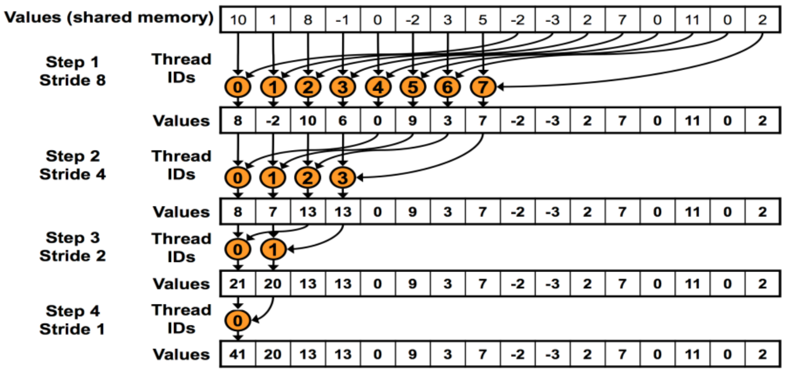
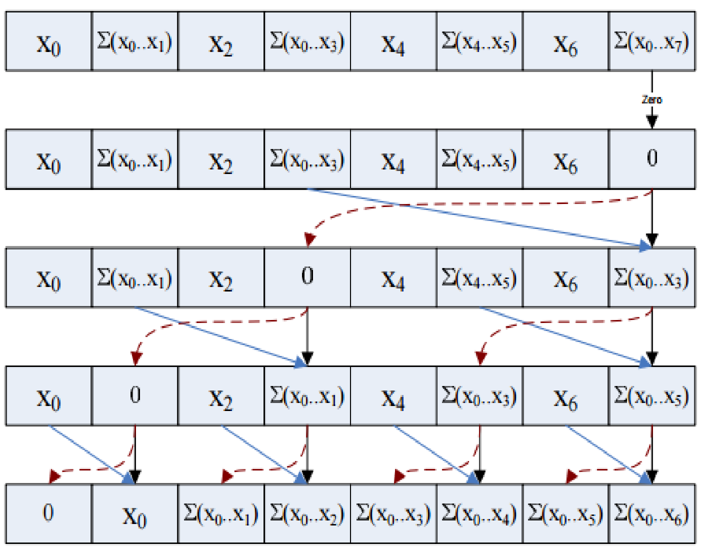
CS 179: GPU Programming

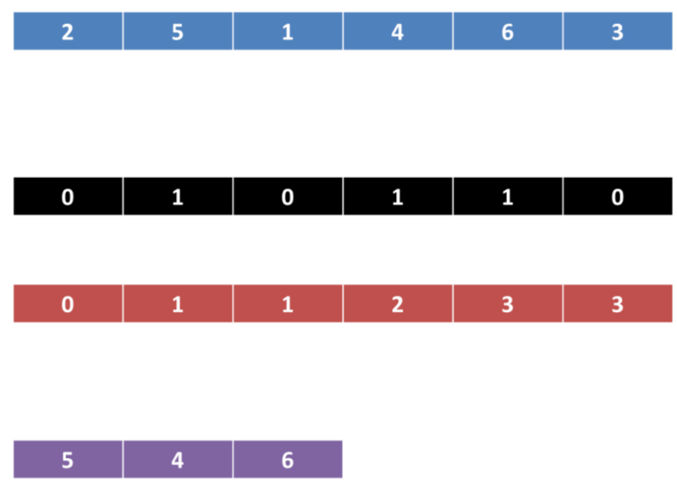
Lecture 8

Last time





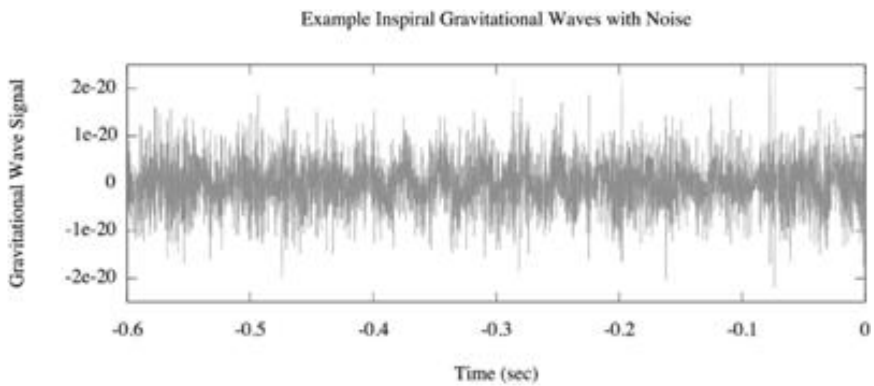
- GPU-accelerated:
  - Reduction
  - Prefix sum
  - Stream compaction
  - Sorting (quicksort)

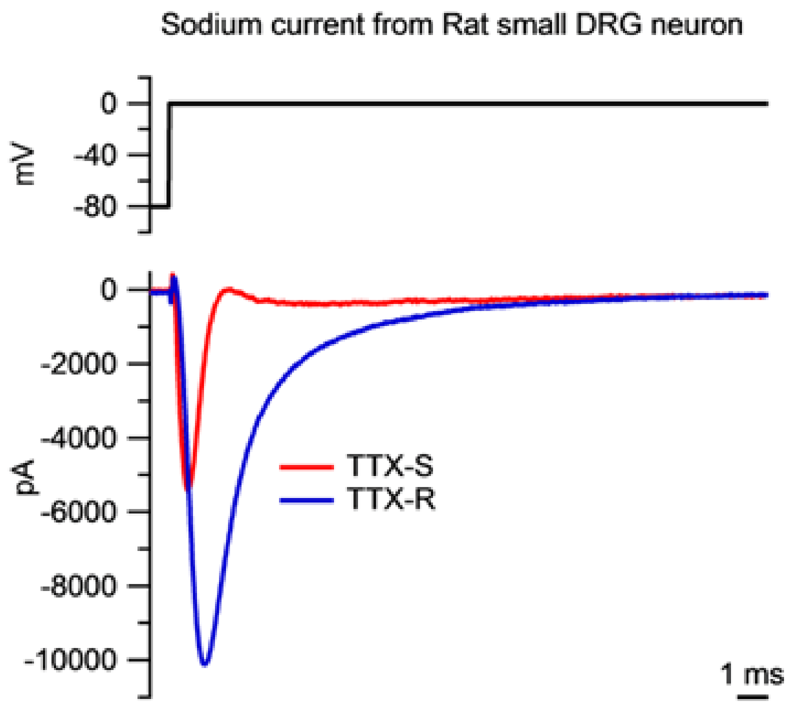


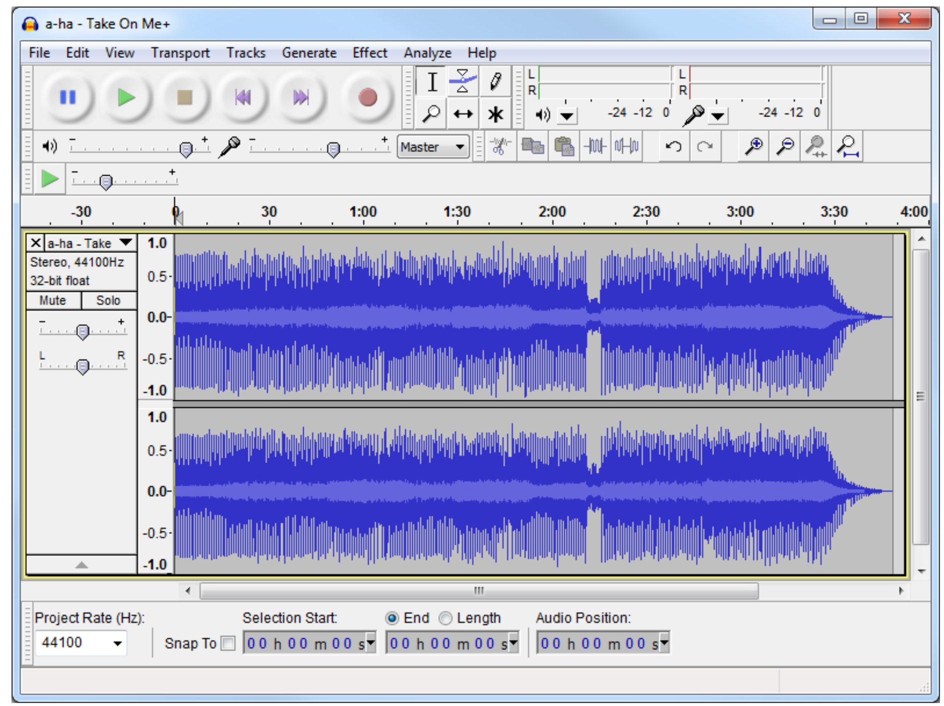
Today

- GPU-accelerated Fast Fourier Transform
- **cuFFT (FFT library)**
- This lecture – details behind FFT algorithm. Shows why you shouldn't re-invent the wheel!
  - Don't implement what a library already does for you, if you don't have to!
- It's not TOO critical for the HW for many of the details about this math – mostly background.
- We will use this FFT in the final weeks of lecture before projects, for implementing CONVOLUTIONAL NETWORKS on GPUs.

Signals (again)







"Frequency content"

- FT answers question: "What frequencies are present in our signals?"
- Key to field of Digital Signal Processing -- see https://en.wikipedia.org/wiki/Digital_signal_processing
- Time domain – higher frequencies are higher pitch
- Spatial domain – works similarly, but with "x" instead of "t"

Eqns for Continuous Fourier Transform

- Fourier Transform (one of several formulations)
  - See https://en.wikipedia.org/wiki/Fourier_transform
- $$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i x \xi}\, dx, \quad \textbf{(Eq.1)}$$
- Starts with input continuous function f(x) where x is in the spatial domain or f(t) for time domain,
  - to output continuous function in frequency domain, ω for time frequency, or ξ for spatial frequency
- Integral is "similar" to a matrix multiply, where integrand has two variables like 2D matrix, and x is like row in matrix, and column in f(x), and the integral is like the sum.

Discrete Fourier Transform (DFT)

- Main DFT Formulation:
  - Converts Time Domain input, (little) x_n to
  - Frequency domain Output, (big) X_k

$$X_k \stackrel{\text{def}}{=} \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i k n / N}, \quad k \in \mathbb{Z}$$

  - X_k - values corresponding to wave k
    - Periodic – calculate for $0 \leq k \leq N - 1$
- Similar to continuous defn, where we can see the 2 D matrix in the exponential, times the column vector (little) x_n . where k is wave number.

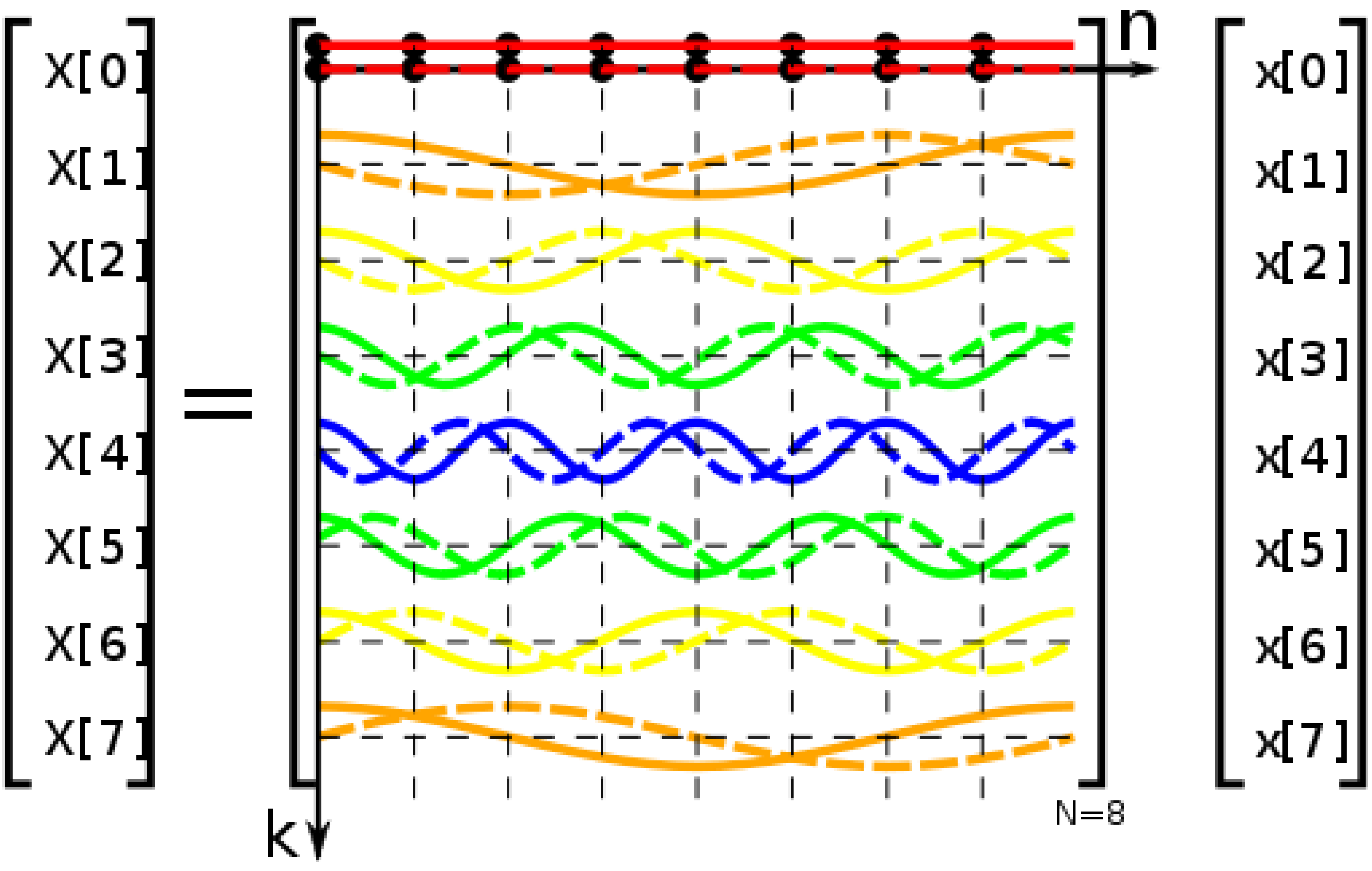
Discrete Fourier Transform (DFT)

$$W = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix},$$

$$\omega = e^{-2\pi i/N}$$

- Given signal $\vec{x} = (x_1, \ldots, x_N)$ over time, $\vec{y} = W\vec{x}$ represents DFT of $\vec{x}$
  - Each row of *W* is a complex sine wave
  - Each row multiplied with $\vec{x}$ - inner product of wave with signal
  - Corresponding entries of $\vec{y}$ - “content” of that sine wave!

Converting (little) x_n to (big) X_k



Solid line = real part

Dashed line = imaginary part

Discrete Fourier Transform (DFT)

- Alternative formulation:

$$X_k \stackrel{\text{def}}{=} \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i k n/N}, \quad k \in \mathbb{Z}$$

  – X_k - values corresponding to wave *k*
    - Periodic – calculate for $0 \le k \le N - 1$

  – Naive runtime: $O(N^2)$
    - Sum of *N* iterations, for *N* values of *k*

Discrete Fourier Transform (DFT)

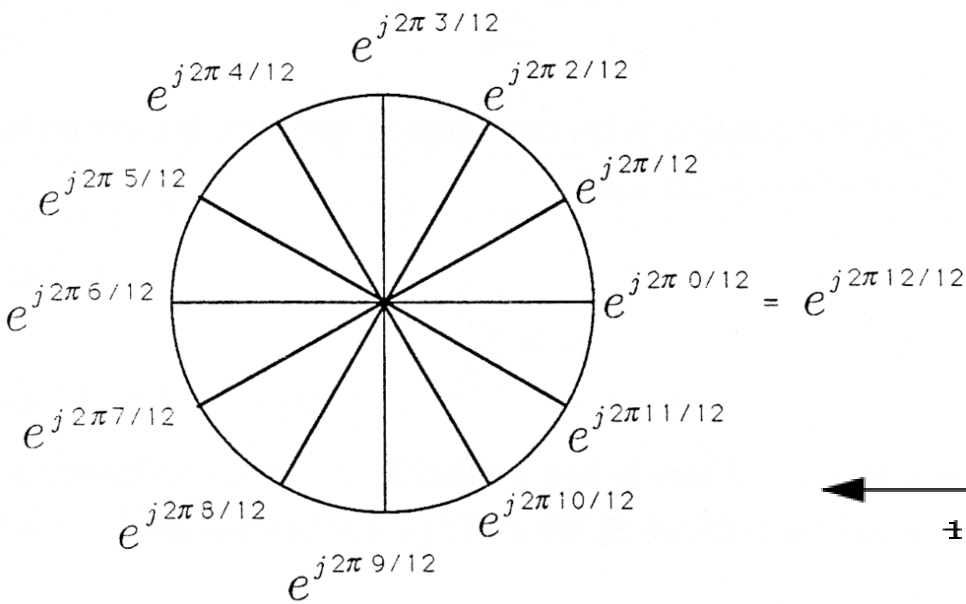
- Alternative formulation:

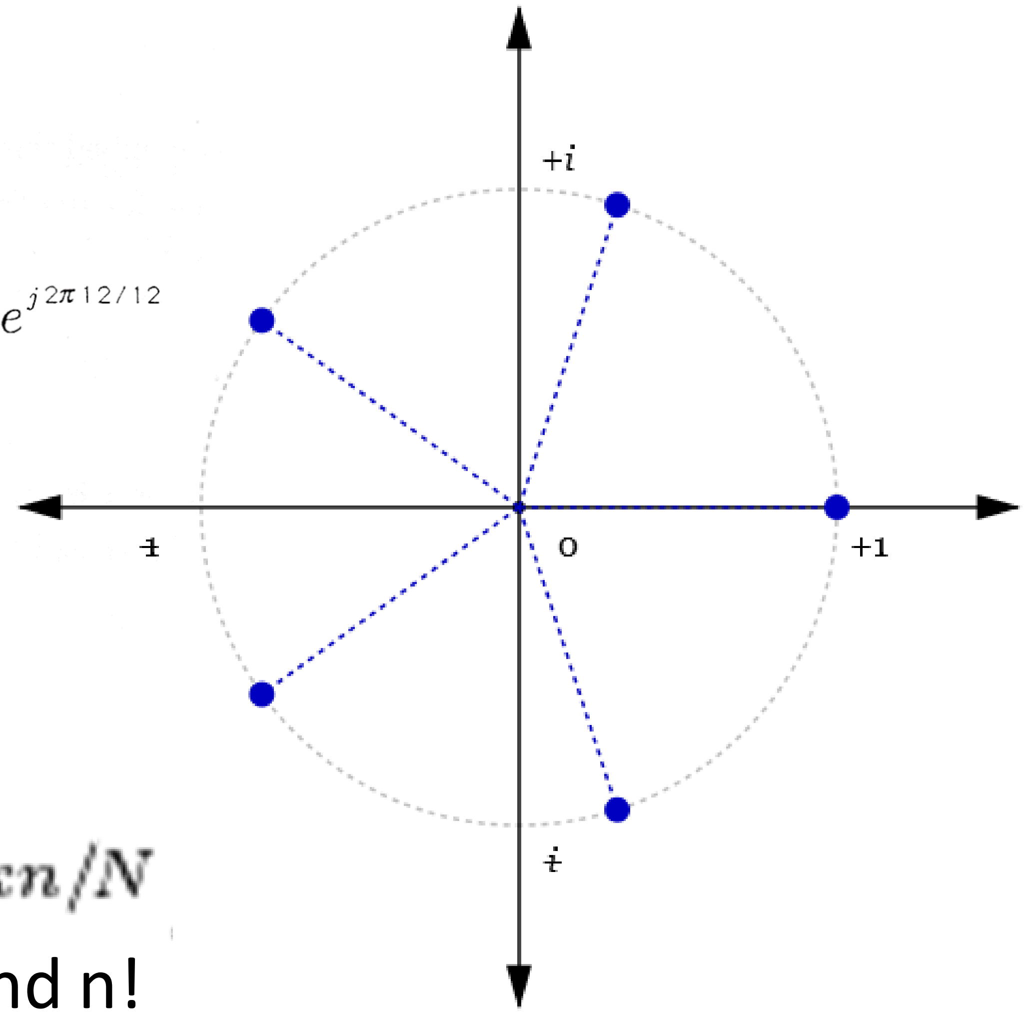
$$X_k \stackrel{\text{def}}{=} \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i k n / N}, \quad k \in \mathbb{Z}$$

  – X_k - values corresponding to wave *k*
    - Periodic – calculate for $0 \le k \le N - 1$

  – Naive runtime: $O(N^2)$
    - Sum of *N* iterations, for *N* values of *k*

Roots of Unity on Complex Plane





Only N values of $e^{-2\pi ikn/N}$
not N^2 for all integers k and n!

Discrete Fourier Transform (DFT)

- Alternative formulation:

$$X_k \stackrel{\text{def}}{=} \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i k n / N}, \quad k \in \mathbb{Z}$$

  - X_k - values corresponding to wave *k*
    - Periodic – calculate for $0 \leq k \leq N - 1$

Number of distinct values: N, not N^2!

Re-expressing DFT, for Proof

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-2\pi i \, kn/N} = \sum_{n=0}^{N-1} x_n \, e^{-2\pi i \left(\frac{k\,n}{N}\right)}$$

$$= \sum_{n=0}^{N-1} x_n \left(e^{-2\pi i/N}\right)^{k\,n}$$

$$X_k = \sum_{n=0}^{N-1} x_n \left(\omega_N\right)^{k\,n}$$

where $\omega_N = e^{-2\pi i/N}$

(Proof -- -- breaks DFT into two DFTs, even/odd)

- Breakdown (assuming N is power of 2):
  - (Let $\omega_N = e^{-2\pi i/N}$, smallest root of unity)

$$\sum_{n=0}^{N-1} x_n {\omega_N}^{kn}$$

(Proof of Recursion)

- Breakdown (assuming N is power of 2):
  - (Let $\omega_N = e^{-2\pi i/N}$, smallest root of unity)

$$\sum_{n=0}^{N-1} x_n {\omega_N}^{kn}$$

$$= \sum_{n=0}^{N/2-1} x_{(2n)} {\omega_N}^{k(2n)} + \sum_{n=0}^{N/2-1} x_{(2n+1)} {\omega_N}^{k(2n+1)}$$

(Proof of Recursion)

- Breakdown (assuming N is power of 2):
  - (Let $\omega_N = e^{-2\pi i/N}$, smallest root of unity)

$$\sum_{n=0}^{N-1} x_n {\omega_N}^{kn}$$

$$= \sum_{n=0}^{N/2-1} x_{(2n)} {\omega_N}^{k(2n)} + \sum_{n=0}^{N/2-1} x_{(2n+1)} {\omega_N}^{k(2n+1)}$$

$$= \sum_{n=0}^{N/2-1} x_{(2n)} {\omega_N}^{k(2n)} + \omega_N \sum_{n=0}^{N/2-1} x_{(2n+1)} {\omega_N}^{k(2n)}$$

(Proof of Recursion)

- Breakdown (assuming N is power of 2):
  - (Let $\omega_N = e^{-2\pi i/N}$, smallest root of unity)

$$\sum_{n=0}^{N-1} x_n {\omega_N}^{kn}$$

$$= \sum_{n=0}^{N/2-1} x_{(2n)} {\omega_N}^{k(2n)} + \sum_{n=0}^{N/2-1} x_{(2n+1)} {\omega_N}^{k(2n+1)}$$

$$= \sum_{n=0}^{N/2-1} x_{(2n)} {\omega_N}^{k(2n)} + \omega_N \sum_{n=0}^{N/2-1} x_{(2n+1)} {\omega_N}^{k(2n)}$$

$$= \sum_{n=0}^{N/2-1} x_{(2n)} {\omega_{N/2}}^{kn} + \omega_N \sum_{n=0}^{N/2-1} x_{(2n+1)} {\omega_{N/2}}^{kn}$$

(Proof of Recursion)

- Breakdown (assuming N is power of 2):
  - (Let $\omega_N = e^{-2\pi i/N}$, smallest root of unity)

$$\sum_{n=0}^{N-1} x_n {\omega_N}^{kn}$$

$$= \sum_{n=0}^{N/2-1} x_{(2n)} {\omega_N}^{k(2n)} + \sum_{n=0}^{N/2-1} x_{(2n+1)} {\omega_N}^{k(2n+1)}$$

$$= \sum_{n=0}^{N/2-1} x_{(2n)} {\omega_N}^{k(2n)} + \omega_N \sum_{n=0}^{N/2-1} x_{(2n+1)} {\omega_N}^{k(2n)}$$

$$= \underbrace{\sum_{n=0}^{N/2-1} x_{(2n)} {\omega_{N/2}}^{kn}}_{\text{DFT of } x_n\text{, even } n!} + \omega_N \underbrace{\sum_{n=0}^{N/2-1} x_{(2n+1)} {\omega_{N/2}}^{kn}}_{\text{DFT of } x_n\text{, odd } n!}$$

DFT of x_n, even n! DFT of x_n, odd n!

(Divide-and-conquer algorithm)

```
Recursive-FFT(Vector x):

    if x is length 1:
         return x

    x_even <- (x0, x2, ..., x_(n-2) )
    x_odd <- (x1, x3, ..., x_(n-1) )

    y_even <- Recursive-FFT(x_even)
    y_odd <- Recursive-FFT(x_odd)

    for k = 0, …, (n/2)-1:
        y[k]         <- y_even[k] + w^k * y_odd[k]
        y[k + n/2]   <- y_even[k] - w^k * y_odd[k]

    return y
```

(Divide-and-conquer algorithm)

```
Recursive-FFT(Vector x):

    if x is length 1:
         return x

    x_even <- (x0, x2, ..., x_(n-2) )
    x_odd <- (x1, x3, ..., x_(n-1) )

    y_even <- Recursive-FFT(x_even)
    y_odd <- Recursive-FFT(x_odd)

    for k = 0, …, (n/2)-1:
        y[k]        <- y_even[k] + w^k * y_odd[k]
        y[k + n/2]  <- y_even[k] - w^k * y_odd[k]

    return y
```

T(n/2)

T(n/2)

O(n)

Runtime

- Recurrence relation:
  - $T(n) = 2T(n/2) + O(n)$

O(n log n) runtime! *Much* better than $O(n^2)$

- (Minor caveat: N must be power of 2)
  - Usually resolvable

Parallelizable?

- O(n²) algorithm certainly is!

```
for k = 0,…,N-1:
    for n = 0,…,N-1:
        ...
```

$$X_k \stackrel{\text{def}}{=} \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i k n / N}$$

- Sometimes parallelization of “bad” algorithm can outweigh runtime for “better” algorithm!
  - (N-body problem, …)

Culey Tukey Recursive Algorithm for FFT

- See https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm
- Recursively re-expresses discrete Fourier transform (DFT) of an arb composite size

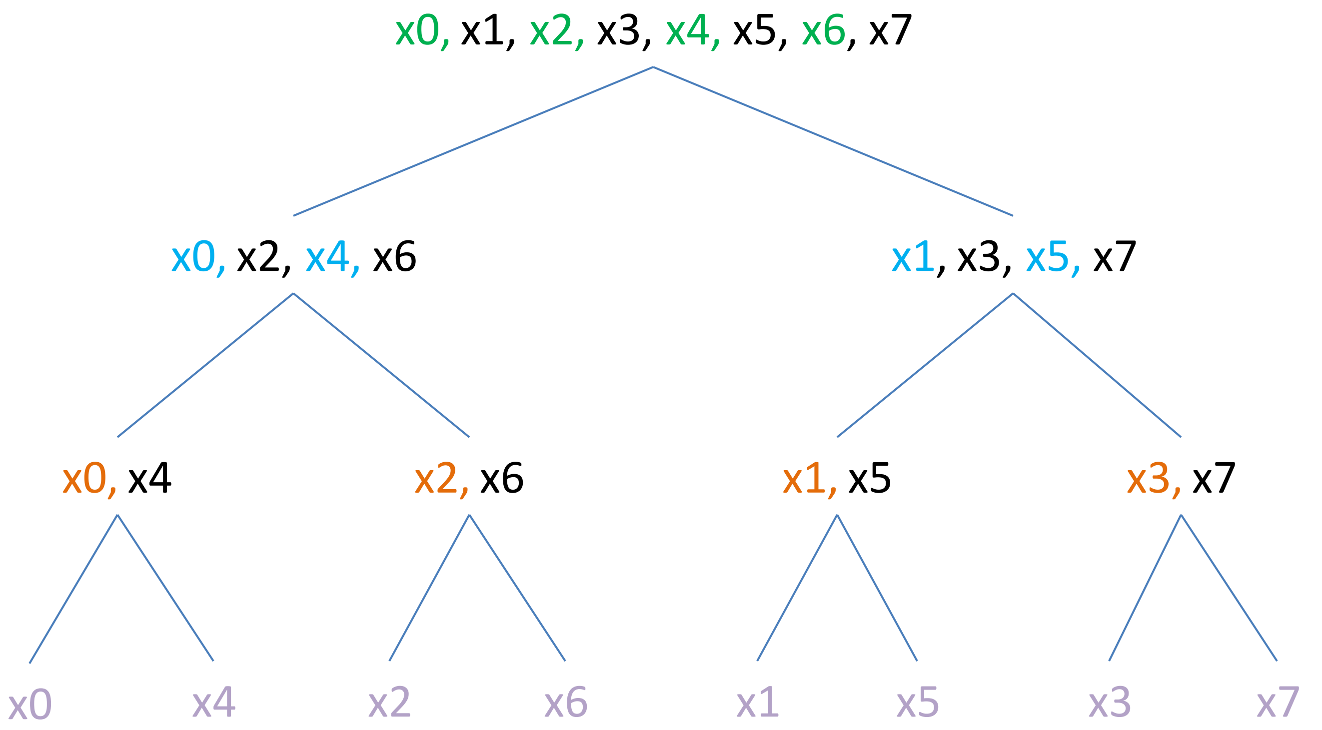
  $N = N_1 \ N_2$

  in terms of in terms of N1 smaller DFTs of sizes N2, recursively

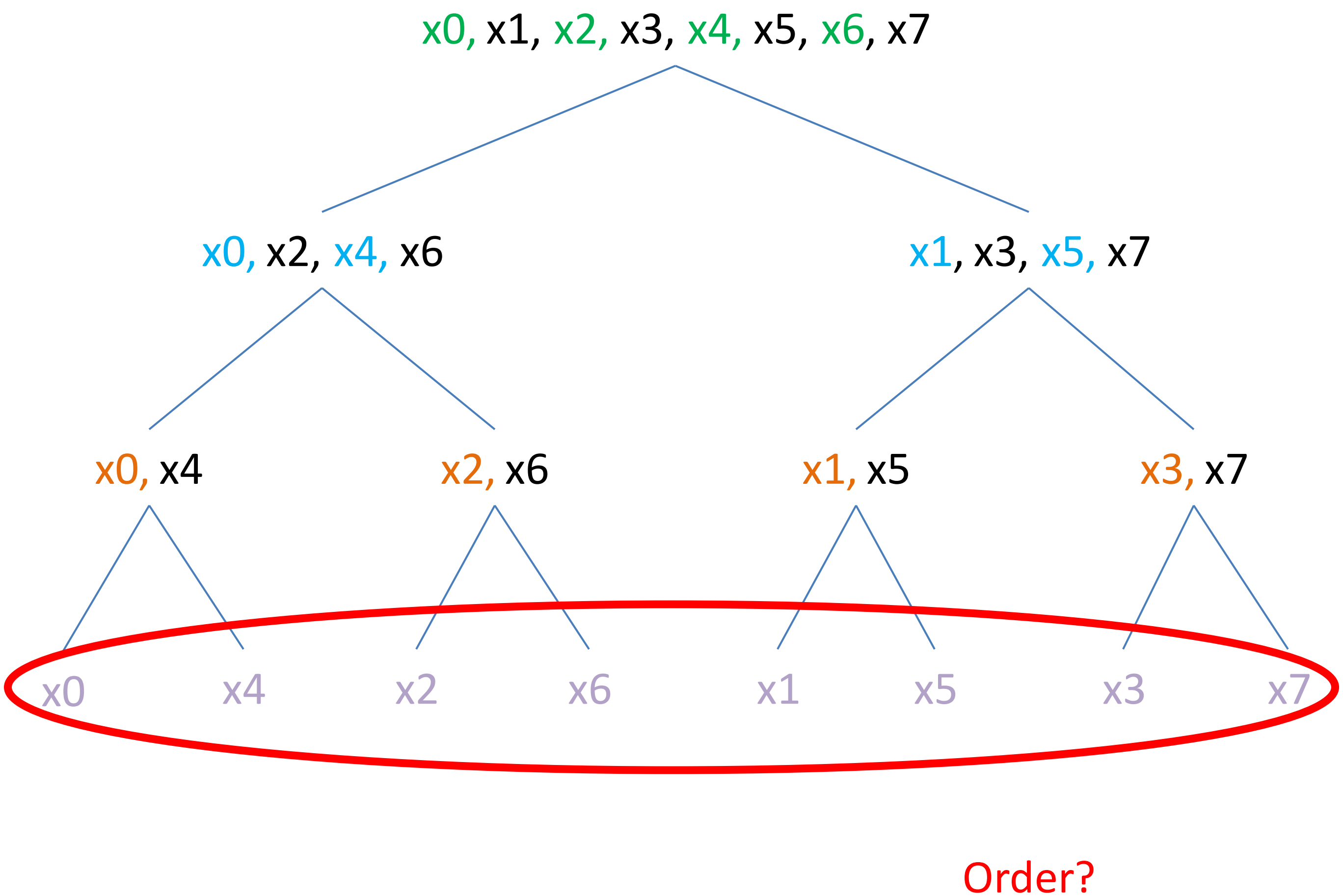
  Reduces computation time to O(N log N) for highly composite N (smooth numbers).

  Specific variants and implementation styles have become known by their own names

Recursive index tree



Recursive index tree



Bit-reversal order

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

Bit-reversal order

0	000	reverse of…	000	0
4	100		001	1
2	010		010	2
6	110		011	3
1	001		100	4
5	101		101	5
3	011		110	6
7	111		111	7

(Divide-and-conquer algorithm review)

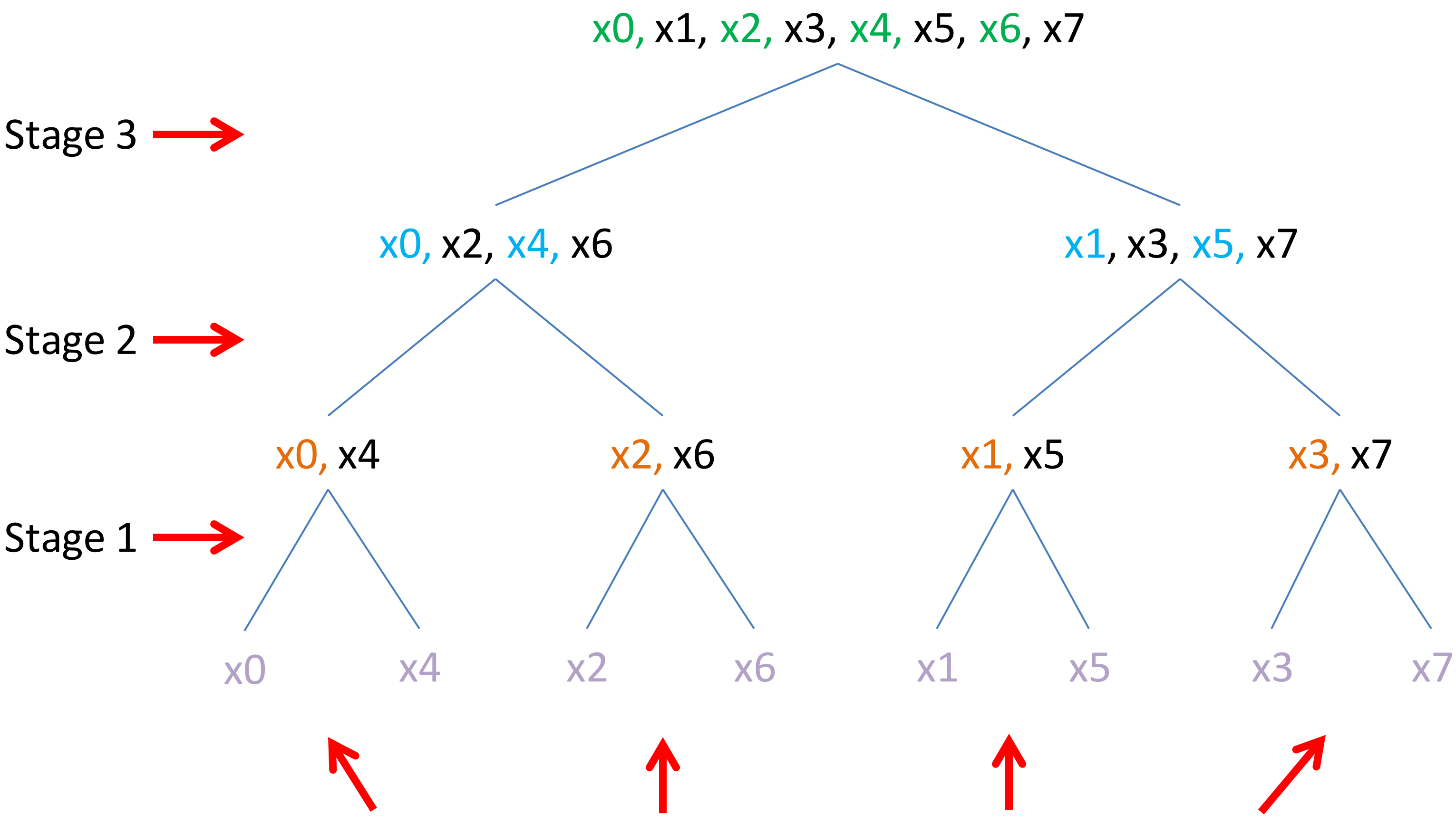
```
Recursive-FFT(Vector x):

    if x is length 1:
         return x

    x_even <- (x0, x2, ..., x_(n-2) )
    x_odd <- (x1, x3, ..., x_(n-1) )

    y_even <- Recursive-FFT(x_even)
    y_odd <- Recursive-FFT(x_odd)

    for k = 0, …, (n/2)-1:
        y[k]        <- y_even[k] + w^k * y_odd[k]
        y[k + n/2]  <- y_even[k] - w^k * y_odd[k]

    return y
```

T(n/2)

T(n/2)

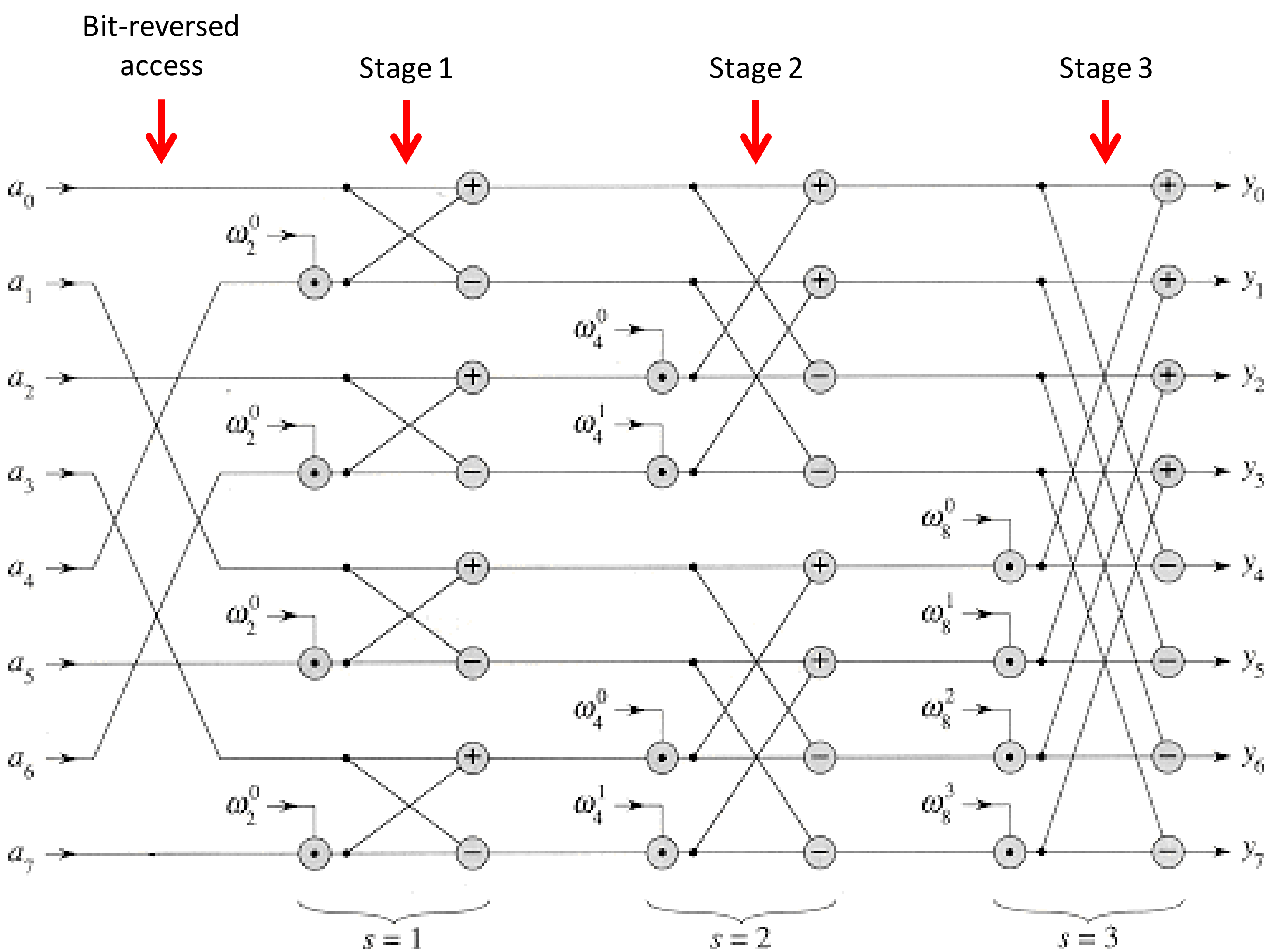
O(n)

Iterative approach



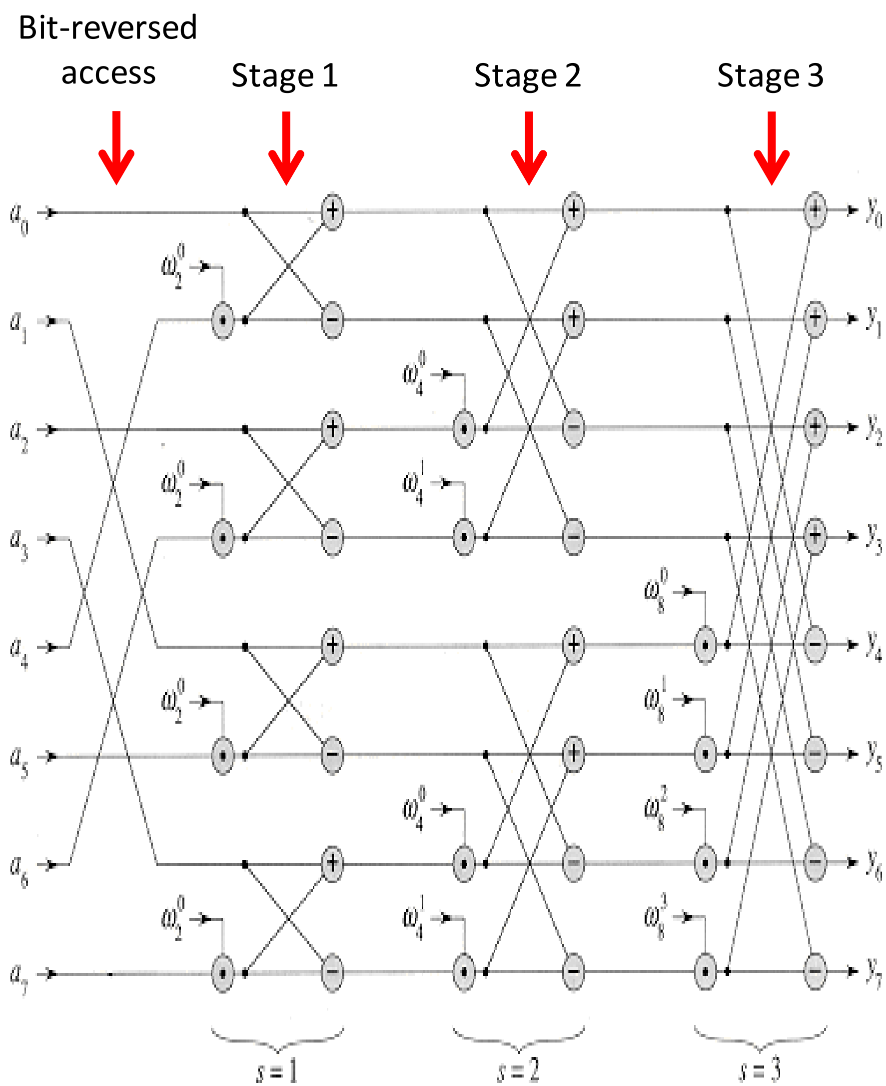
Bit-reversed accesses (a la sum reduction)

Iterative approach



http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap32.htm

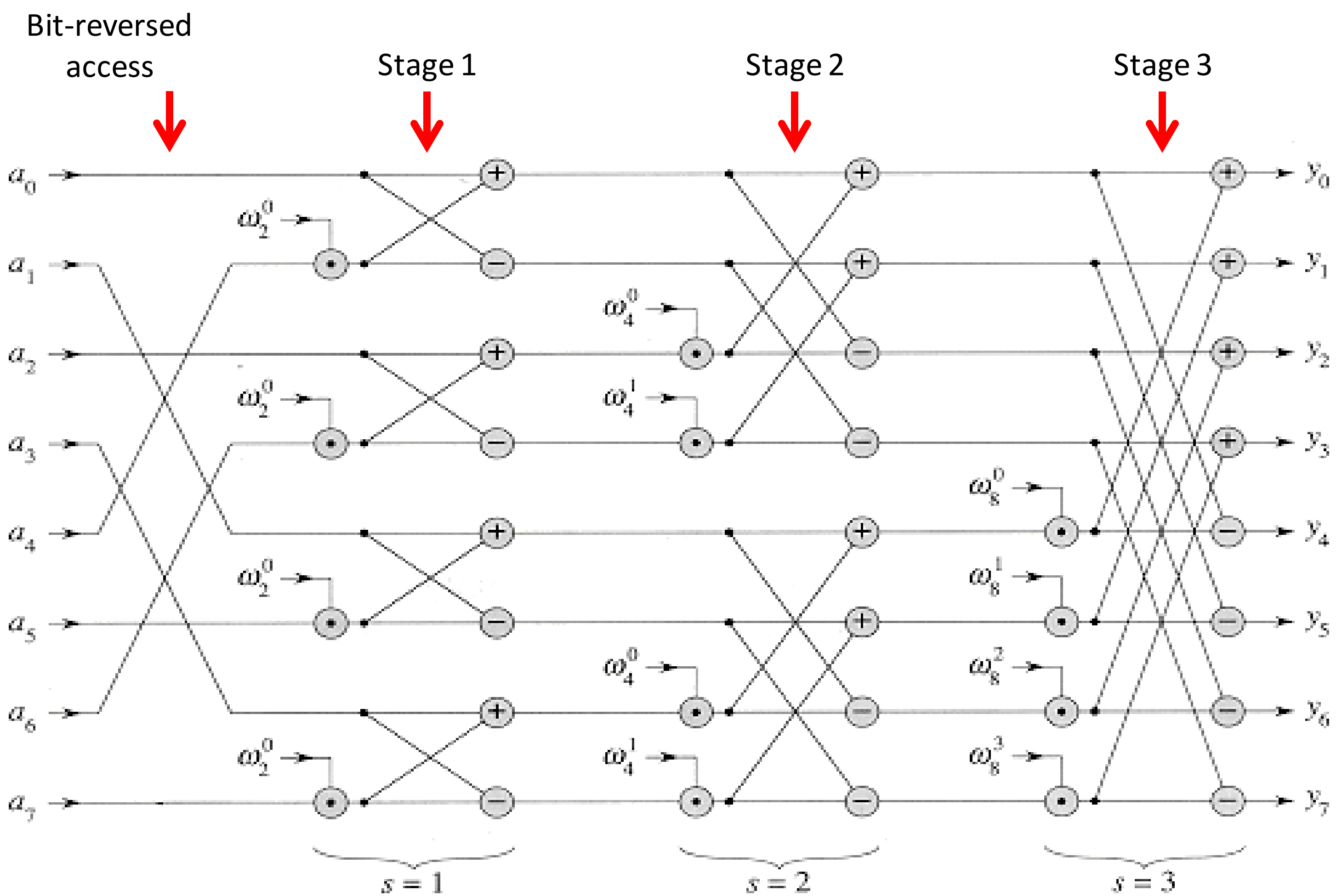
Iterative approach



http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap32.htm

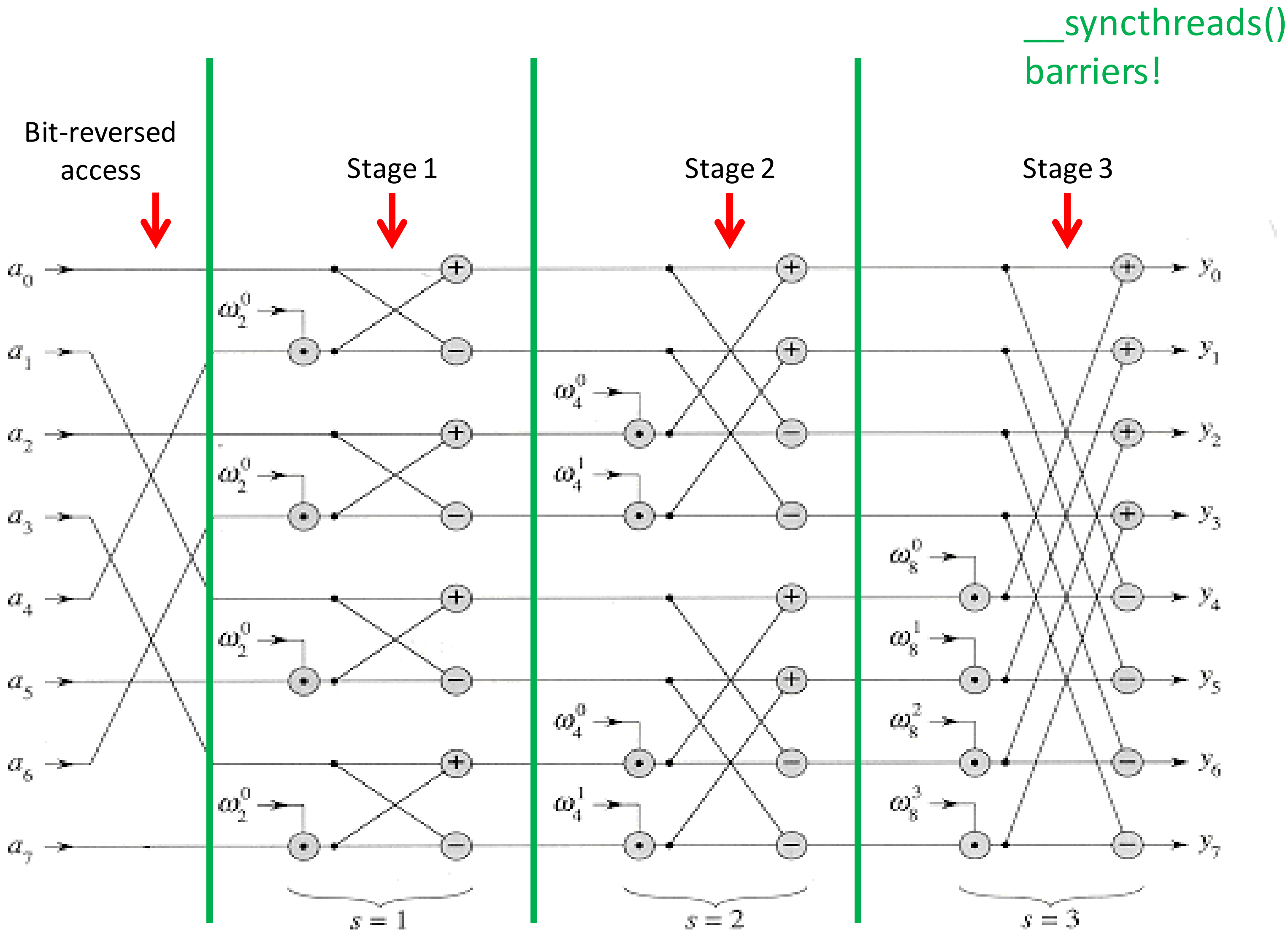
```
Iterative-FFT(Vector x):

    y <- (bit-reversed order x)
    N <- y.length
    for s = 1,2,…,lg(N):

        m <- 2^s
        w_n <- e^(2πj/m)

        for k: 0 ≤ k ≤ N-1, stride m:
            for j = 0,…,(m/2)-1:

                u <- y[k + j]
                t <- (w_n)^j * y[k + j + m/2]

                y[k + j] <- u + t
                y[k + j + m/2] <- u - t


    return y
```

CUDA approach



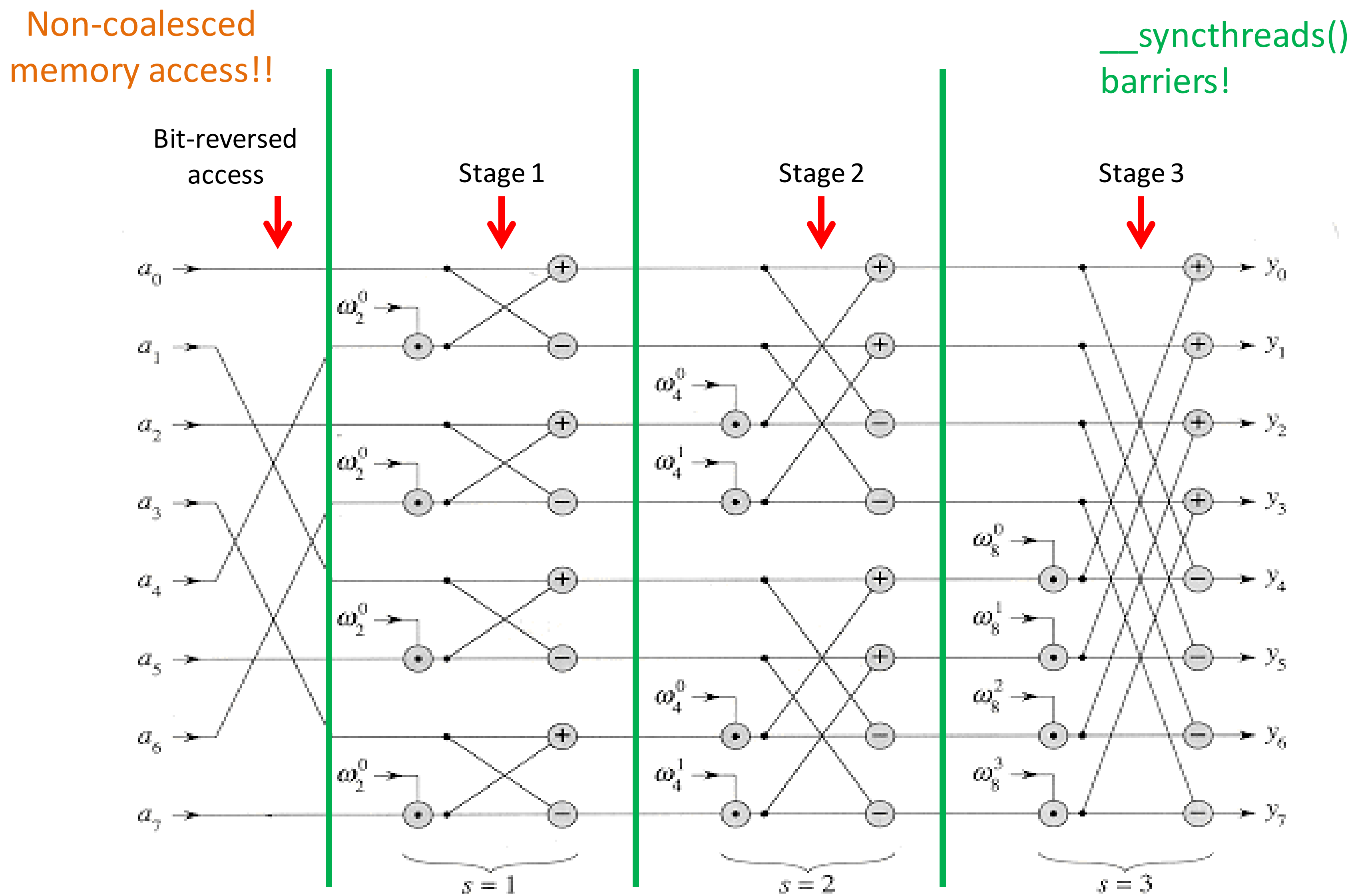
http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap32.htm

CUDA approach



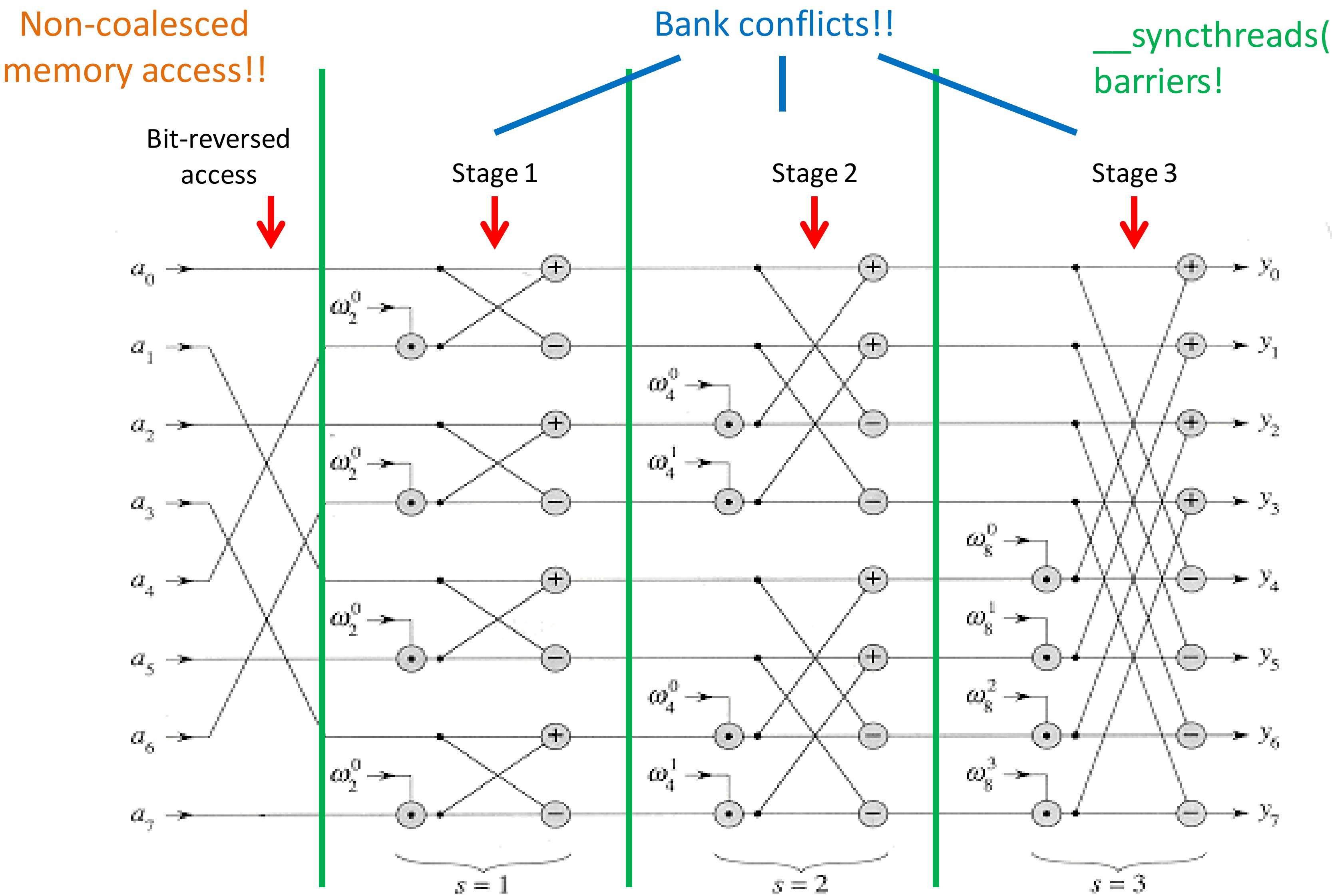
http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap32.htm

CUDA approach



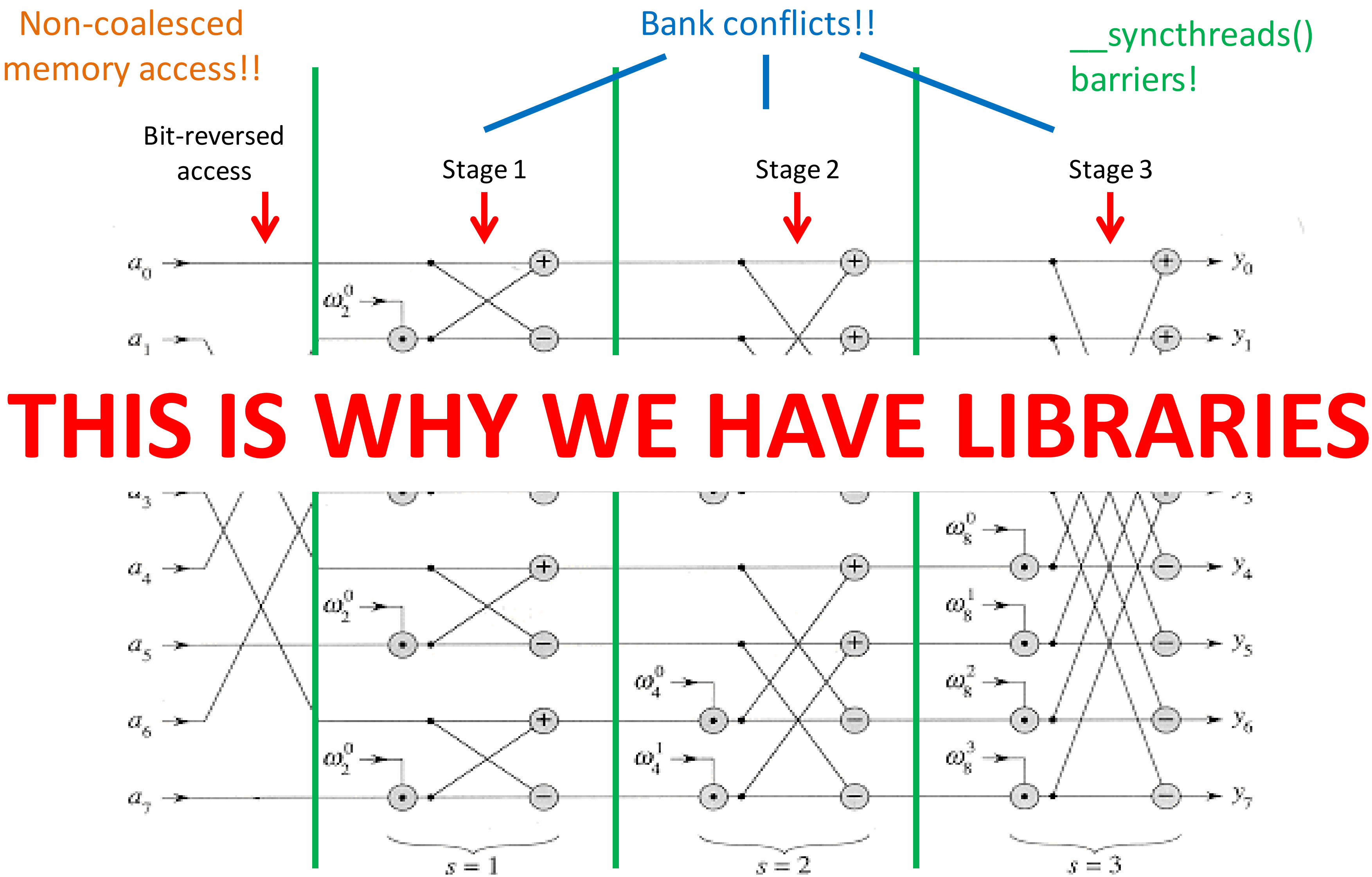
http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap32.htm

CUDA approach



http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap32.htm

CUDA approach



http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap32.htm

cuFFT

- FFT library included with CUDA
  - Approximately implements previous algorithms
    - (Cooley-Tukey/Bluestein)
    - Also handles higher dimensions
  - Handles nasty hardware constraints that you don't want to think about
- Also handles inverse FFT/DFT similarly
  - Just a sign change in complex terms

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi kn/N}, \quad n \in \mathbb{Z}$$

cuFFT 1D example

```
#define NX 262144

cufftComplex *data_host
        = (cufftComplex*)malloc(sizeof(cufftComplex)*NX);
cufftComplex *data_back
        = (cufftComplex*)malloc(sizeof(cufftComplex)*NX);


// Get data...


cufftHandle plan;
cufftComplex *data1;
cudaMalloc((void**)&data1, sizeof(cufftComplex)*NX);
cudaMemcpy(data1, data_host, NX*sizeof(cufftComplex), cudaMemcpyHostToDevice);

/* Create a 1D FFT plan. */
int batch = 1;  // Number of transforms to run
cufftPlan1d(&plan, NX, CUFFT_C2C, batch);

/* Transform the first signal in place. */
cufftExecC2C(plan, data1, data1, CUFFT_FORWARD);


/* Inverse transform in place. */
cufftExecC2C(plan, data1, data1, CUFFT_INVERSE);

cudaMemcpy(data_back, data1, NX*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
```

Correction: Remember to use cufftDestroy(plan) when finished with transforms

cuFFT 3D example

```
#define NX 64
#define NY 64
#define NZ 128

cufftComplex *data_host
        = (cufftComplex*)malloc(sizeof(cufftComplex)*NX*NY*NZ);
cufftComplex *data_back
        = (cufftComplex*)malloc(sizeof(cufftComplex)*NX*NY*NZ);


// Get data...


cufftHandle plan;
cufftComplex *data1;
cudaMalloc((void**)&data1, sizeof(cufftComplex)*NX*NY*NZ);
cudaMemcpy(data1, data_host, NX*NY*NZ*sizeof(cufftComplex), cudaMemcpyHostToDevice);

/* Create a 3D FFT plan. */
cufftPlan3d(&plan, NX, NY, NZ, CUFFT_C2C);

/* Transform the first signal in place. */
cufftExecC2C(plan, data1, data1, CUFFT_FORWARD);


/* Inverse transform in place. */
cufftExecC2C(plan, data1, data1, CUFFT_INVERSE);

cudaMemcpy(data_back, data1, NX*NY*NZ*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
```

Correction: Remember to use cufftDestroy(plan) when finished with transforms

Remarks

- As before, some parallelizable algorithms don’t easily “fit the mold”
  - Hardware matters more!

- Some resources:
  - Introduction to Algorithms (Cormen, et al), aka “CLRS”, esp. Sec 30.5
  - “An Efficient Implementation of Double Precision 1-D FFT for GPUs Using CUDA” (Liu, et al.)