IN SEARCH OF THE OPTIMAL WALSH-HADAMARD TRANSFORM

Jeremy Johnson

Mathematics and Computer Science
Drexel University
Philadelphia, PA 19104
jjohnson@mcs.drexel.edu

Markus Püschel

Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
pueschel@ece.cmu.edu

ABSTRACT

This paper describes an approach to implementing and optimizing fast signal transforms. Algorithms for computing signal transforms are expressed by symbolic expressions, which can be automatically generated and translated into programs. Optimizing an implementation involves searching for the fastest program obtained from one of the possible expressions. In this paper we apply this methodology to the implementation of the Walsh-Hadamard transform. An environment, accessible from MATLAB, is provided for generating and timing WHT algorithms. These tools are used to search for the fastest WHT algorithm. The fastest algorithm found is substantially faster than standard approaches to implementing the WHT. The work reported in this paper is part of the SPIRAL project (see http://www.ece.cmu.edu/∼spiral), an ongoing project whose goal is to automate the implementation and optimization of signal processing algorithms.

1. INTRODUCTION

In this paper we present an approach to implementing and optimizing fast signal transforms in general and the Walsh-Hadamard transform (WHT) in particular. We have chosen the WHT because it simple yet important (for applications of the WHT to signal processing and coding theory see [2] an [8] respectively). Fast algorithms for computing the WHT are similar to the fast Fourier transform (FFT) and its variants [6]. The only difference is that there are no twiddle factors and bit-reversal is not necessary. By removing the extra complexity of the twiddle factors and bit-reversal we can concentrate on divide and conquer strategies and iterative versus recursive algorithms.

Our approach to implementing the WHT is to create a flexible software architecture that can be configured to implement many different algorithms. Internally algorithmic choices are represented in a tree structure similar to the plan data structure of the FFTW package [5]. Externally algorithmic choices are described by a simple grammar which can be parsed to create different algorithms that can be executed and timed. This is similar to the approach advocated in the design of TPL, a language for designing and implementing FFT algorithms, and its predecessors [6, 3, 1].

Our package is written in C; however, the external interface allows the user to explore algorithmic choices without writing C code. A MATLAB interface which allows the user to interact and experiment with our package is provided.

After reviewing the WHT in Section 2 and describing our package in Section 3, we summarize empirical data illustrating performance and optimization of the WHT. In our approach, optimizing the WHT becomes a search problem over the space of special class of trees called partition trees. While the space of trees is too large to search exhaustively, we can use dynamic programming and other techniques to prune the search. Our data shows that the performance varies dramatically from algorithm to algorithm and from machine to machine. The search process is similar to what is done by FFTW; however, we allow a larger search space and study it more systematically.

2. THE WALSH-HADAMARD TRANSFORM

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. The tensor product of two matrices

is obtained by replacing each entry of the first matrix by that element multiplied by the second matrix. Thus, for example,

$$\mathbf{WHT}_4 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}.$$

Algorithms for computing the WHT can be derived using properties of the tensor product [9, 6]. A recursive algorithm for the WHT is obtained from the factorization

$$\mathbf{WHT}_{2^n} = (\mathbf{WHT}_2 \otimes \mathbf{I}_{2^{n-1}})(\mathbf{I}_2 \otimes \mathbf{WHT}_{2^{n-1}}) \quad (1)$$

This equation corresponds to the divide and conquer step in a recursive FFT. An iterative algorithm for computing the WHT is obtained from the factorization

$$\mathbf{WHT}_{2^n} = \prod_{i=1}^{n} (\mathbf{I}_{2^{i-1}} \otimes \mathbf{WHT}_2 \otimes \mathbf{I}_{2^{n-i}}), \quad (2)$$

which corresponds to an iterative FFT. More generally, let $n = n_1 + \cdots + n_t$, then

$$\mathbf{WHT}_{2^n} = \prod_{i=1}^{t} (\mathbf{I}_{2^{n_1+\cdots+n_{i-1}}} \otimes \mathbf{WHT}_{2^{n_i}} \otimes \mathbf{I}_{2^{n_{i+1}+\cdots+n_t}}) \quad (3)$$

This equation encompasses both the iterative and recursive algorithm and provides a mechanism for exploring different breakdown strategies and combinations of recursion and iteration. Alternative algorithms are obtained through different sequences of the application of Equation 3. Each algorithm obtained this way can be represented by a tree, called a partition tree. The root of the partition tree corresponding to an algorithm for computing $\mathbf{WHT}_N$, where $N = 2^n$ is labeled with n. Each application of Equation 3 corresponds to an expansion of a node into children whose sum equals the node. Figure 1 shows the trees for a recursive and iterative algorithm for computing $\mathbf{WHT}_{16}$.

In this paper we explore all WHT implementations corresponding to all possible partition trees. The total number of partition trees of size n is given by the recurrence

$$T_n = 1 + \sum_{n_1+\cdots+n_k=n} T_{n_1} \cdots T_{n_k} \quad (4)$$

Table 1 lists the first few values of T_n. The generating function, $T(z)$, for T_n satisfies the functional equation

$$T(z) = z/(1-z) + T(z)^2/(1-T(z)), \quad (5)$$

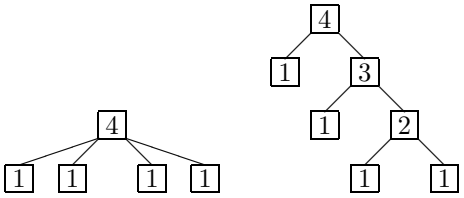

Figure 1: Partition Trees for Iterative and Recursive WHT Algorithms

n	1	2	3	4	5	6	7	8
T_n	1	2	6	24	112	568	3032	16768

Table 1: Number of Partition Trees for $\mathbf{WHT}_{2^n}$

and consequently $T_n = \Theta(\alpha^n/n^{3/2})$, where $\alpha = 4 + \sqrt{8} \approx 6.828427120$. Even if we restrict to binary partition trees, the number of trees is $\Theta(5^n/n^{3/2})$. Hence it is impossible to exhaustively search all possible trees for the optimal algorithm.

3. DESIGN OF THE WHT PACKAGE

The design of our WHT package is based on the design of an FFT package from the thesis of Sebastion Egner [4], and is similar in spirit to the design of FFTW [5].

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 $\mathbf{WHT}_{2^n}$. The string `split[W(n1),...,W(nt)]` corresponds to an application of Equation 3. For example, the strings

```
split[small[1],small[1],small[1],small[1]]

split[small[1],split[small[1],
                     split[small[1],small[1]]]]
```

corresponds to the iterative and recursive algorithms for computing $\mathbf{WHT}_{16}$ depicted in Figure 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 3 is performed using

$$R = N; \quad S = 1;$$

$$
\begin{array}{l}
\text{for } i = 1, \ldots, t \\
\quad R = R/N_i; \\
\quad \text{for } j = 0, \ldots, R-1 \\
\quad\quad \text{for } k = 0, \ldots, S-1 \\
\quad\quad\quad x^{N_i}_{jN_iS+k,S} = \mathbf{WHT}_N \cdot x^{N_i}_{jN_iS+k,S}; \\
\quad S = S * N_i;
\end{array}
$$

This scheme assumes that the algorithm works in-place and is able to accept stride parameters.

Several code generators are provided for producing code for `small[n]`. The resulting functions, similar to the codelets in FFTW, are straight-line code sequences without the overhead of loop control or recursion. The elimination of the control overhead makes them more efficient for small transform sizes than general purpose code; however, the size of the instruction cache eventually causes straight-line code to become slower.

A parser is provided for reading WHT expressions and translating them into a data structure called a WHT tree, which is a partition tree with additional information and is related to an FFTW plan. Each algorithm described by a WHT tree can be used to compute the WHT using an apply function.

Timing and verification programs are provided, which take as input a WHT expression. The timing program can be accessed through MATLAB. MATLAB programs to generate WHT expressions and analyze the resulting algorithms are available.

4. THE SEARCH FOR THE OPTIMAL WHT ALGORITHM

This section summarizes our performance experiments and outlines our search for the optimal WHT algorithm. The key observations are: (1) There is a wide distribution of computing times for different algorithms. (2) The optimal algorithm depends on the computer and compiler. (3) Substantial performance improvement can be obtained by choosing an appropriate algorithm. (4) By providing an environment to generate algorithmic choices and intelligently search for the one with optimal performance it is possible to take advantage of the previous 3 observations.

The WHT package is written in C and is available for download at [7]. Computer experiments were performed on a 233 MHz Pentium II running Linux and a 400 MHz UltraSPARC II running Solaris. On the Pentium gcc version 2.7.2.3 with -O6 was used, and on the UltraSPARC version 5.0 of Sun's C compiler with -fast -xO5 was used. All algorithms operated in-place on double precision real data.

In our first experiment, we compared the iterative algorithm from Equation 2 with the recursive algorithm from Equation 1. From Figure 2 we see that initially the iterative algorithm is faster, but eventually the recursive algorithm becomes faster. The switch over point occurs when the input no longer fits in L2 cache. The recursive algorithm utilizes cache better, however, the overhead for recursion seems to be significant for small sizes.

Figure 2: Ratio of Recursive/Iterative on Pentium

Further improvement can be obtained by using larger unrolled code sequences. We generated straight-line code for small transform sizes and observed improvement over the iterative algorithm up to size 2^8 on both platforms. Code generators based on a recursive algorithm were superior (due to better register utilization) optimization flags The amount of improvement, up to a factor of 7.4, depended on the compiler. We observed that Sun's C compiler performed significantly better than gcc.

The first two experiments suggest a recursive design that switches to an iterative algorithm inside the cache boundary which in turn is built from straight-line code for transforms up to size 256. However, performance can be highly unpredictable and consequently the best thing to do is systematically search all possible implementations to determine the optimal combination of straight-line code, iteration, and recursion.

Since it is infeasible to do an exhaustive search we used dynamic programming to search for an optimal algorithm. We assumed that the optimal algorithm for a given transform size is independent of the context in which it is being called. Under this assumption, we can search for the optimal algorithm for a given size by considering all possible applications of Equation 3 with the previously determined best algorithms used

Figure 3: Distribution of all Split Times on Pentium

for recursive evaluations. The number of cases for size n is equal to 2^{n-1}, the number of ordered partitions of n. Figure 3 shows the times compared to the best time for all possible splits for $n = 12$. The same experiment run on the Sun produces a figure with different characteristics.

When $n > 12$ examing all possible splits becomes too expensive. Thus we restricted dynamic programming to binary splits (there are only $n(n-1)/2$ binary splits). Up to size 12 there was no penalty for this restriction; however, we expect additional improvements when considering non-binary trees for larger sizes. The optimal binary split on the Sun for $n = 20$ was

```
split[split[split[small[5],small[5]],
            small[5]],small[5]]
```

and the resulting computing time was 1.4875e-01 seconds, while the optimal split on the Pentium was

```
split[small[2],split[small[2],split[small[2],
split[small[2],split[small[2],split[small[5],
small[5]]]]]]]
```

with computing time equal to 6.4500e-01. Compared to the iterative algorithm the improvement was a factor of 10.4 on the Sun. Observe that for the Sun the tree has a leftmost expansion, while the tree for the Pentium has a rightmost expansion.

On both machines `small[n]` was optimal up to size $n = 7$ (small[8] was not optimal despite being faster than the iterative algorithm). Despite this, the large values of small were not utilized when building larger transforms. On the Pentium all leftmost factors in the optimal formulas were equal to 2 when the size of the input no longer fit in L1 cache. When comparing the Pentium to the Sun we see that the ratio of the runtimes of the optimal formulas is much larger than the ratios in CPU speed.

Finally we remark that the dynamic programming assumption is not always true. We observed that the runtimes of `small[n]` depend upon the stride at which they are applied. Nevertheless, so far we have been unable to find substantially better algorithms than those determined by dynamic programming. Additional experimental results and details are available from [7].

5. REFERENCES

[1] L. Auslander, J. R. Johnson, and R. W. Johnson. Automatic implementation of FFT algorithms. Technical report, Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA, June 1996. `http://www.ece.cmu.edu/~spiral/tpl.html`.

[2] K.G. Beauchamp. *Applications of Walsh and related functions*. Academic Press, 1984.

[3] 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.

[4] S. Egner. *Zur Algorithmischen Zerlegungstheorie Linearer Transformationen mit Symmetrie*. PhD thesis, Univ. Karlsruhe, Informatik, 1997.

[5] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *ICASSP '98*, volume 3, pages 1381–1384, 1998. `http://www.fftw.org`.

[6] 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.

[7] J. R. Johnson and Markus Püschel. WHT: An adaptable library for computing the Walsh-Hadamard transform, 1999. `http://www.ece.cmu.edu/~spiral/wht.html`.

[8] F.J. MacWilliams and N.J. Sloane. *The theory of error-correcting codes*. North-Holl. Publ. Co., 1992.

[9] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform*, volume 10 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.