Large-scale FFTs and convolutions on Apple hardware

R. Crandall, J. Klivington, and D. Mitchell

Advanced Computation Group, Apple Inc.

September 10, 2008

Abstract: Impressive FFT performance for large signal lengths can be achieved via a matrix paradigm that exploits the modern concepts of cache, memory, and multicore/multithreading. Each of the large-scale FFT implementations we report herein is built hierarchically on very fast FFTs from the standard OS X Accelerate library. (The hierarchical ideas should apply equally well for low-level FFTs of, say, the OpenCL/GPU variety.) By building on such established, packaged, small-length FFTs, one can achieve on a single Apple machine—and even for signal lengths into the billions—sustained processing rates in the multi-gigaflop/s region.

1 Motivation

Large-signal FFTs and convolutions are looming more important than ever the scientific computation field. We denote by N a signal length—or for 2D images say, a pixel count—and realize that N can range from very small (say $N = 256$ for such as speech processing), to very large (say $N \sim 2^{30}$ for experimental mathematics and computational number theory). Let us establish motivation via some scientific/industrial examples of our present focus: Large-scale FFTs.

- Image processing of today can involve images in the region of 10^9 pixels, and so transform/convolution lengths make sense into the gigaelement (2^{30}-element) region or even beyond. The example work [8]—like many other recent works—addresses entire images and FFT, in this way signaling a new era of large-scale FFTs. In our own research group, we have had occasion to effect FFTs on near-gigabyte images, so that lengths $N \sim 2^{29}$ on a luminance or other channel is called for.
- Experimental mathematics; e.g. FFTs of length $N \sim 2^{27}$ have been used for extreme-precision calculation of π, as performed by D. Bailey over recent decades. Such explorations are not merely recreational; indeed, experimental mathematics has uncovered already an impressive galaxy of results that feed back into theory.
- The powerful NERSC supercomputers have run FFTs by researchers in the areas of nanoscience, climate, astrophysics, chemistry [10]. The following items are some additional disciplinary applications of large-scale FFTs.
- Interferometric synthesis imaging in radio astronomy.
- Geophysics and seismology: Exploration imaging, seismic imaging, and so on.
- FFT-based differential equation solvers for problems in computational physics (Poisson equation, Green's functions, etc.). Here N might range well beyond 2^{20}.
- Medical diagnostic imaging : Tomography.
- Cosmology : Particle-mesh codes for fast N-body simulations (this is related to solving Poisson equations on a grid). Indeed, it turns out that even the celebrated N-body problem in its full glory can be cast as a giant-FFT problem.
- Cryptography and computational number theory. In this field, FFTs of length 2^{24} have been used, e.g. in large-prime searches. There seems to be no fundamental limit to signal size N as the field expands and security levels do, too. (However, for exact integer arithmetic as a crypto prerequisite, precision must be enhanced beyond standard double float; yet such changes are entirely consistent with our present large-scale treatment.)
- Search for Extra-Terrestrial Intelligence (SETI), in which (rather small) FFTs are used on microwave-received data, but the lengths may one day expand in such searches.

- Computational genomics: FFTs for correlation-based gene alignment, DNA sequencing, etc. That is, FFTs can be used to recognize specific substrings on a very larger string. In this kind of application lengths again exceeding $N \sim 2^{20}$ can be expected.

There does exist our previous 3D FFT work on cluster solutions [3], but the present treatment entirely supersedes that older work in the sense that it is now possible to effect a gigaelement FFT on a *single* MacPro machine with sufficient RAM, and do this with sustained multigigaflop/s rates. Likewise, another previous paper [4] covers many of the ideas in the present treatment; however, with the latest Apple hardware we have been able to tune some of these "supercomputer FFT" ideas in special ways, depending on knowledge of the newer issues of cache, memory-control, and multicore.

The central idea herein is to build hierarchically on low-level FFTs of the established Accelerate or OpenCL/GPU variety. This hierarchical motif allows for cache advantages—e.g. minimal columnar hopping—and multicore advantages—e.g. the handling of parcels of the FFT on different respective processors.

It is important to note that we are *not* redesigning the already efficient Accelerate FFTs in OS X; rather, we are calling such established functions at low level, effectively "lengthening" Accelerate's span of efficiency.

We choose herein to report FFT performance in Cooley–Tukey gigaflop per second (CT-Gflop/s, or CTGs). This simplifies tabular reporting, and is based on the algorithmic result that the classical, 1D, complex-signal Cooley–Tukey FFT of signal length $N = 2^n$ requires (here and beyond we denote $\lg := \log_2$):

$$5N \lg N = 5 \cdot 2^n n$$

operations. While it is true that various other FFTs of higher radix, or split-radix, can lower this historical prefactor 5, still even the most modern refinements have prefactor close to 5.[1] This approach of convenience was taken for the performance reporting in [6] and in [13]. Important note: For real-signal FFTs of length N, the factor $5 \to 2.5$, since such am FFT can be effected via a length $N/2$ complex variant. The mnemonic for remembering this connection is simple: If a complex and a real signal of equal lengths N have the same gigaflop/s rating, the real-signal version is twice as fast in real time (see for example Figures 3, 4). In a word, we thus establish our Cooley–Tukey gigaflop/s rating as

$$\text{CTGflop/s} := \frac{C \cdot N \lg N}{T},$$

where T is real time for the FFT process and $C = 5, 2.5$ respectively as the signal is complex, real.

Some highlights of our measurements can be listed as follows:

[1]For example, split-radix methods yield essentially $4N \log_2 N$ while further variations have gotten as low as $(34/9)N \log_2 N$ [11] [12].

- We are able to effect a 1D, single-precision, real-signal FFT of length 2^{32} (see Figure 7). This shows that our implementation is truly in 64-bit-pointer territory (2^{32} as a 32-bit integer is 0). And even so, as Figure 7 reveals, we obtain a sustained 6 CTGflop/s at $N = 2^{32}$.

- Our results on Apple hardware compare quite favorably to FFTW, and in many cases to the published Intel Math Kernel results for FFT [9]. One important aspect of these comparisons is that we are now able, on a single Mac Pro machine, to leverage multicore/mulithread ideas. For example, we can now effect a 2D complex FFT at > 7 CTGflop/s sustained, using 8 threads for $N = 2^{24}$ (see Figure 6).

- Even with one thread, we can now outperform the standard FFTW at all lengths for our 1D FFTs. As Figures 3 and 4 show, we are now performing at 2-to-5 CTGs over all lengths up to $N = 2^{25}$ (and in fact beyond that length); moreover by choosing between Accelerate and our new rowFFT, one can always exceed FFTW speed at any length.

- Using the present techniques, FFT-based 2D convolution of a $w \times h$ signal and a $k \times k$ kernel has a "breakover" vs. literal (vImage) convolution at about

  $$\lg(wh) > \frac{k^2}{1000}.$$

  The data from Table 5 reveal this basic heuristic. Though this breakover is roughly estimated thus, the asymptotic character is correct, being as literal acyclic convolution is $O(k^2 wh)$ ops while the FFT approach is $O(wh \lg(wh))$ ops (see Figure 2). In any event, our rowFFT method yields CPU times (Table 5) that, for large images, are the fastest we have seen on any desktop. Indeed we have employed such convolutions in our own image-processing research.

2 Review of FFT types

We shall describe high-performance implementations for these FFT classes:

- **1D, complex FFT:**
  A signal $x := (u_n + iv_n : \ n \in [0, N-1])$ with each u_n, v_n real, has FFT and its inverse defined respectively as [2]

  $$X_k \ := \ \sum_{j=0}^{N-1} x_j e^{-2\pi ijk/N}, \qquad x_j \ := \ \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{+2\pi ijk/N}.$$

[2]Technically, the given exponential sum is a discrete Fourier transform (DFT), while fast Fourier transform (FFT) refers to an algorithm for effecting the DFT. Still, for convenience we embrace this glitch of nomenclature and simply refer to either the algorithm or the summation as "FFT."

In any subsequent pseudocode, we shall think of the FFT as an actual vector-valued function on vectors, i.e.:

$$X = FFT(x), \quad x = FFT^{-1}(X).$$

- **1D, real-signal FFT:**
  This is the previous FFT construct, except that the imaginary parts v_n all vanish, i.e. each input element is $x_n = u_n + 0i$. The real-signal input property translates into Hermitian symmetry in spectral space, i.e.

  $$X_k = X_{N-k}^*,$$

  so that one may report only $N/2+1$ output elements; namely

  $$X_0, X_{N/2}$$

  as a pair of real numbers, and the list of complex values

  $$X_1, \ldots, X_{N/2-1},$$

  from which all higher transform elements $X_k, k > N/2$ can be precisely inferred. The bookkeeping balances out exactly: This Hermitian representation involves $2+2(N/2-1) = N$ total floats, as did the original input data.

- **2D, complex FFT:**
  This assumes complex input data $x = (x_{j,m} : \ j \in [0, H-1], m \in [0, W-1])$, i.e. data in an $H \times W$ array, so that the FFT

  $$X_{k,n} \ := \ \sum_{j=0}^{H-1} \sum_{m=0}^{W-1} x_{j,m} e^{-2\pi i (jk/H + mn/W)}$$

  likewise resides in an $H \times W$ array.

- **2D, real-signal FFT:**
  Here each input datum x_{jm} is real, so that just as in the 1D real-signal case, the FFT is formally the same but there is Hermitian symmetry. This time the symmetry reads

  $$X_{k,n} = X_{H-k,W-n}^*,$$

  which is a symmetry about the array midpoint. The output format we chose is rather complicated, but after all is said and done, the real-signal case involves half the data both on input and output for a given set of W, H parameters. In a later section we discuss how to employ 2D, real-signal FFTs for 2D real convolution and correlation.

- **3D FFT and beyond:**
  3D and beyond are not tabulated in the present paper, although the same principles of matrix factorization, cache, and multicore all apply. (Even the ideas of column-ferrying to avoid literal transpose do apply in the 3D mode, where along with rows

and columns one also has "pencils.") Note the previous papers [3] [13] that explain 3D notions, although it is important to observe that the present paper involves not a machine cluster—which was necessary some years ago for sufficient memory and so on—but implementation of all the ideas on a *single*, perhaps multicore machine.[3]

3 Algorithm overview

Previous, pioneering-papers [7] [2] [15] describe a technique suited for the "supercomputers" of the era 1970-2000, say. The basic idea is to split the FFT work between processors, by arranging the data in a matrix form so that each CPU can effect its own assigned strip of the matrix. Serendipitously enough, this matrix-factorization technique works quite well on modern machinery.

We remind ourselves that the core idea is to employ an efficient Accelerate or OpenCL/GPU-based FFT at low-level, and use this traditional "supercomputer factorization" to exploit the newer memory-cache and multicore advantages.

Recall that for a 2D FFT, where an $H \times W$ matrix is the natural format so the element count is $N := WH$, the procedure is to use the CPUs to FFT all the rows (H such FFTs), then transpose, then again use the CPUs on all rows (now W such FFTs). (If one has a fast way to FFT *columns* then of course the transpose is unnecessary.)

For a 1D FFT of length N, the procedure is the same as for 2D, except that one may choose any matrix layout having $N := WH$, and one must effect a "twist" on each element *between* the row/column mass operations. This twist factor renders all the algebra correct because of the following identity chain (see [5]), where we denote

$$g := e^{2\pi i/N}, \quad g_H := e^{2\pi i/H}, \quad g_W := e^{2\pi i/W},$$

with $N := WH$, that is, the 1D data x is assumed cast into a $H \times W$ matrix.[4] The 1D FFT we call X is then developed according to

$$\begin{aligned} X &= FFT(x) = \left(\sum_{j=0}^{N-1} x_j g^{-jk} \right)_{k=0}^{N-1} \\ &= \left(\sum_{j=0}^{W-1} \sum_{m=0}^{H-1} x_{j+mW} g^{-(j+mW)(k+lH)} \right)_{k+lH=0}^{N-1} \end{aligned}$$

[3]However, if one needs to go into the multi-gigaelement region, say beyond lengths 2^{32}, the machine-distribution ideas of those past papers will again come into play, and would need to be combined with modern ideas herein.

[4]When we say $H \times W$ matrix we mean H rows and W columns, even though one usually says in image processing work that an image is $W \times H$, the image and matrix nevertheless identical.

$$= \left(\sum_{j=0}^{W-1} \left(\sum_{m=0}^{H-1} x_{j+mW} g_H^{-mk} \right) g^{-jk} g_W^{-jl} \right)_{k+lH=0}^{N-1} .$$

But this final expression can be seen as a 2D FFT with an interposed twist factor g^{-jk}. An algorithm that exploits these algebraic machinations is called historically the "four-step" FFT [2][15][7].

Algorithm 1 [1D FFT, factored]: Let x be a signal of length $N = WH$. For algorithmic efficiency we consider said input signal x to be a matrix arranged in "columnwise order"; i.e., the m-th row of the matrix contains $(x_m, x_{m+H}, x_{m+2H}, \ldots, x_{m+(W-1)H})$; thus the input matrix is W wide by H tall. Then, conveniently, each FFT operation of the overall algorithm occurs on some row of some matrix (the k-th row vector of a matrix V will be denoted generally by $V^{(k)}$). The final FFT resides likewise in columnwise order. We denote the relevant N-th root as usual by $g = e^{2\pi i/N}$.

1. Perform H in-place, length-W FFTs, each performed on a row of the column-order input matrix, that is:
   for$(0 \le m < H)$ $T^{(m)} = FFT\left((x_{m+jH})\,|_{j=0}^{W-1} \right)$;
2. Transpose the matrix, that is for $j \in [0, W-1],\ k \in [0, H-1]$
   $U_k^{(j)} := T_j^{(k)}$;
3. Twist the matrix, by multiplying each element $U_k^{(j)}$ by g^{-jk}. Note, importantly, that the previous (transpose) step can be done in-place if desired, or interchanged with this twist step—there are many options.
4. Perform W in-place, length-H FFTs, each performed on a row of the new T, that is:
   for$(0 \le j < W)$ $(X_{jH+kW})|_{k=0}^{H-1} = FFT\left(U^{(j)} \right)$;
   // X is now the FFT in columnwise order, W tall, H wide (see Figure 1).

So, Algorithm 1 expects columnwise order, say an array that runs linearly in memory as $(x_0, x_H, x_{2H}, \ldots)$ and pictured like so:

$$\begin{pmatrix} x_0 & x_H & \cdots & x_{(W-1)H} \\ x_1 & x_{H+1} & \cdots & \cdot \\ \cdots & \cdots & \cdots & \cdot \\ x_{H-1} & \cdots & \cdots & x_{N-1} \end{pmatrix} .$$

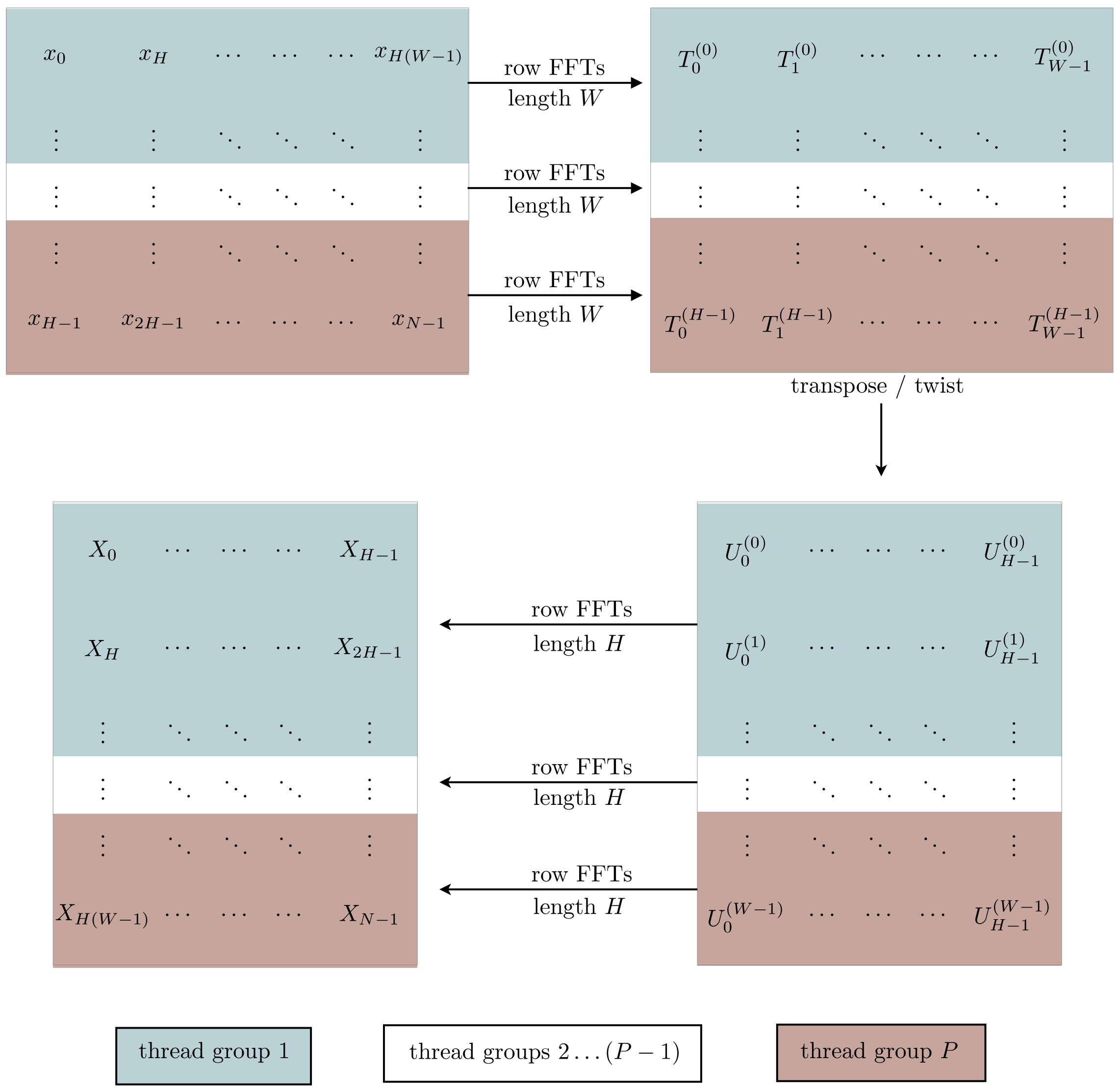
The output of Algorithm 1 is the same columnwise pattern for the transform elements X_j; that is, the sequence $X_0, X_1, X_2, \ldots$ runs down the first column, and so on.

The essential performance boost realized via Algorithm 1 is due to the fact of *always* performing row-FFTs. (A 2D FFT benefits from the same algorithmic presentation; one simply avoids the complex twist in step (2).) The fact of continual row-FFTs has twofold importance: First, cache is exploited because of FFT locality, and second, separate processors (threads) can act on strips of rows independently, with synchronization needed only for the transpose.

If one desires the more natural row-order data format, one may transpose at the beginning and at the end of Algorithm 1, to render what has been called the "six-step" FFT. Incidentally, all of our performance tabulation herein uses the columnwise format when that results in the best performance, on the idea that batch data can have been prestored in said format; plus, when the FFT output is in columnwise order, any spectral checks or accessing can occur with obvious indexing.

A key algorithm step—matrix transposition—is especially easy for square matrices, as one may simply swap against the diagonal. When a 1D complex FFT length is 2^d for odd d, one has a $2^{(d+1)/2} \times 2^{(d-1)/2}$ matrix, which has aspect ratio 2, so that transposition is nontrivial. One way out of this is to reduce the overall FFT by a factor of two, using a one-step butterfly, as in Algorithm 8 of [4].

A pictorial of the ideas behind Algorithm 1 is displayed herein as Figure 1.



x_0 x_H $\cdots$ $x_{H(W-1)}$
x_{H-1} x_{2H-1} $\cdots$ x_{N-1}
row FFTs length W
$T_0^{(0)}$ $T_1^{(0)}$ $\cdots$ $T_{W-1}^{(0)}$
$T_0^{(H-1)}$ $T_1^{(H-1)}$ $\cdots$ $T_{W-1}^{(H-1)}$
transpose / twist
$U_0^{(0)}$ $\cdots$ $U_{H-1}^{(0)}$
$U_0^{(1)}$ $\cdots$ $U_{H-1}^{(1)}$
$U_0^{(W-1)}$ $\cdots$ $U_{H-1}^{(W-1)}$
row FFTs length H
X_0 $\cdots$ X_{H-1}
X_H $\cdots$ X_{2H-1}
$X_{H(W-1)}$ $\cdots$ X_{N-1}
thread group 1
thread groups $2\ldots(P-1)$
thread group P

Figure 1: Pictorial for Algorithm 1. The matrix paradigm requires only row-FFTs throughout the procedure. For signal length $N = WH$, and input data in columnwise order, the length-W row-FFTs can be performed via simultaneous threads. The T-matrix at upper right is then transposed and twisted, to form the U-matrix at lower right. Then again, row-FFTs this time of length H are performed, optionally in threaded fashion.

4 Real-signal enhancements

One approach to 1D, real-signal FFT is to exploit the reality of the input data, say data

$$p = (p_0, p_1, \ldots, p_{N-1}) \quad , \quad p_j \text{ all real,}$$

and use the FFT (N is assumed even):

$$Q_k := \sum_{j=0}^{N/2-1} (p_{2j} + ip_{2j+1})e^{-2\pi ijk/(N/2)}.$$

In the following algorithm we perform a factored FFT of length $N/2$, in which algorithm we employ a complex-column matrix ordering, like so:

$$\begin{pmatrix} p_0 + ip_1 & p_{2h} + ip_{2h+1} & \cdots & p_{2(w-1)h} + ip_{2(w-1)h+1} \\ p_2 + ip_3 & \cdots & \cdots & . \\ \cdots & \cdots & \cdots & . \\ p_{2h-2} + ip_{2h-1} & \cdots & \cdots & p_{2wh-2} + ip_{2wh-1} \end{pmatrix},$$

where $wh = N/2$ is a chosen factorization of the half-length. Clearly, this matrix can be interpreted in engineering terms as a matrix of N floats. (height h and width $2w$).

Algorithm 2 [1D, real-signal FFT]: Given a real signal p of length N, output the 1D FFT of p as N real data in Hermitian order.

1. Choose matrix factorization $wh = N/2$ and ensure the data p_j are arranged in complex column order as above.
2. Perform a 1D, complex, factored FFT (Algorithm 1) to obtain a matrix

$$\begin{pmatrix} Q_0 & Q_w & \cdots & Q_{w(h-1)} \\ Q_1 & \cdots & \cdots & . \\ \cdots & \cdots & \cdots & . \\ Q_{w-1} & \cdots & \cdots & Q_{wh-1} \end{pmatrix}.$$

3. Return the desired FFT as Hermitian array

$$\left(P_0, P_{N/2}, P_1, \ldots, P_{N/2-1}\right),$$

where $P_0, P_{N/2}$ are both real, with all other P_k complex, according to relations

$$P_0 := \Re(Q_0) + \Im(Q_0),$$

$$P_{N/2} := \Re(Q_0) - \Im(Q_0),$$

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

$$2P_k := Q_k + Q^*_{N/2-k} - ie^{-2\pi ik/N}\left(Q_k - Q^*_{N/2-k}\right).$$

Note that the recovery of the P_k involves a twist term $e^{-2\pi ik/N}$ which, as is usual, can be "propagated"—see next section—so as to avoid painful storage of extensive trig tables.

Algorithm 3 [Inverse FFT for Algorithm 2)]: Given the result of Algorithm ?, i.e. data (P_k) in Hermitian order, we recover the original real signal of N elements.

1. Create a complex array Q of length $N/2$, by
$$2Q_k = P_k + P^*_{N/2-k} + ie^{2\pi ik/N}(P_k - P^*_{N/2-k}),$$
for $k \in [0, N/2-1]$.
2. Perform a length-$N/2$ complex inverse FFT on Q to obtain the original data in column-complex order (the same ordering that starts Algorithm 2).

For 2D real-signal FFTs, say with input of $N := H \times W$ real elements, one exploits the Hermitian symmetry to end up with exactly N spectral float values. This can be done as in the treatment [4], with the final 2D FFT occupying a N-element rectangular array according to the scheme:

$$\begin{bmatrix} X_{0,0} & X_{0,W/2} & \Re X_{0,1} & \Im X_{0,1} & \cdots & \Im X_{0,W/2-1} \\ X_{H/2,0} & X_{H/2,W/2} & \Re X_{1,1} & \Im X_{1,1} & \cdots & \Im X_{1,W/2-1} \\ \Re X_{1,0} & \Re X_{1,W/2} & \Re X_{2,1} & \Im X_{2,1} & \cdots & \Im X_{2,W/2-1} \\ \Im X_{1,0} & \Im X_{1,W/2} & \Re X_{3,1} & \Im X_{3,1} & \cdots & \Im X_{3,W/2-1} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \Re X_{H/2-1,0} & \Re X_{H/2-1,W/2} & \Re X_{H-2,1} & \Im X_{H-2,1} & \cdots & \Im X_{H-2,W/2-1} \\ \Im X_{H/2-1,0} & \Im X_{H/2-1,W/2} & \Re X_{H-1,1} & \Im X_{H-1,1} & \cdots & \Im X_{H-1,W/2-1} \end{bmatrix}.$$

The beauty of it is, everything is done in place, and either real- or complex-signal FFTs are performed along the way, yet every single FFT has either W or H float components, never more than that. We have been able to employ this efficient real-signal 2D FFT to advantage, on say image deconvolution (Wiener filtering), being as pixel values are deemed real.

5 Further enhancements suitable for Apple hardware

We hereby list algorithmic enhancements that are key in boosting performance beyond the basic gain of Algorithms 1,2,3.

- **Employ Accelerate or OpenCL/GPU at low hierarchical level:**
The idea is to use the hierarchical matrix-factorization and employ in steps 1 and 4 of Algorithm 1 a proven Accelerate or OpenCL/GPU FFT. This means we are not, in the present work, having to analyze microscopic features such a butterfly diagrams, radix choices, or any of the detailed low-lying FFT structure. We simply trust all that, and work at the top level with an eye to cache, overall memory, and multicore concepts. We expect Accelerate and OpenGL/GPU FFT implementations to be very fast on relatively small memory regions; lo and behold, each of the steps 1 and 4 of Algorithm 1 involves only a *row* FFT, and usually one of small length (a length roughly $\sqrt{N}$ where N is the overall signal length).

- **Invoke "trig propagation:"**
For matrix-twist operations such as step 3 of Algorithm 1, one does not want to ruin the memory footprint by having to store all complex values $e^{-2\pi ih/N}$. One may avoid such costly memory storage (not to mention painful access) by calculating paired trigonometric entities
$$(c, s) := (\cos(2\pi(y + xh)/N), \sin(2\pi(y + xh)/N)$$
suitable for twist operations within matrix algorithms. To calculate (c, s) pairs sequentially across the y-th row, one may initialize variables $(c, s) := (\cos(2\pi y/N), \sin(2\pi y/N))$ and constants $(d, t) := (\cos(2\pi h/N), \sin(2\pi h/N))$, then loop on the vector-update $f(c, s) := (cd - st, sd + ct)$, whence the value of (c, s) within the loop is the desired pair for $x > 1$. The advantage here is manifold: Such trig propagation works over a given row, requiring minimal trig storage merely to ignite a row, plus CPU-expensive table lookups are replaced with simple on-the-fly arithmetic. In our actual implementations we took care to re-ignite the propagation after a certain number of iterates, so as to preserve overall accuracy of the FFT.

- **Invoke real-signal enhancements:**
As detailed in the next section, there are performance (and memory) gains to be realized when the input data is pure real; we call the FFTs then "real-signal" FFTs. Asymptotically there is an expected speed gain of 2x due to such signal reality (and about 1/2 the memory for an in-place rendition).

- **Special schemes for matrix transpose:** Within Algorithm 1 is a matrix transpose, and there is also a transpose for 2D FFTs. For square-matrix scenarios, we implemented a custom, block-oriented transpose to maximize overall speed on Apple hardware. As for the tougher, nonsquare mode, one trivial approach is to waste memory by doing an out-of-place reverse-lexicographic loop. But the in-place nonsquare tranpose is problematic. There is a substantial literature on transposing a nonsquare matrix, yet we found an efficient scheme for doing the *equivalent* of a transpose—equivalent at least in the FFT world—as follows. Observe that a possible algorithm variant runs:

  Algorithm 1b [Column-ferrying]: Assume input matrix in lexicographic (normal) order, then:

1) Ferry 8 or 16 columns at a time into a (relatively) small row buffer, FFT such rows, then ferry the columns back; proceed in this way until all columns have been FFT'd;
2) Apply twist factor (if in 1D mode) else no-op;
3) FFT all the rows in-place;

and now the output is in columnwise order, yet no explicit transpose has been performed. Whether 8 or 16 (or some other number) of columns is to be handled at once depends on memory and CPU properties of the system. Note the beauty of this approach: It does not matter what is the matrix aspect ratio—one can always pick up columns of any height and put them back in methodical fashion.

We have observed the optimal ferrying width to typically be a single cache line (i.e. 64 bytes) which equates to 8 double-precision or 16 single-precision floating-point columns.

6 Convolution and correlation

An ubiquitous application of FFT is convolution, which application enters virtually every computational field. Thus we deem it appropriate to give special attention to convolution and performance of same.

Given two input signals x, y each of length N, various convolutions are signals z having, respectively, the elements

- **Cyclic convolution:**
$$z_n := \sum_{j+k \equiv n \bmod N} x_j y_k,$$
- **Acyclic convolution:**
$$z_n := \sum_{j+k=n} x_j y_k,$$
- **Cyclic correlation:**
$$z_n := \sum_{j-k \equiv n \bmod N} x_j y_k.$$

Each of these entities can be calculated either literally—meaning carry out the indicated summation for each n, an often highly painful process—or via FFT. In a word, the cyclic convolution can be obtained with the 3-FFT process

$$z = FFT^{-1}(FFT(x) * FFT(y)),$$

with acyclic convolution and cyclic correlation working similarly (see [5] for relevant convolution pseudocode). Here, the $*$ symbol means dyadic (i.e. elementwise) multiplication.

A typical image-processing scenario is shown in Figure 2; indeed we have timed this very scenario (see Table 5) to indicate for what parameters a literal (say vImage library convolution) will "break over" — i.e. will be dominated performancewise by—the FFT approach.

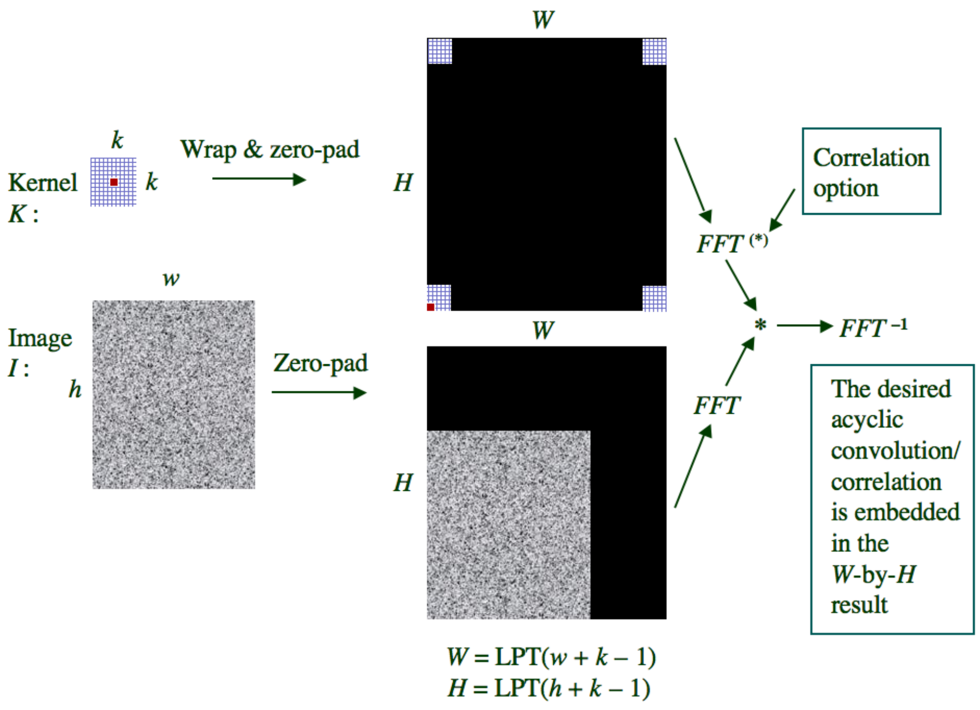


Figure 2: Proper scaling and zero-padding for FFT-based 2D convolution. For the convolution of an image I by a kernel K (the latter being $k \times k$ pixels) we use the least-power-of-2 (LPT) function to zero-pad both image and kernel, expecting to use three FFT calls. For small k, literal acyclic convolution—a very common graphics operation—will beat the FFT approach, but the latter will radically dominate in performance for large enough image-kernel combinations, as Table 5 shows.

The way we intend to effect a convolution in the scheme of Figure 2 is as follows:

Algorithm 4 [2D real-signal, acyclic convolution]: We assume an w (columns) $\times h$ (rows) image and a kernel K of (odd) dimensions $k \times k$, with which we wish to acyclically convolve. We shall exploit the hypothesis that each of the wh pixel values in the image, and k^2 kernel values, are all real.

1. Zero-pad both the image and the kernel as in Figure 2, using the least-power-of-two function LPT(), to end up with a padded $W \times H$ rectangle for both the padded image and padded kernel. (Technically, $\mathrm{LPT}(n) := 2^{\lceil \log_2 n \rceil}$.)

2. Perform a real-signal, 2D, $W \times H$ FFT on the padded image. (There are ways to do this in-place, using previous algorithms for row-FFTs, so that the real character of the image is observed without memory waste, and such that the final, Hermitian tableau at the end of our Section 4 is obtained.)

3. Perform step (2) for the padded kernel (that also has the dimensions W, H as in Figure 2).

4. We now have the Hermitian, 2D FFT for both padded image and padded kernel. Multiply dyadically these two representations. (To effect a proper dyadic multiply, one must follow the Hermitian ordering shown at the end of our Section 4; also found in [4], p. 16.)

5. Take a 2D, inverse FFT starting from the Hermitian representation of the dyadic product from the previous step. The result is the acyclic convolution of image $\times$ kernel, embedded in a padded $W \times H$ real array.

We have used Algorithm 4 on image-processing tasks as part of our internal research; indeed, we discovered some of the Apple-hardware-relevant performance enhancements because of the need for high speed on the image problems which, after all, did require real-signal, 2D FFTs carried out often.

7 Error analysis

Of course, a large-signal, hierarchical FFT no matter how elegant should nevertheless have good error behavior; certainly as good as the error scenario *without* the hierarchical expedients such as matrix-factorization and percolation methods. There is the additional problem that for very long signals, traditional test inputs may not behave well.

We have invoked some special tests begged by the very largeness of our longest FFTs—indeed, it is nontrivial to test just the accuracy of a very large FFT. For one thing, what should be the result of the FFT?

- **End-to-end test:**
  In this test one compares the original signal x to the round-trip signal $x' := FFT^{-1}(FFT(x))$, expecting $x' \approx x$ as measured by errors

$$RMSE := \sqrt{\frac{1}{N}\sum_j |x_j - x'_j|^2},$$

and

$$MXE := \max_j |x_j - x'_j|.$$

- **Chirp test:**
It is known that a complex "chirp" signal, defined for 1D signals as

$$x_j \ := \ e^{i\pi j^2/N}$$

and, for 2D signals as

$$x_{k,j} \ := \ e^{i\pi(j^2/W + k^2/H)}$$

have an FFT of *constant amplitude*; in fact

$$|X_k| = \sqrt{N}$$

for all $k \in [0, N-1]$. The constancy is easily evaluated via an error measure

$$CHE := \max_k \left| \frac{|X_k|}{\sqrt{N}} - 1 \right|.$$

- **Real-signal test:**
Here we present a certain test signal for very large, 1D, real-signal FFTs. This signal is carefully chosen so that either very short, or enormous lengths will be testable. Consider the signal $x := (x_j)$ where

$$x_j := e^{-Mj} - 2e^{-2Mj}, \qquad j \in [0, N-1],$$

where we observe this is a patently real signal when M is itself real. A practical value for M in single-precision work would be $M = 0.05$. This signal is "almost bipolar" in that the DC (0th) component of the FFT is small, and the signal is also temporally limited so that the spectrum is widespread.

The exact FFT of length N is given by

$$X_k = F(M, k) - 2F(2M, k), \qquad k \in [0, N-1],$$

where the function F is defined

$$F(L, k) := \frac{\left(1 - e^{-LN}\right)\left(1 - e^{-L} e^{\frac{2\pi i k}{N}}\right)}{1 - 2e^{-L} \cos\left(\frac{2k\pi}{N}\right) + e^{-2L}}.$$

The idea here is that $x := (x_j : j \in [0, N-1])$ is a signal whose computed FFT can be compared to the above form for X_k, term by term, without having to store neither the (supposedly vast) input data nor the (also vast) output data in memory at the same time when the FFT proper is being calculated. So the whole idea with this real-signal test is to calculate both RMSE and MXE as defined above, storing the results in files; then the FFT itself can be performed, and the results compared to the signals previously stored in files without reading the entire file into memory.

Of course, if one is content with checking just the "round-trip" errors, i.e. errors between the original x and the supposed identity result $FFT^{-1}(FFT(x))$, one can simply use initial/final file-writes and separately compare, and do this moreover on random data that doesn't need a closed form like the third test. Though the error scenario for FFTs in the gigaelement region is largely uncharted, there is some heuristic discussion in [3] on how errors should grow with N.

8 Performance graphs and tables

At the end of the present document appear graphs and tables. The graphs focus upon speed as measured in Cooley–Tukey gigaflops/sec (CTGflop/s, or CTGs), which gigaflop measurement is defined in our opening section. The tables cover mainly in-place FFTs; the existence of out-of-place FFTs, which are generally less efficient for multiple reasons, is caused by the difficulty of nonsquare transpose. That is to say, it is a quicker task to implement out-of-place when the matrix-hierarchical schema are in force. (Our research-code suite still has some out-of-place variants for this reason.) Still, it is alway possible to move to in-place implementations by adroit application of selected transpose and ferrying ideas.

- **Test machinery:**
  Mac Pro 3.2 GHz, two quad-core Xeon CPUs, with 32 GB of RAM, running Mac OS X 10.5.3.
- **Memory requirements:**
  Of course, having this much memory (32 GB) on our test machine is only important for pushing the signal length N as high as possible, as we did for Figure 7. It is easy to work out how much RAM is required for a given FFT length to stay in memory—one simply needs to know whether the FFT is in-place or out-of-place, and know that our ancillary trig tables or auxiliary memory buffers are of negligible size. For example, all of our implemented FFTs of the 1D variety are in-place, and so involve only a tad more than $N \cdot$ sizeof(signal element) bytes of memory. (One must of course allow for the OS X system footprint to get a final RAM estimate.) In this regard see Table 7.
- **Measurement conditions:**
  FFTW tests were run with the FFTW_MEASURE plan flag. Version 3.1.2 of FFTW was used.
- **Timing measurement:**
  Normally, timing information was measured via OS-measured "user time," which is the time actually spent executing user-level code. This excludes such kernel-level tasks as paging. For multicore tests, this is not appropriate; true "wall time"—the actual elapsed time—is used instead. Convolution performance was also measured with wall

time; the vImage convolution code always runs with as many threads as there are processor cores, therefore the FFT convolution tests also run with 8 threads.

9 Acknowledgements

The authors thank R. Altherr, D. Bailey, J. Buhler, S. Noble, I. Ollman, J. Papadopoulos, E. Postpischil, A. Sazegari, and R. van der Merwe for expert input during this research.

References

[1] Accelerate.framework,
`http://developer.apple.com/performance/accelerateframework.html`

[2] D. Bailey,“FFTs in External or Hierarchical Memory,” Journal of Supercomputing, vol. 4, no. 1 (Mar 1990), pg. 23-35.

[3] R. Crandall, E. Jones, J. Klivington, D. Kramer, “Gigaelement FFTs on Apple G5 Clusters,” 2004. `http://www.apple.com/acg/`

[4] R. Crandall and J. Klivington, “Supercomputer-style FFT library for Apple G4,” 6 Jan. 2000, `http://images.apple.com/acg/pdf/g4fft.pdf`

[5] R. Crandall and C. Pomerance, *Prime numbers: A computational perspective*, 2nd ed., Springer, 2002.

[6] M. Frigo and S. Johnson, “The Design and Implementation of FFTW3,” Proceedings of the IEEE 93: 216231, 2005.

[7] W. Gentleman and G. Sande, “Fast Fourier transforms—For Fun and Profit,” AFIPS Proc., vol. 29 (1966), pp. 563–578.

[8] H. Hartenstein and D. Saupe,“Lossless acceleraton of fractal image coding via the Fast Fourier Transform,” preprint, 2000.

[9] Intel Math Kernel Library 10.0 - FFT
`http://www.intel.com/cd/software/products/asmo-na/eng/266852.htm`

[10] `http://www.nersc.gov/∼kramer/LBNL/LBNL-57582.pdf`

[11] S. Johnson and M. Frigo, “A modified split-radix FFT with fewer arithmetic operations,” IEEE Trans. Signal Processing 55 (1): 111119, 2007.

[12] T. Lundy and J. Van Buskirk, “A new matrix approach to real FFTs and convolutions of length 2k,” Computing 80 (1): 23-45, 2007.

[13] S. Noble, “Gigaelement FFTs on x86-based Apple clusters,” `www.apple.com/acg`, 2007.

[14] D. Ohara,“CUDA: A Quick Look and Comparison of FFT Performance,” (2008) `http://www.macresearch.org/cuda-quick-look-and-comparison-fft-performance`

[15] P. Swartztrauber, “FFT algorithms for Vector Computers,” *Parallel Computing* 1 (1984), pp. 45–63.

[16] Xcode, `http://developer.apple.com/tools/macosxtools.html`

[17] Xgrid, `http://www.apple.com/macosx/features/xgrid/`

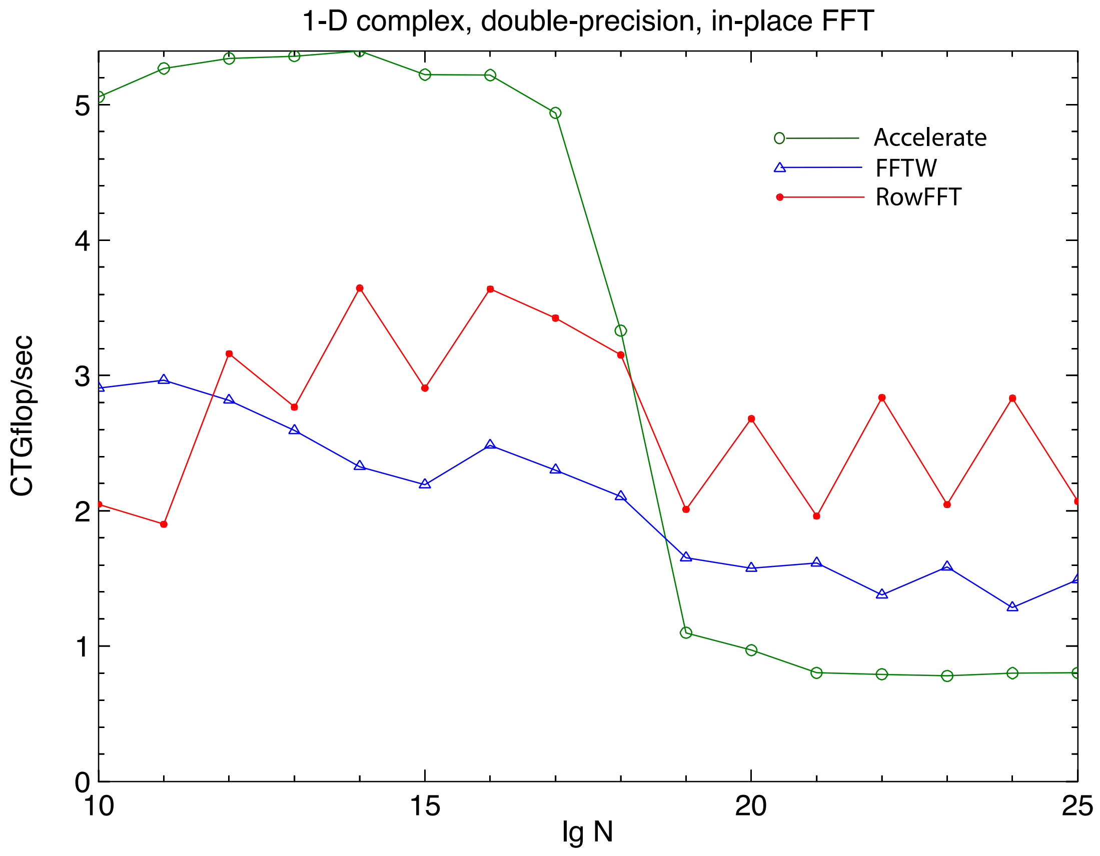


Figure 3: Performance for 1D, complex, double-precision FFT over the three methods: a) direct Accelerate call for length N, b) direct FFTW call for length N, and c) our rowFFT, which is some variant of Algorithm 1 chosen for maximum speed at the given length. For the rowFFT, factorization $N = WH$ is adopted so that all low-level Accelerate FFTs are of length $O(\sqrt{N})$.

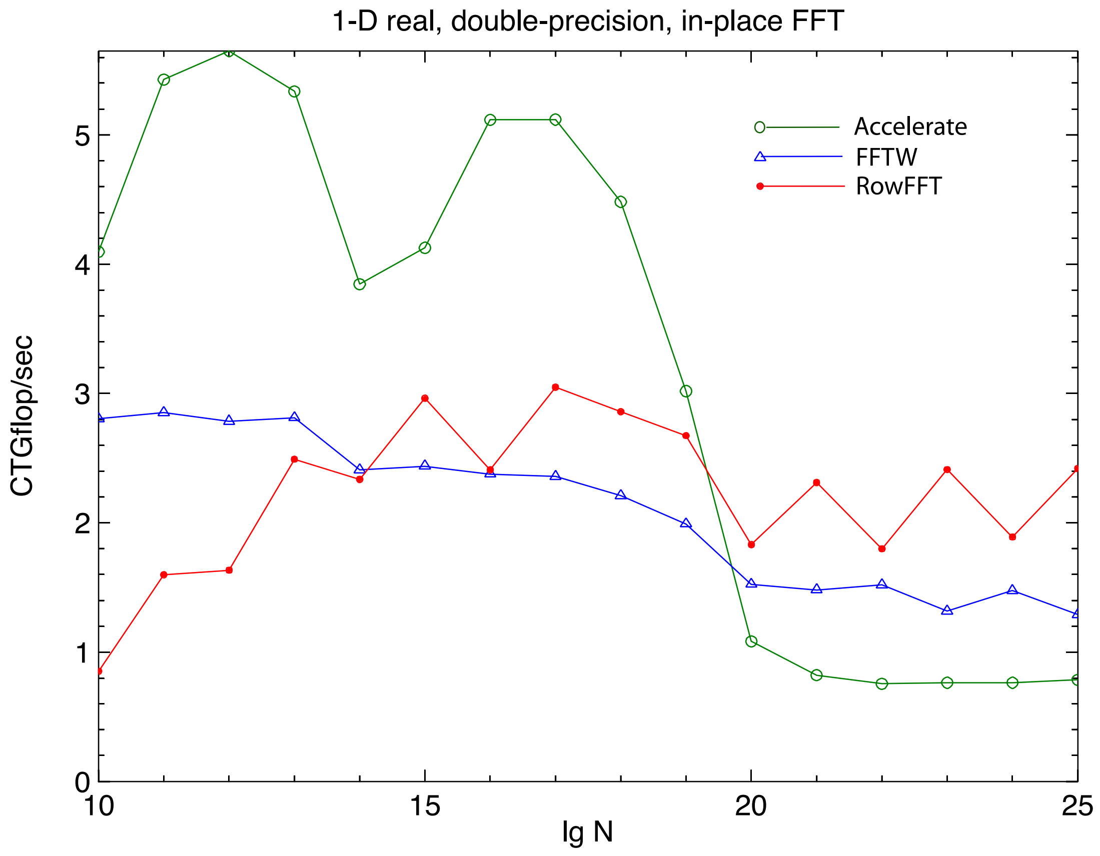


Figure 4: Same as Figure 3, except input signal is pure-real, and so the complexity is about halved. For example, our complex and real rowFFTs each perform at a sustained 2 CTGflop/s for length $N = 2^{25}$, but this means the real-signal FFT (Figure 4, here) is about *twice* as fast in real time.

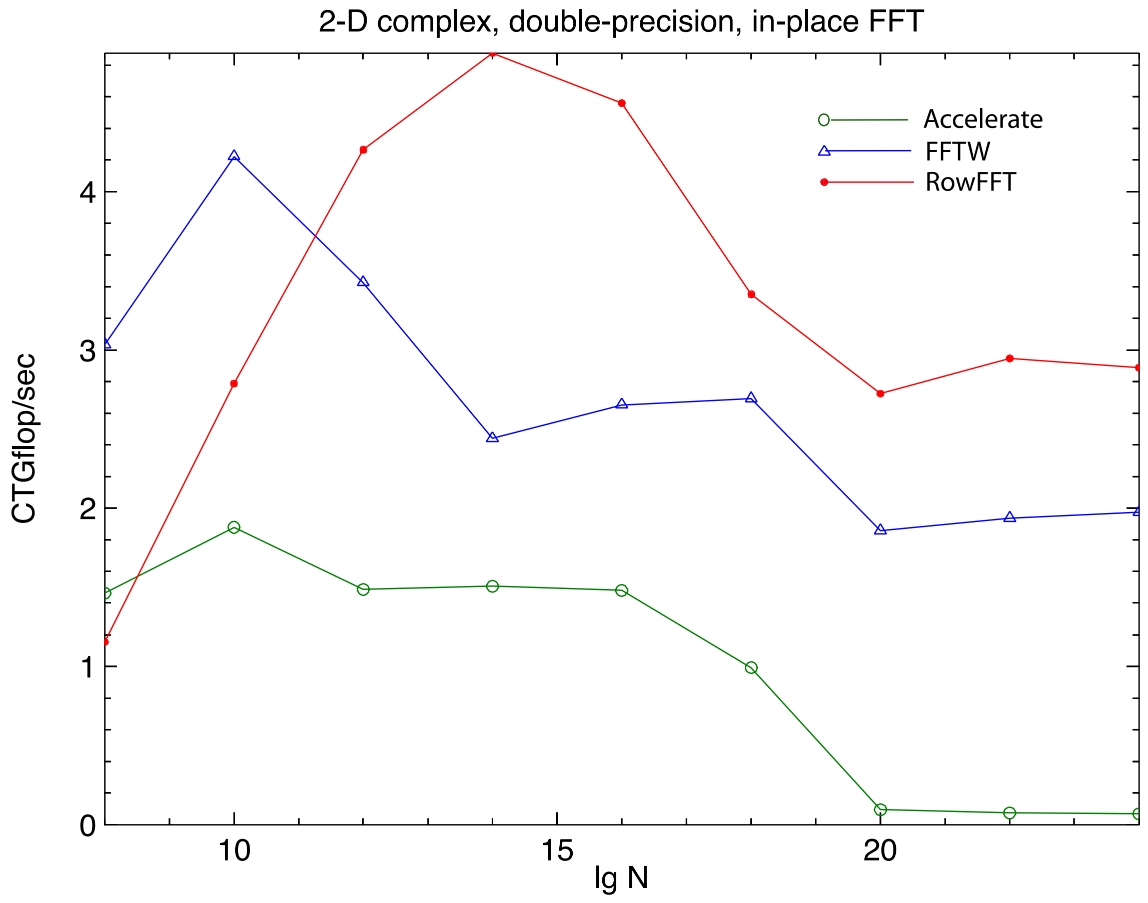


Figure 5: Performance data for 2D double precision, complex FFTs, with $N = 4^k$ and so Algorithm 1 involves a square matrix of side $\sqrt{N}$. Our rowFFT dominates over a wide range of N.

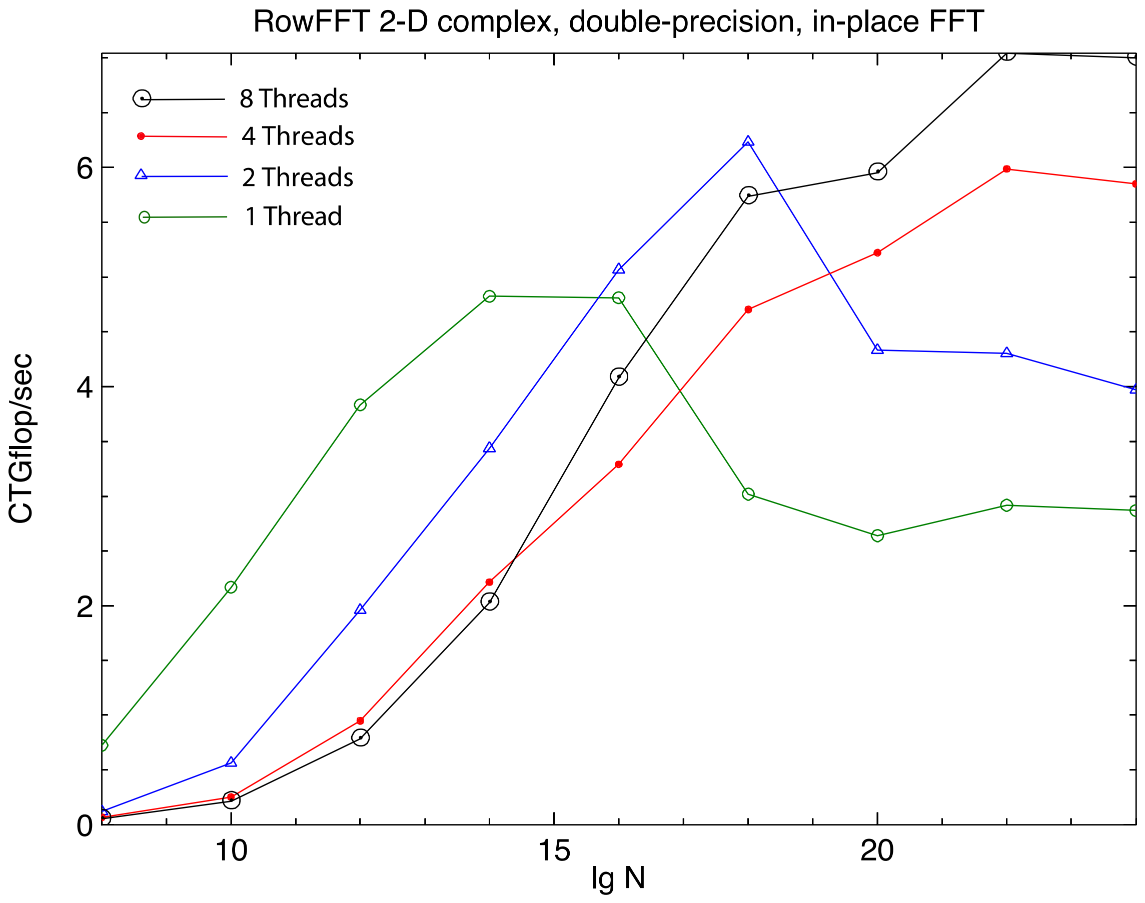


Figure 6: Multithread performance for our rowFFT paradigm. As in Figure 5, N is a perfect square. Note the interesting phenomenon of switching between 1-thread and multi-thread dominance near $N \sim 2^{17}$, and the ultimate 8-thread dominance for the larger N. In fact, 8-thread performance is about 2.5x better than single-thread at large lengths N.

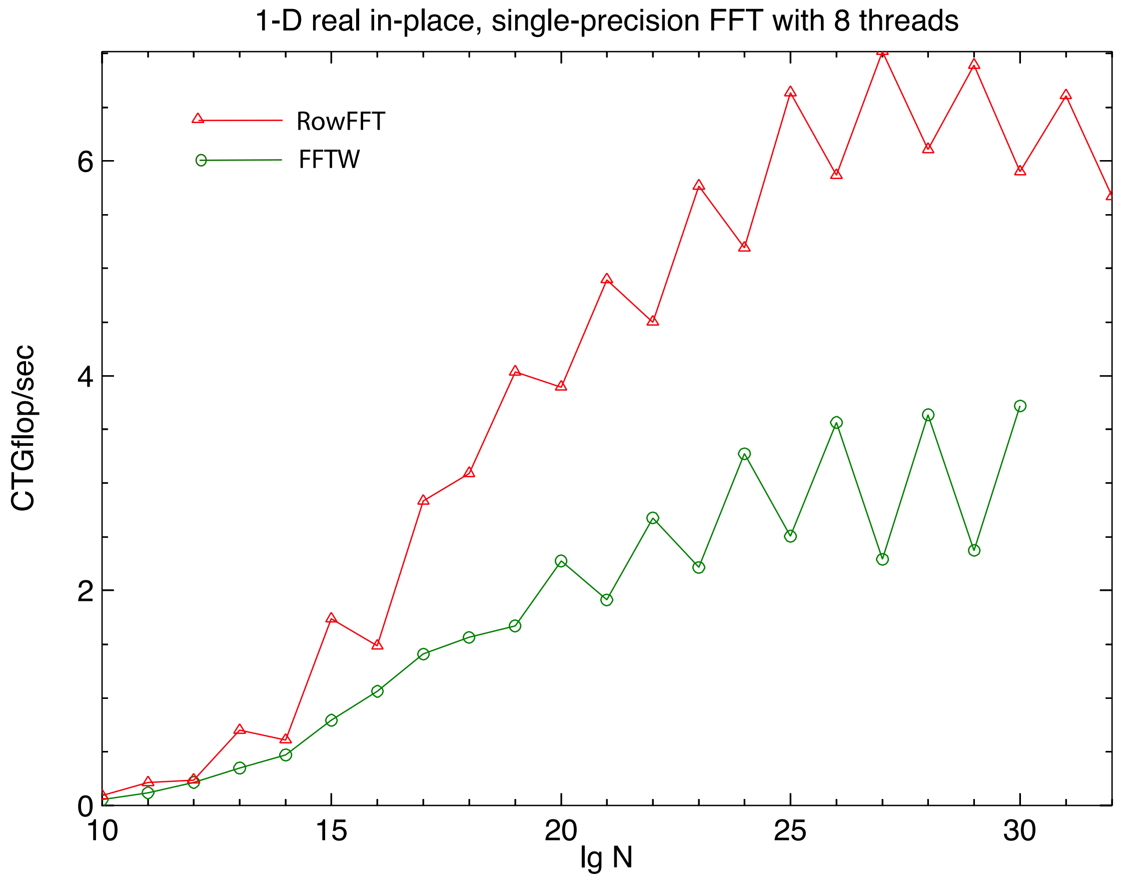


Figure 7: The dominance of rowFFT over standard FFTW for single-precision, real FFT using 8 threads. The fact of rowFFT working through $N = 2^{32}$ is due to true 64-bit routines; FFTW currently does not go that far, being a 32-bit implementation. In any case, the graph reveals no major degradation of CTGigaflop/s numbers even for massive signal lengths N.

1D FFT type	N	Accelerate	FFTW	RowFFT
Complex	2^{10}	**5.06**	2.90	2.05
Complex	2^{11}	**5.27**	2.96	1.90
Complex	2^{12}	**5.34**	2.82	3.16
Complex	2^{13}	**5.36**	2.59	2.77
Complex	2^{14}	**5.40**	2.33	3.64
Complex	2^{15}	**5.22**	2.19	2.91
Complex	2^{16}	**5.22**	2.48	3.64
Complex	2^{17}	**4.94**	2.30	3.42
Complex	2^{18}	**3.33**	2.10	3.15
Complex	2^{19}	1.10	1.65	**2.01**
Complex	2^{20}	0.97	1.57	**2.68**
Complex	2^{21}	0.80	1.62	**1.96**
Complex	2^{22}	0.79	1.38	**2.83**
Complex	2^{23}	0.78	1.58	**2.04**
Complex	2^{24}	0.80	1.28	**2.83**
Complex	2^{25}	0.80	1.49	**2.07**
Real	2^{10}	**4.10**	2.81	0.85
Real	2^{11}	**5.43**	2.85	1.60
Real	2^{12}	**5.65**	2.78	1.63
Real	2^{13}	**5.34**	2.81	2.49
Real	2^{14}	**3.85**	2.41	2.34
Real	2^{15}	**4.13**	2.44	2.96
Real	2^{16}	**5.12**	2.38	2.41
Real	2^{17}	**5.12**	2.36	3.05
Real	2^{18}	**4.48**	2.21	2.86
Real	2^{19}	**3.02**	1.99	2.67
Real	2^{20}	1.08	1.53	**1.83**
Real	2^{21}	0.82	1.48	**2.31**
Real	2^{22}	0.76	1.52	**1.80**
Real	2^{23}	0.76	1.32	**2.41**
Real	2^{24}	0.76	1.48	**1.89**
Real	2^{25}	0.79	1.29	**2.42**

Table 1: 1D, double–precision FFT performance. Under the columns Accelerate, FFTW, and rowFFT (our hierarchical scheme) the numbers are Cooley–Tukey gigaflops/s (CTGflop/s, or CTGs). The boldface maximum speeds show how Accelerate can transition to rowFFT nicely, routinely outperforming the standard FFTW.

2D FFT type	$W \times H$	Accelerate	FFTW	RowFFT
Complex	$2^4 \times 2^4$	1.46	**3.03**	1.15
Complex	$2^5 \times 2^5$	1.88	**4.22**	2.79
Complex	$2^6 \times 2^6$	1.49	3.43	**4.26**
Complex	$2^7 \times 2^7$	1.51	2.44	**4.88**
Complex	$2^8 \times 2^8$	1.48	2.65	**4.56**
Complex	$2^9 \times 2^9$	0.99	2.69	**3.35**
Complex	$2^{10} \times 2^{10}$	0.09	1.86	**2.72**
Complex	$2^{11} \times 2^{11}$	0.07	1.94	**2.95**
Complex	$2^{12} \times 2^{12}$	0.07	1.98	**2.89**

Table 2: 2D, double-precision FFT performance. Again the speed numbers are in CTGs. Note that our hierarchical rowFFT outperforms (boldface speeds) the standard FFTW for sufficiently long signals.

1D FFT type	N	Accelerate	FFTW	RowFFT
Complex	2^{10}	**10.50**	5.77	2.63
Complex	2^{11}	**11.41**	5.67	2.44
Complex	2^{12}	**10.24**	6.14	4.69
Complex	2^{13}	**11.74**	4.28	3.96
Complex	2^{14}	**11.38**	5.87	6.24
Complex	2^{15}	**8.97**	4.71	4.23
Complex	2^{16}	**10.75**	4.54	6.57
Complex	2^{17}	**10.42**	4.78	5.30
Complex	2^{18}	**8.23**	5.69	4.22
Complex	2^{19}	**6.00**	2.96	2.71
Complex	2^{20}	2.61	2.77	**4.21**
Complex	2^{21}	1.89	2.52	**2.60**
Complex	2^{22}	1.69	2.35	**4.31**
Complex	2^{23}	1.58	2.78	**2.81**
Complex	2^{24}	1.59	2.40	**4.47**
Complex	2^{25}	1.58	2.62	**2.97**
Real	2^{10}	**8.19**	5.25	0.97
Real	2^{11}	**11.26**	5.85	1.54
Real	2^{12}	**11.43**	5.26	2.12
Real	2^{13}	**10.98**	5.49	3.66
Real	2^{14}	**8.05**	4.98	3.33
Real	2^{15}	**8.03**	4.89	4.67
Real	2^{16}	**10.14**	4.27	3.70
Real	2^{17}	**9.97**	4.64	4.87
Real	2^{18}	**9.63**	4.56	4.43
Real	2^{19}	**7.61**	4.95	3.89
Real	2^{20}	**5.68**	2.83	2.53
Real	2^{21}	2.53	2.64	**3.48**
Real	2^{22}	1.85	**2.53**	2.41
Real	2^{23}	1.66	2.25	**3.57**
Real	2^{24}	1.56	**2.63**	2.58
Real	2^{25}	1.57	2.31	**3.70**
Real	2^{26}	1.61	2.50	**2.71**

Table 3: 1D, single-precision, in-place FFT performance.

2D FFT type	$W \times H$	Accelerate	FFTW	RowFFT
Complex, In-place	$2^4 \times 2^4$	3.41	**6.30**	1.17
Complex, In-place	$2^5 \times 2^5$	6.71	**7.59**	3.01
Complex, In-place	$2^6 \times 2^6$	6.66	**8.98**	5.62
Complex, In-place	$2^7 \times 2^7$	6.58	5.59	**7.72**
Complex, In-place	$2^8 \times 2^8$	7.35	6.21	**9.26**
Complex, In-place	$2^9 \times 2^9$	**6.75**	5.70	4.74
Complex, In-place	$2^{10} \times 2^{10}$	**4.32**	3.53	4.04
Complex, In-place	$2^{11} \times 2^{11}$	3.27	3.60	**4.46**
Complex, In-place	$2^{12} \times 2^{12}$	3.70	3.95	**4.57**
Complex, In-place	$2^{13} \times 2^{13}$	3.36	3.49	**4.86**

Table 4: 2D, single-precision, in-place FFT performance.

$w \times h$	$k \times k$	vImage time	FFT time
160×120	15×15	0.00059	0.00352
160×120	31×31	0.00109	0.00347
160×120	63×63	0.00502	0.00345
320×240	31×31	0.00185	0.01368
320×240	63×63	0.01039	0.01352
320×240	127×127	0.07226	0.01432
640×480	63×63	0.02067	0.06723
640×480	127×127	0.14269	0.06533
640×480	255×255	1.12207	0.06706
1280×960	31×31	0.01365	0.14122
1280×960	63×63	0.04334	0.15201
1280×960	127×127	0.33376	0.27579
1280×960	255×255	2.26705	0.27755
1280×960	511×511	17.82758	0.27630
4000×4000	63×63	0.22932	1.06677
4000×4000	127×127	1.40919	4.14972
4000×4000	255×255	10.51788	4.22379
4000×4000	511×511	75.23065	4.14980
8000×8000	127×127	2.98060	4.16909
8000×8000	255×255	22.59401	17.31227
8000×8000	511×511	161.04722	17.18709
16000×8000	63×63	1.05044	8.41406
16000×8000	127×127	3.37262	8.46474
16000×8000	255×255	23.11101	16.99576

Table 5: Acyclic convolution performance; times are in “wall time” seconds. The signal dimensions $w \times h$ and convolution/correlation kernel dimensions $k \times k$ are as in Figure 2. The vImage method is literal acyclic convolution. We have endeavored to show (roughly) here where the literal convolution method “breaks over” into a superior FFT method. (See heuristic formula in Section 1 pertaining to such breakover.)

Real	N	end-end RMSE	end-end MXE	fwd RMSE	fwd MXE	CHE
1D double	2^{10}	1.8×10^{-17}	2.2×10^{-16}	1.7×10^{-13}	2.1×10^{-12}	—
1D double	2^{20}	3.9×10^{-18}	2.2×10^{-15}	2.0×10^{-13}	4.7×10^{-12}	—
1D double	2^{31}	8.5×10^{-20}	2.2×10^{-15}	2.0×10^{-13}	5.1×10^{-12}	—
1D single	2^{10}	1.1×10^{-8}	6.0×10^{-8}	3.2×10^{-5}	4.1×10^{-4}	—
1D single	2^{20}	3.6×10^{-10}	6.0×10^{-8}	4.3×10^{-5}	8.5×10^{-4}	—
1D single	2^{31}	9.4×10^{-12}	1.2×10^{-7}	4.3×10^{-5}	8.6×10^{-4}	—
Complex						
1D double	2^{10}	2.9×10^{-16}	9.0×10^{-16}	—	—	1.4×10^{-13}
1D double	2^{20}	5.2×10^{-16}	2.0×10^{-15}	—	—	2.3×10^{-10}
1D double	2^{30}	6.0×10^{-16}	3.1×10^{-15}	—	—	3.6×10^{-7}
1D single	2^{10}	1.8×10^{-7}	4.8×10^{-7}	—	—	2.4×10^{-7}
1D single	2^{20}	2.8×10^{-7}	1.2×10^{-6}	—	—	7.5×10^{-7}
1D single	2^{31}	4.1×10^{-7}	2.8×10^{-6}	—	—	3.1×10^{-6}
2D double	2^{10}	3.4×10^{-16}	1.1×10^{-15}	—	—	7.9×10^{-15}
2D double	2^{20}	4.6×10^{-16}	2.0×10^{-15}	—	—	4.4×10^{-13}
2D double	2^{30}	6.3×10^{-16}	3.4×10^{-15}	—	—	1.6×10^{-11}
2D single	2^{10}	1.1×10^{-7}	2.7×10^{-7}	—	—	1.8×10^{-7}
2D single	2^{20}	2.6×10^{-7}	1.0×10^{-6}	—	—	5.1×10^{-7}
2D single	2^{30}	3.5×10^{-7}	2.1×10^{-6}	—	—	8.9×10^{-7}

Table 6: Error analysis for various of our FFTs.

FFT type	Max size	Wall time (s)	CTGs
Double-precision			
1-D complex out-of-place	2^{29}	15.4	5.01
1-D complex in-place	2^{30}	27.5	5.87
1-D real in-place	2^{31}	31.0	5.36
2-D complex in-place	$2^{15} \times 2^{15}$	27.2	5.93
2-D real out-of-place	$2^{17} \times 2^{13}$	32.0	2.52
Single Precision			
1-D complex in-place	2^{31}	54.9	6.06
1-D real in-place	2^{32}	61.7	5.57
2-D complex in-place	$2^{15} \times 2^{15}$	21.6	7.45
2-D real out-of-place	$2^{17} \times 2^{14}$	48.6	3.42

Table 7: Maximum sizes for in-memory FFTs.

FFT type	$W \times H$	1 CPU	2 CPUs	4 CPUs	8 CPUs
2D Complex	$2^4 \times 2^4$	0.73	**0.12**	**0.07**	**0.05**
2D Complex	$2^5 \times 2^5$	2.17	**0.56**	**0.25**	**0.21**
2D Complex	$2^6 \times 2^6$	3.83	**1.95**	**0.94**	**0.78**
2D Complex	$2^7 \times 2^7$	**4.82**	**3.43**	**2.21**	**2.03**
2D Complex	$2^8 \times 2^8$	**4.81**	**5.07**	3.28	**4.08**
2D Complex	$2^9 \times 2^9$	**3.02**	**6.23**	4.70	**5.74**
2D Complex	$2^{10} \times 2^{10}$	**2.64**	**4.34**	**5.22**	**5.95**
2D Complex	$2^{11} \times 2^{11}$	**2.91**	**4.30**	**5.99**	**7.04**
2D Complex	$2^{12} \times 2^{12}$	**2.86**	**3.97**	**5.85**	**7.00**

Table 8: Multicore performance in CTGs for 2D, complex, in-place, double-precision FFT, via our RowFFT. Boldface CTGs mean that our rowFFT outperforms the corresponding FFTW rating (see next table for FFTW entries).

FFT type	$W \times H$	1 CPU	2 CPUs	4 CPUs	8 CPUs
2D Complex	$2^4 \times 2^4$	4.10	0.10	0.03	0.01
2D Complex	$2^5 \times 2^5$	4.36	0.37	0.15	0.06
2D Complex	$2^6 \times 2^6$	3.34	1.08	0.68	0.28
2D Complex	$2^7 \times 2^7$	2.44	1.55	1.73	1.12
2D Complex	$2^8 \times 2^8$	2.71	2.14	3.09	3.07
2D Complex	$2^9 \times 2^9$	2.69	2.98	4.05	4.63
2D Complex	$2^{10} \times 2^{10}$	1.79	2.97	4.02	4.88
2D Complex	$2^{11} \times 2^{11}$	1.92	2.26	3.38	4.13
2D Complex	$2^{12} \times 2^{12}$	1.96	2.52	2.83	3.22

Table 9: Multicore performance in CTGs for 2D, complex, in-place, double-precision FFT, via FFTW.