

Mechanical Derivation of Fused Multiply–Add Algorithms for Linear Transforms

Yevgen Voronenko, *Student Member, IEEE*, and Markus Püschel, *Senior Member, IEEE*

Abstract—Several computer architectures offer fused multiply–add (FMA), also called multiply-and-accumulate (MAC) instructions, that are as fast as a single addition or multiplication. For the efficient implementation of linear transforms, such as the discrete Fourier transform or discrete cosine transforms, this poses a challenge to algorithm developers as standard transform algorithms have to be manipulated into FMA algorithms that make optimal use of FMA instructions. We present a general method to convert any transform algorithm into an FMA algorithm. The method works with both algorithms given as directed acyclic graphs (DAGs) and algorithms given as structured matrix factorizations. We prove bounds on the efficiency of the method. In particular, we show that it removes all single multiplications except at most as many as the transform has outputs. We implemented the DAG-based version of the method and show that we can generate many of the best-known hand-derived FMA algorithms from the literature as well as a few novel FMA algorithms.

Index Terms—Automatic program generation, discrete cosine transform (DCT), discrete Fourier transform (DFT), fast algorithm, implementation, multiply-and-accumulate (MAC) instruction, multiply and accumulate (MAC).

I. INTRODUCTION

SEVERAL modern processor architectures such as Intel Itanium and IBM Power implement fused multiply–add (FMA) instructions, also called multiply-and-accumulate (MAC) instructions. Given three input operands a, b, c , an FMA computes one of the following expressions:

$$\begin{aligned} y &= a + b \times c \\ y &= a - b \times c \\ y &= -a + b \times c \\ y &= -a - b \times c. \end{aligned}$$

The availability of these instructions has important consequences for numerical algorithms. First, in software, an FMA often executes as fast as a single multiplication; in hardware, the cost of an FMA unit is typically lower than the cost of a separate adder and multiplier. This motivates the need for special FMA algorithms that balance additions and multiplications. Second, FMAs can improve the numerical accuracy. For example, [1] shows that the use of FMAs can significantly improve the accuracy of gen-

eral matrix operations. The inclusion of FMA is planned in the revised IEEE 754 floating-point numbers standard.

For some numerical computations, a conversion to FMAs is straightforward. For example, in a generic matrix-vector or matrix–matrix multiplication, additions and multiplications are naturally paired. For other computations, an efficient conversion to FMAs may require nontrivial transformations. Ideally, compilers should perform this task, but they typically will not change the algorithm to improve FMA utilization. However, in many cases this is necessary. An example are linear transforms such as the discrete Fourier transform (DFT), discrete cosine transforms (DCTs), and others. Consequently, their adaptation to FMA architectures has been the subject of research, and several algorithms have been hand-derived. Examples include for the DFT [2]–[7]. Further, [8] presents FMA algorithms for the scaled DCT of type II and size 8 and its inverse in one and two dimensions. In [9], the authors show how to trade additions for multiplications in the Walsh–Hadamard transform (WHT) to make use of FMAs.

Contribution of This Paper: This paper is an extension of our preliminary results in [10]. We show how to systematically adapt transform algorithms to FMA architectures. Our method is 1) *general*, i.e., it can be applied to any linear transform and transform algorithm, real or complex; 2) *mechanical*, i.e., it can, in principle, be performed by a computer; and 3) it enables *analysis* of the quality of the result. In particular, we show that our method fuses all multiplications with additions except at most as many as the transform has outputs.

We present three instantiations of the FMA optimization method. They are mathematically equivalent but operate with different representations of a transform algorithm. Namely, with algorithms given as directed acyclic graphs (DAGs), algorithms given as sparse structure matrix factorizations using the Kronecker product formalism, and algorithms given recursively. There exist both floating-point and integer (or fixed-point) FMAs, and our algorithm can be used for both data types.

We implemented the DAG-based version as backend to the Spiral program generation system [11], [12] and show that we can automatically generate FMA algorithms that match many of the hand-derived FMA algorithms in operations count. Further, we can also generate FMA algorithms for other transforms and algorithms that were not considered before for FMA optimization. Using Spiral, we also show runtime experiments on an Itanium 2, demonstrating a speed-up for all except very small transform sizes.

Organization of the Paper: The rest of this paper is organized as follows. Section II gives the background on linear transforms, fast transform algorithms, and their different representations. Further, we formally state the problem of FMA

Manuscript received August 9, 2006; revised November 13, 2006. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Hakan Johansson. This work was supported by NSF through awards 0234293, 0310941, 0325687, and by DARPA through the Department of Interior Grant NBCH1050009.

The authors are with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: yvoronen@ece.cmu.edu; puschel@ece.cmu.edu).

Digital Object Identifier 10.1109/TSP.2007.896116

optimization addressed in this paper. The next three sections then present the three different flavors of our FMA optimization method: Section III for algorithms given as dataflow graphs, Section IV for algorithms given as sparse matrix factorizations, and Section V for recursive algorithms. The application to complex transforms and algorithms is explained in Section VI. Section VII compares our generated FMA algorithms with hand-derived FMA algorithms from the literature and shows runtime results. Further, we briefly discuss other known FMA optimization methods. Section VIII concludes the paper.

II. BACKGROUND AND PROBLEM STATEMENT

In this section, we provide basic background about transforms, their fast algorithms, and our formula notation for algorithms. We discuss alternative algorithm representations and formally state the problem of FMA optimization of algorithms.

Transforms: In this paper, we consider *linear* signal transforms. A linear signal transform performs a matrix–vector product $y = Mx$, where x , and y are, respectively, the input and output vectors, and M is the transform matrix. Important examples include the DFT, the real DFT, the DCTs of types 1–4, and the inverse modified DCT (IMDCT). They are defined by the following matrices:

$$\begin{aligned} \text{DFT}_n &= [\omega_n^{k\ell}]_{0 \leq k, \ell < n}, \quad \omega_n = e^{-2\pi j/n} \\ \text{RDFT}_n &= [r_{k\ell}]_{0 \leq k, \ell < n}, \quad r_{k\ell} = \begin{cases} \cos \frac{2\pi k\ell}{n}, & k \leq \lfloor \frac{n}{2} \rfloor \\ -\sin \frac{2\pi k\ell}{n}, & k > \lfloor \frac{n}{2} \rfloor \end{cases} \\ \text{DCT-2}_n &= \left[\cos \frac{k(2\ell+1)\pi}{2n} \right]_{0 \leq k, \ell < n} \\ \text{DCT-3}_n &= \text{DCT-2}_n^T \\ \text{DCT-4}_n &= \left[\cos \frac{(2k+1)(2\ell+1)\pi}{4n} \right]_{0 \leq k, \ell < n} \\ \text{IMDCT}_n &= \left[\cos \frac{(2k+1)(2\ell+1+n)\pi}{4n} \right]_{0 \leq k < 2n, 0 \leq \ell < n}. \end{aligned}$$

In each case, the subscript n specifies the length of the input vector x , i.e., the number of columns of the matrix. All above matrices are square, except for the $2n \times n$ matrix IMDCT_n .

Fast Recursive Algorithms as Breakdown Rules: A general matrix vector product for an $n \times n$ matrix requires $\Theta(n^2)$ arithmetic operations [13]. In contrast, for most transforms, there exist fast algorithms that exploit the structure of the transform to reduce the complexity to $O(n \log n)$. Every fast algorithm can be viewed as a factorization of the dense transform matrix into a product of structured sparse matrices (e.g., [11] and [14]). We will call such factorizations *breakdown rules*, following [11]. A breakdown rule typically decomposes a transform into several smaller transforms or converts it into a different transform of usually lesser complexity. The left-hand side of a breakdown rule is a transform, and the right-hand side is the corresponding factorization, expressed as a *formula*. Here are a few examples of breakdown rules; for example, (1) is the Cooley–Tukey fast Fourier transform (FFT)

$$\begin{aligned} \text{DFT}_n &= (\text{DFT}_k \otimes I_m) \text{diag}_{0 \leq i < n} \omega_{n,m}(i) \\ &\quad \cdot (I_k \otimes \text{DFT}_m) L_k^n, \quad n = km \end{aligned} \quad (1)$$

$$\text{DCT-2}_n = L_{n/2}^n (\text{DCT-2}_{n/2} \oplus \text{DCT-4}_{n/2}) \begin{bmatrix} I_{n/2} & J_{n/2} \\ I_{n/2} & -J_{n/2} \end{bmatrix} \quad (2)$$

$$\text{DCT-4}_n = S_n \cdot \text{DCT-2}_n \cdot \text{diag}_{0 \leq i < n} q_n(i) \quad (3)$$

$$\text{IMDCT}_n = \begin{bmatrix} I_n & \\ -J_n & \\ & I_n \end{bmatrix} \cdot \text{DCT-4}_n. \quad (4)$$

Above, I_n is an $n \times n$ identity matrix, J_n is I_n with columns reversed, and L_k^n with $n = km$ is the $n \times n$ stride permutation matrix defined by the underlying permutation $im + j \mapsto jk + i$ for $0 \leq i < k, 0 \leq j < m$. In words, L_k^n transposes an $m \times k$ matrix stored in row-major order. $L_{n/2}^n$ is also called perfect shuffle. Further, $\text{diag}_{0 \leq i < n} f(i)$ is an $n \times n$ diagonal matrix with diagonal entries $f(i)$:

$$\text{diag}_{0 \leq i < n} f(i) = \begin{bmatrix} f(0) & & & \\ & f(1) & & \\ & & \ddots & \\ & & & f(n-1) \end{bmatrix}.$$

In (1) and (3)

$$\omega_{n,m}(i) = \omega_n^{\lfloor \frac{i}{m} \rfloor (i \bmod m)}, \quad q_n(i) = 1 / \left(2 \cos \frac{(2i+1)\pi}{4n} \right).$$

The operators \oplus and \otimes denote the direct sum and tensor product of matrices, respectively, defined as

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}, \quad A \otimes B = [a_{k\ell} \cdot B]_{k,\ell}, \quad A = [a_{k\ell}]_{k,\ell}.$$

Further

$$S_n = \begin{bmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & & \ddots & \\ & & & & 1 & 1 \\ & & & & & 1 \end{bmatrix}.$$

For a complete description of a transform algorithm we need, besides breakdown rules, also base case rules, or *base cases*, as follows:

$$\text{DFT}_2 = F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (5)$$

$$\text{DCT-2}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1/\sqrt{2} \end{bmatrix} F_2. \quad (6)$$

For a given transform, the recursive composition of breakdown rules and the choices of rules at each level yields a large space of alternative formulas. Each formula corresponds to a fast algorithm. These formulas can be automatically generated and converted into code using Spiral [11].

We will call a set of breakdown rules and base cases *sufficient* for a transform T , if it is possible to fully expand T for a set of sizes using these rules. For example, rules (2), (3), and (6) are sufficient for DCT-2 and DCT-4 of 2-power sizes. Rules (2) and (6) are not sufficient for DCT-2, since the rule for DCT-4 is missing.

Additional Formula Notation: Besides the formula constructs described above, we will also use the following additional notation.

When the size of a diagonal matrix (say, $n \times n$) is obvious from the context, we will write

$$\text{diag}_i f(i) = \text{diag}_{0 \leq i < n} f(i).$$

Sometimes, we will also explicitly enumerate the entries of diagonals as in

$$\text{diag}(a, b) = \begin{bmatrix} a & \\ & b \end{bmatrix}.$$

We will occasionally use permutation matrices other than the stride permutation matrix L . Given a permutation p on n points, i.e., a one-to-one mapping of $\{0, \dots, n-1\}$ onto itself, the corresponding permutation matrix is written as

$$\text{perm } p = \left[e_{p(0)}^n \mid \dots \mid e_{p(n-1)}^n \right]$$

where e_i^n is a column basis vector, i.e., a vector of “0”s with a single “1” at the i th position.

The above and some additional notation are summarized in Table I for the convenience of the reader.

Algorithm Representations: A transform algorithm can be represented in several different ways, as illustrated in Fig. 1 for the DCT-3₄ computed using the transpose of (2), as follows:

- 1) as a formula, i.e., as a structured sparse matrix factorization shown in Fig. 1(a);
- 2) as a dataflow graph, i.e., directed acyclic graph (DAG), shown in Fig. 1(b);
- 3) as a computer program as shown in Fig. 1(d).

In the DAG representation, we assume binary nodes only, i.e., every node has at most two operands. A DAG that does not satisfy this property can be converted into a binary DAG of equal operations count.

The FMA variants of the algorithm [as a DAG in Fig. 1(c) and as a program in Fig. 1(e)] were obtained using the method presented in this paper. In Fig. 1(c), the boxes represent FMAs; the bold edge denotes the input being multiplied. In Fig. 1(e), the macros are defined as $\text{fma}(u, v, w) = u + vw$ and $\text{fms}(u, v, w) = u - vw$.

Derivation of FMA Algorithms—Problem Statement: Since we consider *linear* transforms, the only arithmetic operations occurring in their fast algorithms are additions (we consider subtractions as additions in this paper) and multiplications by constants. We define the (arithmetic) *cost* of an algorithm as the total number of arithmetic operations. Multiplications by ± 1 are not counted. FMAs are counted as a single operation. This implies that the cost is minimized when FMAs are used to the maximal extent. The FMA optimization problem is formulated next.

Problem 1 (FMA Optimization): *Given:* a transform algorithm consisting of additions and multiplications. *Find:* a corresponding FMA algorithm consisting of additions, multiplications, and FMAs of minimal cost.

In this paper, we present an algorithm that makes the FMA optimization mechanical, i.e., in principle, it could be done by a computer. The FMA optimization algorithm comes in three

TABLE I
NOTATION SUMMARY

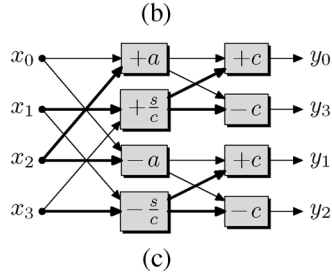
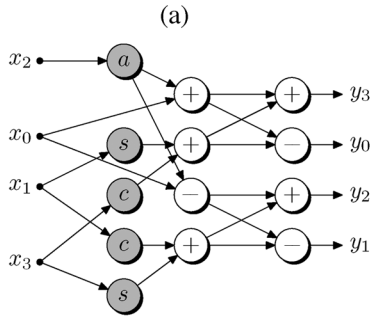
	Notation	Corresponding matrix
General matrix	$[a_{k\ell}]_{1 \leq k, \ell \leq n}$	$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}$
Permutation matrix	$\text{perm } p$	$\left[e_{p(0)}^n \mid \dots \mid e_{p(n-1)}^n \right]$
Conjugation	A^P	$P^{-1}AP$
Diagonal matrix	$\text{diag}_i f$	$\begin{bmatrix} f(0) & & & \\ & f(1) & & \\ & & \ddots & \\ & & & f(n-1) \end{bmatrix}$
	$\text{diag}(a, b)$	$\begin{bmatrix} a & \\ & b \end{bmatrix}$
Butterfly matrix	F_2	$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Stride permutation	L_k^{km}	$\text{perm}_{km} \ i m + j \mapsto j k + i$
Direct Sum	$A \oplus B$	$\begin{bmatrix} A & \\ & B \end{bmatrix}$
Tensor product	$A \otimes B$	$[a_{k\ell} B]_{k, \ell}$
	$I_n \otimes A$	$\begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix}$
	$I_n \otimes_j A_j$	$\bigotimes_{j=1}^n A_j = \begin{bmatrix} A_1 & & \\ & \ddots & \\ & & A_n \end{bmatrix}$
	$A \otimes I_n$	$L_m^{mn} (I_n \otimes A) L_n^{mn}$
	$A_j \otimes_j I_n$	$L_m^{mn} (I_n \otimes_j A_j) L_n^{mn}$

flavors, which differ in the representation of the input transform algorithm.

First, we present a method for FMA optimization using the DAG representation [see Fig. 1(b)]. The method uses local *DAG transformation rules* that fuse additions and multiplication to the extent possible. We implemented this method as a backend to Spiral and were able to generate many of the best known hand-derived FMA algorithms from the literature. The downside of this method is that it requires the explicit construction of the DAG that represents the entire computation. Such a DAG can only be constructed for straightline code, i.e., for code without loops. Therefore, the method is inherently limited to small transforms.

To overcome this limitation, we present next an equivalent FMA optimization method that works with the formula representation of transform algorithms [see Fig. 1(a)]. This time, *formula rewriting rules* are used to create FMAs. Formulas are

$$\text{DCT-3}_4 = (I_2 \oplus J_2)(F_2 \otimes I_2) (F_2 \text{diag}(1, a) \oplus \begin{bmatrix} c & -s \\ s & c \end{bmatrix}) L_2^4$$



DCT3 (y, x)

```
double t1,t2,t3,t4,t5;
t1 = a*x[2];
t2 = x[0] + t1;
t3 = x[0] - t1;
t4 = c*x[1] - s*x[3];
t5 = s*x[1] + c*x[3];
y[0] = t2 + t5;
y[3] = t2 - t5;
y[1] = t3 + t4;
y[2] = t3 - t4;
```

(d)

DCT3 (y, x)

```
double t1,t2,t3,t4;
t1 = fms(x[0], a, x[2]);
t2 = fma(x[0], a, x[2]);
t3 = fms(x[1], s/c, x[3]);
t4 = fma(x[3], s/c, x[1]);
y[0] = fma(t2, c, t4);
y[3] = fms(t2, c, t4);
y[1] = fma(t1, c, t3);
y[2] = fms(t1, c, t3);
```

(e)

Fig. 1. Different representations of an algorithm for DCT-3₄. For brevity, $a = 1/\sqrt{2}$, $c = -\cos(5\pi/8)$, $s = \sin(5\pi/8)$. The macros in (e) are defined as $\text{fma}(u, v, w) = u + vw$ and $\text{fms}(u, v, w) = u - vw$. (a) Sparse matrix product (or formula); (b) standard DAG; (c) FMA DAG; (d) standard code; and (e) FMA code.

more compact than DAGs (for example, tensor products \otimes can represent loops), and therefore large size transforms can be handled efficiently. Further, the structure of the computation (represented by \otimes and \oplus) is preserved. However, this method is inherently limited to transforms of a fixed size.

To overcome the latter restriction, we finally describe an FMA optimization method that operates directly on the breakdown rules [e.g., (1)–(4)]. This is useful when the size of the transform is not known in advance.

As stated previously, we implemented the DAG-based FMA optimization as a backend to Spiral. For formulas and breakdown rules, the FMA optimization can, in principle, again be automated and implemented within Spiral but it is more involved. In many DSP applications, only small transforms are needed.

III. FMA OPTIMIZATION OF DATAFLOW DAGS

In this section, we review the results of our preliminary paper [10] and explain how to automatically adapt transform algorithms to FMA architectures by transforming the algorithm dataflow graph. Since the method works with DAGs, it is inherently nonscalable and can only be used for small transforms. In Sections IV and V, we will generalize this algorithm to handle formulas and breakdown rules and, therefore, transforms of arbitrary size.

A. Algorithm

The main idea behind converting standard DAGs into FMA DAGs is a straightforward mechanic application of local DAG transformation rules at each node in the DAG. The rules in the DAG fuse multiplications with additions to create FMAs and propagate unfused multiplications towards the output nodes. The algorithm is given in Algorithm 1.

Algorithm 1 (FMA Optimization of DAGs):

Input: a DAG \mathcal{D} with add and multiply nodes only; output: an equivalent DAG $\overline{\mathcal{D}}$ with add, multiply, and FMA nodes.

Traverse all nodes in \mathcal{D} , starting at the input nodes and visiting every node exactly once. In each step, choose a *node*, whose predecessors were already visited, and apply one of the transformation rules from Table II, replacing the node by the rule's right-hand side. For each node do the following.

If the node is a multiplication apply:

- rule 1, if a predecessor is an input node or an addition;
- rule 2, if a predecessor is a multiplication.

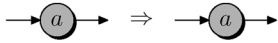



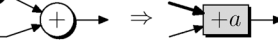




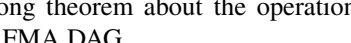
If the node is an addition, apply:

- rule 3, if both predecessors are input nodes or additions;
- rule 4, if exactly one predecessor is a multiplication;
- rule 5, if both predecessors are multiplications.

Terminate when all nodes have been visited. Return the obtained FMA DAG $\overline{\mathcal{D}}$.

Example: The FMA DAG in Fig. 1(c) is the output of Algorithm 1, when applied to the standard DAG in Fig. 1(b). Observe, that the nodes from the first layer of four additions become FMAs and that further multiplications are propagated, which, in turn, convert the second layer of additions into FMAs.

TABLE II
DAG TRANSFORMATION RULES

1.		\Rightarrow		$a \times x \Rightarrow a \times x$
2.		\Rightarrow		$a \times (b \times x) \Rightarrow (a \times b) \times x$
3.		\Rightarrow		$x + y \Rightarrow x + y$
4.		\Rightarrow		$a \times x + y \Rightarrow \text{fma}(y, a, x)$
5.		\Rightarrow		$a \times x + b \times y \Rightarrow a \times \text{fma}(x, \frac{b}{a}, y)$

B. Analysis

Bound on the Number of Unfused Multiplications: Interestingly, we can prove a strong theorem about the operation count or cost of the resulting FMA DAG.

Theorem 1: Assume that the input DAG \mathcal{D} of Algorithm 1 has n output nodes and contains α additions and μ multiplications. Further, assume that the output DAG $\overline{\mathcal{D}}$ of Algorithm 1 contains $\overline{\alpha}$ additions, $\overline{\mu}$ multiplications, and $\overline{\phi}$ FMAs. Then

$$\overline{\alpha} + \overline{\phi} = \alpha \quad (7)$$

$$\overline{\mu} \leq n \quad (8)$$

$$\text{cost}(\overline{\mathcal{D}}) = \overline{\alpha} + \overline{\mu} + \overline{\phi} \leq \alpha + n. \quad (9)$$

Further, if rule 2 in Table II is never used, then

$$\overline{\mu} + \overline{\phi} \geq \mu. \quad (10)$$

Finally, the bounds in (8)–(10) are sharp.

Proof: Inspecting rules 2–5 in Table II shows that each one leaves the number of additions (in the mathematical sense, i.e., also counting the additions in FMAs) unchanged. This yields (7).

Inspecting rules 3–5 shows that each multiplication node with a successor is either converted into an FMA (rule 4), or propagated (rules 2 and 5). Thus, every multiplication in $\overline{\mathcal{D}}$ has to be at the output. Since the number of outputs is n , (8) follows; (9) is the sum of (7) and (8).

Rules 1 and 3–5 do not decrease the number of actual multiplications (i.e., also counting those in FMAs). Thus, (10) holds if rule 2 is never applied.

Consider a DAG for a diagonal matrix, i.e., every input is multiplied by a constant to yield the output. In this case equality holds in (8)–(10). Thus, these bounds are sharp. ■

Note the inequality in (10) in contrast to the equality in (7). This means that the number of multiplications (counting also those in the FMAs) can actually increase in contrast to the number of additions. A simple example where this happens is (a is a constant)

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \text{diag}(1, a) = \begin{bmatrix} 1 & a \\ 1 & -a \end{bmatrix}. \quad (11)$$

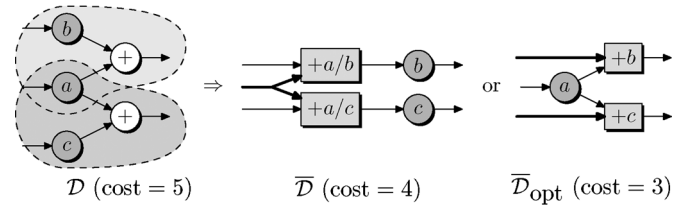


Fig. 2. Example where Algorithm 1 produces a suboptimal solution.

Namely, the DAG for the left formula has $\mu = 1$, and the DAG for the FMA optimized DAG (right formula) has $\overline{\mu} = 0$ and $\overline{\phi} = 2$.

Asymptotic Runtime: Algorithm 1 visits every node of the given DAG exactly once; therefore, the asymptotic runtime of the algorithm is $O(N)$, where N is the number of nodes in the DAG, or equivalently the arithmetic cost of the input transform algorithm.

Optimality: In general, Algorithm 1 does not produce an optimal solution (i.e., with minimum cost) for a given transform algorithm. The simplest case where this happens is shown in Fig. 2. The input DAG \mathcal{D} has three inputs, two outputs, and contains three multiplications and two additions. Algorithm 1 produces an FMA DAG $\overline{\mathcal{D}}$ with four operations: two FMAs and two unfused multiplications. However, a better solution $\overline{\mathcal{D}}_{\text{opt}}$ with only three operations can be obtained, if the shared multiplication by a is not propagated. The result is one multiplication and two FMAs.

Theorem 1 still holds, but a better solution can be obtained if one of the multiplications is not propagated.

Implementation: We implemented Algorithm 1 as a backend to the Spiral program generator [11] and applied it to a number of transforms and sizes. The results and a comparison to published, hand-derived FMA algorithms are in Section VII.

IV. FMA OPTIMIZATION OF FORMULAS

In this section, we show that the FMA optimization performed on a DAG can be performed directly on an equivalent formula. This allows us to derive FMA algorithms for large

transforms, for which the DAG-based method is unpractical. Further, the derivation of FMA formulas maintains the structural information in a formula represented by \cdot , \otimes , and \oplus (see Table I). We will first explain the notation used in this section, then present the algorithm and give a simple FMA optimization example for a DCT-2₄ formula.

A. Problem Statement

From Theorem 1, we know that any DAG corresponding to a linear transform can be transformed into an FMA DAG with all unfused multiplications collected at the outputs. We will capture this at the formula level using the notions of *perfect FMA formula* and *FMA factorization*.

Informally, a perfect FMA formula leaves no unfused multiplications when mapped to code (a rigorous definition is below). FMA factorization is the process of factoring any formula A as $A = DA'$, where D is the diagonal collecting the unfused multiplications at the output, and A' is a perfect FMA formula.

The problem of FMA optimization of formulas is now equivalent to the problem of FMA factorization and can be stated as follows.

Problem 2 (FMA Optimization of Formulas): Given: a formula A . Find: an FMA factorization $A = DA'$ with minimal arithmetic cost, where A' is a perfect FMA formula and D is a diagonal matrix collecting the leftover multiplications.

Now, we define perfect FMA formulas rigorously. The definition is recursive.

Definition 1 (Perfect FMA Formula): Diagonal matrices with entries ± 1 , all permutation matrices, and all explicit matrices with at least one entry ± 1 in each nonzero row are perfect FMA formulas. Further, if A and B are perfect FMA formulas, then so are AB , $A \otimes I$, $I \otimes A$, and $A \oplus B$.

For example, F_2 and $\begin{bmatrix} 1 & a \\ 1 & -a \end{bmatrix}$ are both perfect FMA formulas. Diagonal matrices with entries other than ± 1 are not perfect FMA formulas. If P is a permutation matrix, then

$$\left(I_n \otimes \begin{bmatrix} 1 & a \\ 1 & -a \end{bmatrix} \right) P \tag{12}$$

is a perfect FMA formula, but the equivalent

$$\left(I_n \otimes \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \text{diag}(1, a) \right) P \tag{13}$$

is not, since one of the subformulas, $\text{diag}(1, a)$, is not a perfect FMA formula. The intuition behind the definition is to make the mapping of perfect FMA formulas to code straightforward by constructing these formulas from perfect FMA building blocks. In (13) above, the matrix–vector product of $\text{diag}(1, a)$ with an input vector can only be implemented using a single unfused multiplication, i.e., it is not a perfect FMA building block, while

$$\begin{bmatrix} 1 & a \\ 1 & -a \end{bmatrix}$$

can clearly be implemented with two FMAs.

B. Algorithm

An FMA factorization of a transform algorithm given as a formula is obtained using Algorithm 2.

Algorithm 2 (FMA Optimization of Formulas):

Input: a formula A ; output: a factorization $A = \text{diag}_i f(i)A'$, where A' is a perfect FMA formula.

Apply the rewrite rules (14)–(23) in Table III to A , starting from the right of A , and proceeding towards the left, until no more rules are applicable.

Similarly to Algorithm 1, Algorithm 2 uses a set of local formula rewrite rules. Each rule moves diagonal matrices towards the left in the formula. The rewriting process stops when no more rules are applicable, and when this happens, the formula A will be in the form DA' as desired.

As in Algorithm 1, Algorithm 2 also requires the modification of the constants used in the algorithm. Therefore, even though Algorithm 2 is simple in concept, maintaining the exact closed form of the modified constants can become cumbersome.

Example: To demonstrate Algorithm 2, we compute the FMA factorization of a DCT-2₄ formula. A formula for any 2-power size of DCT-2 or DCT-4 can be obtained by using the breakdown rules (2) and (3) and by terminating with the base case (6). The result for size 4 is

$$\begin{aligned} \text{DCT-2}_4 &= L_2^4 (\text{DCT-2}_2 \oplus S_2 \text{DCT-2}_2 \text{diag}(c, d)) \begin{bmatrix} I_2 & J_2 \\ I_2 & -J_2 \end{bmatrix} \\ &= L_2^4 \left(\begin{bmatrix} 1 & \\ & a \end{bmatrix} F_2 \oplus \begin{bmatrix} 1 & a \\ & a \end{bmatrix} F_2 \begin{bmatrix} c & \\ & d \end{bmatrix} \right) \begin{bmatrix} I_2 & J_2 \\ I_2 & -J_2 \end{bmatrix} \end{aligned} \tag{24}$$

where $a = 1/\sqrt{2}$, $c = q_2(0) = 1/(2 \cos(\pi/8))$, and $d = q_2(1) = 1/(2 \cos(3\pi/8))$.

We apply Algorithm 2 to (24). Since the first (rightmost) factor is already a perfect FMA formula, we start with the second factor—the direct sum. The left summand is already in the desired form. We apply rule (23) twice to the right summand, as follows:

$$\begin{aligned} \begin{bmatrix} 1 & a \\ & a \end{bmatrix} F_2 \begin{bmatrix} c & \\ & d \end{bmatrix} &= \begin{bmatrix} 1 & a \\ & a \end{bmatrix} \begin{bmatrix} c & \\ & c \end{bmatrix} \begin{bmatrix} 1 & d/c \\ 1 & -d/c \end{bmatrix} \\ &= \begin{bmatrix} c & \\ & ca \end{bmatrix} \begin{bmatrix} 1 & a \\ & 1 \end{bmatrix} \begin{bmatrix} 1 & d/c \\ 1 & -d/c \end{bmatrix}. \end{aligned}$$

Thus, we get

$$\begin{aligned} \text{DCT-2}_4 &= L_2^4 \left(\begin{bmatrix} 1 & \\ & a \end{bmatrix} F_2 \oplus \begin{bmatrix} c & \\ & ca \end{bmatrix} \begin{bmatrix} 1 & a \\ & 1 \end{bmatrix} \begin{bmatrix} 1 & d/c \\ 1 & -d/c \end{bmatrix} \right) \\ &\quad \cdot \begin{bmatrix} I_4 & J_4 \\ I_4 & -J_4 \end{bmatrix}. \end{aligned}$$

Now we apply rule (17) to the direct sum and get

$$\begin{aligned} \text{DCT-2}_4 &= L_2^4 \text{diag}(1, a, c, ca) \left(F_2 \oplus \begin{bmatrix} 1 & a \\ & 1 \end{bmatrix} \begin{bmatrix} 1 & d/c \\ 1 & -d/c \end{bmatrix} \right) \\ &\quad \cdot \begin{bmatrix} I_4 & J_4 \\ I_4 & -J_4 \end{bmatrix}. \end{aligned} \tag{25}$$

Applying rule (20) completes the FMA factorization

$$\underbrace{\text{DCT-2}_4}_A = \underbrace{\text{diag}(1, c, a, ca)}_D \cdot \underbrace{L_2^4 \left(F_2 \oplus \begin{bmatrix} 1 & a \\ & 1 \end{bmatrix} \begin{bmatrix} 1 & d/c \\ & -d/c \end{bmatrix} \right) \begin{bmatrix} I_4 & J_4 \\ I_4 & -J_4 \end{bmatrix} }_{A'} \quad (26)$$

Above, we identified the principal components of the obtained FMA factorization of A in (24): the perfect FMA formula A' , and the collected unfused multiplications D .

C. Analysis

Since Algorithm 2 is effectively equivalent to Algorithm 1, all statements made in Section III-B remain valid. We adapt Theorem 1.

Theorem 2: Assume that the input formula A of Algorithm 2 is an algorithm for a transform matrix with n rows (or outputs) and requires α additions and μ multiplications. Further, assume that the output FMA factorization DA' of Algorithm 2 requires $\bar{\alpha}$ additions and $\bar{\phi}$ FMAs for A' (since A' is a perfect FMA formula, it does not use standalone multiplications), and $\bar{\mu}$ multiplications for D (i.e., D has $\bar{\mu}$ diagonal elements $\neq \pm 1$). Then

$$\text{cost}(A') = \bar{\alpha} + \bar{\phi} = \alpha \quad (27)$$

$$\text{cost}(D) = \bar{\mu} \leq n \quad (28)$$

$$\text{cost}(DA') = \bar{\alpha} + \bar{\phi} + \bar{\mu} \leq \alpha + n. \quad (29)$$

Further, the bounds above are sharp.

V. FMA OPTIMIZATION OF BREAKDOWN RULES

The FMA optimization methods Algorithm 1 and Algorithm 2 are only applicable if the size of the transform is known in advance. Ideally, we should also be able to perform FMA optimization for transforms of variable size n . This requires us to work directly with the breakdown rules. Algorithm 2 as is cannot be applied to breakdown rules, because they are recursive. Hence, we extend Algorithm 2 in this section to handle breakdown rules such as (1)–(4). We will state the problem, describe the algorithm, give an example, and analyze the algorithm.

A. Problem Statement

We would like to apply an analogue of Algorithm 2 to the right-hand sides of breakdown rules. These contain transforms, and therefore the FMA optimization procedure is necessarily recursive. For example, the right-hand side of the DCT-2 rule (2) includes a smaller DCT-2 and DCT-4.

The problem can be stated as follows.

Problem 3 (FMA Optimization of Breakdown Rules):

Given: k transforms and a sufficient set (as defined in Section II) of $n \geq k$ breakdown rules (this includes base cases). The rules can be written as

$$T_1 \rightarrow A_1, \dots, T_n \rightarrow A_n$$

where the T_i are not necessarily different transforms (each transform usually requires several rules). *Find:* a sufficient set of FMA breakdown rules and base cases

$$T_1 \rightarrow D_{T_1} A'_1, \dots, T_n \rightarrow D_{T_n} A'_n, \text{ and} \\ T_{n+1} \rightarrow D_{T_{n+1}} A'_{n+1}, \dots, T_{n+m} \rightarrow D_{T_{n+m}} A'_{n+m}.$$

The m auxiliary transforms T_{n+i} , $1 \leq i \leq m$, arise in the process of FMA optimization in our algorithm as explained below. Specifically, the T_{n+i} are scaled variants of the original transforms; they are obtained because of the propagation of unfused multiplications inside the rule. This will become clear in the next section, when we explain the actual algorithm.

B. Algorithm

Basic Idea: Recall that the main idea of Algorithm 2 is to use rewriting to propagate multiplications in the formula from right to left. The formulas in breakdown rules contain again transforms that must be handled recursively. Our solution is to replace each transform with a corresponding auxiliary transform that has a perfect FMA breakdown rule.

Consider an example. Suppose, we want to compute FMA factorizations based on the rules (2) and (3). Starting with rule (2) we proceed as in Algorithm 2. The rightmost factor does not incur multiplications. Next, we have to recursively obtain FMA factorizations of DCT-2 $_{n/2}$ and DCT-4 $_{n/2}$. Assume we have done this already and obtained

$$\text{DCT-2}_{n/2} = D_{n/2}^2 \text{DCT-2}'_{n/2} \text{ and}$$

$$\text{DCT-4}_{n/2} = D_{n/2}^4 \text{DCT-4}'_{n/2}$$

where $D_{n/2}^2$ and $D_{n/2}^4$ are diagonal matrices. Inserting into rule (2) yields

$$\text{DCT-2}_n = L_{n/2}^n \left(D_{n/2}^2 \text{DCT-2}'_{n/2} \oplus D_{n/2}^4 \text{DCT-4}'_{n/2} \right) \cdot \begin{bmatrix} I_{n/2} & J_{n/2} \\ I_{n/2} & -J_{n/2} \end{bmatrix}. \quad (30)$$

Next, we apply rule (17) to obtain

$$L_{n/2}^n \left(D_{n/2}^2 \oplus D_{n/2}^4 \right) \left(\text{DCT-2}'_{n/2} \oplus \text{DCT-4}'_{n/2} \right) \begin{bmatrix} I_{n/2} & J_{n/2} \\ I_{n/2} & -J_{n/2} \end{bmatrix}$$

and then rule (20) to get

$$\underbrace{\left(D_{n/2}^2 \oplus D_{n/2}^4 \right)^{L_{n/2}^n}}_{D_n^2} \cdot \underbrace{L_{n/2}^n \left(\text{DCT-2}'_{n/2} \oplus \text{DCT-4}'_{n/2} \right)}_{\text{DCT-2}'_n} \begin{bmatrix} I_{n/2} & J_{n/2} \\ I_{n/2} & -J_{n/2} \end{bmatrix}. \quad (31)$$

Equation (31) gives a recurrence for D_n^2 and a breakdown rule for DCT-2' $_n$, namely

$$D_n^2 = \left(D_{n/2}^2 \oplus D_{n/2}^4 \right)^{L_{n/2}^n} \quad (32)$$

TABLE III

REWRITE RULES FOR FMA CONVERSION OF FORMULAS. A_m AND B_n DENOTE $m \times m$ AND $n \times n$ FORMULAS (THE RULES ARE ALSO APPLICABLE TO FORMULAS FOR NON-SQUARE MATRICES WITH VERY MINOR MODIFICATIONS). IN THE LAST TWO RULES FOR MATRICES WE USE $z(k)$ TO DENOTE THE COLUMN INDEX OF THE PROPAGATED MULTIPLICATION IN THE k -TH ROW OF THE MATRIX. THERE IS A DEGREE OF FREEDOM IN CHOOSING THIS VALUE, BUT TO MINIMIZE THE TOTAL OPERATIONS COUNT, IT SHOULD BE CHOSEN SO THAT THE PROPAGATED DIAGONAL HAS AS MANY 1S AS POSSIBLE

$(A_m \oplus B_n) \text{diag}_i f(i)$	\rightarrow	$(A_m \text{diag}_i f(n+i) \oplus B_n \text{diag}_i f(i))$	(14)
$(A_m \otimes I_n) \text{diag}_i f(i)$	\rightarrow	$(A_m \text{diag}_i f(j+ni) \otimes_j I_n)$	(15)
$(I_n \otimes A_m) \text{diag}_i f(i)$	\rightarrow	$(I_n \otimes_j A_m \text{diag}_i f(mj+i))$	(16)
$(\text{diag}_i f(i)A_m \oplus \text{diag}_i g(i)B_n)$	\rightarrow	$\text{diag}_i h(i)(A_m \oplus B_n), \quad h(i) = \begin{cases} f(i), & 0 \leq i < m, \\ g(i), & \text{else.} \end{cases}$	(17)
$(\text{diag}_i f(j,i)A_m \otimes_j I_n) \text{diag}_i f(i)$	\rightarrow	$\text{diag}_i f(i \bmod n, \lfloor \frac{i}{n} \rfloor)(A_m \otimes_j I_n)$	(18)
$(I_n \otimes_j \text{diag}_i f(j,i)A_m) \text{diag}_i f(i)$	\rightarrow	$\text{diag}_i f(\lfloor \frac{i}{n} \rfloor, i \bmod n)(I_n \otimes_j A_m)$	(19)
$\text{perm } p \cdot \text{diag}_i f(i)$	\rightarrow	$\text{diag}_i f(p(i)) \cdot \text{perm } p$	(20)
$\text{diag}_i f(i) \cdot \text{diag}_i g(i)$	\rightarrow	$\text{diag}_i f(i)g(i)$	(21)
$[a(k, \ell)]_{k, \ell}$ not perfect	\rightarrow	$\text{diag}_k a(k, z(k)) \left[\frac{a(k, \ell)}{a(k, z(k))} \right]_{k, \ell}, \quad z(k) = \text{any } i \text{ such that } a(k, i) \neq 0$	(22)
$[a(k, \ell)]_{k, \ell} \text{diag}_i f(i)$	\rightarrow	$\text{diag}_k a(k, z(k))f(z(k)) \left[\frac{a(k, \ell)f(\ell)}{a(k, z(k))f(z(k))} \right]_{k, \ell}, \quad z(k) \text{ as above.}$	(23)

$$\text{DCT-2}'_n = L_{n/2}^n \left(\text{DCT-2}'_{n/2} \oplus \text{DCT-4}'_{n/2} \right) \begin{bmatrix} I_{n/2} & J_{n/2} \\ I_{n/2} & -J_{n/2} \end{bmatrix}. \quad (33)$$

The example above illustrates why new auxiliary transforms arise, and it shows that the diagonal matrices collecting the left-over multiplications have to be computed via a recurrence.

Performing the FMA factorization of rule (3) reveals additional necessary auxiliary transforms. Namely, in (3) we encounter $\text{DCT-2}_n Q_n$, where Q_n is a diagonal matrix. Following Algorithm 2, we have to propagate this diagonal through the transform. We will do this by creating a new auxiliary transform $\text{DCT-2}_n(D) = \text{DCT-2}_n D$, where D is an arbitrary diagonal, and then handle the propagation as before. Namely, assume that we already obtained the FMA factorization $\text{DCT-2}_n(D) = D_n^2(D) \text{DCT-2}_n(D)'$, then inserting into rule (3) and propagation yields

$$\begin{aligned} \text{DCT-4}_n &= S_n \text{DCT-2}_n(Q_n) \\ &= S_n D_n^2(Q_n) \text{DCT-2}_n(Q_n)' \\ &= D_n^2(Q_n) S_n' \text{DCT-2}_n(Q_n)'. \end{aligned} \quad (34)$$

Above, $S_n' = (D_n^2(Q_n))^{-1} S_n D_n^2(Q_n)$, and we used the fact that S_n has ones on the main diagonal to propagate the unchanged $D_n^2(Q_n)$ using rule (23). This gives us a recurrence for D_n^4 and also a breakdown rule for $\text{DCT-4}'_n$, as follows:

$$D_n^4 = D_n^2(Q_n) \quad (35)$$

$$\text{DCT-4}'_n = S_n' \text{DCT-2}_n(Q_n)'. \quad (36)$$

Note that the set of breakdown rules obtained above is insufficient, because a breakdown rule for $\text{DCT-2}_n(D)'$ is missing. It can be obtained by multiplying (2) by D from the right and repeating the procedure outlined above.

General Algorithm: We generalize and formalize the idea illustrated in the above example in Algorithm 3. The new algorithm is essentially an extension of Algorithm 2. The rule set of Table III is extended with two additional rewrite rules to handle transforms, and the FMA optimization procedure consists of three distinct steps, in contrast to the single rewriting pass of Algorithm 2.

Algorithm 3 (FMA Optimization of Breakdown Rules):

Input: A sufficient set of n breakdown rules including base cases $T_1 \rightarrow A_1, \dots, T_n \rightarrow A_n$ for the transforms T_1, \dots, T_n (not necessarily different). Output: a sufficient set of FMA breakdown rules and base cases $T_1 \rightarrow D_1 A_1', \dots, T_n \rightarrow D_n A_n'$, and $T_{n+1} \rightarrow D_{n+1} A_{n+1}', \dots, T_{n+m} \rightarrow D_{n+m} A_{n+m}'$. The T_{n+i} are auxiliary transforms, the A_i' are perfect FMA formulas, and the D_i are diagonal matrices.

For each distinct transform T_j , define the following auxiliary transforms: T_j' , $T_j(D)$, and $T_j(D)'$, where D is a diagonal matrix. Extend the rule set of Algorithm 2 with the following rewrite rules:

$$T_j \rightarrow D_{T_j} T_j' \quad (37)$$

$$T_j D \rightarrow T_j(D) \quad (38)$$

$$T_j(D) \rightarrow D_{T_j(D)} T_j(D)'. \quad (39)$$

Each transform T_j is replaced by the corresponding perfect FMA auxiliary transform T_j' and a leftover diagonal, which we denote with D_{T_j} here. A transform followed by a diagonal matrix is replaced by an auxiliary transform $T_j(D)$, which in turn is FMA factorized using the rule (39) as $D_{T_j(D)} T_j(D)'$.

Using the extended rule set, perform the following three steps.

Step 1: FMA optimization of the base cases. FMA factorize the base cases using Algorithm 2.

Step 2: FMA optimization of the given breakdown rules. Apply the rewrite rules in Table III to the right-hand side of every given breakdown rule. Two sets of FMA breakdown rules will be obtained: for the original transforms

$$T_1 \rightarrow D_{T_1} A'_1, \dots, T_n \rightarrow D_{T_n} A'_n$$

and for the corresponding perfect FMA auxiliary transforms obtained implicitly by omitting the left (propagated) diagonal matrix (i.e., as matrices, $T'_i = D_{T_i}^{-1} T_i$)

$$T'_1 \rightarrow A'_1, \dots, T'_n \rightarrow A'_n.$$

Step 3: FMA optimization of auxiliary transform rules. In Step 2, rule (38) introduces auxiliary transforms $T(D)$. Create breakdown rules for the auxiliary transforms by multiplying the rules for T with a diagonal matrix on the right, and apply the rewrite rules to the right-hand side of the new rules to obtain breakdown rules of the form $T(D) \rightarrow DA'$ and, implicitly, $T(D)' = A'$. Repeat this step until no new auxiliary transforms appear.

C. Example

The best way to explain the algorithm is with an example. We continue and complete our example from Section V-B and show the FMA optimization of DCT-2 and DCT-4 using the sufficient set of breakdown rules consisting of (2), (3), and (6).

The FMA conversion will require auxiliary transforms and will also involve propagated diagonals defined by a set of recurrences. Both of these were already briefly discussed earlier in Section V-B. We use the following notation for the necessary auxiliary transforms and FMA factorizations:

$$\begin{aligned} \text{DCT-2}_n &= D_n^2 \text{DCT-2}'_n \\ \text{DCT-2}_n(D) &= D_n^2(D) \text{DCT-2}_{n/2}(D)' \\ \text{DCT-4}_n &= D_n^4 \text{DCT-4}'_n \\ \text{DCT-4}_n(D) &= D_n^4(D) \text{DCT-4}_n(D)'. \end{aligned}$$

In the above, D denotes an arbitrary diagonal matrix.

Now, we follow the three steps in Algorithm 3 to obtain the FMA breakdown rules and base cases for the following transforms: DCT-2, DCT-4, DCT-2', DCT-4', DCT-2(D), DCT-4(D), DCT-2(D)', and DCT-4(D)'. We will also obtain recurrences for the propagated diagonals D_n^2 , $D_n^2(D)$, D_n^4 , and $D_n^4(D)$.

Step 1: FMA optimization of the base case. There is only one base case $\text{DCT-2}_2 = \text{diag}(1, 1/\sqrt{2})F_2$, and it is already FMA factorized. This implies the following:

$$\begin{aligned} \text{DCT-2}'_2 &= F_2 \\ D_2^2 &= \text{diag}(1, 1/\sqrt{2}). \end{aligned}$$

Step 2a: FMA optimization of rule (2). This step was already performed in Section V-B. The results are (31), (32), and (33).

Step 2b: FMA optimization of rule (3). This step was already performed in Section V-B. The results are (34), (35), and (36).

Step 3: FMA optimization of the auxiliary transform rules. The previous step requires an FMA breakdown rule for $\text{DCT-2}_n(Q_n)$. We obtain the base case and the breakdown rule for this auxiliary transform, for the case of an arbitrary diagonal matrix D . We denote the first half of D with D_1 and the second half with D_2 , i.e., $D = D_1 \oplus D_2$; the sizes are omitted for simplicity. Now, the base case is simply

$$\begin{aligned} \text{DCT-2}_2(D) &= \text{diag}(1, 1/\sqrt{2})F_2D \\ &= \text{diag}(D_1, 1/\sqrt{2}D_1) \begin{bmatrix} 1 & D_2D_1^{-1} \\ 1 & -D_2D_1^{-1} \end{bmatrix} \end{aligned}$$

where D_1 and D_2 are scalars in this case. The breakdown rule for $\text{DCT-2}_2(D)$ based on rule (2) becomes

$$\begin{aligned} \text{DCT-2}_n D &= L_{n/2}^n (\text{DCT-2}_{n/2} \oplus \text{DCT-4}_{n/2}) \\ &\quad \cdot \begin{bmatrix} I_{n/2} & J_{n/2} \\ I_{n/2} & -J_{n/2} \end{bmatrix} D \\ &= L_{n/2}^n (\text{DCT-2}_{n/2} \oplus \text{DCT-4}_{n/2}) \\ &\quad \cdot (D_1 \oplus D_1) \cdot \begin{bmatrix} I_{n/2} & J_{n/2}D_2D_1^{-1} \\ I_{n/2} & -J_{n/2}D_2D_1^{-1} \end{bmatrix} \\ &= L_{n/2}^n (\text{DCT-2}_{n/2}D_1 \oplus \text{DCT-4}_{n/2}D_1) \\ &\quad \cdot \begin{bmatrix} I_{n/2} & J_{n/2}D_2D_1^{-1} \\ I_{n/2} & -J_{n/2}D_2D_1^{-1} \end{bmatrix} \\ &= \left(D_{n/2}^2(D_1) \oplus D_{n/2}^4(D_1) \right)^{L_{n/2}^n} \\ &\quad \cdot L_{n/2}^n (\text{DCT-2}_{n/2}(D_1)' \oplus \text{DCT-4}_{n/2}(D_1)') \\ &\quad \cdot \begin{bmatrix} I_{n/2} & J_{n/2}D_2D_1^{-1} \\ I_{n/2} & -J_{n/2}D_2D_1^{-1} \end{bmatrix}. \end{aligned}$$

Note that $D_2D_1^{-1}$ is a diagonal matrix. Further, the above also implies

$$\begin{aligned} D_2^2(D) &= \text{diag}(D_1, aD_1) \\ \text{DCT-2}_2(D)' &= \begin{bmatrix} 1 & D_2D_1^{-1} \\ 1 & -D_2D_1^{-1} \end{bmatrix} \\ D_n^2(D) &= \left(D_{n/2}^2(D_1) \oplus D_{n/2}^4(D_1) \right)^{L_{n/2}^n} \\ \text{DCT-2}_n(D)' &= L_{n/2}^n (\text{DCT-2}_{n/2}(D_1)' \oplus \text{DCT-4}_{n/2}(D_1)') \\ &\quad \cdot \begin{bmatrix} I_{n/2} & J_{n/2}D_2D_1^{-1} \\ I_{n/2} & -J_{n/2}D_2D_1^{-1} \end{bmatrix}. \end{aligned}$$

Finally, we need an FMA breakdown rule for $\text{DCT-4}_n(D_1)$; we derive it for a general diagonal matrix D , as follows:

$$\begin{aligned} \text{DCT-4}_n(D) &= S_n \text{DCT-2}_n Q_n D \\ &= S_n D_n^2(Q_n D) \text{DCT-2}_n(Q_n D)' \\ &= D_n^2(Q_n D) S_n'' \text{DCT-4}_n(Q_n D)' \end{aligned}$$

with $S_n'' = (D_n^2(Q_n D))^{-1} S_n D_n^2(Q_n D)$. From the above, we obtain

$$\begin{aligned} D_n^4(D) &= D_n^2(Q_n D), \text{ and} \\ \text{DCT-4}_n(D)' &= S_n'' \text{DCT-2}_n(Q_n D)'. \end{aligned}$$

TABLE IV
 FMA BREAKDOWN RULES AND DIAGONAL RECURRENCES OBTAINED BY APPLYING ALGORITHM 3 TO (2), (3), AND (6)

FMA breakdown rules	Diagonal recurrences
$\text{DCT-}2_n = D_n^2 \text{DCT-}2'_n$ (40)	$D_2^2 = \text{diag}(1, a)$ (50)
$\text{DCT-}4_n = D_n^4 \text{DCT-}4'_n$ (41)	$D_n^2 = (D_{n/2}^2 \oplus D_{n/2}^4)^{L_{n/2}^n}$ (51)
$\text{DCT-}2'_2 = F_2$ (42)	$D_n^4 = D_n^2(Q_n)$ (52)
$\text{DCT-}2'_n = L_{n/2}^n (\text{DCT-}2'_{n/2} \oplus \text{DCT-}4'_{n/2}) \begin{bmatrix} I_{n/2} & J_{n/2} \\ I_{n/2} & -J_{n/2} \end{bmatrix}$ (43)	$D_n^2(D) = (D_{n/2}^2(D_1) \oplus D_{n/2}^4(D_1))^{L_{n/2}^n}$ (53)
$\text{DCT-}4'_n = S'_n \text{DCT-}2_n(Q_n)'$ (44)	$D_n^4(D) = D_n^2(Q_n D)$ (54)
$S'_n = (D_n^2(Q_n))^{-1} S_n D_n^2(Q_n)$ (45)	$Q_n = \text{diag}_k q_n(k) = \text{diag}_{0 \leq k < n} 1 / (2 \cos \frac{(2k+1)\pi}{4n})$. (55)
$\text{DCT-}2_2(D)' = \begin{bmatrix} 1 & D_2/D_1 \\ 1 & -D_2/D_1 \end{bmatrix}$ (46)	
$\text{DCT-}2_n(D)' = L_{n/2}^n (\text{DCT-}2_{n/2}(D_1)' \oplus \text{DCT-}4_{n/2}(D_1)') \cdot \begin{bmatrix} I_{n/2} & J_{n/2} D_2 D_1^{-1} \\ I_{n/2} & -J_{n/2} D_2 D_1^{-1} \end{bmatrix}$ (47)	
$\text{DCT-}4_n(D)' = S''_n \text{DCT-}2_n(Q_n D)'$ (48)	
$S''_n = (D_n^2(Q_n D))^{-1} S_n D_n^2(Q_n D)$. (49)	

In summary, we obtained the FMA breakdown rules and the recurrences for the necessary diagonal matrices listed in Table IV [equations (40)–(55)].

Pseudocode: The breakdown rules in Table IV can be naturally implemented in the setting of a recursive library. We illustrate this by providing MATLAB pseudocode.

First, Table V shows the code for the standard (non-FMA) implementation of rules (2), (3), and (6). The occurring constants are assumed to be precomputed for efficiency.

Second, Table VI gives pseudocode for the FMA optimized implementation of $\text{DCT-}2_n$ and $\text{DCT-}4_n$ using the FMA rules in Table IV. We show only the code for $\text{DCT-}2_n(D)'$ and $\text{DCT-}4_n(D)'$, since the other breakdown rules result in very similar code.

D. Analysis

Bound on the Number of Unfused Multiplications: Algorithm 3 eventually produces an FMA factorization for each given transform of the form DA' . Thus, Theorem 2 holds. However, note that this requires that all of the constants be precomputed using the recurrences for the occurring diagonal matrices.

Convergence: At first glance it may not be obvious that Algorithm 3 converges, but it does. Namely, every given transform T_i spawns at most the three new transforms T'_i , $T_i(D)$, and $T_i(D)'$. Similarly, each supplied rule for T_i spawns exactly one rule for each of these auxiliary transforms. Since each rule is rewritten by the algorithm exactly once, the algorithm terminates.

Further Improvements: In some cases, it is possible to further reduce the number of leftover multiplications by taking into account the position of trivial (± 1) matrix entries. This is necessary in rewrite rules (22) and (23). To maximally reduce the

 TABLE V
 MATLAB PSEUDO CODE FOR BREAKDOWN RULES (2) AND (3)
 AND BASE CASE (6)

```

y = DCT2(x, n)
if n==2 then
    % base case
    y(1) = x(1) + x(2)
    y(2) = a*(x(1) - x(2))
else
    % t = [[I_n, J_n], [I_n, -J_n]] x
    t(1:n/2) = x(1:n/2) + flipud(x(n/2+1:n))
    t(n/2+1:n) = x(1:n/2) - flipud(x(n/2+1:n))
    % y = L^n_{n/2} (DCT2_{n/2} (+) DCT4_{n/2}) t
    y(1:2:n) = DCT2(t, n/2)
    y(2:2:n) = DCT4(t, n/2)
endif

y = DCT4(x, n)
% t = DCT2_n * Q_n * x
k = (0:n-1)
q = 1 / (2*cos(pi*(2*k-1)/(4*n)))
t = DCT2(q * x)
% y = S_n * t
y(1:n-1) = t(1:n-1) + t(2:n)
y(n) = t(n)
    
```

TABLE VI
MATLAB PSEUDO CODE FOR FMA BREAKDOWN RULES (46), (47), AND (48)

```

y = DCT2_D'(x, n, d)
if n==2 then
    % base case
    y(1) = x(1) + x(2)* (d(2)/d(1))
    y(2) = x(1) - x(2)* (d(2)/d(1))
else
    d1 = d(1:n/2)
    d2 = d(n/2+1:n)
    r = d2./d1 % to be precomputed
    % t = [[I_n, d2/d1 J_n], [I_n, -d2/d1 J_n]] x
    t(1:n/2) = x(1:n/2) + flipud(r*x(n:-1:n/2+1))
    t(n/2+1:n) = x(1:n/2) - flipud(r*x(n:-1:n/2+1))
    % y = L^_n_n/2 (DCT2_n/2 (+) DCT4_n/2) t
    y(1:2:n) = DCT2_D'(t, n/2, d2)
    y(2:2:n) = DCT4_D'(t, n/2, d1)
endif

y = DCT4_D'(x, n, d)
k = (0:n-1)
qd = d / (2*cos(pi*(2*k-1)/(4*n))) % precomputed
t = DCT2_D'(x, n, qd)
% y = S'_n * t
d = D2(qd) % precomputed unfused diagonal D^2_n(.)
r = d(2:n) / d(1:n-1) % precomputed
y(1:n-1) = t(1:n-1) + r*t(2:n)
y(n) = t(n)

```

number of multiplications, one then needs to introduce additional transforms of the form $T(D)$, and $T(D)'$ for each pattern of trivial entry locations in the propagated diagonals. Therefore, the number of auxiliary transforms becomes larger, and the procedure more complicated. In the worst case, the algorithm might not converge, but we have not encountered this case. This technique makes it possible to completely eliminate leftover multiplications in the Cooley–Tukey FFT (1), where the so-called “twiddle factors” $\omega_{n,m}(i)$ have several “1”s in known positions.

Implementation: Algorithm 3, as Algorithm 2, can be implemented using a rewriting system [15] as part of a formula-based program generator like Spiral [11], and the authors plan to do so in the future. To do this in a meaningful way, however, it is necessary to ensure that the implementation is compatible with Spiral’s formula-based loop optimizations [16], optimizations for vector instruction sets [17], and parallelization techniques [18]. Discussing these issues exceeds the scope of this paper.

Other Comments: Note that in some applications (such as JPEG image compression), a transform is directly followed by a scaling step. In these cases, the leftover multiplications can be fused with this step and the auxiliary perfect FMA transforms T' become interesting in their own right.

VI. COMPLEX TRANSFORMS AND ALGORITHMS

Some transforms used in signal processing are complex, most notably the DFT. These transform consequently also have complex algorithms, i.e., DAGs, formulas, and breakdown rules. When implemented on a computer, the complex arithmetic is eventually broken down into real operations, so all our FMA optimization algorithms remain usable. However, there are choices and some important details that we discuss in this section.

Conversion From Complex to Real Arithmetic: The conversion from complex to real arithmetic works as follows. Every complex addition can be converted into two real additions. Every complex multiplication $y = cx$ can be converted into

$$\begin{bmatrix} y_r \\ y_i \end{bmatrix} = \begin{bmatrix} c_r & -c_i \\ c_i & c_r \end{bmatrix} \begin{bmatrix} x_r \\ x_i \end{bmatrix} \quad (56)$$

where the subscripts indicate the real and imaginary parts. (56) can be implemented using three additions and three multiplications, but for our purpose, the straightforward way with two additions and four multiplications is preferable, since multiplications can be fused. If FMAs are to be used, then (56) can be done using two additions and two FMAs. If the multiplications happen to be by a root of unity, i.e., the matrix in (56) is a rotation, then three FMAs can be used instead (see (61) in Section VII below).

Finally, a complex FMA of the form $y = x_1 + cx_2$ corresponds to

$$\begin{bmatrix} y_r \\ y_i \end{bmatrix} = \begin{bmatrix} 1 & 0 & c_r & -c_i \\ 0 & 1 & c_i & c_r \end{bmatrix} \begin{bmatrix} x_{1,r} \\ x_{1,i} \\ x_{2,r} \\ x_{2,i} \end{bmatrix}$$

which is a perfect FMA formula in the sense of Definition 1 and can be implemented with four real FMAs.

Complex FMA Optimization: Assume a given complex DAG (or formula) to be FMA optimized. We have the following two choices:

- 1) first, we FMA optimize the complex DAG using Algorithm 1, then we convert the resulting complex FMA DAG to a real DAG as described above;
- 2) first, we convert into a real DAG as described above, then we FMA optimize the DAG using Algorithm 1.

First, it is interesting to establish that the two methods in general not only produce different results but also results with different cost. Consider the following example of a complex formula, which we first FMA optimize and then convert to real arithmetic:

$$\begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix} \text{diag}(1, c) \rightarrow \begin{bmatrix} 1 & c \\ 1 & -c \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & c_r & -c_i \\ 0 & 1 & c_i & c_r \\ 1 & 0 & -c_r & c_i \\ 0 & 1 & -c_i & -c_r \end{bmatrix}.$$

The result costs eight FMAs. Now, we do the steps in the opposite order: first, we convert to real arithmetic, then we FMA optimize

$$\begin{aligned}
 & \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \text{diag}(1, c) \\
 & \rightarrow \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & c_r & -c_i \\ 0 & 0 & c_i & c_r \end{bmatrix} \\
 & \rightarrow \begin{bmatrix} 1 & 0 & c_r & 0 \\ 0 & 1 & 0 & c_r \\ 1 & 0 & -c_r & 0 \\ 0 & 1 & 0 & -c_r \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -c_i/c_r \\ 0 & 0 & c_i/c_r & 1 \end{bmatrix}. \quad (57)
 \end{aligned}$$

The result costs six FMAs, two less than before. Note that in the last step, we could have propagated c_i instead of c_r , or a mix of both.

The question now is whether the second method is always superior. This is not the case as one simple example shows. Namely, consider a complex DAG with one input x , one output y , and two consecutive multiplications: $y = dcx$. Let $a = dc$. Then, the first method will produce, via rule 2 in Table II, as follows:

$$dc \rightarrow a \rightarrow \begin{bmatrix} a_r & -a_i \\ a_i & a_r \end{bmatrix}$$

with a cost of two FMAs and two multiplications. The second method yields

$$\begin{aligned}
 dc & \rightarrow \begin{bmatrix} d_r & -d_i \\ d_i & d_r \end{bmatrix} \begin{bmatrix} c_r & -c_i \\ c_i & c_r \end{bmatrix} \\
 & \rightarrow \text{diag}(c_r d_r, c_r d_r) \begin{bmatrix} 1 & -d_i/d_r \\ d_i/d_r & 1 \end{bmatrix} \begin{bmatrix} 1 & -c_i/c_r \\ c_i/c_r & 1 \end{bmatrix}
 \end{aligned}$$

with a cost of four FMAs and two multiplications.

However, it turns out that this is the only case, i.e., if rule 2 is not used in the first method, then the second method is always better in a specified sense. We record this in the following theorem including bounds on the cost.

Theorem 3: Let \mathcal{D} be a complex DAG with n output nodes α , additions, and μ multiplications (all complex). Assume method 2 is applied (first convert to real arithmetic, then FMA optimize) to get a real DAG $\overline{\mathcal{D}}$ with $\overline{\alpha}$ additions, $\overline{\mu}$ multiplications, and $\overline{\phi}$ FMAs (all real). Then

$$\overline{\alpha} + \overline{\phi} = 2(\alpha + \mu) \quad (58)$$

$$\overline{\mu} \leq 2n \quad (59)$$

$$\text{cost}(\overline{\mathcal{D}}) = \overline{\alpha} + \overline{\mu} + \overline{\phi} \leq 2(\alpha + \mu + n). \quad (60)$$

The bounds are sharp. Further, if method 1 (first FMA optimize, then convert to real arithmetic) is applied and rule 2 in Table II is never used in the FMA optimization, then method 2 is better with respect to the worst case cost of the output DAG.

Proof: The real DAG corresponding to \mathcal{D} has $2\alpha + 2\mu$ additions, 4μ multiplications, and $2n$ outputs. Now, (58)–(60) follow from Theorem 1. Starting with a complex diagonal matrix yields the sharpness of all the bounds.

Now assume we use method 1. FMA optimization of \mathcal{D} yields a complex FMA DAG with a additions, m multiplications, and

f FMAs (all complex) and the equations in Theorem 1 hold including (10) since we assume rule 2 is never used: $a + f = \alpha$, $m \leq n$, $m + f \geq \mu$. Converting to real arithmetic yields the DAG $\overline{\mathcal{D}}$ with $2a$ additions, $2m$ multiplications, and $4f + 2m$ FMAs. Now, the sharp worst case bound becomes

$$\text{cost}(\overline{\mathcal{D}}) = 2a + 4m + 4f \leq 2(\alpha + n) + 2(m + f).$$

Since $m + f \geq \mu$, the result follows. \blacksquare

VII. RESULTS AND DISCUSSION

A. Comparison to Published FMA Algorithms

FMA Optimization of DAGs: We implemented Algorithm 1 as backend to the Spiral program generator [11] and applied it to a number of transforms and sizes. Spiral contains most of the published breakdown rules and thus is able to generate many different algorithms (or formulas) for any given transform. Each of these formulas is compiled into actual code and its runtime measured. A search mechanism then finds the fastest implementation for the transform on the given computing platform. In our first experiment, we were not interested in runtime (which is considered below in Section VII-C) but in the lowest arithmetic cost achievable. Thus, instead of searching for the fastest, we modified Spiral to search for the algorithm with the lowest cost. The results are in Table VII and explained next.

First, we performed the search with standard scalar code (counting additions and multiplications only). The results are in the column labeled “Std.” Next, we enabled the FMA optimization backend (Algorithm 1) in Spiral and repeated the search, this time generating the algorithms with the minimum FMA cost. The results are in the column “FMA.” The column “%imp” shows the total arithmetic cost improvement in percent.

We compared our generated algorithms against the best published (to our best knowledge) hand-derived algorithms in the last column. This also includes FFTW [20], [21], a library that provides some of the transforms and which also uses a program generator for small sizes but considers only one algorithm for each transform. For some transforms, no benchmark was available. For the others, our generated algorithms match in practically all cases the benchmarks. The fact that they match exactly is not entirely surprising, since the hand-derived FMA algorithms are, in most cases, obtained by implicitly performing an instantiation of our general method.

The gains in arithmetic cost achievable with FMA optimization increase with the transform size n . Given a class of algorithms, it is straightforward to obtain an upper bound on the asymptotic behavior of this gain. For example, for DFTs of two-power size n , one of the best available algorithms is the split-radix FFT [22], which requires $(8/3)n \log_2(n) + O(n)$ real additions and $(4/3)n \log_2(n) + O(n)$ real multiplications. Hence, if all multiplications are fused, the gain approaches 1/3. Similarly, for the DCT, type 2 and 3, many of the best algorithms require $(3/2)n \log_2(n) + O(n)$ additions and $(1/2)n \log_2(n) + O(n)$ multiplications [23]. Hence, in the best case, the gain approaches 1/4.

We also automatically reproduced other hand-derived algorithms, including the radix-2, -3, -4, and -5 DFT kernels in [4],

TABLE VII
BEST ARITHMETIC COST FOR TRANSFORMS FOUND BY SPIRAL
FOR STANDARD AND FMA ALGORITHMS ALONG WITH THE
COST OF THE BEST KNOWN FMA ALGORITHMS

	n	Std	FMA	%imp	Best published
DFT $_n$	3	16	12	25	12 [3]
	4	16	16	0	16 [3]
	5	48	32	33	32 [3]
	6	44	36	18	36 [19]
	7	96	60	37	60 [19]
	8	56	52	7	52 [2]
	9	120	80	33	80 [19]
	10	116	84	28	84 [19]
	11	240	140	42	140 [19]
	12	112	96	14	96 [19]
	13	292	216	26	176 [19]
	14	220	148	33	148 [19]
	15	224	156	30	156 [19]
	16	168	144	14	144 [2]
	32	456	372	18	372 [2]
	RDFT $_n$	3	6	5	17
4		6	6	0	6 [3]
5		18	14	22	14 [19]
6		18	16	11	16 [19]
7		42	27	36	27 [19]
8		22	20	9	20 [3]
9		52	36	31	42 [19]
10		46	38	17	38 [19]
11		110	65	41	65 [19]
12		46	40	13	40 [19]
13		114	82	28	82 [19]
14		98	68	31	68 [19]
15		92	71	23	71 [19]
16		70	58	17	58 [3]
32		198	156	21	156 [3]
DCT-2 $_n$		3	6	5	17
	4	13	10	23	-
	5	18	14	22	-
	6	22	20	9	-
	7	42	27	36	-
	8	41	30	27	-
16	113	82	27	-	
DCT-3 $_n$	4	13	8	38	-
	8	41	26	37	-
	16	113	72	36	-
DCT-4 $_n$	4	20	12	40	-
	8	56	36	36	-
	16	144	96	33	-
IMDCT $_n$	6	33	21	36	-
	18	168	109	35	-

the radix-6 DFT kernel in [6], the radix-16 DFT kernel in [7], and the 1-D and 2-D scaled DCT-2 of sizes 8 and 8×8 from [8]. In the latter reference, the authors do not consider the actual DCT-2 but a scaled version since in the considered applica-

tion (JPEG image compression) the scaling factors can be fused with a subsequent scaling step. Our algorithm always propagates all multiplications to the output and is hence particularly well suited for these scenarios.

FMA Optimization of Formulas and Breakdown Rules:

Reference [3] describes several DFT algorithms using formulas and shows how to implement these algorithms with FMAs. The authors do not express the resulting FMA algorithm using formulas, but Algorithm 2 reproduces their results.

The application of Algorithm 3 to radix-2, radix-4, radix-8, and split-radix DFT breakdown rules, yields recursive DFT FMA algorithms similar to the ones in [2].

B. Other Ways to Perform FMA Optimization

Rotations via Lifting Steps: Many transforms are orthogonal and have “orthogonal” algorithms. This means that the algorithm consists exclusively of butterflies F_2 and 2×2 Givens rotation matrices R_α defined as

$$R_\alpha = \begin{bmatrix} \cos \alpha & \sin \alpha \\ -\sin \alpha & \cos \alpha \end{bmatrix}.$$

For a general α , R_α incurs four additions and two multiplications, or, alternatively, two multiplications and two FMAs. It is well known, however, that a rotation can be factored into three lifting steps, each requiring a single FMA for a total of three FMAs [24], [25]:

$$R_\alpha = \begin{bmatrix} 1 & \frac{1-\cos \alpha}{\sin \alpha} \\ & 1 \end{bmatrix} \begin{bmatrix} 1 & \\ -\sin \alpha & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{1-\cos \alpha}{\sin \alpha} \\ & 1 \end{bmatrix}. \quad (61)$$

This is a simple method for FMA optimization and optimal for a standalone rotation. However, in the context of a larger algorithm lifting steps are suboptimal in general, since our propagation method produces two FMAs only and two leftover multiplications, which potentially can be fused with later subsequent additions.

Converting Additions Into Multiplications: In certain cases, it is possible to modify a transform algorithm to trade additions for multiplications with an integer constant. The simplest example of such a modification is $x + x = 2x$. Coppersmith and Linzer in [9] exploit such modifications to show that the Walsh–Hadamard transform (WHT) kernel of size 4, which requires eight additions and no multiplications, can be modified to use seven additions and three multiplications. The latter are then fused with the additions for a total cost of only seven operations.

The WHT of size 4 is equal to the discrete Hartley transform (DHT) of size 4; thus, this technique could also be used to reduce the FMA operation count of algorithms that use size-4 DHT kernels.

Heuristic Method for DAGs: In [10], we also described an alternative heuristic method for performing FMA optimization of DAGs, which can yield better arithmetic cost by not always propagating multiplications to improve on situations like Fig. 2. The method involves using cost-weighted propagation rules and can further reduce the number of unfused multiplications. However, when combined with search over different available algorithms, it did not yield relevant improvements.

C. FMA Optimization and Runtime

An interesting issue is the effect of FMA optimization on the program's runtime. It is well known that on modern architectures, the runtime is often dominated by other factors besides arithmetic cost. For example, even though our FMA optimization method reduces the number of instructions, it may increase the total amount of performed multiplications [counting the multiplications in FMAs; see (11)]. Since the constants are held in registers, this increases the register pressure. Further, FMA optimization increases the number of unique constants and hence the number of load instructions. Both of the above reduce the performance gains achievable by FMA optimization. Finally, the effect of FMA optimization on performance depends on the platform and compiler considered.

We performed an evaluation of our FMA optimization for small transforms, implemented in straightline code (no loops) using Algorithm 1. As in Section VII-A, we used Spiral with FMA optimization in the backend. For a user-selected transform, Spiral enumerates many different programs, measures their runtime and returns the fastest on the computer on which it is installed. Using Spiral, we generated the fastest implementations for the same set of transforms as in Section VII-A with and without FMA optimization enabled. This implies that Spiral may find different algorithms for standard and FMA code and is, hence, a very fair evaluation of the performance potential of FMA operations for transforms.

Our benchmark platform was a 1500-MHz Itanium 2 computer with 4 GB of RAM, using the Intel Compiler 9.0 with command line options “-O3 -IPF-fp-relaxed.”

The results for various transforms are shown in Table VIII and organized as follows. The first column shows the transform size. The remaining columns are organized into three groups. In each group, the fastest found standard C code is compared with the fastest found FMA optimized C code. In the first group, called Ops (C), the number of operations in the C source code is compared. In the second group, called Ops (assembly), the number of arithmetic operations in the compiler generated assembly code is compared. These numbers are important, because the C compiler performs its own optimizations which create new FMAs from existing adds and multiplies, and can potentially destroy our generated FMAs. Finally, in the third group the actual runtime (in processor cycles) is compared. Note that on the Intel Itanium processors, unlike in the Pentium family, the hardware cycle counter is very stable and precise, and these runtimes can be reproduced to within a cycle. We have subtracted the function call overhead (14 cycles) from all runtimes.

First, we observe that the runtime improvements are directly related to the improvements in operations in the assembly code but not to the improvements in the C code. This is due to the FMA optimization performed by the compiler. For example, consider the DFT of size 32. The fastest found C code has 456 operations, which the compiler reduces to 400 in the final assembly code. The fastest FMA C code, using our FMA optimization, has 372 operations (including FMAs). This number remains in the assembly code, i.e., the compiler could not further reduce. The same behavior can be observed across all trans-

TABLE VIII
PERFORMANCE EVALUATION OF SPIRAL GENERATED TRANSFORM CODE
WITH AND WITHOUT FMA OPTIMIZATION (USING ALGORITHM 1)
ON AN ITANIUM 2 SYSTEM

	Ops (C)				Ops (assembly)				Runtime, cycles			
	Std	FMA	%imp		Std	FMA	%imp		Std	FMA	%imp	
DFT_n												
2	4	4	0.0		4	4	0.0		10	10	0.0	
3	16	12	25.0		12	12	0.0		18	18	0.0	
4	16	16	0.0		16	16	0.0		18	18	0.0	
5	52	40	23.1		44	40	9.1		33	32	3.0	
6	44	36	18.2		36	36	0.0		26	26	0.0	
7	120	84	30.0		92	84	8.7		56	55	1.8	
8	56	52	7.14		52	52	0.0		35	35	0.0	
9	120	80	33.3		88	80	9.1		54	52	3.7	
10	124	100	19.4		108	100	7.4		62	60	3.2	
11	304	220	27.6		254	220	13.4		142	127	10.6	
12	112	96	14.3		96	96	0.0		57	57	0.0	
13	292	216	26.0		236	216	8.5		137	126	8.0	
14	268	196	26.9		212	196	7.5		119	111	6.7	
15	236	180	23.7		192	180	6.3		109	104	4.6	
16	168	144	14.3		150	144	4.0		87	83	4.6	
32	456	372	18.4		400	372	7.0		223	213	4.5	
RDFT_n												
2	2	2	0.0		2	2	0.0		24	23	4.2	
3	6	5	16.7		5	5	0.0		15	15	0.0	
4	6	6	0.0		6	6	0.0		16	16	0.0	
5	20	14	30.0		14	14	0.0		20	19	5.0	
6	18	16	11.1		16	16	0.0		21	21	0.0	
7	42	27	35.7		27	27	0.0		24	24	0.0	
8	22	20	9.09		20	20	0.0		19	19	0.0	
9	52	36	30.8		40	37	7.5		35	34	2.9	
10	48	38	20.8		38	38	0.0		28	27	3.6	
11	110	65	40.9		65	65	0.0		48	48	0.0	
12	46	40	13.0		39	40	-2.6		28	29	-3.6	
13	156	82	47.4		90	82	8.9		58	57	1.7	
14	98	68	30.6		68	68	0.0		45	44	2.2	
15	98	71	27.6		71	72	-1.4		46	46	0.0	
16	70	58	17.1		61	58	4.9		41	38	7.3	
32	198	158	20.2		170	158	7.1		98	91	7.1	
DCT-2_n												
2	3	3	0.0		3	3	0.0		13	13	0.0	
3	6	5	16.7		5	5	0.0		14	14	0.0	
4	13	11	15.4		11	11	0.0		19	19	0.0	
5	20	14	30.0		14	14	0.0		22	20	9.1	
6	22	20	9.09		19	20	-5.3		26	26	0.0	
7	42	27	35.7		27	27	0.0		26	24	7.7	
8	41	33	19.5		33	33	0.0		27	28	-3.7	
16	113	87	23.0		89	87	2.2		59	57	3.4	
32	289	221	23.5		231	221	4.3		142	138	2.8	
DCT-3_n												
4	13	8	38.5		9	8	11.1		18	17	5.6	
8	41	26	36.6		31	26	16.1		29	25	13.8	
16	113	72	36.3		90	72	20.0		57	52	8.8	
32	289	186	35.6		209	186	11.0		137	127	7.3	
DCT-4_n												
4	20	14	30.0		14	14	0.0		24	24	0.0	
8	56	38	32.1		41	38	7.3		34	33	2.9	
16	144	98	31.9		111	98	11.7		72	73	-1.4	
32	352	238	32.4		271	238	12.2		165	153	7.3	
IMDCT_n												
6	34	24	29.4		26	27	-3.8		33	29	12.1	
18	168	118	29.8		130	127	2.3		103	102	1.0	

forms, which means that our FMA optimization is better than the compiler's.

Regarding runtimes, we also observe that small transforms generally do not benefit from FMA optimization. They usually have little or no multiplications. In contrast, most larger sizes benefit from our FMA optimization.

In very few cases, there is a small performance degradation. For RDFT₁₂ and DCT-2₆, our FMA optimization produces slightly suboptimal arithmetic cost compared to the compiler's optimization. This is due to a situation similar to Fig. 2. In two other cases of slowdowns, RDFT₁₅ and IMDCT₆, the C compiler destroys our generated FMAs and increases the operations count.

The compiler options deserve special attention. The compiler cannot be forced to use FMAs, and by default the GNU C compiler 4.2 (gcc) and the Intel compiler 9.0 generate suboptimal code for a pair of "overlapping" FMAs such as $a + b \times c$ and $a - b \times c$. In this case, $b \times c$ is treated as a common subexpression and computed separately from the addition and subtraction, resulting in three operations rather than the obvious two FMAs. There is no way to solve this in gcc. However, the Intel compiler developers told us that they studied this problem, and implemented a special compiler pass [26] to reconstruct locally optimal FMA instruction sequences. This pass is enabled with "-IPF-fp-relaxed" and is the reason why we used this option. As a consequence, in most cases our FMA cost is unaltered by the compiler.

Finally, we note that performance optimization of software is only one application of FMA optimization. For example, in specialized hardware implementations, MAC units are always an attractive solution, since they require little more area than a multiplier and can increase accuracy.

VIII. CONCLUSION

We have shown that the derivation of FMA algorithms for linear transforms is straightforward using a set of transformation rules. The method fuses all multiplications except as many as the transform has outputs in the worst case. Further, the method is applicable to complex and real transform alike. We have shown three flavors of the method in Algorithms 1, 2, and 3 that mathematically perform the same procedure, but differ in the representation of the given transform algorithm.

We implemented the DAG-based method (Algorithm 1) in Spiral and were able to automatically reproduce many hand-derived FMA algorithms from the literature as well as produce some new ones.

In summary, this paper provides a general tool to map any linear transform to any FMA architecture automatically and efficiently.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful suggestions to improve the paper, and K. Serebryany for his suggestions on how to prevent the Intel compiler from destroying FMA operations.

REFERENCES

- [1] Y. Nievergelt, "Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit," *ACM Trans. Math. Softw. (TOMS)*, vol. 29, no. 1, pp. 27–48, 2003.
- [2] E. Linzer and E. Feig, "Implementation of efficient FFT algorithms on fused multiply-add architectures," *IEEE Trans. Signal Process.*, vol. 41, no. 1, p. 93, Jan. 1993.
- [3] C. Lu, "Implementation of multiply-add FFT algorithms for complex and real data sequences," in *Proc. Int. Symp. Circuits Systems (ISCAS)*, 1991, vol. 1, pp. 480–483.
- [4] S. Goedecker, "Fast radix 2, 3, 4, and 5 kernels for fast Fourier transformations on computers with overlapping multiply-add instructions," *SIAM J. Scientif. Comput.*, vol. 18, no. 6, pp. 1605–1611, 1997.
- [5] C. Lu, J. W. Cooley, and R. Tolimieri, "FFT algorithms for prime transform sizes and their implementations on VAX, IBM3090VF, and IBM RS/6000," *IEEE Trans. Signal Process.*, vol. 41, no. 2, pp. 638–648, Feb. 1993.
- [6] D. Takahashi, "A new radix-6 FFT algorithm suitable for multiply-add instruction," in *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP)*, Jun. 2000, vol. 6, pp. 3343–3346.
- [7] D. Takahashi, "A radix-16 FFT algorithm suitable for multiply-add instruction based on Goedecker method," in *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP)*, Apr. 2003, vol. 2, pp. 665–668.
- [8] E. Linzer and E. Feig, "New scaled DCT algorithms for fused multiply/add architectures," in *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP)*, 1991, vol. 3, pp. 2201–2204.
- [9] E. F. D. Coppersmith and E. Linzer, "Hadamard transforms on multiply/add architectures," *IEEE Trans. Signal Process.*, vol. 42, no. 4, pp. 969–970, Apr. 1994.
- [10] Y. Voronenko and M. Püschel, "Automatic generation of implementations for DSP transforms on fused multiply-add architectures," in *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP)*, 2004, vol. 5, pp. 101–104.
- [11] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proc. IEEE (Special Issue on Program Generation, Optimization, and Adaptation)*, vol. 93, no. 2, pp. 232–275, 2005.
- [12] Spiral website [Online]. Available: www.spiral.net, ", " vol. , pp. –,
- [13] P. Bürgisser, M. Clausen, and M. A. Shokrollahi, *Algebraic Complexity Theory*. New York: Springer, 1997.
- [14] C. Van Loan, *Computational Framework of the Fast Fourier Transform*. Philadelphia, PA: SIAM, 1992.
- [15] N. Dershowitz and D. A. Plaisted, "Rewriting," in *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. New York: Elsevier, 2001, vol. 1, ch. 9, pp. 535–610.
- [16] F. Franchetti, Y. Voronenko, and M. Püschel, "Loop merging for signal transforms," in *Proc. Programming Language Design Implementation (PLDI)*, 2005, pp. 315–326.
- [17] F. Franchetti and M. Püschel, "A SIMD vectorizing compiler for digital signal processing algorithms," in *Proc. IEEE Int. Parallel Distributed Processing Symposium (IPDPS)*, 2002, pp. 20–26.
- [18] F. Franchetti, Y. Voronenko, and M. Püschel, "FFT program generation for shared memory: SMP and multicore," in *Proc. Supercomputing*, 2006.
- [19] FFTW 3.1.2 2006 [Online]. Available: www.fftw.org
- [20] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP)*, 1998, vol. 3, pp. 1381–1384 [Online]. Available: www.fftw.org
- [21] M. Frigo, "A fast Fourier transform compiler," in *Proc. ACM SIGPLAN Conf. Programming Language Design Implementation (PLDI)*, 1999, pp. 169–180.
- [22] H. V. Sorensen, H. M. T. , C. S. Burrus, and M. T. Heideman, "On computing the split-radix FFT," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-34, no. 1, pp. 152–156, 1986.
- [23] K. R. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, Applications*. New York: Academic, 1990.
- [24] F. Bruekers and A. Enden, "New networks for perfect inversion and perfect reconstruction," *IEEE J. Sel. Areas Commun.*, vol. 10, pp. 130–137, Jan. 1992.
- [25] W. Sweldens, "The lifting scheme: A custom-design construction of biorthogonal wavelets," *Appl. Comput. Harmon. Anal.*, vol. 3, no. 2, pp. 186–200, 1996.
- [26] K. Serebryany, "Optimal placement of fused multiply-add (FMA) instructions," presented at the 6th Workshop on Explicitly Parallel Instructions, Computing Architectures and Compiler Technology (EPIC-6), San Jose, CA, Mar. 2007.



Yevgen Voronenko (S'04) received the B.S. degree in computer science from Drexel University, Philadelphia, PA, in 2003. He is currently working towards the Ph.D. degree in electrical and computer engineering at Carnegie Mellon University, Pittsburgh, PA.

His research interests include rewriting systems, software engineering, programming languages, and compiler design.



Markus Püschel (M'00–SM'05) received the Diploma (M.Sc.) degree in mathematics and the Ph.D. degree in computer science from the University of Karlsruhe, Germany, in 1995 and 1998, respectively.

From 1998 to 1999, he was a Postdoctoral Researcher in mathematics and computer science, Drexel University, Philadelphia, PA. Since 2000, he has been with Carnegie Mellon University, Pittsburgh, PA, where he is an Associate Research Professor of electrical and computer engineering.

His research interests include computing, compilers, applied mathematics and algebra, and signal processing theory/software/hardware.

Dr. Püschel is on the Editorial Board of the IEEE TRANSACTIONS ON SIGNAL PROCESSING and was on the Editorial Board of the IEEE SIGNAL PROCESSING LETTERS and a Guest Editor of the *Journal of Symbolic Computation* and the PROCEEDINGS OF THE IEEE.