BRUTE-FORCE SEARCH OF FAST CONVOLUTION ALGORITHMS

Steve Haynal

SofterHardware
College Place, WA 99324
steve@softerhardware.com

Heidi Haynal

Dept. of Mathematics & Computer Science
Walla Walla University
College Place, WA 99324
heidi.haynal@wallawalla.edu

ABSTRACT

Recent research presents a technique to enumerate all valid assignments of "twiddle factors" for power-of-two fast Fourier transform (FFT) flow graphs. Brute-force search employing state-of-the-art Boolean satisfiability (SAT) solvers can then be used to find FFT algorithms within this large solution space which have desirable characteristics. Surprisingly, this approach has discovered FFT algorithms requiring fewer operations than the split-radix algorithm even when all twiddle factors are n^{th} roots of unity.

This paper reviews and then extends this prior research to examine fast discrete convolution algorithms when implemented via FFT and inverse FFT (IFFT) algorithms. In particular, we find that the combination of FFT and IFFT algorithms in fast convolution permits greater freedom when selecting valid twiddle factor assignments. We exploit this freedom and use SAT solvers to find new fast convolution algorithms with the lowest operation counts known.

Index Terms— Fast Convolution, Fast Fourier Transform, Operation Count, Arithmetic Complexity, Satisfiability Solver

1. INTRODUCTION

Convolution, an integral that expresses the amount of overlap of one function f as it is shifted over another function g, is a foundational concept in mathematics. Its applications include filtering, artificial reverberation, image processing, probability distribution, computation of prime length FFTs, and polynomial multiplication. Given this wide range of application, it is vital to develop and study fast and efficient algorithms to compute convolution.

When computing convolution, the circular convolution theorem is commonly applied to transform the problem into one of computing the Fourier transform and its inverse,

$$f * g = \mathcal{F}^{-1}\{\mathcal{F}\{g\} \cdot \mathcal{F}\{h\}\}. \tag{1}$$

For the discrete case involving sequences of finite length n, this can be expressed in terms of the fast Fourier transform as,

$$y(n) = \mathbf{IFFT}[\mathbf{FFT}[g(n)]\mathbf{FFT}[h(n)]]. \tag{2}$$

Consequently, the fast Fourier transform and its inverse are central to implementing fast circular convolution algorithms.

In terms of lowest arithmetic complexity as measured by the fewest floating point operations, or FLOPs, the split-radix[1][2][3] FFT held the record for more than 30 years and requires $4n\log_2 n - 6n + 8$ FLOPs for a size-n FFT where n is some power of two, $n = 2^m$. In 2004 Lundy and Van Buskirk[4] demonstrated improvements to the split-radix operation count by using "twiddle factors" or multiplication coefficients that are not n^{th} roots of unity. Frigo and Johnson[5] generalized Van Buskirk's pioneering work in the context of optimizing the conjugate-pair split-radix algorithm[6]. Bernstein[7] then described a version of Johnson's algorithm, which is distinct from Van Buskirk's, in terms of algebraic twisting and named it the tangent FFT. These new FFTs, which we will refer to collectively as tangent FFTs, exhibit a modest ($\sim 5.6\%$) reduction in FLOP count when compared to the split-radix, requiring roughly $\frac{34}{9}n\log_2 n$ operations rather than the previous $4n\log_2 n - 6n + 8$. More recently, SAT-based brute-force search[8] was used to find FFT algorithms with lower FLOP count than the split-radix but higher FLOP count than tangent FFTs due to the constraint that twiddle factors must be n^{th} roots of unity.

Assuming a goal of minimizing FLOP count in a fast convolution algorithm, utilizing one of the new tangent FFTs[4][5][7] will produce the best known result. This paper asks whether this bound can be lowered by employing SAT-based brute-force techniques[8] when considering fast convolution's two FFTs and inverse FFT in concert. More precisely, for n-tuple weights $W_g(n)$,$W_h(n)$,$W_y(n)$, do weighted FFT and inverse FFT algorithms exist such that

$$y(n) = \mathbf{IFFT}[W_y(n)(W_g(n)\mathbf{FFT}[g(n)])(W_h(n)\mathbf{FFT}[h(n)])] \tag{3}$$

is a valid fast convolution and requires fewer FLOPs than the best known. All multiplication is pointwise. An n-tuple weight $W(n)$ consists of individual weights, $(\omega_n^a, \omega_n^b, \ldots, \omega_n^z)$, where ω_n represents the complex n^{th} root of unity $e^{-i\frac{2\pi}{n}}$. And in order to be a valid fast convolution, the pointwise multiplication $W_g(n)W_h(n)W_y(n)$ must equal $(1, 1, \ldots, 1)$.

This paper does find and present fast convolution algorithms with lower operation counts than previously known. FLOP count is minimized as it is a clearly defined objective to communicate this research as well as to establish lower FLOP bounds on some instances of this open problem. Since the SAT-based search employed is exhaustive, further FLOP count improvement can only be achieved by studying algorithms outside those captured by this fairly broad formulation. Furthermore, the techniques presented are general, and can be used to search for fast convolution algorithms meeting objectives other than lowest operation count.

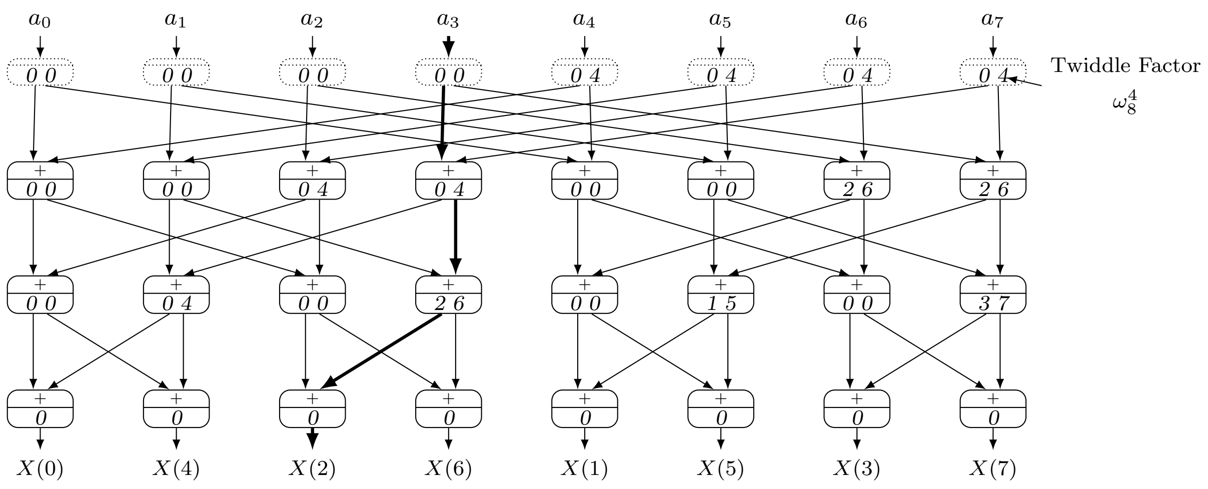

Fig. 1. Size-8 Radix-2 FFT Flow Graph

2. BACKGROUND

The discrete Fourier transform (DFT) is defined as

$$X(k) = \sum_{j=0}^{n-1} a_j \omega_n^{jk}. \tag{4}$$

The DFT requires $O(n^2)$ addition and multiplication floating-point operations on real numbers, or FLOPs, when directly implemented as a computer algorithm. Because of this high arithmetic complexity, the fast Fourier transform (FFT), popularized by Cooley and Tukey[9] in 1965 and which requires only $O(n \log n)$ FLOPs to implement the DFT, is used in practice.

2.1. A Flow Graph Representation of the FFT

Signal-flow graphs are a widely used formalism to represent and analyze FFTs[10][11]. Such flow graphs are central to this paper and are described next.

The flow graph implied by a size-8 FFT, $n = 8$, is shown in Figure 1. The input operands of the FFT, labeled $a_0 ... a_{n-1}$, are shown at the top and the output values, labeled $X(0)...X(n-1)$, are shown at the bottom. Each node represents complex addition and/or multiplication operations applied to its input operands to generate its output values. A node first sums both input operands, when two input operands are available, to create an internal weighted sum. Next, this internal sum is weighted by some ω_n^{tfp}, an n^{th} root of unity. This multiplication constant is commonly referred to as a **twiddle factor**, and only integer powers tfp for left and right twiddle factors, respectively, are shown per node in Figure 1.

The choice of twiddle factor values is the main differentiating factor in terms of FLOP counts among common power-of-two FFTs. Multiplication by ± 1 or $\pm i$ is free, multiplication by $\pm\sqrt{i}$ or $\pm\sqrt{-i}$ requires 4 FLOPs, and multiplication by any other n^{th} root of unity takes 6 FLOPs. FFTs with lower arithmetic complexity, such as the split-radix[1][2][3] algorithm, use an assignment of twiddle factors that reduces the total number of required FLOPs. For example, a node's left and right twiddle factors will be the same or differ by -1, a free multiplication, so that at most only one total complex multiplication is required per node. To further illustrate, the twiddle factors in the final row of nodes are usually free multiplications by $\omega_n^0 = 1$. Such FFT flow graphs, with no addition in the top row of nodes and no multiplication in the bottom row of nodes, clearly require the expected $O(n \log n)$ FLOPs.

A path from an input operand a_j to a result value $X(k)$ must apply the correct weight to a_j to implement a valid DFT. This must be true for all paths. In Figure 1, the only path from a_3 to $X(2)$ is in bold. Along this path, the total weight $\omega_8^6 = \omega_8^0\omega_8^4\omega_8^2\omega_8^0$ is applied to a_3. For $k = 2$, this is the weight required on a_3 by Equation 4.

2.2. Generating and Searching Families of FFTs

In the recent paper[8], the authors developed and formalized this path-based analysis of FFT flow graphs. Each node in a size-n FFT flow graph can be marked with a **weight stride** invariant. This invariant assignment is independent of any particular FFT algorithm implemented by the flow graph and can be used to generate the family of all FFTs realizable by the flow graph where all twiddle factors have modulus one.

In Figure 2, we can trace how input operands a_1, a_3, a_5 and a_7 are combined to form the internal sum for the node labeled G, $a_1\omega_8^0 + a_3\omega_8^2 + a_5\omega_8^4 + a_7\omega_8^6$. The key observation is that the two input operands summed at this node, $a_1\omega_8^0 + a_5\omega_8^4$ from the left and $a_3\omega_8^2 + a_7\omega_8^6$ from the right, must have an integer difference (mod n), or weight stride, of 2 between the powers put on ω_n to form the weights on any two successive a_j in order to create a correct partial sum to support results $X(1)$ and $X(5)$. This is because once combined by a sum, weights on individual a_j cannot be independently changed in these flow graphs. Notice that the weight stride between successive terms a_1, a_3, a_5 and a_7 in $X(1) = a_0\omega_8^0 + a_1\omega_8^1 + a_2\omega_8^2 + a_3\omega_8^3 + a_4\omega_8^4 + a_5\omega_8^5 + a_6\omega_8^6 + a_7\omega_8^7$ is 2, although the actual weights have changed from those seen in the internal sum of node G. Likewise, the weight

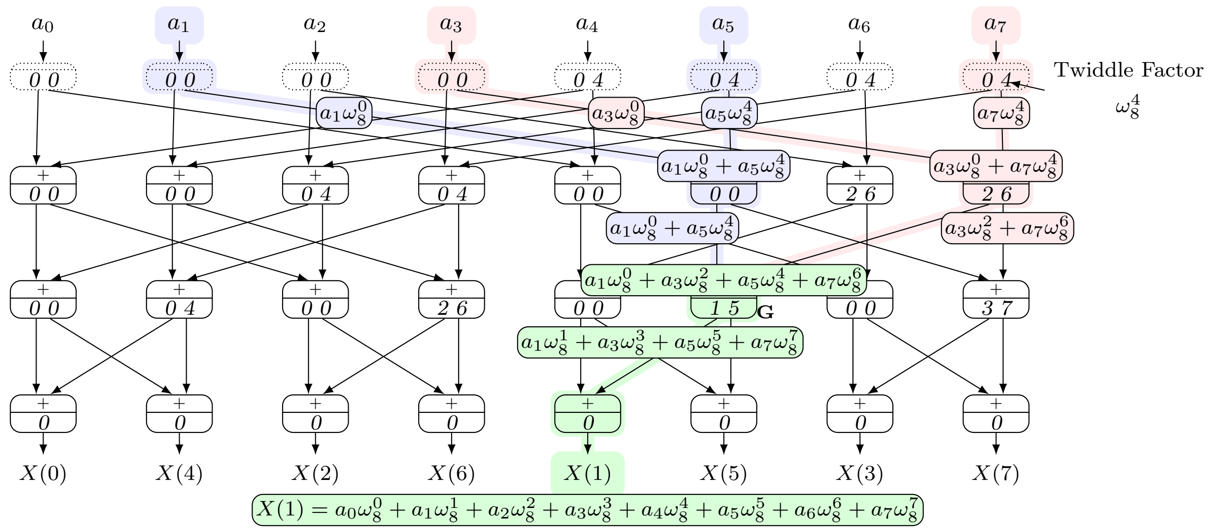
Twiddle Factor
ω_8^4
$a_1\omega_8^0 + a_5\omega_8^4$
$a_3\omega_8^0 + a_7\omega_8^4$
$a_1\omega_8^0 + a_3\omega_8^2 + a_5\omega_8^4 + a_7\omega_8^6$
$a_1\omega_8^1 + a_3\omega_8^3 + a_5\omega_8^5 + a_7\omega_8^7$
G
$X(1) = a_0\omega_8^0 + a_1\omega_8^1 + a_2\omega_8^2 + a_3\omega_8^3 + a_4\omega_8^4 + a_5\omega_8^5 + a_6\omega_8^6 + a_7\omega_8^7$

Fig. 2. Weight Stride in a Size-8 Radix-2 DIF FFT Flow Graph

stride between successive terms a_1, a_3, a_5 and a_7 in $X(5) = a_0\omega_8^0 + a_1\omega_8^5 + a_2\omega_8^2 + a_3\omega_8^7 + a_4\omega_8^4 + a_5\omega_8^1 + a_6\omega_8^6 + a_7\omega_8^3$, which is also created using the internal sum of node G, is 2 (mod 8). Since the weight stride is fixed by a sum, it must be permanently and correctly set at the time of the sum to support correct final result values. Hence, the invariant weight stride for node G is, and can only be, 2.

Once all invariant weight strides are determined through graph analysis, there is a simple linear algorithm to generate a valid set of twiddle factors. Figure 3 shows a size-8 FFT flow graph where each node is marked with its invariant weight stride at its center. First, we randomly assign a weight for the a_j with lowest index j at each node. For the node G in Figure 3, we arbitrarily picked 5, which is notated as an underlined integer in the top left corner of node G. A property of these flow graphs is that the incoming left arc always has the a_j with lowest index j. Now, since the weight stride for node G is 2, the weight of the a_j with lowest index j present on the right incoming arc must be $7 = 5 + 2$. This random assignment is repeated for every node in the flow graph. Next, we examine the difference in weights between parent and child nodes to determine the required twiddle factors. For example, node H is a child of node G, and its expected weight on that path is 3. Therefore, the twiddle factor for the path from H to G must be $6 = 3 - 5(\text{mod}8)$. This twiddle factor appears in the lower right corner of node G. In this example, 5 is used as it is now the weight of the a_j with lowest index j after the addition of the two incoming operands of node G. This computation is repeated for all nodes to determine all twiddle factors. Finally, the same bold path in Figure 1 appears in bold in Figure 3. The total weight of $\omega_8^6 = \omega_8^5\omega_8^1\omega_8^5\omega_8^3$ is the expected final weight on a_3 for $X(2)$, even though the individual weights differ from Figure 1.

With this new way to generate *all* valid twiddle factor assignments for a FFT flow graph, we can compute the number of valid FFTs that exist for a given flow graph. A power-of-two size-n flow graph's solution family has cardinality $2^{n \log_2 n \log_2 n}$, and is the number of valid FFTs realizable by the given flow graph. For example, a size-256 flow graph has 2^{16384} unique twiddle-factor assignments that lead to valid FFTs. You'll agree this is a large number when reminded that the estimated number of atoms in the universe is roughly 2^{264}. Yet this number is very small when compared to all valid and invalid n^{th} root of unity assignments possible for twiddle factors, 2^{34816}. Thus, for a size-256 flow graph, there is just a 1 in 2^{18432} chance of guessing correct twiddle factors.

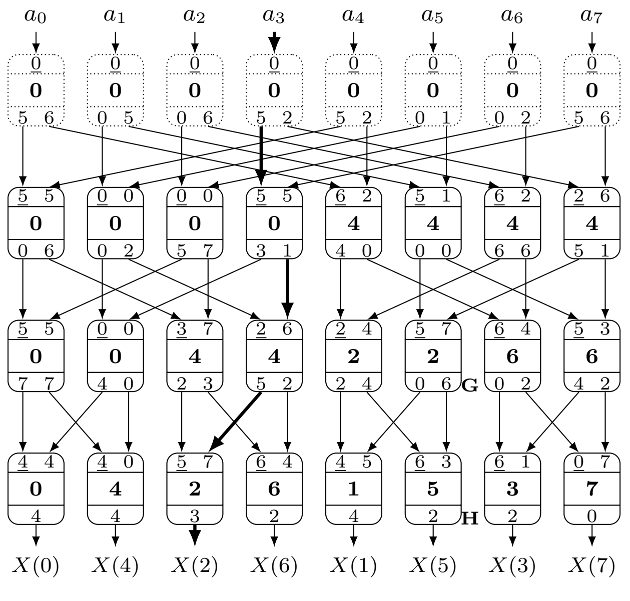

Fig. 3. Random Size-8 Radix-2 DIF FFT Flow Graph

To search for desirable instances, such as those requiring the fewest FLOPs, in this large family of valid FFTs, we cast the search as a Boolean satisfiability (SAT) problem[12][13][14][15]. In particular, to easily accommodate integer arithmetic (mod n), we employed satisfiability modulo theory (SMT) solvers for quantifier-free finite-precision bit-vector arithmetic (QF_BV)[16][17][18][19][20][15]. Although a naïve casting of the problem to SMT is straight-forward, sophisticated partitioning, symmetry reduction and local constraints, all proven to not exclude any fewest FLOPs solutions, are required. These techniques are detailed in the paper[8] and are similar in character to techniques used in the semiconductor industry to formally verify correctness of large logic circuits[21], although properties unique to FFT flow graphs are exploited.

With this SMT-based search, FFTs of size-256 and size-512 that require fewer FLOPs than the best known split-radix, when all twiddle factors are modulus one, are found. Size-256 FFTs that require 6616 FLOPs, compared to 6664 FLOPs for the split-radix, exist. Size-512 FFTs that require 15128 FLOPs, compared to 15368 FLOPs for the split-radix, also exist and are found. Furthermore, by brute-force proof, no lower arithmetic complexity solutions exist for the size-256 FFT under the constraints of the fixed flow graph structure and all twiddle factors of modulus one. Finally, to illustrate the generality of this technique, FFTs of size-64 that require only 3 unique twiddle factors, potentially simplifying the implementation of multiplication in hardware, are found. Witness code examples[22] of all these cases are available on the internet.

3. FAST CONVOLUTION ALGORITHMS

As the first step in searching families of fast convolution algorithms, the techniques summarized in Section 2.2 must be modified to find weighted FFTs (e.g. $W_g(n)\mathbf{FFT}[g(n)]$ as seen in Equation 3) with lower FLOP count than standard FFTs, if they exist. To do this search, two modifications to the FFT search must be made. First, multiplications occurring in the bottom row of nodes in the FFT must incur no cost. In standard FFTs, it is never beneficial to include a multiplication here. In fact, prior work[8] proves this is never beneficial in terms of FLOP count and uses this to partition the search. Intuitively, these final multiplications "undo" any residual *weight on base* that may be present so that the FFT is indeed correct. These multiplications will not be needed in the final fast convolution as $W_g(n)W_h(n)W_y(n)$ will equal $(1, 1, \ldots, 1)$. Finally, since the search partitions must include nodes from the last row, each partition is larger than in prior FFT-only work[8], and hence only FFT flow graphs up to size-128 are feasible to search.

As seen in Table 1, we implemented the two modifications described and found weighted FFTs with lower FLOP counts than standard FFTs. For a size-64 weighted FFT, 3.6 seconds of compute time on a 64-bit Intel Core i7 Linux machine was required to find a size-64 solution and 4.6 seconds to prove that no better solution exists. For a size-128 weighted FFT, 30 minutes of compute time was required to find a 2744 FLOP solution and 28 hours to prove that no solution with fewer FLOPs exists. Compute time exceeded our 36 hour limit when searching larger FFTs. Witness algorithms[22] are posted on the internet.

Table 1. FFT FLOP Counts

FFT	$n = 64$	$n = 128$
Split-Radix	1160	2824
Tangent	1152	2792
Weighted	1136	2744

The weighted inverse FFT expects certain weights to be present on some of its input operands. This is symmetrical to the weighted FFT where weights are present on some output operands. Because of this inherit symmetry, the lowest FLOP counts for the weighted inverse FFT are identical to those in Table 1 and search compute times are similar.

A common use of fast convolution algorithms, such as seen in Bluestein's FFT algorithm and some types of filters, will have constant value input operands for $h(n)$ in Equation 2. Hence, $\mathbf{FFT}[h(n)]$ is precomputed and reused repeatedly to reduce computation. Likewise, $W_h(n)\mathbf{FFT}[h(n)]$ from Equation 3 may also be precomputed. Note that in this case, it does not matter what $W_g(n)$ and $W_y(n)$ are, as values for $W_h(n)$ can always be picked such that $W_g(n)W_h(n)W_y(n)$ will equal $(1, 1, \ldots, 1)$. For this case, we save 32 FLOPs and 96 FLOPS for size-64 and size-128 fast convolution algorithms respectively, when compared to the best known alternative using tangent FFT and IFFT algorithms.

Another constraint we can apply to the formulation is to minimize the elements in $W(n)$ that are some weight ω_n other than 1. This can be done by adding up all such weights in the SAT model and requiring that this sum be less than a specified bound. This bound is iteratively lowered until the model becomes unsatisfiable. When we do this for the size-64 FFT, we find that 16 of the 64 output operands must have a weight to achieve the lowest possible FLOP count of 1136. Interestingly, all 16 weights can be the same, and there are only two possibilities when all the weights are the same. These two possibilities happen to be inverses, $\omega_n^a\omega_n^{-a} = 1$, and consequently a fast convolution algorithm can be created with an unweighted IFFT. In this case, the savings are exactly the same as seen before.

The problem becomes harder if we allow greater freedom in the number and values of weights. If we do not minimize the elements in $W(n)$, we do see a great number of possibilities. We iterated through tens of unique sets of weights before stopping. Section 2.2 discussed the enormous number of valid FFTs, even with the same FLOP counts, and we believe the same is true for $W(n)$. Furthermore, if we relax the constraint to minimize total FLOP count in an FFT or IFFT, it will also allow greater freedom in the possibilities for $W(n)$. What we can say, at least for a size-64 fast convolution algorithm, is that the lowest operation count is between 32 and 48 FLOPs (best possible savings for IFFT and both FFTs) lower than alternatives using tangent FFTs with the lowest known FLOP counts.

4. CONCLUSIONS

This paper extended and employed novel SAT-based brute-force search techniques to find fast convolution algorithms with the lowest known operation counts. Witness algorithms are posted on the internet[22]. More importantly, it established bounds for the lowest possible FLOP counts in fast convolution algorithms given the constraints of the formulation. One must look outside of this defined solution space to further lower the operation count. For example, the underlying graph structure must change, or multiplication coefficients that are not n^{th} roots of unity must be allowed. Future work will explore expanding the solution space, as well as formulating search objectives other than minimizing operation count.

5. REFERENCES

[1] R. Yavne, "An economical method for calculating the discrete Fourier transform," in *Proc. AFIPS Fall Joint Computer Conf.* ACM, 1968, vol. 33, pp. 115–125.

[2] P. Duhamel and M. Vetterli, "Fast Fourier transforms: a tutorial review and a state of the art," *Signal Processing*, vol. 19, no. 4, pp. 259–299, 1990.

[3] H. Sorensen, M. Heideman, and C. Burrus, "On computing the split-radix FFT," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 34, no. 1, pp. 152–156, 2003.

[4] T. Lundy and J. Van Buskirk, "A new matrix approach to real FFTs and convolutions of length 2^k," *Computing*, vol. 80, no. 1, pp. 23–45, 2007.

[5] S.G. Johnson and M. Frigo, "A modified split-radix FFT with fewer arithmetic operations," *Signal Processing, IEEE Transactions on*, vol. 55, no. 1, pp. 111–119, 2007.

[6] I. Kamar and Y. Elcherif, "Conjugate pair fast Fourier transform," *Electronics Letters*, vol. 25, no. 5, pp. 324–325, 1989.

[7] D. Bernstein, "The tangent FFT," *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pp. 291–300, 2007.

[8] S. Haynal and H. Haynal, "Generating and searching families of FFT algorithms," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, pp. 145–187, 2011.

[9] J.W. Cooley and J.W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[10] C.S. Burrus, "Appendix 1: FFT flowgraphs," `http://cnx.rice.edu/content/m16352/latest/`, September 2009.

[11] E.O. Brigham, *The fast Fourier transform and its applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[12] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[13] J.P.M. Silva and K.A. Sakallah, "GRASP a new search algorithm for satisfiability," in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design.* IEEE Computer Society, 1997, pp. 220–227.

[14] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conference, 2001. Proceedings.* IEEE, 2001, pp. 530–535.

[15] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into SAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, no. 3-4, pp. 1–25, 2006.

[16] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T)," *Journal of the ACM (JACM)*, vol. 53, no. 6, pp. 937–977, 2006.

[17] S. Ranise and C. Tinelli, "The SMT-LIB standard: Version 1.2," *Department of Computer Science, The University of Iowa, Tech. Rep*, 2006.

[18] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 174–177, 2009.

[19] B. Dutertre and L. De Moura, "The yices SMT solver," `http://yices.csl.sri.com`, 2010.

[20] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The mathsat 4 smt solver," in *Computer Aided Verification.* Springer, 2008, pp. 299–303.

[21] M. Ganai and A. Gupta, *SAT-based scalable formal verification solutions*, Springer, 2007.

[22] S. Haynal, "Supporting Code Examples," `http://www.softerhardware.com/fft`, 2012.