Algorithms of Scientific Computing

Fast Fourier Transform (FFT)

Michael Bader
Technical University of Munich
Summer Term 2016
The Pair DFT/IDFT as Matrix-Vector Product

DFT and IDFT may be computed in the form

\[ F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \omega^{'-nk}_N \quad \quad f_n = \sum_{k=0}^{N-1} F_k \omega^{'nk}_N \]

or as matrix-vector products

\[ F = \frac{1}{N} W^H f , \quad \quad f = W F , \]

with a computational complexity of \( O(N^2) \).

Note that

\[ \text{DFT}(f) = \frac{1}{N} \text{IDFT}(\tilde{f}) . \]

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 \( n = 0, 1, \ldots, \frac{N}{2} - 1 \):

\[
x_n = \sum_{k=0}^{N-1} X_k \omega_N^{nk} = \sum_{k=0}^{\frac{N}{2}-1} \left( X_{2k} \omega_N^{2nk} + X_{2k+1} \omega_N^{(2k+1)n} \right).
\]

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

\[
x_n = \sum_{k=0}^{N-1} X_k \omega_N^{nk} = \sum_{k=0}^{\frac{N}{2}-1} Y_k \omega_N^{nk} + \omega_N^n \sum_{k=0}^{\frac{N}{2}-1} Z_k \omega_N^{nk}.
\]

\[
\begin{align*}
&:= y_n \\
&:= z_n
\end{align*}
\]

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

Do the same even vs. odd separation for indices $\frac{N}{2}, \ldots, N - 1$:

$$x_{n+\frac{N}{2}} = y_{n+\frac{N}{2}} + \omega_{N}^{(n+\frac{N}{2})}z_{n+\frac{N}{2}}$$

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:

$$x_{n} = y_{n} + \omega_{N}^{n}z_{n}$$

$$x_{n+\frac{N}{2}} = y_{n} - \omega_{N}^{n}z_{n}$$
Fast Fourier Transform – Butterfly Scheme

\[
(x_0, x_1, \ldots, x_{N-1}) = \text{IDFT}(X_0, X_1, \ldots, X_{N-1})
\]
\[
\downarrow
\]
\[
(y_0, y_1, \ldots, y_{N/2-1}) = \text{IDFT}(X_0, X_2, \ldots, X_{N-2})
\]
\[
(z_0, z_1, \ldots, z_{N/2-1}) = \text{IDFT}(X_1, X_3, \ldots, X_{N-1})
\]
Fast Fourier Transform – Butterfly Scheme (2)
Recursive Implementation of the FFT

\[ \text{rekFFT}(X) \rightarrow x \]

(1) Generate vectors \( Y \) and \( Z \):

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

(2) compute 2 FFTs of half size:

\[ \text{rekFFT}(Y) \rightarrow y \quad \text{and} \quad \text{rekFFT}(Z) \rightarrow z \]

(3) combine with “butterfly scheme”:

\[ \text{for } k = 0, \ldots, \frac{N}{2} - 1: \quad \begin{cases} 
  x_k &= y_k + \omega_N^k z_k \\
  x_{k + \frac{N}{2}} &= y_k - \omega_N^k z_k 
\end{cases} \]
Observations on the Recursive FFT

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

\[
C(N) = \begin{cases} 
O(1) & \text{for } N = 1 \\
O(N) + 2C\left(\frac{N}{2}\right) & \text{for } N > 1
\end{cases}
\]

$\Rightarrow$ $C(N) = O(N \log N)$

- Algorithm splits up in 2 phases:
  - resorting of input data
  - combination following the “butterfly scheme”

$\Rightarrow$ 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 */
\nfor (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”

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

- k-loop und j-loop are “permutable”!
- How and when are the $\omega_L^j$ computed?
Iterative Implementation – Variant 1

```c
/** 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 $\omega^j_L$
Iterative Implementation – Variant 2

```c
/* 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 $\omega_j^L$ only computed once

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

$\Rightarrow$ worse cache performance (inefficient use of cache lines)

$\Rightarrow$ not suitable for vectorisation
Separate Computation of $\omega^j_L$

- necessary: $N - 1$ factors
  $$\omega_2^0, \omega_4^0, \omega_4^1, \ldots, \omega_L^0, \ldots, \omega_{L/2-1}^L, \ldots, \omega_N^0, \ldots, \omega_{N/2-1}^N$$

- 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
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
  \((B\) the size of the cache line)

Compare with recursive scheme:

- if \(L < M_C\) \((M_C\) the cache size), entire FFT of size \(L\) could be computed in cache
- then maybe only \(N \log N/(M_C B)\) cache line transfers?
Butterfly Phase with Loop Blocking

```c
/** 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)

```c
/** 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)
Consider Loop-Blocking Implementation:

```c
/** 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:
  \[ \approx \frac{M}{\text{words per cache line}} \] (ideal case)
Consider Non-Blocking Implementation:

```c
/** 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:
  \[ \approx \frac{M}{\text{words per cache line}} \] (ideal case) per L-iteration
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
- "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
- "ceilings": fewer Flop/s due to "bad" instruction mix (no vectorization, bad branch prediction, no multi-add instructions, etc.)