Messages for this week

- Projects
  - Project plan presentations are scheduled for 3 – 5pm today and tomorrow
  - Project plans to be included in the COMP4601 Wiki by the weekend
- Seminars
  - Meet with group, read and discuss papers suggested for your topic
  - Today's lecture is the last; seminars start in Week 7
- Labs
  - Handin on Ch. 4 lab due next Monday
  - Handin on Ch. 5 lab should be released by Friday and is due Monday, Week 8

Ch 5. Fast Fourier Transform

UNSW
SYDNEY

From last week…Discrete Fourier Transform

- DFT is a fundamental signal processing technique
- Transforms a discrete signal (fixed number of samples) in the time domain to a discrete signal in the frequency domain i.e. as a sum of sinusoids
  - This allows for advanced filtering and modulation techniques as well as fast large-integer and polynomial mutiplication
- At its core, DFT performs a matrix-vector multiplication, where the matrix is a fixed set of coefficients

- I assume you have read the mathematical introduction to DFTs at the start of Ch. 4 and the intro to FFTs at the start of Ch. 5

DFT

- The DFT converts a finite number of equally spaced samples into a finite number of complex sinusoids
- Given N complex samples of signal $g[]$ in the time domain, N complex signal values, $G[]$, are computed in the frequency domain using the expression $G = S \cdot g$,

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

Hence, $G[k] = \sum_{n=0}^{N-1} g[n] s^{kn}$ for $k = 0, \cdots, N-1$

A visualization of DFT coefficients (8 point DFT)

$$\begin{bmatrix} G[0] \\ G[1] \\ G[2] \\ G[3] \\ G[4] \\ G[5] \\ G[6] \\ G[7] \end{bmatrix} = \begin{bmatrix} \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow \\ \rightarrow & \searrow & \downarrow & \swarrow & \leftarrow & \nwarrow & \uparrow & \nearrow \\ \rightarrow & \downarrow & \leftarrow & \uparrow & \rightarrow & \downarrow & \leftarrow & \uparrow \\ \rightarrow & \swarrow & \uparrow & \searrow & \leftarrow & \nearrow & \downarrow & \nwarrow \\ \rightarrow & \leftarrow & \rightarrow & \leftarrow & \rightarrow & \leftarrow & \rightarrow & \leftarrow \\ \rightarrow & \nwarrow & \downarrow & \nearrow & \leftarrow & \searrow & \uparrow & \swarrow \\ \rightarrow & \uparrow & \leftarrow & \downarrow & \rightarrow & \uparrow & \leftarrow & \downarrow \\ \rightarrow & \nearrow & \uparrow & \nwarrow & \leftarrow & \swarrow & \downarrow & \searrow \end{bmatrix} \begin{bmatrix} g[0] \\ g[1] \\ g[2] \\ g[3] \\ g[4] \\ g[5] \\ g[6] \\ g[7] \end{bmatrix}$$

- Note the abundance of symmetry in matrix S

Fast Fourier Transform

- The DFT computation using matrix-vector multiplication requires $O(n^2)$ multiply and add operations for an input with n samples
- This complexity can be reduced by exploiting the redundancy in the structure of the coefficient matrix
- The FFT uses divide and conquer on the symmetry of the S matrix to reduce the complexity to $O(n\log n)$
- We'll consider the FFT algorithm developed by Cooley & Tukey and published in 1965

- In the following, we'll take a quick look at the background before studying the algorithm and its optimization

The 2-point DFT

- Let the so-called "*twiddle factor*"

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

- Then we can rewrite the 2-point DFT as

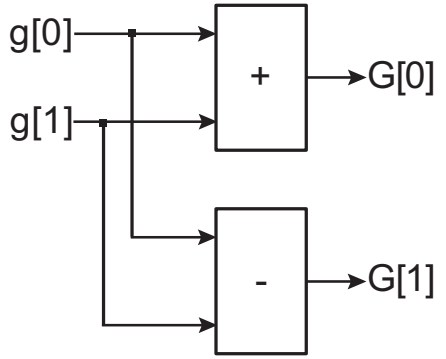
$$\begin{bmatrix} G[0] \\ G[1] \end{bmatrix} = \begin{bmatrix} W_2^{00} & W_2^{01} \\ W_2^{10} & W_2^{11} \end{bmatrix} \cdot \begin{bmatrix} g[0] \\ g[1] \end{bmatrix}$$

which expands to

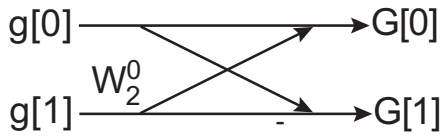
$$\begin{aligned} G[0] &= g[0] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 0}{2}} + g[1] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 1}{2}} \\ &= g[0] + g[1] \end{aligned}$$

$$\begin{aligned} G[1] &= g[0] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 0}{2}} + g[1] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 1}{2}} \\ &= g[0] - g[1] \end{aligned}$$

and can be represented as a dataflow graph



or as a butterfly operation



The 4-point DFT

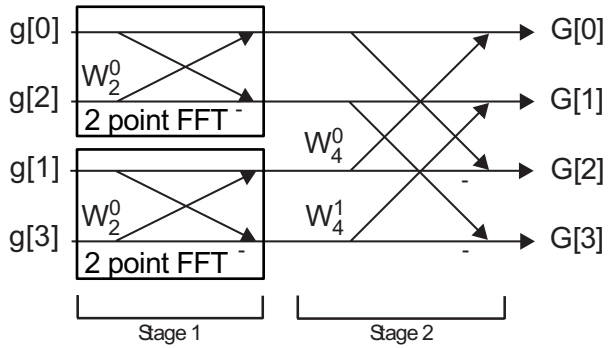
- Similarly, we can write

$$\begin{bmatrix} G[0] \\ G[1] \\ G[2] \\ G[3] \end{bmatrix} = \begin{bmatrix} W_4^{00} & W_4^{01} & W_4^{02} & W_4^{03} \\ W_4^{10} & W_4^{11} & W_4^{12} & W_4^{13} \\ W_4^{20} & W_4^{21} & W_4^{22} & W_4^{23} \\ W_4^{30} & W_4^{31} & W_4^{32} & W_4^{33} \end{bmatrix} \cdot \begin{bmatrix} g[0] \\ g[1] \\ g[2] \\ g[3] \end{bmatrix}$$

as

$$G[0] = (g[0] + g[2]) + e^{\frac{-j2\pi 0}{4}}(g[1] + g[3])$$
$$G[1] = (g[0] - g[2]) + e^{\frac{-j2\pi 1}{4}}(g[1] - g[3])$$
$$G[2] = (g[0] + g[2]) - e^{\frac{-j2\pi 0}{4}}(g[1] + g[3])$$
$$G[3] = (g[0] - g[2]) - e^{\frac{-j2\pi 1}{4}}(g[1] - g[3])$$

- Which leads to the following recursive diagrammatic representation:



UNSW SYDNEY

Deriving the N-point DFT structure

We can rearrange the general expression

$$G[k] = \sum_{n=0}^{N-1} g[n] \cdot e^{\frac{-j2\pi kn}{N}} \text{ for } k = 0,..N-1$$

to obtain

$$G[k] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{\frac{-j2\pi kn}{N/2}} + e^{\frac{-j2\pi k}{N}} \cdot \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{\frac{-j2\pi kn}{N/2}}$$

and write it as

$$G[k] = A_k + W_N^k B_k$$

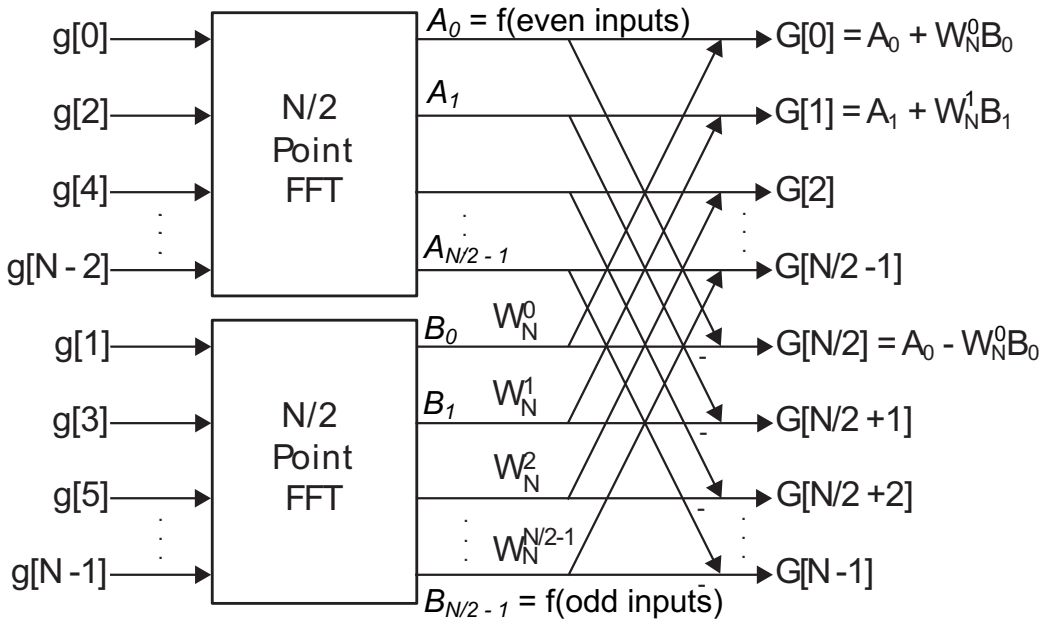
For the N/2 higher frequencies, the general expression can be rearranged as

$$G[k+N/2] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{\frac{-j2\pi kn}{N/2}} - e^{\frac{-j2\pi k}{N}} \cdot \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{\frac{-j2\pi kn}{N/2}}$$

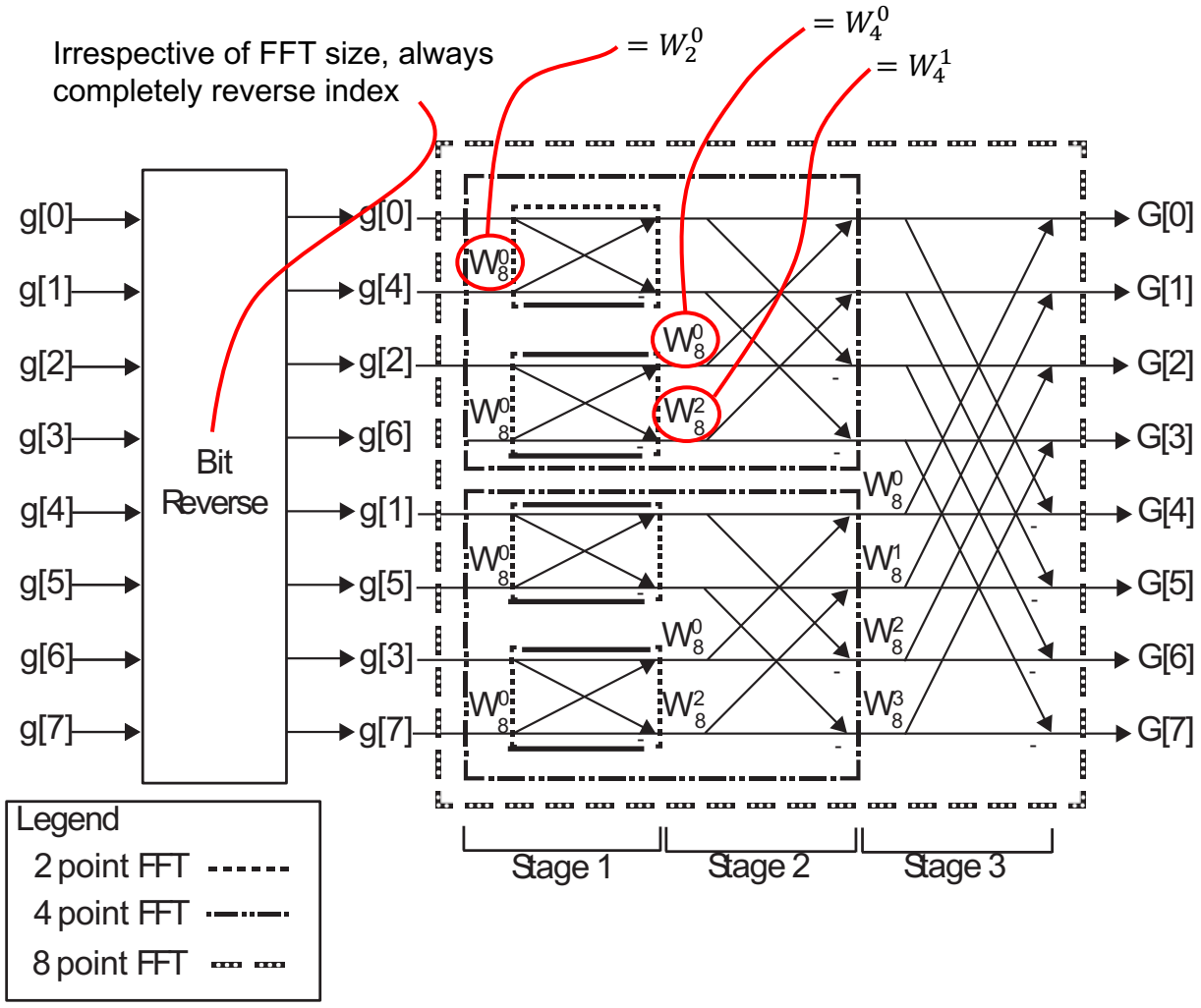
which can be written as

$$G[k+N/2] = A_k - W_N^k B_k$$

A general recursive structure for an N-point DFT is thus obtained:



An 8-point FFT



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

Obtained by recursively dividing the indices into smaller lists of "even" and "odd" entries i.e.

$$
\begin{array}{ccc}
0 \rightarrow 0 \rightarrow 0 \\
1 & 2 & 4 \\
2 & 4 & 2 \\
3 & 6 & 6 \\
4 & 1 \rightarrow 1 \\
5 & 3 & 5 \\
6 & 5 & 3 \\
7 & 7 & 7
\end{array}
$$

As another example, how many stages does a 64-point FFT have?

What index does g[37] in the 64-point FFT get mapped to? (Hint: g[37]=g[100101])

How many butterfly ops are there per stage in a 64-point FFT?

The Cooley-Tukey FFT algorithm

- We have been looking at the algorithm over the last few slides; next, we'll look at code that is typical for a software implementation
- When performed sequentially, the $O(n\log n)$ operations in the FFT require $O(n\log n)$ time steps.
- But since each butterfly op is independent of the other butterfly ops in the same stage, in theory, all $n/2$ ops per stage could be performed in parallel with a task interval of 1
  - In practice, this is rarely done, except for small sample sizes – consider that a 1024-point complex FFT using floats running at 250MHz would require 1024*4*2*250*10^6 = 2TB/s data throughput & considerable dynamic power! We have to consider throughput requirements and resource availability…
  - How many butterfly ops need to be performed per sec assuming a 1024-point FFT at 250MHz with a task interval of 1?

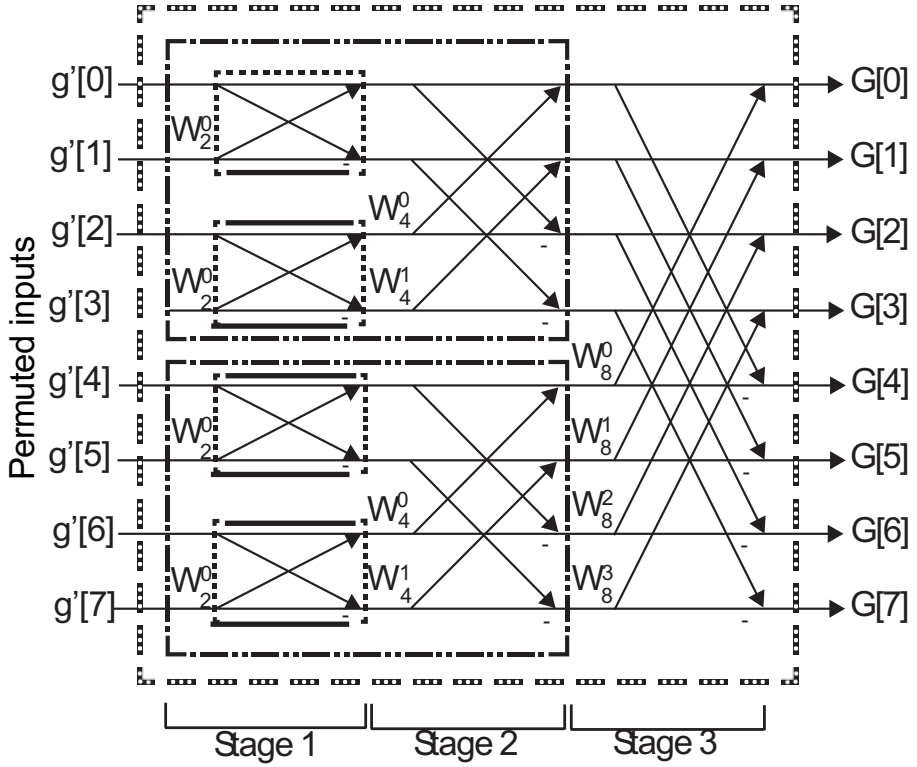
SW implementation of FFT

```
void fft(DTYPE X_R[SIZE], DTYPE X_I[SIZE]) {
    DTYPE temp_R; // temporary storage complex variable
    DTYPE temp_I; // temporary storage complex variable
    int i, j, k;  // loop indexes
    int i_lower;  // Index of lower point in butterfly
    int step, stage, DFTpts;
    int numBF;    // Butterfly Width
    int N2 = SIZE2; // N2=N/1

    bit_reverse(X_R, X_I);

    step = N2;
    DTYPE a, e, c, s;

stage_loop: // Do M = log2(SIZE) stages of butterflies

    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;
```

```
butterfly_loop: // Perform butterflies for j-th stage
                // Perf b'flies for one sub-DFT at this stage

        for (j = 0; j < numBF; j++) {
            c = cos(a);
            s = sin(a);
            a = a + e;

dft_loop: // Compute butterflies that use same W**k
          // Comp b'flies for all sub-DFTs at this stage

            for (i = j; i < SIZE; i += DFTpts) {
                i_lower = i + numBF; // index of lower point
                                     // in butterfly (closer
                                     // to bottom in diagram)
                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; // declarations & statements in red
                       // are redundant
        }
        step = step / 2;
    }
}
```

Nested loop in FFT SW

```
stage_loop: // Do M = log2(SIZE) stages of butterflies

    for (stage = 1; stage <= M; stage++) {
        DFTpts = 1 << stage; // DFT = 2^stage = points in
                             // sub DFT
        numBF = DFTpts / 2;  // Butterfly WIDTHS in sub-DFT
        e = -6.283185307178 / DFTpts;
        a = 0.0;
butterfly_loop: // Perform butterflies for j-th stage
                // Perf b'flies for one sub-DFT at this stage

        for (j = 0; j < numBF; j++) {
            c = cos(a);
            s = sin(a);
            a = a + e;

dft_loop: // Compute butterflies that use same W**k
          // Comp b'flies for all sub-DFTs at this stage

            for (i = j; i < SIZE; i += DFTpts) {
                i_lower = i + numBF; // index of lower point
                                     // in butterfly (closer
                                     // to bottom in diagram)
                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;
            }
        }
    }
```



SW implementation of `bit_reverse`

```
#include "math.h"
#include "fft.h"

// reverse the bits of input integer
unsigned int reverse_bits(unsigned int input) {
    int i, rev = 0;
    for (i = 0; i < M; i++) {
        rev = (rev << 1) | (input & 1);
        input = input >> 1;
    }
    return rev;
}
```

```
// swap each input vector entry with the entry having
// the reversed entry index
void bit_reverse(DTYPE X_R[SIZE], DTYPE X_I[SIZE]) {
    unsigned int reversed;
    unsigned int i;
    DTYPE temp;

    for (i = 0; i < SIZE; i++) {
        reversed = reverse_bits(i); // Find the bit-
                                    // reversed index
        if (i <= reversed) {
            // Swap the real values
            temp = X_R[i];
            X_R[i] = X_R[reversed];
            X_R[reversed] = temp;

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

Task pipelining the FFT algorithm

- The *N*-point FFT algorithm can be divided into $\log_2(N) + 1$ stages
  - The first stage swaps the elements of the input array to it's bit-reversed index postn
  - Followed by $\log_2(N)$ stages of butterfly ops
  - Each stage is described as an independent task with all stages being interlinked in a pipeline
- These stages can be executing concurrently on different data sets
- Such a hardware optimization, which is known as *task pipelining*, is relevant to many applications

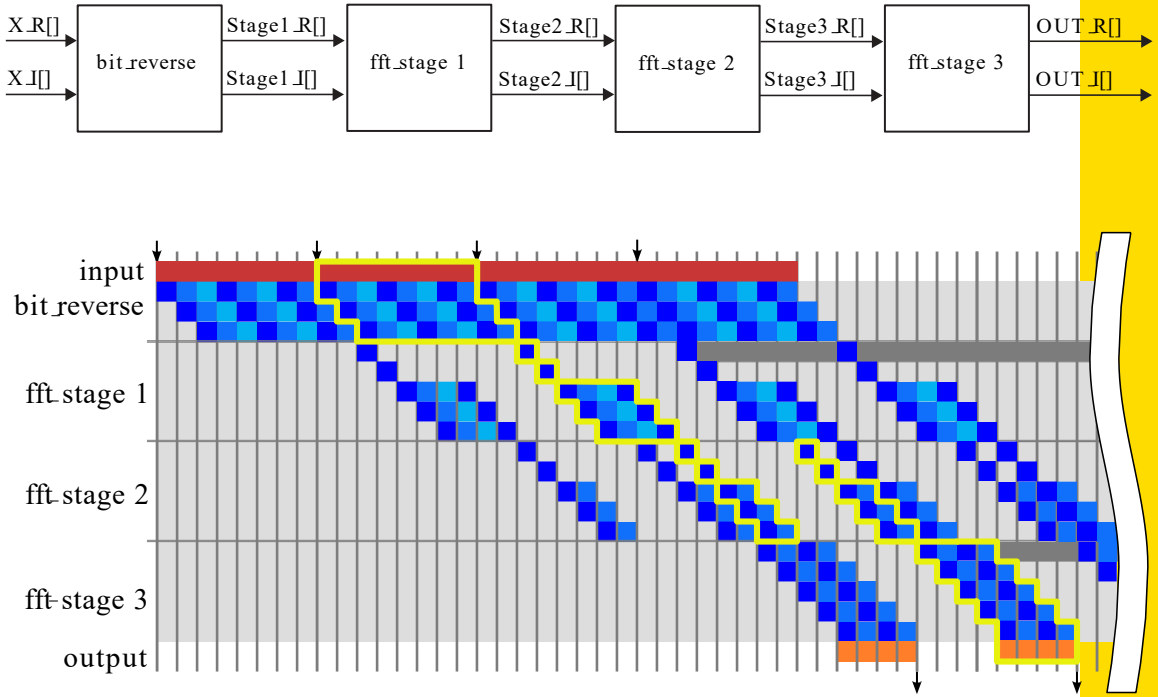
Restructured code to enable task pipelining

```
void fft_stage(int stage, DTYPE X_R[SIZE], DTYPE X_I[SIZE],
               DTYPE Out_R[SIZE], DTYPE Out_I[SIZE])
{
  int DFTpts = 1 << stage; // points in sub-DFT
  int numBF = DFTpts / 2;  // Butterfly WIDTHS in sub-DFT
  DTYPE e = -6.283185307178 / DFTpts;
  DTYPE a = 0.0;

  // Perform butterflies for j-th stage
butterfly_loop:
  for (int j = 0; j < numBF; j++) {
    DTYPE c = cos(a);
    DTYPE s = sin(a);
    a = a + e;

    // Compute butterflies that use same W**k
dft_loop:
    for (int i = j; i < SIZE; i += DFTpts) {
      int i_lower = i + numBF; // idx of lower point in bfly
      DTYPE temp_R = X_R[i_lower] * c - X_I[i_lower] * s;
      DTYPE temp_I = X_I[i_lower] * c + X_R[i_lower] * s;
      Out_R[i_lower] = X_R[i] - temp_R;
      Out_I[i_lower] = X_I[i] - temp_I;
      Out_R[i] = X_R[i] + temp_R;
      Out_I[i] = X_I[i] + temp_I;
    }
  }
}
```

```
void fft_streaming(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],
    Stage2_R[SIZE], Stage2_I[SIZE],
    Stage3_R[SIZE], Stage3_I[SIZE];

  bit_reverse(X_R, X_I, Stage1_R, Stage1_I);
  fft_stage(1, Stage1_R, Stage1_I, Stage2_R, Stage2_I);
  fft_stage(2, Stage2_R, Stage2_I, Stage3_R, Stage3_I);
  fft_stage(3, Stage3_R, Stage3_I, OUT_R, OUT_I);
}
```

Executing the task pipeline

- Rather than waiting for the first task (8-point data sample) to complete all four function calls, the second task commences after the first task has only finished the first function
- The first task continues to execute each stage in the pipeline in order, followed by the remaining tasks in order
- Once the pipeline is full all four subfunctions are executing concurrently, but each operates on different input data
- For this to work, each call to `fft_stage` must be implemented with independent hardware AND sufficient storage is needed to contain the intermediate computations of each stage



Applying `dataflow` at the loop level

- The `dataflow` directive can construct separate pipeline stages, or *processes*, from both functions and loops
- For example, the original code could have been pipelined by unrolling the `stage_loop`
  - This approach has the benefit of preserving the structure of the original code and leaving it parameterized so that it can accept differently sized FFTs

```
void fft_streaming(DTYPE X_R[SIZE], DTYPE X_I[SIZE],
                   DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE])
{
  #pragma HLS dataflow
  DTYPE Stage_R[M][SIZE], Stage_I[M][SIZE];
  #pragma HLS array_partition variable=Stage_R
                                dim=1 complete
  #pragma HLS array_partition variable=Stage_I
                                dim=1 complete

  bit_reverse(X_R, X_I, Stage_R[0], Stage_I[0]);

stage_loop: // Do M-1 stages of butterflies
  for (int stage = 1; stage < M; stage++) {
    #pragma HLS unroll
    fft_stage(stage, Stage_R[stage-1],
                     Stage_I[stage-1],
                     Stage_R[stage],
                     Stage_I[stage]);
  }

  fft_stage(M, Stage_R[M-1], Stage_I[M-1],
               OUT_R, OUT_I);
}
```

`dataflow` vs `pipeline`

- The **dataflow** directive and the **pipeline** directive both generate circuits capable of pipelined execution. The key difference is in the granularity of the pipelines.
- The **pipeline** directive constructs an architecture that is efficiently pipelined at the cycle level and is characterized by the II of the pipeline. Operators are statically scheduled.
- The **dataflow** directive constructs an architecture that is efficiently pipelined for operations that take a (possibly unknown) number of clock cycles, such as the behavior of a loop operating on a block of data. These coarse-grained operations are not statically scheduled – their behavior is controlled dynamically by the handshake of data through the pipeline.
- In the case of the FFT, it makes sense to use the **dataflow** directive at the top level to form a coarse-grained pipeline, combined with the **pipeline** directive within each loop to form fine-grained pipelines of the operations on each individual data element.

Optimizing dataflow processes

- Using the **dataflow** directive to full effect requires that the behavior of each individual process in the pipeline be optimized.
- Each process in the pipeline can be optimized using the techniques we have seen previously, such as code restructuring, pipelining, and unrolling.
  - Start with small functions, since it is easier to comprehend what is going on and to determine the best optimization strategy.
- In general, it is important to optimize the individual tasks while considering overall top level performance goals.
  - After optimizing individual functions, move up the hierarchy considering higher level functions given the particular implementations chosen for the lower level functions.
- For dataflow designs, it should be understood that the interval achieved for the overall pipeline can never be smaller than the largest interval of any individual process. Hence it's desirable to minimize interval differences across the pipeline rather than to aggressively optimize each function
  - The resource usage of some processes may thus benefit from being slowed down.

Task pipeline buffers

- The **dataflow** directive must implement memories to pass data between processes
- When Vivado HLS can determine that processes access the data sequentially, it implements *FIFOs* between the processes, which can considerably conserve resources
- When the accesses are random, or if the tool can't determine that the access pattern is sequential, it implements *ping-pong buffers* instead so that the producer can be writing to one buffer while the consumer is reading from the other

Concluding remarks

- “The overall goal is to create the most optimal design, which is a function of your application needs. This may be to create the smallest implementation. Or the goal could be creating something that can perform the highest throughput regardless of the size of the FPGA or the power/energy constraints. Or the latency of delivering the results may matter if the application has real-time constraints. All of the optimizations change these factors in different ways.”
- “In general, there is no one algorithm on how to optimize your design. It is a complex function of the application, design constraints, [target technology & tools] and the inherent abilities of the designer… it is important that the designer have a deep understanding of the application, the design constraints, and the abilities of the synthesis tool.”
- “The designer could translate C/MATLAB/Java/Python code into Vivado HLS and get a working implementation. And that same designer could somewhat blindly apply directives to achieve better results. But that designer is not going to get anywhere close to optimal results without a deep understanding of the algorithm.”