AUTOMATIC GENERATION OF IMPLEMENTATIONS FOR DSP TRANSFORMS ON FUSED MULTIPLY-ADD ARCHITECTURES

Yevgen Voronenko and Markus Püschel

Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA, U.S.A.

ABSTRACT

Many modern computer architectures feature fused multiply-add (FMA) instructions, which offer potentially faster performance for numerical applications. For DSP transforms, compilers can only generate FMA code to a very limited extent because optimal use of FMAs requires modifying the chosen algorithm. In this paper we present a framework for automatically generating FMA code for every linear DSP transform, which we implemented as an extension to the SPIRAL code generation system. We show that for many transforms and transform sizes, our generated FMA code matches the best-known hand-derived FMA algorithms in terms of arithmetic cost. Further, we present actual runtime results that show the speed-up obtained by using FMA instructions.

1. INTRODUCTION

Many modern processor architectures, including the Motorola G4, Intel Itanium and Itanium 2, feature *fused multiply-add (FMA)* instructions, which perform an operation of the form

$$y = a * x_1 + x_2 \quad \text{or} \quad y = a * x_1 - x_2$$

as fast as a single addition or multiplication. Thus, by using these instructions, a speed-up can be obtained for computations that contain a mix of additions and multiplications. One example of such computations are algorithms for linear digital signal processing (DSP) transforms, such as the discrete Fourier transform (DFT), the discrete cosine transforms (DCTs), and many others. Since general-purpose compilers can only make very restricted use of FMAs for these algorithms, there has been a number of efforts to mathematically convert transform algorithms into FMA versions.

FMA algorithms for the DFT, based on the Cooley-Tukey FFT, the split-radix FFT, the prime factor FFT, and Winograd's algorithms were developed in [1], [2]. Reference [3] presents FMA algorithms for the scaled DCT, type II, of size 8 and its inverse in one and two dimensions.

All of the available literature has one aspect in common: authors modify the algorithms *manually*, acting as "human compilers." In this paper we describe an *entirely automatic* method to convert any given transform algorithm into an FMA algorithm. Based on our method we can prove an upper bound of the multiplications that are "left over," i.e., not fusable with an addition. We incorporated our FMA method as a backend into the SPIRAL transform code generation system [4, 5], and used it to validate our approach in two directions. First, we show that in many cases the arithmetic cost of our generated FMA algorithms matches the cost of the best-known hand-derived algorithms. Second, we present results showing the actual runtime speed-up achieved by generating FMA code versus the fastest SPIRAL generated scalar code.

This work was supported by NSF awards 0234293 and 0310941.

Organization. In Section 2 we introduce the transforms we consider, briefly explain the SPIRAL code generator, and introduce the representation of algorithms as directed acyclic graphs (DAGs). Section 3 describes two methods for converting a given transform algorithm, represented as a DAG, into an FMA algorithm, gives bounds for the arithmetic cost of this FMA algorithm, and discusses the methods' limitations. We conclude by presenting experimental results in Section 4.

2. BACKGROUND

Transforms. There are many different DSP transforms used in signal processing. In this paper we consider the DFT, its real version (RDFT), the DCTs of type II–IV, and the inverse modified discrete cosine transform (IMDCT). They are defined (in their unscaled versions) by the matrices

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

Note that $\mathrm{DCT}^{(\mathrm{III})}$ is the transpose of $\mathrm{DCT}^{(\mathrm{II})}$ and that IMDCT_n is an $2n \times n$ matrix. The sizes 36×18 and 12×6 are used in MP3 audio coding.

Algorithms. For each transform there are several different ways to compute it recursively from smaller transforms. Mathematically, these recursions, or *rules*, can be written as a factorization of the transform matrix into a product of structured sparse matrices. Here are a few simple examples:

$$\begin{aligned}
\mathrm{DFT}_{nm} &= (\mathrm{DFT}_n \otimes \mathbf{1}_m) \cdot D_{n,m} \cdot (\mathbf{1}_n \otimes \mathrm{DFT}_m) \cdot P_{n,m},\\
\mathrm{DCT}_{2n}^{(\mathrm{II})} &= P_{2n} \cdot (\mathrm{DCT}_n^{(\mathrm{II})} \oplus \mathrm{DCT}_n^{(\mathrm{IV})}) S_{2n},\\
\mathrm{DCT}_n^{(\mathrm{IV})} &= S_n \cdot \mathrm{DCT}_n^{(\mathrm{II})} \cdot D_n,\\
\mathrm{IMDCT}_n &= S_n \cdot \mathrm{DCT}_n^{(\mathrm{IV})},
\end{aligned}$$

where P, D, S denote certain permutations, diagonals, and other sparse matrices, respectively; their exact form is not of importance in this paper. The symbols may have different meanings in different recursions. Further, $\mathbf{1}_n$ denotes the $n \times n$ identity matrix, $\oplus$

denotes the direct sum, and $\otimes$ the tensor or Kronecker product of matrices, defined by

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

Recursively applying these rules, until all occurring transforms are expanded, yields a *formula* representing an algorithm for the original transform. The choices of rules at each level yields, for each transform, a large space of alternative formulas with practically equal arithmetic cost but different data flow. For example, the current version of SPIRAL reports 19,504 different formulas for the $\mathrm{DCT}^{(\mathrm{II})}_{16}$.

SPIRAL. SPIRAL is a code generator for DSP transforms [4, 5]. For a given transform, SPIRAL generates one out of many possible formulas, i.e., algorithms, translates it into code, measures its runtime, and, in a feedback loop, triggers the generation of different formulas, thus searching for the best match between algorithm and target platform.

SPIRAL provides us with an ideal environment to include our FMA generation technique and evaluate its performance. For example, for a given transform, besides searching for the fastest FMA code, we can also search for the FMA algorithm with the best arithmetic cost, i.e., the one that requires the fewest operations.

DAGs. Our method for generating FMA algorithms does not operate on the formula representation of an algorithm, but on a different, equivalent representation of the algorithm as a *directed acyclic graph* or *DAG* describing the data flow of the computation. The nodes in the graph are the arithmetic operations, incoming edges are the operands, and outgoing edges are the result. DAGs are best explained through an example. Fig. 1 (a) shows an algorithm for the $\mathrm{DCT}^{(\mathrm{II})}_4$ as sparse matrix product. The corresponding DAG is shown in Fig. 1 (b). The white nodes are additions (or subtractions), and the shaded nodes are multiplications by the denoted constant. For simplicity we restrict ourselves to binary additions, so that in our DAGs an addition of three values is represented as two consecutive additions. Fig. 1 (c) shows an FMA DAG for the $\mathrm{DCT}^{(\mathrm{II})}_4$ obtained from Fig. 1 (b) using our method presented in Section 3. The rectangular boxes containing a number a denote an FMA instruction $y = a * x_1 + x_2$; the bold incoming edge denotes the operand that is multiplied by a.

We consider two types of DAGs in this paper. *Standard DAGs* only contain additions and multiplications and are denoted by D. *FMA DAGs* contain additions, multiplications, and FMAs and are denoted by $\overline{D}$. We define the *(arithmetic) cost* of a DAG, written $\mathrm{cost}(D)$ or $\mathrm{cost}(\overline{D})$, as its number of nodes.

3. FMA DAG GENERATION

The goal of our FMA generation method is to convert a given DSP transform algorithm, represented as a standard DAG D, into an FMA DAG $\overline{D}$ that requires fewer arithmetic operations and, barring numerical precision, has the same input/output behavior. Formally,

$$D \Rightarrow \overline{D}, \quad \text{where} \quad \mathrm{cost}(\overline{D}) \leq \mathrm{cost}(D).$$

In other words, we want to *fuse* multiplications and additions in D to the maximum extent possible. A straightforward way to achieve this is to find subexpressions in D of the form $a * x_1 + x_2$ and convert them into an FMA, which is what general purpose compilers do. However, this does not deal efficiently with cases such as $a * x_1 + b * x_2$, which will always leave an unfused multiplication,

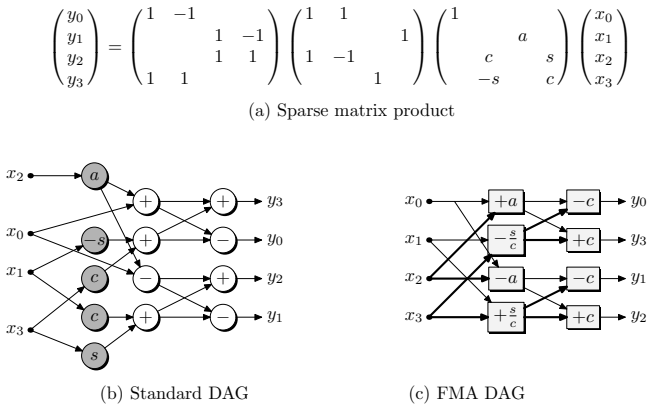

Fig. 1. Different representations of an algorithm for $\mathrm{DCT}^{\mathrm{III}}_4$

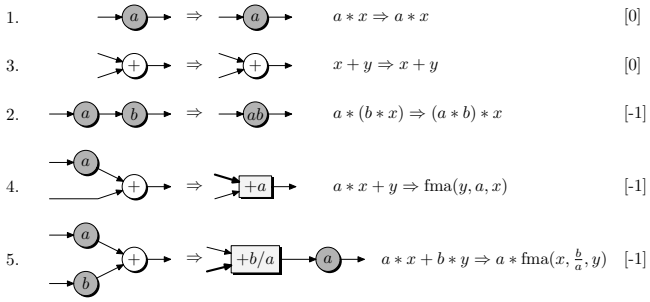

Fig. 2. DAG transformation rules for the basic method. Applying a rule changes the cost of the DAG by the number in square brackets.

whereas it is often possible to do better. As in previous work, we use the fact that

$$a * x + b * y = a(x + \tfrac{b}{a} y), \tag{1}$$

i.e., the multiplication by a is *propagated* so that it may be fused with subsequent additions. We present two methods based on this principle:

- The *basic method* is an unconditioned successive propagation of multiplications.
- The *heuristic method* improves the basic method by using a cost measure to decide whether a propagation is beneficial.

Basic method. The basic method is based purely on a propagation of multiplications. It can be described by a set of DAG manipulation rules that cover all possible local node configurations. There are five essential rules, which are given in Fig. 2 in two equivalent forms: as a DAG manipulation and as a code manipulation. The rightmost column shows the effect of the rule on the cost of the DAG in square brackets. The rules cover all possible cases of nodes.

Algorithm 1 (Basic method). *Given a standard DAG D for a transform algorithm, convert D into an FMA DAG $\overline{D}$ with equivalent input/output behavior.*

Traverse the nodes of the DAG from input to output, visiting each node exactly once. In each step, an unvisited node is selected for which all predecessors (operands) have been already visited. If the current node is a multiplication:

- *If the predecessor is a multiplication, apply rule 3 (fuse the multiplications).*
- *Otherwise, if there is no predecessor or the predecessor is an addition, apply rule 1 (do nothing).*

If the current node is a addition:

- *If exactly one predecessor is a multiplication, apply rule 4 (create an FMA).*
- *If both predecessors are multiplications, apply rule 5 (create an FMA and propagate a multiplication).*
- *Otherwise, apply rule 2 (do nothing).*

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

Informally, the algorithm always propagates a multiplication if possible, otherwise it tries to create an FMA first, and if this is not possible it resorts to using the base-case multiply and add rules. Note that propagation may result in two consecutive multiplications by a constant, which can then be premultiplied (rule 3, which is often referred to as constant folding).

It is intriguing that this rather simple algorithm allows us to prove an upper bound for the multiplications left "unfused" and thus an upper bound for $\mathrm{cost}(\overline{D})$.

Theorem 1. *Assume that the standard DAG D for a DSP transform algorithm contains A additions, M multiplications, and n outputs. Further, assume the output $\overline{D}$ of Algorithm 1 contains $\overline{A}$ additions, $\overline{M}$ multiplications and $\overline{F}$ FMAs. Then*

$$\overline{A} + \overline{F} = A, \quad (2)$$

$$\overline{M} \leq n, \quad (3)$$

$$\mathrm{cost}(\overline{D}) = \overline{A} + \overline{M} + \overline{F} \leq A + n. \quad (4)$$

Further, the bounds in (3) *and* (4) *are sharp.*

Proof. Inspecting rules 1–5 in Fig. 2 shows that each one leaves the number of additions (in the mathematical sense, i.e., also counting the additions in FMAs) unchanged. This yields (2).

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

Consider a DAG for a diagonal matrix, i.e., every input is multiplied to yield the output. In this case equality holds in (3) and (4), thus these bounds are sharp. □

Implementation. For the implementation of the basic method we have used an adapted version of *iburg* [6], a generator of code generators for expression trees (each node has a unique parent). *iburg* takes as input a specification of a composite instruction set in the form of a cost-weighted *tree grammar* and a set of functions which traverse the expression tree. It outputs a code generator that converts a given tree into a minimal-cost expression tree built from the specified instructions (called tree cover) in $O(n)$ time, where n is the number of nodes in the tree.

In our case, we need a DAG cover (versus a tree cover) for which *iburg* can not be directly used. We have modified *iburg* to generate code generators for DAGs using the approach described in [7]. The author proposes treating overlapping tree constituents of the DAG independently of each other, discussing the optimality for this case. In general, optimality is no longer guaranteed, because shared nodes can cause suboptimal covering.

For the basic method, the tree grammar from Fig. 2 was used directly and the *iburg*-generated code generator is equivalent to Algorithm 1. Since the number of rules is small, it is easy to identify instances of suboptimal covering, i.e., the shortcomings of Algorithm 1. The main problem is demonstrated in Fig. 3. Since the basic method considers the overlapping subtrees separately it will propagate multiplications in two separate subtrees, resulting in cost of 4 ($\overline{D}$); not propagating a yields the (optimal) cost of 3 ($\overline{D}_{\mathrm{opt}}$).

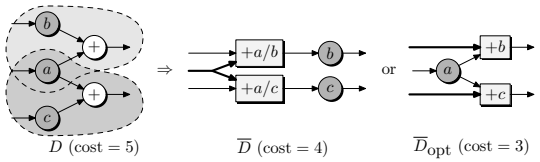

Fig. 3. Example were the basic method is sub-optimal.

Heuristic method. As a workaround to the problem in Fig. 3, we introduce a simple heuristic in the *iburg* rules: we assign a penalty for propagating shared multiplications. We set the cost of a regular operation to 10 and the penalty of 1 for propagation of a shared multiplication. Thus, the shared multiplication is propagated only if it decreases the cost of at least one outgoing path.

For example, in Fig. 3, the propagation of the shared multiplication a does not decrease the cost of any path, so the heuristic method will choose not to propagate it and produces $\overline{D}_{\mathrm{opt}}$ instead.

Unfortunately, with this change, Theorem 1 does not hold anymore. *iburg* defers the actual code generation decisions until the entire DAG has been visited, which can lead to a propagation of a shared multiplication along one path and a computation of the same multiplication along another path, which in turn leads to an artificially high number of total multiplications. Thus, using the heuristic method may lead to a worse cost compared to the basic method. However, for the DAGs we evaluated, this was not a problem. Since SPIRAL searches through a large number of different algorithms, it can discard those for which the heuristic method performs worse.

4. RESULTS

To evaluate our FMA generation methods, we included them in the SPIRAL code generator. This way, we were immediately able to perform tests for a variety of transforms and a large number of different algorithms. Furthermore, we could use SPIRAL's search to find, for each transform, the FMA algorithm with minimal cost and the one with the fastest runtime. In both cases we considered the same transforms, namely those defined in Section 2. Since our method is currently restricted to producing straight-line code (i.e., without loops), we only consider small transforms of sizes $n \leq 32$.

Arithmetic cost. In this set of experiments we let SPIRAL search for algorithms with minimal cost. The results are shown in Table 1. The first column shows the transform; the second column its size. The third column is the minimal cost of a standard DAG (algorithm) D found. The fourth and fifth columns ($\overline{D}_1$ and $\overline{D}_2$) show the best cost found with the basic and heuristic method, respectively. Columns three through five represent independent searches, i.e., the underlying algorithms most likely differ. The last column shows the best known (BK) cost of an FMA algorithm and the reference, to our best knowledge. A dash indicates that we did not find a reference.

	n	SPIRAL			BK $\overline{D}$
		D	$\overline{D}_1$	$\overline{D}_2$	
DFT_n	3	16	12	12	12 [2]
	4	16	16	16	16 [2]
	5	52	40	40	32 [2]
	6	44	36	36	-
	8	56	52	52	52 [1]
	16	168	144	144	144 [1]
	32	456	372	372	372 [1]
RDFT_n	3	6	5	5	6 [2]
	4	6	6	6	6 [2]
	5	20	14	14	16 [2]
	6	18	16	16	-
	8	22	20	20	20 [2]
	16	70	58	58	58 [2]
	32	198	156	156	156 [2]
$\mathrm{DCT}_n^{(\mathrm{II})}$	4	13	10	10	-
	8	41	30	30	-
	16	113	82	82	-
$\mathrm{DCT}_n^{(\mathrm{III})}$	4	13	8	8	-
	8	41	26	26	-
	16	113	72	72	-
$\mathrm{DCT}_n^{(\mathrm{IV})}$	4	20	12	12	-
	8	56	36	36	-
	16	144	96	94	-
IMDCT_n	6	33	21	21	-
	18	168	109	109	-

Table 1. Best arithmetic cost found by SPIRAL for standard DAGs and FMA DAGs along with the best known FMA algorithms

	n	D		$\overline{D}_1$			$\overline{D}_1$		
		ns	ops	ns	ops	spd	ns	ops	spd
DFT_n	3	65	16	72	12	-9%	71	12	-8%
	4	60	16	68	16	-13%	68	16	-12%
	5	152	52	132	40	13%	130	40	14%
	6	150	44	129	36	14%	128	36	15%
	8	181	56	152	52	16%	151	52	17%
	16	459	168	419	144	8%	426	144	7%
	32	1640	456	1150	380	30%	1150	380	30%
RDFT_n	3	42	6	44	5	-6%	45	5	-6%
	4	36	6	37	6	-2%	37	6	-3%
	5	77	20	89	14	-16%	80	14	-4%
	6	61	18	69	16	-14%	69	16	-14%
	8	78	22	87	20	-12%	89	20	-14%
	16	214	70	187	60	13%	191	60	11%
	32	576	210	473	164	18%	470	164	18%
$\mathrm{DCT}_n^{(\mathrm{II})}$	4	52	13	56	12	-7%	57	11	-8%
	8	113	41	105	36	7%	107	33	5%
	16	280	113	231	96	18%	229	90	18%
$\mathrm{DCT}_n^{(\mathrm{III})}$	4	63	13	60	9	3%	61	9	2%
	8	103	41	102	29	1%	102	29	1%
	16	292	113	259	81	11%	257	81	12%
$\mathrm{DCT}_n^{(\mathrm{IV})}$	4	64	20	69	16	-7%	69	16	-8%
	8	135	56	115	44	15%	116	40	14%
	16	330	144	326	112	1%	324	104	1%
IMDCT_n	6	112	34	107	33	4%	112	30	0%
	18	485	168	444	147	8%	446	120	8%

Table 2. Fastest runtimes (and associated cost) in ns found by SPIRAL for standard DAGs and FMA DAGs.

We observe that our method always reduces cost and that the best costs found between both methods are equal. Further, our costs match the best-known costs, if available, with the exception of DFT_5 since Winograd's algorithms used in [2] are not yet included in SPIRAL. For the RDFT of sizes 3 and 5 we improve on [2] . More importantly, we provide new FMA algorithms for a variety of transforms including the IMDCT_{18} used in MP3, in which case the FMA cost is about 35% lower than the original cost.

Runtime. In this set of experiments, we searched (again using independent searches) for the best runtimes of standard DAGs D, and FMA DAGs $\overline{D}$ created with the basic and heuristic method ($\overline{D}_1$ and $\overline{D}_2$). The benchmarks were performed on a Power Mac G4 933 MHz, using Apple's gcc 1041 (based on gcc 3.1) for the PowerPC; the command line options were -O2 -fomit-frame-pointer.

The results are collected in Table 2, which is structured analogously to Table 1. For each found algorithm, we provide its runtime in nanoseconds (ns) and its arithmetic cost (ops). For the fastest FMA DAGs found we provide the runtime speed-up (spd) compared to the best standard algorithm found.

We observe that for algorithms with few operations, FMA code is actually slower, but for larger sizes it is consistently faster, up to 30% for a DFT_{32}. Further, lower arithmetic cost does not necessarily imply faster runtime. In fact in most cases, the fastest algorithms are not the ones with the lowest arithmetic cost. Further, both FMA generation methods performed about equally well. Finally we note that Table 2 is a fair evaluation of the speed-up one can expect from FMA code due to SPIRAL's search over the algorithm space.

Conclusion. We presented an automatic method to convert any DSP transform algorithm into an FMA algorithm, proving an upper bound for the left-over multiplications. We included this method in SPIRAL to enable FMA code generation for a large number of transforms. In terms of arithmetic cost our method matches most of the best-known FMA algorithms and also produces new FMA algorithms. A runtime evaluation shows a speed-up of up to 30% for all except very small sizes. Our method is currently restricted to straight-line code but will be extended to loop code and thus to all transform sizes in the near future.

5. REFERENCES

[1] E. Linzer and E. Feig, "Implementation of efficient FFT algorithms on fused multiply-add architectures," in *IEEE Transactions on Signal Processing*, 1993, vol. 41, p. 93.

[2] C. Lu, "Implementation of multiply-add FFT algorithms for complex and real data sequences," in *Proc. ISCAS*, 1991, vol. 1, pp. 480–483.

[3] E. Linzer and E. Feig, "New scaled DCT algorithms for fused multiply/add architectures," in *Proc. ICASSP*, 1991, vol. 3, pp. 2201–2204.

[4] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms," to appear in *Journal of High Performance Computing and Applications*, 2004, http://www.spiral.net.

[5] M. Püschel and J.M.F. Moura, "Generation and manipulation of DSP transform algorithms," in *10th IEEE Digital Signal Processing Workshop*, 2002, pp. 344–349.

[6] C. W. Fraser, D. R. Hanson, and T. A. Proebsting, "Engineering a simple, efficient code-generator generator," *ACM LOPLAS*, vol. 1, no. 3, pp. 213–226, Sep 1992.

[7] M. Anton Ertl, "Optimal code selection in DAGs," in *Proc. POPL*, 1999, pp. 242–249.