New FFTW Developments

Matteo Frigo
FFTW

- C library for computing discrete Fourier transforms.
- Arbitrary size, multiple dimensions, real and complex.
- Widespread use. Ships with Matlab 6.
- Unusual implementation:
  - Adapts to hardware. Portability and high performance at the same time.
  - Computational kernels (95% of the code) generated automatically.
FFTW version 3

- Released in April 2003.
- Available at http://www.fftw.org/, GPL.
- Fourier, real Fourier, cosine, sine, and Hartley transforms.
- Support for interleaved and split complex arrays.
- SIMD (SSE, SSE2, 3DNow!, Altivec) support, even for sizes \( \neq 2^k \).
- Fused multiply-add support (PowerPC, IA-64, ...).
- Faster out-of-cache transforms.
- Improved accuracy.
Technical innovations in FFTW3

- New flexible infrastructure for automatic adaptation. Experimental FFTW system for convolutions.

- New SIMD complex-FFT algorithm.

- Automatic parallelizer extracts 2-way SIMD parallelism from sequential FFT programs. [Kral]

- New algorithm for real transforms of prime size, based upon [Rader 1968].

- New optimizer derives DCT/DST algorithms automatically from an FFT algorithm.

- Portable SIMD framework. Same C code works on SSE, SSE2, 3DNow!, Altivec. [Franchetti]
Outline

- Benchmarks.
- Structure of FFTW3.
- How FFTW3 uses SIMD instructions:
  - The complex-FFT SIMD algorithm.
  - The automatic vectorizer.
- Conclusions.
Benchmarks

Next four slides:

- FFTW3 vs. Intel MKL/IPPS.
- double precision, SSE2.
- 2.8 GHz Pentium IV, icc -O3, 256 KB L2.
- fftw-3.0.1, MKL 6.0, IPPS 3.0, icc 7.1.
- “mflops” := \( 5N \log N \) / time in \( \mu s \).

Many more benchmarks at

http://www.fftw.org/
FFT W3 vs. Intel, 1D, $2^k$

![Graph comparing FFTW3 and Intel's performance](image-url)
FFTW vs. Intel, 1D

speed (mflops)

fftw3 out-of-place
fftw3 in-place
intel-mkl-dfti out-of-place
intel-mkl-dfti in-place

9
12
15
18
24
36
80
108
210
504
1000
1960
4725
10368
27000
75600
165375

0
500
1000
1500
2000
2500
3000
3500
FFT W3 vs. Intel, 2D, $2^k$

Legend:
- ff tw3 out-of-place
- ff tw3 in-place
- intel-mkl-f
- intel-mkl-dfti in-place
- intel-mkl-dfti out-of-place

Graph showing speed (mflops) vs. problem size for FFTW3 and Intel MKL in-place and out-of-place FFT implementations.
FFT W3 vs. Intel, 2D

speed (mflops)

FFT W3 in-place
FFT W3 out-of-place
Intel-MKL-DFTI in-place
Intel-MKL-DFTI out-of-place

5x5 6x6 7x7 9x9 10x10 11x11 12x12 13x13 14x14 15x15 25x24 48x48 49x49 60x60 72x56 75x75 80x80 84x84 96x96 100x100
Structure of FFTW3

- A **problem** describes a computation to be performed.
  - E.g., 7 forward complex DFTs of size 13.

- A **plan** performs the computation described by a problem.
  - Most of the code for plans is generated automatically.

- A **solver** looks at the problem and it either fails or returns a plan.
  - E.g.: radix-$r$ Cooley-Tukey solver, for various $r$.
  - E.g.: multidimensional solver reduces $d$-dimensional problems to 1D problems.

- Given problem, the **planner** tries all applicable solvers and picks the fastest.
  - Various heuristics to speed up the search.
FFTW3 is flexible

- The planner knows nothing about FFTs.
  - Same planner works for FFT, DCT, convolution, etc.
- To try a new algorithm/trick, just add a solver.
  - Decimation in time or decimation in frequency?
  - Compute twiddle factors or load them from memory?
  - Copy the input into a buffer for better locality?
  - Best order in $d$-dimensional transforms?
Why a planner?

FFTW3 on 2.2 GHz Pentium IV vs. FFTW3 with plans from Pentium III, Athlon.
FFTW’s SIMD strategy

Two competing approaches:

1. Special SIMD FFT algorithm.
   - Semi-portable C code.
   - Works for 2-way and 4-way SIMD.
   - Complex transforms only.

2. Automatic vectorizer.
   - Generates machine-specific assembly.
   - Works for 2-way SIMD only.
   - Currently only supports 3DNow!.


**FFTW’s SIMD FFT algorithm**

**Cooley-Tukey:** \( r \) FFTs of size \( n/r \), plus \( n/r \) FFTs of size \( r \).

**Idea:** Compute the *complex* FFT of size \( r \) as a vector of two *real* FFTs of size \( r \):

\[
\text{FFT}(X) = \text{FFT}(\text{Re}(X)) + i \cdot \text{FFT}(\text{Im}(X)).
\]

Arithmetic complexity same as Cooley-Tukey.

Works for 2-way SIMD. For 4-way, do two steps in parallel.

*Other implementations compute multiple butterflies in parallel. FFTW exploits parallelism within a butterfly.*
Discussion of SIMD algorithm

Good:

- Works whenever Cooley-Tukey works, including odd sizes.
- 2-way SIMD: no need to worry about strides/alignment.
- Works for normal complex arrays—no weird data formats.
- Uses fewer registers than scalar code for same $r$.

Bad:

- Hard to implement, but we generate code automatically anyway.
- Extra shuffling at the end of each radix-$r$ step.
Automatic two-way vectorizer
Example: size-3 FFT

```plaintext
r0 = in[0];
r2 = in[2];
r4 = in[4];
r6 = r2 + r4;
r8 = r4 - r2;
r10 = .866 * r8;
r12 = .5 * r6;
r14 = r0 - r12;
r16 = r0 + r6;
r18 = r14 + r11;
r20 = r14 - r11;
out[0] = r16;
out[2] = r18;
out[4] = r20;
r1 = in[1];
r3 = in[3];
r5 = in[5];
r7 = r3 + r5;
r9 = r3 - r5;
r11 = .866 * r9;
r13 = .5 * r7;
r15 = r1 - r13;
r17 = r1 + r7;
r19 = r15 + r10;
r21 = r15 - r10;
out[1] = r17;
out[3] = r19;
out[5] = r21;
```
Two-way vectorized FFT

(r0, r1) = (in[0], in[1]);
(r2, r3) = (in[2], in[3]);
(r4, r5) = (in[4], in[5]);
(r6, r7) = (r2, r3) + (r4, r5);
(r8, r9) = ((r2, r3) - (r4, r5)) * (-1.0, 1.0);
(r10, r11) = (.866, .866) * (r8, r9);
(r12, r13) = (.5, .5) * (r6, r7);
(r14, r15) = (r0, r1) - (r12, r13);
(r16, r17) = (r0, r1) + (r6, r7);
(r10, r11) = (r11, r10);
(r18, r19) = (r14, r15) + (r10, r11);
(r20, r21) = (r14, r15) - (r10, r11);
(out[0], out[1]) = (r16, r17);
(out[2], out[3]) = (r18, r19);
(out[4], out[5]) = (r20, r21);
**Vectorization algorithm**

**Input:** FFT program, given as flow graph.

**Goal:** pair each node with some other node.

**How:** brute force.

- Maintain a set of paired nodes, initially empty.
- While unpaired nodes exist:
  - Nondeterministically choose two unpaired nodes.
  - Pair them according to “pairing rules”. If this step fails, backtrack.
- Perform local optimizations.
Example: pairing rule for loads

If \( \text{addr}1 = \text{addr}0 + 1 \), rewrite

\[
\begin{align*}
    v_0 &= \text{mem}[\text{addr}0]; \\
    v_1 &= \text{mem}[\text{addr}1];
\end{align*}
\]

into

\[
(v_0, v_1) = (\text{mem}[\text{addr}0], \text{mem}[\text{addr}0 + 1]);
\]

Pair \((v_0, v_1)\).

(SIMD load works only on consecutive addresses.)
When does the vectorizer work?

- Complex transforms of any size: OK.
  - Vectorizer usually (but not always) pairs real and imaginary part of a complex number (modulo conjugation and swap).

- Real transforms of even size: OK.
  - Not clear what the vectorizer does.

- Real transforms of odd size: FAIL.
  - Cannot pair an odd number of loads, stores, or flops.
Conclusion

FFTW3 is a flexible framework for implementing FFTs and related computations.

In progress:

- Implement SIMD real transforms.
- Convolutions.
- Large 1D transforms.