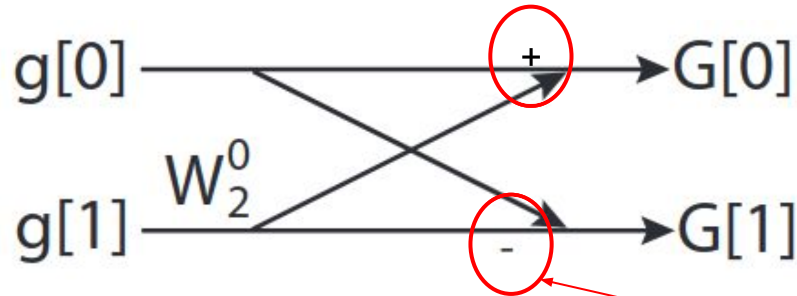
Sub-problem 2
Evaluate $f_o(x^2)$: $[a_1, a_3, \dots, a_{n-1}]$
on $[x_1^2, x_2^2, \dots, x_{\frac{n}{2}}^2]$

$$f(x_i) = f_e(x_i^2) + x_i f_o(x_i^2)$$
$$f(-x_i) = f_e(x_i^2) - x_i f_o(x_i^2)$$

Recursive Algotithm $O(n \log n)$

Background

- We usually represent computations in butterfly architecture.



$$G[0] = g[0] + W_2^0 g[1]$$

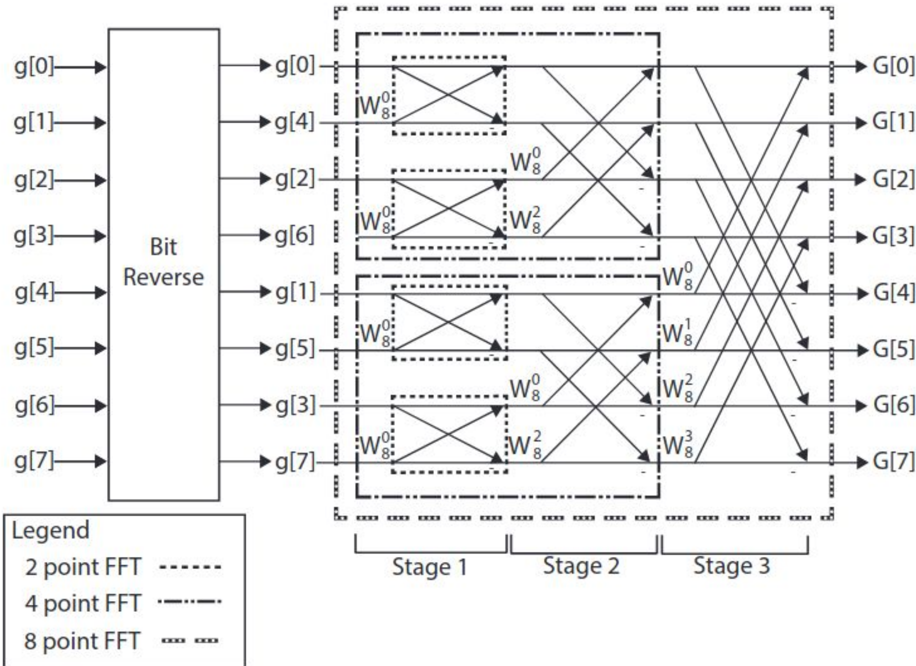
$$G[1] = g[0] - W_2^0 g[1]$$

Multiply by “-1” at lower position

W is rotation transform

Background

- Butterfly architecture for 8 point FFT.



Where $W_N^k = e^{\frac{-j2\pi \cdot k}{N}}$

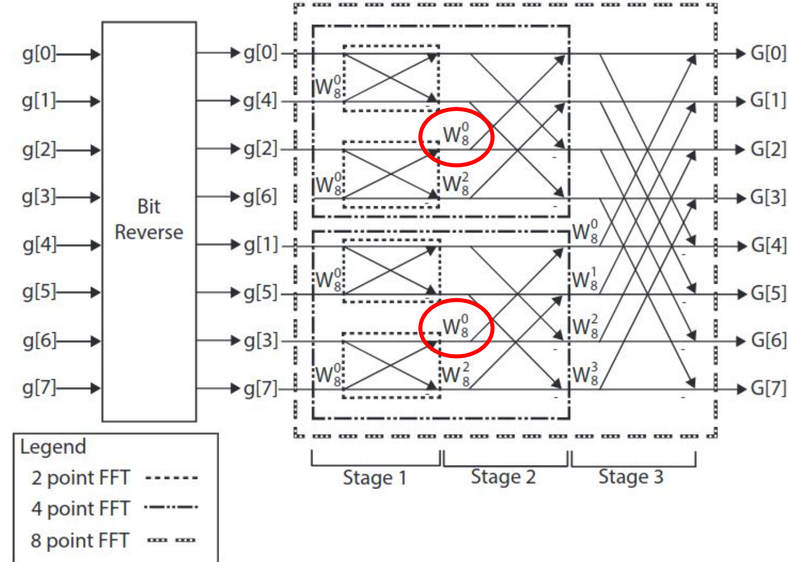
Multiply by “-1” on lower position

Number of stages = logN

Stages are executed sequentially

Initial Implementation for Stage

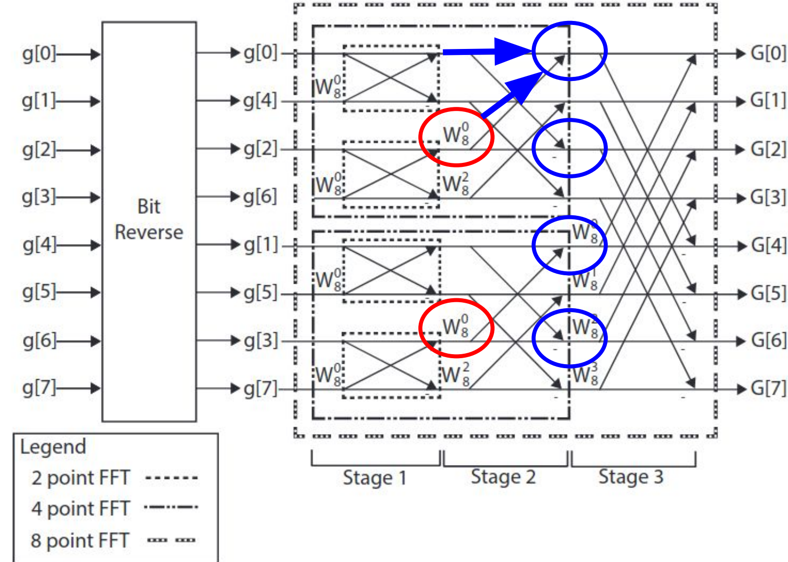
```
stage_loop:
    for (stage = 1; stage <= M; stage++) {
        DFTpts = 1 << stage; // DFT = 2^stage = points in sub DFT
        numBF = DFTpts / 2; // Butterfly WIDTHS in sub-DFT
        k = 0;
        e = -6.283185307178 / DFTpts;
        a = 0.0;
        // Perform butterflies for j-th stage
        butterfly: for (j = 0; j < numBF; j++) {
            c = cos(a); s = sin(a);
            a = a + e;
            // Compute butterflies that use same W**k
            DFTpts: for (i = j; i < SIZE; i += DFTpts) {
                i_lower = i + numBF;
                temp_R = X_R[i_lower] * c - X_I[i_lower] * s;
                temp_I = X_I[i_lower] * c + X_R[i_lower] * s;



            }
            k += step;
        }
        step = step / 2;
    }
```



- Compute rotated vector.

Initial Implementation for Stage

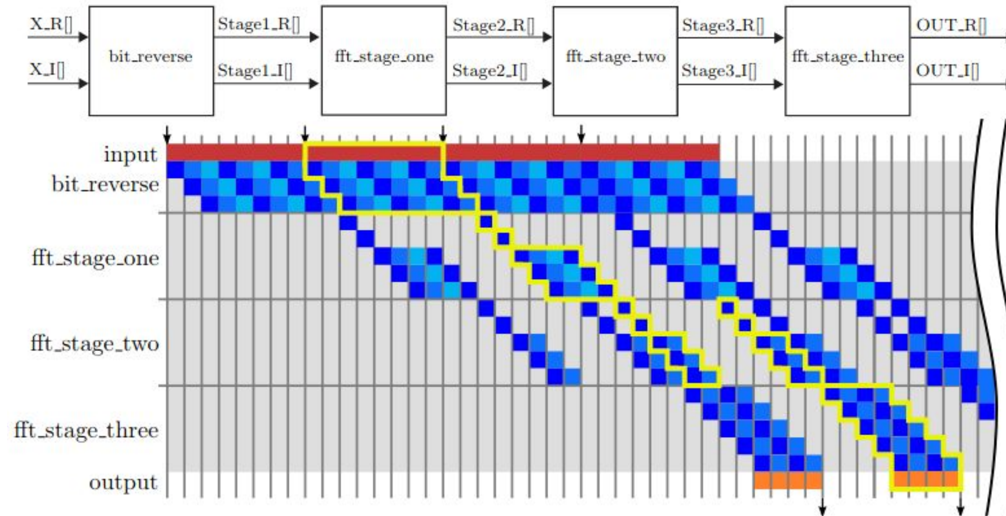
```
stage_loop:
    for (stage = 1; stage <= M; stage++) {
        DFTpts = 1 << stage; // DFT = 2^stage = points in sub DFT
        numBF = DFTpts / 2; // Butterfly WIDTHS in sub-DFT
        k = 0;
        e = -6.283185307178 / DFTpts;
        a = 0.0;
        // Perform butterflies for j-th stage
        butterfly: for (j = 0; j < numBF; j++) {
            c = cos(a); s = sin(a);
            a = a + e;
            // Compute butterflies that use same W**k
            DFTpts: for (i = j; i < SIZE; i += DFTpts) {
                i_lower = i + numBF;
                temp_R = X_R[i_lower] * c - X_I[i_lower] * s;
                temp_I = X_I[i_lower] * c + X_R[i_lower] * s;
                X_R[i_lower] = X_R[i] - temp_R;
                X_I[i_lower] = X_I[i] - temp_I;
                X_R[i] = X_R[i] + temp_R;
                X_I[i] = X_I[i] + temp_I;
            }
            k += step;
        }
        step = step / 2;
    }
```



- Compute stage output

Increasing Throughput: Task Pipelining

- Using dataflow implementation
  - Computing multiple FFTs concurrently
  - Example: 4 concurrent 8 point FFT computations.

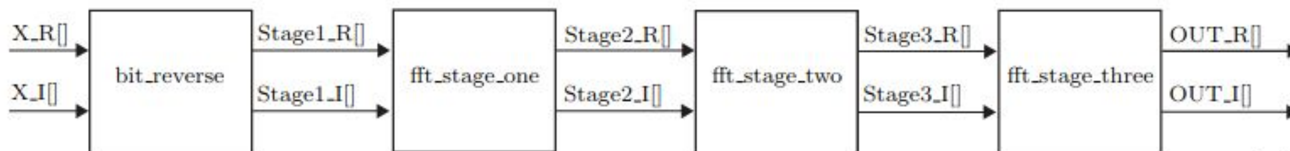


Increasing Throughput: Task Pipelining

- Using dataflow implementation

```
void fft(DTYPE X_R[SIZE], DTYPE X_I[SIZE], DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE])
{
    #pragma HLS dataflow
    DTYPE Stage1_R[SIZE], Stage1_I[SIZE];
    DTYPE Stage2_R[SIZE], Stage2_I[SIZE];
    DTYPE Stage3_R[SIZE], Stage3_I[SIZE];

    bit_reverse(X_R, X_I, Stage1_R, Stage1_I);
    fft_stage_one(Stage1_R, Stage1_I, Stage2_R, Stage2_I);
    fft_stages_two(Stage2_R, Stage2_I, Stage3_R, Stage3_R);
    fft_stage_three(Stage3_R, Stage3_I, OUT_R, OUT_I);
}
```

- HLS pragma
- Ping-Pong Buffer



Increasing Throughput: Task Pipelining

- Example: 1024 point FFT implementation
  - Before task pipelining
    - Max task latency = 2.3M cycles
    - Max task interval = 2.3M cycles
  - After task pipelining
    - Max task latency = 6k cycles
    - Max task interval = 1k cycles
  - **2000x throughput improvement than non-optimized version.**

		Before	After
Latency (cycles)	min	1794	6319
	max	2361054	6319
Latency (absolute)	min	17.940 us	63.190 us
	max	23.611 ms	63.190 us
Interval (cycles)	min	1794	1028
	max	2361054	1028

Reduce DSP Consumption

- In each stage, cos and sin are implemented using CORDIC
    - Which don’t consume DSP resource
- However, in the FFT algorithm, not much angles we need to compute.
    - Ex. 1024 point FFT only needs fixed 512 cos and sin values
    - We precompute all the values and store in BRAM
        - Consuming only 4 BRAMs in total 280 available on PYNQ-Z2

Reduce DSP Consumption

- In the 1024 point FFT, there are 10 stages

```
void fft(DTYPE X_R[SIZE], DTYPE X_I[SIZE], DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE])
{
#pragma HLS DATAFLOW

    //Call fft
    DTYPE Stage1_R[SIZE], Stage1_I[SIZE];
    DTYPE Stage2_R[SIZE], Stage2_I[SIZE];
    DTYPE Stage3_R[SIZE], Stage3_I[SIZE];
    DTYPE Stage4_R[SIZE], Stage4_I[SIZE];
    DTYPE Stage5_R[SIZE], Stage5_I[SIZE];
    DTYPE Stage6_R[SIZE], Stage6_I[SIZE];
    DTYPE Stage7_R[SIZE], Stage7_I[SIZE];
    DTYPE Stage8_R[SIZE], Stage8_I[SIZE];
    DTYPE Stage9_R[SIZE], Stage9_I[SIZE];
    DTYPE Stage10_R[SIZE], Stage10_I[SIZE];

    bit_reverse(X_R, X_I, Stage1_R, Stage1_I);

    fft_stages(Stage1_R, Stage1_I, 1, Stage2_R, Stage2_I);
    fft_stages(Stage2_R, Stage2_I, 2, Stage3_R, Stage3_I);
    fft_stages(Stage3_R, Stage3_I, 3, Stage4_R, Stage4_I);
    fft_stages(Stage4_R, Stage4_I, 4, Stage5_R, Stage5_I);
    fft_stages(Stage5_R, Stage5_I, 5, Stage6_R, Stage6_I);
    fft_stages(Stage6_R, Stage6_I, 6, Stage7_R, Stage7_I);
    fft_stages(Stage7_R, Stage7_I, 7, Stage8_R, Stage8_I);
    fft_stages(Stage8_R, Stage8_I, 8, Stage9_R, Stage9_I);
    fft_stages(Stage9_R, Stage9_I, 9, Stage10_R, Stage10_I);
    fft_stages(Stage10_R, Stage10_I, 10, OUT_R, OUT_I);
}
```

- Each stage cost 24 DSPs for float point multiplication and addition
- 10 stages need 240 DSPs in total
  - Exceed the available DSP resource on PYNQ-Z2
  - 220 DSPs on PYNQ-Z2

Reduce DSP Consumption

- For stage 1 computation, we actually do not need any DSP.

```
void fft_stages(DTYPE X_R[SIZE], DTYPE X_I[SIZE],
        int stage, DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]) {

    int DFTpts = 1 << stage;    // DFT = 2^stage = points in sub DFT
    int numBF = DFTpts / 2;     // Butterfly WIDTHS in sub-DFT
    int step = SIZE >> stage;
    // Perform butterflies for j-th stage
   butterfly_loop:
    for (int j = 0; j < numBF; j++) {
      // Compute butterflies that use same W**k
    dft_loop:
        for(int t = 0; t < step; t++) {
            int i = j + t*DFTpts;
            //    for (int i = j; i < SIZE; i += DFTpts) {
  #pragma HLS pipeline
            int k = j*step;
        DTYPE c = W_real[k];
        DTYPE s = W_imag[k];
        int i_lower = i + numBF; // index of lower point in butterfly
        DTYPE temp_R = (DTYPE)X_R[i_lower] * c - (DTYPE)X_I[i_lower] * s;
        DTYPE temp_I = (DTYPE)X_I[i_lower] * c + (DTYPE)X_R[i_lower] * s;
        OUT_R[i_lower] = X_R[i] - (DTYPE)temp_R;
        OUT_I[i_lower] = X_I[i] - (DTYPE)temp_I;
        OUT_R[i] = X_R[i] + (DTYPE)temp_R;
        OUT_I[i] = X_I[i] + (DTYPE)temp_I;
      }
    }

}
```

```
void fft_stage_first(DTYPE X_R[SIZE], DTYPE X_I[SIZE],
        DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]) {

    dft_loop:
        for(int t = 0; t < SIZE/2; t++) {
            int i = t*2;
            //    for (int i = j; i < SIZE; i += DFTpts) {
  #pragma HLS pipeline

        int i_lower = i + 1; // index of lower point in butterfly
        DTYPE temp_R = X_R[i_lower];
        DTYPE temp_I = X_I[i_lower];
        OUT_R[i_lower] = (DTYPE)((FTYPE)X_R[i] - (FTYPE)temp_R);
        OUT_I[i_lower] = (DTYPE)((FTYPE)X_I[i] - (FTYPE)temp_I);
        OUT_R[i] = (DTYPE)((FTYPE)X_R[i] + (FTYPE)temp_R);
        OUT_I[i] = (DTYPE)((FTYPE)X_I[i] + (FTYPE)temp_I);
      }
}
```

- **Specialize a design for stage 1**
- **At stage 1, rotation is not needed**
- **Using fixed point arithmetic.**

Reduce DSP Consumption

```
void fft(DTYPE X_R[SIZE], DTYPE X_I[SIZE], DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE])
{
#pragma HLS DATAFLOW

    //Call fft
    DTYPE Stage1_R[SIZE], Stage1_I[SIZE];
    DTYPE Stage2_R[SIZE], Stage2_I[SIZE];
    DTYPE Stage3_R[SIZE], Stage3_I[SIZE];
    DTYPE Stage4_R[SIZE], Stage4_I[SIZE];
    DTYPE Stage5_R[SIZE], Stage5_I[SIZE];
    DTYPE Stage6_R[SIZE], Stage6_I[SIZE];
    DTYPE Stage7_R[SIZE], Stage7_I[SIZE];
    DTYPE Stage8_R[SIZE], Stage8_I[SIZE];
    DTYPE Stage9_R[SIZE], Stage9_I[SIZE];
    DTYPE Stage10_R[SIZE], Stage10_I[SIZE];

    bit_reverse(X_R, X_I, Stage1_R, Stage1_I);

//  fft_stages(Stage1_R, Stage1_I, 1, Stage2_R, Stage2_I);
    fft_stage_first(Stage1_R, Stage1_I, Stage2_R, Stage2_I);
    fft_stages(Stage2_R, Stage2_I, 2, Stage3_R, Stage3_I);
    fft_stages(Stage3_R, Stage3_I, 3, Stage4_R, Stage4_I);
    fft_stages(Stage4_R, Stage4_I, 4, Stage5_R, Stage5_I);
    fft_stages(Stage5_R, Stage5_I, 5, Stage6_R, Stage6_I);
    fft_stages(Stage6_R, Stage6_I, 6, Stage7_R, Stage7_I);
    fft_stages(Stage7_R, Stage7_I, 7, Stage8_R, Stage8_I);
    fft_stages(Stage8_R, Stage8_I, 8, Stage9_R, Stage9_I);
    fft_stages(Stage9_R, Stage9_I, 9, Stage10_R, Stage10_I);
    fft_stages(Stage10_R, Stage10_I, 10, OUT_R, OUT_I);
```

Replace stage 1

- **DSP consumption reduced from 240 to 216**
  - **Available DSPs on PYNQ-Z2 is 220**
- **Without any accuracy loss**
  - **We using ap_fixed<32,11> arithmetic.**
  - **RMSE(R) = 0.000450454419479**
  - **RMSE(I) = 0.00054114358499б**
- **Performance:**
  - **Clock rate @ 100MHz**
  - **Task interval = 1028 cycles**
  - **Throughput = 97 kHZ**

Bits Reversing and Optimizations

- Assume 8 point FFT
  - Index size can be represented by 3-bit unsigned integer.

Index	Binary	Reversed Binary	Reversed Index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

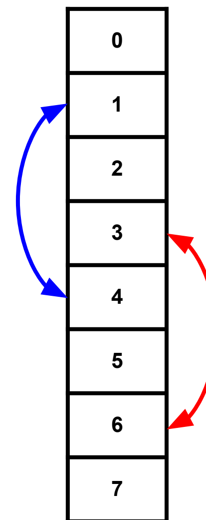
```
void bit_reverse(
        DTYPE X_R[SIZE], DTYPE X_I[SIZE],
        DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]){

    unsigned int reversed;
    unsigned int i;

    for (int i = 0; i < SIZE; i++) {
#pragma HLS PIPELINE
        reversed = reverse_bits(i); // Find the bit reversed index
          if (i <= reversed) {
            // Swap the real values
            OUT_R[i] = X_R[reversed];
            OUT_R[reversed] = X_R[i];

            // Swap the imaginary values
            OUT_I[i] = X_I[reversed];
            OUT_I[reversed] = X_I[i];
          }
    }
}
```

Reversing pair of value every cycle

Bits Reversing and Optimizations

- Dependency constraints reported by the Vivado compiler
  - II = 2 is synthesised.
  - Inter-Iterations: Read to array after Write to array
- However, in our case, operations are independent across iterations
  - Add two derivatives on dependency
  - Further refer to, HLS Pragma

```
#pragma HLS dependence variable=OUT_R inter false
#pragma HLS dependence variable=OUT_I inter false
```



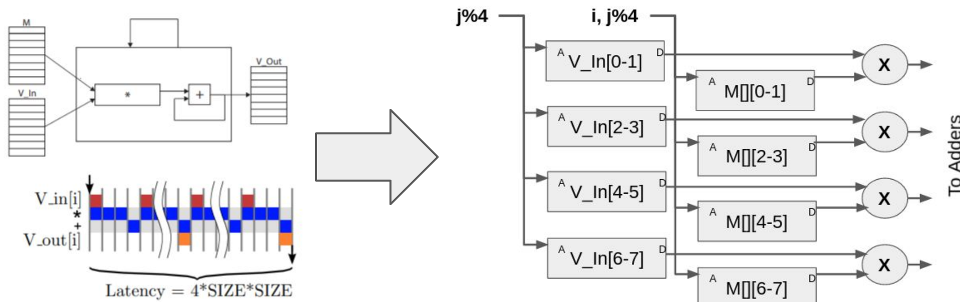
Summarize

- We learned how to convert samples from time domain to frequency domain
  - DFT and FFT
- DFT computation can be mapped to Matrix-Vector Multiplication
  - And optimizing through similar techniques.
- Arithmetic complexity can be further reduced by using FFT algorithm
  - However, significant efforts should be considered to have efficient hardware implementation.
  - Our implementation on PYNQ-Z2 for 1024 point FFT can achieve throughput about 97 kHz
    - 2000x improvement in throughput than non-optimized version.

Labs

Lab 0: Matrix-Vector Multiplication and Optimizations

- Starting from baseline version without any optimization.
  - [Download here](#).
  - Size = 1024, data type = 32-bit integer
- Optimizations
  - Pipeline
  - Partial Unroll
  - Memory Partition.

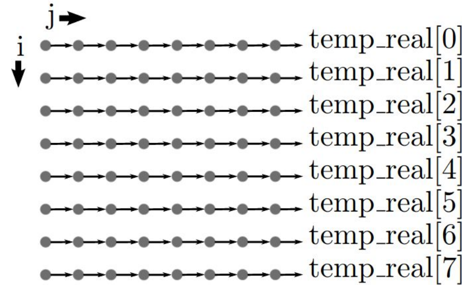


Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
65628	65628	0.656 ms	0.656 ms	65628	65628	none

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	132	-
FIFO	-	-	-	-	-
Instance	-	82	5773	11766	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	75	-
Register	0	-	2624	438	-
Total	0	82	8397	12411	0
Available	280	220	106400	53200	0
Utilization (%)	0	37	7	23	0

Lab1: DFT and Optimizations

- Starting from baseline version without any optimization.
  - Download here.
  - Size = 1024, data type = float
- Optimizations
  - Pipeline
  - Loop Interchange



Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1051717	1051717	10.517 ms	10.517 ms	1051717	1051717	none

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	228	-
FIFO	-	-	-	-	-
Instance	32	210	14794	19488	-
Memory	4	-	0	0	0
Multiplexer	-	-	-	200	-
Register	0	-	1486	295	-
Total	36	210	16280	20211	0
Available	280	220	106400	53200	0
Utilization (%)	12	95	15	37	0

Lab2: FFT and Optimizations

- Starting from baseline version without any optimization.
  - Refer here for more instructions, and download initial implementation.
  - Size = 1024, data type = float.
- Optimizations are mentioned in previous slides.