Generating Parallel Programs for Fast Signal Transforms using SPIRAL[*]

J. R. Johnson[†]

1 Introduction

Algorithms for many important signal processing computations can be expressed using linear algebra. This allows alternative algorithms to be written as mathematical formulas. Such formulas can be translated into efficient programs and optimizing the implementation of an algorithm becomes a search problem over the space of mathematical formulas representing the desired computation. The SPIRAL project [9] utilizes this formulation to automatically implement and optimize fast signal transforms.

SPIRAL can be extended to generate and optimize parallel implementations of fast signal transforms. Interpreting such formulas as parallel programs and utilizing formula manipulation to derive and optimize parallel programs was suggested in [5, 7]. An early implementation of some of these ideas was presented in [1].

As a precursor to extending SPIRAL to incorporate parallel computation, we used the Walsh-Hadamard transform (WHT) to develop a prototypical implementation. In this talk we review the WHT package, a self-optimizing package for implementing the WHT, and discuss extensions of the package used to obtain shared-memory and distributed memory parallel implementations. The WHT transform corresponds to a matrix-vector product and alternative algorithms can be obtained by factoring the WHT matrix. By searching over alternate factorizations, we can systematically explore different sequential and parallel implementations of the WHT.

[*]This work was supported by DARPA through research grant DABT63-98-1-0004 administered by the Army Directorate of Contracting.

[†]Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA 19104. email: `jjohnson@mcs.drexel.edu`

2 WHT Package

The Walsh-Hadamard transform of a signal x, of size $N = 2^n$, is the matrix-vector product $\mathbf{WHT}_N \cdot x$, where

$$\mathbf{WHT}_N = \bigotimes_{i=1}^{n} \mathbf{DFT}_2 = \overbrace{\mathbf{DFT}_2 \otimes \cdots \otimes \mathbf{DFT}_2}^{n}.$$

The matrix

$$\mathbf{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

is the 2-point DFT matrix, and $\otimes$ denotes the tensor or Kronecker product.

Let $n = n_1 + \cdots + n_t$, then

$$\mathbf{WHT}_{2^n} = \prod_{i=1}^{t} (\mathbf{I}_{2^{n_1+\cdots+n_{i-1}}} \otimes \mathbf{WHT}_{2^{n_i}} \otimes \mathbf{I}_{2^{n_{i+1}+\cdots+n_t}}) \tag{1}$$

Equation 1 encapsulates $\Theta(\alpha^n/n^{3/2})$, where $\alpha = 4 + \sqrt{8} \approx 6.828427120$, alternative algorithms for computing the WHT. This equation provides a mechanism for exploring different breakdown strategies and combinations of recursion and iteration.

The WHT package [6] (available at http://www.ece.cmu.edu/~spiral), provides an environment for experimenting with alternative algorithms and implementations for computing the WHT. Algorithm alternatives are represented syntactically using a grammar for describing the breakdown strategy.

```
W(n) ::= small[n]  |
     split[W(n1),...,W(nt)]  # n=n1+...+nt
```

The nonterminal symbol `W(n)` gets expanded into a string, called a WHT expression, corresponding to an algorithm for computing $\mathbf{WHT}_{2^n}$. Algorithms are built up from the symbol `small[n]`, which corresponds to a sequence of unrolled straight-line code for computing $\mathbf{WHT}_{2^n}$. Different code generators are provided to explore different unrolled code sequences. The string `split[W(n1),...,W(nt)]` corresponds to an application of Equation 1.

Let $N = N_1 \cdots N_t$, where $N_i = 2^{n_i}$, and let $x_{b,s}^M$ denote the vector $(x(b), x(b+s), \ldots, x(b+(M-1)s))$. Then evaluation of $x = \mathbf{WHT}_N \cdot x$ using Equation 1 is performed using

```
$R = N$;  $S = 1$;
for $i = 1, \ldots, t$
```

```
$R = R/N_i$;
for $j = 0, \ldots, R-1$
  for $k = 0, \ldots, S-1$
    $x^{N_i}_{jN_iS+k,S} = \mathbf{WHT}_N \cdot x^{N_i}_{jN_iS+k,S}$;
$S = S * N_i$;
```

3 Implementation Alternatives and Cache Utilization

Computation of factors of the form $(\mathbf{WHT}_n \otimes I_{2^m})$ access data at stride 2^m and this can lead to poor utilization of cache due to conflict misses and inefficient use of the cache line. Techniques to improve this situation involve dynamically rearranging [11] the data and loop interleaving [2]. Both techniques can be described using mathematical formulas and were introduced into the WHT package by introducing additional the split nodes `splitddl` and `smallil`w, where $0 \leq w \leq 5$ is the interleaving factor.

Strided data access can be converted to sequential data access at the cost of a runtime permutation called a stride permutation. This is accomplished using the identity $(A_m \otimes B_n) = L^{mn}_m (B_n \otimes A_m) L^{mn}_n$, where L^N_d is a permutation matrix corresponding to matrix transposition of an $N/d \times d$ matrix (see [8] for a discussion of matrix transposition algorithms expressed as factorizations of the permutation matrix L^N_d). A `splitddl` node corresponds to

$$\mathbf{WHT}_{2^n} = L^{2^n}_{2^{n_2}} (I_{2^{n_1}} \otimes \mathbf{WHT}_{2^{n_2}}) L^{2^n}_{2^{n_1}} (I_{2^{n_1}} \otimes \mathbf{WHT}_{2^{n_2}}), \tag{2}$$

where $n = n_1 + n_2$. Alternative permutations P can be used instead of L in Equation 2 which may lead to better implementations.

$$\mathbf{WHT}_{2^n} = P^{-1} (I_{2^{n_1}} \otimes \mathbf{WHT}_{2^{n_2}}) P (I_{2^{n_1}} \otimes \mathbf{WHT}_{2^{n_2}}) \tag{3}$$

Using the pseudo code following Equation 1, computation of $\mathbf{WHT}_n \otimes I_m$ repeatedly computes $\mathbf{WHT}_n$ where data is accessed at stride m. This may a cache line to be replaced before all elements are used. This situation can be alleviated by interleaving the computation of $\mathbf{WHT}_n$ over several iterations of the loop. In general $\mathbf{WHT}_{2^n} \otimes I_{2^{m+k}}$ can be written as $(WHT_{2^n} \otimes I_{2^k}) \otimes I_{2^m}$ where the computation of $(WHT_{2^n} \otimes I_{2^k})$, `smallil`$k$ is interleaved.

```
#begin parallel region
R = N;
S = 1;
id = get_thread_id();
num = get_total_thread();
for i = 1, ..., t
    R = R / N$_i$;
    for id = id, ..., R * S - 1, step = num
        j = id / S;
        k = id mod S;
        $x^{N_i}_{jN_iS+k,S} = \mathrm{WHT}_{N_i} * x^{N_i}_{jN_iS+k,S}$;
    S = S * N$_i$;
    #parallel barrier
#end parallel region
```

Figure 1: Pseudo-code of a simple parallel WHT algorithm

4 SMP Implementation

On a shared-memory parallel computer multiple threads can be used to compute the WHT tasks arising in the computation of `split` nodes. The extension of the WHT package to shared-memory multiprocessors was introduced in [3].

The `parallel split` node is similar to the `split` node except that the work is distributed over a collection of parallel threads. Additional code is required to create, manage, and synchronize the threads.

Figure 1 lists the pseudo-code, using OpenMP [10], of a simple parallel WHT algorithm. The inner loop allocates the work (recursive WHT applications) for each stage in the factorization in Equation 1. Since the input from each stage depends on the output from the previous stage, a barrier synchronization is inserted between stages. Alternatively, new threads could be created and joined each iteration of the outer loop with the use of a parallel region. This would simplify the code, but would add substantial overhead due to the repeated initialization of threads.

5 Distributed Memory Implementation

A distributed memory implementation of the WHT can be obtained by partitioning the input vector amongst the processors and interpreting WHT factorizations as distributed computations. The following equation naturally suggests a distributed algorithm.

$$\mathbf{WHT}_{2^n} = \prod_{i=1}^{t} L^{2^n}_{2^{n_i}} (\mathbf{I}_{2^{n-n_i}} \otimes \mathbf{WHT}_{2^{n_i}}) \tag{4}$$

At the i-th stage, $i = 1, \ldots, t$, each processor computes its share of the $\mathbf{WHT}_{2^{n_i}}$ computations arising in $(I_{2^{n-n_i}} \otimes \mathbf{WHT}_{2^{n_i}})$. After the computations the data is permuted amongst the processors corresponding to the permutation $L^{2^n}_{2^{n_i}}$. This computation is denoted by a `dsplit` node in the WHT package.

Additional factorizations of the form

$$\mathbf{WHT}_{2^n} = \prod_{i=1}^{t} P_i (\mathbf{I}_{2^{n-n_i}} \otimes \mathbf{WHT}_{2^{n_i}}), \tag{5}$$

where P_i is a permutation are possible. Different sequences of permutations P_i, $i = 1, \ldots, t$ may lead to different performance characteristics [4] and can be explored by introducing generalized `dsplit` nodes in the WHT package.

References

[1] D. L. Dai, S. K. S. Gupta, S. D. Kaushik, J. H. Lu, R. V. Singh, C.-H. Huang, P. Sadayappan, and R. W. Johnson. EXTENT: A portable programming environment for designing and implementing high-performance block recursive algorithms. In *Supercomputing 1994*, pages 49–58, 1994.

[2] K.-S. Gatlin and L. Carter. Faster FFTs via architecture-cognizance. In *Proceedings of PACT 2000*, October 2000.

[3] J. R. Johnson and K. Chen. A prototypical self-optimizing package for parallel implementation of fast signal transforms. In *International Parallel and Distributed Processing Symposium, IPDPS 2002*, 2002.

[4] J. R. Johnson, R. W. Johnson, C. Marshall, J. Mertz, D. Pryor, and J. Weckel. Data flow, the fft, and the cray t3e. In *SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[5] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuits, Systems, and Signal Processing*, 9(4):449–500, 1990.

[6] Jeremy Johnson and Markus Püschel. In search of the optimal Walsh-Hadamard transform. *ICASSP*, 2000.

[7] R. W. Johnson, C.-H. Huang, and J. R. Johnson. Multilinear algebra and parallel programming. *J. Supercomputing*, 5:189–218, 1991.

[8] S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. Efficient transposition algorithms for large matrices. In *Supercomputing 1993*, 1993.

[9] J. M. F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, and M. M. Veloso. `SPIRAL: Portable Library of Optimized Signal Processing Algorithms`, 1998. http://www.ece.cmu.edu/˜spiral.

[10] OpenMP. *OpenMP C and C++ Application Pragram Interface, Version 1.0*, 1998. `http://www.openmp.org`.

[11] Neungsoo Park and Viktor K. Prasanna. Cache conscious Walsh-Hadamard transform. *ICASSP*, 2001.