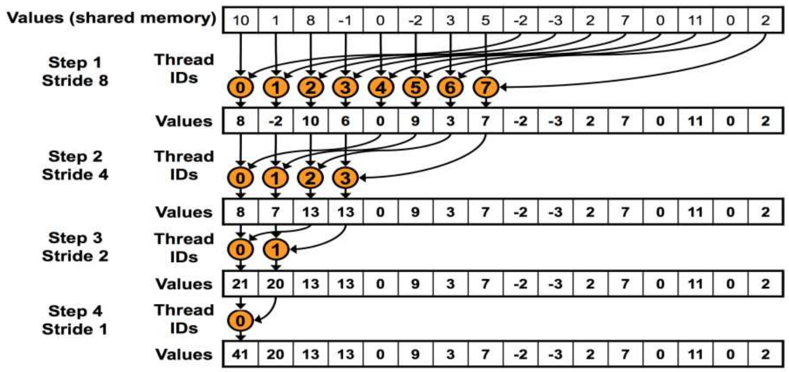
CS 179: GPU Programming

Lecture 8

Last time

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



Values (shared memory) 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2

Step 1 Stride 8 — Thread IDs 0 1 2 3 4 5 6 7

Values 8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 2 Stride 4 — Thread IDs 0 1 2 3

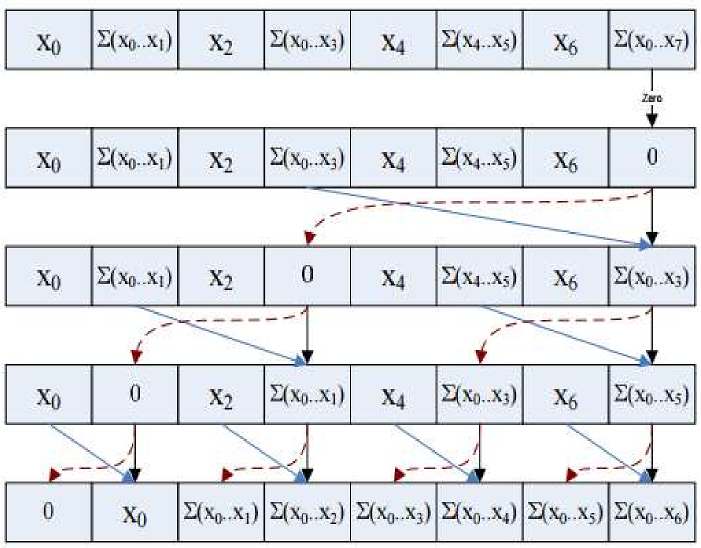
Values 8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 3 Stride 2 — Thread IDs 0 1

Values 21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 4 Stride 1 — Thread IDs 0

Values 41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2



x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_0..x_7)$

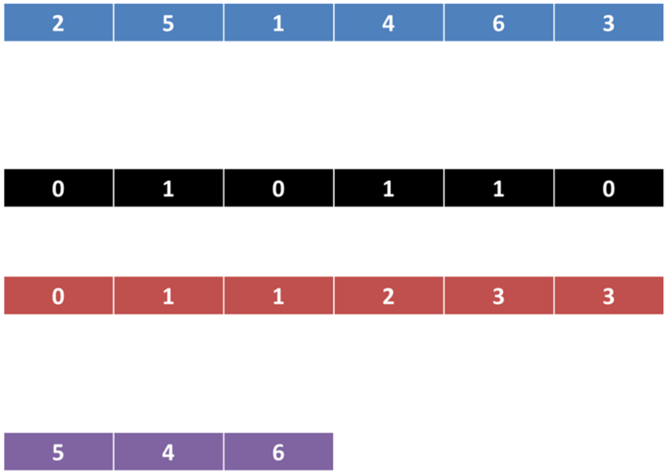
Zero

x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | 0

x_0 | $\Sigma(x_0..x_1)$ | x_2 | 0 | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_0..x_3)$

x_0 | 0 | x_2 | $\Sigma(x_0..x_1)$ | x_4 | $\Sigma(x_0..x_3)$ | x_6 | $\Sigma(x_0..x_5)$

0 | x_0 | $\Sigma(x_0..x_1)$ | $\Sigma(x_0..x_2)$ | $\Sigma(x_0..x_3)$ | $\Sigma(x_0..x_4)$ | $\Sigma(x_0..x_5)$ | $\Sigma(x_0..x_6)$



2 5 1 4 6 3

0 1 0 1 1 0

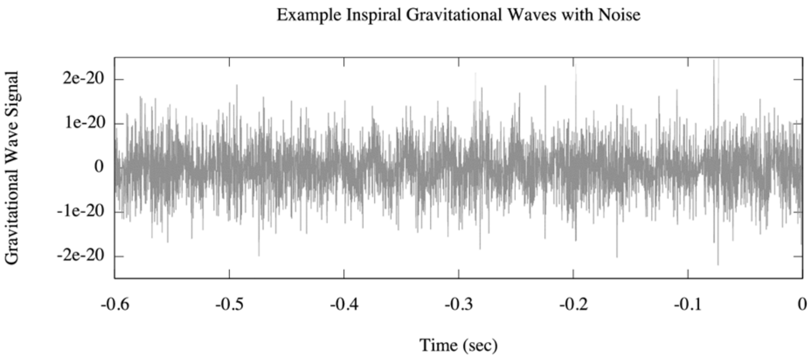
0 1 1 2 3 3

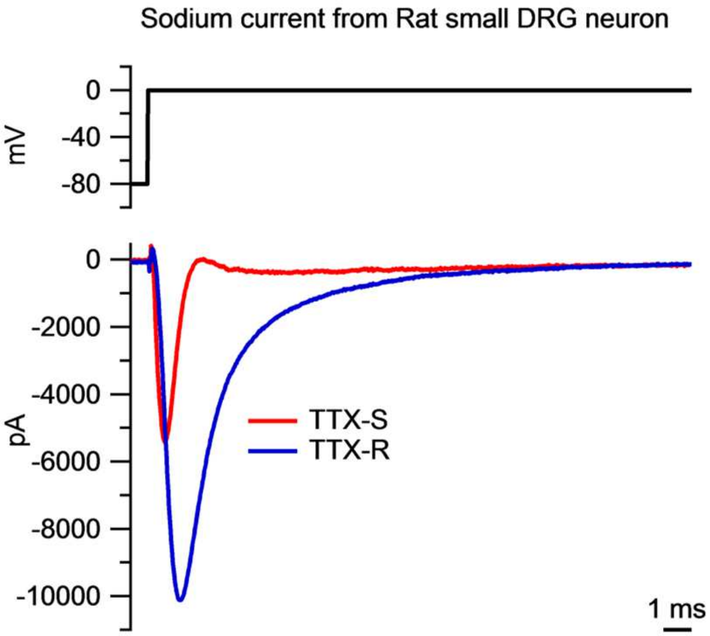
5 4 6

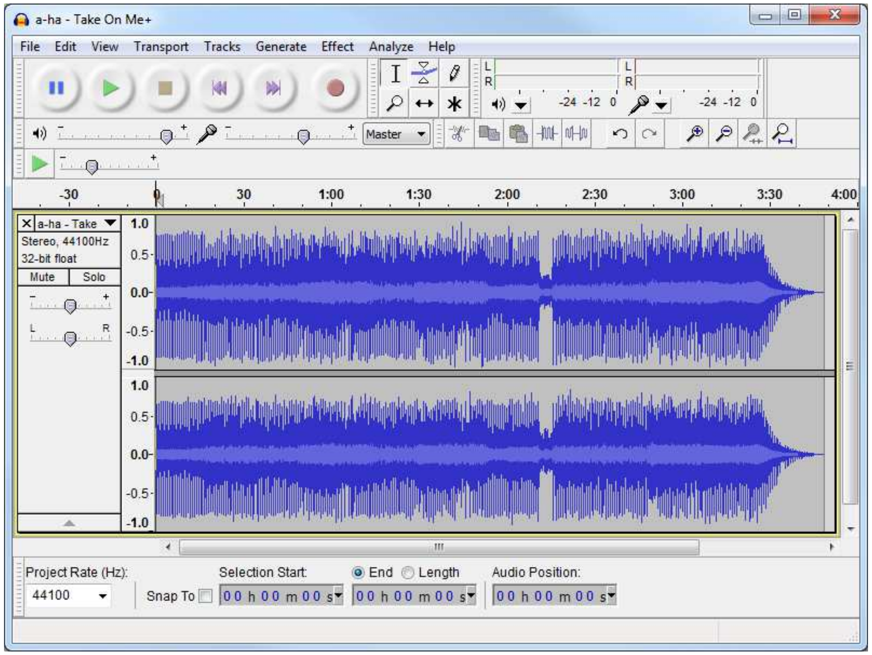
Today

- GPU-accelerated Fast Fourier Transform
- cuFFT (FFT library)

Signals (again)







Stereo, 44100Hz
32-bit float

Project Rate (Hz): 44100

Snap To □

Selection Start: 00 h 00 m 00 s

● End ○ Length 00 h 00 m 00 s

Audio Position: 00 h 00 m 00 s

“Frequency content”

- What frequencies are present in our signals?

Discrete Fourier Transform (DFT)

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

- 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!

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

Discrete Fourier Transform (DFT)

- Alternative formulation:

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

  - X_k - values corresponding to wave *k*
    - Periodic – calculate for 0 ≤ *k* ≤ *N* - 1

Discrete Fourier Transform (DFT)

- Alternative formulation:

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

  - X_k - values corresponding to wave *k*
    - Periodic – calculate for 0 ≤ *k* ≤ *N* - 1

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

Discrete Fourier Transform (DFT)

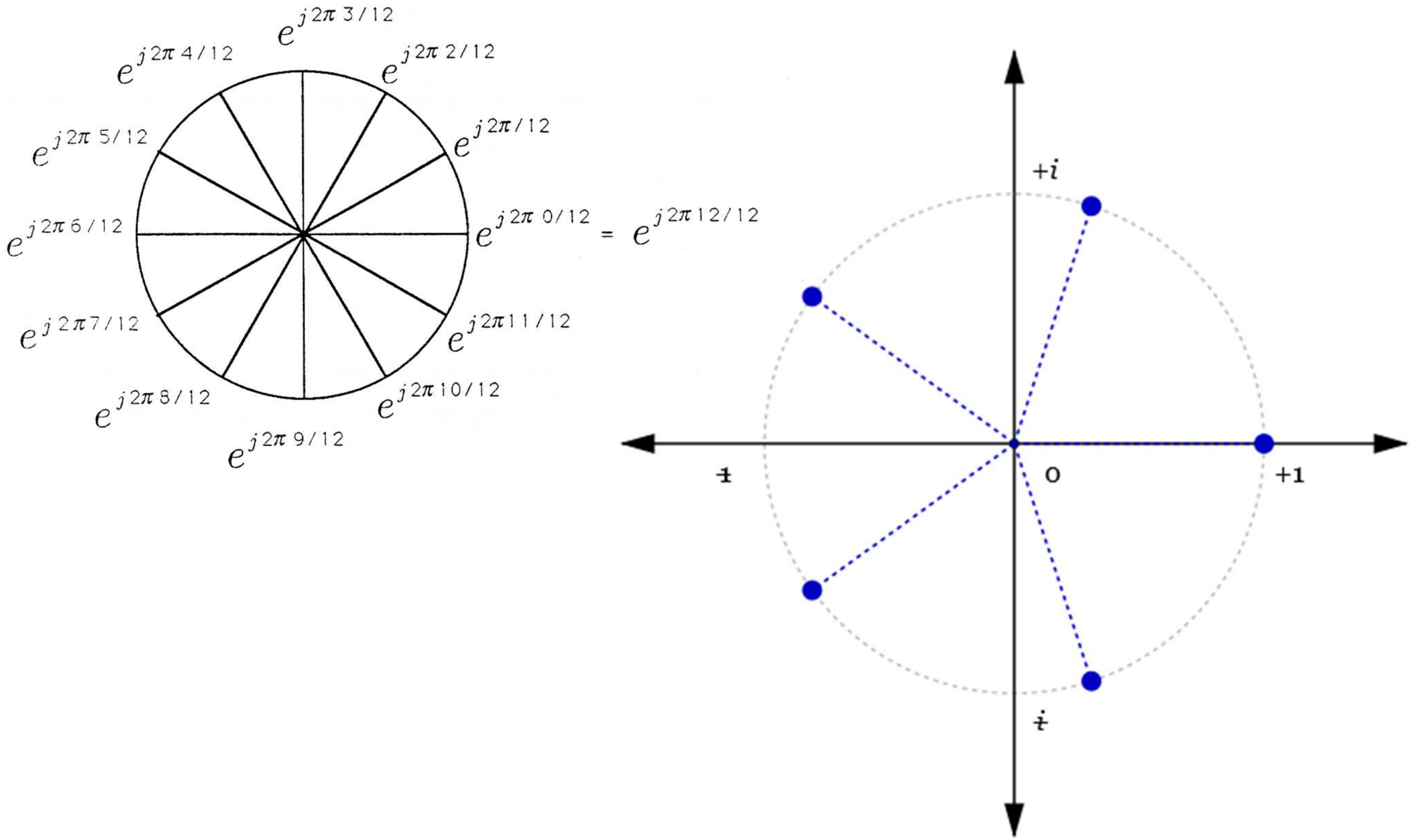
- Alternative formulation:

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

  - X_k - values corresponding to wave *k*
    - Periodic – calculate for 0 ≤ *k* ≤ *N* - 1

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

Roots of unity



Discrete Fourier Transform (DFT)

- Alternative formulation:

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

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

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

(Proof)

- 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)

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

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

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

(Proof)

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

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

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

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

(Proof)

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

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

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

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

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

(Proof)

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

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

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

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

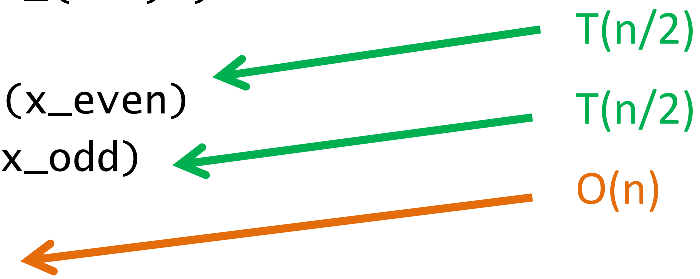
$$= \underbrace{\sum_{n=0}^{2N-1} x_{(2n)} {\omega_{N/2}}^{kn}}_{\text{DFT of } x_n\text{, even n!}} + \omega_N \underbrace{\sum_{n=0}^{2N-1} x_{(2n+1)} {\omega_{N/2}}^{kn}}_{\text{DFT of } x_n\text{, odd n!}}$$

(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?

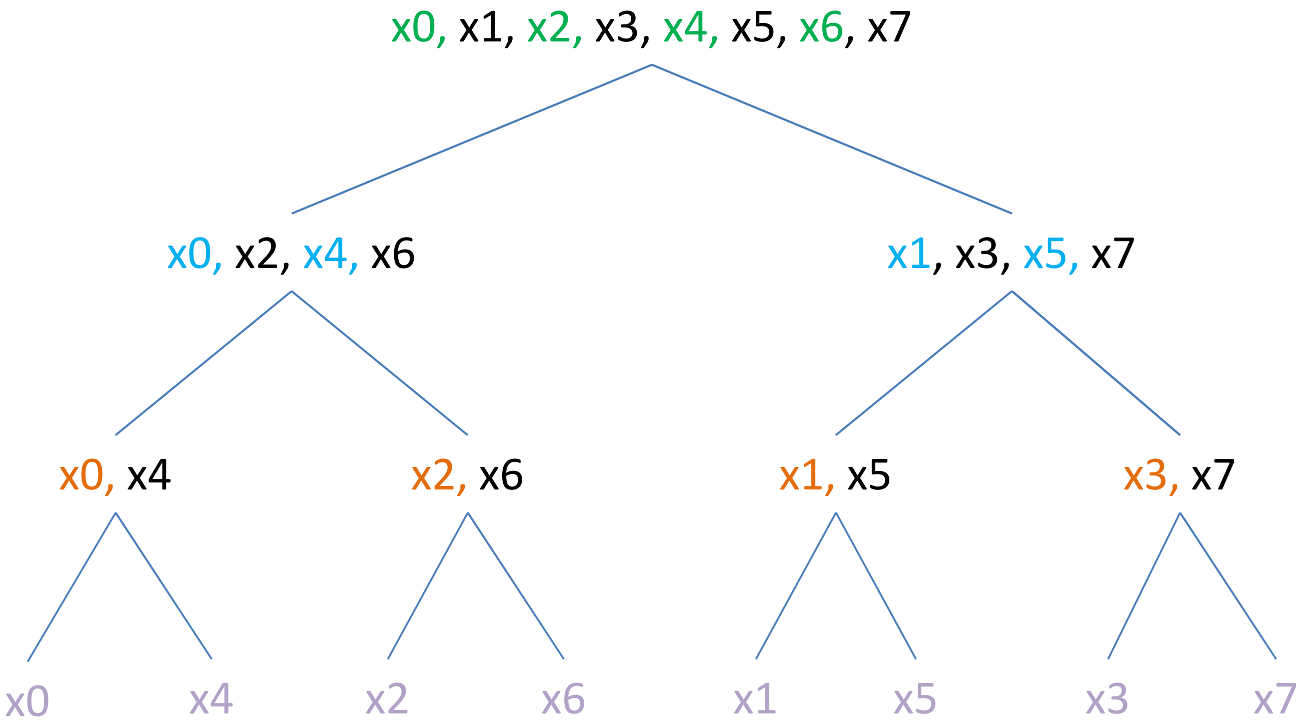
- O(n^2) algorithm certainly is!

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

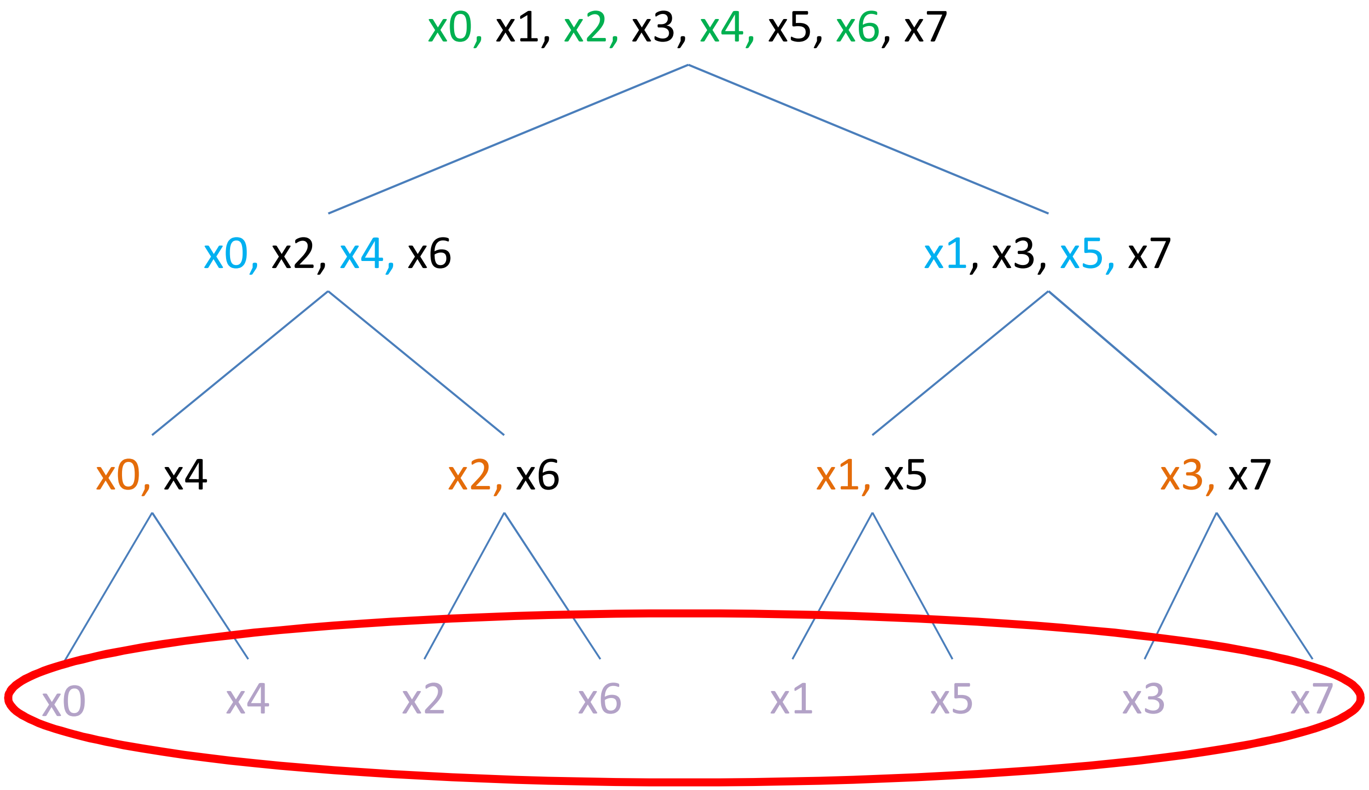
$$X_k \overset{\text{def}}{=} \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi ikn/N}$$

- Sometimes parallelization outweighs runtime!
  - (N-body problem, …)

Recursive index tree



Recursive index tree



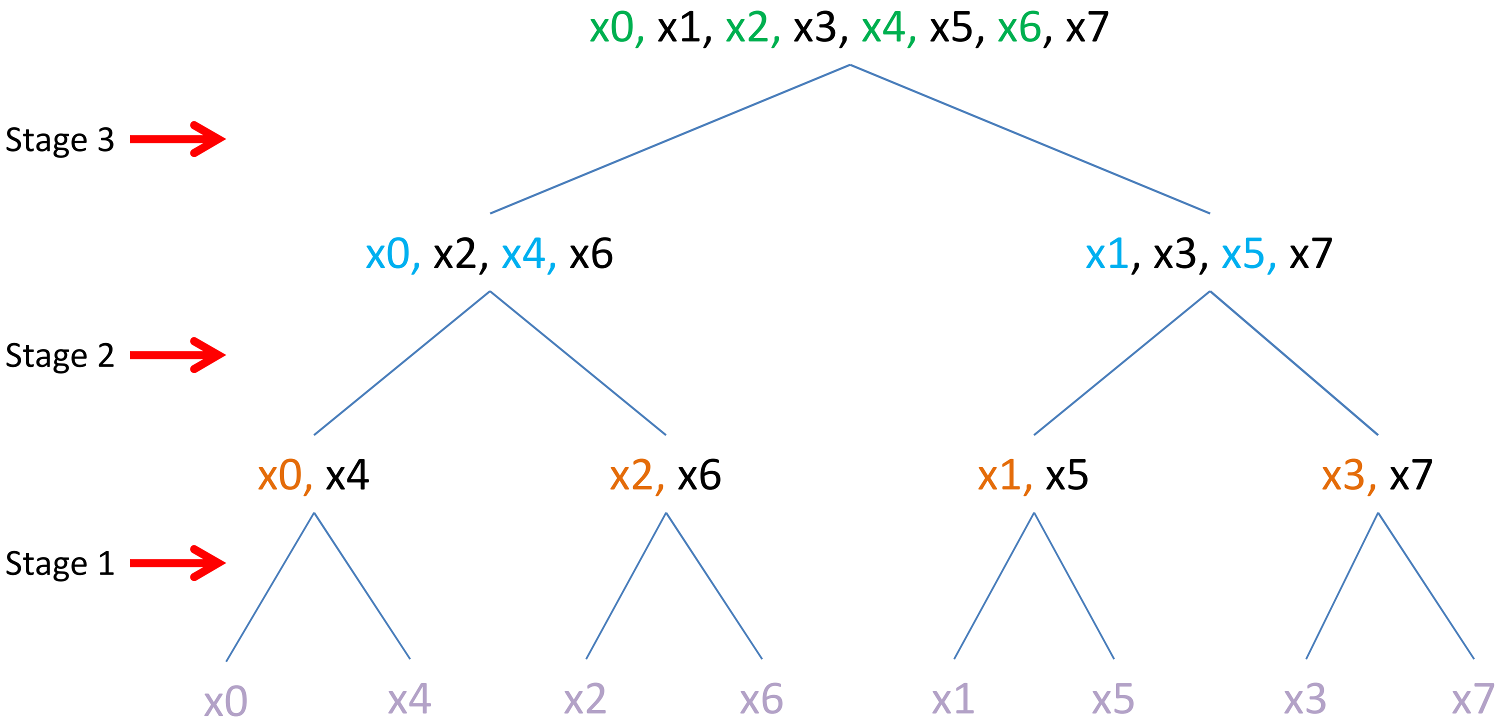
Order?

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

Bit-reversal order

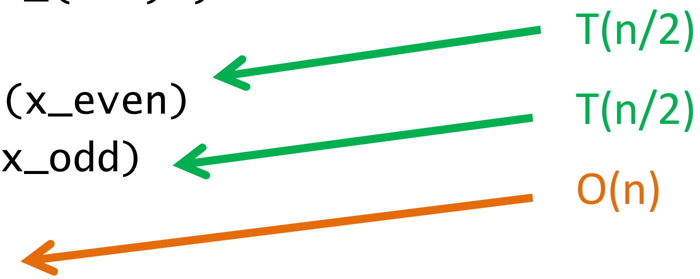
0	000	reverse of…	000
4	100		001
2	010		010
6	110		011
1	001		100
5	101		101
3	011		110
7	111		111

Iterative approach



(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
```

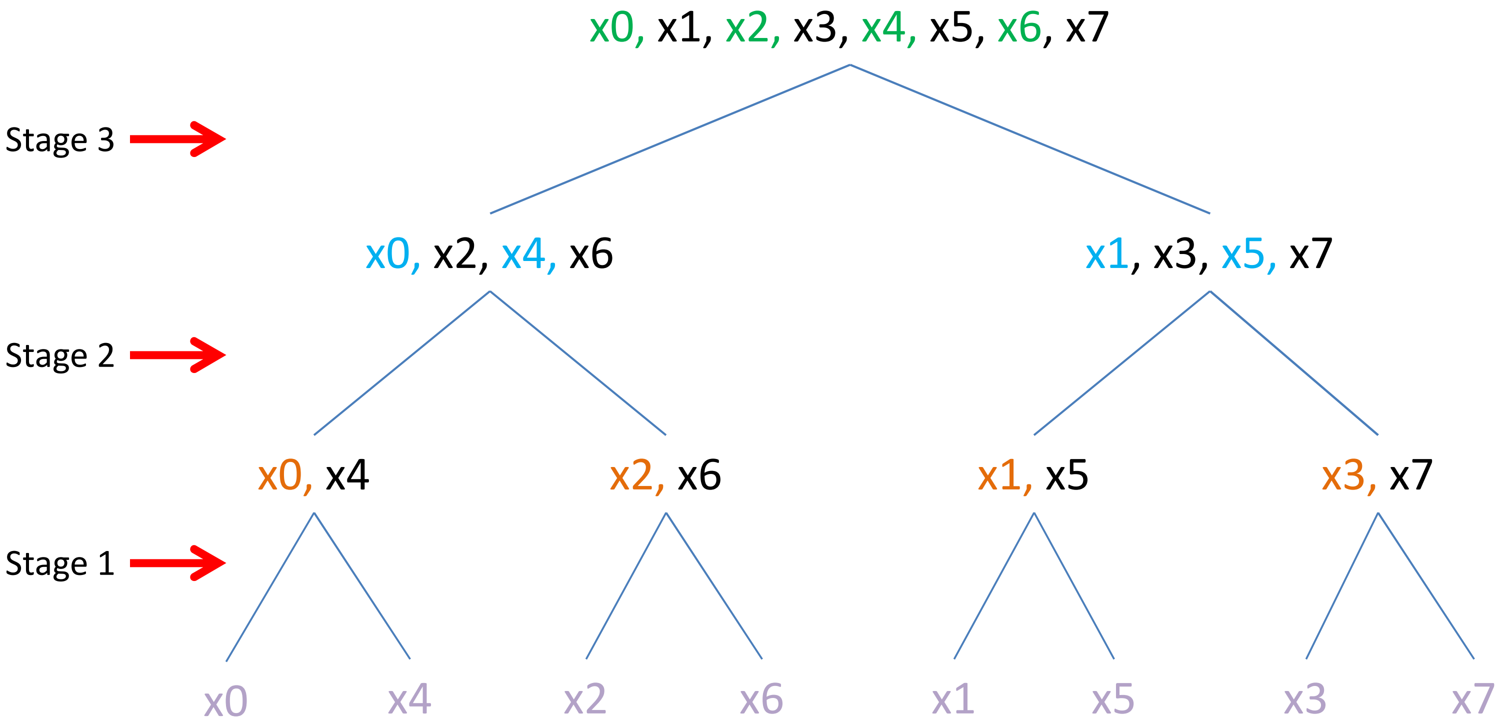


T(n/2)

T(n/2)

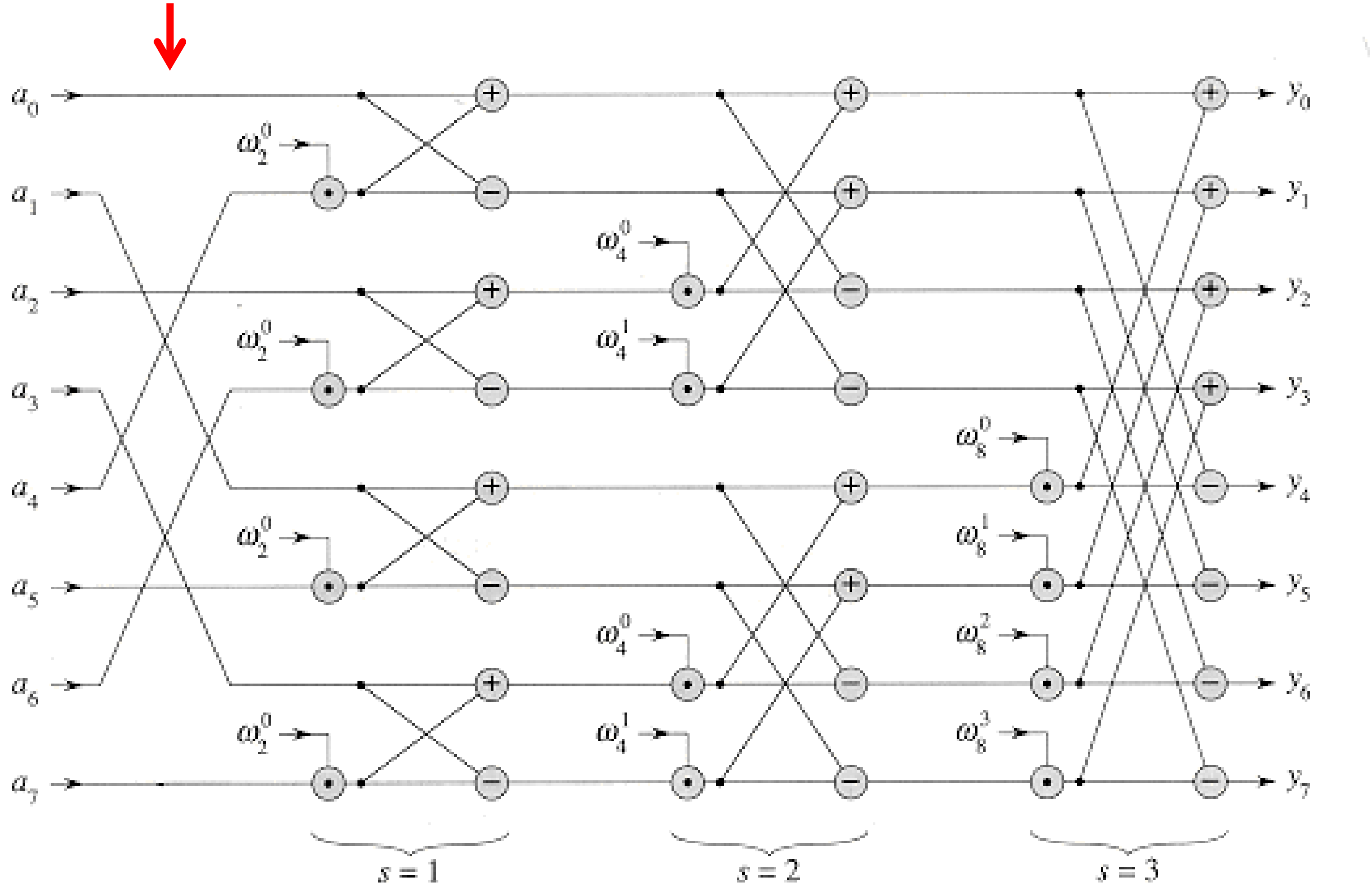
O(n)

Iterative approach



Iterative approach

Bit-reversed access



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

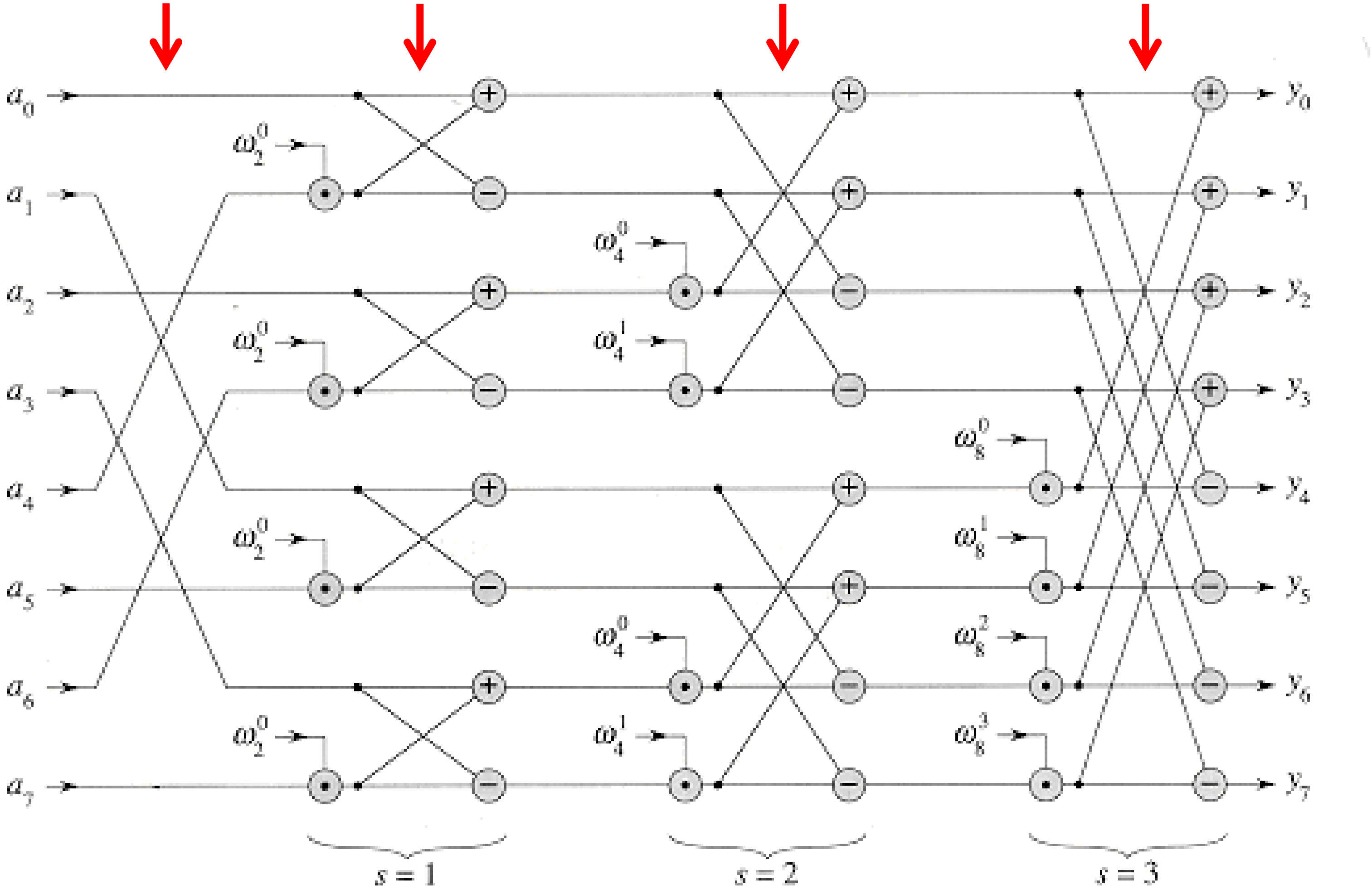
Iterative approach

Bit-reversed access

Stage 1

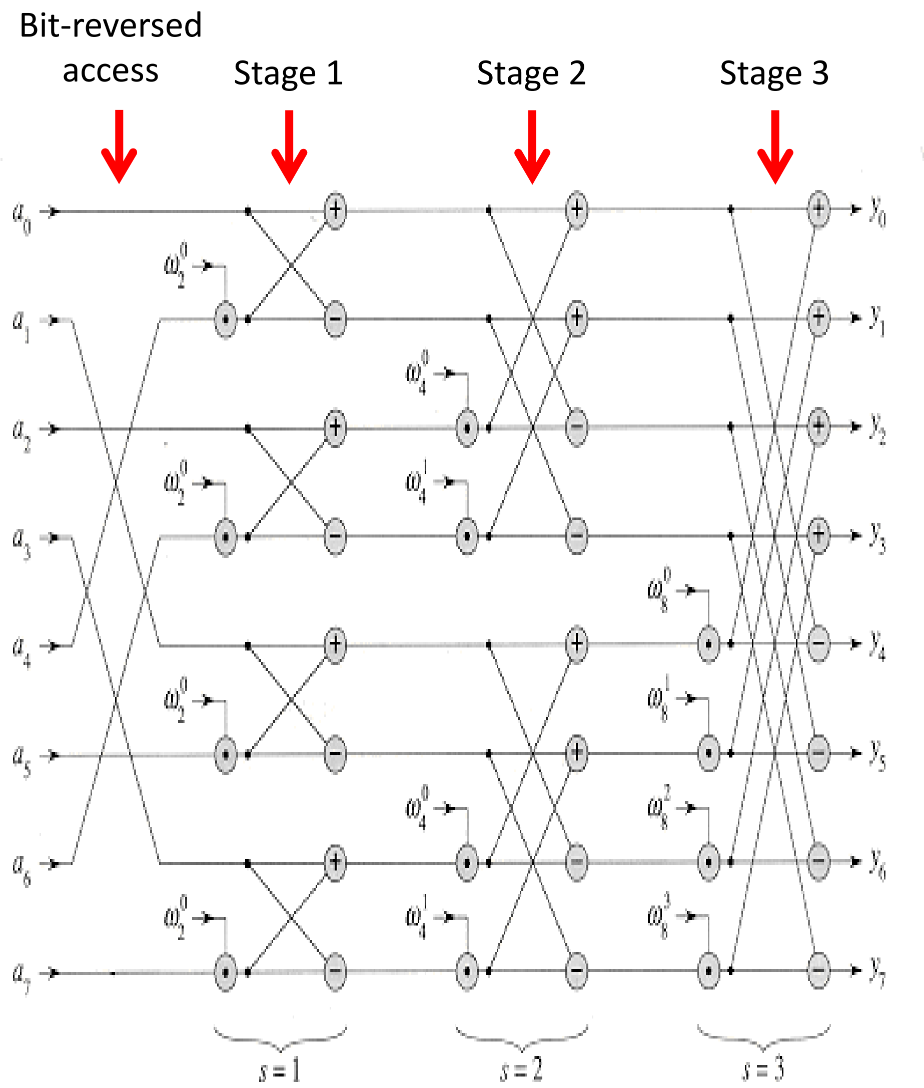
Stage 2

Stage 3



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

Iterative approach



Bit-reversed access

Stage 1

Stage 2

Stage 3

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,…,log(N):

        m <- $2^s$
        $w_n$ <- $e^{2\pi j/m}$

        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
```

Iterative approach

```
Iterative-FFT(Vector x):

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

        m <- $2^s$
        $w_n$ <- $e^{2\pi j/m}$

        for k: $0 \le k \le N-1$, 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
```

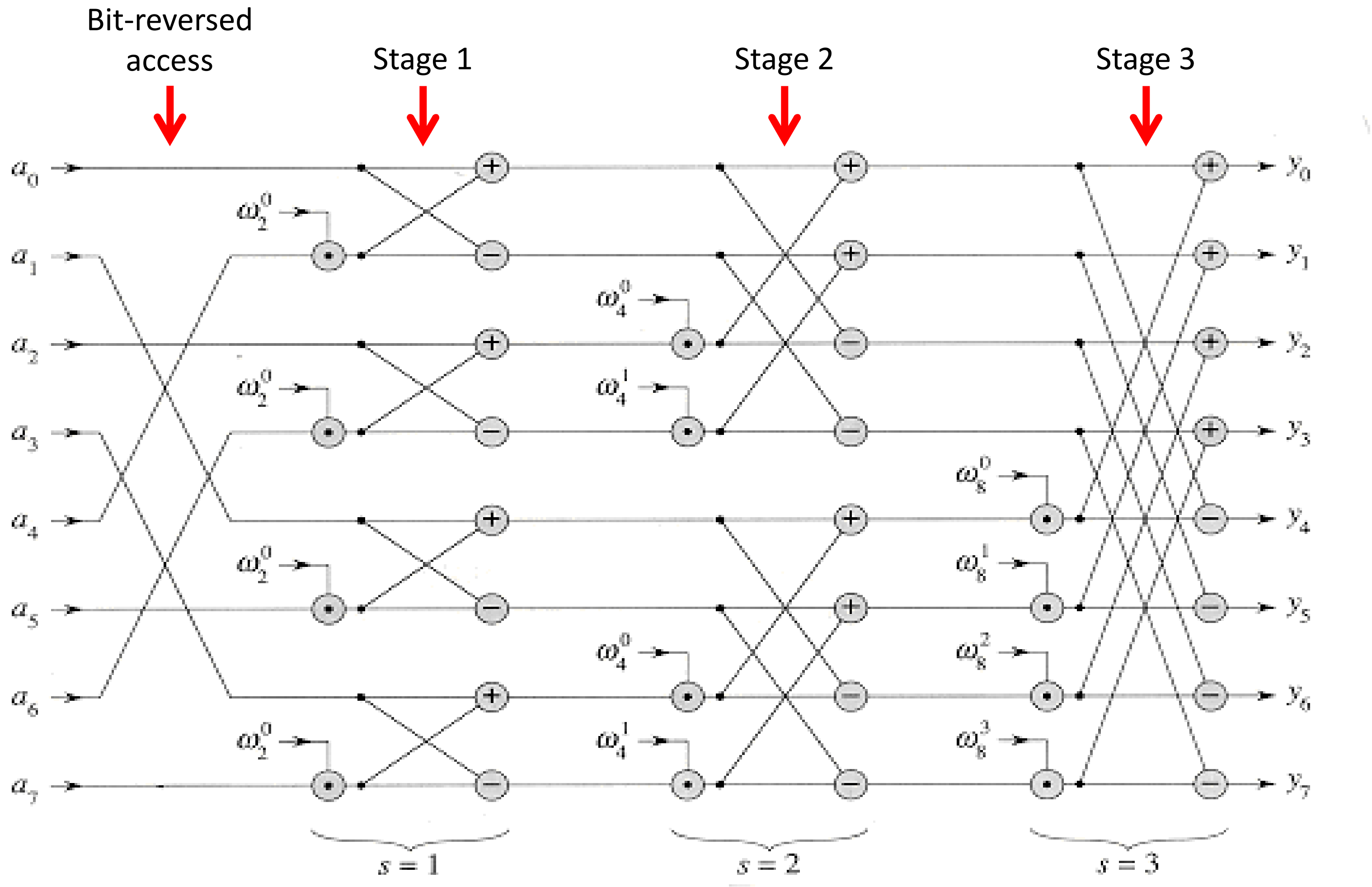
O(log N) iterations

O(N/m) iterations

O(m) iterations

Total: O(N log N) runtime

CUDA approach



Bit-reversed access

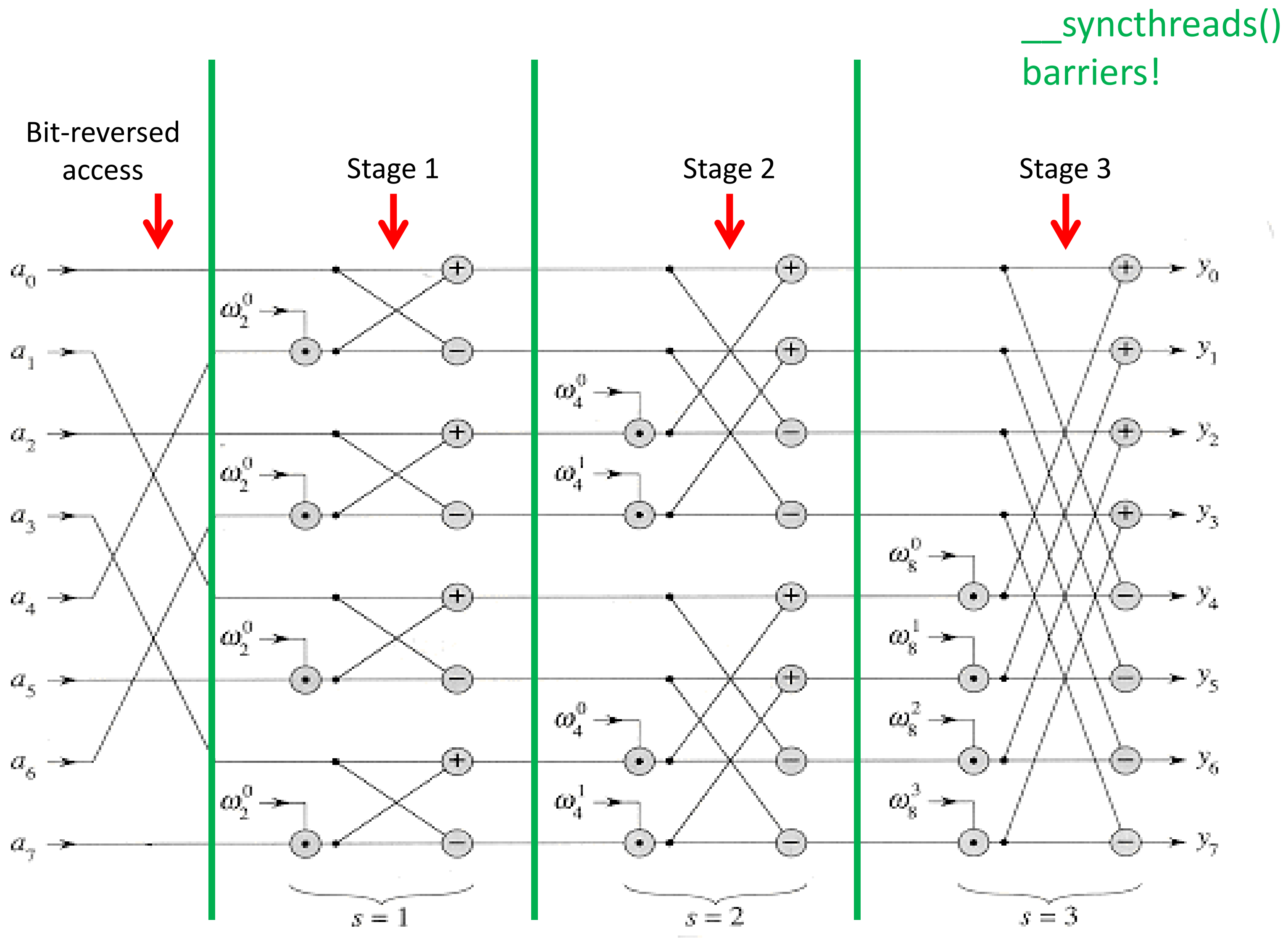
Stage 1

Stage 2

Stage 3

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

CUDA approach



__syncthreads()
barriers!

Bit-reversed access

Stage 1

Stage 2

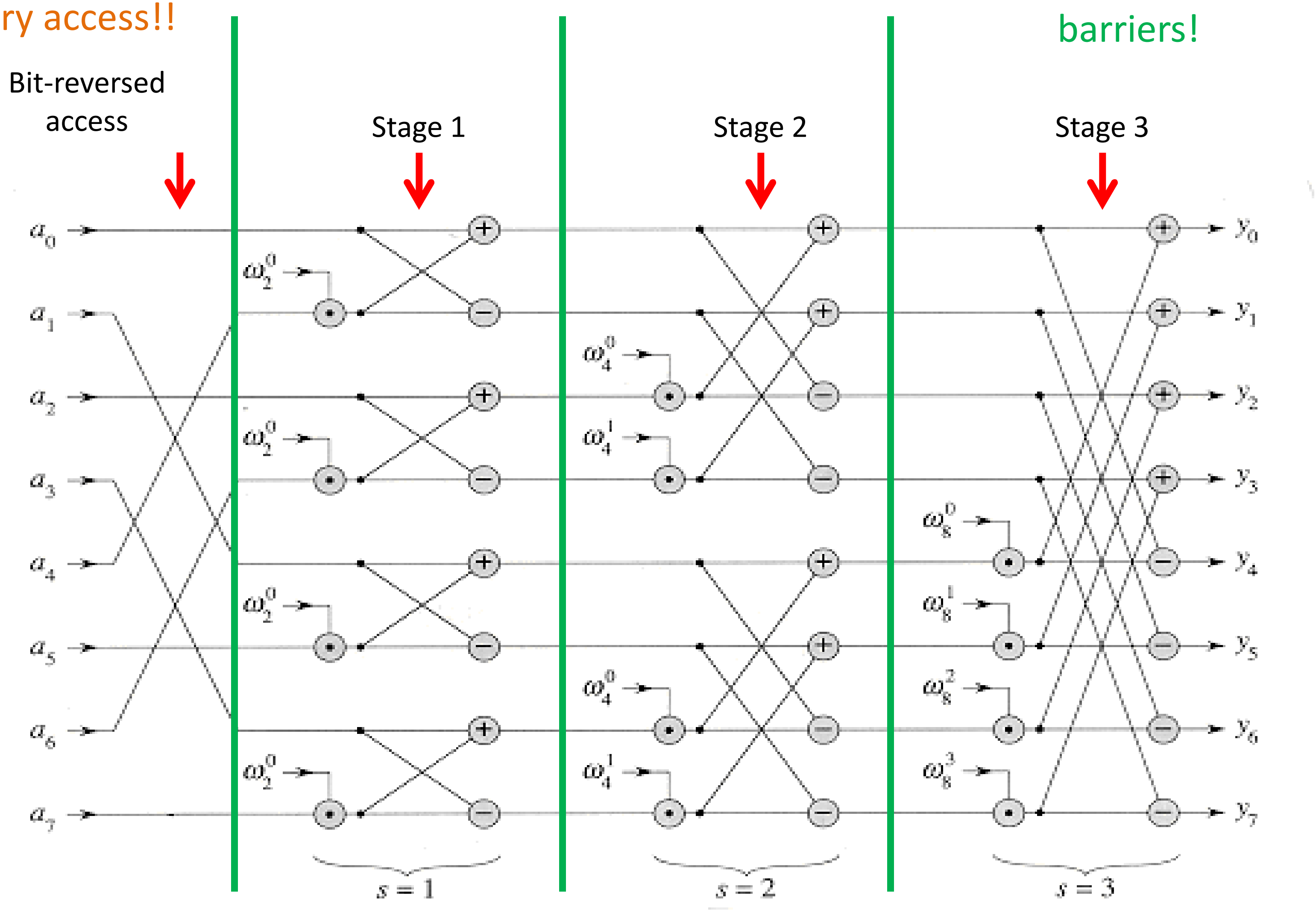
Stage 3

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

CUDA approach

Non-coalesced memory access!!

__syncthreads() barriers!



Bit-reversed access

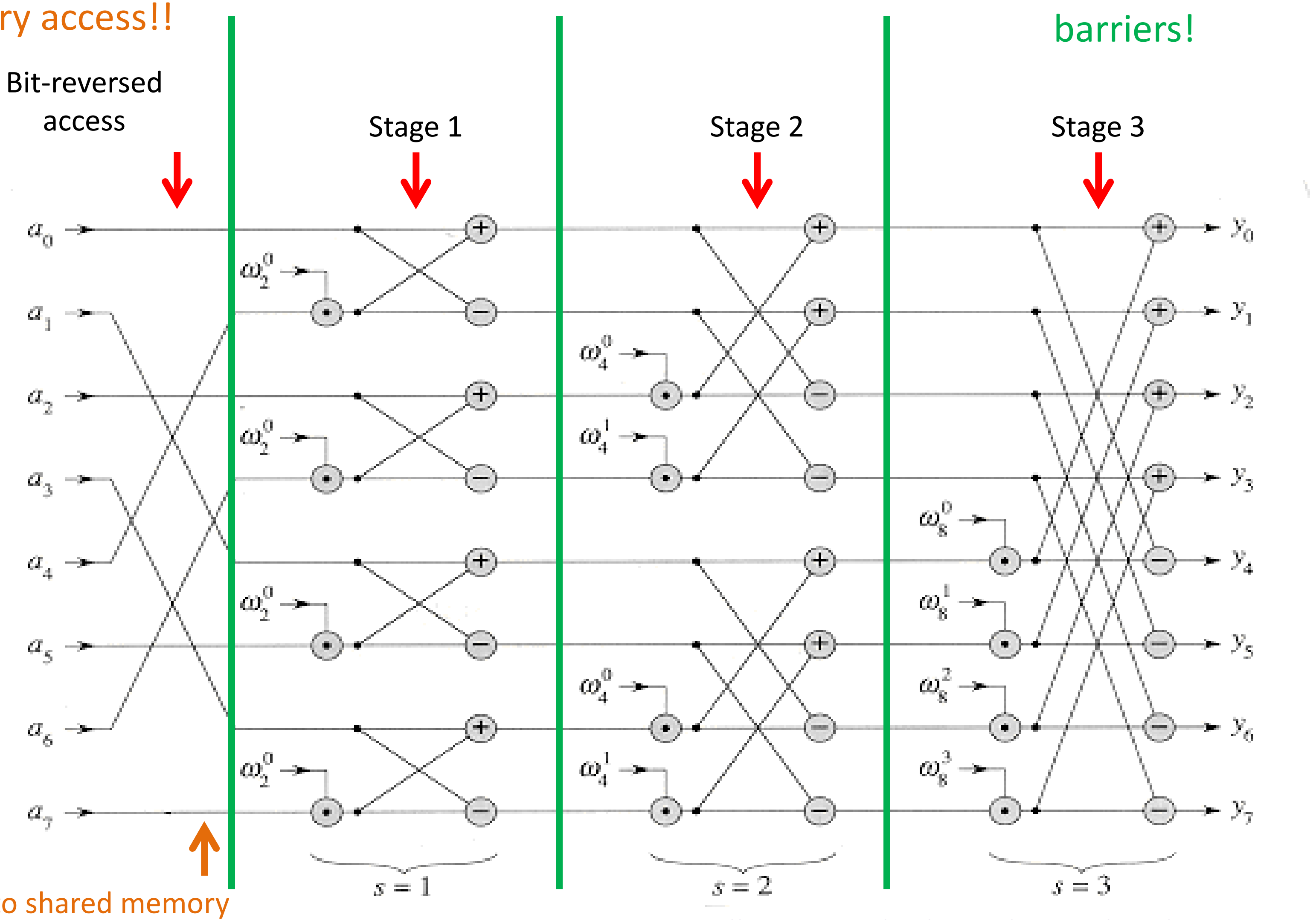
Stage 1

Stage 2

Stage 3

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

CUDA approach



Non-coalesced memory access!!

Bit-reversed access

Stage 1

Stage 2

Stage 3

__syncthreads() barriers!

Load into shared memory

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

CUDA approach

Non-coalesced memory access!!

Bank conflicts!!

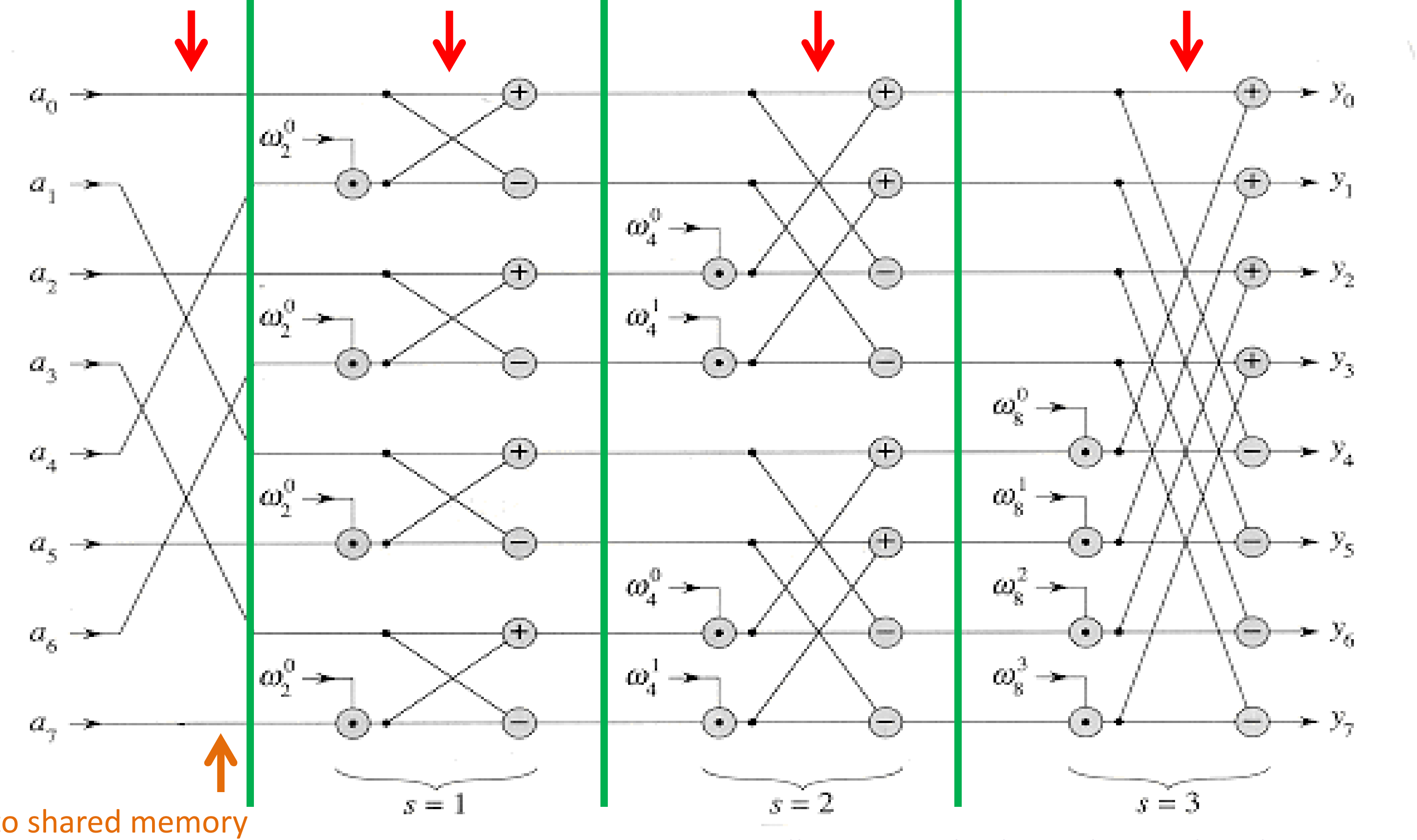
__syncthreads() barriers!

Bit-reversed access

Stage 1

Stage 2

Stage 3



Load into shared memory

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

Generalization

- Above algorithm works for array sizes of 2^n (integer n)
- Generalize?

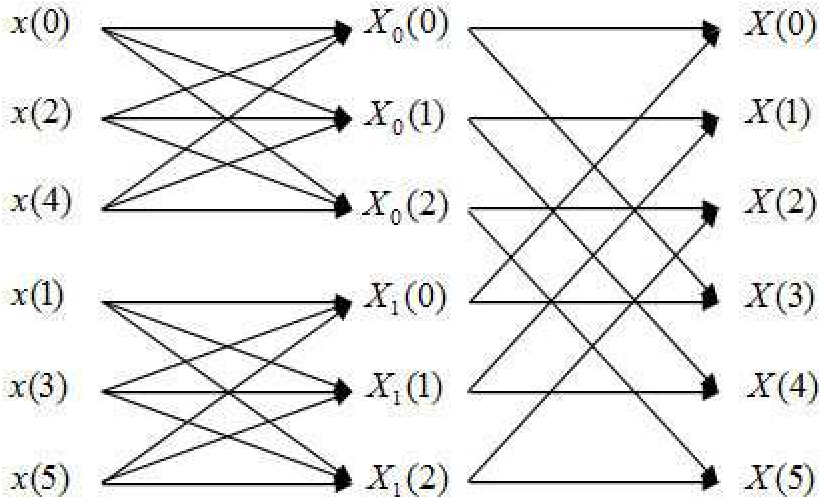
“Bit”-reversal order

- N = 6: (*small* example)
  - – Factors into 3,2
    - 1st digit: base 3
    - 2nd digit: base 2

0	00		00	0
1	01		10	2
2	10	→	20	4
3	11		01	1
4	20		11	3
5	21		21	5

Generalization

- Radix-3 case (followed by radix-2)



https://www.projectrhea.org/

- Suppose N = N_1N_2 (factorization…)
- Algorithm: (recursive)
  - Take FFT along rows
  - Do multiply operations (complex roots of unity)
  - Take FFT along columns

- “Cooley-Tukey FFT algorithm”

Inverse DFT/FFT

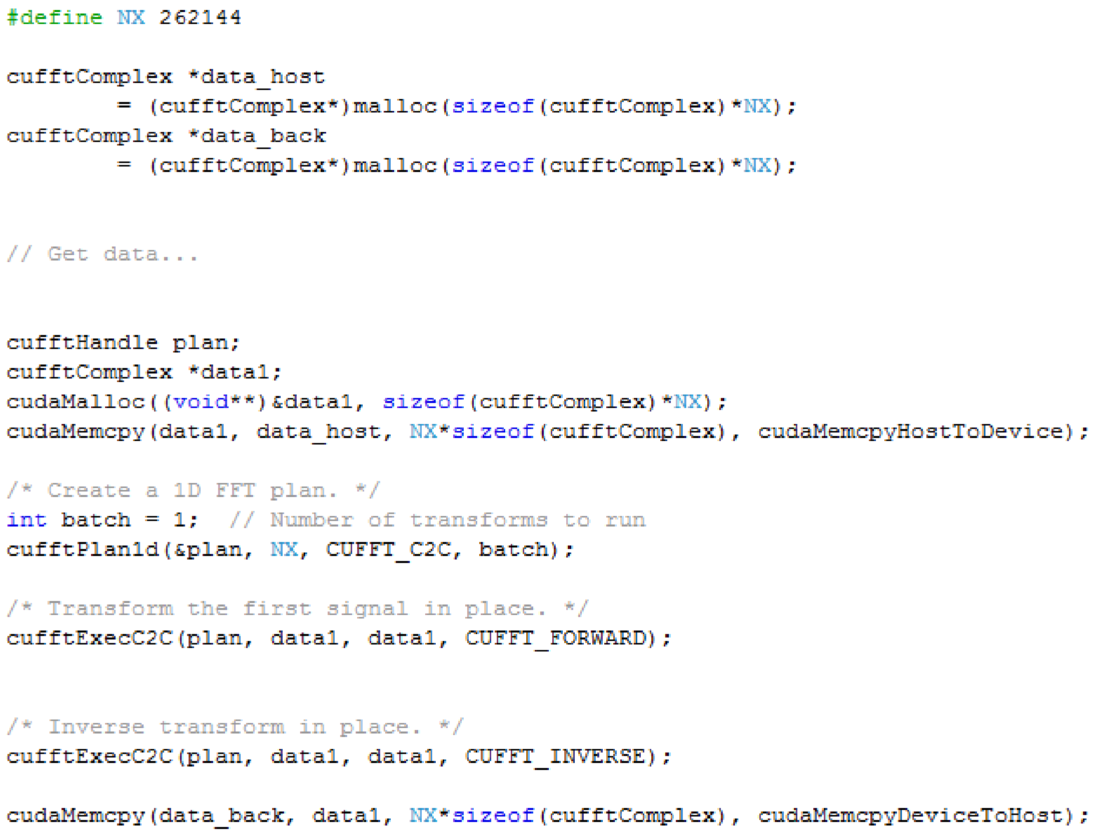
- Similarly parallelizable!
  - (Sign change in complex terms)

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

cuFFT

- FFT library included with CUDA
  - Approximately implements previous algorithms
    - (Cooley-Tukey/Bluestein)
    - Also handles higher dimensions

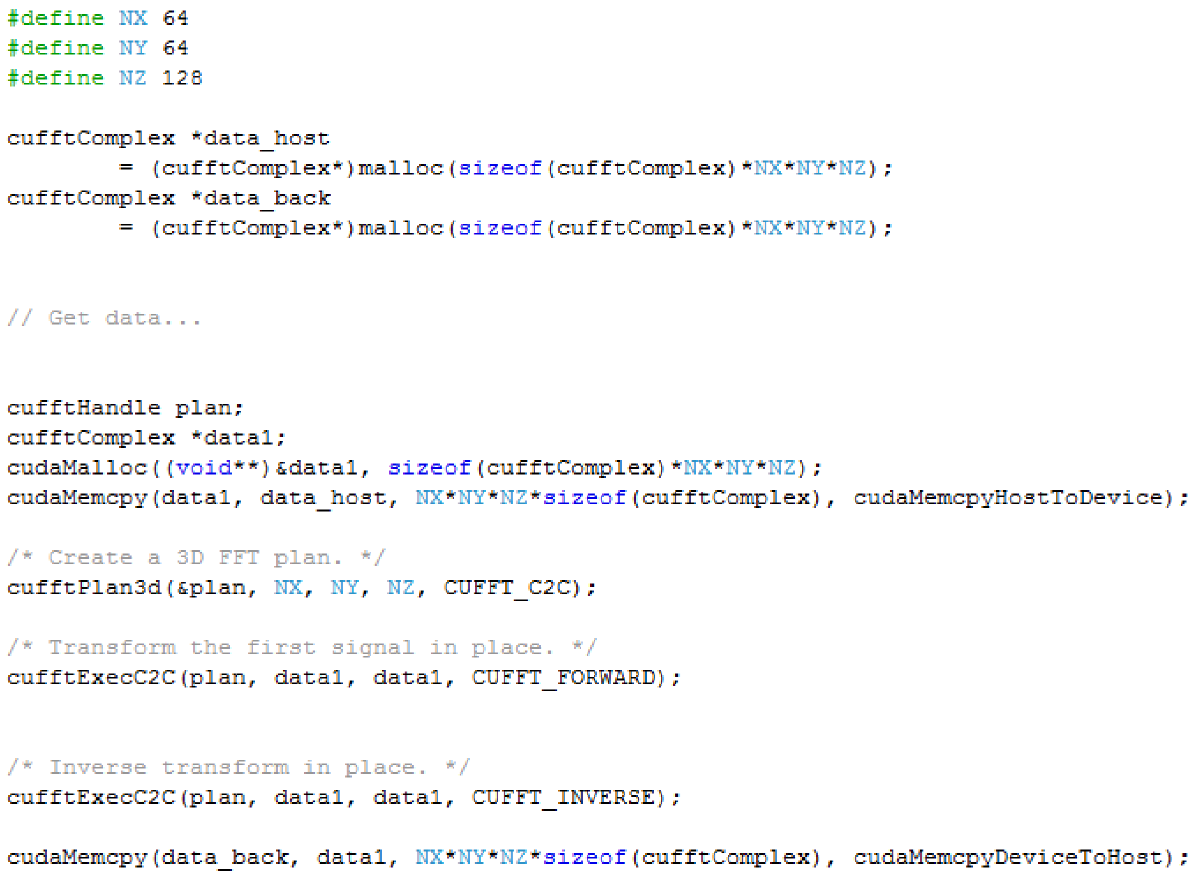
cuFFT 1D example



```
#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.)