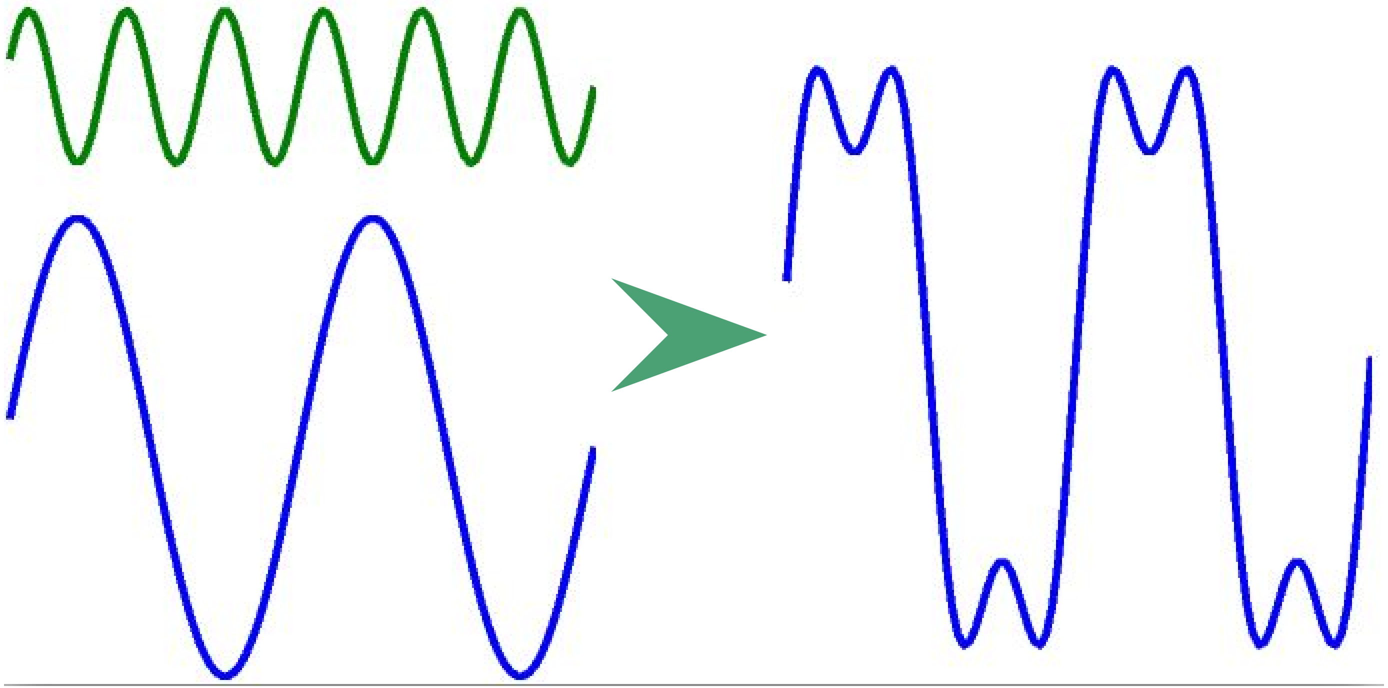
Parallel FFT: CUDA and Cilk

Ari Bruck
Eric Addison
EEW382V - Parallel Algorithms - Summer 17

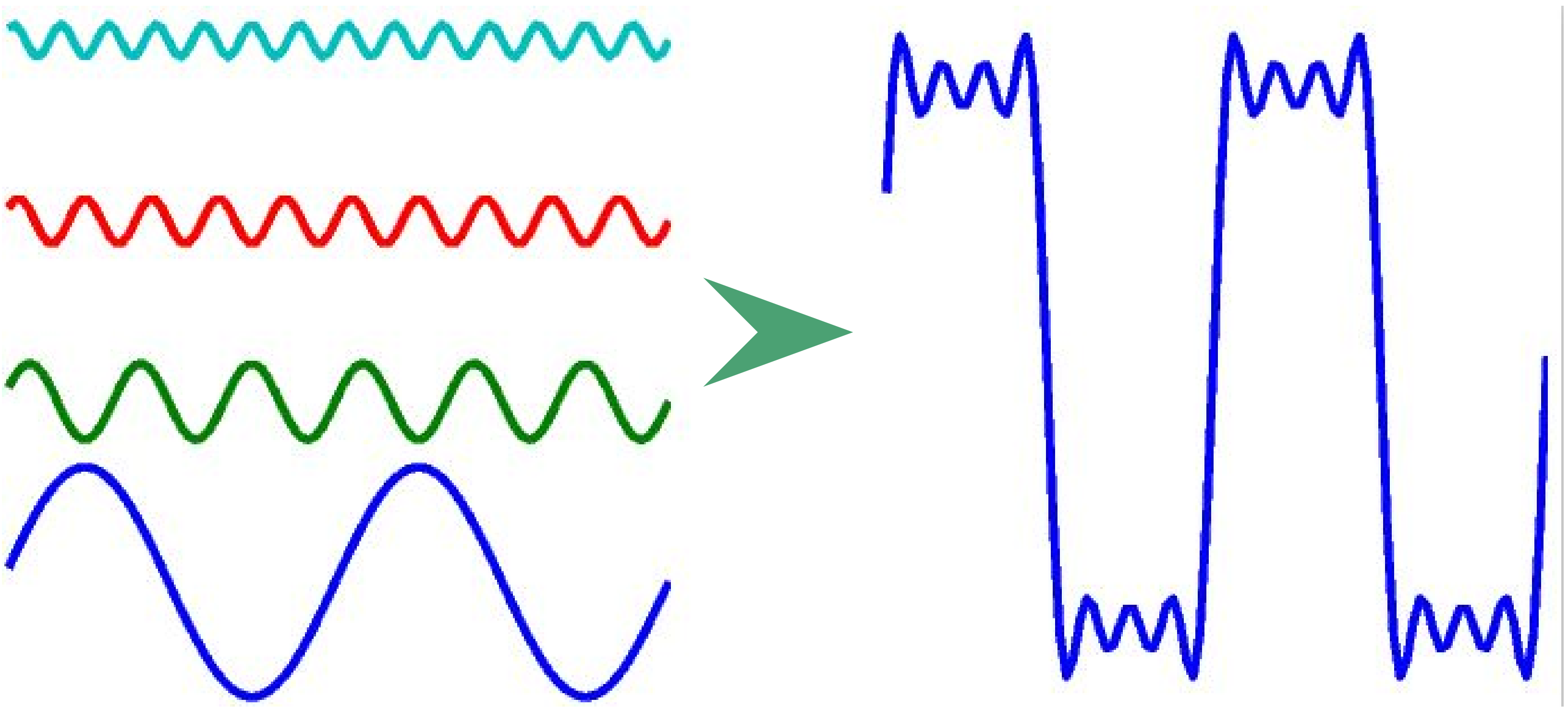
Topics

1. Fourier Decomposition
2. Discrete Fourier Transforms
3. Fast Fourier Transforms
    a. Recursive FFT
    b. Iterative FFT
    c. Parallel FFT
4. Parallel FFT using Cilk
5. Parallel FFT using CUDA
6. Performance
7. Conclusion

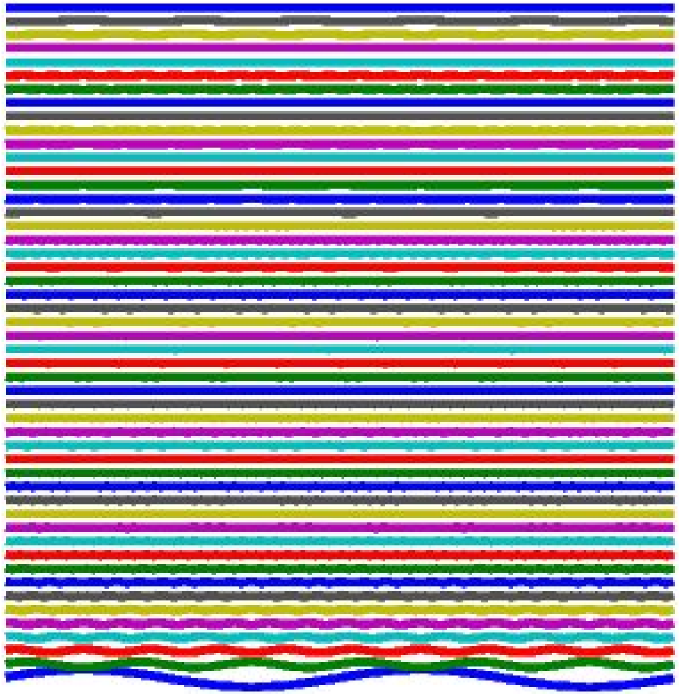
Fourier Decomposition

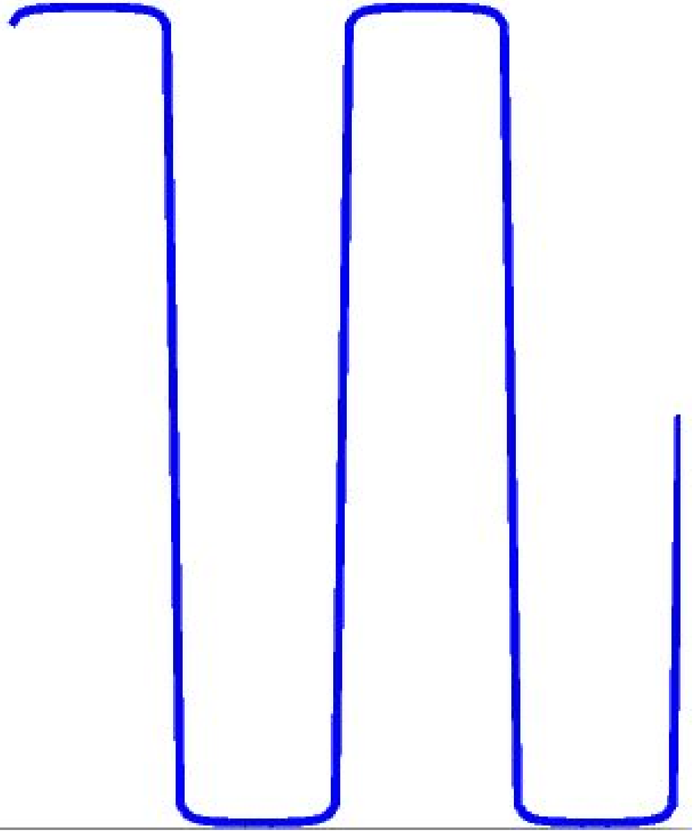


Fourier Decomposition



Fourier Decomposition





Fourier Decomposition

$$x(t) = \sum_{k=-\infty}^{\infty} c_k e^{ik\omega_0 t} \longleftrightarrow X[k] = \frac{1}{T}\int_0^T x(t)e^{-ik\omega_0 t}dt$$

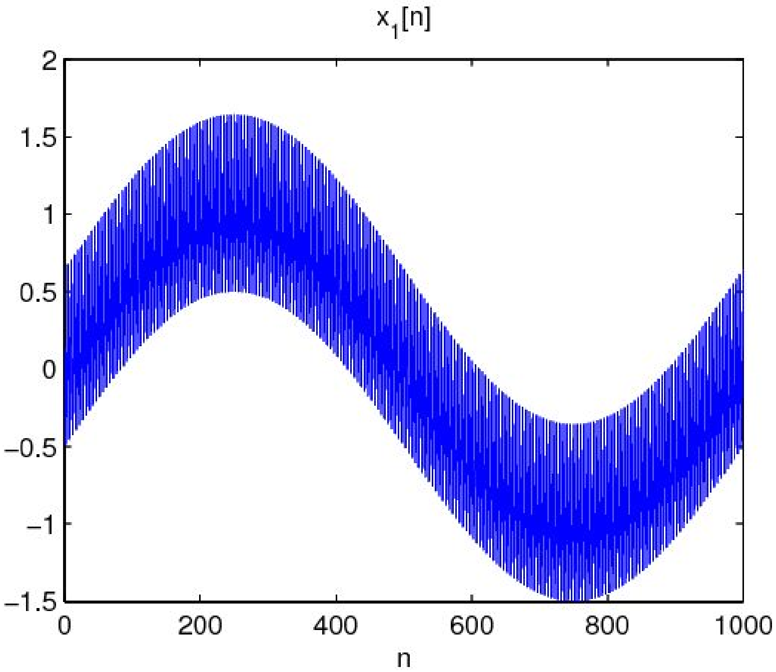
$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)e^{i\omega t}d\omega \longleftrightarrow X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-i\omega t}dt$$

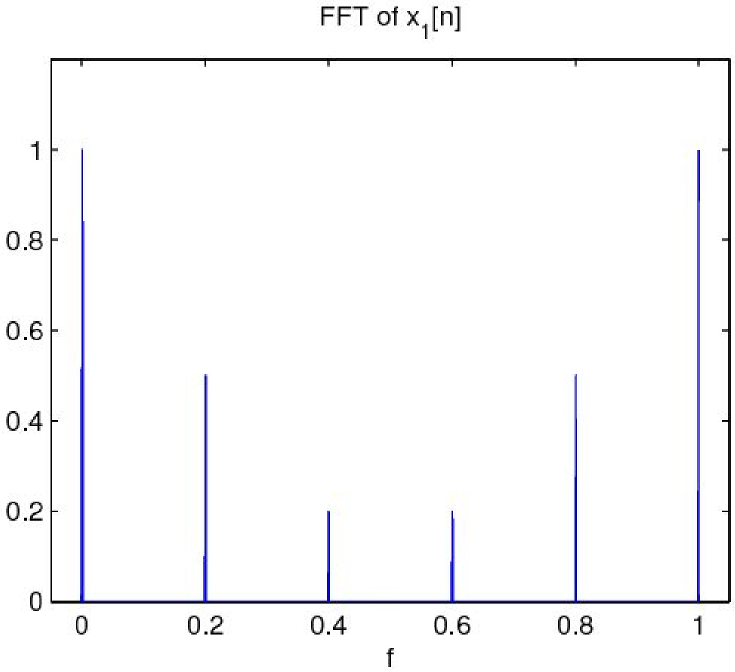
$$x[n] = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(e^{i\omega})e^{i\omega n}d\omega \longleftrightarrow X(e^{i\omega}) = \sum_{n=-\infty}^{\infty} x[n]e^{-i\omega n}$$

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{i2\pi nk/N} \longleftrightarrow X[k] = \sum_{n=0}^{N-1} x[n]e^{-i2\pi nk/N}$$

Fourier Analysis

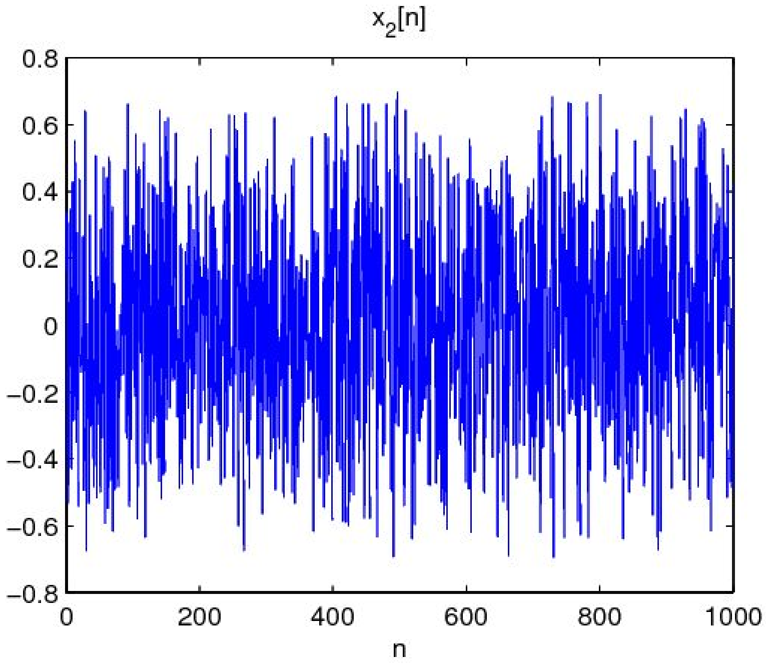
$$x_1[n] = \sin(2\pi f_1 n) + 0.5\cos(2\pi f_2 n) + 0.2\cos(2\pi f_2 n + \pi/4)$$

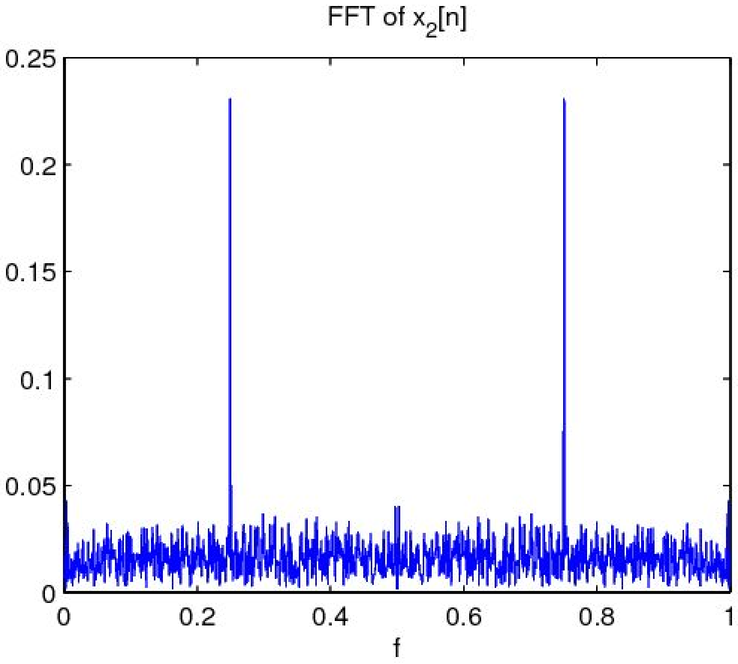




Fourier Analysis

$$x_2[n] = U + 0.2\sin(2\pi f_0 n)$$





Discrete Fourier Transform

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-i2\pi nk/N}$$

We can implement this on a computer!

DFT Implementation

```
DFT(x)
N = x.length
v = -i*2*pi/N          // i == imaginary unit
for k in 1..N
      X[k] = 0
       for j in 1.. N
             X[k] += x[j] * exp(v*k)
```

The Fast Fourier Transform

Q: What is a *Fast* Fourier Transform???

The Fast Fourier Transform

Q: What is a *Fast* Fourier Transform???

A: A Fourier Transform that is not slow!!!

The Fast Fourier Transform

Q: What is a *Fast* Fourier Transform???

A: A Fourier Transform that is not slow!!!

“Fast Fourier Transform” refers to a class of algorithms that can compute a DFT in better than $O(N^2)$ time.

Cooley-Tukey FFT Algorithm

Divide and Conquer!

$$\begin{aligned}
X[k] &= \sum_{n=0}^{N-1} x[n] e^{-i2\pi nk/N} \\
&= \sum_{n=0}^{N/2-1} x[2n] e^{-i2\pi(2n)k/N} + \sum_{n=0}^{N/2-1} x[2n+1] e^{-i2\pi(2n+1)k/N} \\
&= \sum_{n=0}^{N/2-1} x[2n] e^{-i2\pi(2n)k/N} + e^{-i2\pi k/N} \sum_{n=0}^{N/2-1} x[2n+1] e^{-i2\pi(2n)k/N} \\
&= \sum_{n=0}^{N/2-1} x_e[n] e^{-i2\pi nk/(N/2)} + e^{-i2\pi k/N} \sum_{n=0}^{N/2-1} x_o[n] e^{-i2\pi nk/(N/2)} \\
&= X_e[k] + e^{-i2\pi k/N} X_o[k]
\end{aligned}$$

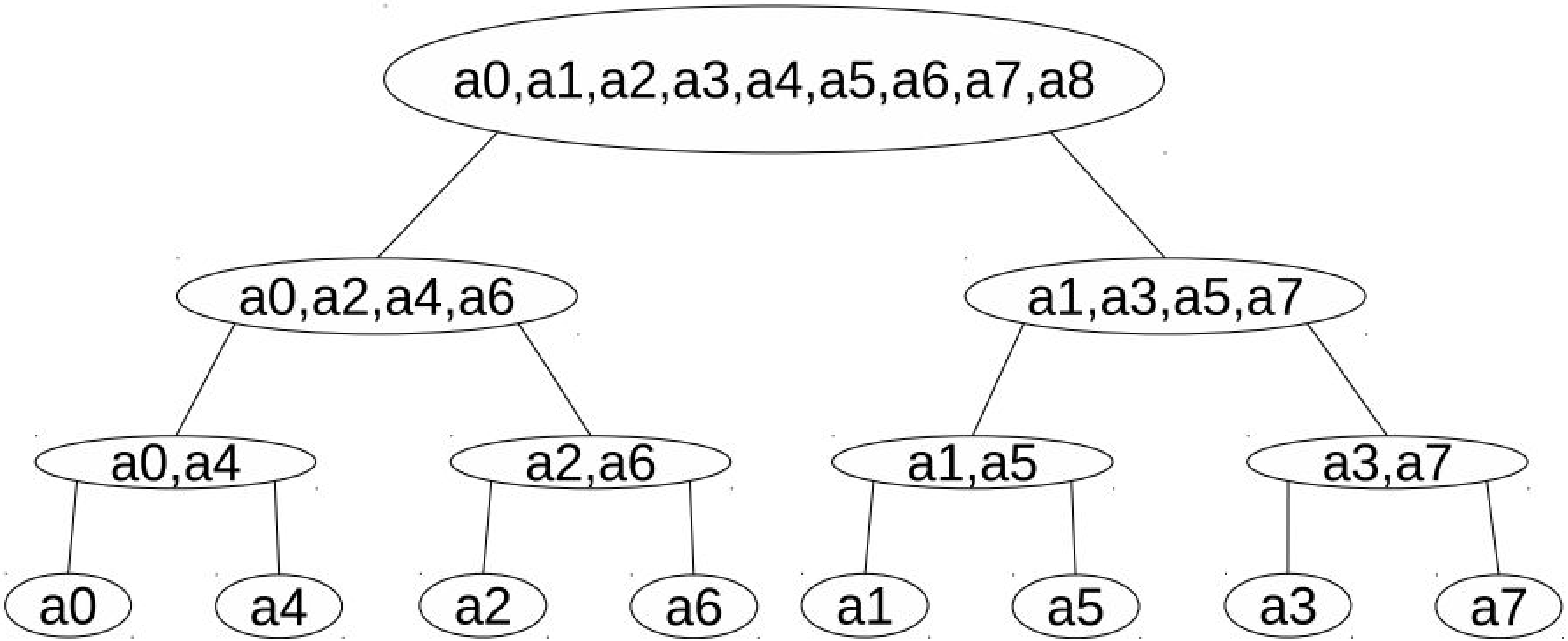
Recursive FFT

```
FFT_REC(x)
    N = x.length
    if N <= 1
        return
    v = -i*2*pi/N        // i == imaginary unit
    for s in 2..log(N)
        E = FFT_REC( [x[2*n] for n = 0..N/2-1] )
        O = FFT_REC( [x[2*n+1] for n = 0..N/2-1] )
        for j in 0..N/2-1
            w = exp(v*j)
            x[2*j] = E[j] + w*O[j]
            x[2*j+N/2] = E[j] - w*O[j]
    return x
```

Recursive FFT Time Complexity

$$
\begin{aligned}
T(N) &= T(N/2) + T(N/2) + \mathcal{O}(N/2) \\
&= (T(N/4) + T(N/4) + \mathcal{O}(N/4)) \\
&\quad + (T(N/4) + T(N/4) + \mathcal{O}(N/4)) + \mathcal{O}(N/2) \\
&= 4T(N/4) + 2\mathcal{O}(N/4) + \mathcal{O}(N/2) \\
&= \ldots \\
&= 2^h T(1) + \sum_{i=1}^{h} 2^i \mathcal{O}(N/2^i) \\
&= N\mathcal{O}(1) + \sum_{i=1}^{\log N} \mathcal{O}(N) \\
&= \mathcal{O}(N) + \mathcal{O}(N \log N) \\
&= \mathcal{O}(N \log N)
\end{aligned}
$$

Iterative FFT Algorithm



Iterative FFT: Continued

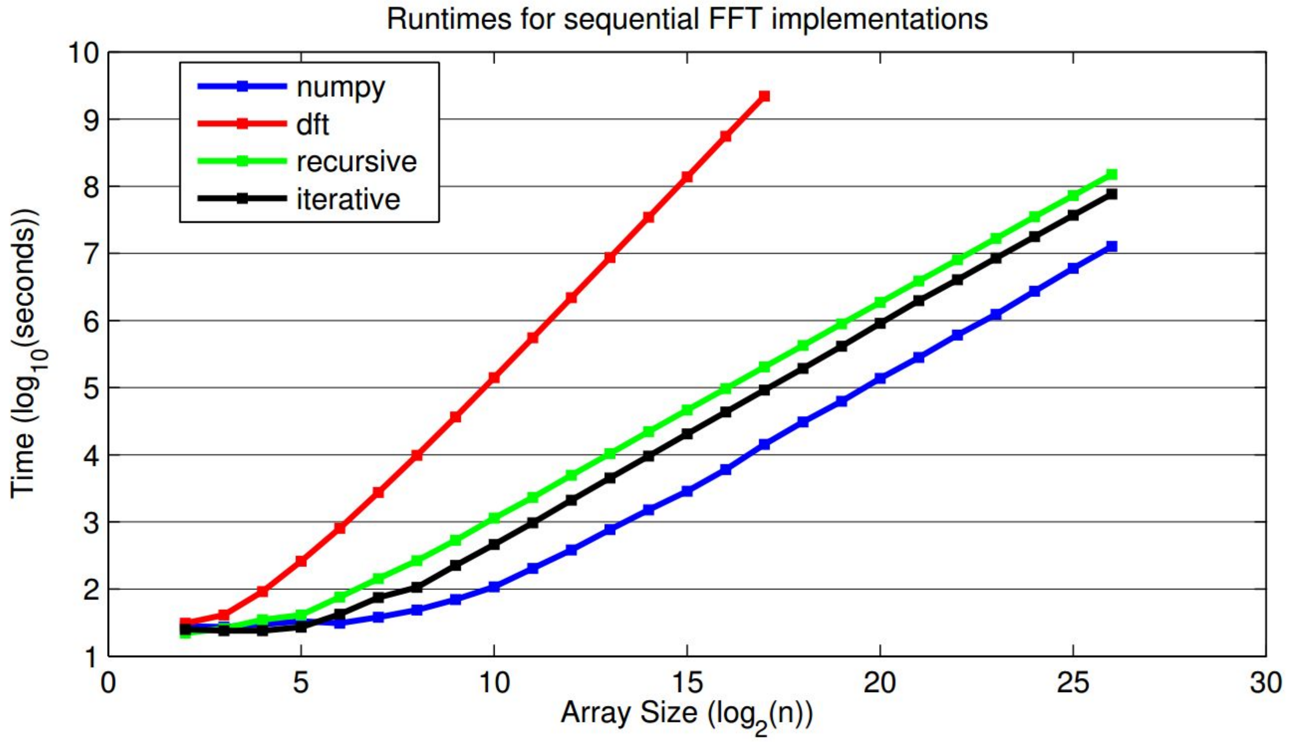
- Example: 8-element vector = (0,4,2,6,1,5,3,7).
  - Original input (000,001,010,011,100,101,110,111)
  - Bit-Reversed (000,100,010,110,001,101,011,111)

```
Bit-Reverse(x)
      y = empty vector
      for k in 0..x.length-1
            k2 = reverse_bits(k)
            y[k] = x[k2]
      return y
```

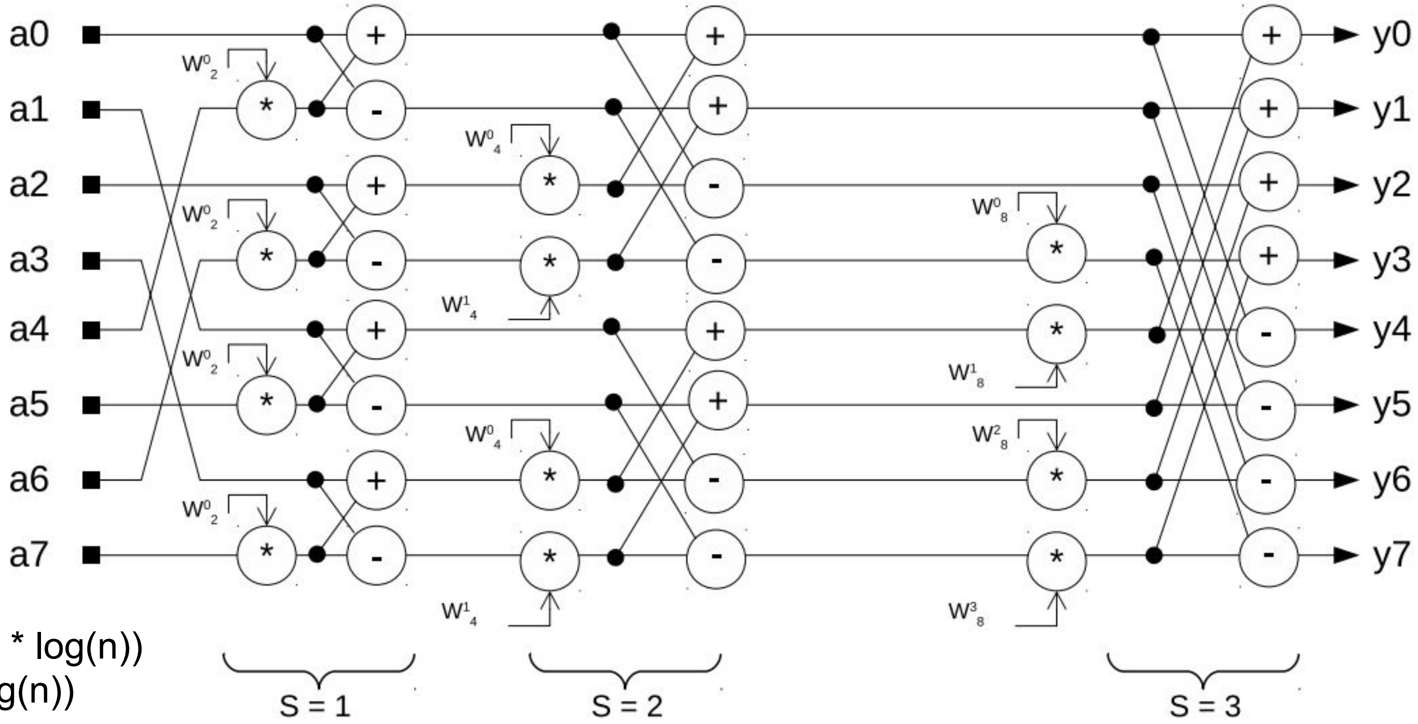
Iterative FFT: Continued

```
FFT_ITER(x)
    Bit-Reverse (x)
    N = x.length
    for s in 1..log(N)
        m = 2^s
        wm = exp(-i2*pi/m)
        w = 1
        for j in 0..(m/2-1)
            for k in j..(n-1) by m
                t = w * x[k + m/2]
                u = x[k]
                x[k] = u + t
                x[k + m/2] = u - t
            w = w*wm
    return x
```

FFT Algorithm Performance



Parallel FFT Circuit



W(n) = O(n * log(n))
T(n) = O(log(n))

Cilk: Introduction

- Cilk is an extension to C/C++ developed by MIT and further enhanced by Intel
- It allows a programmer to easily parallelize a program by inserting certain keywords into targeting portions of an algorithm
- The Cilk framework can easily leverage the hardware capabilities of the system
- The Cilk scheduler is responsible for dividing work between multiple processors
- Parallelizable functions must be exposed by the programmer using a variety of keywords including “spawn” and “sync”
- Cilk enforces rules regarding parent-child relationships. For example, a parent cannot return until all children have “sync[ed]”.

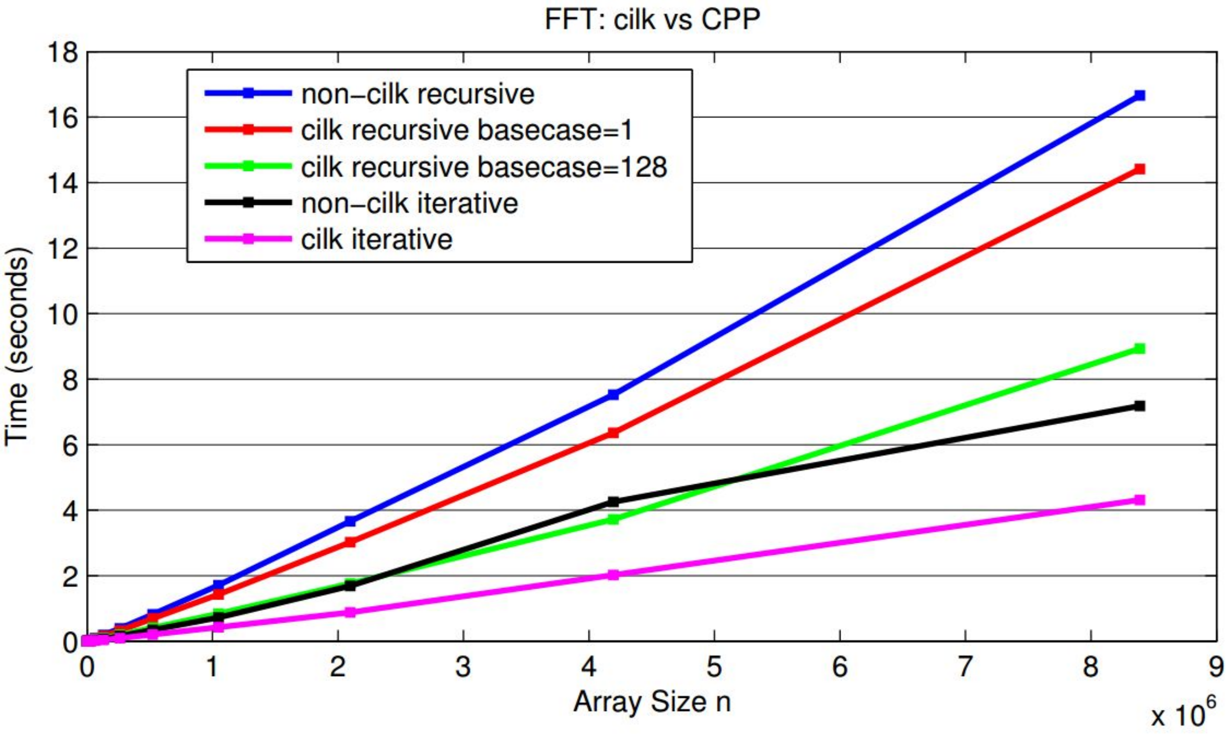
Cilk: Recursive FFT

```
FFT_REC(x)
        N = x.length
        if N <= 1
                return
        v = -i*2*pi/N          // i == imaginary unit
        for s in 2..log(N)
    cilk_spawn E = FFT_REC( [x[2*n] for n = 0..N/2-1] )
    cilk_spawn O = FFT_REC( [x[2*n+1] for n = 0..N/2-1] )

                cilk_sync

            cilk_for j in 0..N/2-1
                w = exp(v*j)
                x[2*j] = E[j] + w*O[j]
                x[2*j+N/2] = E[j] - w*O[j]
        return x
```

Cilk: Iterative FFT

```
FFT_ITER(x)
    Bit-Reverse (x)
    N = x.length
    for s in 1..log(N)
        m = 2^s
        wm = exp(-i2*pi/m)
        w = 1
        cilk_for j in 0..(m/2-1)
            for k in j..(n-1) by m
                t = w * x[k + m/2]
                u = x[k]
                x[k] = u + t
                x[k + m/2] = u - t
            w = w*wm
    return x
```

Cilk Performance



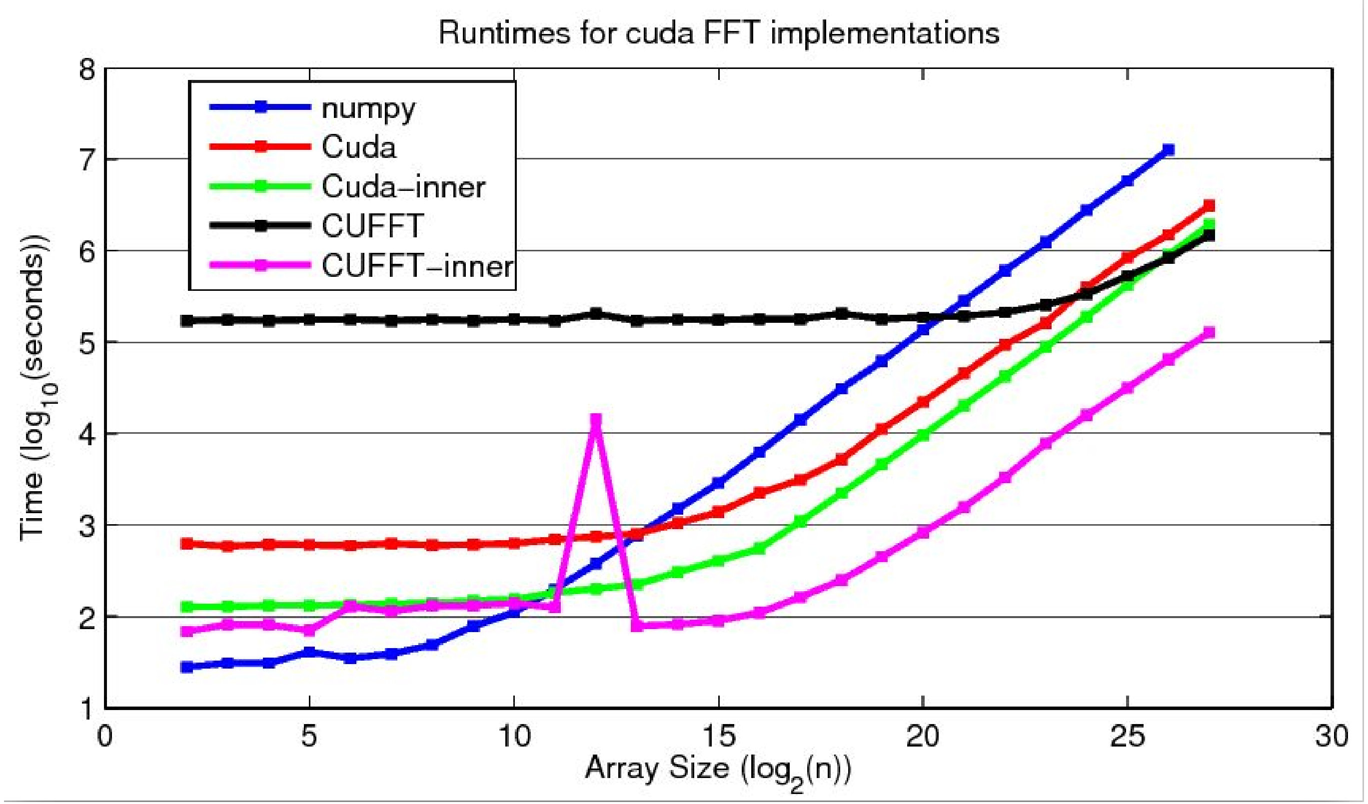
Cilk Memory Usage

- Cilk iterative solution uses the same amount of heap space as the C++ solution
  - This might indicate that cilk is using vectorization under the covers with the cilk_for() construct
- Cilk recursive solution uses 1.4x more heap space than the C++ solution
  - For test size of 2^{12} it uses ~40MB of heap space vs. ~28MB
- Cilk manipulates the stack, so any stack tracing/profiling induces the "Heisenberg Uncertainty Principle"
  - Act of measuring changes the result

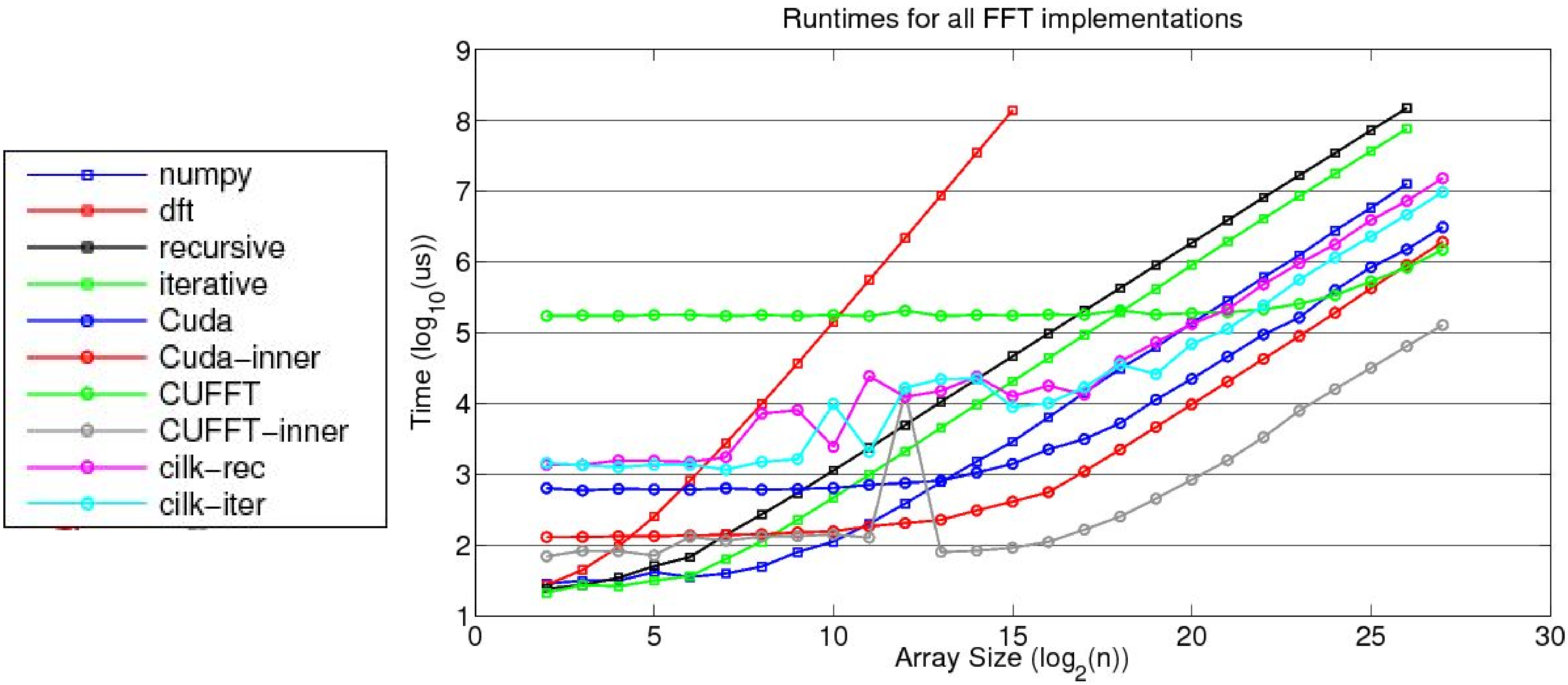
CUDA: Iterative FFT

- Load input data from CPU to GPU
- call `bit_reverse_kernel()` to perform array index bit-reversal in global memory
- call `fft_kernel_shared()` to perform the partial FFT using shared memory
- loop over `fft_kernel_finish()` to finish the FFT in global memory
- Retrieve data from GPU memory to CPU

CUDA Performance



Overall Performance



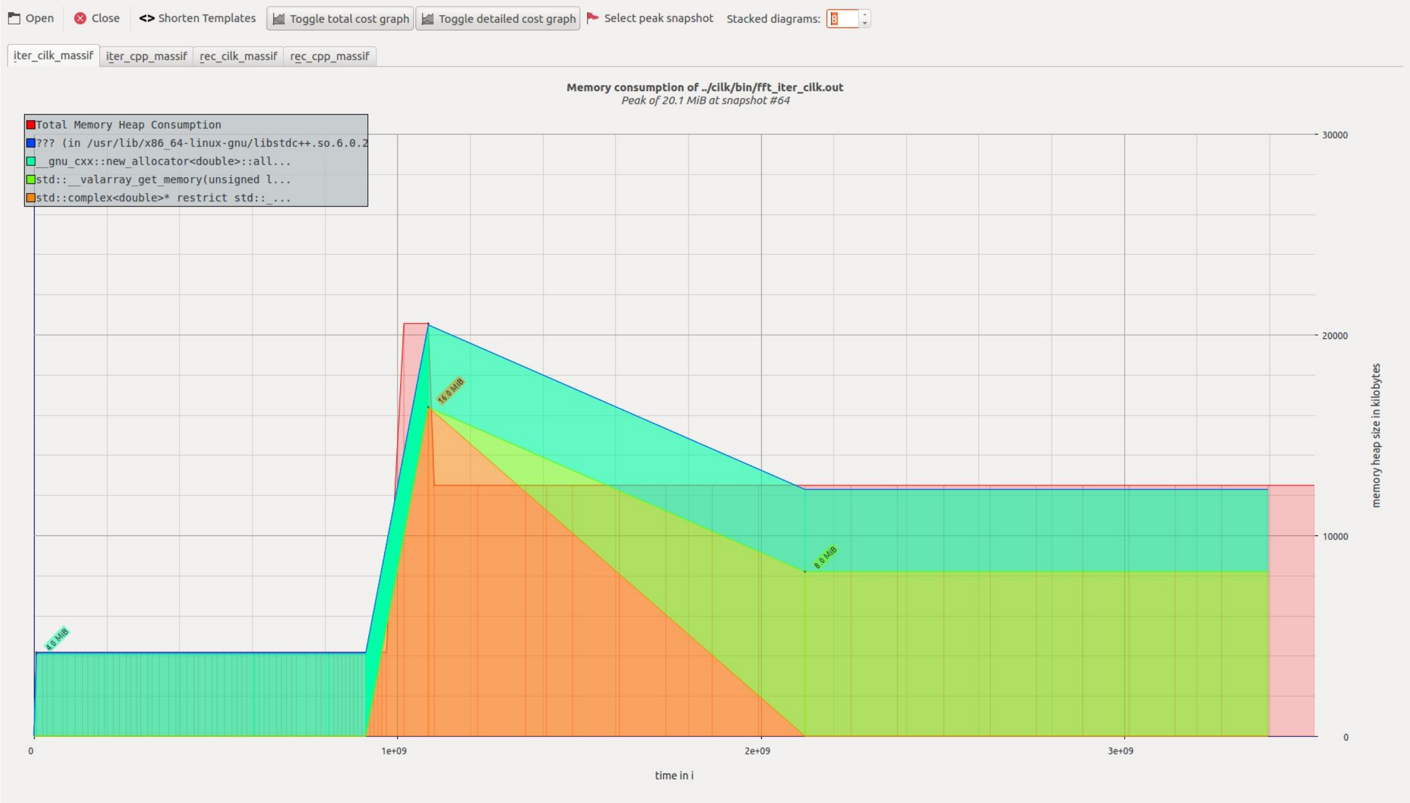
Conclusion

- $O(n^2)$ Sucks!
- Recursive Solution is better
    - Modern processors do not optimize well for it
- Iterative Solution is best
- Cilk Parallelism is expensive for small input vectors but scales well
    - Easy in terms of programming
- CUDA is blazing fast, but suffers from memory latency
- CuFFT Algorithm is fast but spends a lot of time “setting up the plan”

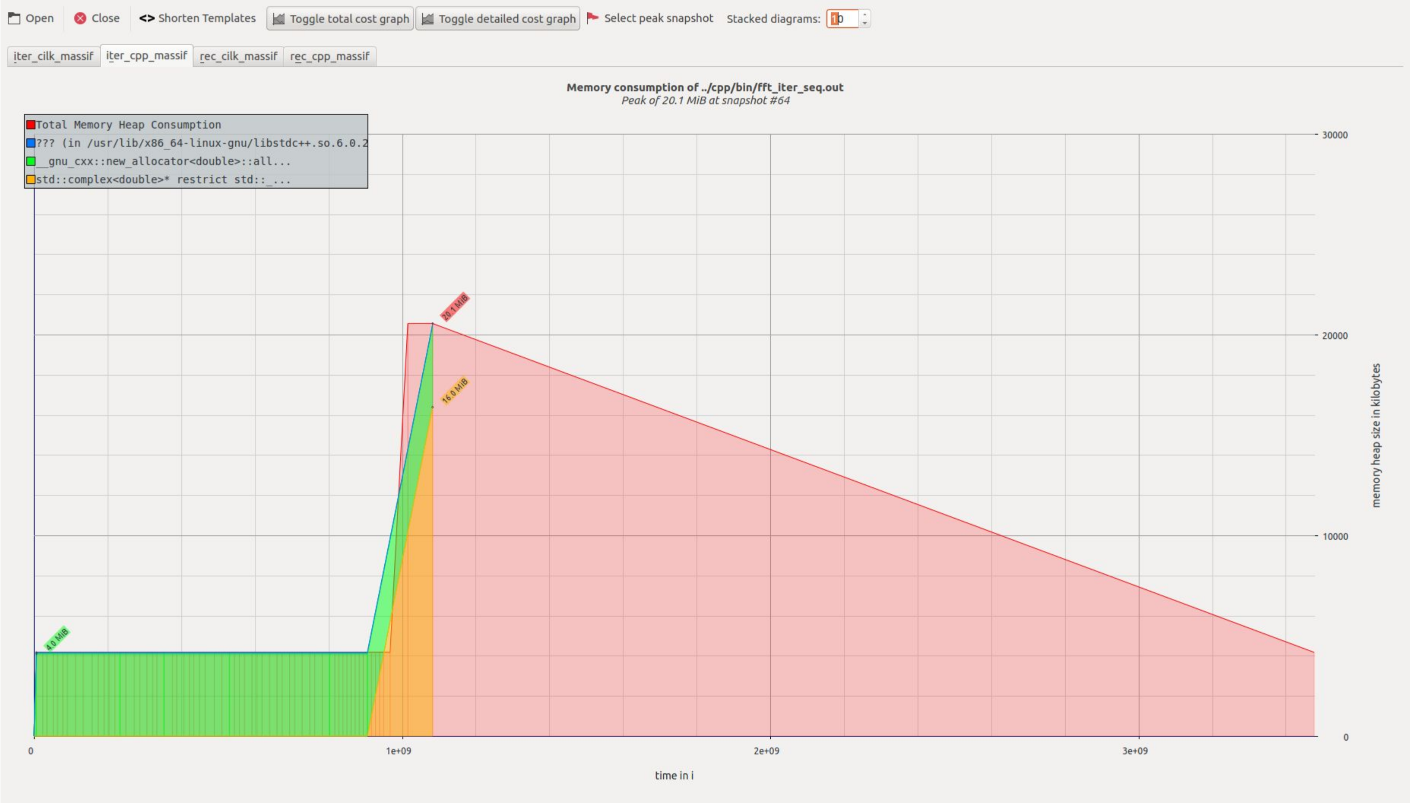
Q&A

Backup Slides

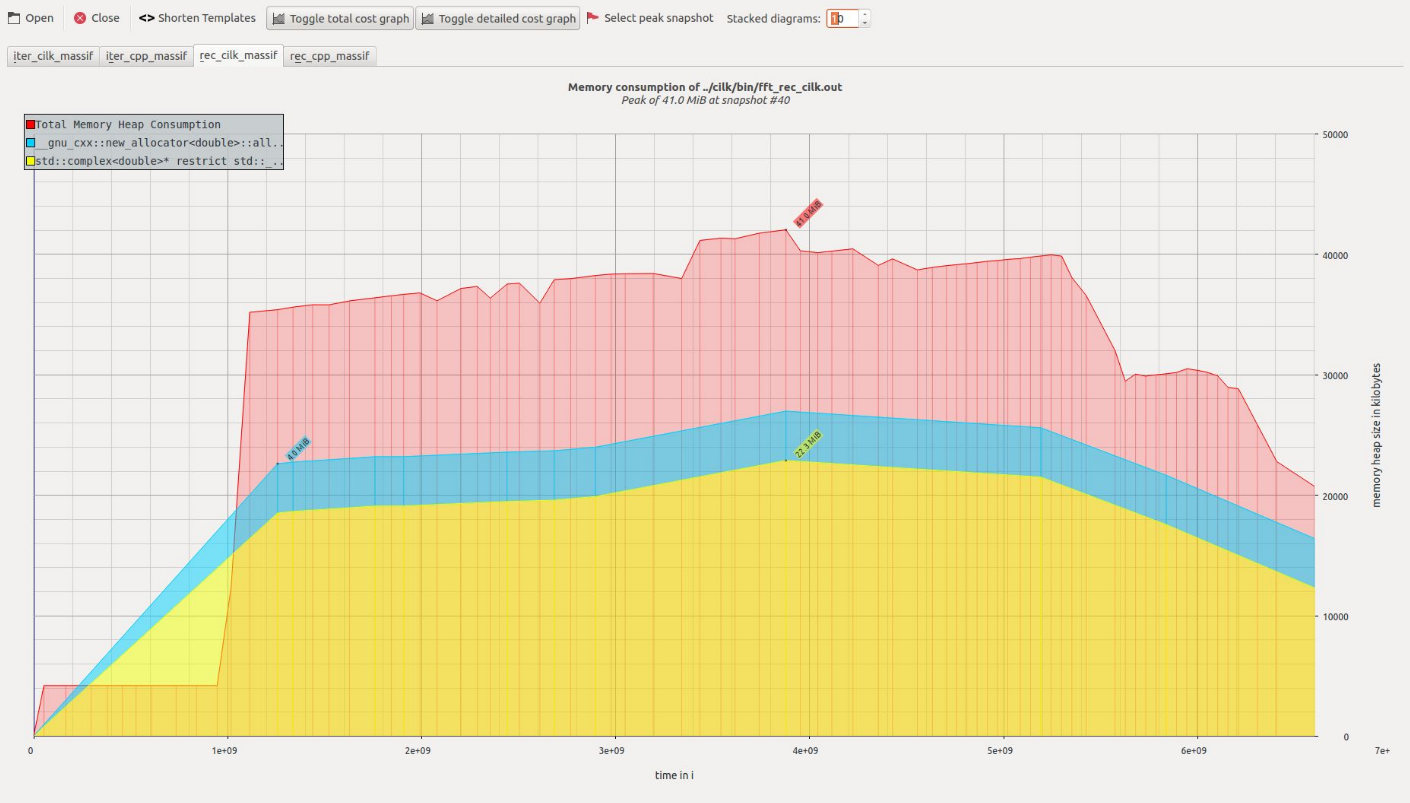
Cilk Iterative FFT Heap Usage



C++ Iterative FFT Heap Usage



Cilk Recursive FFT Heap Usage



C++ Recursive FFT Heap Usage