

Algorithms of Scientific Computing

Fast Fourier Transform (FFT)

Michael Bader
Technical University of Munich

Summer 2018



TUM Uhrenturm

The Pair DFT/IDFT as Matrix-Vector Product

DFT and IDFT may be computed in the form

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \omega_N^{-nk} \quad f_n = \sum_{k=0}^{N-1} F_k \omega_N^{nk}$$

or as matrix-vector products

$$\mathbf{F} = \frac{1}{N} \mathbf{W}^H \mathbf{f}, \quad \mathbf{f} = \mathbf{W} \mathbf{F},$$

with a **computational complexity of $\mathcal{O}(N^2)$** .

Note that

$$\text{DFT}(f) = \frac{1}{N} \overline{\text{IDFT}(\bar{f})}.$$

A fast computation is possible via the **divide-and-conquer** approach.

Fast Fourier Transform for $N = 2^p$

Basic idea: sum up even and odd indices separately in IDFT

→ first for $n = 0, 1, \dots, \frac{N}{2} - 1$:

$$x_n = \sum_{k=0}^{N-1} X_k \omega_N^{nk} = \sum_{k=0}^{\frac{N}{2}-1} X_{2k} \omega_N^{2nk} + \sum_{k=0}^{\frac{N}{2}-1} X_{2k+1} \omega_N^{(2k+1)n}$$

We set $Y_k := X_{2k}$ and $Z_k := X_{2k+1}$, use $\omega_N^{2nk} = \omega_{N/2}^{nk}$, and get a sum of two IDFT on $\frac{N}{2}$ coefficients:

$$x_n = \sum_{k=0}^{N-1} X_k \omega_N^{nk} = \underbrace{\sum_{k=0}^{\frac{N}{2}-1} Y_k \omega_{N/2}^{nk}}_{:= y_n} + \omega_N^n \underbrace{\sum_{k=0}^{\frac{N}{2}-1} Z_k \omega_{N/2}^{nk}}_{:= z_n} .$$

Note: this formula is actually valid for all $n = 0, \dots, N - 1$; however, the IDFTs of size $\frac{N}{2}$ will only deliver the y_n and z_n for $n = 0, \dots, \frac{N}{2} - 1$ (but: y_n and z_n are periodic!)

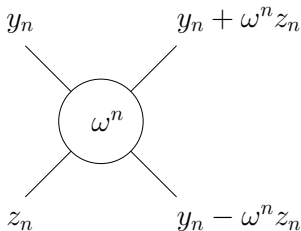
Fast Fourier Transform (FFT)

Consider the formula $x_n = y_n + \omega_N^n z_n$ for indices $\frac{N}{2}, \dots, N-1$:

$$x_{n+\frac{N}{2}} = y_{n+\frac{N}{2}} + \omega_N^{(n+\frac{N}{2})} z_{n+\frac{N}{2}} \quad \text{for } n = 0, \dots, \frac{N}{2} - 1$$

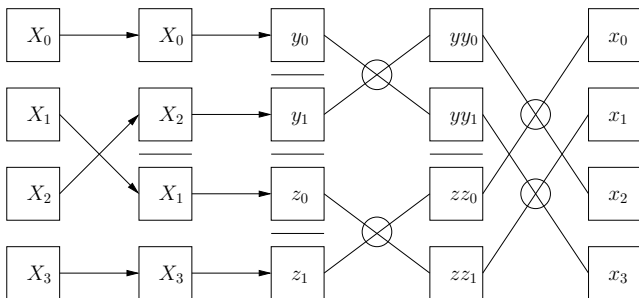
Since $\omega_N^{(n+\frac{N}{2})} = -\omega_N^n$ and y_n and z_n have a period of $\frac{N}{2}$, we obtain the so-called **butterfly scheme**:

$$\begin{aligned} x_n &= y_n + \omega_N^n z_n \\ x_{n+\frac{N}{2}} &= y_n - \omega_N^n z_n \end{aligned}$$

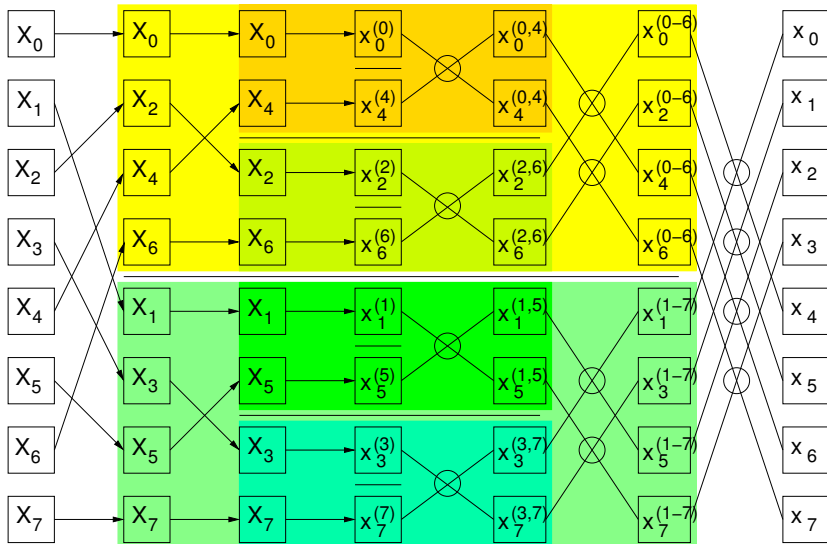


Fast Fourier Transform – Butterfly Scheme

$$\begin{aligned}
 (x_0, x_1, \dots, x_{N-1}) &= \text{IDFT}(X_0, X_1, \dots, X_{N-1}) \\
 &\Downarrow \\
 (y_0, y_1, \dots, y_{\frac{N}{2}-1}) &= \text{IDFT}(X_0, X_2, \dots, X_{N-2}) \\
 (z_0, z_1, \dots, z_{\frac{N}{2}-1}) &= \text{IDFT}(X_1, X_3, \dots, X_{N-1})
 \end{aligned}$$



Fast Fourier Transform – Butterfly Scheme (2)



Recursive Implementation of the FFT

$\text{rekFFT}(\mathbf{X}) \longrightarrow \mathbf{x}$

(1) Generate vectors \mathbf{Y} and \mathbf{Z} :

$$\text{for } n = 0, \dots, \frac{N}{2} - 1: \quad Y_n := X_{2n} \quad \text{und} \quad Z_n := X_{2n+1}$$

(2) compute 2 FFTs of half size:

$$\text{rekFFT}(\mathbf{Y}) \longrightarrow \mathbf{y} \quad \text{and} \quad \text{rekFFT}(\mathbf{Z}) \longrightarrow \mathbf{z}$$

(3) combine with “butterfly scheme”:

$$\text{for } k = 0, \dots, \frac{N}{2} - 1: \quad \begin{cases} X_k &= y_k + \omega_N^k z_k \\ X_{k+\frac{N}{2}} &= y_k - \omega_N^k z_k \end{cases}$$

Observations on the Recursive FFT

- Computational effort $C(N)$ ($N = 2^p$) given by recursion equation

$$C(N) = \begin{cases} \mathcal{O}(1) & \text{for } N = 1 \\ \mathcal{O}(N) + 2C\left(\frac{N}{2}\right) & \text{for } N > 1 \end{cases} \Rightarrow C(N) = \mathcal{O}(N \log N)$$

- Algorithm splits up in 2 phases:
 - resorting of input data
 - combination following the “butterfly scheme”
- ⇒ Anticipation of the resorting enables a simple, iterative algorithm without additional memory requirements.

Sorting Phase of the FFT – Bit Reversal

Observation:

- even indices are sorted into the upper half, odd indices into the lower half.
 - distinction even/odd based on least significant bit
 - distinction upper/lower based on most significant bit
- ⇒ An index in the sorted field has the **reversed** (i.e. mirrored) binary representation compared to the original index.

Sorting of a Vector ($N = 2^p$ Entries, Bit Reversal)

```

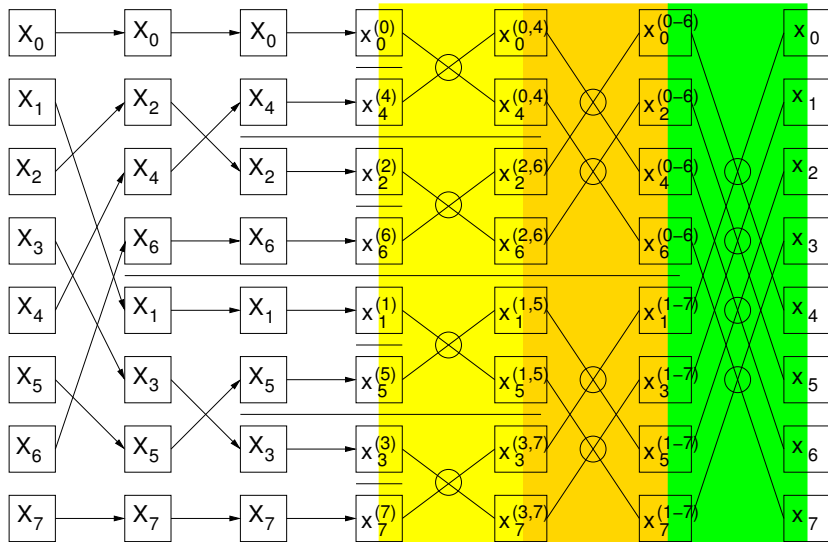
/** FFT sorting phase: reorder data in array X */
for(int n=0; n<N; n++) {
    // Compute p-bit bit reversal of n in j
    int j=0; int m=n;
    for(int i=0; i<p; i++) {
        j = 2*j + m%2; m = m/2;
    }
    // if j>n exchange X[j] and X[n]:
    if (j>n) { complex<double> h;
        h = X[j]; X[j] = X[n]; X[n] = h;
    }
}

```

Bit reversal needs $\mathcal{O}(p) = \mathcal{O}(\log N)$ operations

- ⇒ Sorting results also in a complexity of $\mathcal{O}(N \log N)$
- ⇒ Sorting may consume up to 10–30 % of the CPU time!

Iterative Implementation of the “Butterflies”



Iterative Implementation of the “Butterflies”

```

{Loop over the size of the IDFT}
for(int L=2; L<=N; L*=2)
    {Loop over the IDFT of one level}
    for(int k=0; k<N; k+=L)
        {perform all butterflies of one level}
        for(int j=0; j<L/2; j++) {
            {complex computation:}
             $z \leftarrow \omega_L^j * X[k+j+L/2]$ 
             $X[k+j+L/2] \leftarrow X[k+j] - z$ 
             $X[k+j] \leftarrow X[k+j] + z$ 
        }
    }

```

- k-loop und j-loop are “permutable”!
- How and when are the ω_L^j computed?

Iterative Implementation – Variant 1

```

/** FFT butterfly phase: variant 1 */
for(int L=2; L<=N; L*=2)
  for(int k=0; k<N; k+=L)
    for(int j=0; j<L/2; j++) {
      complex<double> z = omega(L,j) * X[k+j+L/2];
      X[k+j+L/2] = X[k+j] - z;
      X[k+j] = X[k+j] + z;
    }

```

Advantage: consecutive (“stride-1”) access to data in array X

⇒ suitable for vectorisation

⇒ good cache performance due to prefetching (stream access) and usage of cache lines

Disadvantage: multiple computations of ω_L^j

Iterative Implementation – Variant 1 Vectorised

```

/** FFT butterfly phase: variant 1 vectorised*/
for(int L=2; L<=N; L*=2)
  for(int k=0; k<N; k+=L)
    for(int j=0; j<L/2; j+= SIMD_LENGTH) {
      kjStart = k+j;
      kjEnd = k+j + (SIMD_LENGTH-1);
      complex<double> z[0: SIMD_LENGTH-1];
      z[0: SIMD_LENGTH-1] = omega(L,j) * X[kjStart+L/2 : kjEnd+L/2];
      X[kjStart+L/2 : kjEnd+L/2] = X[kjStart :kjEnd] - z;
      X[kjStart :kjEnd] = X[kjStart :kjEnd] + z;
    }

```

Comments on the code:

- we assume that our CPU offers SIMD instructions that can add/multiply short vectors of SIMD_LENGTH complex numbers
- with the notation $X[kjStart :kjEnd]$ we mean a subvector of SIMD_LENGTH consecutive complex numbers starting from element $X[kjStart]$
- Note: the term $\omega(L,j)$ has to be changed into such a short vector, as well

Iterative Implementation – Variant 2

```

/** FFT butterfly phase: variant 2 */
for(int L=2; L<=N; L*=2)
  for(int j=0; j<L/2; j++) {
    complex<double> w = omega(L,j);
    for(int k=0; k<N; k+=L) {
      complex<double> z = w * X[k+j+L/2];
      X[k+j+L/2] = X[k+j] - z;
      X[k+j] = X[k+j] + z;
    }
  }

```

Advantage: each ω_L^j only computed once

Disadvantage: “stride-L”-access to the array X

- ⇒ worse cache performance (inefficient use of cache lines)
- ⇒ not suitable for vectorisation

Separate Computation of ω_L^j

- necessary: $N - 1$ factors

$$\omega_2^0, \omega_4^0, \omega_4^1, \dots, \omega_L^0, \dots, \omega_L^{L/2-1}, \dots, \omega_N^0, \dots, \omega_N^{N/2-1}$$

- are computed in advance, and stored in an array w , e.g.:

```
for(int L=2; L<=N; L*=2)
    for(int j=0; j<L/2; j++)
        w[L/2+j] ←  $\omega_L^j$ ;
```

- Variant 2: access on w in sequential order
- Variant 1: access on w local (but repeated) and compatible with vectorisation
- Important: weight array $w[:]$ needs to stay in cache!
(as accesses to main memory can be slower than recomputation)

Cache Efficiency – Variant 1 Revisited

```

/** FFT butterfly phase: variant 1 */
for(int L=2; L<=N; L*=2)
  for(int k=0; k<N; k+=L)
    for(int j=0; j<L/2; j++) {
      complex<double> z = w[L/2+j] * X[k+j+L/2];
      X[k+j+L/2] = X[k+j] - z;
      X[k+j] = X[k+j] + z;
    }

```

Observation:

- each L-loop traverses entire array X
- in the ideal case $(N \log N)/B$ **cache line transfers** (N/B per L-loop, B the size of the cache line), unless all N elements fit into cache

Compare with recursive scheme:

- if $L < M_C$ (M_C the cache size), then the entire FFT fits into cache
- is it thus possible to require only $N \log N / (M_C B)$ cache line transfers?

Butterfly Phase with Loop Blocking

```

/** FFT butterfly phase: loop blocking for k */
for(int L=2; L<=N; L*=2)
  for(int kb=0; kb<N; kb+=M)
    for(int k=kb; k<kb+M; k+=L)
      for(int j=0; j<L/2; j++) {
        complex<double> z = w[L/2+j] * X[k+j+L/2];
        X[k+j+L/2] = X[k+j] - z;
        X[k+j] = X[k+j] + z;
      }

```

Question: can we make the L-loop an inner loop?

- kb-loop and L-loop may be swapped, if $M > L$
- however, we assumed $N > M_C$ (“data does not fit into cache”)
- we thus need to split the L-loop into a phase $L=2..M$ (in cache) and a phase $L=2*M..N$ (out of cache)

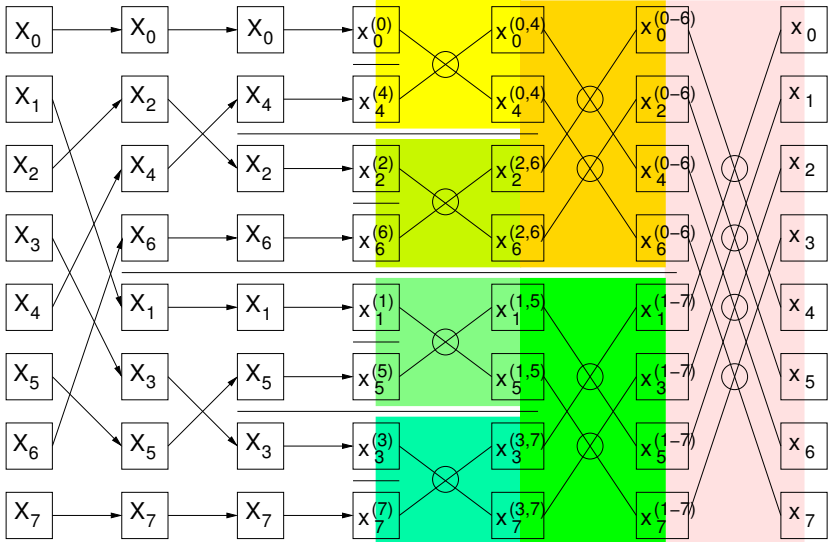
Butterfly Phase with Loop Blocking (2)

```

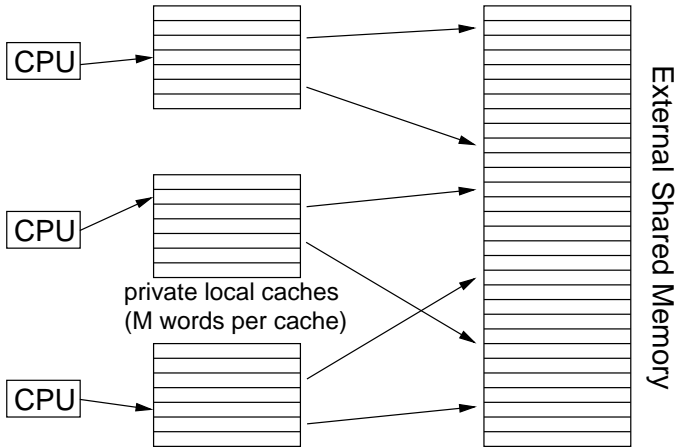
/** perform all butterfly phases of size M */
for(int kb=0; kb<N; kb+=M)
  for(int L=2; L<=M; L*=2)
    for(int k=kb; k<kb+M; k+=L)
      for(int j=0; j<L/2; j++) {
        complex<double> z = w[L/2+j] * X[k+j+L/2];
        X[k+j+L/2] = X[k+j] - z;
        X[k+j] = X[k+j] + z;
      }
/** perform remaining butterfly levels of size L>M */
for(int L=2*M; L<=N; L*=2)
  for(int k=0; k<N; k+=L)
    for(int j=0; j<L/2; j++) {
      complex<double> z = w[L/2+j] * X[k+j+L/2];
      X[k+j+L/2] = X[k+j] - z;
      X[k+j] = X[k+j] + z;
    }

```

Loop Blocking and Recursion – Illustration



Outlook: Parallel External Memory and I/O Model



[Arge, Goodrich, Nelson, Sitchinava, 2008]

Outlook: Parallel External Memory

Classical I/O model:

- large, global memory (main memory, hard disk, etc.)
- CPU can only access smaller working memory (cache, main memory, etc.) of M_C words each
- both organised as cache lines of size B words
- algorithmic complexity determined by memory transfers

Extended by Parallel External Memory Model:

- multiple CPUs access private caches
- caches fetch data from external memory
- exclusive/concurrent read/write classification (similar to PRAM model)

Outlook: FFT and Parallel External Memory

Consider Loop-Blocking Implementation:

```
/** perform all butterfly phases of size M */  
for(int kb=0; kb<N; kb+=M)  
  for(int L=2; L<=M; L*=2)  
    for(int k=kb; k<kb+M; k+=L)  
      for(int j=0; j<L/2; j++) {  
        /* ... */  
      }
```

- choose M such that one kb -Block (M elements) fit into cache
- then: L -loop and inner loops access only cached data
- number of cache line transfers therefore:
 $\approx M$ divided by words per cache line (ideal case)

Outlook: FFT and Parallel External Memory (2)

Consider Non-Blocking Implementation:

```
/** perform remaining butterfly levels of size  $L > M$  */  
for(int L=2*M; L<=N; L*=2)  
  for(int k=0; k<N; k+=L)  
    for(int j=0; j<L/2; j++) {  
      /* ... */
```

- assume: N too large to fit all elements into cache
- then: each L -loop will need to reload all elements X into cache
- number of cache line transfers therefore:
 $\approx M$ divided by words per cache line (ideal case) **per L -iteration**

Compute-Bound vs. Memory-Bound Performance

Consider a memory-bandwidth intensive algorithm:

- you can do a lot more flops than can be read from memory
- **computational intensity** of a code:
number of performed flops per accessed byte

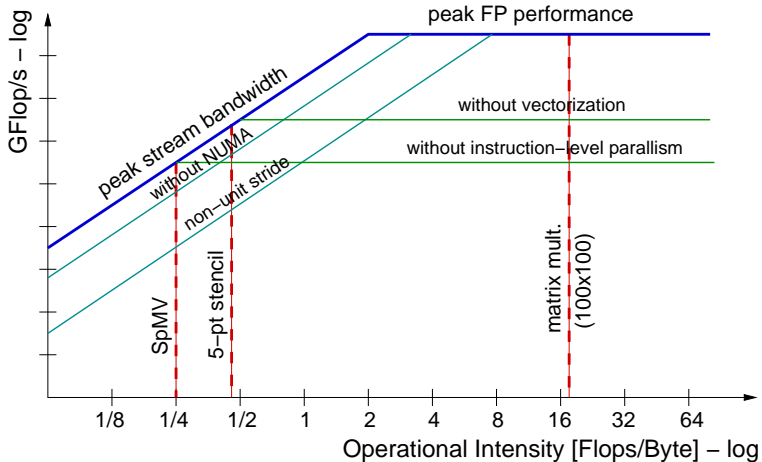
Memory-Bound Performance:

- computational intensity smaller than critical ratio
- you could execute additional flops “for free”
- speedup only possible by reducing memory accesses

Compute-Bound Performance:

- enough computational work to “hide” memory latency
- speedup only possible by reducing operations

Outlook: The Roofline Model



[Williams, Waterman, Patterson, 2008]

Outlook: The Roofline Model

Memory-Bound Performance:

- available bandwidth of a bytes per second
- computational intensity small: x flops per byte
- CPU thus executes x/a flops per second
- linear increase of the Flop/s with variable $x \rightsquigarrow$ linear part of “roofline”
- **“ceilings”**: memory bandwidth limited due to “bad” memory access (strided access, non-uniform memory access, etc.)

Compute-Bound Performance:

- computational intensity small: x flops per byte
- CPU executes highest possible Flop/s \rightsquigarrow flat/constant “rooftop”
- **“ceilings”**: fewer Flop/s due to “bad” instruction mix (no vectorization, bad branch prediction, no multi-add instructions, etc.)