

Supercomputer-style FFT library for Apple G4

Richard Crandall and Jason Klivington
Advanced Computation Group, Apple Computer

Abstract. We describe herein a G4 Velocity Engine implementation of fast Fourier transform (FFT) and associated convolution/correlation routines. Though arbitrary signal lengths (i.e. all powers of 2) are handled, our design emphasis is on very long signals (length $N = 2^{16}$ and on into the millions), for which cache considerations are paramount. The core of the library is a particular variant of full-complex FFT that for signal length $N = 2^{10}$ executes at 1.15 gigaflops (500 MHz G4). This cache-friendly, core FFT plays a dominant role in the long-signal cases such as two-dimensional FFT and convolution. More important perhaps than the core performance benchmark is the manner in which one can sift through the myriad prevailing (and new) FFT frameworks, to arrive at a suitable such framework for the Velocity Engine. Presumably these means of adapting algorithms to the Velocity Engine architecture and features will carry over to other engineering problems.

6 Jan 2000
c. 1999, 2000 Apple Computer, Inc.
All Rights Reserved.

1. FFT library design

In conceiving of a “supercomputer-style” FFT library, one considers at least two facets: vectorization of FFT frameworks and allowance for arbitrary (especially very large) signal lengths. Add to these considerations the machine issues of cache, memory access, and on the scientific side issues of convolution and related signal-processing operations. As a basic design we take the 10 fundamental ingredients of such an FFT library to be:

- 1) Complex FFT;
- 2) Real-signal FFT;
- 3) Inverse FFTs of the above;
- 4) Complex cyclic convolution;
- 5) Real-signal cyclic convolution;
- 6) 2-dimensional complex FFT;
- 7) 2-dimensional real-signal FFT;
- 8) Inverse FFTs for these 2-dimensional cases;
- 9) 2-dimensional complex convolution;
- 10) 2-dimensional real-signal convolution.

Moreover we demand that for each of these ingredients, performance does not degrade too harshly for out-of-cache signal lengths. The computational world of large-signal processing is by now fairly involved, and many disparate applications abound these days [Crandall and Fagin 1994][Crandall 1996][Crandall et al. 1999], such applications ranging from massive image processing to computational number theory.

In our library design the signal length for each ingredient is taken as $N = 2^k$, or for the 2-dimensional cases the two lengths are $W = 2^j$, $H = 2^k$. (Below we describe what can be done for lengths not powers of 2; it is well known that any signal length whatsoever can be handled via appropriate power-of-two lengths; e.g., via Rader or Bluestein convolution) The reason we set separately the real-signal cases is, of course, that when the signal is pure-real, or the inverse FFT creates a real signal, the operations can proceed nearly twice as quickly, so the real cases are genuinely distinct. Of course there are many further features a library may well possess, such as alternative signal lengths N (i.e. not restricted to powers of 2), digital filter modules, and so on. But the above ingredients can be considered “essential,” and on that notion we carefully adapted algorithms with a view to G4 Velocity Engine architecture. If one desires other options, one may observe that:

- Non-power-of-2 signal lengths can be done via fast convolution, to maintain the standard $O(N \log N)$ operation complexity, in particular prime-length FFTs can be effected in this fashion;
- Likewise non-power-of-2 cyclic convolution can be performed via zero-padding to an appropriate power of 2;

- Likewise 2-dimensional image convolution can be effected via appropriate zero-padding of the two lengths;
- Either acyclic or negacyclic convolution can be effected via, respectively, zero-padding and domain-twisting based always on the cyclic convolution implementation;

Because of these options, one sees that although the library ingredients do not completely cover the long-signal processing world, said ingredients can at least be used as fundamental routines to extend functionality.

We should admit right off that our conceptual bias in this library design is toward “long-signal processing,” meaning that the we are not addressing the fine art of small-length FFTs or small convolutions; expecting instead to handle signal lengths such as $N = 2^{17}$ for the typical “SETI@home” FFT-intensive signal processing (the SETI-type large-signal applications were in fact a primary motive for the current library design) or $N = 2^{20}$ and on into yet more millions for computational number theory. Likewise we are interested in relatively large 2-dimensional signals (images), and general convolutions (cyclic, acyclic, negacyclic) for lengths into the millions.

Though there are many degrees of freedom in any library design, there are yet many more degrees attendant on the question: “What style of FFT is best?” We speak here of the complex-data FFT (ingredient (1) above), because one straightforward method for the real-signal FFT is to use a complex, half-length FFT, and likewise the convolution functions merely depend in turn on the FFTs. We are aware that there exist direct, real-signal FFTs such as split-radix variants which do not start with a complex FFT *per se*. but it is not clear whether the subsequent slight reduction in computational complexity is realizable within a vector-processing paradigm. Thus we have assumed power-of-2 signal lengths, and arbitrary ones at that. It will turn out that FFTs for very long lengths can use to good effect the specially-tuned, smaller-length FFTs. The next sections describe the options for such recursion, and the manner in which we arrived at an acceptable option for each library ingredient.

2. FFT and convolution/correlation nomenclature

We denote a 1-dimensional signal of length N by:

$$x = \{x_0, \dots, x_{N-1}\}$$

and a 2-dimensional signal of width W and height H by:

$$x = \{x_{jk} : j \in [0, H-1]; k \in [0, W-1]\}.$$

We use W, H notation to remind ourselves that this could be an image signal for example (but it need not be such). We define the general complex discrete Fourier transforms (DFTs) for the above signals by:

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

in the 1-dimensional case, with inverse transform

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

whereas for 2-dimensional signals we use:

$$X_{ab} = \sum_{j=0}^{H-1} \sum_{k=0}^{W-1} x_{jk} e^{-2\pi i (j a / H + k b / W)},$$

with associated inverse transform:

$$x_{jk} = \frac{1}{WH} \sum_{a=0}^{H-1} \sum_{b=0}^{W-1} X_{ab} e^{+2\pi i (a j / H + b k / W)}.$$

The FFT is then, of course, an algorithm for rapid evaluation of the DFT X , and the inverse FFT recovers x from X . In what follows we shall sometimes write:

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

thinking of $FFT()$ as a function for the sake of clarity. In practice, all of our library FFTs will be “in-place,” so that under the FFT a signal x is simply replaced with the DFT X .

In the case that the original x is pure-real, one can create a complex signal:

$$y = \{x_0 + ix_1, x_2 + ix_3, \dots\},$$

in which case one can perform the length- $N/2$ (complex) FFT of y , and use the result to reconstruct the FFT of the original x . These machinations for real signals work out because of the Hermitian symmetry:

$$X_k = X_{N-k}^*, \tag{2.1}$$

valid whenever the signal x is pure-real, which symmetry means that only half the signal length need be involved during FFT. We are aware that the construction of a y signal as above is not the only way to perform a real-signal FFT; for example, there are numerous split-radix and Hartley transforms that can be used to infer the real-signal DFT. However we find that the slight complexity gains of the modern alternatives for pure-real signals are offset by the tremendous advantages of vector processing in regard to the y -signal approach. Thus we have adopted the older paradigm of using the complex y signal’s DFT directly to infer the DFT of the pure-real x . On the other hand, when a 2-dimensional, W -by- H signal (image) x is pure-real, we have the Hermitian symmetry

$$X_{ab} = X_{H-a, W-b}^*. \tag{2.2}$$

Again, as one might expect, such real-signal cases allow the FFT to run nearly twice as fast. The Hermitian algebra for this 2-dimensional setting is much more complicated than for the above 1-dimensional real-signal scenario, but indeed said algebra can be effected with some care, and we have done so.

As for convolution nomenclature, we specify a 1-dimensional cyclic convolution by its n -th element:

$$(x \times y)_n = \sum_{j+k \equiv n \pmod{N}} x_j y_k,$$

where each of x, y has length N , as does the cyclic $x \times y$. The acyclic convolution is the same definition but with equality replacing the equivalence relation " \equiv " (and no modular reduction therefore), in which case the convolution length is $2N - 1$. The acyclic and other forms of convolution such as negacyclic and right-angle varieties can be effected, as is well known, via the cyclic itself. For this reason the cyclic convolution can be deemed fundamental, we concentrate upon it in the library design. The aforementioned supercomputing notions (vectorization, cache, memory issues) are easy to apply in the convolution domain, primarily because of the convolution theorem which states

$$x \times y = FFT^{-1}(FFT(x) \cdot FFT(y)),$$

where " \cdot " here denotes dyadic (spectral, elementwise) multiplication. The two-dimensional cyclic convolution is analogously defined:

$$(x \times y)_{mn} = \sum_{p+q \equiv m \pmod{H} \atop j+k \equiv n \pmod{W}} x_{pj} y_{qk},$$

where each of x, y is a W -by- H signal (image). The convolution theorem also generalizes in the obvious fashion. As for the closely-related notion of signal correlation, we note that the correlation of two 1-dimensional signals can be written:

$$(x \text{ cor } y)_n = \sum_{j-k \equiv n \pmod{N}} x_j y_k,$$

or in spectral form:

$$x \text{ cor } y = FFT^{-1}(FFT(x) \cdot \overline{FFT(y)}).$$

We deem the difference between convolution and correlation to be therefore essentially trivial, in that only the indicated extra conjugations are required on y , $\overline{FFT(y)}$; and if y is pure-real there is just the one conjugation.

It is well known that the 2-dimensional DFT above can be evaluated simply by FFT'ing all the rows, then all the columns of an image; furthermore all of these operations can proceed in-place, so that an entire image can be replaced by its DFT using $O(WH \log(WH))$ operations in this fashion. Important for supercomputer-style vectorization is the notion of using 2-dimensional concepts for 1-dimensional FFTs. Indeed, there are ways to "factor" a 1-dimensional DFT of factored length $N = WH$ into a certain 2-dimensional form, using the algebraic reduction:

$$X = DFT(x) = \left\{ \sum_{j=0}^{N-1} x_j e^{-2\pi i j k / N} \right\}_{k=0}^{N-1} \quad (2.3)$$

$$= \{ \sum_{J=0}^{W-1} \sum_{M=0}^{H-1} (x_{J+MW} e^{-2\pi i MK/W}) e^{-2\pi i JK/N} e^{-2\pi i JL/H} \}_{K+LH=0}^{N-1}.$$

The notation $\{ \}_a^b$ is to be read as in standard summation; i.e. the contents of the braces is unioned from condition a to condition b . Thus the condition counter $K + LH$ in the final expression above is essentially a lexicographic counter: we think of all N data as arranged lexioographically in a W -by- H matrix, whence L is the row- and K the column-index counter. Though rather unwieldy, this DFT factorization reveals that a 1-dimensional DFT, namely X , can be obtained via a certain combination of 1-dimensional DFTs and a specific “twist” operation, as we shall detail in a later section. Thus, while 2-dimensional FFTs are not the same as lexicographic, 1 dimensional versions, they are very nearly the same thing and merely differ by some operations of complexity $O(N)$. This near-equivalence of the two forms brings certain advantages with respect to cache and memory issues, for the matrix format allows a preponderance of localized operations on the data. This is why the factorization scheme has been so popular in the supercomputer sector *per se*.

3. FFT framework options—1-dimensional core-routine scenario

Here we tour briefly some options for FFT framework, arriving eventually at an efficient such framework for G4 Velocity Engine implementation.

For reference purposes, let us simply display without preamble the celebrated Cooley-Tukey FFT, which is the grandparent of all divide-and-conquer algorithms. We display here the decimation-in-time (DIT) variant of this standard FFT:

Algorithm 1: [Cooley-Tukey] in-place, DIT FFT with scramble.

// We assume a complex input signal $x = \{x_0, \dots, x_{N-1}\}$, with $N = 2^n$. The twist phase is to be $-2^{-1}ij/(2m)$ for forward FFT , $+2^{-1}ij/(2m)$ for FFT^{-1} .

```

FFT, FFT-1(x){
    scramble(x);                                // Call the bit-scramble function.
    n = len(x);
    for(m = 1; m < n; m = 2m) {
        for(j = 0; j < m; j++) {
            a = e2-1ij/(2m);
            for(i = j; i < n; i = i + 2m)
                {xi, xi+m} = {xi + axi+m, xi - axi+m};
        }
    }
    return x, x/N;                             // Return x/N for FFT-1 case.
}

scramble(x){
    n = len(x);
    j = 0;
    for(i = 0; i < n - 1; i++) {
        if(i < j) {xi, xj} = {xj, xi};
        k = n/2;
        while(k > j) {
            j = j - k;
            k = k/2;
        }
        j = j + k;
    }
}

```

We note the familiar, traditional features such as the eventual arrival of the DFT data in-place, but only because of an initial bit-scramble via the *scramble()* procedure. It is instructive also to note that the stride of the innermost loop is unsatisfactory for vectorization and contiguous-memory advantage. The stride, being $2m$ as written, is not only not 1 but it is variable. This and other aspects of the Cooley-Tukey FFT will have to be addressed before we arrive at a suitable vectorizable FFT. There is also a decimation-in-frequency (DIF) variant of the above, but possessed of the same limitations such as a *scramble()* procedure (in the DIF case appearing at the end of the *FFT()* function), and the same basic stride variance that is so harmful during vectorization.

It is of course possible to carry out multiple Cooley-Tukey FFTs at once, for example four FFTs at one time on a G4 engine, but that is a direction orthogonal to the present treatment (and besides, we are trying here to arrive at single *FFT()* function not a multiply-parallel one).

An incremental improvement is obtained via the Stockham FFT framework:

Algorithm 2: [Stockham] in-place FFT with buffer but no scramble.

// We assume a complex input signal $x = \{x_0, \dots, x_{N-1}\}$, with $N = 2^n$. The twist phase is as in Algorithm 1.

```

FFT, FFT-1(x){
  for(q = 1; q <= n; q++) {
    L = 2q; r = N/L; L2 = L/2; r2 = N/L2;
    y = x; // Copy entire length-N signal.
    for(j = 0; j < L2; j++) {
      a = ej2 / L;
      for(k = 0; k < r; k++) {
        t = a * yjr2+k+r;
        xjr2+k = yjr2+k + t;
        x(j+L2)r2+k = yjr2+k - t;
      }
    }
  }
  return x, x/N; // Return x/N for FFT-1 case.
}

```

Now we see two major improvements: nonexistence of bit-scramble, and unit stride (in the inner k -loop). Unfortunately the full-array copies must occur on the order of $\log N$ times, which means that although there are no numerical operations involved, still there must be in some sense $O(N \log N)$ memory motions. It turns out we can do better than this, and still preserve unit stride and avoid bit-scramble.

Motivated by the improvements inherent in the Stockham framework, we turn to an improved version which we shall call the “ping-pong” variant of Stockham’s idea, or “PPFFT.” We observe that similar frameworks have appeared before, as in the vast and clever collection of library options (including automatic, dynamic-coding routines, and many variations including non-power-of-two signal lengths) known as “FFTW/FFTPACK” [Frigo and Johnson 1999].

Algorithm 3: PPFFT, in-order, with buffer but no copy.

// We assume a complex input signal $x = \{x_0, \dots, x_{N-1}\}$, with $N = 2^n$. The twist phase is as in Algorithm 1.

```

FFT, FFT-1(x){
    J = 1;
    X = x; Y = y; // Assign ping-pong memory pointers.
    for(k = n; k > 0; k--) {
        m = 0;
        while(m < N/2) {
            a = e2im/N;
            for(j = J; j > 0; j--) {
                Y0 = X0 + XN/2;
                YJ = a(X0 - XN/2);
                ++X;
                ++Y;
            }
            Y = Y + J;
            m = m + J;
        }
        J = 2 * J;
        X = X - N/2;
        Y = Y - N;
        {X, Y} = {Y, X}; // Ping-pong (swap) the pointers, not the data!
    }
    if(n even) return (complex data at X); // Return X/N for FFT-1 case.
    else return(complex data at Y); // Y/N for FFT-1.
}

```

Now we have not only unit stride in the inner j -loop, but we have dispensed with the expensive array copying. There still exists an auxiliary buffer, but that one buffer eliminates, as we have seen, several problems now.

Based on all the above observations, and repeated experiments on variants and sub-variants, we chose the PPFFT Algorithm 3 for our actual core FFT of the library. One reason why the buffer involved is not really expensive is that, as we shall see, very long signal lengths can be reduced to the core length in a certain sense, so that if the core buffer copy is 2^{10} data wide, even a signal length of $N = 2^{16}$ or more will use just that length-1024 buffer during appropriate recursion.

Note the finale of Algorithm 3, in which the parity of the power n in $N = 2^n$ determines which of the two ping-pong buffers X, Y contains the actual FFT. Thus Algorithm 3 is not in-place always (sometimes the required data is in the “other” buffer Y). Yet, we discovered during G4 implementation of Algorithm 3 that the parity decision at the algorithm’s end can be avoided by clever intervention into the last stages of the inner loop. Furthermore the loop allows some additional means of unrolling and convenient vectorization attendant on such unrolling, as discussed in a later section.

The net result for our library was a core FFT routine based on Algorithm 3 and performing at 1.15 gigaflop/s (500 MHz. G4) for length $N = 1024$ (see Figure 1). Moreover this core FFT (which works in fact at any length $N = 2^n$) can be used for very large N as a recursion bottom, so that the cache and locality advantages of this PPFFT can be exploited even for long N . Because, as we have explained, real-signal DFTs can be extracted from half-length complex-signal DFTs, the existence of a suitable core FFT gives rise not only to the long-signal cases but all the real-signal cases as well. Next we turn to a description of how the reduction of long-signal FFTs to the core FFT can be achieved.

4. Building 1-dimensional FFTs from core-complex FFT

We have mentioned that—via Hermiticity relation (2.1)—a real-signal DFT can be calculated on the basis of a half-length complex FFT. One algorithm for this is as follows:

Algorithm 4: Real-signal (forward) FFT from complex FFT.

// We assume a pure-real input signal $x = \{x_0, \dots, x_{N-1}\}$, with $N = 2^n$.

```

FFT(x){
  y = {x0 + ix1, x2 + ix3, ..., xN-2 + ixN-1};
  Y = FFT(y); // Use complex FFT of length N/2.
  S =  $\frac{1}{2} \{Y_k + Y_{N/2-k}\}_{k=0}^{N/2}$ ;
  T =  $\frac{1}{2i} \{Y_k - Y_{N/2-k}\}_{k=0}^{N/2}$ ;
  U =  $\{S_k + e^{-2ik/N} T_k\}_{k=0}^{N/2}$ ;
  return X = {Re(U0), Re(UN/2), U1, ..., UN/2-1};
}

```

Note that the first two returned elements, $Re(U_0), Re(U_{N/2})$ are just the theoretically real-valued $U_0, U_{N/2}$; we are just specifying that only one float value per such element is required. The rest of the returned data starting with U_1 , which is generally a complex, or two-float entity, has the Hermitian symmetry of relation (2.1).

As for a real-inverse FFT, meaning we know *a priori* the result of the inverse transform to be real in theory, the complementary algorithm is:

Algorithm 5: Real-inverse FFT from complex FFT.

// We assume a Hermitian DFT $X = \{X_0, X_{N/2}, X_1, \dots, X_{N/2-1}\}$ as an output from Algorithm 4.

```

FFT-1(X){
  S =  $\frac{1}{2} \{X_k + X_{N/2-k}\}_{k=0}^{N/2-1}$ ;
  T =  $\frac{1}{2} \{(X_k - X_{N/2-k})e^{2\pi i k/N}\}_{k=0}^{N/2-1}$ ;
  x = FFT-1(S + iT);           // Use complex inverse FFT of length N/2.
  return x;
}

```

This real-inverse algorithm might look simpler than its complement, Algorithm 4, but in reality they should take almost exactly the same effort. For example, it is a hidden but important fact that because the real-signal transform (output of Algorithm 4) is stored in Hermitian order $\{X_0, X_{N/2}, X_1, \dots\}$, some extra data movement is required within Algorithm 5, in construction of the S, T signals.

This having been said, and again we admit there are alternative FFT frameworks for pure-real signals and real-inverse transforms, we hereby dispense with the real-signal issue for our library design, assuming thus that the requirements (2), (3), (5) at the start of Section 1—these required library components involving 1 dimension—are now satisfied via a suitably fast, complex FFT. We shall take up the issue of 2-dimensional real-signal cases in a later section.

The PPFFT will, of course, overflow any cache at some signal length. In fact, the gigaflop-level for length $N = 2^{10}$ begins to degrade somewhat, immediately for lengths $2^{11}, 2^{12}, \dots$. The situation can be remedied to a significant extent, especially for signal lengths $N \geq 2^{16}$ and beyond, by using a known supercomputer-motivated expedient: the “four-step” FFT, which involves a matrix decomposition based on a factorization $N = WH$. The basic method was foreshadowed in the work of [Gentleman and Sande 1966] and subsequently developed over the last few decades [Agarwal and Cooley 1986] [Swarztrauber 1987] [Ashworth and Lyne 1988] [Bailey 1990] [Crandall et al. 1999] with, as we have intimated, a special impetus coming during the supercomputing era of the 1980s. The essential idea is to use relation (2.3) to forge a “rows-only” FFT framework in which a certain matrix consists of rows of data. For convenience in what follows, we assume the input data and the eventual DFT are both in *columnwise* order; i.e. we choose a factorization $N = WH$ and assume prearranged data:

$$X = \begin{matrix} & X_0 & X_H & \cdots & X_{(W-1)H} \\ X_1 & X_1 & X_{H+1} & \cdots & X_{(W-1)H+1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ X_{H-1} & X_{H-1} & X_{2H-1} & \cdots & X_{N-1} \end{matrix}$$

which can be thought of simply as a lexicographic arrangement; i.e. the data in actual memory are arranged:

$$X_0, X_H, X_{2H}, \dots$$

and all of this arrangement is to hold also for the resulting $X = \text{FFT}(x)$.

Algorithm 6: Four-step FFT based on matrix decomposition.

// We assume a signal x of length $N = WH$, thought of as a matrix arranged in columnwise order. The twist phase is as in Algorithm 1.

```
FFT, FFT-1( $x$ ) {  
    for( $r = 0$ ;  $r < H$ ;  $r++$ ) FFT, FFT-1( $x + rW$ ); // FFT each row in-place.  
    { $x_{a+bW}$ } = { $e^{-2iab/N} x_{b+aH}$ }; // Transpose and twist the data matrix.  
    for( $r = 0$ ;  $r < W$ ;  $r++$ ) FFT, FFT-1( $x + rH$ ); // FFT each row in-place.  
    return  $x$ ;  
}
```

The name “four-step” caught on because the transpose/twist operation can be thought of as two steps in itself. Incidentally, it is interesting that a 2-dimensional FFT is actually *simpler* than a 1-dimensional FFT, because as we shall see the two-dimensional FFT does not have the twist factor $e^{-2iab/N}$. Of course, the reputation of the 2-dimensional cases being tougher has only, therefore, to do with the normally greater data size of a 2-dimensional data structure.

The primary feature of Algorithm 6 is that only row-FFTs are performed. Thus, if the matrix be factored $N = WH$ such that both dimensions are fairly near cache size, performance enhancement will occur. The only obstacle, then, to very high performance of Algorithm 6 is the costly transpose step. Not only that, to perform FFTs on data that is *not* arranged columnwise, rather in the conventional, lexicographic normal order, one needs a “six-step” FFT, taking the overall form:

```
transpose  $x$ ;  
take FFT( $x$ ) with Algorithm 6;  
transpose  $x$ ;
```

in this way forcing columnwise order at the beginning and unraveling same at the end. But this involves three transposes total, which can be expensive. One way to avoid all transposes is the following, which assumes normal, lexicographic order on input, but leaves the data in columnwise order at the end.

Algorithm 7: Column-ferrying, transpose-free (forward) matrix FFT.

// We assume a signal x of length $N = WH$, thought of as a matrix arranged lexicographically. The twist phase is as in Algorithm 1.

```
FFT(x){
  for( $c = 0$ ;  $c < W$ ;  $c++$ ) {
     $\{y_j\} = \{x_{c+jW}\}_{j=0}^{H-1}$ ;           // Ferry a column out to a singleton row  $y$ .
    FFT(y);                               // FFT the singleton row.
     $\{x_{c+jW}\} = \{e^{-2\pi i c j / N} y_j\}$ ; // Twist-and-ferry the column back.
  }
  for( $r = 0$ ;  $r < H$ ;  $r++$ ) FFT( $x + rW$ ); // FFT each row in-place.
  return x;                             //The DFT is returned in columnwise order.
}
```

Note this is a forward FFT only. For the inverse FFT, we can accept columnwise order, start on the *rows*, then twist-ferry (with conjugate phase factor) to do the columns; essentially just run Algorithm 7 in reverse. The best aspect of Algorithm 7 is that, for Velocity Engine implementation, we can use the following ideas:

- The column ferrying can be done with *vectors*, such as two complex numbers (four components) at a time; i.e. we can ferry two columns at once to two auxiliary rows and perform two rapid FFTs on said rows;
- The Velocity Engine is well-suited for the rapid twist-factor complex multiplication at algorithm center;
- If one insists upon a lexicographic-lexicographic transform, the final columnwise order does need an extra transpose after the whole forward Algorithm 7; however during convolution for example we can *leave* the columnwise order and start an $FFT^{-1}()$ function *assuming* columnwise input, as intimated in the comments right after the Algorithm 7 display;

In our implementation, we did not explore the notion of asymmetrical $W \gg H$ or $H \ll W$, but we did work out a one-step recursion so that every length $N = 2^n$ can eventually involve an exactly square matrix. Clearly if n is an even power, then $N = (2^{n/2})^2$ so each of W, H is simply \sqrt{N} . When the power n is odd, however, we were able to use the following:

Algorithm 8: Reduction of FFT to square-length FFT.

// We assume a signal x of length $N = 2^n$, with n odd.

```
FFT(x){
   $x^{(L)} = \{x_j\}_{j=0}^{N/2-1};$  // Left half-signal.
   $x^{(R)} = \{x_j\}_{j=N/2}^{N-1};$  // Right half-signal.
   $\{x^{(L)}, x^{(R)}\} = \{x^{(L)} + x^{(R)}, x^{(L)} - x^{(R)}\};$ 
   $\{x_j^{(R)}\} = \{x_j^{(R)} e^{-2\pi i j / N}\};$  // Twist operation.
   $X^{(L)} = FFT(x^{(L)});$  // FFT of square length.
   $X^{(R)} = FFT(x^{(R)});$  // FFT of square length.
  return  $X = \{X_0^{(L)}, X_0^{(R)}, X_1^{(L)}, X_1^{(R)}, \dots\};$ 
}
```

Of course, Algorithm 8 is the classical DIF recursion for FFTs, and can also be thought of as an $N/2$ -by-2 matrix FFT, but in any case we explicitly display Algorithm 8 in this treatment for completeness. Indeed, we now have what is—if not an optimal picture—a complete picture for arbitrary-length 1-dimensional FFTs. The overall strategy we adopted then is:

Algorithm X: General strategy for large-signal FFT.

// Next, complex-signal case.

```
If signal length  $N$  is less than a fixed breakover value, invoke Algorithm 3;
Else if columnwise order acceptable, perform Algorithm 7;
Else if  $N = 2^n$  with  $n$  even, perform Algorithm 7 with final transpose;
Else ( $n$  is odd here) perform Algorithm 8;
```

// Next, real-signal case.

Use Algorithms 4,5 which call in turn the complex case.

G4-specific implementation details are discussed in Section 6, with some performance data shown in Figure 1.

5. FFT framework options—2-dimensional scenario

The handling of 2-dimensional DFTs is quite straightforward in the complex-signal case; much more intricate in the real-signal case. For complex signals we have:

Algorithm 9: 2-dimensional FFT.

// We assume a signal x of length $N = WH$, thought of as a matrix arranged lexicographically.

```
FFT(x){
  for( $c = 0$ ;  $c < W$ ;  $c++$ ) {
     $\{y_j\} = \{x_{c+jW}\}_{j=0}^{H-1}$ ;           // Ferry a column out to a singleton row  $y$ .
    FFT(y);                               // FFT the singleton row.
     $\{x_{c+jW}\} = \{y_j\}$ ;                 // Ferry the column back.
  }
  for( $r = 0$ ;  $r < H$ ;  $r++$ ) FFT( $x + rW$ );    // FFT each row in-place.
  return  $x$ ;                               // 2-dimensional DFT in normal order.
}
```

Of course, this is very much like Algorithm 7, for as we have said the 2-dimensional case is essentially the matrix FFT without twist factors (and in fact, without any problems with columnwise ordering at algorithm end).

Next, we give the hard, Hermitian case for 2-dimensional pure-real signals. The Hermitian relation (2.2) is in force, and this means one ought to be able to perform the 2-dimensional FFT twice as fast, and this is indeed so. The following algorithm shows precisely what can be done to halve the work in this real-signal case:

Algorithm 10: 2-dimensional FFT for pure-real signal (image).

// We assume a real signal x of length $N = WH$, thought of as a matrix arranged lexicographically.

```

FFT(x){
  for( $r = 0; r < H; r++$ ) FFT( $x + rW$ );           // Algorithm 4 on each row.
  for( $c = 0; c < 2; c++$ ) {                         //  $c$  indexes real columns
     $\{y_j\} = \{x_{c+jW}\}_{j=0}^{H-1}$ ;               // Ferry a real column out.
    FFT(y);                                         // Real-signal FFT for  $c = 0, 1$ .
     $\{x_{c+jW}\} = \{y_j\}$ ;                         // Ferry the column back.
  }
  for( $c = 1; c < W/2; c++$ ) {                       // note  $c$  now indexes complex columns
     $\{y_j\} = \{x_{c+jW}\}_{j=0}^{H-1}$ ;               // Ferry a complex column out.
    FFT(y);                                         // Complex FFT for these  $c \geq 2$  cases.
     $\{x_{c+jW}\} = \{y_j\}$ ;                         // Ferry the column back.
  }
  return x;                                         // Returning an equivalent Hermitian form  $X^{(h)}$  of the DFT.
}

```

This 2-dimensional FFT has half the complexity of the full complex-signal case, the only drawback being the peculiar order of the final elements. Said ordering upon output of Algorithm 10 turns out to be:

$$X^{(h)} = \begin{array}{cccccc} X_{0,0} & X_{0,W/2} & \text{Re}X_{0,1} & \text{Im}X_{0,1} & \cdots & \text{Im}X_{0,W/2-1} \\ X_{H/2,0} & X_{H/2,W/2} & \text{Re}X_{1,1} & \text{Im}X_{1,1} & \cdots & \text{Im}X_{1,W/2-1} \\ \text{Re}X_{1,0} & \text{Re}X_{1,W/2} & \text{Re}X_{2,1} & \text{Im}X_{2,1} & \cdots & \text{Im}X_{2,W/2-1} \\ \text{Im}X_{1,0} & \text{Im}X_{1,W/2} & \text{Re}X_{3,1} & \text{Im}X_{3,1} & \cdots & \text{Im}X_{3,W/2-1} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \text{Re}X_{H/2-1,0} & \text{Re}X_{H/2-1,W/2} & \text{Re}X_{H-2,1} & \text{Im}X_{H-2,1} & \cdots & \text{Im}X_{H-2,W/2-1} \\ \text{Im}X_{H/2-1,0} & \text{Im}X_{H/2-1,W/2} & \text{Re}X_{H-1,1} & \text{Im}X_{H-1,1} & \cdots & \text{Im}X_{H-1,W/2-1} \end{array}$$

It is evident upon inspection that the above tableau, together with Hermitian symmetry (2.2), completely specifies the standard, 2-dimensional DFT.

We shall not display an explicit real-inverse, 2-dimensional FFT because it is simply the logical reversal of Algorithm 10; namely, do the columns (using Algorithm 5 for real-inverse and Algorithm 3 for complex-inverse) then do the rows, using Algorithm 5.

When it comes to 2-dimensional cyclic convolution, the full-complex case is easy: one just performs the usual three FFTs described in Section 1. For real convolution, the theory is the same but now we have to perform the dyadic product in the correct Hermitian form. The manner, then, of cyclically convolving two pure-real 2-dimensional signals x, y is thus: apply Algorithm 10 twice, to obtain the Hermitian-equivalent transforms $X^{(h)}, Y^{(h)}$, then multiply dyadically then do the inverse algorithm (backtracking of Algorithm 10). The dyadic multiplication can be gleaned directly from the above output tableau for $X^{(h)}$: one

simply uses the tableau of a real-signal DFT X , say, and one of Y , and replaces say Y with the dyadic elements, as in the usual complex arithmetic for a *pair* of components:

$$\begin{aligned} & (ReX_{ab}, ImX_{ab}) \quad (ReY_{ab}, ImY_{ab}) \\ & (ReX_{ab}ReY_{ab} - ImX_{ab}ImY_{ab}, ReX_{ab}ImY_{ab} + ImX_{ab}ReY_{ab}), \end{aligned}$$

but with care taken to handle the real cases (upper left quad of DFT values in the $X^{(h)}$ tableau) more simply.

Because matrix operations figure so strongly into the large-signal FFTs described herein, it is important to summarize the issues attendant on matrix transpose. The four- and six-step FFTs (with 1-dimensional DFT being the goal) and their variants all involve a matrix transpose, unless of course one uses the columnwise form Algorithm 7, which form is relevant to 1-dimensional convolution schemes in which the internal columnwise order wrapping around the dyadic multiply is admissible. The bottom line is: if, for matrix-based, 1-dimensional FFT, one desires normal (lexicographic) ordering of both input signal x and output transform X , then a matrix transpose is necessary somewhere in the procedure. (For 2-dimensional (image) FFTs we have already seen that transposes can be entirely removed via the column-ferrying expedient.) Intuitively speaking, the transpose is the price to pay for the holographic property of the DFT; that is, to achieve the cache-friendly matrix factorization, the penalty is some kind of matrix re-ordering, i.e. transposition. In this regard, we have already mentioned how we always achieve a *square* matrix for any initial signal length $N = 2^n$, in that odd exponents n involve the one-step recursion of Algorithm 8. What does one do for non-square matrices? One answer is simply to write a general matrix transpose. There are block-oriented transpose algorithms [van Loan 1992, and references therein], some of which are partially cache-friendly (we say it that way because whenever the matrix is not square, there is always some kind of extra cache hitting).

6. Velocity Engine implementation details

As was mentioned in Section 3, the ping-pong FFT (PPFFT) was chosen as our fundamental framework for complex FFT implementation. A primary motivating factor for this choice is the unit stride for both input and output data in the PPFFT. The unit stride is important because of the load-store properties of the G4 vector engine, namely, that vectors must be loaded four floats at a time, and from a 16-byte aligned address. A unit stride for input and output data allows us to easily implement a vector-based algorithm which operates on adjacent input elements and generates adjacent output elements. Vector implementation of a Cooley-Tukey style FFT, which requires storing of isolated (i.e. non-adjacent) data elements, would require (at least) one load and one masking instruction to only store one complex value (two floats) using the vector load/store operations. One could have chosen to execute several Cooley-Tukey procedures in parallel, and that is of course an interesting option. We chose instead to concentrate on the vectorization of a single, complex PPFFT.

While the PPFFT proved to be the most favorable implementation, we investigated several other possibilities which showed some promise. And while the PPFFT has the

significant benefit of requiring no bit-scramble step, it is worth mentioning that the Velocity Engine instruction set provides an opportunity for a clever bit-scramble optimization. Other than the obvious load/store operations, the majority of the time spent in the scramble routine goes towards calculating the bit-reversal of the element indices. For example, given a length 2^8 signal, the element at position 25 (00011001 in binary) must be swapped with the element at position 152 (10011000 binary), and these bit-reversed indices must be calculated, typically using some kind of shift/mask/bit-set loop, which, for longer signal lengths, can involve many loop iterations, typically $\log_2 N$ steps. Since this must be performed for all entries, this implies that the bit-reversal algorithm is an $O(N \log N)$ operation when executed on an entire signal of length N .

This same index bit-reversal can be accomplished using precisely six AltiVec instructions, regardless of signal length (and therefore index bit-length). More interesting is the fact that, since it is a vector implementation, as many as sixteen of these bit reversals can be calculated simultaneously, since this bit reversal can be invoked on sixteen one-byte vector elements at a time. (Obviously, for longer signal lengths, the number of indices calculated will decrease, depending on whether the index representation is bounded by a 16-bit or 32-bit limit.) And, while the scramble operation is of secondary interest in optimization of the FFT, this serves as an excellent example of the algorithmic enhancements that are available with the Velocity Engine.

This bit-reversal is accomplished in the following steps, assuming a signal length of 2^{18} . The vector *vNormalIndex* is assumed to be a vector of four unsigned long (32-bit) indices to be bit-reversed. We also assume the following vector definitions:

```
vReverseBytesPermuter = (3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8, 15, 14, 13, 12);
```

```
vReverseNybblePermHi = (0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15);
```

```
vReverseNybblePermLo =
```

```
(0, 128, 64, 192, 32, 160, 96, 64, 16, 144, 80, 208, 48, 176, 112, 240);
```

```
vSignalBitLengthAdjuster = (vector unsigned long)(14);
```

Given these vector definitions, the following AltiVec instructions will result in a bit reversal of the elements in the vector *vNormalIndex*.

```
vReversedBytes =
```

```
vec_perm(vNormalIndex, vNormalIndex, vReverseBytesPermuter);
```

```
vHiNybbles =
```

```
vec_sra(vReversedBytes, (vector unsigned char)(4));
```

```
vHiNybblesSwapped =
```

```
vec_perm(vReverseNybblePermHi, vReverseNybblePermHi, vHiNybbles);
```

```

vLoNybblesSwapped =
vec_perm(vReverseNybblesPermLo, vReverseNybblesPermLo, vReversedBytes);
vReversedIndices =
vec_or(vHiNybblesSwapped, vLoNybblesSwapped);
vReversedIndices =
vec_sra(vReversedIndices, vSignalBitLengthAdjuster);

```

The first permute instruction reverses the order of bytes in each index. Next, each byte is shifted right by four bits, to yield a vector of high nybbles of each byte. The reversed bytes vector is used as a permute vector to generate a vector of reversed low nybbles. Similarly, the high-nybble vector is used as a permute vector to generate a vector that contains the bit-reversed high nybbles. These two reversed-bit nybble vectors are merged with a bitwise or instruction. Finally, the indices are shifted right to adjust for the bit length of the signal length.

At the core of the FFT algorithm is a set of operations commonly referred to as a butterfly. Given two complex input elements a and b , we generate two complex output elements from, say, a DIF butterfly:

$$\begin{matrix} c \\ d \end{matrix} = \begin{matrix} a + b \\ (a - b)e^i \end{matrix}$$

Expanding the equation for the complex value $d = \{d_r, d_i\}$, we have

$$d_r = (a_r - b_r) \cos \quad + (a_i - b_i) \sin$$

$$d_i = (a_r - b_r) \sin \quad - (a_i - b_i) \cos$$

Velocity Engine implementation of these butterfly operations is straightforward. First, it is assumed, since each vector contains four floats, that two butterflies are being calculated simultaneously. The calculation of the sum c is simply accomplished with a vector add operation. The calculation of d is somewhat more involved. Given the input vectors

$$vA = (a_{mr}, a_{mi}, a_{(m+1)r}, a_{(m+1)i})$$

$$vB = (b_{mr}, b_{mi}, b_{(m+1)r}, b_{(m+1)i})$$

and the cos and sin vectors

$$vSin = (\sin \quad, -\sin \quad, \sin \quad, -\sin \quad)$$

$$vCos = (\cos \quad, \cos \quad, \cos \quad, \cos \quad)$$

We can calculate the output vector vD as follows. First, we calculate a difference vector:

$$vDiff = (a_{mr} - b_{mr}, a_{mi} - b_{mi}, a_{(m+1)r} - b_{(m+1)r}, a_{(m+1)i} - b_{(m+1)i})$$

Next, we create a swapped difference vector:

$$vSwappedDiff = (a_{mi} - b_{mi}, a_{mr} - b_{mr}, a_{(m+1)i} - b_{(m+1)i}, a_{(m+1)r} - b_{(m+1)r})$$

Finally, we perform a multiply and a mul-add (actually, two mul-adds, since there is no simple multiply implemented in the floating point domain for the AltiVec instruction set).

$$vButterfly = \text{vec_madd}(vDiff, vCos, vZero)$$

$$vButterfly = \text{vec_madd}(vSwappedDiff, vSin, vButterfly)$$

For the basic PPFFT, we chose to use a lookup table for loading sin and cos values to create the $vSin$ and $vCos$ vectors. While there is a potential cache miss associated with loading these cos and sin values (along with the cost of building the initial table), our focus was on optimization of longer signal lengths for which a recursive square matrix FFT would be used. In this scenario, the low-level FFT would be called repeatedly for the same short signal length, and so for all but the first iteration, the cos and sin lookup table can be assumed to be in cache, and so the cost of loading is very small.

There are other areas of our library, however, where we have instead chosen to use an incremental calculation method for generating sin and cos vectors. Assuming a base angle θ , and an incremental angle $\Delta\theta$, we take advantage of the following relationships. Given $a = 2 \sin(\Delta\theta/2)^2$ and $b = \sin(\Delta\theta)$ we can iterate a certain, convenient recursion formula:

$$\cos(\theta + \Delta\theta) = \cos\theta - a \cos\theta - b \sin\theta$$

and

$$\sin(\theta + \Delta\theta) = \sin\theta - a \sin\theta + b \cos\theta.$$

This is useful for a twist operation that requires a constant angle increment for the cos and sin multipliers. For example, to have cos and sin vectors with elements that range from 0 to 2π in N steps, we start with

$$vCos = (\cos(0), \cos(0), \cos(2\pi/N), \cos(2\pi/N))$$

$$vSin = (\sin(0), -\sin(0), \sin(2\pi/N), -\sin(2\pi/N))$$

$$vIncrementA = (2 \sin(\pi/2N)^2, 2 \sin(\pi/2N)^2, 2 \sin(\pi/2N)^2, 2 \sin(\pi/2N)^2)$$

$$vIncrementB = (\sin(\pi/N), -\sin(\pi/N), \sin(\pi/N), -\sin(\pi/N))$$

Then, to calculate the vectors $vCos$ and $vSin$ for the next iteration, we use the following AltiVec instructions

$$vCos = \text{vec_nmsub}(vCos, vIncrementA, vCos);$$

$$vCos = \text{vec_nmsub}(vSin, vIncrementB, vCos);$$

$$vSin = \text{vec_nmsub}(vSin, vIncrementA, vSin);$$

$$vSin = \text{vec_madd}(vCos, vIncrementB, vSin);$$

which will yield

$$vCos = (\cos(4/N), \cos(4/N), \cos(6/N), \cos(6/N))$$

$$vSin = (\sin(4/N), -\sin(4/N), \sin(6/N), -\sin(6/N)).$$

While this is a convenient way to efficiently calculate incremental cos and sin vectors, it degenerates for long run lengths, especially for $N > 2^{16}$. In these cases, the same incremental calculation technique is used, but in the scalar domain, and using double-precision floats, which provide sufficient precision to support longer run lengths. The updated scalar values are then transferred to the vector domain for use in the twist multiplication.

One enhancement to the basic PPFFT which has shown significant benefits has been the adoption of a “double-butterfly” strategy—essentially a radix-4 optimization. The standard PPFFT framework requires $\log_2 N$ passes through the data, which means a total of $N \log_2 N$ loads and stores of complex elements. We observed, however, that a careful choice of input and output indices would allow us to perform two steps of the ping pong operation simultaneously, thus allowing us to reduce by half the number of required load/store operations. This is of particular importance for longer run lengths, where cache misses contribute to substantial performance degradation.

7. Performance results

Figure 1 shows various options for length- $(N = 2^n)$ complex FFT. The combination of PPFFT (Algorithm 3) breaking over at $N = 2^{15}$ (Algorithm 7) is currently the optimal manifestation of the overall Algorithm X. As for the real-signal cases, summary timing for optimal deployment of Algorithm X (with lexicographic, not columnwise branch) is:

N	μsec (500 MHz clock)
512	17.8
1024	35.6
2048	73.7
4096	177
8192	482
16384	1086
32768	2250
65536	5103
131072	10545
262144	31764
524288	99250
1048576	255072
2097152	459408

As expected, these microsecond timings are fairly close to 1/2 their complex-signal counterparts inferred from Figure 1.

8. The future

During this research on supercomputer-style FFT library design, we observed several optimization paths, any one of which might bring the long-signal performance up yet further. We list such possible future research paths as follows:

- Experiments to determine optimal matrix factorization; i.e., for $N = WH$ we have described means for forcing $W = H$ via one-level recursive FFT, but perhaps a much more asymmetrical factorization is called for. Certainly when one dimension of a matrix at least fits within cache, that is an interesting situation.
- Split-radix techniques will cause all of the timings to drop, ideally by a factor of $4/5$. That is because Cooley-Tukey and Stockham operation counts are $5N \log_2 N$ whereas alternative-radix counts can be as low as $4N \log_2 N$. Note that gigaflop ratings might *not* increase for alternative radices, even though timings should drop; the reason for this discrepancy being, of course, the lower operation count for non-power-of-two radix.
- Perhaps the notion of double-precision cos/sin function updating can be exploited further; i.e., allowing the G4 scalar engine to aid more than we have indicated on side calculations.
- There are various alternative FFT frameworks [van Loan 1992] that should be investigated, along the lines of supercomputer-style array indexing. There is also the very fast indexing scheme of E. Mayer for transpose-less, FFT-based convolution, as reported in [Crandall et al. 1999].

On the basis of such future research directions, it should be possible to “bring up” the performance tail of Figure 1 for very long signal lengths, as we have doen to some extent using the ideas herein.

Acknowledgments

The authors are indebted to G. Miranker, G. Fisher, J. Papadopoulos, E. Mayer and especially A. Sazegari for insight and support relevant to this research. M. Frigo and S. Johnson of the FFTW project (MIT) were quite helpful on various technical issues.

References

- Agarwal R C and Cooley J W 1986, "Fourier Transform and Convolution Subroutines for the IBM 3090 Vector Facility", IBM Journal of Research and Development, vol. 30, p. 145 - 162.
- Ashworth M and Lyne A G 1988, "A Segmented FFT Algorithm for Vector Computers", Parallel Computing, vol. 6 (1988), p. 217 -224.
- Bailey D 1990, "FFTs in External or Hierarchical Memory," *J. Supercomp.* **4** 23-35.
- Crandall R and Pomerance C 2000, *Prime numbers: a computational perspective*, Springer-Verlag, New York (to appear).
- Crandall R 1994b, *Projects in Scientific Computation*, TELOS/Springer-Verlag, New York, Berlin, Heidelberg.
- Crandall R and Fagin B 1994, "Discrete Weighted Transforms and Large-Integer Arithmetic," *Math. Comp.* **62**, 205, 305-324.
- Crandall R, Doenias J, Norrie C, and Young J 1995, "The Twenty-second Fermat Number is Composite," *Math. Comp.*, **64**, 210, 863-868.
- Crandall R 1996, *Topics in Advanced Scientific Computation*, TELOS/Springer-Verlag, New York, Berlin, Heidelberg.
- Crandall R, Mayer E, and Papadopoulos J 1999, "The twenty-fourth Fermat number is composite," manuscript: <http://www.perfsci.com>.
- Frigo M and Johnson S 1999, FFT-oriented website: <http://www.tw.org>
- Gentleman W M and Sande G 1966, "Fast Fourier Transforms – For Fun and Profit", AFIPS Proceedings, vol. 29, p. 563 - 578.
- Swarztrauber P N 1987, "Multiprocessor FFTs", Parallel Computing, vol. 5, p. 197 - 210.
- van Loan C 1992, *Computational Frameworks for the Fast Fourier Transform*, SIAM.

