in Proceedings of the Scalable High Performance Computing Conference, Knoxville, TN, IEEE, May, 1994, pp. 462-469

Computing the Pipelined Phase-Rotation FFT

David R. O'Hallaron, Peter J. Lieu
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

L.P. Withers, Jr., John E. Whelchel
E-Systems, Inc., Melpar Div.
44983 Knoll Square
Ashburn, Virginia 22011

Abstract

The phase-rotation FFT is a new form of the FFT that replaces data movement with multiplications by constant phasor multipliers. The result is an FFT that is simple to pipeline. This paper completes the pipelined design of the the original phase-rotation FFT, provides a fundamental new description of the algorithm directly in terms of the parallel pipeline, and describes a radix-2 implementation on the iWarp computer system that balances computation and communication to run at the full-bandwidth of the communications links, regardless of the input data set size.

1 Introduction

The Fast Fourier Transform (FFT) is an important algorithm with applications in signal processing and scientific computing. A typical real-time signal processing application performs FFTs on a continuous stream of sensor inputs arriving at fixed intervals. A pipeline of FFT stages is a natural approach for processing such an input stream. Unfortunately, conventional FFT algorithms are difficult to pipeline because the input streams are permuted between each pipeline stage. The Whelchel phase-rotation FFT [8] is a *new* form of the FFT that replaces data movement at runtime with multiplications by precomputed constants. The result is an FFT that is simple to pipeline.

The Whelchel phase-rotation FFT [8] derives from the Pease constant-geometry FFT [6], which itself derives from the original Cooley-Tukey FFT [3] expressed in terms of Kronecker products. The phase-rotation FFT of radix r is designed for a pipeline of r parallel data channels. At each time step, in each stage, the pipeline carries the next r data points, one from each channel, into a Discrete Fourier Transform (DFT) kernel. Unlike earlier pipelined FFTs [4, 5], the phase-rotation FFT has the key property that no data is switched across channels, except within the DFT kernel and at the input and output. Thus, if the phase-rotation FFT is implemented in hardware, no commutator switches or multiport memories are needed.

Supported in part by the Air Force Office of Scientific Research under contract F49620–92–J–0131, in part by the Advanced Research Projects Agency under contract MDA972-90-C-0035, and in part by an E-Systems IR&D program. Authors' email addresses: ohallaron@cs.cmu.edu, pjl@cs.cmu.edu, lwithers@melpar.esys.com, jwhelchel@melpar.esys.com.

The phase-rotation approach extends easily to higher radices, reducing memory and latency while preserving the high throughput and parallel shuffling simplicity of lower radix versions. The phase-rotation FFT has also been extended to a vector-radix, multidimensional parallel-pipeline FFT with the same qualities of the one-dimensional algorithm, and without transposes [9].

This paper reports the results of a project to implement the phase-rotation FFT on a parallel computer system. After a brief overview of the phase-rotation concept in Section 2, the paper introduces a parallel-pipeline digit-reversing step that completes the pipelined design of the original phase-rotation FFT [8] in Section 3. Section 4 provides, for the first time, a set of recipes that generate the twiddles and shuffle addresses necessary to implement the algorithm directly in a parallel pipeline. Finally, Section 5 describes fine-grained mapping strategies to implement the N-point radix-2 phase-rotation FFT on the iWarp system, that balance computation and communication to run at the full 40 Mbytes/sec rate of the iWarp physical links for input data sets of any size N.

2 The basic idea

This section introduces the concept of the phase-rotation FFT. Starting with the Pease constant-geometry FFT, we informally derive the pipelined phase-rotation FFT, identifying the key insights along the way.

2.1 Constant-geometry FFT

Figure 1(a) shows the flowgraph for a radix-r N-point decimation-in-frequency (DIF) constant-geometry FFT, with $r = 2$ and $N = r^n = 8$. There are n stages. Each stage computes N/r *kernels*. Each kernel is an operator that performs an r-point DFT. For radix 2, each kernel inputs two complex numbers and outputs two complex numbers. (For simplicity, twiddles and the final digit-reversing shuffle are not explicitly shown in the flowgraph.)

Each stage in the constant-geometry FFT performs an identical perfect stride-by-s shuffle of its data vector, where $s = N/r$. An easy way to define the perfect shuffle is as follows: If the data vector is regarded as an $s \times r$ array, stored in column-major order, then the perfect shuffle simply transposes it into an $r \times s$ array. For example, the following transpose is a stride-by-4 perfect shuffle, for $N = 8$ points and radix $r = 2$:

$$\begin{bmatrix} 0 & 4 \\ 1 & 5 \\ 2 & 6 \\ 3 & 7 \end{bmatrix} \xrightarrow{T} \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}$$

The data items in this example, labeled by their indices in the original column vector, are regarded as equivalent to a 4×2 array composed by a stride-by-4 unstacking of the 8-point column vector. After the transpose, the 2×4 array is equivalent to a new 8-point column vector composed by a stride-by-2 stacking. As we shall see, this transpose creates difficulties when we try to pipeline the constant-geometry FFT. And it is precisely these difficulties that the phase-rotation FFT addresses.

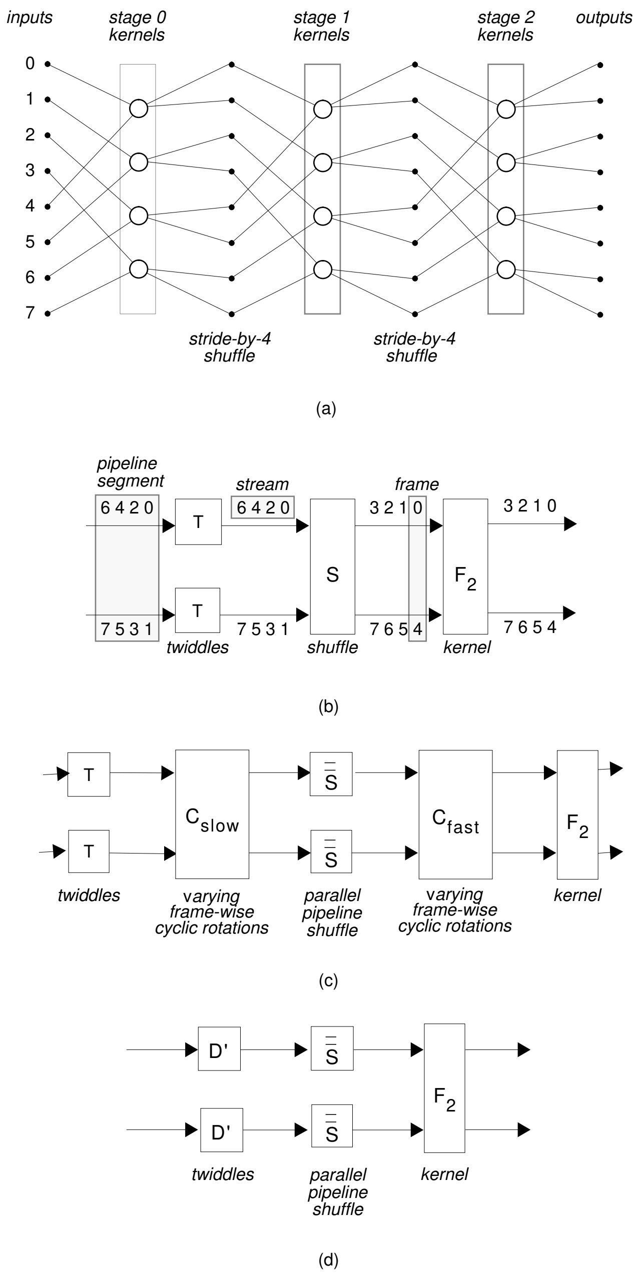

Figure 1: Derivation of the phase-rotation FFT. (a) Initial constant-geometry FFT. (b) Pipelined constant geometry FFT. (c) Pipelined FFT based on cyclic rotations. (d) Pipelined phase-rotation FFT.

2.2 Pipelining the FFT

Each stage of the constant-geometry FFT can be computed on a single processor by pipelining the data. For example, Figure 1(b) shows the pipeline for a single stage with radix $r = 2$. The pipeline consists of a sequence of operators connected by *pipeline segments*. Each pipeline segment consists of r parallel *channels*. Each channel carries a *stream* of N/r data points, which are labeled in this example by their indices from the original column vectors in Figure 1(a). For each pipeline segment, the r data points, each in the same position within its stream, are known as an *r-frame*, or simply, a *frame*. For example, in Figure 1(b), the first frame in the pipeline segment between $\mathbf{S}$ and $\mathbf{F}$ is (0,4), the second frame is (1,5), and so on.

At each time step, the r twiddle operators ($\mathbf{T}$) collectively read a frame (one complex number per operator), perform an element-wise complex multiplication, and write the resulting frame. Notice that each stream is operated on independently. Similarly, the kernel operator ($\mathbf{F}$) reads a frame, computes the radix-r kernel, and writes the resulting frame. In this case, the streams are not independent; each data item in the output frame is a function of every data point in the input frame.

The twiddle and kernel operators pipeline nicely because during each time step they independently read and write a single number from each stream. However, the pipelined shuffle operator ($\mathbf{S}$) is less well behaved. To produce one output frame, the shuffle operator must read and store the r data points from each stream. Thus, $\mathbf{S}$ requires r memory cycles to produce each frame. (Notice that $\mathbf{S}$ transposes the data directly into an $r \times s$ pipeline segment; but even starting with data already in an $r \times s$ pipeline, $\mathbf{S}$ still performs "row-to-column" motions.) This is an example of the *memory-bank conflict* discussed in [7, pp.31-32]. The conflict is clear in Figure 1(b). To assemble its first output frame, $\mathbf{S}$ must read both 0 and 4 from the upper stream to its left. Then it must read 1 and 5 from the lower stream, and so on.

We would like to replace the troublesome perfect shuffle operation with a *parallel-pipeline shuffle*, where each stream is read and written independently and in parallel. The next section describes the insights that make this possible.

$$\begin{array}{ccc}
\begin{bmatrix}0&2&4&6\\1&3&5&7\end{bmatrix} & \xrightarrow{\mathbf{S}} & \begin{bmatrix}0&1&2&3\\4&5&6&7\end{bmatrix} \\
\downarrow \mathbf{C}_{slow} & & \uparrow \mathbf{C}_{fast} \\
\begin{bmatrix}0&2&5&7\\1&3&4&6\end{bmatrix} & \xrightarrow{\overline{\overline{\mathbf{S}}}} & \begin{bmatrix}0&5&2&7\\4&1&6&3\end{bmatrix}
\end{array}$$

Figure 2: Replacing the perfect shuffle with three simpler shuffles.

2.3 The phase-rotation concept

This section describes how to replace the perfect shuffle by a parallel-pipeline shuffle, so that we can access the data streams in parallel. The basic idea is to rotate the data within frames, and then compensate for these motions by phase rotations of the twiddle factors.

We begin with a "detour" around the perfect shuffle. That is, we find a sequence of three simpler shuffles that is equivalent to the perfect shuffle. This idea is shown graphically in Figure 2 for an radix-2 example. Each radix-2 pipeline segment is represented as matrix. Each row in the matrix corresponds to a stream, and each column corresponds to a frame. Frames (columns) are arranged left-to-right in reverse-time order in the matrix.

The first step in Figure 2 is a set of cyclic rotations, called $\mathbf{C}_{slow}$, which rotates each frame. These rotations are frame-wise in the sense that only data points contained in the same frame are rotated across the streams. Notice that in the radix-2 case, half of the rotations leave the corresponding frame unchanged. The next step is a parallel-pipeline shuffle $\overline{\overline{\mathbf{S}}}$, which permutes the data in each stream. Notice that no data points need to be transferred between streams in this step. The last step is another set of frame-wise cyclic rotations in the opposite direction, called $\mathbf{C}_{fast}$, which leave the data in the same order that the perfect shuffle would. Note that $\mathbf{C}_{slow}$ and $\mathbf{C}_{fast}$ change the number of rotations per frame at different paces, one slow and one fast.

If we apply the idea in Figure 2 to each stage of the pipelined FFT in Figure 1(b), replacing each perfect shuffle with three simpler shuffles, we get a pipelined FFT based on cyclic rotations, which is shown in Figure 1(c).

The kind of basic frame-wise rotations in Figure 1(c) that is applied at slow-varying, and then fast-varying rates, is represented in general by the $r \times r$ cyclic (circular) shift permutation matrix $\mathbf{C}_r$, made by permuting the rows of the identity matrix $\mathbf{I}_r$ down by one row, and moving the bottom row up to the top. For example,

$$\mathbf{C}_4 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} .$$

The key insight of the phase-rotation FFT is that the cyclic shift theorem for the DFT can be applied to the cyclic shift operators in Figure 1(c). In matrix form, the cyclic shift theorem for a DFT is the relation

$$\mathbf{F}_r\mathbf{C}_r = \mathbf{D}_r\mathbf{F}_r, \qquad (1)$$

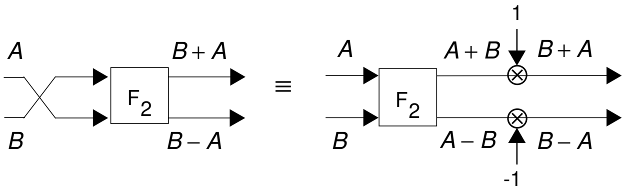

Figure 3: Interpretation of $\mathbf{F}_r\mathbf{C}_r = \mathbf{D}_r\mathbf{F}_r$

where $\mathbf{D}_r = diag(1, \omega, \omega^2, ..., \omega^{r-1})$ is a set of twiddles, and the DFT matrix of size r is

$$\mathbf{F}_r = \frac{1}{\sqrt{r}}(\omega^{jk})_{j,k=0}^{r-1},$$

where $\omega = \exp -\frac{2\pi i}{r}$. For the pipelined FFT, (1) says that phasor multipliers after a DFT kernel give the same effect as a physical data rotation before the DFT kernel. Likewise, a physical rotation after the kernel is equivalent to phasor multipliers before it. The meaning of (1) is shown graphically in Figure 3 for a pipelined radix-2 kernel. The shift theorem implies that the data rotations in Figure 1(c) can be replaced by constant phasor multipliers. These phasors can then be absorbed by the twiddle factors on either side of the kernel, leaving only a parallel-pipeline shuffle. The result is the pipelined phase-rotation FFT, which is shown in Figure 1(d). This completes the informal derivation of the phase-rotation FFT.

The structure of the phase-rotation FFT in Figure 1(d) is similar to the original pipelined FFT in Figure 1(b), except that the phase-rotation FFT's shuffle is simpler. Though the twiddle values have been modified, the arithmetic steps for twiddle and kernel operators are unchanged. The important difference is that the perfect shuffle operator has now been replaced by a parallel-pipeline shuffle that requires no communication across the streams. There is, however, an additional set of twiddles during the final digit-reversing step.

3 Improved phase-rotation FFT

In this section we define an improved version of the original phase-rotation FFT described in [8]. The new version replaces the digit-reversing permutation at the end of the original phase-rotation FFT with a parallel-pipeline shuffle followed by frame-wise cyclic rotations. This last substitution completes the task of pipelining the constant-geometry FFT, so that in every stage, all communication between streams is limited to data points within a single frame.

For radix r and $N = r^n$ points ($n > 1$), the 1-dimensional phase-rotation FFT is a matrix factorization of the N-point DFT matrix $\mathbf{F}_N$. Starting with the Pease constant-geometry factorization, we replace its perfect shuffles $\mathbf{S}$ by $\mathbf{S} = \mathbf{C}_{fast}\overline{\overline{\mathbf{S}}}\mathbf{C}_{slow}$. Similarly, at the left end we replace the radix-r index-digit-reversing permutation $\mathbf{Q} = \mathbf{Q}_{N,r}$ of N data points by $\mathbf{Q} = \mathbf{C}_{slow}^T\overline{\overline{\mathbf{Q}}}\mathbf{C}_{slow}$, where $\overline{\overline{\mathbf{Q}}}$ is another parallel-pipeline shuffle that will be defined formally in Section 4. The phase-rotation FFT is then defined by:

$$\begin{aligned}\mathbf{F}_N &= \mathbf{Q}\cdot\prod_{j=1}^{n}\left(\mathbf{FST}_j\right) = \cdots \begin{pmatrix}\text{vigorous}\\ \text{algebraic}\\ \text{shuffling}\end{pmatrix}\cdots \\ &= \mathbf{C}_{slow}^T\cdot\overline{\overline{\mathbf{Q}}}\mathbf{D}'_{fast}\left[\prod_{j=1}^{n}\left(\mathbf{F}\overline{\overline{\mathbf{S}}}\mathbf{D}'_j\right)\right]\cdot\mathbf{C}_{slow}. \quad (2)\end{aligned}$$

Let $s = N/r$ as before, and $r' = N/r^2$. $\mathbf{F}$ is a direct (tensor, Kronecker) product $\mathbf{I}_s\otimes\mathbf{F}_r = \text{diag}(\mathbf{F}_r, \mathbf{F}_r, ..., \mathbf{F}_r)$. We interpret this as a kernel DFT $\mathbf{F}_r$ operating on s successive frames of r points placed in the pipeline. For $j = 1 : n$, the other parts of (2) are defined by

$$\begin{aligned}
\mathbf{C}_{slow} &= \bigoplus_{k=0}^{r-1}\left(\mathbf{I}_{r'}\otimes\mathbf{C}_r^k\right)\\
\mathbf{C}_{fast} &= \mathbf{I}_{r'}\otimes\left(\bigoplus_{k=0}^{r-1}(\mathbf{C}_r^T)^k\right)\\
\omega_j &= exp\left(-\frac{2\pi i}{r^j}\right)\\
\mathbf{D}_r &= diag(1,\omega_1,\omega_1^2,...,\omega_1^{r-1})\\
\mathbf{D}_{r^{j+1}} &= diag(1,\omega_{j+1},\omega_{j+1}^2,...,\omega_{j+1}^{r^j-1})\\
\mathbf{D}_{slow}^{-1} &= \bigoplus_{k=0}^{r-1}\left(\mathbf{I}_{r'}\otimes\mathbf{D}_r^{-k}\right)\\
\mathbf{D}'_{slow} &= \mathbf{C}_{fast}^T\mathbf{D}_{slow}^{-1}\mathbf{C}_{fast}\\
\mathbf{D}''_{slow} &= \overline{\overline{\mathbf{S}}}^T\mathbf{D}'_{slow}\overline{\overline{\mathbf{S}}}\\
\mathbf{D}_{fast}^{-1} &= \mathbf{I}_{r'}\otimes\left(\bigoplus_{k=0}^{r-1}\mathbf{D}_r^{-k}\right)\\
\mathbf{D}'_{fast} &= \mathbf{C}_{slow}^T\mathbf{D}_{fast}^{-1}\mathbf{C}_{slow}\\
\tilde{\mathbf{T}}_j &= \mathbf{I}_{\frac{N}{r^{(j+1)}}}\otimes\left(\bigoplus_{k=0}^{r-1}\mathbf{D}_{r^{j+1}}^k\right)\\
\mathbf{T}_j &= \mathbf{S}^{j\;T}\tilde{\mathbf{T}}_j\mathbf{S}^j\\
\mathbf{T}'_j &= \mathbf{C}_{slow}\mathbf{T}_j\mathbf{C}_{slow}^T\\
\mathbf{D}'_1 &= \left(\overline{\overline{\mathbf{S}}}^T\mathbf{D}_{slow}^{-1}\overline{\overline{\mathbf{S}}}\right)\cdot\mathbf{T}'_1\mathbf{D}_{fast}^{-1}\\
\mathbf{D}'_j &= \mathbf{D}''_{slow}\mathbf{T}'_j\mathbf{D}_{fast}^{-1}\;, \quad j = 2 : n-1\\
\mathbf{D}'_n &= \mathbf{D}''_{slow}\mathbf{T}'_n = \mathbf{D}''_{slow}\;. \qquad (3)
\end{aligned}$$

The direct sums are of the form

$$\bigoplus_{k=0}^{r-1}\mathbf{A}_k = \text{diag}(\mathbf{A}_0, \mathbf{A}_1, ..., \mathbf{A}_{r-1}),$$

and $\mathbf{A}^T$ denotes the transpose of $\mathbf{A}$. See [9] for more on the basic definitions and relations used to derive (2), as well as the generalization to higher dimension FFTs.

The stages in (2) are counted in reverse time order by the index j. This is in keeping with the fact that (2) is a decimation-in-frequency (DIF) version of the FFT. The transpose of (2), with the product $\prod_{j=n}^{1}$, is the decimation-in-time (DIT) version of the phase-rotation FFT. Also note the extra twiddles $\mathbf{D}'_{fast}$ before digit-reversal in (2). They are always rth roots of unity, so that for radices $r = 2$ and 4 they can be applied without complex multiplication.

A $\mathbf{C}_{slow}$ shuffle and its inverse remain at the input and output ends of the pipeline, respectively. As we have seen, $\mathbf{C}_{slow}$ is a completely frame-wise rotation. It rotates (commutates) the data within each successive frame (column r-vector) of the $r \times s$ pipeline segment for a stage. There is also an implicit frame-wise broadcast within each FFT kernel engine, when an r-point DFT is somehow computed. So in the phase-rotation FFT, data motion is all parallel, except for frame-wise motions at I/O and at every FFT kernel. The simplicity of the phase-rotation FFT is that no data point ever moves both down and across the pipeline in one time-step.

4 Pipeline recipes

While the structure of the pipelined phase-rotation FFT is extremely simple, experience has taught us that generating the appropriate twiddles and shuffle indices from the matrix formulations of (2) and (3) is difficult and confusing. To address this problem, we have developed a collection of recipes for generating the phase-rotation twiddles and shuffle indices off-line. The recipes are defined for any 1D phase-rotation FFT of $N = r^n$ points. Following [7], they are written in a MATLAB-like format.

As we saw in (2), the pipelined phase-rotation FFT performs a typical "twiddle, shuffle, kernel" cycle at each stage. Only the twiddles vary from stage to stage, and there is a digit-reversing shuffle equivalent at the end. To implement this FFT using parallel $r \times s$ pipeline segments (one per stage), we insert the N-vector of input data $\mathbf{x}$ into the pipeline as an $r \times s$ array X: the first r points of $\mathbf{x}$ go into the first frame (column) X, the second r points go into the second frame, and so on. We must also have a shuffle address and a twiddle factor ready for each point in the pipeline. In other words, we would like to fill one $r \times s$ copy A of the pipeline segment with addresses, and another copy D with twiddles.

Then the processors in each stage of the pipeline will know what to do at each time-step $t = 0{:}s-1$. Using the current frame of addresses, they will fetch the current r-frame of data $X(0{:}r-1, A(0{:}r-1,t))$ and the current r-frame of twiddles $D(0{:}r-1, A(0{:}r-1,t))$ (pointwise in parallel), multiply these two frames pointwise, then do an r-point DFT $\mathbf{F}_r$ of the twiddled data frame. That is how each stage $\mathbf{F}\overline{\overline{\mathbf{S}}}\mathbf{D}'_j$ is implemented in the parallel pipeline.

The twiddle and shuffle recipes in this section are "in place" in the sense that they work inside the $r \times s$ pipeline segments that will contain the desired addresses and twiddles. (They are not "in place" in the usual sense, since we will freely use an input and an output copy of a pipeline segment.) This approach avoids constructing and operating with large $N \times N$ matrices (each containing only N non-zero elements). Each parallel-pipeline function recipe is given a name similar to that of the $N \times N$ matrix factor in the FFT (2) that it effectively implements.

4.1 Shuffle recipes

As a convention, pipeline addresses (pipeline array row and column indices) run 0:$r-1$ and 0:$s-1$, respectively. To do parallel-pipeline shuffles, we only need the horizontal (column) addresses, since the data inside each pipe will only jump within that stream (row). The cross-stream shuffles, Cslow and Cfast, are implemented using π_r and its inverse, respectively. π_r is a cyclic rotation of a frame (a vertical slice of the parallel pipeline) that has the effect of $\mathbf{y}_r = \mathbf{C}_r\mathbf{x}_r$. π_r takes a column r-vector $\mathbf{x}_r = (x_0, x_1, x_2, ..., x_{r-1})^T \mapsto \mathbf{y}_r = (x_{r-1}, x_0, x_1, ..., x_{r-2})^T$.

function Y = Cslow(X)
$col = 0$
for $k = 1 : r$
  for $j = 1 : r'$
    $Y(:, col) = \pi_r^k(X(:, col))$
    $col = col + 1$
  end
end

function Y = Cfast(X)
$col = 0$
for $j = 1 : r$
  for $k = 1 : r'$
    $Y(:, col) = \pi_r^{-k}(X(:, col))$
    $col = col + 1$
  end
end

The inverses of Cslow and Cfast are formed simply by reversing π_r in the recipes above. Next, we define some perfect shuffles.

function Y = S(X) !stride by s
$col = 0$
for $row = 0 : r - 1$
  for $k1 = 0 : r : s - r$
    $k2 = k1 + r - 1$
    $Y(row, k1 : k2) = X(:, col)$
    $col = col + 1$
  end
end

function Y = $\mathrm{S}^{-1}(X)$!stride by r
$col = 0$
for $row = 0 : r - 1$
  for $k1 = 0 : r : s - r$
    $k2 = k1 + r - 1$
    $Y(:, col) = X(row, k1 : k2)$
    $col = col + 1$
  end
end

To implement the parallel-pipeline shuffles, $\overline{\overline{\mathrm{S}}}$, $\overline{\overline{\mathrm{S}}}^{-1}$, and $\overline{\overline{\mathrm{Q}}}$, we will use the parallel-pipeline addresses A, which are computed by the following function:

function A = $\overline{\overline{\mathrm{S}}}$_addresses$(r, s)$
$\mathbf{a} = (0, r', ..., (r-1)r')^T$
$col = 0$
for $j = 1 : r'$
  for $k = 1 : r$
    $A(:, col) = \mathbf{a}$
    $col = col + 1$
    $\mathbf{a} = \pi_r(\mathbf{a})$
  end
  $\mathbf{a} = \mathbf{a} + \mathbf{1}_r$

end

function $Y = \overline{\overline{\mathrm{S}}}(X)$
A = $\overline{\overline{\mathrm{S}}}$_addresses$(r, s)$
for $row = 0 : r - 1$
  $Y(row, :) = X(row, A(row, :))$
end

function $Y = \overline{\overline{\mathrm{S}}}^{-1}(X)$
A = $\overline{\overline{\mathrm{S}}}$_addresses$(r, s)$
$[AA, I]$ = sort(A)
for $row = 0 : r - 1$
  $Y(row, :) = X(row, I(row, :))$
end

In the above functions, sort(A) sorts each row of an array A in ascending order. It returns the row-sorted array AA and the corresponding array of addresses I where the successive row elements were found in A. After we have sorted the addresses A for $\overline{\overline{\mathrm{S}}}$, I has the addresses for $\overline{\overline{\mathrm{S}}}^{-1}$.

The pipeline addresses for $\overline{\overline{\mathrm{Q}}}$ are obtained by blockwise perfect shuffles (along the length of the pipeline) of the addresses for $\overline{\overline{\mathrm{S}}}$:

function $Y = \overline{\overline{\mathrm{Q}}}(X, n)$
A = $\overline{\overline{\mathrm{S}}}$_addresses$(r, s)$

if $n > 2$
  for $ns = (n-2) : -1 : 1$
    $stride = r^n s$
    $block = r^{n-2-ns}$! block length
    $col2 = 0$
    for $k_1 = 1 : stride$
      $col1 = (k_1 - 1) * block$
      for $k = 1 : r$
        for $j = 1 : block$
          $B(:, col2) = A(:, col1)$
          $col1 = col1 + 1$
          $col2 = col2 + 1$
        end
        $col1 = col1 + (stride - 1) * block$
      end
    end
    $A = B$
  end
end

for $row = 0 : r - 1$
  $Y(row, :) = X(row, A(row, :))$
end

4.2 Twiddle recipes

Every twiddle matrix $\mathbf{D}$ is diagonal, so it operates on a data vector as a point-to-point vector multiply. Given some permutation matrix $\mathbf{P}$, a new twiddle matrix $\mathbf{PDP}^T$ is equivalent to a rediagonalizing of the vector shuffle of the diagonal of $\mathbf{D}$, that is, $\mathbf{PDP}^T$ = diag($\mathbf{P}*$diag($\mathbf{D}$)). (This is a MATLAB notation: diag() puts the diagonal of a matrix in a vector, and puts a vector in the diagonal of a matrix.) Since we want to perform shuffles within pipeline arrays, we reshape the twiddle N-vector diag($\mathbf{D}$) as an $r \times s$ pipeline array D, just as we originally reshaped the data vector. Then we shuffle the pipelined twiddles, to effect the equivalent of the vector shuffle $\mathbf{P}*$diag($\mathbf{D}$). So we interpret the $\mathbf{PDP}^T$ operator as an in-pipeline shuffle of the pipelined twiddles D, which are then in position to operate on the pipelined data X directly by point-to-point multiplication, $Y = D. * X$. (As mentioned, the data will actually be twiddled frame-by-frame in the pipelined implementation.)

We will interpret the twiddles expressed in (3) this way. Each twiddle function below returns an $r \times s$ array D of twiddle factors (the actual twiddling of the data is not included):

function D_{slow} = Dslow_twiddles(r, s)
$\omega_j = exp(-2\pi i/r)$
$t = 0$
for $j = 0 : (r-1)$
  for $k = 0 : (r'-1)$
    $D_{slow}(:, t) = (1, \omega_r^k, \omega_r^{2k}, ..., \omega_r^{(r-1)k})^T$
    $t = t + 1$
  end
end

function D_{fast} = Dfast_twiddles(r, s)
$\omega_j = exp(-2\pi i/r)$
$t = 0$
for $k = 0 : (r'-1)$
  for $j = 0 : (r-1)$
    $D_{fast}(:, t) = (1, \omega_r^j, \omega_r^{2j}, ..., \omega_r^{(r-1)j})^T$
    $t = t + 1$
  end
end

The inverses of D_{slow} and D_{fast} are just their complex conjugates, and are generated simply by replacing ω_j by ω_j^{-1}. For stages $j = 1{:}n$ (counted down from n), we generate pipelined twiddles $\tilde{T}_j$ by

function $\tilde{T}_j$ = $\tilde{\mathrm{T}}$_twiddles(r, s, j)
$\omega_j = exp(2\pi i/r^{j+1})$
$\omega'_j = \omega^r_{r^{j+1}}$
for $k = 0 : (r-1)$! direct sum loop
  $t_1 = k \cdot r^{j-1}$
  for $p = 0 : (r-1)$
    $\tilde{T}_j(p, t_1) = \omega_j^{kp}$
  end
  $t_1 = t_1 + 1$
  $t_2 = t_1 + r^{j-1}$
  for $t = t_1 : t_2$! fill next column from last
    $\tilde{T}_j(:, t) = \omega_j'^k \cdot \tilde{T}_j(:, t-1)$
  end
end

if $j < n$
  $t_2 = r^j$
  for $k = 0 : (N/r^{j+1})$
    $t_1 = t_2$
    $t_2 = k \cdot r^j$
    $t = 0$
    for $t_0 = t_1 : t_2$
      $\tilde{T}_j(:, t_0) = \tilde{T}_j(:, t)$! copy columns
      $t = t + 1$
    end
  end
end

The rest of the twiddle arrays can now be defined in terms of the shuffles:

D'_{slow} = $\mathrm{S}^{-1}(D^{-1}_{slow})$
D''_{slow} = Cslow(D'_{slow})

D'_{fast} = Cslow(D^{-1}_{fast})

T_j = $\mathrm{S}^{-1}(\tilde{T}_j)$
T'_j = Cslow(T_j)

$D'_1 = \bar{\bar{\mathrm{S}}}(D^{-1}_{slow}). * T'_1. * D^{-1}_{fast}$
if $1 < j < n$
  D'_j = $D''_{slow}. * T'_j. * D^{-1}_{fast}$
end
$D'_n = D''_{slow}$

5 Implementation issues

In this section we describe issues that arise when the phase-rotation FFT is implemented on a real parallel system. In particular, we describe implementation approaches for the radix-2 FFT on the iWarp system. The main result is a scalable implementation of the pipelined phase-rotation FFT that runs at the full 40 Mbytes/second rate of the iWarp physical links.

5.1 iWarp

The iWarp is a private-memory multicomputer developed jointly by Intel and Carnegie Mellon [1]. iWarp systems are 2-dimensional tori of *nodes*, ranging in size from 4 to 1024 nodes. Each node consists of an iWarp *component*, up to 16 Mbytes of off-chip local memory, and a set of 8 unidirectional communication *links* that physically connect the node to four neighboring nodes. Each component is a VLSI chip that contains a *processing agent* and a *communication agent*. The processing agent is a general-purpose load-store microprocessor that runs at a maximum rate of 20 MFLOPs. Thus, a *clock*, or cycle time, is 50 ns. The local memory is accessed at a rate of 160 Mbytes/sec. Each link runs at 40 Mbytes/sec, for a maximum aggregate bandwidth of 320 Mbytes/sec per node.

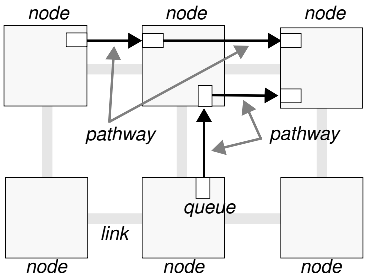

Figure 4: iWarp communication concepts.

The key feature of iWarp is its communication system, which is summarized in Figure 4. Each communication agent contains a set of 20 hardware FIFO *queues*. Each queue can hold up to 8 32-bit words, and can be accessed by user programs at the cost of a register access. iWarp nodes communicate with other nodes using unidirectional point-to-point structures called *pathways*. Each pathway is a sequence of queues. Pathways can be created and destroyed dynamically at runtime. Data traveling along a pathway passes from queue to queue *automatically*, without disturbing the computations on intermediate nodes. Multiple pathways can share the same link by multiplexing in a round-robin fashion, one word at a time. Every pathway on a link that has data to send is guaranteed a proportional fraction of the link bandwidth. Of course, if only one pathway has data to send, then it gets all of the link bandwidth.

5.2 Mapping strategies on iWarp

The problem is to develop a mapping of the flowgraph in Figure 1(d) to an iWarp array. The simplest mapping strategy is to assign each flowgraph node to a unique processor node of a linear array, route the flowgraph arcs through this array, and then embed the resulting linear array in the iWarp torus. This approach, called the PHASE5 mapping because it uses 5 iWarp nodes for each FFT stage, is shown in Figure 5(a).

Each iWarp node in PHASE5 executes a small *node program* that implements its flowgraph operator. Each twiddle node (D') repeatedly reads a complex number from its input pathway multiplies it by the appropriate twiddle (precomputed off-line using the recipes in Section 4.2), and sends the result to its output pathway. Each shuffle operator ($\overline{\overline{S}}$) repeatedly reads a complex data item from its input pathway, stores it in memory, and uses the appropriate shuffle index (again precomputed off-line using the recipes in Section 4.1) to send an appropriate double-buffered data point to the output pathway. The kernel node (F) repeatedly reads two complex numbers from its input pathways, performs the radix-2 DFT kernel operation, and outputs two complex numbers to its output pathways.

Another approach, the PHASE3 mapping, combines the twiddle and shuffle operators on a single node, as shown in Figure 5(b), so that each stage requires 3 nodes instead of 5 nodes. As we shall see, the communication and computation throughputs of the two mappings are identical. The advantage of the PHASE3 mapping is that it is more node-efficient, requiring fewer nodes per stage than

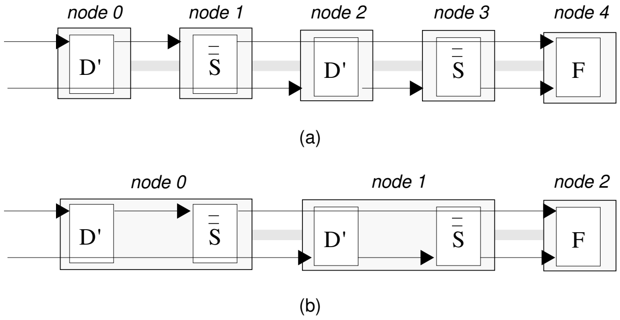

Figure 5: Strategies for mapping one stage of the FFT onto a linear array. (a) PHASE5 mapping. (b) PHASE3 mapping.

the PHASE5 mapping. The advantage of the PHASE5 mapping is its simplicity. Each node is assigned exactly one operator from the flowgraph.

Figure 6 shows a working implementation of a 16K-point radix-2 phase-rotation FFT on a 64-node iWarp array at Carnegie Mellon. The implementation is based on the PHASE3 mapping from Figure 5(b). The large squares are iWarp nodes, labeled with the corresponding operator and stage number, where D is a twiddle/shuffle pair and F is a kernel. The small squares are queues. The arrows are iWarp pathways. As an artifact of our display program intermediate queues are not drawn. Each of the 14 FFT stages uses 3 nodes, with an additional 3 nodes for the parallel-pipeline digit-reversing step at the end.

5.3 Performance

Each iteration of each node program in the PHASE3 and PHASE5 mappings runs in at most 8 clocks. At the peak rate of 40 Mbytes/sec, each link can produce and consume a 32-bit floating-point number every 2 clocks. Further, each data point in the pipeline is a complex number consisting of a pair of 32-bit floating-point words. As a result, each pathway consumes exactly half of the available link bandwidth. Since each link is shared by two pathways, and since the iWarp communication agent gives each pathway an equal share of the link bandwidth, without disturbing the computations on intermediate nodes, each link is fully utilized. The result is a radix-2 FFT that runs at the full 40 Mbytes/sec rate of an iWarp link, regardless of the number of points in the FFT! Since each sample consists of 8 bytes, the FFT runs at a constant rate of 5 Msamples/sec. Given a sufficient number of nodes, the iWarp phase-rotation FFT will produce arbitrarily large FFTs at this rate. Perhaps even more important, the performance is the same on smaller FFTs.

Another way to characterize performance is by computational throughput, expressed in millions of floating-point operations per second (MFLOPS). However, there is a subtlety involved in using MFLOPS as a performance measure. The iWarp phase-rotation FFT performs 16 floating-point operations (2 adds and 4 multiplies by each of the two twiddle operators, and 4 adds by the kernel operator). These 16 floating-point operations per iteration reduce to 10, when one of the twiddles is always 1 and can be omitted, as in the radix-2 Pease and Cooley-Tukey FFTs. This reduced fig-

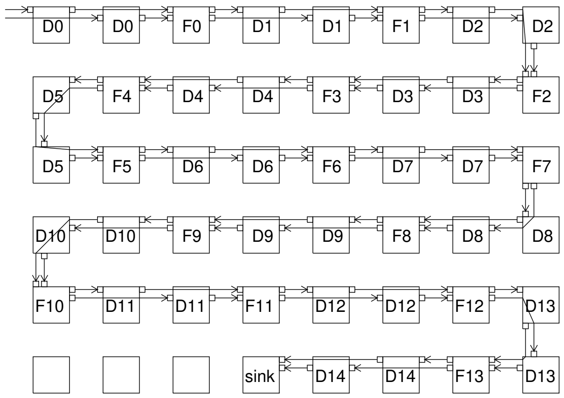

Figure 6: 16K-point pipelined phase-rotation FFT running at 40 Mbytes/sec (350 MFLOPS) on iWarp

ure results in the standard formula for computing FFT MFLOPS, $5N \log_2 N$ floating-point operations per N-point FFT [2]. While one of the two twiddles is always ± 1 in every iteration of the radix-2 phase-rotation FFT, we have not yet discovered an efficient, load-balanced pipeline mapping that takes advantage of this fact. Therefore, to compare the phase rotation FFT fairly with other FFTs, we count its 16 FLOPs per iteration as equivalent to only 10 FLOPs.

Since each node program executes its computation in at most 8 clocks, and since each clock is 50 ns, each stage of the iWarp phase-rotation FFT runs at a rate of 25 MFLOPS for an aggregate performance over all $\log N$ stages of $25 \log N$ MFLOPS. For example, the 16K-point FFT in Figure 6 achieves a measured performance of $25 \times 14 = 350$ MFLOPS (single precision) on the iWarp systems at CMU. By comparison, a highly optimized 16K-point FFT has been measured at 237 MFLOPS (double precision) on a single-processor Cray Y-MP [2, p.114]. The numbers are not directly comparable because of the different floating-point precisions, but they do suggest that the absolute performance of the phase-rotation FFT is quite good.

6 Concluding remarks

We have described an improved version of the Whelchel pipelined phase-rotation FFT, developed recipes for generating the appropriate twiddles and shuffle indices off-line and directly in terms of the parallel pipeline, outlined mapping approaches for the radix-2 case on the iWarp parallel computer, and presented measured performance results of an implementation on iWarp.

The improvement on the original phase-rotation FFT is significant in that it eliminates a potential pipeline bottleneck during the digit reversing step at the end. The twiddle and shuffle recipes should be helpful to the programmer who wants to implement the pipelined phase rotation FFT. The iWarp implementation validates a simple and realistic approach for building scalable pipelined FFTs on a programmable parallel system. Further, the implementation demonstrates that, given a balanced parallel computer architecture with word-level access to the communication links, it is possible to build FFTs that run at the full link bandwidth of the links, even when the FFTs are relatively small.

Other parallel systems are being considered as targets for the multidimensional phase-rotation FFT. For example, the Maspar MP2 provides indirect addressing and routing capabilities, which would facilitate fetching data into and out of the kernel FFTs and performing data communication with large kernels. As another example, the Cray T3D multicomputer, like iWarp, provides direct, low-latency, word-level access to the communication system, which would support the fine grained parallelism found in the phase-rotation FFT.

Acknowledgements

We would like to thank Tom Warfel, LeeAnn Tzeng, Doug Noll, and Doug Smith for their help and suggestions.

References

[1] BORKAR, S., COHN, R., COX, G., GLEASON, S., GROSS, T., KUNG, H. T., LAM, M., MOORE, B., PETERSON, C., PIEPER, J., RANKIN, L., TSENG, P. S., SUTTON, J., URBANSKI, J., AND WEBB, J. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88* (Nov. 1988), pp. 330–339.

[2] CARLSON, D. Ultrahigh-performance FFTs for the CRAY-2 and CRAY Y-MP supercomputers. *Journal of Supercomputing 6* (1992), 107–115.

[3] COOLEY, J., AND TUKEY, J. An algorithm for the machine computation of complex Fourier series. *Mathematics of Computation 19* (Apr. 1965), 297–301.

[4] CORINTHIOS, M. The design of a class of Fast Fourier Transform computers. *IEEE Transactions on Computers C-20* (June 1971), 617–623.

[5] MCCLELLAN, J., AND PURDY, R. Radar signal processing. In *Applications of Digital Signal Processing*, A. Oppenheim, Ed. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[6] PEASE, M. An adaptation of the Fast Fourier Transform for parallel processing. *Journal of the Association for Computing Machinery 15* (1968), 252–264.

[7] VAN LOAN, C. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, PA, 1992.

[8] WHELCHEL, J., O'MALLEY, J., RINARD, W., AND MCARTHUR, J. The systolic phase rotation FFT - a new algorithm and parallel processor architecture. In *Proceedings of ICASSP '90* (Apr. 1990), pp. 1021–1024.

[9] WITHERS, JR., L., AND WHELCHEL, J. The multidimensional phase-rotation FFT - a new parallel architecture. In *Proceedings of ICASSP '91* (May 1991), pp. 2889–2892.