Fast Fourier Methods in HPC

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2013

Fourier Methods

Sometimes called “spectral methods,” Fourier analysis has many applications in a diverse set of fields:

- Optics
- Crystallography
- Acoustics
- Seismology
- Imaging (all areas)
- Broad range of engineering & scientific disciplines

Periodicity

Generally Fourier methods are applied in the **time** and **frequency** domains, but any periodicity will do (e.g. **real-space** and **reciprocal-space** in crystallography).

- $x(t)$ will be our representative function to be sampled, e.g. in the time domain
- $X(f)$ will be its **transformed** equivalent, in the frequency, f, domain.

Fourier Series

We can represent our periodic function $x(t)$ as linear combination of sines and cosines (making use of their orthogonality):

$$x(t) = \frac{a_0}{2} + \sum_{j=1}^{\infty} \left[a_j \cos\left(\frac{2\pi jt}{T}\right) + b_j \sin\left(\frac{2\pi jt}{T}\right) \right],$$

where T is the period, or

$$x(t) = \sum_{j=-\infty}^{\infty} X_j e^{2\pi ijt/T}$$

and X_j is the $j-th$ Fourier coefficient of $x(t)$.

Fourier Transforms

The Fourier transform is a generalization of the Fourier series (take the limit that $T \to \infty$, and $t/T \to dt$,

$$x(t) = \int_{-\infty}^{\infty} X(f) e^{2\pi i f t} df,$$

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-2\pi i f t} dt,$$

and $X(f)$ is the **spectrum**, or Fourier transform (FT) of $x(t)$.

FT Symmetries

We can make use of mathematical symmetries to simplify the calculation of FTs:

- $x(t)$ real, $X(-f) = [X(f)]^*$
- $x(t)$ imaginary, $X(-f) = -[X(f)]^*$
- $x(t)$ even, $X(-f) = X(f)$
- $x(t)$ odd, $X(-f) = -X(f)$
- $x(t)$ real and even, $X(f)$ also real and even
- $x(t)$ real and odd, $X(f)$ imaginary and odd
- $x(t)$ imaginary and even, $X(f)$ imaginary and even
- $x(t)$ imaginary and odd, $X(f)$ real and odd

Discretization

Now we consider the case where we **sample** our function, i.e. we have a collection of N sample points:

$$x(t) = \{x_0, x_1, \ldots, x_{N-1}\}$$

and then we have a **discrete Fourier transform** (DFT),

$$X_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j e^{-2\pi i (jk/N)},$$

and its inverse:

$$x_k = \sum_{j=0}^{N-1} X_j e^{2\pi i (jk/N)}.$$

Note that there is not much of a consensus on where the N factors go ...

DFT Properties

- Scale factor, $1/N$, often omitted (or $1/\sqrt{N}$ used in each transform
- Some use positive coefficients in transform, negative in the inverse (or vice-versa)
- Serial time complexity: $\mathcal{O}(N^2)$ multiplications and additions (and this is for 1D!)

More DFT Technicalities

Given a uniform sampling interval, $x_i = i\Delta$, the Nyquist critical frequency is $1/2\Delta$:

- Basically 2 points per cycle (peak and trough of a sinusoid)
- Sampling theorem - if our continuous function $x(t)$ is bandwidth-limited, i.e. all of its frequencies are less than the Nyquist critical frequency ($X(f) = 0$ for $f > 1/2\Delta$), then $x(t)$ is completely determined by its samples, x_i.
- If $x(t)$ is not bandwidth-limited, the power spectrum for higher frequencies gets shifted (or aliased) into the range $[-1/2\Delta, 1/2\Delta]$. Filtering is your friend in this case (as is close monitoring of the Fourier transform as the critical frequency is approached).

DFTs in Parallel

Recall that our expression for a DFT (ignoring the scale factor),

$$X_k = \sum_{j=0}^{N-1} x_j e^{-2\pi i(jk/N)},$$

if we define the **twiddle factor**, $w = e^{-2\pi i/N}$,

$$X_k = \sum_{j=0}^{N-1} x_j w^{jk},$$

and for the inverse transform we simply take $w \rightarrow w^{-1}$.

Serial Pseudo-code

Pseudo-code for the DFT might look something like:

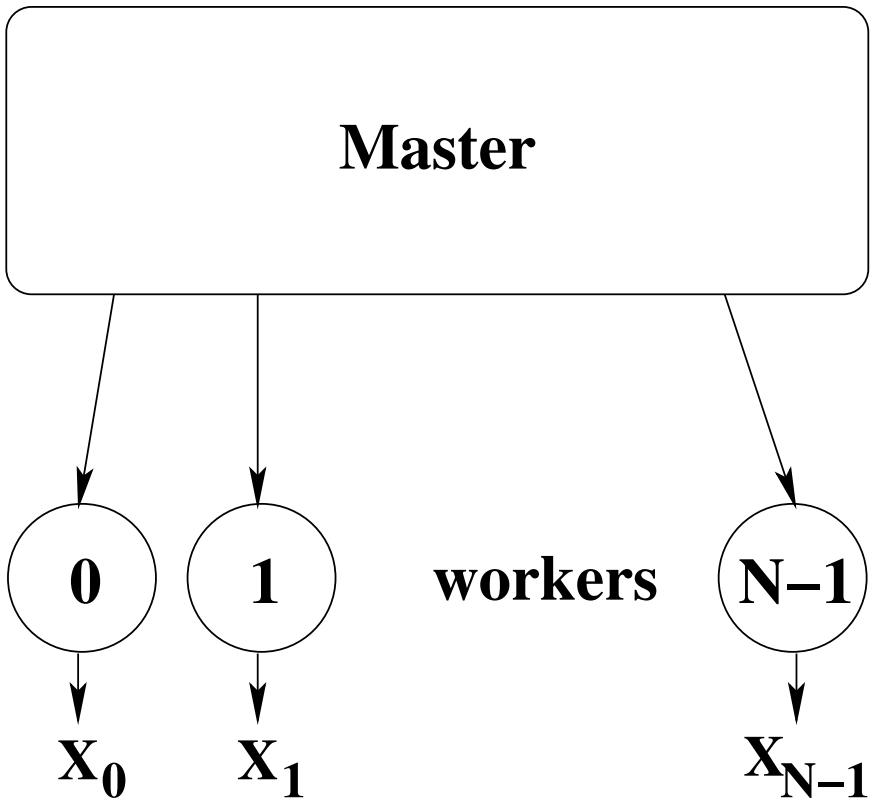
```
do k=0,N-1
   do j=0,N-1
      X(k) = X(k) + x(j)*( w**(j*k) )
   end do
end do
```

or more efficiently:

```
do k=0,N-1
   a = 1
   do j=0,N-1
      X(k) = X(k) + x(j)*a
      a = a*(w**k)
   end do
end do
```

Parallel DFT - Master/Worker

One method of parallelization of the DFT is to use a master-worker implementation, illustrated below:



Analysis

In this master-worker scheme for the DFT:

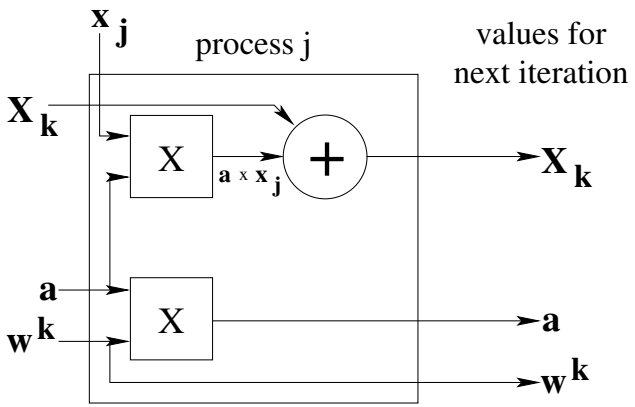
- Master process can pre-compute a (twiddle) factors
- All workers need copies of $\{x_j\}$, which increases memory requirements
- Cost is $\mathcal{O}(N)$ for N processes, but typically N greatly exceeds the number of available processes

Parallel DFT - Pipeline

For pipelining, we can unfold the innermost loop in the calculation of the DFT,

```
X(k)=0
a=1
X(k)=X(k)+a*x(0)
a=a*(w**k)
X(k)=X(k)+a*x(1)
...
```

and each step of the pipeline would look something like:



DFT as a Matrix-Vector Product

Given that each transform value X_k can be written as a vector in terms of the x_j:

$$X_k = x_0 w^0 + x_1 w^1 + \ldots + x_{N-1} w^{N-1},$$

from which we can pull a matrix equation:

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ \vdots \\ X_{N-1} \end{pmatrix} = \frac{1}{N} \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & w & w^2 & \ldots & w^{N-1} \\ 1 & w^2 & w^3 & \ldots & w^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{N-1} & w^{2(N-1)} & \ldots & w^{(N-1)(N-1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{pmatrix}$$

and then subject that matrix to all of parallel tools for solving linear systems. That would be $\mathcal{O}(N^2)$, however, and we can do quite a bit better than that.

Multidimensional DFTs

A two-dimensional DFT is simple to write:

$$X_{lm} = \sum_{j=0}^{N-1} \sum_{k=0}^{M-1} x_{jk} e^{-2\pi i(jl/N + km/M)},$$

simple rearrangement leads to

$$X_{lm} = \sum_{j=0}^{N-1} \left[\sum_{k=0}^{M-1} x_{jk} e^{-2\pi ikm/M} \right] e^{-2\pi ijl/N},$$

and we see that we have a 1D DFT operating on a row to produce a transformed row, then similarly for the columns. The rows can be calculated independently of each other, as can the columns. Very high degree of parallelism (as is 3D).

Fast Fourier Transforms

The method of **fast Fourier transforms (FFTs)** is by far the most widely used technique in Fourier methods. Origins can be traced to the following paper:

- Cooley and Tukey, "An Algorithm for Machine Computation of Complex Fourier Series," Math. Comp. **19**, 297-301 (1965).

but, in fact, the method is quite a bit older than that - see *Numerical Recipes* for further references.

FFT Serial Cost

The idea behind FFTs is to reduce the serial cost of a DFT to $\mathcal{O}(N \log N)$ rather than $\mathcal{O}(N^2)$. Unfortunately there are quite a few formulations for FFTs - the following reference describes 8!:

- P. N. Swarztrauber, "Multiprocessor FFTs," Parallel Comp. **5**, 197-210 (1987).

FFT Formulation

In the following we will make the assumption that N is a power of two. Recall our DFT expression:

$$X_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j w^{jk}, \qquad w = e^{-2\pi i/N}.$$

and now we take a general divide-and-conquer approach ...

FFT: Divide and Conquer

Our first step is to break up the DFT into two sums,

$$X_k = \frac{1}{N}\left[\sum_{j=0}^{N/2-1} x_{2j} w^{2jk} + \sum_{j=0}^{N/2-1} x_{2j+1} w^{(2j+1)k}\right]$$

$$= \frac{1}{2}\left[\frac{2}{N}\sum_{j=0}^{N/2-1} x_{2j} w^{2jk} + w^k \frac{2}{N}\sum_{j=0}^{N/2-1} x_{2j+1} w^{2jk}\right]$$

but if we take $k \in [0, N/2-1]$ this is just the sum of two $N/2$-point DFTs,

$$X_k = \frac{1}{2}\left[X_{even} + w^k X_{odd}\right], \qquad k \in [0, N/2-1]$$

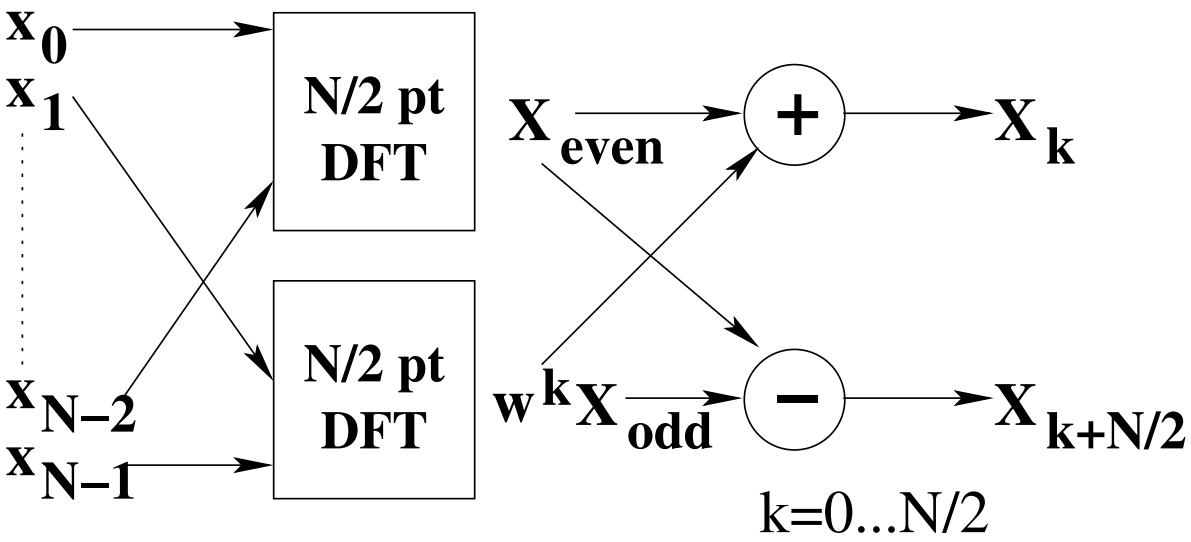
and similarly for $k \in [N/2, N-1]$:

$$\begin{aligned} X_{k+N/2} &= \frac{1}{2}\left[X_{even} + w^{k+N/2} X_{odd}\right], \\ &= \frac{1}{2}\left[X_k^{even} - w^k X_k^{odd}\right], \end{aligned}$$

since $w^{N/2} = e^{-\pi i} = -1$.

FFT: Divide and Conquer, Illustrated

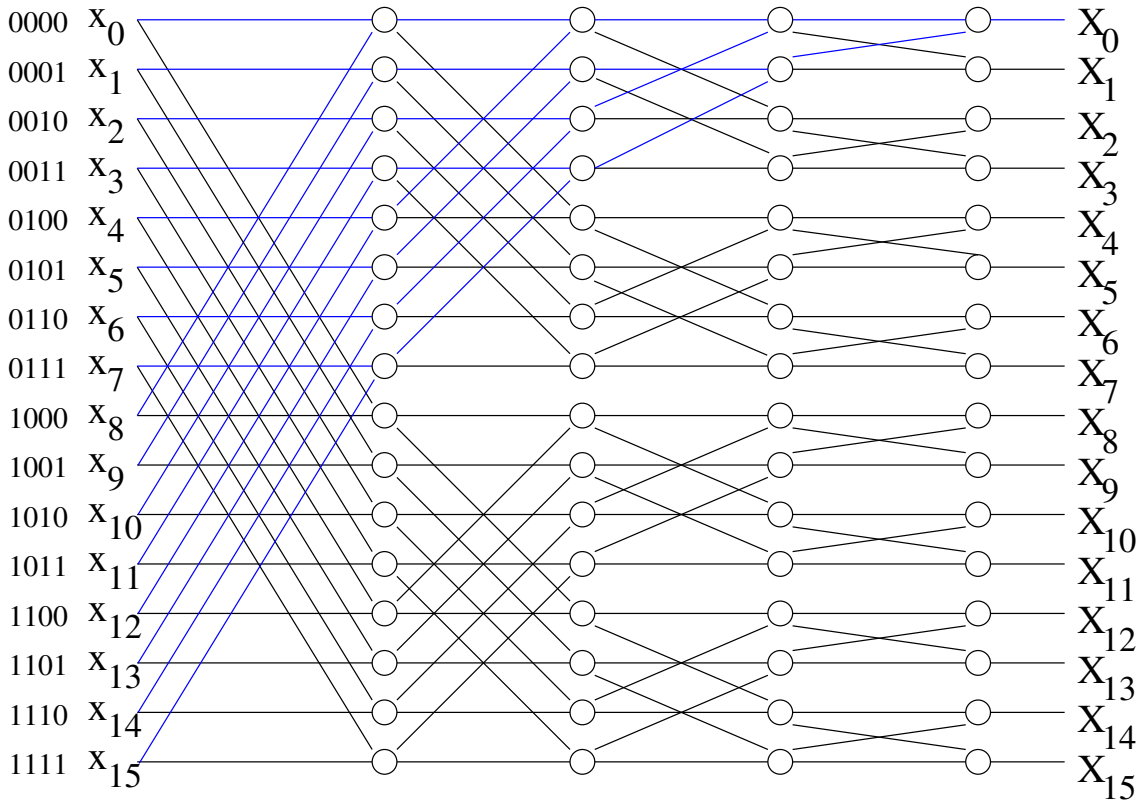
We can illustrate this divide and conquer approach as follows:



and this can be done repeatedly until there is just one element in the sum.

FFT: Divide and Conquer, Example

"Butterfly" diagram:



(16-point FFT. Note the binary order of the input data values x_j, $\log_2 16 = 4$ stages, twiddle and sign factors omitted for clarity, higlighted in blue is D&C computation of X_0)

FFT Serial Time

The time complexity for FFT:

- Can be implemented recursively or iteratively
- Cost is $\mathcal{O}(N \log N)$ (N computations at each of $\log N$ stages)
- Note the *bit-reversal* ordering (i.e. x_0 and x_8 are combined first, not x_0 and x_1; x_1 and x_9 are combined first, not x_1 and x_2, etc.)

Only Powers of 2?

Are FFTs limited to powers of 2? No, there are other **radices** (the fundamental prime factorization of N) available, usually small integers less than 10.

- Radix-4 is a popular choice, as the twiddle factors are all $1, -1, i$, or $-i$.
- Mixed-radix FFTs can be done - typically using prime factorization and then FFTs on the prime radix values (e.g. $1000 = 2^3 \times 5^3$, needing 6 stages, or $1000 = 10^3$ in 3 stages of radix-10.

More FFT Goodness

Some other FFT goodies:

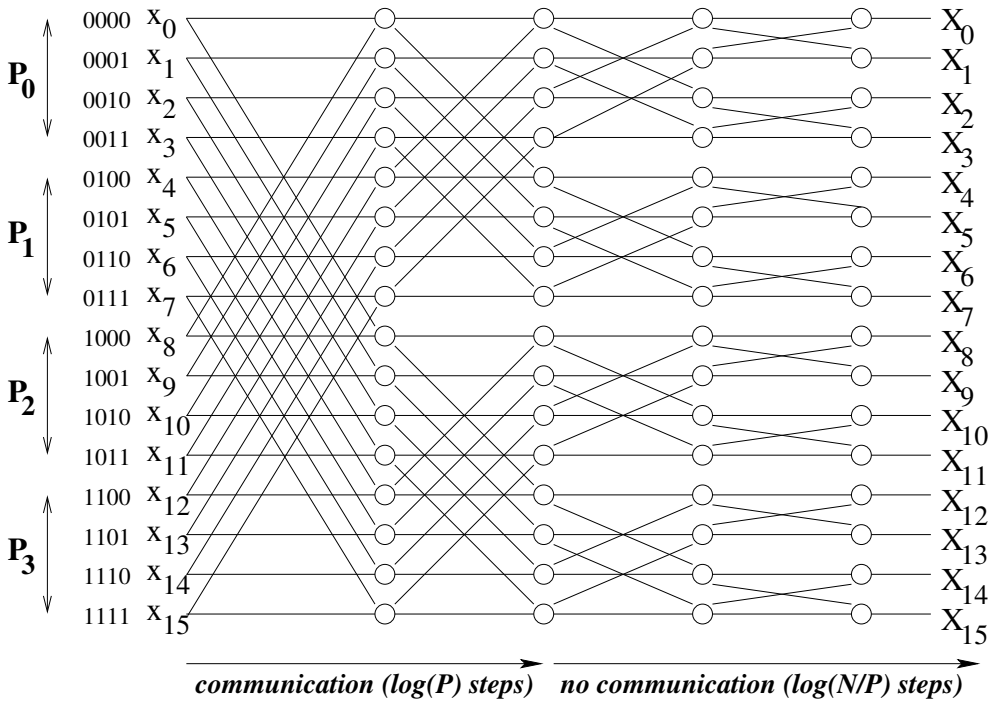
- Specialized hardware for bit-reversal, which is slow in software (but can be very fast in specialized hardware, like FPGAs) - almost all digital signal processors (DSPs) have specialized hardware for bit-reversal
- Implementations - there are optimized FFT implementations already in the wild, so you will seldom code one yourself, except as a learning exercise

Binary Exchange Algorithm

In our previous example of the 16-point FFT, suppose that we have one processor for each data point (i.e. $N_p = 16$).

- Results at each stage get passed to processes whose binary address differ by only a single bit (e.g. processor 0 communicates with processor 8 in the first step, then 4, then 2, then 1 in the last step
- Maps very well onto a hypercube
- Even if $N > p$ communications have the same pattern ...

Binary Exchange Illustrated



Note that the most significant $\log N_p$ bits represent the process number, while the remaining bits describe the data point within the group.

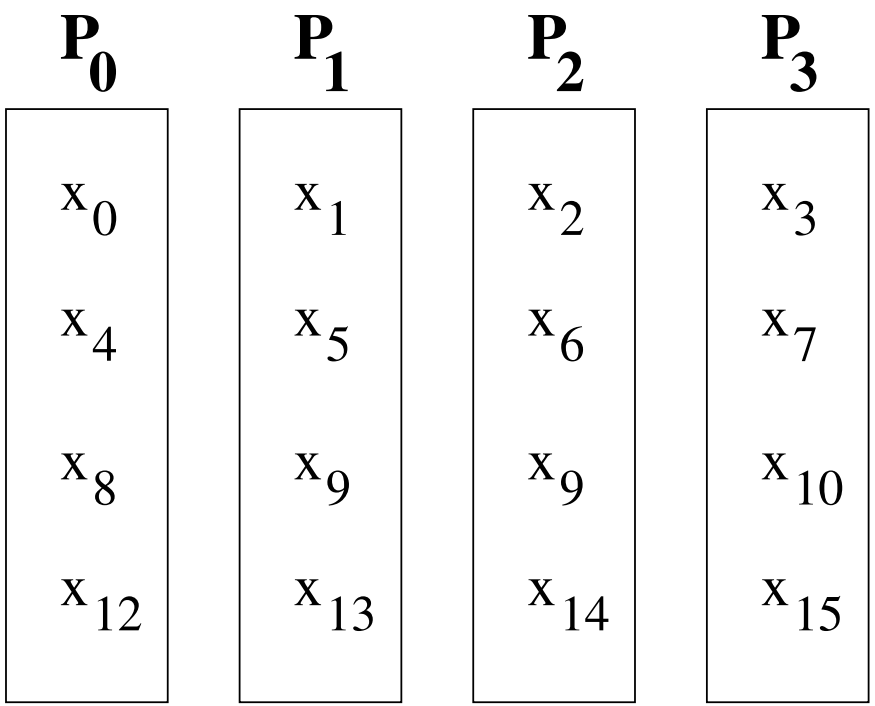
Binary Exchange Costs

Costs for binary exchange:

- Computations: $\tau_{comp} = \mathcal{O}(N \log N)$
- Communication: if $N_p = N$, we have 1 data exchange between pairs at each step, so $\tau_{comm} = \mathcal{O}(\log N)$ assuming simultaneous exchange of data at each step
- Communication: if $N_p < N$, $\tau_{comm} = \mathcal{O}(\log N_p)$ assuming simultaneous exchange of data at each step

Transpose Algorithm

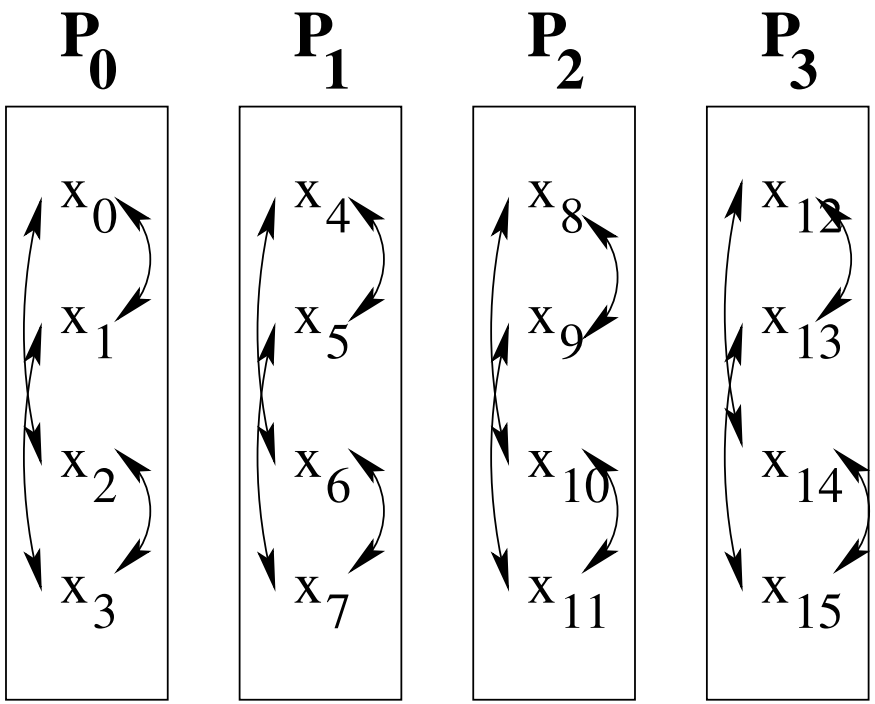
Best illustrated through example - consider our 16-point FFT with the data laid out in row order in a 2D array:



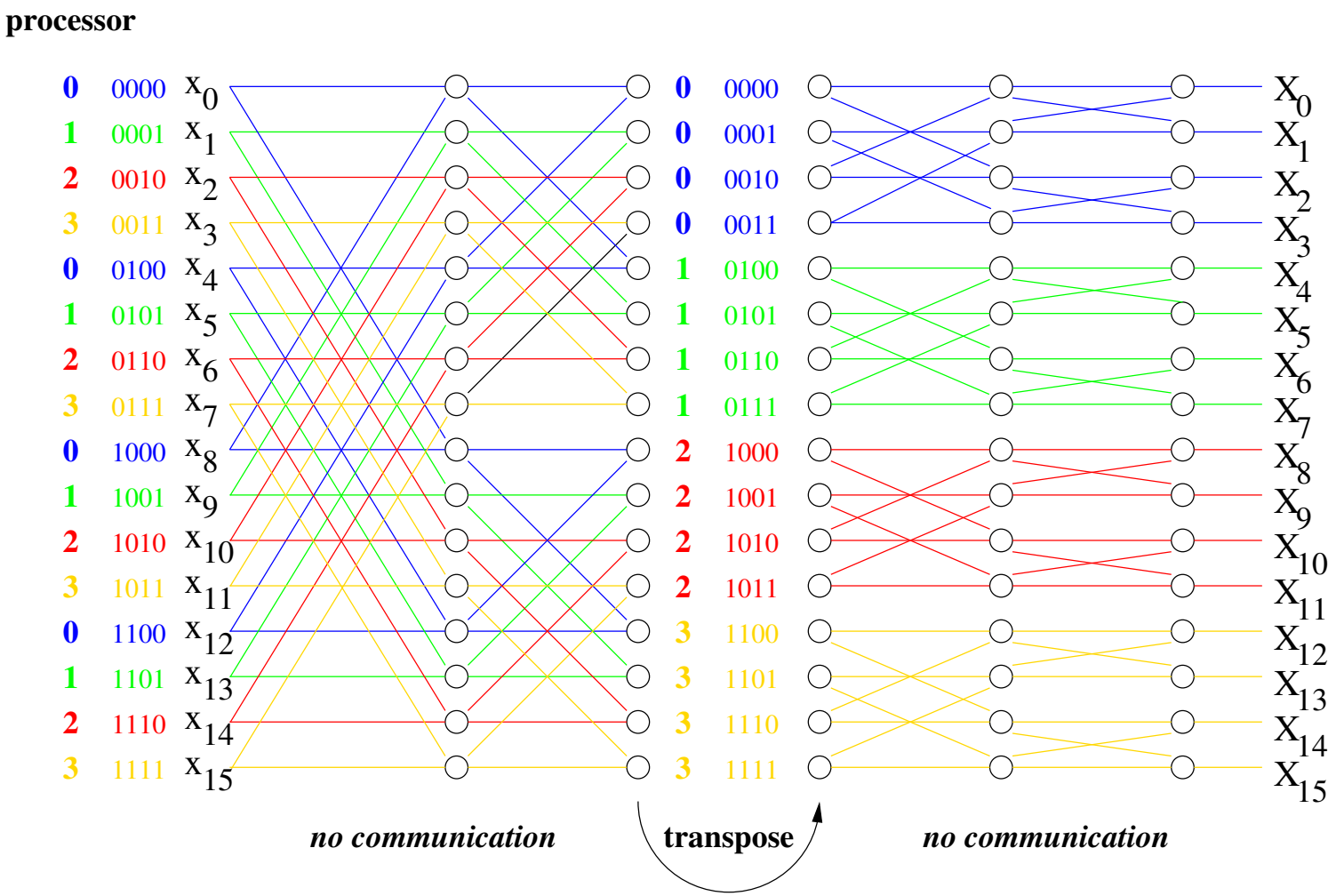
Steps for transpose method for our 16-point FFT:

- Apply 1st two steps of FFT algorithm, which occur within processor
- Transpose matrix of values
- Apply last 2 steps of FFT algorithm, which after transposition also occur entirely within processor

Illustration (hopefully!) of 16-point FFT using transpose algorithm (animated in non-handout version)



Or, perhaps more helpfully, a butterfly diagram of the N=16, 4-process FFT with transpose:



Poisson, Meet Fourier

We can take our 2D Poisson equation example,

$$\nabla^2 u(x, y) = \rho(x, y),$$

discretize as before, and for Dirichlet boundary conditions express both sides as Fourier Series:

$$u_{ij} = \sum_i \sum_j u_{ij} \sin(\pi i x) \sin(\pi j y),$$

where

$$u_{ij} = \frac{2}{N}\frac{2}{M} \sum_{n=1}^{N-1} \sum_{m=1}^{M-1} \hat{U}_{mn} \sin\left(\frac{\pi i m}{M}\right) \sin\left(\frac{\pi j n}{N}\right)$$

Similarly for ρ.

The resulting equation for the Fourier coefficients is:

$$\hat{U}_{mn} = \frac{\hat{\rho}_{mn} h^2}{2\left(\cos\frac{\pi m}{M} + \cos\frac{\pi n}{N} - 2\right)}$$

and the general solution method is:

- Compute $\hat{\rho}_{mn}$ as the Fourier Transform (sine transform with these boundary conditions)
- Compute $\hat{U}_{mn}$ as above
- Compute u_{ij} by inverse Fourier Transform (inverse sine transform in this case)

FFT Solution Characteristics

General aspects of solving boundary value problems with FFTs:

- Complexity goes as $\mathcal{O}(N \log N)$ (bit worse than multigrid), communication costs as $\mathcal{O}(\log N)$ (bit better than multigrid)
- Can be generalized to inhomogeneous boundary conditions (add to right-hand side), cosine expansion for Neumann conditions, etc.
- Non-constant coefficients through, e.g. *cyclic reduction* or *Fourier analysis + cyclic reduction (FACR)*, c.f. *Numerical Recipes*.

FFT Libraries

Few people implement their own FFTs - in this section we will discuss some optimized libraries that have already been developed for use on various chip architectures.

FFTW

FFTW, also known as the Fastest Fourier Transform in the West:

- www.fftw.org
- Open-source (GPL) from MIT (M. Frigo and S. G. Johnson)
- C subroutine library for DFTs in one or more dimensions
- Arbitrary input size, multiple strided transforms
- Real/Complex data
- Competitive with vendor-tuned FFTs

FFTW (cont'd)

More FFTW properties:

- High-speed (supports SSE/SSE2/SSE3/3DNow vectorizations on Intel/AMD chips), also AVX for Intel vector extensions (versions >=3.3), and a `FORTRAN 2003` interface (versions >=3.3)
- Portable to new platforms (just add C compiler)
- C/Fortran interfaces
- Parallel - both multithreaded and MPI (MPI in version 3 requires version >=3.3)
- Intel MKL has FFTW "wrappers" around its DFTs to allow easy use of MKL for FFTW-based code (both multi-threaded and MPI-based DFTs).

Where does FFTW get its speed from?

- Uses a large variety of algorithms internally
- At compile time it *measures* the performance of various algorithms (and underlying parameters) on different sizes for the given system architecture - basically it can tune itself to your hardware
- Winner of 1999 J. H. Wilkinson Prize for Numerical Software
- Used as the FFT code in MATLAB

Simple Example in FFTW

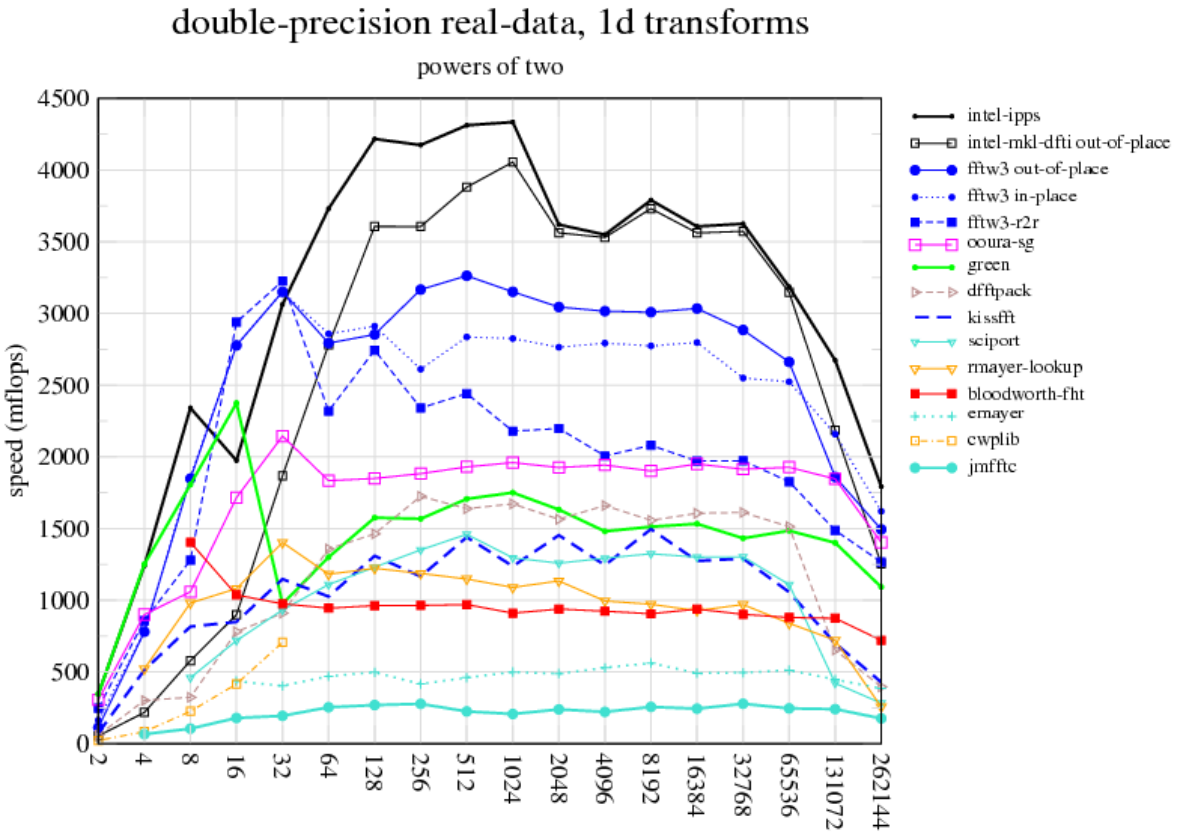
Simple FFTW code fragment for 1D complex transform:

```
#include <fftw3.h>
 ...
{
   fftw_complex *in, *out;
   fftw_plan p;
   ...
   in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
   out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
   p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
   ...
   fftw_execute(p); /* repeat as needed */
   ...
   fftw_destroy_plan(p);
   fftw_free(in);
   fftw_free(out);
}
```

`FFTW_ESTIMATE` (builds plan, probably sub-optimal) versus `FFTW_MEASURE` (run and measure several FFTs to determine optimal plan for requested FFT size, typically takes at least a few seconds).

FFTW Benchmark Results (power of 2)

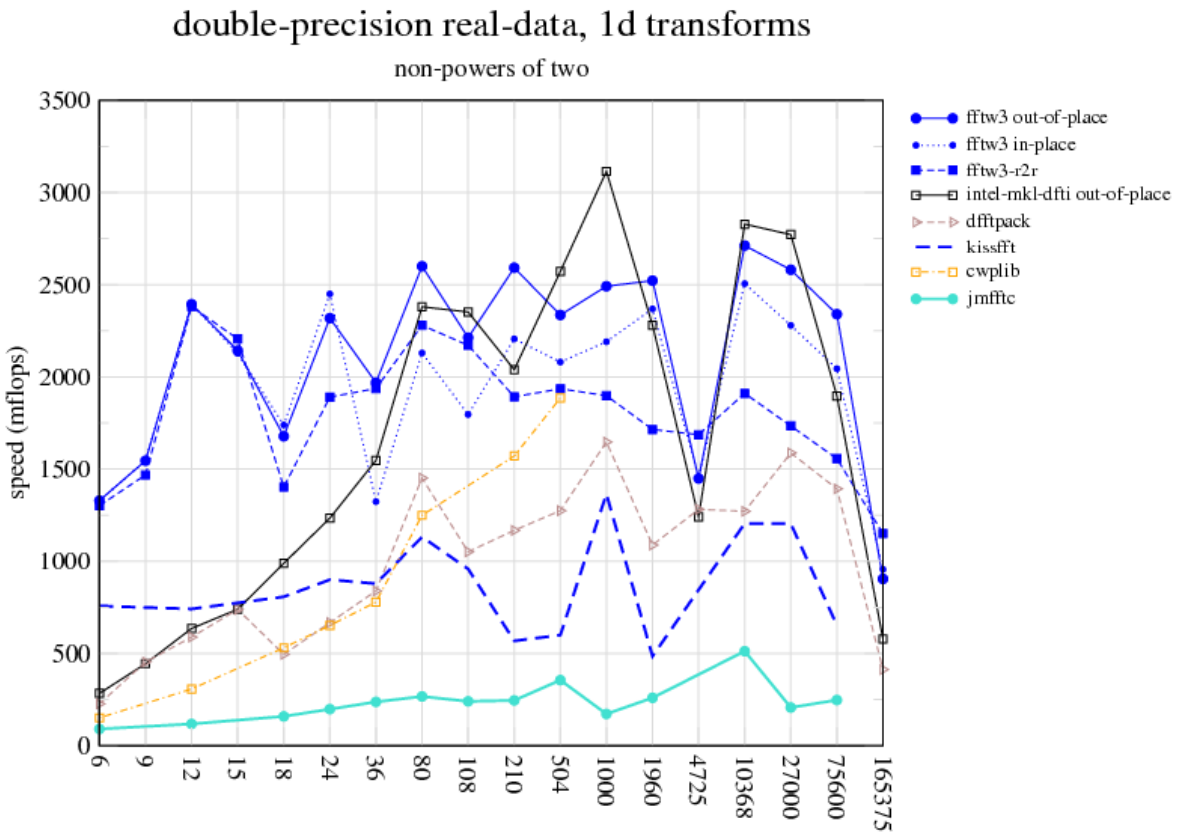
3.6GHz Intel Xeon (7.2GFlop/s TPP)



N.B., in-place refers to overwriting the input arrays with the output, hence saving on auxiliary storage.

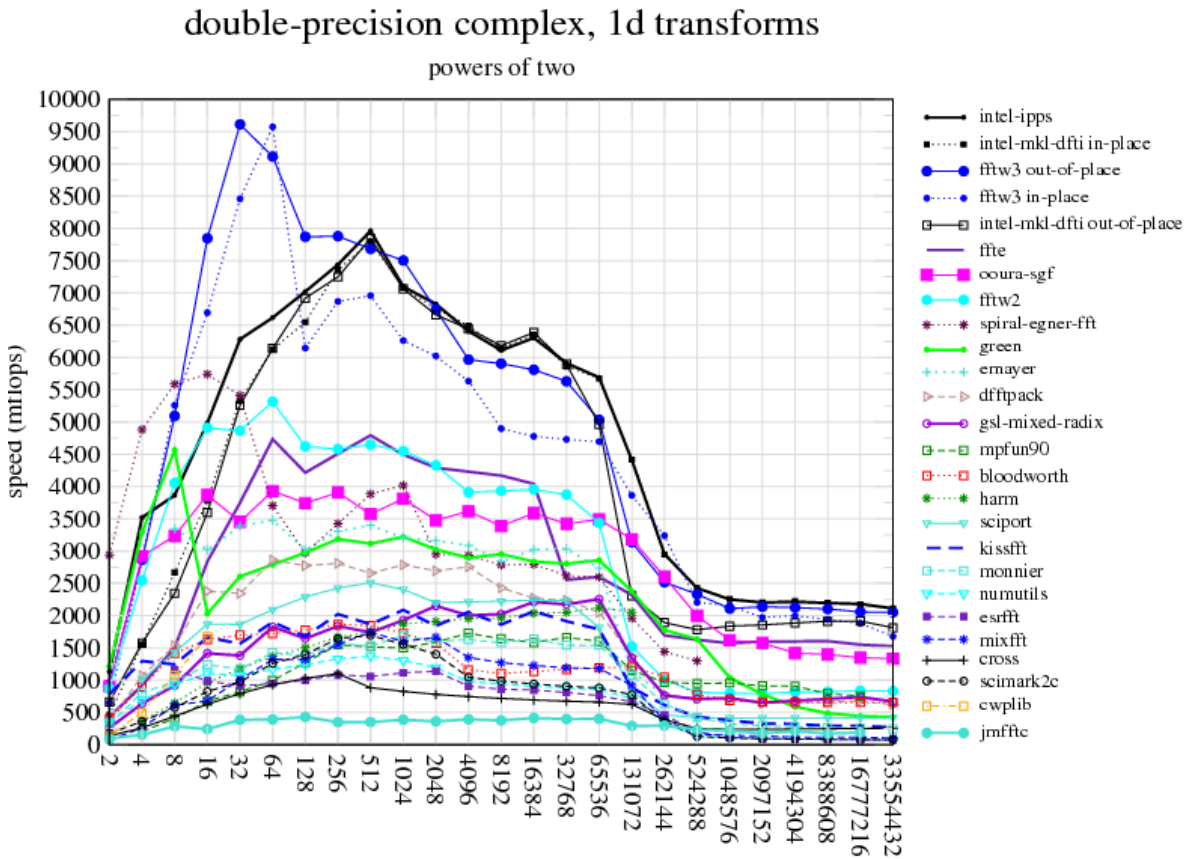
FFTW Benchmark Results (non-power of 2)

3.6GHz Intel Xeon (7.2GFlop/s TPP)



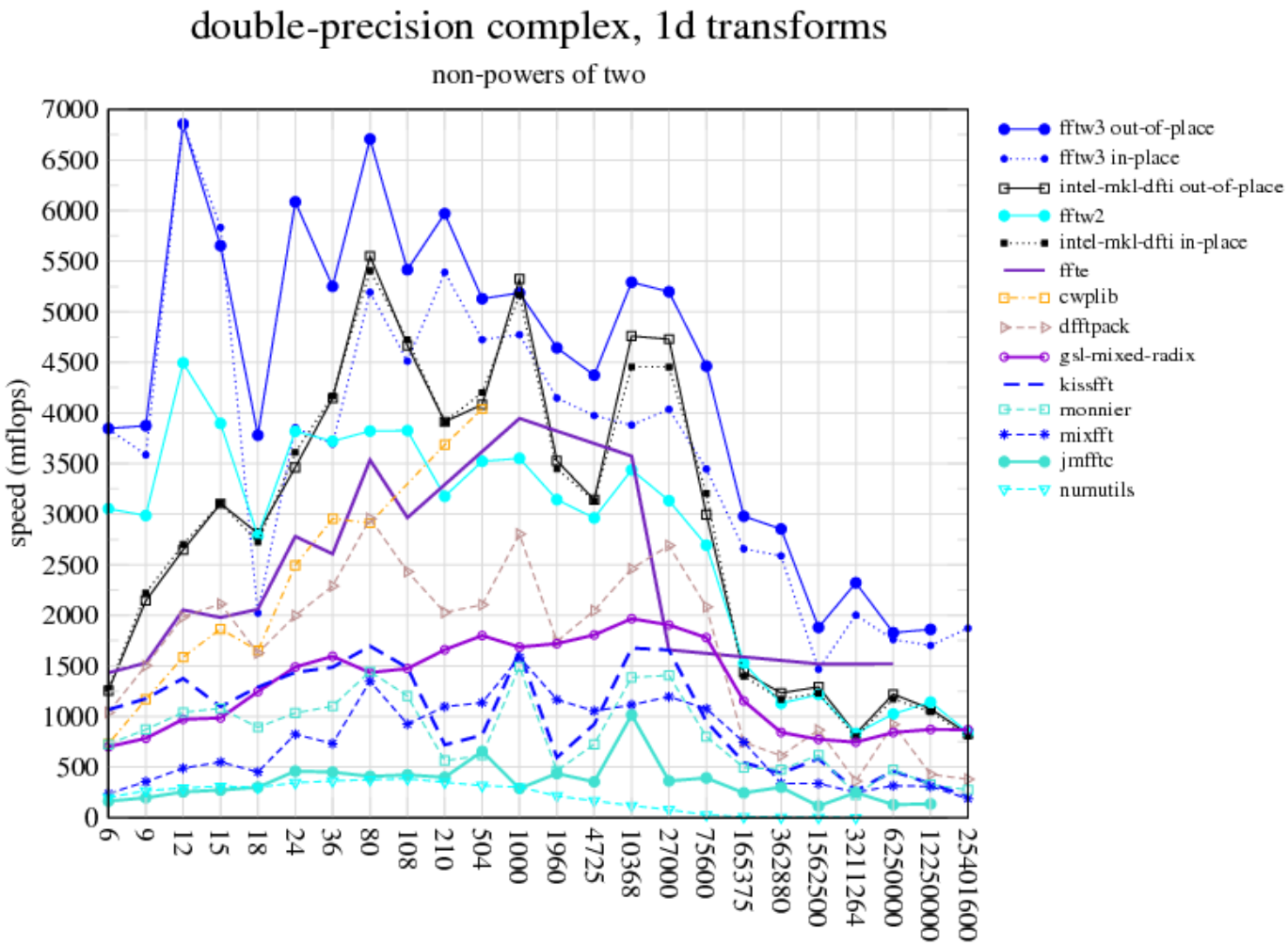
FFTW Benchmark Results (power of 2)

3.0GHz Intel Core2 Duo (12GFlop/s TPP)

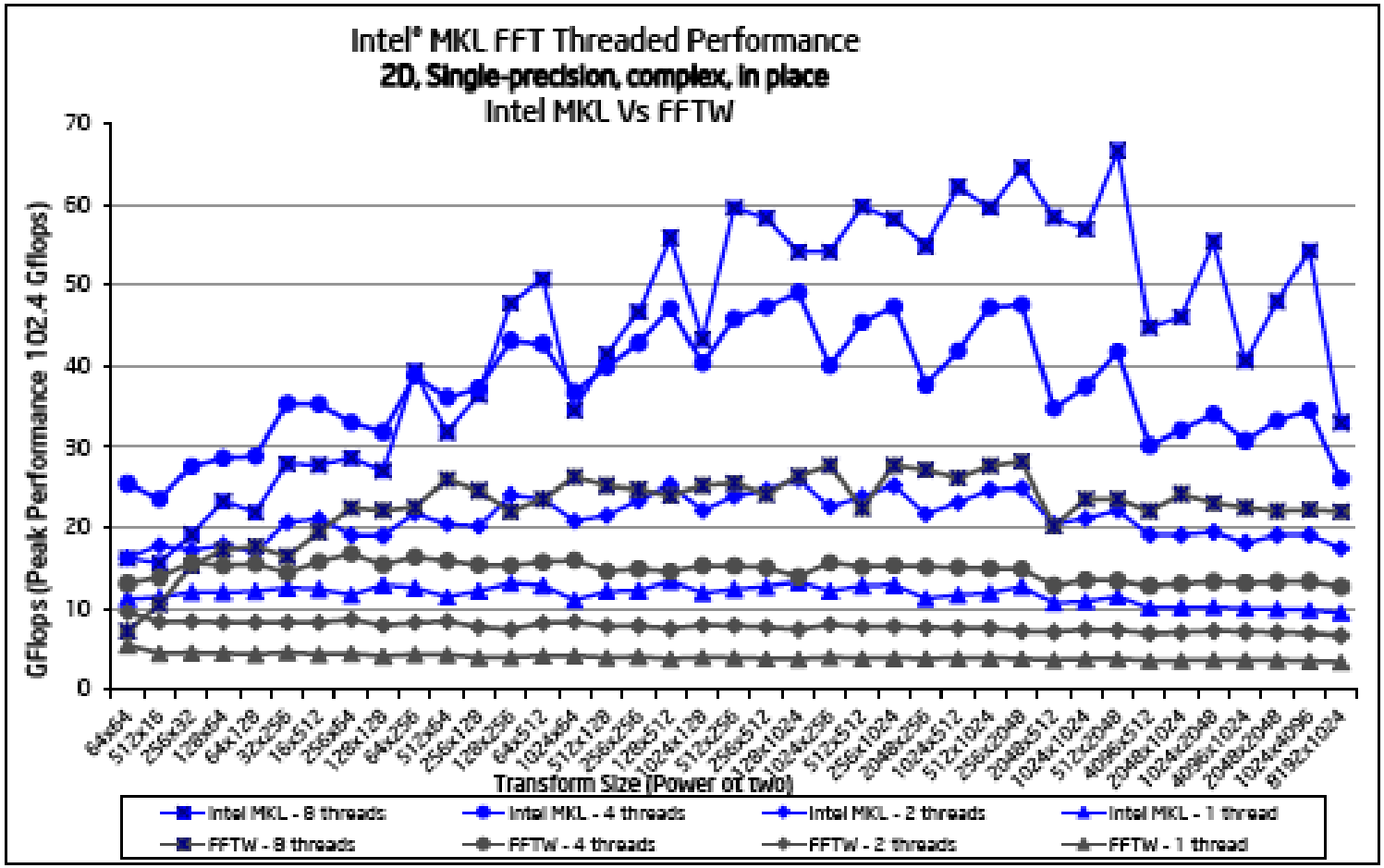


FFTW Benchmark Results (non-power of 2)

3.0GHz Intel Core2 Duo (12GFlop/s TPP)



FFTW vs. Intel MKL



Configuration Info

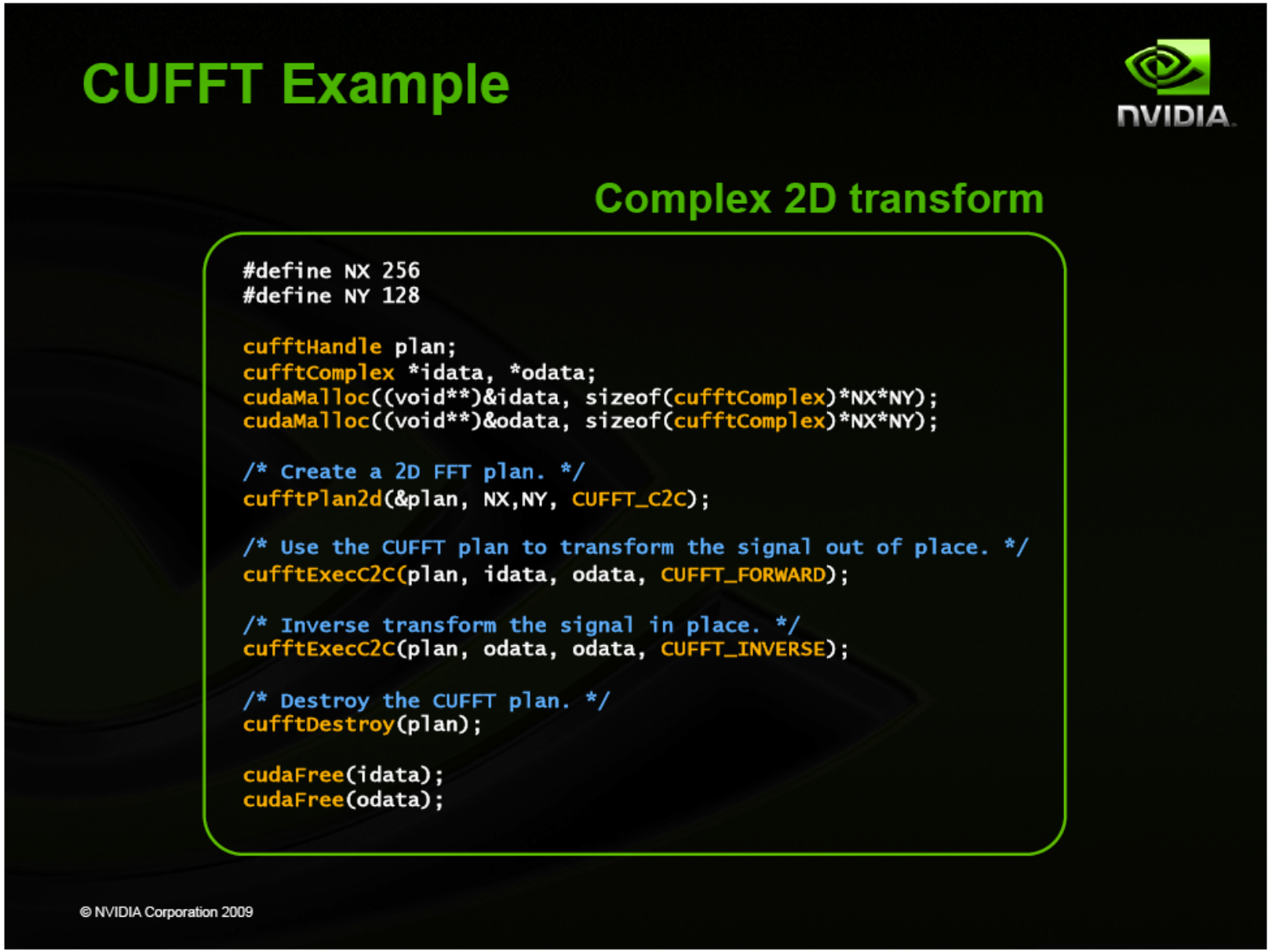
- Versions: Intel® MKL 10.2 FFTW3.2.1
- Hardware: Quad-Core Intel® Xeon® Processor W5570 3.2Ghz 8MB L2 cache 12GB Memory
- OS: Fedora 10 x86_64

CUFFT, CUDA FFT Library

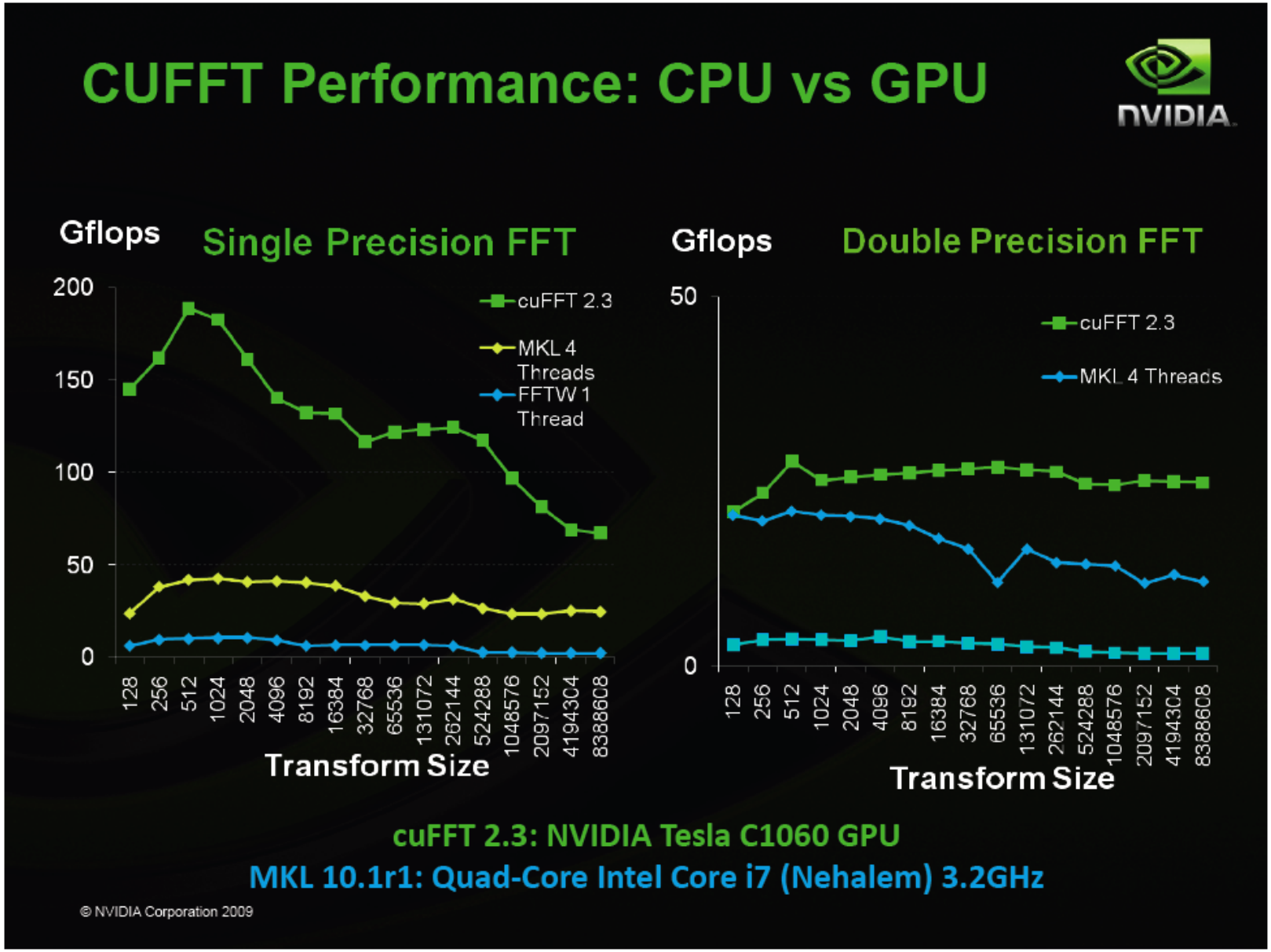
Parallel FFTs are well suited to calculations on GPUs (predictable data flow), consider **CUFFT = CUDA FFT** Library:

- Uses 'plans' like FFTW - good for tipping off compiler/run-time as to optimal configuration for FFT
- 1D, 2D, and 3D transforms
- Currently 1D length up to 8M (batched execution of multiple 1D FFTs)
- 2D and 3D lengths $\in [2, 16384]$
- Actively being improved; some interesting recent papers on auto-tuning:

  A. Nukuda and S. Matsuoka, "Auto-tuning 3D FFT Library for CUDA GPUs," Proc. of IEEE/ACM Supercomputing (Portland, OR, 2009).



2D complex transform, SC09 Tutorial (Portland, OR, 2009).



2D complex transform, SC09 Tutorial (Portland, OR, 2009).

Early Results for CUFFT on Fermi