Some Optimizations of Hardware Multiplication by Constant Matrices

Nicolas Boullis, Arnaud Tisserand
Arénaire Project (CNRS–ENSL–INRIA–UCBL)
LIP, École Normale Supérieure de Lyon
46 allée d'Italie, F-69364 Lyon, France
Nicolas.Boullis@ens-lyon.fr, Arnaud.Tisserand@ens-lyon.fr

Abstract

This paper presents some improvements on the optimization of hardware multiplication by constant matrices. We focus on the automatic generation of circuits that involve constant matrix multiplication (CMM), i.e. multiplication of a vector by a constant matrix. The proposed method, based on number recoding and dedicated common sub-expression factorization algorithms was implemented in a VHDL generator. The obtained results on several applications have been implemented on FPGAs and compared to previous solutions. Up to 40% area and speed savings are achieved.

1 Introduction

Important optimizations of the speed, area and power consumption of circuits can be achieved by using dedicated operators instead of general ones whenever possible. Multiplication by constant is a typical example. Indeed, if one operand of the multiplication is constant, one can use some shifts and additions/subtractions to perform the operation instead of using a full multiplier. This usually leads to smaller, faster and less power-consuming circuits.

Applications involving multiplication by constant are common in signal processing, image processing, control and data communication. Finite impulse response (FIR) filters, discrete cosine transform (DCT) and discrete Fourier transform (DFT), for instance, are central operations in high-throughput systems, and they use a huge amount of such operations. Their optimization widely impacts the performance of the global system that uses them. In [11] there is an analysis of this operation frequency.

The problem of the optimization of multiplication by constant has been studied for a long time. For instance, the famous recoding presented by Booth in [3] can simplify the multiplication by constant operation as well as the full multiplication. This recoding and the algorithm proposed by Bernstein in [1] were widely used on processors without multiplication unit.

The main goal in this problem is the minimization of the computation quantity. The multiplication by constant problem seems to be simple, but its resolution is a hard problem due to its combinatorial properties. This problem can occur in more or less complex contexts. In the case of a single multiplication of one variable by one constant, it seems possible to explore the whole parameter space. But in the case of the multiplication of several variables by several constants, the space to explore is so huge that we have to use heuristics.

A first solution proposed to optimize multiplication by constant was the use of the constant recoding, such as Booth's. This solution just avoids long strings of consecutive ones in the binary representation of the constant. Better solutions are based on the factorization of common subexpressions, simulated annealing, tree exploration...

Our work deals with constant matrix multiplication (CMM), i.e. one useful form of the multiplication of several variables by several constants. A lot of applications involve such linear operations. This method is based on constants recoding followed by some dedicated common subexpression factorization algorithms. The proposed method was implemented in a VHDL generator. The generated results for several applications have been implemented on Xilinx FPGAs and compared to other solutions. Some significant improvements have been obtained: up to 40% area saving in the DCT case for instance.

This paper is organized as follows. The problem is presented in Section 2. In Section 3 some related works are presented. Our algorithms are presented in Section 4. The developed generator and the target architectures are discussed in Section 5. Finally, the results of the implementation of some applications and their comparison to other solutions are presented in Section 6.

2 Problem Definition

In this paper, and in the related works, the central problem is the substitution of full multipliers by an optimized sequence of shifts and additions and/or subtractions. We focus on integers but all the results can be extended to fixed-point representations.

All the values are represented using a standard radix-2 notation or two's complement unless it is specified. The notation $x \ll k$ denotes the k-bit left shift of the variable x (i.e. $x \times 2^k$). As we look at FPGA implementations, we assume that shift is just routing and addition and subtraction have the same area and speed cost.

As an example, let us compute p as the product of the input variable x by the constant $c = 111463 = 11011001101100111_2$. The simplest algorithm consists in using the distributiveness of multiplication. There is one addition of x (after some potential shift) for each one in the binary representation of c. In the case $c = 111463$, it leads to 10 additions:

$$\begin{aligned} 111463x &= x \ll 16 + x \ll 15 + x \ll 13 + x \ll 12 \\ &\quad + x \ll 9 + x \ll 8 + x \ll 6 + x \ll 5 \\ &\quad + x \ll 2 + x \ll 1 + x. \end{aligned}$$

The central point in this problem is the minimization of the total number of operations. It can be significantly reduced by using a recoding of the constant and/or subexpression elimination and sharing. The theoretical complexity of this problem seems to be still unknown.

Depending on the target application, this problem can occur at different levels of complexity. It starts with the multiplication of one constant by one variable. After, the multiple constant multiplication (MCM) problem appears with the multiplication of several constants by one variable [17]. In this present work, we deal with a more general version of this problem with the multiplication of one constant matrix by one variable vector: the constant matrix multiplication (CMM).

3 Related Works

There are, at least, four types of methods to address the multiplication by constant problem:

- Direct recoding methods.
- Evolutionary methods.
- Cost-function based search methods.
- Pattern search methods.

3.1 Direct Recoding Methods

In the radix-2 signed digit (SD) representation, the digits belong to the set $\{\overline{1} = -1, 0, 1\}$. A number is said to be in the canonical signed digit (CSD) format if no two non-zero digits are consecutive, then the number of non-zero digits is minimal. Using the CSD format on a n-bit value, the number of non-zero digits is bounded by $(n+1)/2$ and it tends asymptotically to an average value of $n/3 + 1/9$, as shown in [7]. For our example, using Booth's canonical recoding we have: $111463 = 11011001101100111_2 = 100\overline{1}0\overline{1}0100\overline{1}0\overline{1}0100\overline{1}_2$ and the product $p = c \times x$ reduces to 7 additions/subtractions:

$$\begin{aligned} 111463x &= x \ll 17 - x \ll 14 - x \ll 12 + x \ll 10 \\ &\quad - x \ll 7 - x \ll 5 + x \ll 3 - x. \end{aligned}$$

The KCM algorithm [5] was specifically designed for LUT-based FPGAs. It decomposes the binary representation of the variable into 4-bit chunks (a radix-16 representation). There is a more general version of this decomposition problem with distributed arithmetic. For instance, in [19] it was used on a DCT operator with an area saving of 17% compared to the direct implementation of the whole computation.

There are some recent works on the use of high-radix recoding. For instance, in [2], they use a radix-8 representation with punctured coefficients with the digits in the set $\{0, \pm1, \pm2, \pm4\}$ instead of the set $\{0, \pm1, \pm2, \pm3, \pm4\}$. This is a lossy representation, so they have to deal with some additional accuracy requirements. In our case, we want to study this problem for a lossless representation at first, but this approach seems to be interesting.

Another recoding solution was proposed with the use of multiple-radix representations and especially with the double-base number system (DBNS) [6]. In this solution, the authors use both radices 2 and 3 simultaneously, i.e. the values are expressed by $a = \sum_{i,j} a_{i,j} 2^i 3^j$ with $a_{i,j} \in \{0, 1\}$. This multiple-radix representation, sometimes useful in some analog circuits, does not seem to be efficient in the multiplication by constant problem in digital circuits.

3.2 Evolutionary Methods

Some evolutionary methods such as evolutionary graph generation [8] have been proposed to generate arithmetic circuits and especially for constant multipliers. These methods based on genetic algorithms seem to provide very bad results. For instance, in [8] the results are slightly better than the straightforward CSD encoding which is very far from the best known results. Furthermore, it seems that these methods are limited to the problem of multiplication

by one constant and have never been used to produce real circuits.

3.3 Cost-Function based Search Methods

The algorithm presented by Bernstein in [1] allows to reuse some intermediate values that are just used once in recoding methods. A more detailed and corrected version of this algorithm can be found in [4]. The algorithm based on a tree exploration defines three kinds of operations: $t_{i+1} = t_i \ll k$, $t_{i+1} = t_i \pm x$ and $t_{i+1} = t_i \ll k \pm t_i$. A cost can be specified for each operation according to the target technology. The cost function used to guide the exploration is the sum of the costs of all the implied operations. This algorithm only shares a few common sub-expressions. For our example $p = c \times x$, this algorithm gives a 5-addition solution:

$$\begin{aligned} t_1 &= ((x \ll 3 - x) \ll 2) - x, \\ t_2 &= t_1 \ll 7 + t_1, \\ p &= (((t_2 \ll 2) + x) \ll 3) - x. \end{aligned}$$

There are some other cost-function based search methods such as simulated annealing. In [14], this technique was used to produce multiplication by a small set of constants. The same multiplier is used for a small set of different coefficients. This problem is different from our's.

In [15] a greedy algorithm is used to determine a solution with a low total operation cost. A 28% average area saving is achieved. This solution seems to be limited due to local attraction of the greedy algorithm.

3.4 Pattern Search Methods

Most of the pattern search methods are based on the same general idea. The algorithm recursively builds a set of useful constants to be optimized. This set is initialized with the recoded initial constants. The different methods differ in the way they match the common sub-expressions and the way they reuse and share them.

The multiple constant multiplication (MCM) solution presented in [17] performs a tree exploration with selection of matching parts of the SD representation of the constants. This paper is the most cited one, and it presents a lot of details about the algorithm as well as about the comparisons.

In [9] the matches between constants are represented using a graph. The exploration of this graph is used to produce a specific form of FIR filters with a reduced number of adders.

A solution based on an algebraic formulation of the possible matches between constants is presented in [12]. Unfortunately, the authors use random filters for their tests without specifying the coefficients. So it is difficult to compare their results to the other solutions.

In [13] a factorization method based on the selection of the best pair of matching digits is used. This solution can be easily extended to the selection of common parts of words larger than two digits.

We will base our solution on extensions and improvements of the algorithms presented in [10] and [16]. A detailed description of this idea is presented below.

One can notice that among all the abundant bibliography about the multiplication by constant problem there is no general solution to the CMM problem.

4 Proposed Algorithms

4.1 Lefèvre's algorithm

In 2001, Lefèvre proposed a new algorithm to efficiently multiply a variable integer by a given set of integer constants [10]. As a special case, this algorithm can be used to multiply a variable by a single constant.

4.1.1 Definitions

The principle of the algorithm is to keep a list of constants to be optimized, and to find a "pattern" that appears several times in the set of constants. The constants are recoded using the CSD format in the very beginning. A pattern is a sequence of digits in $\{\overline{1}, 0, 1\}$. The number of non-zero digits in the pattern is called its weight.

A pattern P is said to occur in a constant C with a shift α when for each 1 in position k of P, there is a 1 in position $k + \alpha$ in C, and for each $\overline{1}$ in position k of P, there is a $\overline{1}$ in position $k + \alpha$ in C. And a pattern is said to occur negatively when there is a $\overline{1}$ in C for each 1 in P, and a 1 in C for each $\overline{1}$ in P. This last point is one of the main difference between the two papers [10] and [16], Lefèvre's algorithm allows to use more complex patterns and leads to slightly better optimizations.

When two occurrences of two patterns in the same constant match the same non-zero digit of the constant, the two occurrences are said to conflict. For example, in the number $51 = 10\overline{1}010\overline{1}_2$, the pattern $10\overline{1}$ occurs positively with shift 0, negatively with shift 2 and positively with shift 4. The first and third occurrences both conflict with the second one. And the pattern 10001 occurs negatively with shift 0 and positively with shift 2. Those occurrences are overlapping but not conflicting.

On our previous example $p = c \times x$, Lefèvre's algorithm gives a solution with only 4 additions:

$$\begin{aligned} t_1 &= (x \ll 3) - x, \\ t_2 &= (t_1 \ll 2) - x, \\ p &= (t_2 \ll 12) + (t_2 \ll 5) + t_1. \end{aligned}$$

4.1.2 Description of the algorithm

The principle of the algorithm can be described by the pseudo-code presented in Algorithm 1.

Algorithm 1 : Principle of Lefèvre's algorithm.

```
While ( there are some patterns with weight ≥ 2
        and at least 2 non-conflicting occurrences )
    choose a pattern with maximal weight and 2
        non-conflicting occurrences
    remove the chosen pattern for both occurrences
    add the pattern as a new constant
```

Then, multiplication by each constant in the final set can be implemented in the usual way: for each 1 ($\overline{1}$) in position p, add (subtract) x shifted by p bits to the left. And then, by rolling back the algorithm, each constant can be computed by shifts and additions/subtractions, with roughly one addition/subtraction for each chosen occurrence of a pattern.

4.2 Extensions and enhancements to Lefèvre's algorithm

Mathematically speaking, Lefèvre's algorithm deals with the multiplication of a number by a constant vector. The first thing to do is to extend it for the multiplication of a vector by a constant matrix. This extension is rather straightforward: we simply replace each constant with a constant vector. Patterns are then replaced by vectors of patterns and shifts are done component-wise. With this algorithm, it is possible to share all kinds of expression.

For example, let us consider the computation of $A = 5x + 5y + z$ and $B = 5x + 5y + 4z$. The algorithm will first share the computation of $5x + 5y$ between A and B. After, it will share $x + y$ in $(5x + 5y) = 4(x + y) + (x + y)$, effectively sharing the multiplication by 5 between x and y.

One point is kept unspecified in Lefèvre's algorithm: which maximal pattern and which occurrences to choose? In his implementation, Lefèvre simply chose the first maximal pattern he found, with the first two occurrences. This solution is probably not the best, so we tried to find something better.

The first idea was to find all the maximal-weight patterns with at least two non-conflicting occurrences, and all their occurrences. And then, we try to choose a set of patterns and a set of, at least two, occurrences such that two chosen occurrences (of the same pattern or of different patterns) do not conflict. The choice is performed in order to maximize the gain in the weight of all the constants; with a constant with i occurrences, we gain $i - 1$ times its weight. As all the chosen patterns have the same maximal weight, we want to maximize the sum, for each pattern, of the number of occurrences diminished by one.

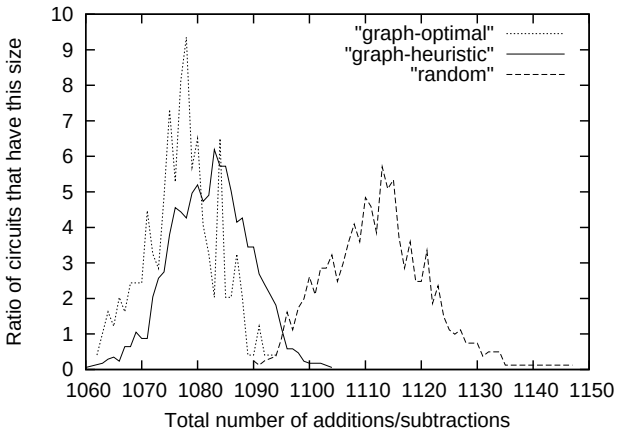

Figure 1. Heuristics Influence Distributions (from the optimization of many large 2D DCT operators).

We tried three different solutions for this. The first one, called "random", and which is the closest to the original algorithm, is to recursively choose, on random, a pattern with 2 non-conflicting occurrences, and to remove everything that conflicts with these occurrences. The second one, called "graph-heuristic", is to recursively choose a pattern with a maximal set of non-conflicting occurrences, and a minimal set of conflicts with the other patterns, and then remove everything that conflicts with these occurrences. And the third one, called "graph-optimal", is to build all the maximal sets of patterns and non-conflicting occurrences, and to choose the best one. This last solution can be very computationally intensive.

We tried to compare those three solutions, by running them several times for the same constant matrix (a huge 2D IDCT operator). The results in Figure 1 show that the "graph-optimal" and "graph-heuristic" are roughly equivalent, and better than the "random", with a tiny advantage to "graph-optimal". The time required to generate these results is less than one minute for "graph-heuristic" and "random", while it can grow to hours for "graph-optimal". Hence, we generally choose the "graph-heuristic" solution, so we can do lots of tries (thanks to its speed), and then choose the best solution.

4.3 Beyond the mathematical optimization

The improvements described above only deal with the minimization of the total number of additions and subtractions. Translated to hardware, this is not enough. Some additions and subtractions should be reordered without changing their total number, thanks to properties such as associativity and commutativity.

First of all, one may want to have a small circuit. When

three numbers a, b and c are added, the order in which they are added influences the size of the adders. For example, if a and b are narrow numbers, while c is wide, computing $(a + b) + c$ leads to a smaller circuit than $(a + c) + b$ or $(b + c) + a$. Hence, for space optimization, it is generally better to add the narrowest numbers first.

On the other hand, one may want to have a fast circuit. The order in which three numbers are added influences also the worst case delay of the circuit. For example, if a and c are available early while b is available late, the result is available earlier if we compute $(a + c) + b$ than $(a + b) + c$ or $(b+c)+a$. Hence, for speed optimization, it is preferable to add the earliest available values first.

Of course, those two kinds of optimization may often conflict. Hence, the user must specify if he or she prefers to optimize for speed or for area. Of course, if the user prefers to optimize speed, and if two solutions are equivalent for speed, the smallest one is chosen.

Moreover, the algorithmic optimization is not enough. We need to generate some real circuits. Hence, we decided to generate some VHDL code. Our VHDL code generator is currently targeted for Xilinx FPGAs. So additions and subtractions are performed on the dedicated carry-propagate adders and subtracters. The generator is able to produce VHDL code for fully parallel circuits, or for digit-serial circuits with radix 2^n for any n.

5 Implementation

Our implementation is mainly in two parts. The first part performs the mathematical optimization, with our extended and enhanced version of Lefèvre's algorithm. This part was written in C++, and is approximately 1500 line long. This part is not a program by itself, but a collection of simple classes that can be easily interfaced with any C or C++ program. Hence, it would be easy, for example, to interface this with a program that computes coefficients for FIR filters. Then the user could simply choose the type of filter and the pass and cut-off frequencies, and the program would compute the coefficients and generate some efficient VHDL code for it.

After the mathematical optimization, everything is implemented as plug-ins. Hence, there are, for example, plug-ins that optimize the order of the additions and subtractions, or plug-ins that generate the output VHDL code. This structure with plug-ins make the whole thing very modular. Hence, if someone wants, for example, to generate some Verilog code, or some assembly language code for a DSP, it is sufficient to write a new output plug-in. Then if someone wants to get pipelined circuits, he or she should write a new pipelining plug-in, and it can then be used with any output plug-in. Those plug-ins are also written in C++, but could be written in C or any language that can be interfaced with C. The collection of plug-ins is currently approximately 2500 line long.

The plug-ins communicate between themselves and the main program with simple interfaces that describe the circuit as a graph. In this representation, vertices represent mathematical values. Hence, there are vertices for input values, for output values, and also for intermediate values. Then there is an arc, from vertex x to vertex y, tagged with $(shift, sign, delay)$, if x shifted $shift$ bits to the left and delayed of $delay$ clock cycles is a positive or negative part (according to $sign$) of y. The $delay$ part will be useful for filters or for pipelined circuits. This representation has the quality of being independent from the desired output.

Let's give an simple example of how this can be used, and what is generated. As a simple example, we will consider building a constant multiplier by 111463. The corresponding source code and generated VHDL are presented Figure 2 and Figure 3 respectively.

```
#include "decompose.h"

int main(int argc, char **argv) {

  int value[] = {111463};

  Expr::initialize(argc, argv);

  CombinationSet<int> work(1);
  work.addLine(value);

  Key<int> *k;
  k = work.findBestPattern();
  while ((k!=NULL)&&(k->weight()>1)) {
    work.applyKey(*k, &cout);
    delete k;
    k = work.findBestPattern();
  }
  work.finish();
  return 0;
}
```

Figure 2. Multiplication by 111463 optimization source code.

6 Results and Comparisons

The synthesis have been performed using Xilinx ISE XST 4.2.03i tools for a Virtex XCV200-5 FPGA. The operators are not pipelined. The area is measured in number of slices (2 LUTs per slice in Virtex devices). The required area compared to the 2352 available slices in a XVC200 device is also reported in parentheses. The delay is expressed in nanoseconds. The number of additions/subtractions and the number of FA cells are computed by our generator (the two last lines of Figure 3); the number of FA cells is only an estimation .

Only a few papers give enough elements to compare our solutions. In [17] and [13], there are useful values for the

```
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.STD_LOGIC_arith.all;

entity MULT_111463 is
  port (
    x1  : in  std_logic_vector (7 downto 0);
    y1  : out std_logic_vector (24 downto 0)
  );
end MULT_111463;

architecture struct of MULT_111463 is
  signal t3     : std_logic_vector (22 downto 0);
  signal t2     : std_logic_vector (15 downto 0);
  signal t1     : std_logic_vector (22 downto 0);
begin -- struct
-- t3    = 20641*x1
  t3 <= ( t2(15)&t2(15)&t2(15 downto 0) & "00000" )
       +( t1(22 downto 0) );
-- t2    = 129*x1
  t2 <= ( x1(7)&x1(7)&x1(7)&x1(7)&x1(7)&x1(7)&x1(7)
          &x1(7)&x1(7 downto 0) )
       +( x1(7)&x1(7 downto 0) & "0000000" );
-- t1    = 16513*x1
  t1 <= ( t2(15)&t2(15)&t2(15)&t2(15)&t2(15)&t2(15)
          &t2(15)&t2(15 downto 0) )
       +( x1(7)&x1(7 downto 0) & "00000000000000" );
-- y1    = 111463*x1
  y1 <= ( t1(21 downto 0) & "000" )
       -( t3(22)&t3(22)&t3(22 downto 0) );
end struct;
-- Number of additions: 4
-- Estimated number of FA cells:  61
```

Figure 3. Multiplication by 111463 generated VHDL.

DCT application. Table 1 presents the number of additions/-subtractions for the 1D 8-point DCT for several word sizes. Our generator improves the best previous results from 17% up to 44%. Table 2 gives the synthesis results for the corresponding generated operators.

operator	initial	[17]	[13]	our
DCT 8b	300	94	73	56
DCT 12b	368	100	84	70
DCT 16b	521	129	114	89
DCT 24b	789	212	—	119

Table 1. Number of additions/subtractions comparison for some 1D 8-point DCT operators.

We performed some other comparisons on some error-correcting codes from [17] and [13]: the 8×8 Hadamard matrix transform, $(16, 11)$ Reed-Muller, $(15, 7)$ BCH and $(24, 12, 8)$ Golay codes. The comparison with the previous works in [17] and [13] is presented in Table 3 and the corresponding synthesis results are presented in Table 4. These results show that for very simple operators such as a small BCH code, some improvements are still possible. In the case of the 8×8 Hadamard matrix transform, we obtained the same results than the previous work [17], which results

operator	synthesis		generator	
	slices	delay	# $\pm$	# FA
DCT 8b	401 (17%)	19.5	56	739
DCT 12b	647 (27%)	21.7	70	1202
DCT 16b	1085 (46%)	25.7	89	2009
DCT 24b	2106 (89%)	27.9	119	3934

Table 2. Synthesis results of some 1D 8-point DCT generated operators.

may be optimal.

operator	initial	[17]	[13]	our
8×8 Had.	56	24	—	24
$(16, 11)$ R.-M.	61	43	31	31
$(15, 7)$ BCH	72	48	47	44
$(24, 12, 8)$ Golay	76	—	47	45

Table 3. Number of additions/subtractions comparison for some error-correction benchmarks.

operator	synthesis		generator	
	slices	delay	# $\pm$	# FA
8×8 Had.	128 (5%)	11.9	24	240
$(16, 11)$ R.-M.	39 (1%)	12.1	31	86
$(15, 7)$ BCH	461 (19%)	18.2	44	861
$(24, 12, 8)$ Golay	63 (2%)	12.2	45	136

Table 4. Synthesis results for some error-correction benchmarks.

Table 5 presents the synthesis results of the same operator with the three possible optimizations of our generator: none, area or speed. The operator is a 1D 8-point IDCT for 14-bit constants and 8-bit inputs. From the same initial addition/subtraction number the optimizations presented in Section 4.3 lead to significant improvements, 40% for the speed optimization for instance. The generation time for all these operators is around a few seconds on a standard desktop PC.

Some low-pass FIR filters used in [18] and [9] have been synthesized, these results are presented in Table 7. The specifications of the different filters are presented in Table 6. Their coefficients have been generated using the `remez` Matlab function: `remez`$(\#\text{tap}, [0 \;\; f_p \;\; f_s \;\; 1], [1 \;\; 1 \;\; 0 \;\; 0])$, and rounded to the target width. As the other authors do not specify the way they generate the coefficients, it is not possible to fairly compare our results. There are several function that use the Remez's algorithm in Matlab, and there are many options. Without more details it is not possible to

operator	synthesis		generator	
	slices	delay	# ±	# FA
IDCT	769 (32%)	30.2	81	1382
IDCT area	665 (28%)	19.8	81	1196
IDCT speed	666 (28%)	18.0	81	1241

Table 5. Influence of the generator optimizations on a 1D 8-point IDCT operator.

determine which sets of coefficients have been used in the previous papers.

Filter	f_p	f_s	#tap	width
FIR 1	0.15	0.25	60	14
FIR 2	0.15	0.20	60	16
FIR 3	0.10	0.15	60	14
FIR 4	0.10	0.12	100	18

Table 6. Low-pass FIR filters specifications.

operator	synthesis		generator	
	slices	delay	# ±	# FA
FIR 1	792 (33%)	29.1	86	1464
FIR 2	1075 (45%)	29.3	104	2021
FIR 3	792 (33%)	27.3	88	1509
FIR 4	1978 (84%)	34.9	173	3752

Table 7. Low-pass FIR filters synthesis results.

In Section 4.3, we said that our generator can produce digit-parallel as well as digit-serial circuits using different output plug-ins. Table 8 presents the synthesis results of a 1D 8-point IDCT operator for several solutions: digit-parallel and radix-2, 4, 8, 16, 64 and 256 digit-serial versions. Digit-serial implementations lead to small area and short cycle time operators. But in order to fairly compare digit-serial v.s. digit-parallel solution, we have to compare with pipelined parallel operator.

operator	slices	delay
parallel	614	40
serial radix-2	85	22
serial radix-4	153	36
serial radix-8	194	46
serial radix-16	242	47
serial radix-64	349	47
serial radix-256	446	48

Table 8. Synthesis results for 1D 8-point digit-parallel and digit-serial IDCT operators.

7 Conclusion

A new algorithm for the problem of multiplication by constant was presented. We generalized the previous results by dealing with the problem of the optimization of multiplication of one vector by one constant matrix. This algorithm is based on extensions and enhancements of previous algorithms [10] and [16]. Compared to the best previous results, our solution leads to a significant drop in the total number of additions/subtractions, up to 40%.

We implemented this algorithm in a VHDL generator. Based on a simple mathematical description of the computation, the generator produces an optimized VHDL code for Xilinx FPGAs. At the moment, the generated operators are non-pipelined parallel or digit-serial ones. We will extend our generator to produce pipelined circuits to reach higher clock frequencies.

We will also improve our algorithm in the case of operators that use different time samples of the same inputs such as filters. At the moment, the algorithm only factorizes some common sub-expressions without any temporal aspect.

We also want to extend our algorithm and generator to standard-cell based ASICs. The way to implement the adder/subtracter will widely impact the performance of the complete operator. The optimization required for low-power consumption may also change our solutions.

Another way to explore in the future, is the use of lossy representations such as [2]. In a lot of applications, the models include some approximations and the coefficients quantization. It may be a good idea to allow small perturbations of the coefficients.

Acknowledgments

The authors would like to thank the “Ministère Français de la Recherche” (grant # 1048 CDR 1 “ACI jeunes chercheurs”) and the Xilinx University Program for their support.

References

[1] R. Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7):641–652, July 1986.

[2] P. Boonyanant and S. Tantaratana. FIR filters with punctured radix-8 symmetric coefficients: Design and multiplier-free realizations. *Circuits Systems Signal Processing*, 21(4):345–367, 2002.

[3] A. D. Booth. A signed binary multiplication technique. *Quart. J. Mech. App. Math.*, IV(2):236–240, 1951.

[4] P. Briggs and T. Harvey. Multiplication by integer constants. Technical report, Rice University, 1994.

[5] K. D. Chapman. Fast integer multipliers fit in FPGAs. *EDN Magazine*, May 1994.

[6] V. S. Dimitrov, G. A. Jullien, and W. C. Miller. Theory and applications of the double-base number system. *IEEE Transactions on Computers*, 48(12):1098–1106, 1999.

[7] R. I. Hartley. Subexpression sharing in filters using canonic signed digit multipliers. *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, 43(10):677–688, Oct. 1996.

[8] N. Homma, T. Aoki, and T. Higuchi. Evolutionary graph generation system with transmigration capability and its application to arithmetic circuit synthesis. *IEE Proceedings*, 149(2):97–104, Apr. 2002.

[9] H.-J. Kang and I.-C. Park. FIR filter synthesis algorithms for minimizing the delay and the number of adders. *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, 48(8):770–777, Aug. 2001.

[10] V. Lefèvre. Multiplication par une constante. *Réseaux et Systèmes Répartis, Calculateurs Parallèles*, 13(4-5):465–484, 2001.

[11] D. J. Magenheimer, L. Peters, K. W. Pettis, and D. Zuras. Integer multiplication and division on the HP precision architecture. *IEEE Transactions on Computers*, 37(8):980–990, Aug. 1988.

[12] M. Martínez-Peiró, E. I. Boemo, and L. Wanhammar. Design of high-speed multiplierless filters using a nonrecursive signed common subexpression algorithm. *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, 49(3):196–203, Mar. 2002.

[13] A. Matsuura, M. Yukishita, and A. Nagoya. A hierarchical clustering method for the multiple constant multiplication problem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E80–A(10):1767–1773, Oct. 1997.

[14] M. F. Mellal and J.-M. Delosme. Multiplier optimization for small sets of coefficients. In *International Workshop Logic and Architecture Synthesis*, pages 13–22, Grenoble, France, Dec. 1997.

[15] H. T. Nguyen and A. Chatterjee. Number-splitting with shift-and-add decomposition for power and hardware optimization in linear DSP synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(4):419–424, Aug. 2000.

[16] R. Paško, P. Schaumont, V. Derudder, S. Vernalde, and D. Ďuračková. A new algorithm for elimination of common subexpressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):58–68, Jan. 1999.

[17] M. Potkonjak, M. B. Srivastava, and A. P. Chandrakasan. Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(2):151–165, Feb. 1996.

[18] H. Samueli. An improved search algorithm for the design of multiplierless FIR filters with power-of-two coefficients. *IEEE Transactions on Circuits and Systems*, 36(7):1044–1047, July 1989.

[19] S. Yu and E. E. Swartzlander. DCT implementation with distributed arithmetic. *IEEE Transactions on Computers*, 50(9):985–991, Sept. 2001.