AN EFFICIENT RECURSIVE ALGORITHM AND AN EXPLICIT FORMULA FOR CALCULATING UPDATE VECTORS OF RUNNING WALSH-HADAMARD TRANSFORM

Barzan Mozafari, Mohammad H. Savoji

Shahid Beheshti University
Electrical and Computer Engineering Department
Evin, 1983963113, Tehran, Iran
{mozafari,m-savoji}@sbu.ac.ir

ABSTRACT

Walsh-Hadamard transform (WHT) has many applications in digital signal processing including bioinformatics. While there are efficient algorithms for implementing this transform such as Fast WHT (FWHT), performing WHT continuously on a sliding window over a long sequence is time consuming. As it is not reasonable to compute a separate WHT upon arrival of every new sample, another implementation named running WHT (RWHT) has been introduced in [1] which needs to have some update vectors pre-calculated. In this paper we report on an efficient recursive algorithm to find these vectors. Also we propose an easy-to-compute explicit formula for computing coefficients of these update vectors in a computation time independent of the vector size. A proof of the formula and a comparison between the proposed recursive algorithm and an algorithm based on the formula are also given.

1. INTRODUCTION

Many computations in signal processing can be formulated using linear algebra to allow finding alternative algorithms. Such mathematical formulae can be translated into efficient programs whose optimizations become a search problem in the space of formulae representing the desired computation. Some research foci such as SPIRAL project [2] utilize these formulations to automatically implement and optimize fast signal transforms. The simpler the formula is, the more efficient implementation is possible, especially when we can find an explicit formula instead of a recursive method.

Several researches have been carried out on finding efficient algorithms for computing WHT and optimizing its implementation in different situations. While traditional FWHT implementation, like in [3], is a good choice for usual applications, there are some other interests to reformulate the implementation algorithms, e.g. [2] and [4] utilize parallel implementations to achieve faster transforms, or [5] finds the optimal implementation of WHT using its definition in terms of Tensor products. Also [6] analyzes WHT in the benefit of cache memory utilization. Although there are general attempts like [7] for efficiently transposing large matrices in the memory to convert 'strided' data access to sequential, still memory and running time considerations arise when we encounter a large volume of input data or a large number of iterations.

One of these important cases where we need a specialized algorithm is when we have a large input sequence $X = \{\ldots, x_{i+1}, x_i, x_{i-1}, \ldots\}$ and need to compute WHT on a sliding window of N samples such as in power spectrum analysis of DNA sequences [8].

Consider a sliding window of size 2^n over the above large data stream for which we are interested to perform a running WHT. If the left edge of our sliding window is currently on the c'th term (x_c), it means that we have already computed WHT on the previous sub-sequence, namely $\{x_c, x_{c-1}, \ldots, x_{c-2^n+1}\}$ and now by moving the window leftward we want to drop the rightmost term x_{c-2^n+1} and include x_{c+1} term instead, in the next calculation of WHT. The straightforward method in which we calculate WHT for each window independently no longer is efficient because each term is involved in our calculations more than once (2^n times for every term). So in this case it is clearly unreasonable to still use the usual algorithms, like FWHT.

Reference [1] introduced a special algorithm for such applications where only the two leftmost and rightmost terms are changed in each iterative execution of WHT. This algorithm is called Running WHT (RWHT). After a few definitions we briefly review RWHT proposed by [1] and bring our more explicit and detailed formulation which we then mathematically prove. Thereafter we show some improvements in the calculation of the update vectors that reduce the time and memory utilization by a constant factor of $1/16$. We propose also an explicit direct formula to find every entry of an arbitrary-size update vector in a constant time by a very easy-to-compute efficient expression. Finally we include some practical time comparisons between these implementations.

2. WHT AND RUNNING WHT (RWHT)

2.1. Our denotations

To apply WHT on a 2^n-size vector X and obtain a 2^n-size transformed vector Y we use $2^n \times 2^n$ Walsh matrix W_n which is defined recursively as follows: $W_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ and for $n > 1$, $W_n = \begin{bmatrix} W_{n-1} & W_{n-1} \\ W_{n-1} & -W_{n-1} \end{bmatrix}$.

In order to transform X we have, $Y = X \cdot W_n$. This formula is the formal definition of WHT which is rarely used in practice directly. Since WHT is performed on vectors with sizes of a power of 2, for simplicity we denote any 2^n-element vector by an index n, e.g. X_n. Also we define $X_{n,c}$, W_n and $Y_{n,c}$ as:
$X_{n,c} = [x_c, x_{c-1}, \ldots, x_{c-2^n+1}]_{1\times 2^n}$,$Y_{n,c} = X_{n,c} \cdot W_n$
Therefore our objective is to calculate $Y_{n,c+1}$, by having $Y_{n,c}$, with the minimum computation cost. FWHT is usually applied when the data is transformed as a whole block whilst RWHT is used on a sliding window over a large data sequence. In section 5, we will briefly mention the running time and complexity of these algorithms. First, we review the RWHT algorithm introduced by [1].

2.2. Running WHT (RWHT)

When our window slides over the input sequence by one sample, we do not need to perform another WHT on the new window. The WHT of the new window can be obtained by adding the updating vector to a reordered version of the previously transformed vector. Using our denotation, it is proposed by Deng that for $n > 1$, we have

Theorem 1.

$$Y_{n,c+1} = \Gamma(Y_{n,c}) + U_{n,c} \qquad \text{(I)}$$

where $\Gamma()$ *(that reorders a vector) will be defined in definition 1.*

Definition 1. Γ is an easy-to-compute substitutive transform on a 1×2^k matrix (vector) for $k > 1$, that is defined recursively. For $k = 2$, we define $\Gamma([a, b, c, d]) = [a, -b, d, -c]$. For $k > 2$ we define recursively $\Gamma(Y_n) = [\Gamma(Y_n^1) \quad \Gamma(Y_n^2)]$ where Y_n^1 and Y_n^2 are first and second halves of Y_n, e.g.
$\Gamma([0, 1, 2, 3, 4, 5, 6, 7]) = [0, -1, 3, -2, 4, -5, 7, -6]$.

Lemma 1. $\Gamma(A) + \Gamma(B) = \Gamma(A + B)$

The proof is trivial according to definition 1.

Definition 2. For $n = 2$, according to [1] it is defined that $U_{2,c} = [\Delta, \Delta, \Delta, \Delta]$ where $\Delta = x_{c+1} - x_{c-3}$. Now for $n > 2$, we can calculate recursively $U_{n,c}$ by the following formula:

$$U_{n,c} = [U_{n-1,c} + U_{n-1,c-2^{n-1}} \qquad U_{n-1,c} - U_{n-1,c-2^{n-1}}] \qquad \text{(II)}$$

Notice that $U_{n,c}$ is a matrix of size 1×2^n but the size of $U_{n-1,c}$ is $1 \times 2^{n-1}$. Although [1] has not presented a proof for (I), but by having $\Gamma()$ transform and U matrix clearly defined, and using our accurate notation, we can mathematically prove formula (I). A proof for $n = 2, 3$ can be found in [1], but here we give a complete mathematical proof for every $n > 1$ using a similar idea. Notice that by using relation (II) it is fairly straightforward to write a recursive program in order to find $U_{n,c}$ for an arbitrary n. Remember that c is a parameter that changes during sliding WHT computations.

Proof of Theorem (1). To prove (I) we use mathematical induction on n. Refer to [1] for the base case of $n = 2$. Supposing that (I) holds for every $n < k$, we prove it for $n = k$. Let's show two halves of $Y_{k,c+1}$ by $Y_{k,c+1}^1$ and $Y_{k,c+1}^2$, namely: $Y_{k,c+1} = [Y_{k,c+1}^1 \quad Y_{k,c+1}^2]$

$$\begin{aligned} Y_{k,c+1} &= X_{k,c+1} \cdot W_k \\ &= [X_{k-1,c+1} \quad X_{k-1,c-2^{k-1}+1}] \cdot \begin{bmatrix} W_{k-1} & W_{k-1} \\ W_{k-1} & -W_{k-1} \end{bmatrix} \end{aligned}$$

Hence,

$$\begin{aligned} Y_{k,c+1}^1 &= X_{k-1,c+1} \cdot W_{k-1} + X_{k-1,c-2^{k-1}+1} \cdot W_{k-1} \\ Y_{k,c+1}^2 &= X_{k-1,c+1} \cdot W_{k-1} - X_{k-1,c-2^{k-1}+1} \cdot W_{k-1} \end{aligned}$$

Using our induction assumption for $n = k - 1$:

$$\begin{aligned} Y_{k,c+1}^1 &= X_{k-1,c+1} \cdot W_{k-1} + X_{k-1,c-2^{k-1}+1} \cdot W_{k-1} \\ &= \Gamma(Y_{k-1,c}) + U_{k-1,c} + \Gamma(Y_{k-1,c-2^{k-1}}) \\ &\quad + U_{k-1,c-2^{k-1}} \\ &= \Gamma(X_{k-1,c} \cdot W_{k-1}) + \Gamma(X_{k-1,c-2^{k-1}} \cdot W_{k-1}) \\ &\quad + U_{k-1,c} + U_{k-1,c-2^{k-1}} \end{aligned}$$

And similarly

$$\begin{aligned} Y_{k,c+1}^2 &= \Gamma(X_{k-1,c} \cdot W_{k-1}) - \Gamma(X_{k-1,c-2^{k-1}} \cdot W_{k-1}) \\ &\quad + U_{k-1,c} - U_{k-1,c-2^{k-1}} \end{aligned}$$

Therefore, according to lemma 1 and formula (II),

$$\begin{aligned} Y_{k,c+1} &= \Gamma([X_{k-1,c}\, X_{k-1,c-2^{k-1}}] \cdot \begin{bmatrix} W_{k-1} & W_{k-1} \\ W_{k-1} & -W_{k-1} \end{bmatrix}) \\ &\quad + U_{k,c} \\ &= \Gamma(X_{k,c} \cdot W_k) + U_{k,c} = \Gamma(Y_{k,c}) + U_{k,c} \end{aligned}$$

□

3. EFFICIENT IMPLEMENTATION OF RWHT

According to (I), we have to find $U_{n,c}$ for any desired n. So our purpose in this paper, is to present a general method to pre-calculate $U_{n,c}$ for any arbitrary value of n by an efficient implementation. In order to use recursive formula (II) more efficiently, it is not difficult to show that the following lemmas hold for $U_{n,c}$. They can all be proved following an induction similar to the proof of theorem 1.

Lemma 2. *Each row of* $U_{n,c}$ *is in the form of:*

$$u_{1,j} = \sum_{i=c+1}^{c-2^n+1} a_i^j \cdot x_i \quad \textit{for} \quad 0 \le j \le 2^n - 1 \qquad \text{(III)}$$

where $a_i^j \in \{-2, -1, 0, 1, 2\}$ *are constants and independent of c. Therefore in each step of the window sliding we can compute WHT for the window starting at c only by multiplying* a_i^j*s by the corresponding consecutive terms as predicted in formula* (I)*.*

Lemma 3. *In every column of* $U_{n,c}$ *we have* $a_{c+k}^j = 0$ *where* $(k \mod 4) \neq 1$*, e.g.* $\forall j : a_c^j = a_{c-1}^j = a_{c-2}^j =$

0. *So to maintain the sum of* (III) *it is sufficient to only compute and maintain* a_i^j*s for* $i = c+1, c-3, c-7, \cdots$. *In this way we save* $3/4$ *of the computations and memory needed in calculating relation* (II).

Lemma 4. *The row coefficients* (a_i^j) *have the same value in the groups of 4-consecutive rows. Namely,* $\forall i, 0 \le k \le 2^{n-2}-1 : a_i^{4k} = a_i^{4k+1} = a_i^{4k+2} = a_i^{4k+3}$. *Therefore it is sufficient to calculate and maintain the coefficients of only* $1/4$ *of the rows. So far, using lemma 3 and lemma 4, we have decreased the computations and memory needs of relation* (II) *down to* $1/16$ *of their previous values.*

4. AN EXPLICIT AND DIRECT FORMULA FOR UPDATING VECTORS

According to the above lemmas, the matrix consisting of a_i^js of $U_{n,c}$, called R_n, can be shown as follows:

$$R_n = \begin{bmatrix} a_0^0 & 0 & 0 & 0 & a_4^0 & \dots & a_{2^n}^0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_0^{2^n-1} & 0 & 0 & 0 & a_4^{2^n-1} & \dots & a_{2^n}^{2^n-1} \end{bmatrix}_{2^n \times (2^n+1)}$$

Note that R_n is expressed in a transposed form for convenience, i.e. the first row of R_n contains a_i^js of the first column of $U_{n,c}$ ($U_{n,c}$ has 2^n columns and 2^n+1 (a_i^j)s in each column). Now by eliminating duplicated rows in R_n and those zero entries mentioned in lemmas 3 and 4 for conciseness, we obtain a $2^{n-2} \times (2^{n-2}+1)$ matrix, called T_{n-2}. In order to reconstruct R_n from T_{n-2} we can use the following relation:

$$R_n[i,j] = \begin{cases} 0 & \text{for } (j \bmod 4) =_{1,2,3} \\ R_n[i-(i \bmod 4), j] & \text{for } (i \bmod 4) =_{1,2,3} \\ T_{n-2}[i/4, j/4] & \text{otherwise} \end{cases}$$

This relation implies that by having a direct formula for $T_n[i,j]$, we will have $R_n[i,j]$ and $U_{n,c}[1,j]$ consequently. Now we propose an explicit direct formula for $T_n[i,j]$ as given below. Notice that although its definition looks complicated it is very efficient and easy-to-compute indeed, in terms of needed primary binary operations. We will mention intuitively how this formula is obtained later on.

For $0 \le i < 2^n, 0 \le j \le 2^n$ we can prove:

$$T_n[i,j] =$$

$$\begin{cases} \text{If } n = 0 : \quad (-1)^j \\ \text{If } j = 0 : \quad 1 \\ \text{If } j = 2^n : \quad S(i) \\ \text{If } (i\&(2^{P(j)+1}-1)) < 2^{P(j)} : \\ \quad (-1)^{W((i\&j)>>(P(j)+1))} \cdot (1 + S(i\&(2^{P(j)+1}-1))) \\ \text{If } (i\&(2^{P(j)+1}-1)) \ge 2^{P(j)} : \\ \quad (-1)^{W((i\&j)>>(P(j)+1))} \cdot (1 - S(i\&(2^{P(j)+1}-1))) \end{cases}$$

(IV)

here $P(x)$, $W(x)$ and $S(x)$ have simple and easy-to-compute definitions.

Definition 3. For $x \ge 0$, $P(x)$ is the maximum power of 2 which divides x, e.g. $P(20) = 2$ and $P(7) = 0$. $W(x)$ is the number of (1)s in the binary representation of x, e.g. $W(0) = 0$ and $W(10) = 2$. Finally $S(x) = (-1)^{W(x)+1}$. By $\&$ and $>>$ we mean respectively 'Binary AND' and 'Binary shift to right' on the operand's unsigned binary representation, e.g. $10\&5 = 15$, $17 >> 1 = 8$.

The origin of (IV) is easier to explain by considering the intuitive diagram in figure 1 which shows how T_{n+1} is constructed from T_n recursively. This diagram is a result of (II) and lemmas 2,3 and 4.

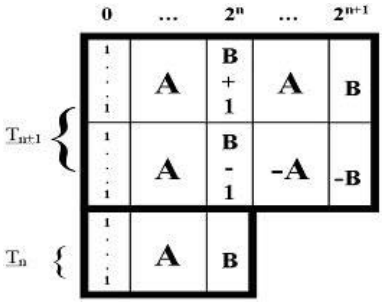

Fig. 1. T_{n+1} is constructed from T_n by the above method where 0'th column always consists of 1. B is the last column of T_n and A is the rest sub-matrix of T_n. 2^n'th column of T_{n+1} has 2 halves, one is $B+1$ and the other equals to $B-1$.

The correctness of (IV) for $n = 0$ or $j = 0$ is trivial by studying the appendix and figure 1 together. The special case of $j = 2^n$ which equals to $S(x) = (-1)^{W(x)+1}$ is correct because of the definition of $W(x)$ while the number of 1s in the binary representation of x indicates the number of times when x'th row falls in the second half of the matrix in the recursive operation and therefore is multiplied by a (-1) factor. In other cases, recursive calculation of $T_n[i,j]$ generally continues until j falls on the middle column where we can have its value in term of $S(x)$ which is the value of the element of the last column in the x'th row. The comparison between $(i\&(2^{P(j)+1}-1))$ and $2^{P(j)}$ determines whether the row falls over or below the middle row (Refer to figure 1) when the column falls on the middle column. This comparison decides which of the last two relations in (IV) must be used.

5. COMPARISON AND CONCLUSION

It has been proved [1] that the RWHT requires at least $3/4 \times 2^n$ less operations than FWHT where the size of the input vector (window) is 2^n. While the utilization of WHT on large data sequences via RWHT is of interest, having an easy-to-implement and efficient algorithm for pre-calculation of update vectors is a necessity. Now we briefly consider the time-complexity of both recursive and direct formulae for calculating $U_{n,c}$.

Having the straightforward recursive relation (II) we can compute $U_{n,c}$ simply by starting from $U_{2,c}$ and con-

tinue up to n. In this case, the total complexity will be:

$$\sum_{k=3}^{k=n}(2^{2k}+2^k) = (4^{n+1}-1)/3 + 2^{n+1} - 29 \text{ for } n > 2.$$

By exploiting the improvements suggested by lemmas 2 , 3 and 4 the overall computation time will decrease down to $(2^{n+1}-8)/16$. Obviously these improvements are more considerable for large window sizes. Using the direct formula (IV), the implementation is easier to optimize. This is because the computations in (IV) can be efficiently implemented only by using binary SHIFTs and ANDs, since the base of exponents are $1,-1$ or 2, they can be seen as some shifts in binary representation. Finally, the time complexity of calculating $T_n[i,j]$ is $O(1)$ for every n, i and j. Therefore, the computation of $U_{n,c}$ which is reduced to computation of T_{n-2} (through R_n) needs $(2^{2n-4}+2^{n-2})$ units of (bit-wise) simple operations. Note that we can find R_n from T_{n-2} on the fly because of their simple relation.

Although asymptotically analyzing the time complexity for both recursive and direct formula results in a $O(4^n)$ complexity, clearly the direct formula is easier to compute and needs less memory and stack size to implement. We have also compared recursive algorithm with the direct formula, in practice, by executing two efficient programs under the same conditions. The practical results for different update vector sizes are provided (in milliseconds) in table 1. While it can be seen that the direct formula again has much less running time in practice, its more important advantage is in not being limited by the vector size. In fact, there's a limitation on the vector size for recursive version because of its memory requirements while in explicit formula there's no need for memory as we can calculate each term independently and store it in external memory.

Vector Size	**Recursive Vector Update**	**Direct Vector Update**
1024	359	1
2048	375	16
4096	438	78
8192	609	312

Table 1. Execution time comparison (in milliseconds) between recursive and direct algorithms for calculating update vectors of RWHT

6. REFERENCES

[1] G. Deng and A. Ling, "A running walsh-hadamard transform algorithm and its application to isotropic quadratic filter implementation," in *Proc. of EUSIPCO*, 1996.

[2] J.R. Johnsony, "Generating parallel programs for fast signal transforms using spiral," in *Workshop on Performance Optimization for High-Level Languages and Libraries*, 2002.

[3] World Wide Web, "Music-dsp source code archive," http://www.musicdsp.org.

[4] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing fourier transform algorithms on various architectures," in *Proc. of Circuits, Systems, and Signal Processing 9, 449–500*, 1990.

[5] J. Johnson and M. Puschel, "In search of the optimal walsh-hadamard transform," in *Proc. of ICASSP*, 2000.

[6] N. Park and V. K. Prasanna, "Cache conscious walsh-hadamard transform," in *Proc. of ICASSP*, 2001.

[7] S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan, "Efficient transposition algorithms for large matrices," in *Proc. of Supercomputing*, 1993.

[8] J. Berger, S.K. Mitra, and J. Astola, "Power spectrum analysis for dna sequences," in *Proc. of 7th International Symposium on Signal Processing and Its Applications*, Jul 2003, pp. 29–32.

A. APPENDIX

In this section we bring the first values of $U_{n,c}$, R_n and T_n to make the context easier to be comprehended. Remember that T_n is constructed from R_{n+2} and vice versa. Also $U_{n,c}$ and R_n which both are 1-row matrices are shown in a 1-column format here for convenience. For $n=2$:

$$U_{2,c} = \begin{bmatrix} x_{c+1}-x_{c-3} \\ x_{c+1}-x_{c-3} \\ x_{c+1}-x_{c-3} \\ x_{c+1}-x_{c-3} \end{bmatrix}, R_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & -1 \end{bmatrix}$$

and so $T_0 = [1 \quad -1]$. For $n=3$:

$$U_{3,c} = \begin{bmatrix} x_{c+1}-x_{c-7} \\ x_{c+1}-x_{c-7} \\ x_{c+1}-x_{c-7} \\ x_{c+1}-x_{c-7} \\ x_{c+1}-2x_{c-3}+x_{c-7} \\ x_{c+1}-2x_{c-3}+x_{c-7} \\ x_{c+1}-2x_{c-3}+x_{c-7} \\ x_{c+1}-2x_{c-3}+x_{c-7} \end{bmatrix}$$

$$R_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 1 \end{bmatrix}$$

and so $T_1 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & -2 & 1 \end{bmatrix}$.