Faster Walsh-Hadamard and Discrete Fourier Transforms from Matrix Non-rigidity

Josh Alman
Columbia University
USA
josh@cs.columbia.edu

Kevin Rao
Columbia University
USA
kevinrao99@gmail.com

ABSTRACT

We give algorithms with lower arithmetic operation counts for both the Walsh-Hadamard Transform (WHT) and the Discrete Fourier Transform (DFT) on inputs of power-of-2 size N.

For the WHT, our new algorithm has an operation count of $\frac{23}{24}N\log N + O(N)$. To our knowledge, this gives the first improvement on the $N\log N$ operation count of the simple, folklore Fast Walsh-Hadamard Transform algorithm.

For the DFT, our new FFT algorithm uses $\frac{15}{4}N\log N + O(N)$ real arithmetic operations. Our leading constant $\frac{15}{4} = 3.75$ improves on the leading constant of 5 from the Cooley-Tukey algorithm from 1965, leading constant 4 from the split-radix algorithm of Yavne from 1968, leading constant $\frac{34}{9} = 3.777\ldots$ from a modification of the split-radix algorithm by Van Buskirk from 2004, and leading constant 3.76875 from a theoretically optimized version of Van Buskirk's algorithm by Sergeev from 2017.

Our new WHT algorithm takes advantage of a recent line of work on the non-rigidity of the WHT: we decompose the WHT matrix as the sum of a low-rank matrix and a sparse matrix, and then analyze the structures of these matrices to achieve a lower operation count. Our new DFT algorithm comes from a novel reduction, showing that parts of the previous best FFT algorithms can be replaced by calls to an algorithm for the WHT. Replacing the folklore WHT algorithm with our new improved algorithm leads to our improved FFT.

CCS CONCEPTS

• **Theory of computation** → **Design and analysis of algorithms**; *Algebraic complexity theory*; • **Mathematics of computing** → *Computation of transforms.*

KEYWORDS

Fourier Transform, Hadamard Transform, Matrix Rigidity

ACM Reference Format:

Josh Alman and Kevin Rao. 2023. Faster Walsh-Hadamard and Discrete Fourier Transforms from Matrix Non-rigidity. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC '23), June 20–23, 2023, Orlando, FL, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3564246.3585188

STOC '23, June 20–23, 2023, Orlando, FL, USA

ACM ISBN 978-1-4503-9913-5/23/06…$15.00
https://doi.org/10.1145/3564246.3585188

1 INTRODUCTION

Two of the most important and widely-used linear transforms are the Discrete Fourier Transform (DFT) and the Walsh-Hadamard Transform (WHT). In addition to their breadth of applications, these transforms can be computed quickly: They can be applied to a vector of length N using only $O(N\log N)$ arithmetic operations, for instance, using the split-radix Fast Fourier Transform (FFT) algorithm [26] and the folklore fast Walsh–Hadamard transform, which make use of the recursive definitions of these transforms.

Determining how much these algorithms can be sped up is an important question in both practice and theory. This is typically phrased as determining the smallest 'leading constant' c such that they can be computing using $(c + o(1))N\log N$ arithmetic operations (where log denotes the base 2 logarithm). In practice, even modest improvements to c can be impactful; the current best algorithm for the DFT, which improves the leading constant from the split-radix algorithm by slightly over 5%, is implemented today in software libraries that have been widely deployed [12][1]. In theory, it is popularly conjectured that an operation count of $\Omega(N\log N)$ is necessary to compute these transforms, i.e., that one cannot achieve arbitrarily small values of c for either of these transforms, but this conjecture is still open. One piece of evidence for this conjecture is that there haven't been many improvements to these constants: prior to this work there have been only three FFT improvements since the original Cooley-Tukey algorithm [8] over 50 years ago, and there has never been an improvement over the folklore algorithm for the WHT.

In this paper, we give improved algorithms, leading to smaller leading constants, for both the WHT and the DFT on inputs of power-of-two size. Our new algorithm for the WHT makes use of a recent line of work on the *matrix rigidity* of the WHT, and our new algorithm for the DFT uses a novel *reduction* to the WHT. Our approach for the DFT is quite different from prior Fast Fourier Transform (FFT) improvements: We focus on decreasing the number of additions and subtractions used by the algorithms, whereas prior improvements focused on the 'twiddle factor' multiplications.

[1] To see it in action, see lines 162 - 239 in `fftw-3.3.10/genfft/fft.ml` of the `fftw-3.3.10` package [12].

1.1 WHT Result

Let N be a power of 2, and let $\mathbb{F}$ be any field. The $N \times N$ WHT matrix, H_N, is defined recursively by

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \text{ and } H_N = \begin{bmatrix} H_{N/2} & H_{N/2} \\ H_{N/2} & -H_{N/2} \end{bmatrix}$$

for $N \geq 4$. The goal of the WHT is, given as input a vector $x \in \mathbb{F}^N$, to compute the vector $y = H_N x$. We focus on $\mathbb{F}$ whose characteristic is not 2, since the problem is trivial in such fields.

The folklore fast Walsh-Hadamard transform algorithm follows directly from this recursive definition: Letting $T_H(N)$ denote the number of arithmetic operations used in computing $H_N x$, we get the base case $T_H(2) = 2$, and the recurrence $T_H(N) = 2T_H(N/2)+N$, which yield $T_H(N) = N \log N$.

In particular, it gives a leading constant of 1, and to our knowledge, no algorithm using fewer arithmetic operations was previously known. Our first main result improves this constant from 1 to $23/24 < 0.96$.

Theorem 1.1. *For any field $\mathbb{F}$, given $x \in \mathbb{F}^N$ for N a power of 2, we can compute $H_N x$ using at most $\frac{23}{24} N \log N + \frac{13}{12} N$ field operations.*

Furthermore, our algorithm only uses fairly simple field operations: $\frac{22}{24} N \log N + \frac{1}{12} N$ additions and subtractions, $\frac{1}{24} N \log N$ "divide by 2" operations, and $N - 1$ "multiply by a power of 2" operations where the power of 2 is at most N. When working over the reals in a typical computer architecture, these division and multiplication operations can be implemented using 'bit shifts' which are substantially faster than a general addition or multiplication operation.

In the full version of the paper, we also give an alternative algorithm which uses $\frac{7}{8} N \log N + O(N)$ additions and subtractions, $\frac{1}{8} N \log N$ "divide by 2" operations, and N "multiply by a power of 2" operations. Although this alternative algorithm does not use fewer total operations than the folklore Fast Walsh-Hadamard Transform, it does use fewer when "divide by 2" operations are ignored, and may be even faster in certain settings.

The main idea behind our new algorithm is to take advantage of the *matrix non-rigidity* of the matrix H_N. A matrix M is called *rigid* if it is impossible to change a 'small' number of its entries to make it have 'low' rank[2]. Rigidity was introduced by Valiant [24] to prove *lower bounds* on the complexity of multiplying matrices like H_N by input vectors. Valiant showed that if M is rigid, then one cannot multiply M by an input vector using $O(N)$ arithmetic operations in an $O(\log N)$-depth arithmetic circuit. (All the algorithms discussed in this paper can be seen as $O(\log N)$-depth arithmetic circuits.) Hence, proving that H_N or a similar matrix is rigid would be a first step toward proving that $\Omega(N \log N)$ operations are required to multiply it by a vector.

However, a recent line of work [4, 5, 7, 10, 11, 15] has proved that many matrices, including the WHT [5] and DFT [11] matrices, are *not* rigid. A formal converse to Valiant's result is not known, and in particular, it is not immediate that this leads to improved algorithms for any of these matrices. Nonetheless, we design our faster algorithm for the WHT by making use of a rigidity upper bound from recent work of one of the authors [4], combined with new observations about the structure of changes one makes to H_N to reduce its rank.

[2]See the surveys [16, 21] for a formal definition and a discussion of many applications of rigidity throughout theoretical computer science.

Table 1: History of the best FFT algorithms, for N a power of 2. For all these algorithms, a leading constant of c means the algorithm uses $cN \log N + O(N)$ real operations, where the O hides a modest constant.

Algorithm	Leading constant of operation count
Cooley-Tukey [8]	5
Split Radix (SR) [26]	4
Modified Split Radix (MSR) [6, 14, 17]	$34/9 = 3.7777\ldots$
"Theoretically Optimized" Split Radix [22]	3.76875
This paper	$15/4 = 3.75$

1.2 DFT Result

The DFT matrix is defined over the complex numbers $\mathbb{C}$. Let $\omega_N := e^{i\pi/N} \in \mathbb{C}$ denote the Nth root of unity, with $i = \sqrt{-1}$. The $N \times N$ DFT matrix, $F_N \in \mathbb{C}^{N \times N}$, is given by, for $a, b \in \{0, 1, \ldots, N-1\}$, $C[a, b] = \omega_N^{a \cdot b}$. The goal of the DFT is, given as input a vector $x \in \mathbb{C}^N$, to compute the vector $y = F_N x$. We focus here on the case where N is a power of 2.

FFT algorithms (for the DFT) have customarily been measured by the number of *real* operations they perform, rather than number of complex operations, to correspond more closely with costs in real computer architectures. In other words, the problem we solve is: We are given as input $2N$ real numbers, corresponding to the real and imaginary parts of the entries of $x \in \mathbb{C}^N$, the goal is to output $2N$ real numbers equal to the real and imaginary parts of the entries of $y = F_N x$, and we count the number of real arithmetic operations performed. For example, we add complex numbers using 2 real additions, and multiply two complex numbers using 6 real operations (4 multiplications and 2 additions).

Table 1 summarizes the history of the best FFT algorithms. The classic Cooley-Tukey algorithm [8] achieves a leading constant of 5, and a few years later in 1968, Yavne [26] introduced the 'split-radix' (SR) variant on Cooley-Tukey which improves the constant to 4. Over thirty years later, in 2004, Van Buskirk publicly posted software [25] which improved on the operation count of SR using a 'Modified Split-Radix' (MSR) approach. By the year 2007, three different groups independently gave theoretical analyses of Van Buskirk's algorithm (or equivalent variants on it), proving its correctness and formally showing that it achieved a leading constant of $34/9 = 3.777\ldots$ [6, 14, 17]. As mentioned above, this MSR algorithm is used in widely-deployed systems today [12]. More recently, in 2017, Sergeev [22] showed that the operation-saving technique of MSR could be applied more effectively in the limit as the 'base circuit' of the algorithm gets larger and larger, improving the constant to 3.76875.

Our second main result improves the constant to $15/4 = 3.75$:

Theorem 1.2. *Given $x \in \mathbb{C}^N$ for N a power of 2, we can compute $F_N x$ using at most $\frac{15}{4} N \log N - \frac{223}{108} N + o(N^{0.8})$ real operations.*

Furthermore, similar to Theorem 1.1, some of the operations in Theorem 1.2 are simple multiplications or divisions by small powers of 2 which could be implemented with fast 'bit shifts': $\frac{1}{36} N \log N$ are "divide by 2" operations, and $N - 1$ are "multiply by a power of 2" operations.

At a high level, the prior improvements beyond Cooley-Tukey focused on the number of real operations used for multiplying inputs by 'twiddle factor' complex numbers throughout the algorithm. They noticed that the operation counts could be reduced by taking advantage of symmetries in the twiddle factors. For example, a key observation of SR is that, given two complex numbers a and b, one can simultaneously compute both $a \cdot b$ and $a \cdot b^*$ using only 8 real operations (where b^* denotes the complex conjugate of b), whereas multiplying a by two arbitrary complex numbers in general is done with 12 real operations.

Our improvement focuses instead on the additions and subtractions used to combine the recursive calls at each layer of these algorithms. We show how to rearrange the order that computations are performed in, so that many of these additions and subtractions can be replaced by a reduction to the WHT. We show:

Lemma 1.1. *Suppose the WHT of a vector $x \in \mathbb{R}^N$ for N a power of 2 can be performed using $cN \log N + O(N)$ real operations. Then, the DFT of a vector $y \in \mathbb{C}^N$ can be performed using $\left(\frac{2}{3}c + \frac{28}{9}\right) N \log N + O(N)$ real operations.*

In particular, setting $c = 1$ (from the folklore fast Walsh-Hadamard transform) in Lemma 1.1 recovers the leading constant 34/9 achieved by MSR, but using our improved $c = 23/24$ from Theorem 1.1 yields the leading constant 15/4 which we state in Theorem 1.2. By the nature of our reduction, our new FFT algorithm is faster (in operation count or in practice) on input size N whenever our algorithm from Theorem 1.1 for the WHT is faster for input size $N/8$. (Indeed, on input size N, our FFT algorithm involves computing many WHTs of size $N/8$ and smaller.)

Although the WHT and DFT matrices differ in critical ways, they have similar recursive structures, and it is natural to wonder whether there is a formal connection showing that an improvement to either one gives an improvement for the other. To our knowledge, our Lemma 1.1 is the first such connection.

1.3 Verification Code

We have implemented the algorithms for Theorem 1.1 and Theorem 1.2, which can be used to verify their correctness and operation counts. Our implementation is available at code.joshalman.com/WHT-and-FFT-from-Non-Rigidity.

1.4 Other Related Work

Barriers for the DFT. All known algorithms for the DFT, including ours, use $\Omega(N \log N)$ arithmetic operations, but it remains an open question to prove that an algorithm using $o(N \log N)$ operations is impossible. A number of previous works have shown barrier results: if one places some restrictions on the structure and properties of FFT algorithms, then $\Omega(N \log N)$ operations are required. For example, Morgenstern [18] proved a $\frac{1}{2} N \log N$ lower bound for the complexity of FFT algorithms with coefficients (the fixed complex numbers that one multiplies by throughout the algorithm) of magnitude at most 1. Works by Papadimitriou [20] and by Haynal and Haynal [13] give $\Omega(N \log N)$ lower bounds by making assumptions about the 'flow graph' of the FFT algorithm, requiring, among other things, that there is a unique path in this graph from any input to any output. Pan [19] similarly gives an $\Omega(N \log N)$ lower bound assuming the flow graph is 'asynchronous'.

The Cooley-Tukey algorithm conforms to all of these assumptions, and SR conforms to all but the asynchronicity assumption. The MSR algorithm does not conform to these assumptions, and our new FFT algorithm introduces new ideas which further violate them. We use real coefficients of magnitude as large as N (the "multiply by powers of 2" operations discussed earlier); these are larger than the coefficients used in the MSR algorithm, which were only slightly super-constant. Nonetheless, we use our coefficients in such a way that on the same inputs, the largest intermediate values one computes using our algorithm, MSR, and SR are roughly the same.[3] Our new WHT algorithm, which we use as a subroutine in our DFT algorithm, makes use of an algebraic identity which computes and combines multiple intermediate values that depend on many of the inputs. This results in a flow graph where each input has many paths to each output.

A more recent line of work by Ailon [1–3] gives an $\Omega(N \log N)$ lower bound for WHT algorithms that don't have $\Omega(n)$ different 'ill-conditioned' intermediate steps. Our new WHT algorithm starts with $\Theta(n)$ steps of multiplying inputs by large powers of 2, which are each much more ill-conditioned than is required by the barriers.

In other words, the new ideas behind our improved algorithm seemingly require further violations of the assumptions that are known to lead to $\Omega(N \log N)$ lower bounds. This suggests that new barriers are needed, but also that studying techniques that overcome these barriers more carefully may help lead to further improvements.

Matrix Rigidity and Linear Circuits. Our faster WHT algorithm uses a rigidity decomposition recently given by [4, Lemma 4.6] for the matrix H_8. The prior work [4] used this and other rigidity decompositions to give a smaller constant-depth circuit for the WHT in a different model of computation, called the 'linear circuit model'. The linear circuit model differs from our setting in how it measures complexity. While we count the number of arithmetic operations used by an algorithm, linear circuits make use of 'linear gates' which may compute arbitrary linear combinations of their inputs, and they only count the total number of input wires to their gates. In particular, in that model, adding two inputs has a cost of 2 (since an addition gate would have 2 input wires), but multiplying by scalars is free. For example, replacing an addition of complex numbers with a multiplication would decrease the cost in the linear circuit model (where multiplications are free), but would not change the number of arithmetic operations that we count, and would even increase the cost when we are counting real operations

[3] Notably, our large coefficients are all powers of 2 which are simple to determine, and the time to compute the other 'twiddle factors' used by our algorithm is nearly identical to MSR.

(since complex multiplications are computed with 6 real operations, but additions use only 2).

Because of the differences in these models, our algorithms do not translate into improved linear circuits, and the algorithms of [4] do not give improved arithmetic operation counts. Notably, in the prior work [4], only the number of entries which are changed in the rigidity decomposition seems to matter, whereas we also need to analyze the pattern of these changes. The prior work [4] ultimately designs its best linear circuit by making use of a different rigidity decomposition for the matrix H_{16} (which strictly improves on the circuit they design using their decomposition of H_8), whereas, despite some effort, we have not been able to design an improved algorithm with a decomposition of H_{16} rather than the decomposition of H_8 that we use.

Although it is known that the DFT is not rigid [11], we do not explicitly use this fact in our FFT algorithm. We only use the non-rigidity of the WHT, and then reduce the DFT to the WHT. Making use of the non-rigidity of the DFT is an exciting, open direction.

2 TECHNICAL OVERVIEW

We begin by giving an overview of our new algorithm for the DFT. We show how to rewrite and rearrange the computations involved in the MSR algorithm so that subcomputations can be extracted which are equivalent to the WHT. We will then give an overview of our new WHT algorithm based on a non-rigidity decomposition of the WHT matrix.

2.1 Improving the Split-Radix Algorithm

The main idea behind our new FFT algorithm is to start with the MSR algorithm and perform a number of steps where its computations are rewritten or rearranged. In this overview, we will instead show how to apply these rewrites and rearrangements to the simpler SR algorithm. Since the MSR algorithm has a very similar overall structure (just with some complicated details inserted), we ultimately apply the same steps we outline here to obtain our final algorithm.

We start with the split-radix FFT algorithm (for the unfamiliar reader, see the full version of the paper for a derivation of this algorithm; for completeness, we rederive both SR and MSR there):

Algorithm 1 Split-Radix FFT

1: **procedure** FFT(x) $\to y_{0,1,\ldots N-1}$ ▷ $x \in \mathbb{C}^N$
2: $A \leftarrow FFT(x[2j]_{j=0}^{N/2-1})$ ▷ $A \in \mathbb{C}^{N/2}$
3: $B \leftarrow FFT(x[4j+1]_{j=0}^{N/4-1})$ ▷ $B \in \mathbb{C}^{N/4}$
4: $C \leftarrow FFT(x[4j-1]_{j=0}^{N/4-1})$ ▷ $C \in \mathbb{C}^{N/4}$
5: **for** $k \in [0, 1, \ldots N/4-1]$ **do**
6: $y_k \leftarrow A_k + \omega_N^k B_k + \omega_N^{-k} C_k$
7: $y_{k+N/4} \leftarrow A_{k+N/4} - i\omega_N^k B_k + i\omega_N^{-k} C_k$
8: $y_{k+N/2} \leftarrow A_k - (\omega_N^k B_k + \omega_N^{-k} C_k)$
9: $y_{k+3N/4} \leftarrow A_{k+N/4} + i\omega_N^k B_k - i\omega_N^{-k} C_k$
10: **end for**
11: **end procedure**

We can equivalently view this algorithm in the following recursive matrix form:

$$F(x) = F_N \Pi_N^{-1} \Pi_N x$$
$$= \begin{bmatrix} I_{N/4} & & D_N & D'_N \\ & I_{N/4} & -iD_N & iD'_N \\ I_{N/4} & & -D_N & -D'_N \\ & I_{N/4} & iD_N & -iD'_N \end{bmatrix}$$
$$\times \begin{bmatrix} F_{N/2} & & \\ & \begin{bmatrix} F_{N/4} & \\ & F_{N/4} \end{bmatrix} & \end{bmatrix} \begin{bmatrix} x[2i]_{i=0}^{N/2-1} \\ x[4i+1]_{i=0}^{N/4-1} \\ x[4i-1]_{i=0}^{N/4-1} \end{bmatrix}$$

Here, Π_N is a permutation matrix which reorders the entries of x so that the subvectors in the algorithm, to which we will recursively apply the FFT, appear contiguously for the sake of clarity.[4] $D_N, D'_N \in \mathbb{C}^{N/4 \times N/4}$ are the diagonal matrices given by $D_{N,N}[j,j] = \omega_N^j$ and $D'_{N,N}[j,j] = \omega_N^{-j}$ for $j \in \{0, \ldots N/4-1\}$.

We now will explain our new idea that modifies this algorithm (as well as the MSR algorithm) to get a speedup. At each step, we write all changes from the previous algorithm in blue.

We begin by focusing on lines 6 - 9 from the split-radix algorithm. Take, for example, line 6.

$$y_k \leftarrow A_k + \omega_N^k B_k + \omega_N^{-k} C_k$$

The important observation here is that since ω_N^k and ω_N^{-k} are complex conjugates, they are identical except for a negated imaginary part. We will take advantage of this to rewrite the line in a convenient way; let α be any complex number and α^* be its complex conjugate. We will use the following algebraic rearrangement of the split-radix algorithm's computations (inspired by similar manipulations used in work on radix-3 FFT algorithms, e.g., [9, 23]).

Lemma 2.1. *Let A, B, C, α be complex numbers written as*

$$A = a + a'i, \quad B = b + b'i, \quad C = c + c'i$$
$$\alpha = r + r'i, \quad \alpha^* = r - r'i$$

for real numbers $a, a', b, b', c, c', r, r'$. Then,

$$A + (\alpha B + \alpha^* C) = [a + (r(b+c) + r'(c'-b'))] + [a' + (r(b'+c') + r'(b-c))]i.$$

Proof. Starting with our substitution,

$$\begin{aligned} A + \alpha B + \alpha^* C =& (a + a'i) + (r + r'i)(b + b'i) + (r - r'i)(c + c'i) \\ =& a + rb - r'b' + rc + r'c' \\ &+ a'i + rb'i + r'bi + rc'i - r'ci \\ =& [a + (r(b+c) + r'(c'-b'))] \\ &+ [a' + (r(b'+c') + r'(b-c))]i. \end{aligned}$$

□

We will analogously rewrite lines 7 through 9 via a similar calculation:

[4] Note that permutation matrices can be applied to an input vector without any arithmetic operations, by simply reordering the entries. In other words, the Π_N matrix will only implicitly be implemented in any actual algorithm by the way the algorithms access their input, and thus does not add any computational complexity.

Corollary 2.1. *Analogously,*

$$\begin{aligned}
A-(\alpha B+\alpha^* C) =&[a-(r(b+c)+r'(c'-b'))]\\
&+[a'-(r(b'+c')+r'(b-c))]i,\\
A-i(\alpha B-\alpha^* C) =&[a+(r'(b+c)+r(b'-c'))]\\
&+[a'+(r'(b'+c')+r(c-b))]i,\\
A+i(\alpha B-\alpha^* C) =&[a-(r'(b+c)+r(b'-c'))]\\
&+[a'-(r'(b'+c')+r(c-b))]i.
\end{aligned}$$

We use these four results and substitution to rewrite the split-radix algorithm as

Algorithm 2 Split-Radix FFT (Rewritten)

1: **procedure** FFT(x) $\to y_{0,1,\dots N-1}$ ▷ $x \in \mathbb{C}^N$
2: $A \leftarrow FFT(x[2j]_{j=0}^{N/2-1})$ ▷ $A \in \mathbb{C}^{N/2}$
3: $B \leftarrow FFT(x[4j+1]_{j=0}^{N/4-1})$ ▷ $B \in \mathbb{C}^{N/4}$
4: $C \leftarrow FFT(x[4j-1]_{j=0}^{N/4-1})$ ▷ $C \in \mathbb{C}^{N/4}$
5: **for** $k \in [0,1,\dots N/4-1]$ **do**
6: $a+a'i, z+z'i, b+b'i, c+c'i \leftarrow A_k, A_{k+N/4}, B_k, C_k$
7: $r+r'i \leftarrow \omega_N^k$
8: $y_k \leftarrow [a+r(b+c)+r'(c'-b')] +[a'+r(b'+c')+r'(b-c)]i$
9: $y_{k+N/4} \leftarrow [z+r'(b+c)+r(b'-c')] +[z'+r'(b'+c')+r(c-b)]i$
10: $y_{k+N/2} \leftarrow [a-r(b+c)-r'(c'-b')] +[a'-r(b'+c')-r'(b-c)]i$
11: $y_{k+3N/4} \leftarrow [z-r'(b+c)-r(b'-c')] +[z'-r'(b'+c')-r(c-b)]i$
12: **end for**
13: **end procedure**

Observe that instead of depending on A, B, C, the output of the *FFT* function (i.e., the quantities calculated on lines 8 through 11) can now be thought of as depending on $a, a', b+c, b'+c', b-c, b'-c'$, which in turn depend only on $A, B+C, B-C$ (where addition and subtraction of vectors is done entry-wise). In other words, the computations in lines 8 through 11 depend on $B+C = FFT(x[4j+1]_{j=0}^{N/4-1}) + FFT(x[4j-1]_{j=0}^{N/4-1})$ and $B-C = FFT(x[4j+1]_{j=0}^{N/4-1}) - FFT(x[4j-1]_{j=0}^{N/4-1})$, which by the linearity of the FFT, are equivalent to $FFT(x[4j+1]_{j=0}^{N/4-1} + x[4j-1]_{j=0}^{N/4-1})$ and $FFT(x[4j+1]_{j=0}^{N/4-1} - x[4j-1]_{j=0}^{N/4-1})$. Thus, we can replace the lines

$$B \leftarrow FFT(x[4j+1]_{j=0}^{N/4-1})$$
$$C \leftarrow FFT(x[4j-1]_{j=0}^{N/4-1})$$

with

$$\tilde{x_B} \leftarrow x[4j+1]_{j=0}^{N/4-1} + x[4j-1]_{j=0}^{N/4-1}$$
$$\tilde{x_C} \leftarrow x[4j+1]_{j=0}^{N/4-1} - x[4j-1]_{j=0}^{N/4-1}$$
$$\tilde{B} \leftarrow FFT(\tilde{x_B})$$
$$\tilde{C} \leftarrow FFT(\tilde{x_C})$$

and substitute $\tilde{b}$ for every instance of $b+c$, $\tilde{b'}$ for every instance of $b'+c'$, $\tilde{c}$ for every instance of $b-c$, and $\tilde{c'}$ for every instance of $b'-c'$, to get

Algorithm 3 Split-Radix FFT (Intermediate modifications)

1: **procedure** FFT(x) $\to y_{0,1,\dots N-1}$ ▷ $x \in \mathbb{C}^N$
2: $\tilde{x_B} \leftarrow x[4j+1]_{j=0}^{N/4-1} + x[4j-1]_{j=0}^{N/4-1}$
3: $\tilde{x_C} \leftarrow x[4j+1]_{j=0}^{N/4-1} - x[4j-1]_{j=0}^{N/4-1}$
4: $A \leftarrow FFT(x[2j]_{j=0}^{N/2-1})$ ▷ $A \in \mathbb{C}^{N/2}$
5: $\tilde{B} \leftarrow FFT(\tilde{x_B})$ ▷ $\tilde{B} \in \mathbb{C}^{N/4}$
6: $\tilde{C} \leftarrow FFT(\tilde{x_C})$ ▷ $\tilde{C} \in \mathbb{C}^{N/4}$
7: **for** $k \in [0,1,\dots N/4-1]$ **do**
8: $a+a'i, z+z'i, \tilde{b}+\tilde{b'}i, \tilde{c}+\tilde{c'}i \leftarrow A_k, A_{k+N/4}, \tilde{B}_k, \tilde{C}_k$
9: $r+r'i \leftarrow \omega_N^k$
10: $y_k \leftarrow [a+(r(\tilde{b})+r'(\tilde{-c'}))] + [a'+(r(\tilde{b'})+r'(\tilde{c}))]i$
11: $y_{k+N/4} \leftarrow [z+(r'(\tilde{b})+r(\tilde{c'}))] + [z'+(r'(\tilde{b'})+r(\tilde{-c}))]i$
12: $y_{k+N/2} \leftarrow [a-(r(\tilde{b})+r'(\tilde{-c'}))] + [a'-(r(\tilde{b'})+r'(\tilde{c}))]i$
13: $y_{k+3N/4} \leftarrow [z-(r'(\tilde{b})+r(\tilde{c'}))] + [z'-(r'(\tilde{b'})+r(\tilde{-c}))]i$
14: **end for**
15: **end procedure**

In this form, each layer of our recursive algorithm first does some additions and subtractions on the input, then makes recursive calls to the FFT function, then finally manipulates the results of those calls. We can now reorder the operations in the algorithm, so that it first does the additions and subtractions in lines 2 and 3 in *all* of the recursive calls before performing lines 10 - 13 in *any* of the recursive calls. Our key insight is that if we combine all of these additions and subtractions together and do them simultaneously, there is a faster way to compute that resulting transformation by making use of our faster algorithm for the WHT.

To explain this idea more precisely, it will help to look again at the matrix form of the algorithm. Namely, $F_N x = (F_N \Pi_N^{-1})(\Pi_N x)$ can be factored as the product

$$\underbrace{\begin{bmatrix} I_{N/4} & & R_F & iR'_F \\ & I_{N/4} & R'_F & -iR_F \\ I_{N/4} & & -R_F & -iR'_F \\ & I_{N/4} & -R'_F & iR_F \end{bmatrix}}_{TW_N} \begin{bmatrix} F_{N/2} & \\ & \begin{bmatrix} F_{N/4} & \\ & F_{N/4} \end{bmatrix} \end{bmatrix}$$
$$\times \underbrace{\begin{bmatrix} I_{N/2} & \\ & \begin{bmatrix} I_{N/4} & I_{N/4} \\ I_{N/4} & -I_{N/4} \end{bmatrix} \end{bmatrix}}_{HL_N} \begin{bmatrix} x[2i]_{i=0}^{N/2-1} \\ x[4i+1]_{i=0}^{N/4-1} \\ x[4i-1]_{i=0}^{N/4-1} \end{bmatrix}$$

where R_F and R'_F are diagonal matrices of reals with $R_F[j,j] + iR'_F[j,j] = \omega_N^j$. [5]

[5] Notice that the middle two matrices in this factorization (the matrix with recursive calls and HL) commute. This is exactly the observation made earlier about the linearity of the FFT.

Now we can see that in each level of our recursion we get a "twiddle matrix" TW_N to the left of our recursive calls and a "WHT-looking matrix" HL_N to the right of our recursive calls. Hence, when computing Algorithm 3, we effectively multiply x on the left by a number of HL matrices (corresponding to all the additions and subtractions of lines 2 and 3 in all the recursive calls) followed by a number of of TW matrices (corresponding to all the manipulations of lines 10 - 13 in all the recursive calls).

Thusfar, we have only rearranged computations of the split-radix algorithm, and one can verify that our current algorithm still has an identical operation count as the normal split-radix. Our improvement now comes from a new approach for simultaneously multiplying the input by all of the HL matrices. Let $H'_N \in \{-1,0,1\}^{N\times N}$ be the linear transform corresponding to applying all the HL matrices to the input x of length N. H'_N is thus recursively defined with base cases $H'_1 = \begin{bmatrix}1\end{bmatrix}$ and $H'_2 = \begin{bmatrix}1 & \\ & 1\end{bmatrix}$ and the recursion

$$H'_N = \begin{bmatrix} H'_{N/2} & \\ & \begin{bmatrix} H'_{N/4} & H'_{N/4} \\ H'_{N/4} & -H'_{N/4}\end{bmatrix}\end{bmatrix}.$$

In particular, from this recursive definition, we observe that H'_N can be written as a permutation of a direct sum of WHT matrices (see the full version of the paper for a proof), giving our reduction of the DFT to the WHT via this family of H' matrices. We can thus apply our new faster algorithm for the WHT (which we describe next) in order to get an improved operation count for H'_N and thus for the entire DFT[6].

To summarize, after computing the transformation H' on the input, we recursively call the proper twiddle matrices on the proper subsets of the input. The final result is:

Algorithm 4 Final Split-Radix FFT (with the full "Walsh-Hadamard Uprooting" trick)

1: **procedure** TW(x) $\to y_{0,1,\ldots N-1}$ ▷ $x \in \mathbb{C}^N$
2: $A \leftarrow TW(x[2j]_{j=0}^{N/2-1})$ ▷ $A \in \mathbb{C}^{N/2}$
3: $\tilde{B} \leftarrow TW(x[4j+1]_{j=0}^{N/4-1})$ ▷ $B \in \mathbb{C}^{N/4}$
4: $\tilde{C} \leftarrow TW(x[4j-1]_{j=0}^{N/4-1})$ ▷ $C \in \mathbb{C}^{N/4}$
5: **for** $k \in [0,1,\ldots N/4-1]$ **do**
6: $a+a'i, z+z'i, \tilde{b}+\tilde{b}'i, \tilde{c}+\tilde{c}'i \leftarrow A_k, A_{k+N/4}, \tilde{B}_k, \tilde{C}_k$
7: $r+r'i \leftarrow \omega_N^k$
8: $y_k \leftarrow [a+(r(\tilde{b})+r'(-\tilde{c}'))]+[a'+(r(\tilde{b}')+r'(\tilde{c}))]i$
9: $y_{k+N/4} \leftarrow [z+(r'(\tilde{b})+r(\tilde{c}'))]+[z'+(r'(\tilde{b}')+r(-\tilde{c}))]i$
10: $y_{k+N/2} \leftarrow [a-(r(\tilde{b})+r'(-\tilde{c}'))]+[a'-(r(\tilde{b}')+r'(\tilde{c}))]i$
11: $y_{k+3N/4} \leftarrow [z-(r'(\tilde{b})+r(\tilde{c}'))]+[z'-(r'(\tilde{b}')+r(-\tilde{c}))]i$
12: **end for**
13: **end procedure**
14: **procedure** FFT(x) $\to y_{0,1,\ldots N-1}$ ▷ $x \in \mathbb{C}^N$
15: $y \leftarrow TW(H'(x))$
16: **end procedure**

[6] Applying the folklore WHT algorithm instead will simply give the exact same operation count as the SR FFT algorithm.

For our overall running time analysis, we have $T_{FFT}(N) = T_{H'}(N) + T_{TW}(N)$.[7] The straightforward algorithm for H' would yield $T_{H'}(N) = \frac{2}{3}N\log N$, but since H' is a direct sum of WHT matrices and (below) we improve the leading constant for computing the WHT by a factor of $\frac{23}{24}$, we ultimately improve the leading constant for computing H' from $\frac{2}{3}$ to $\frac{2}{3}\cdot\frac{23}{24} = \frac{23}{36}$. In the full version of the paper, we perform this analysis more carefully and show that $T_{H'}(N) < \frac{23}{36}N\log N + \frac{25}{12}N + o(N^{0.8})$. All that remains is to calculate $T_{TW}(N)$.

In order to implement TW using as few operations as possible, we compute each of $r\tilde{b}-r'\tilde{c}', r\tilde{b}'+r'\tilde{c}, r'\tilde{b}+r\tilde{c}', r'\tilde{b}'-r\tilde{c}$ exactly once, and then use each of the results twice. Overall, this and the operations to combine with a, a', z, z' total to 20 real operations on vectors of length $N/4$ per iteration of our loop, or $20(N/4) = 5N$ real operations total for one call of $TW(x)$ for length N input x. This gives us the recurrence $T_{TW}(N) = 5N + T_{TW}(N/2) + 2T_{TW}(N/4) = \frac{10}{3}N\log N + O(N)$. This is the same operation count as SR achieves for the corresponding part of its calculations. Combining with $T_{H'}(N)$ gives $T_{FFT}(N) = \frac{143}{36}N\log N + O(N) \approx 3.972N\log N + O(N)$, an improvement over the $4N\log N$ operation count of split-radix. In the full version of the paper, we apply the same ideas to the MSR algorithm to improve its operation count instead and obtain the lower order terms of the operation count.

2.2 Faster Walsh-Hadamard Transform

The starting point for our new algorithm for the WHT is the following decomposition of the matrix H_8 as the sum of a low-rank matrix and a sparse matrix, which was introduced by [4, Lemma 4.6]:

$$H_8 = \begin{bmatrix} 1&1&1&1&1&1&1&1\\ 1&-1&1&-1&1&-1&1&-1\\ 1&1&-1&-1&1&1&-1&-1\\ 1&-1&-1&1&1&-1&-1&1\\ 1&1&1&1&-1&-1&-1&-1\\ 1&-1&1&-1&-1&1&-1&1\\ 1&1&-1&-1&-1&-1&1&1\\ 1&-1&-1&1&-1&1&1&-1 \end{bmatrix} =$$

$$\underbrace{\begin{bmatrix} 1&1&1&1&1&1&1&1\\ 1&-1&-1&-1&-1&-1&-1&-1\\ 1&-1&-1&-1&-1&-1&-1&-1\\ 1&-1&-1&-1&-1&-1&-1&-1\\ 1&-1&-1&-1&-1&-1&-1&-1\\ 1&-1&-1&-1&-1&-1&-1&-1\\ 1&-1&-1&-1&-1&-1&-1&-1\\ 1&-1&-1&-1&-1&-1&-1&-1 \end{bmatrix}}_{\text{low rank}} + \underbrace{\begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&2&0&2&0&2&0\\ 0&2&0&0&2&2&0&0\\ 0&0&0&2&2&0&0&2\\ 0&2&2&2&0&0&0&0\\ 0&0&2&0&0&2&0&2\\ 0&2&0&0&0&0&2&2\\ 0&0&0&2&0&2&2&0 \end{bmatrix}}_{\text{sparse}}$$

Intuitively, we will use this decomposition because low rank and sparse matrices can be multiplied by a vector using few operations. Suppose we wanted to multiply H_8 times a length 8 input $[a,b,c,d,e,f,g,h]$. For the low rank matrix we compute $tot = (b+c+d+e+f+g+h)$ one time and then simply compute $a+tot$ (the desired first output entry) and $a-tot$ (the desired output for

[7] Here we use the notation that $T_A(N)$ is the operation count for applying algorithm A to an input vector of length N.

Algorithm 5 Fast WHT from Non-rigidity of H_8

procedure $H_8(x, k) \to y$ ▷ $k \in \mathbb{N}$ ▷ This algorithm returns $2^k H(x)$
  if $N \leq 4$ **then** Scale the inputs by 2^k, use the folklore $N \cdot \log N$ operation WHT, and **end procedure**
  end if
  $a \leftarrow H(x[j]_{j=0}^{N/8-1},\ k)$ ▷ a, b, c, d, e, f, g, h are length $N/8$ vectors.
  $b \leftarrow H(x[j]_{j=N/8}^{N/4-1},\ k+1)$
  $c \leftarrow H(x[j]_{j=N/4}^{3N/8-1},\ k+1)$
  $d \leftarrow H(x[j]_{j=3N/8}^{N/2-1},\ k+1)$
  $e \leftarrow H(x[j]_{j=N/2}^{5N/8-1},\ k+1)$
  $f \leftarrow H(x[j]_{j=5N/8}^{3N/4-1},\ k+1)$
  $g \leftarrow H(x[j]_{j=3N/4}^{7N/8-1},\ k+1)$
  $h \leftarrow H(x[j]_{j=7N/8}^{N-1},\ k+1)$
  $B_1 \leftarrow b + c$ ▷ Addition done entry-wise
  $B_2 \leftarrow d + h$
  $B_3 \leftarrow f + g$
  $tot \leftarrow B_1 + B_2 + B_3 + e$ ▷ So $tot = b + c + d + e + f + g + h$
  $tot \leftarrow tot/2$ ▷ Scalar division, done over all entries
  $diff \leftarrow a - tot$
  $D \leftarrow diff + d$
  $E \leftarrow diff + e$
  $H \leftarrow diff + h$
  $y[j]_{j=0}^{N/8-1} \leftarrow a + tot$
  $y[j]_{j=N/8}^{N/4-1} \leftarrow E + c + g$
  $y[j]_{j=N/4}^{3N/8-1} \leftarrow E + b + f$
  $y[j]_{j=3N/8}^{N/2-1} \leftarrow E + B_2$
  $y[j]_{j=N/2}^{5N/8-1} \leftarrow D + B_1$
  $y[j]_{j=5N/8}^{3N/4-1} \leftarrow H + c + f$
  $y[j]_{j=3N/4}^{7N/8-1} \leftarrow H + b + g$
  $y[j]_{j=7N/8}^{N-1} \leftarrow D + B_3$
end procedure

all 7 other entries), for a total of 8 operations. For the sparse matrix, we can perform only 2 additions then double the result for each of the 7 nonzero rows, for a total of 21 operations. Including the 8 additions to add the results of these two matrices together, this would give an operation count of 37 for computing H_8.

This is a larger operation count than we are aiming for; the fast Walsh-Hadamard transform uses only 24 operations. A key insight is that while computing these two matrices separately and adding the results is quite costly, we can reuse computations between the two. This allows us to save on computing each matrix and also on combining their results. For example, in the process of computing tot we start by adding $b + c$, a value which is also used to compute the 5th row of our sparse matrix, so we can do that addition only once across the two matrices. Using observations like this, we get down to an operation count of 29.

This is still worse than the baseline of 24 operations for computing H_8. Our last main observation is that 7 out of these 29 operations are multiplying each of the inputs b through h by 2. To reduce the cost of these multiplications, we take a hint from a key idea behind the MSR algorithm for the DFT and ask: what if those inputs, which are the outputs from recursive calls, were already scaled up by a factor of 2? If instead of $[a, b, c, d, e, f, g, h]$ we received $[a, 2b, 2c, 2d, 2e, 2f, 2g, 2h]$ as input, now we can eliminate seven "multiply by 2" operations and only divide one time on the sum $2b + 2c + 2d + 2e + 2f + 2g + 2h$ to get tot. In total, this would reduce the number of operations by 6, down from 29 to 23.

To achieve this, we observe that $2H(x) = H(2x)$ by the linearity of the WHT, so we can "push down" the issue of multiplying by 2 into the recursive call. When we've reached the base case of our recursion, all of the "multiply by 2" operations that have been pushed down finally accumulate and we multiply one time by a power of 2, thus turning many "multiply by 2" operations into a few "divide by 2" operations and a single "multiply by 2^k for some k" operation. This ultimately gives us Algorithm 5 based on using H_8 as our recursive step.

See the full version of the paper where we explain the intuition and derivation of Algorithm 5 in more detail. There, we calculate that the operation count of this algorithm is $\frac{23}{24} N \log N + \frac{N}{24}(\log N \bmod 3) + N - 1$. The leading constant $\frac{23}{24}$ comes directly from our improvement from 24 to 23 operations for computing H_8.

ACKNOWLEDGEMENTS

We would like to thank Nir Ailon, Chi-Ning Chou, Sandeep Silwal, and anonymous reviewers for helpful comments on an earlier draft and Igor Sergeev for answering our questions about his algorithm in [22]. This research was supported in part by NSF Grant CCF-2238221 and a grant from the Simons Foundation (Grant Number 825870 JA).

REFERENCES

[1] Nir Ailon. 2013. A lower bound for fourier transform computation in a linear model over 2x2 unitary gates using matrix entropy. *arXiv preprint arXiv:1305.4745* (2013).
[2] Nir Ailon. 2014. An n\log n Lower Bound for Fourier Transform Computation in the Well Conditioned Model. *arXiv preprint arXiv:1403.1307* (2014).
[3] Nir Ailon. 2015. Tighter fourier transform lower bounds. In *Automata, Languages, and Programming: 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*. Springer, 14–25.
[4] Josh Alman. 2021. Kronecker products, low-depth circuits, and matrix rigidity. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. 772–785.
[5] Josh Alman and Ryan Williams. 2017. Probabilistic rank and matrix rigidity. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. 641–652.
[6] Daniel J Bernstein. 2007. The tangent FFT. In *International Symposium on Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes*. Springer, 291–300.
[7] Vishwas Bhargava, Sumanta Ghosh, Mrinal Kumar, and Chandra Kanta Mohapatra. 2022. Fast, algebraic multivariate multipoint evaluation in small characteristic and applications. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*. 403–415.
[8] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.
[9] Eric Dubois and A Venetsanopoulos. 1978. A new algorithm for the radix-3 FFT. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26, 3 (1978), 222–225.
[10] Zeev Dvir and Benjamin L Edelman. 2019. Matrix Rigidity and the Croot-Lev-Pach Lemma. *Theory Of Computing* 15, 8 (2019), 1–7.

[11] Zeev Dvir and Allen Liu. 2020. Fourier and Circulant Matrices are Not Rigid. *Theory Of Computing* 16, 20 (2020), 1–48.
[12] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, Vol. 3. IEEE, 1381–1384.
[13] Steve Haynal and Heidi Haynal. 2011. Generating and searching families of FFT algorithms. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 4 (2011), 145–187.
[14] Steven G Johnson and Matteo Frigo. 2006. A modified split-radix FFT with fewer arithmetic operations. *IEEE Transactions on Signal Processing* 55, 1 (2006), 111–119.
[15] Bohdan Kivva. 2021. Improved upper bounds for the rigidity of Kronecker products. *arXiv preprint arXiv:2103.05631* (2021).
[16] Satyanarayana V Lokam et al. 2009. Complexity lower bounds using linear algebra. *Foundations and Trends® in Theoretical Computer Science* 4, 1–2 (2009), 1–155.
[17] T Lundy and James Van Buskirk. 2007. A new matrix approach to real FFTs and convolutions of length 2 k. *Computing* 80, 1 (2007), 23–45.
[18] Jacques Morgenstern. 1973. Note on a lower bound on the linear complexity of the fast Fourier transform. *Journal of the ACM (JACM)* 20, 2 (1973), 305–306.
[19] Victor Ya Pan. 1986. The trade-off between the additive complexity and the asynchronicity of linear and bilinear algorithms. *Information processing letters* 22, 1 (1986), 11–14.
[20] Christos H Papadimitriou. 1979. Optimality of the fast Fourier transform. *Journal of the ACM (JACM)* 26, 1 (1979), 95–102.
[21] C Ramya. 2020. Recent Progress on Matrix Rigidity–A Survey. *arXiv preprint arXiv:2009.09460* (2020).
[22] Igor Sergeevich Sergeev. 2017. On the real complexity of a complex DFT. *Problems of Information Transmission* 53, 3 (2017), 284–293.
[23] Yoiti Suzuki, Toshio Sone, and Kenuti Kido. 1986. A new FFT algorithm of radix 3, 6, and 12. *IEEE transactions on acoustics, speech, and signal processing* 34, 2 (1986), 380–383.
[24] Leslie G Valiant. 1977. Graph-theoretic arguments in low-level complexity. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 162–176.
[25] James Van Buskirk. 2004. comp.dsp. Usenet posts.
[26] R Yavne. 1968. An economical method for calculating the discrete Fourier transform. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. 115–125.

Received 2022-11-07; accepted 2023-02-06