EURASIP Journal on Advances in Signal Processing
a SpringerOpen Journal

RESEARCH Open Access

FIR filter optimization for video processing on FPGAs

Martin Kumm[1*], Diana Fanghänel[1], Konrad Möller[1], Peter Zipf[1] and Uwe Meyer-Baese[2]

Abstract

Two-dimensional finite impulse response (FIR) filters are an important component in many image and video processing systems. The processing of complex video applications in real time requires high computational power, which can be provided using field programmable gate arrays (FPGAs) due to their inherent parallelism. The most resource-intensive components in computing FIR filters are the multiplications of the folding operation. This work proposes two optimization techniques for high-speed implementations of the required multiplications with the least possible number of FPGA components. Both methods use integer linear programming formulations which can be optimally solved by standard solvers. In the first method, a formulation for the pipelined multiple constant multiplication problem is presented. In the second method, also multiplication structures based on look-up tables are taken into account. Due to the low coefficient word size in video processing filters of typically 8 to 12 bits, an optimal solution is found for most of the filters in the benchmark used. A complexity reduction of 8.5% for a Xilinx Virtex 6 FPGA could be achieved compared to state-of-the-art heuristics.

Introduction

Two-dimensional linear filters with finite impulse response (FIR) are one of the most fundamental operations used in image and video processing. They are used, e. g., in applications which contain contrast improvement, denoising, sharpening, target matching, and feature enhancement [1]. Compared to infinite impulse response filters, FIR filters have a strict stability, and high-throughput implementations are easily possible using pipelining as no recursions are involved. However, they are computationally expensive as many multiply accumulate (MAC) operations are necessary for each pixel of the resulting image. While this is very demanding for a microprocessor or digital signal processor, the inherent parallelism of field programmable gate arrays (FPGAs) can be used to accelerate the FIR operation.

Modern FPGAs directly incorporate embedded multipliers or DSP blocks which also include pre- and post-adders for MAC operations. Xilinx's DSP blocks (Xilinx Inc., San Jose, CA, USA) of Virtex 5/6, Spartan 6, and the 7 series FPGAs provide 18×25-bit signed multipliers. More flexible are the variable precision DSP blocks of the latest FPGAs of Altera, the Stratix V, Cyclone V, and Aria V devices (Altera, San Jose, CA, USA). Each DSP block can be configured as three independent 9×9-bit multipliers, two independent 16×16-bit, 15×17-bit, or 14×18-bit multipliers, or a single 18×36-bit or 27×27-bit multiplier.

However, embedded multipliers and DSP blocks are limited resources even on modern low-cost FPGAs, and they may have a higher power consumption compared to constant multiplication using the carry-chain resources [2]. Especially in image processing, embedded multipliers are often underutilized because of the small word lengths used. Typically, only 8 bits per color and 10 bits for a luminance representation are used.

In most filter applications, the coefficients are fixed, which can be used to reduce the complexity of the multiplication. Furthermore, partial results can be shared inside a single multiplier or between multipliers of different constants to reduce hardware resources. Different methods have been proposed over the years for such multiple constant multiplications (MCM):

(a) MCM using additions, subtractions, and bit shifts [3-33];
(b) MCM using look-up tables (LUTs) and adders [34,35];
(c) Distributed arithmetic (DA) [36-42].

*Correspondence: kumm@uni-kassel.de
[1] Digital Technology Group, University of Kassel, Kassel 34121, Germany
Full list of author information is available at the end of the article

Springer

© 2013 Kumm et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

In method (a), constant multiplications are realized using additions, subtractions, and bit shifts only. These operations form a so-called adder graph, so this method is called the adder graph MCM method in the following. It was originally developed for software or VLSI applications [3] but also maps well to the fast carry chains of FPGAs. In method (b), the input word is split into smaller chunks that fit into the input word size of the FPGA LUTs. These LUT results are shifted and added afterward to form the multiplication result. LUTs and adders are also used in method (c), but there the folding equation of the FIR filter is rearranged in such a way that identical LUTs can be used. This is very beneficial in sequential FIR implementations but costly in parallel implementations. Because it was shown in the recent years that multiplier blocks using add, subtract, and shift operations (method a) consume considerable less logic resources compared to parallel DA implementations [5-7], the DA approach is not further considered. Due to the relatively large routing delays compared to the fast carry chain, a pipelined implementation of the adder graph is necessary to obtain the maximum speed of the FPGA [2,5-10]. It was shown by Faust et al. [35] that the LUT-based approach (method b) is competitive to the adder graph method. Thus, pipelined circuits using the combination of methods (a) and (b) are investigated in this paper.

Contribution of this work

The main contribution of this article is the description of two novel optimization methods, one for the adder graph MCM problem including pipelining (the pipelined MCM problem [9]) and one for a combination of this method with a pipelined realization of the LUT-based method mentioned above. Each method is formulated as a boolean integer linear program (BILP, or 0-1 ILP) and then reduced to a mixed integer linear program (MILP). Hence, if the MILP solver finds a solution in reasonable time, an optimal solution for the given cost model is found. To the best of our knowledge, this is the first time an optimal method for solving the pipelined MCM (PMCM) problem is proposed.

The complexity of the adder graph MCM method heavily depends on the coefficient values, while the complexity of the LUT-based approach mainly depends on the input word size. Therefore, sometimes, one method or the other delivers better results. For this, a combination of both methods is proposed in this work by incorporating the LUT-based multipliers in the integer linear programming (ILP) formulation of the PMCM problem. Due to the low coefficient word size of typically 8 to 12 bits in image processing, a short convergence time of the ILP solver is very likely, which makes the proposed optimization an ideal candidate for image processing.

The remaining of this paper is organized as follows. The related work is discussed in the next section, followed by an introduction of the used FIR architectures for image processing. Then, an ILP formulation for the PMCM problem is described which is later extended for additional LUT-based multiplication. Finally, results from the optimizations and FPGA synthesis are presented and discussed, followed by a conclusion.

Related work

MCM using additions, subtractions, and bit shifts

Different methods have been proposed over the years to realize constant multiplication using additions, subtractions, and bit shifts only. Finding the optimal configuration of these operations is known as MCM problem, which has been an active research topic for almost the last two decades [3-33]. The objective is usually defined by minimizing the number of adders and subtractors (shifts are assumed to be free, as they can be implemented using wires).

An example adder graph which realizes a multiplier block with the constants of the set $\{44, 130, 172\}$ is shown in Figure 1a. Each node in the graph corresponds to an adder or subtractor, indicated by '+' or '−'. The numeric node value represents the realized multiple of the input, i. e., node '1' corresponds to the input of the MCM block. To have a unique representation, all node values are defined to be odd. They can be made even by a simple bit shift as shown at the output. All edge weights represent left shifts, e. g., node '5' is realized by left shifting the input x by 2 bits and adding the unshifted input: $2^2x + 2^0x = 5x$. Right shifts are represented by negative edge weights.

The MCM problem is NP complete [4]. Hence, most of the proposed algorithms are heuristics, and less work was directed toward optimal solutions. Early work was done by Bull and Horrocks [3] which was later extended by Dempster and Macleod to the modified Bull and Horrocks algorithm [14]. In the same work, the n-dimensional reduced adder graph (RAG-n) algorithm was proposed which was one of the leading heuristics for years. Major improvements could be achieved by the work of Voronenko and Püschel with their cumulative benefit heuristic (H_{cub}) [4]. By spending a bit more algorithmic complexity and evaluating adder graph topologies up to a depth of three, they could reduce the required additions/subtractions by 20% on average compared to RAG-n. A competing approach based on difference graphs was proposed by Gustafsson [16]. It tends to be beneficial compared to H_{cub} in case large coefficient sets and/or low coefficient word lengths are used but may be worse in other cases.

Many approaches use ILP formulations, for which optimal solvers exist. However, the search space is often

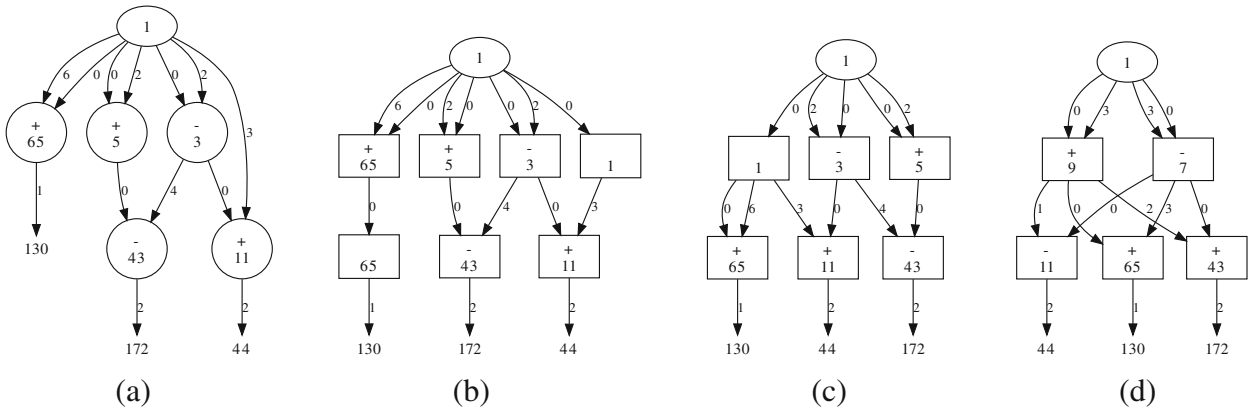

Figure 1 Different multiplier block realizations of the coefficients {44,130,172}. (**a**) Adder graph obtained by H_{cub} AD min, (**b**) PAG using ASAP pipelining, (**c**) PAG using optimal pipelining, and (**d**) optimal PAG.

significantly reduced due to the used number representation which leads to non-optimal results. Minimum signed digit (MSD) number systems like canonic signed digit (CSD) are often used as they have a reduced complexity compared to the binary representation [18]. In MSD, a number is coded using the digits $\{-1, 0, 1\}$, such that the number of non-zero digits is minimal and, hence, the number of partial products is reduced. A 0-1 ILP model that uses subexpressions of length two in the CSD number system (i. e., subexpressions with at most two non-zeros) was used by Yurdakul and Dündar [19]. A 0-1 ILP model which can be used for any number system was proposed by Flores et al. [18]. Results for binary, CSD and MSD were presented. This work was further extended by Aksoy et al. with additional delay constraints [20], low-level area models [21], and a heuristic variation [22].

So far, the discussed publications did not result in globally optimal solutions as they use the reduced search space of a given number representation. Breadth-first search [23] and depth-first search [17] algorithms were proposed by Aksoy et al. to optimally solve the MCM problem in a graph-based way. The depth-first search is able to solve MCM instances in a reasonable time but cannot handle different constraints or cost metrics. Another interesting method to optimally solve the MCM problem was given by Gustafsson who transferred the MCM problem to the problem of finding a Steiner hypertree in a directed hypergraph [24]. He used an optimal 0-1 ILP formulation which is very generic and can be flexibly adopted to different cost metrics (at adder or logic level) and different constraints (adder depth and fan-out). The main drawback is its computational complexity. Nevertheless, it can be used for small MCM instances or to find lower bounds [25] by relaxing the model to a continuous LP problem. A 0-1 ILP formulation for optimally solving the special case of minimum depth adder graphs in a graph-based fashion was recently proposed [26]. In this work, the search space of a graph-based search is compared to the MSD search space in terms of variables of the ILP. The graph-based search needs three times more variables for 8-bit coefficients and 18 times more variables for 13-bit coefficients compared to MSD.

In the recent time, more and more MCM algorithms with different objectives were proposed. One objective is minimizing the power of the adder graph by reducing or minimizing the adder depth (AD) of each output, which is defined as the number of adder stages needed to compute a coefficient [26-29]. Limiting the maximum AD of *all* outputs can be used to find adder graphs with low delay [30]. If this delay is still too large, pipelining can be used to speed up the circuit [5,8,9,31].

Pipelining plays a crucial role in high-performance adder graph realizations on FPGAs as they have relatively large routing delays compared to their fast carry chains. However, the addition of pipeline registers may significantly increase the complexity. An example of the pipelined adder graph (PAG) of Figure 1a using an as-soon-as-possible (ASAP) scheduling for placing the pipeline registers is shown in Figure 1b. Here, each rectangular node includes a pipeline register, i. e., nodes without any '+' or '−' operator are pure registers. Many resources can be saved by finding the optimal schedule of pipeline registers for a given adder graph. Compared to the ASAP schedule, 29% less pipelined operations and speedups of about 300% were achieved on average using a slice overhead of only 18% compared to the non-pipelined adder graph [8]. The PAG using the optimal schedule of the adder graph of Figure 1a is shown in Figure 1c. However, directly considering pipelining during the adder graph optimization can further reduce the resources as demonstrated in Figure 1d. Heuristics for this kind of direct optimization were proposed recently [7,9]. A reduction

of pipelined operations by 10% compared to the optimal pipelined adder graphs [8] could be achieved by the reduced pipelined adder graph (RPAG) algorithm [9]. Slice reductions of 16% on average were reported using the best result out of three algorithms (C1$^+$, RSG Improved and a genetic algorithm) [7].

MCM using look-up tables

A totally different method for single constant multiplication which uses the look-up tables of FPGAs was proposed by Wirthlin [34]. The idea is to split the multiplication into smaller chunks, e. g., 4 bits for FPGAs with four-input LUTs, which can be directly realized using a single stage of LUTs. These LUT results have to be shifted and added to get the final product. Several techniques were proposed to minimize the number of redundant LUTs [34]. An extension of LUT-based multipliers to MCM was presented by Faust et al. [35], where identical LUTs are also shared between different constant multipliers. It was shown that the maximal number of required LUTs is far less than combinatorially possible. Their benchmark results include a comparison with adder graph MCM, which shows that their graph-based MinLD MCM algorithm [29] sometimes uses less resources and sometimes more resources than the LUT-based method for an input word size of 8 bits on an FPGA with four-input LUTs. As their method does not include pipelining, shorter delays could be achieved using the LUT-based MCM.

Two-dimensional FIR filter architectures

An image of height M and width N is usually represented by an $M \times N$ matrix $\mathbf{X}$. The matrix elements are written in lower case in the following, i. e., $x_{m,n}$ denotes the luminance of a pixel at position (m, n). The two-dimensional folding with the $P \times Q$ folding matrix $\mathbf{H}$ is defined as

$$y_{m,n} = \sum_{p=0}^{P-1} \sum_{q=0}^{Q-1} x_{m+p-u,n+q-v} \, h_{p,q}, \tag{1}$$

where $u = \left\lfloor \frac{P}{2} \right\rfloor$ and $v = \left\lfloor \frac{Q}{2} \right\rfloor$ denote the center of the folding matrix $\mathbf{H}$. Note that P and Q are often identical (matrix $\mathbf{X}$ is square) and odd.

Architectures for computing this folding equation can be classified by the computed output pixels per clock cycle. A sequential realization may compute a single MAC operation per clock cycle producing one output pixel every PQ clock cycles, like this is done on conventional microprocessors. As we are interested in accelerating the filter operation, parallel realizations are considered in the following. A direct implementation of the folding equation for a 3×3 folding matrix in a parallel way which is able to compute one output pixel every clock cycle is shown in Figure 2a. First, the input pixel stream is clocked into a chain of registers and FIFO buffers to provide a 3×3 block of the image. Then, this block is processed in parallel by a sum-of-products (SOP) operation according to (1), i. e., each pixel of the block is multiplied by its corresponding element in the folding matrix and all products are added to yield the final output pixel. An alternative architecture can be obtained by transposing the structure of Figure 2a by reversing the directions of each edge, replacing branches by adders and vice versa, and swapping the input and output [43]. The resulting architecture is shown in Figure 2b. The SOP block results in an MCM block after transposition, i. e., now a single input has to be multiplied by all elements of the folding matrix. Note that the outputs of both architectures are invalid for the boundary pixels of the input image. Thus, if the output image has to have the same size as the input image, a suitable boundary treatment (setting boundary pixels to black/white, copying from neighbor pixels, etc.) has to be implemented [43].

MCM blocks of Figure 2b can be obtained by one of the algorithms discussed above. The SOP circuit of Figure 2a can be either obtained by using LUT multipliers and additional adders, or by transposing the adder graph which was obtained by an adder graph MCM method. It is well known that the adder cost for a single-input single-output adder graph is equal to its transposed form [44]. More general, Gustafsson et al. derived the cost of an adder graph with N_i inputs and N_o outputs after transposition for the generalized case of vector matrix multiplication [32]:

$$N_A^T = N_A + N_o - N_i \tag{2}$$

Here, N_A and N_A^T are the adder cost before and after transposition, respectively. MCM is a special case of vector matrix multiplication with $N_i = 1$, so if $N_{A,\mathrm{MCM}}$ adders are needed for an optimal MCM adder graph with N unique coefficients, $N_{A,\mathrm{SOP}} = N_{A,\mathrm{MCM}} + N - 1$ adders are needed for the corresponding optimal SOP. Therefore, the eight additional adders shown in the transposed form in Figure 2b are exactly the additional adders needed to compute the SOP in Figure 2a. Hence, from its complexity, there is no difference between the direct or transposed form. If we take a look on pipelined implementations of MCM blocks, the situation becomes worse. While transposing a pipelined MCM block still leads to a valid pipeline, the pipeline registers may not any longer be located in the critical path between the adders. Therefore, we concentrate in the following on pipelined implementations of the MCM block, which can be directly incorporated in the transposed form, as shown in Figure 2b.

A totally different filter structure can be realized if the folding matrix is separable, i. e., matrix $\mathbf{H}$ can be separated in two vectors h_1 and h_2 with $\mathbf{H} = h_1 \cdot h_2$. Then, the two-dimensional filter can be realized by cascading

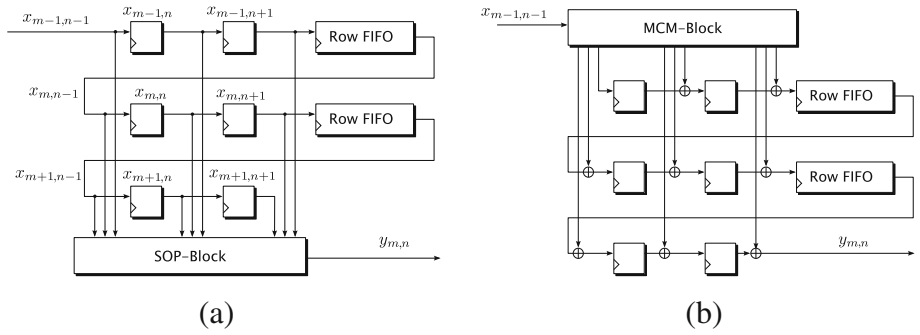

Figure 2 Architectures of a two-dimensional FIR filter. (**a**) Direct form using SOP, (**b**) transposed form using MCM.

two one-dimensional filters, one for processing the rows and one for the columns. This reduces the PQ multiplications to $P + Q$ multiplications [43]. If the filter cannot be separated, then there exist methods to decompose the filter into a sum of separable filters using singular value decomposition [45].

However, only a fraction (which is typically less than one third) of the multipliers of the unseparated folding matrix have to be realized due to symmetric, zero, or power-of-two coefficients. Furthermore, in both MCM methods discussed above, many resources can be saved when intermediate computation results are shared between the constant multipliers. This is only possible if all multipliers have the same input which is not the case for separated filters. Therefore, we concentrate on the architecture shown in Figure 2b as it is generic (any folding matrix can be realized) and pipelined MCM blocks can be directly used.

Optimally solving the pipelined MCM problem

Pipelined MCM problem formulation

In an adder graph, each node unequal one is generated by a so called $\mathcal{A}$-operation, introduced by Voronenko et al. [4] (compare with Figure 1a)

$$\mathcal{A}_q(u,v) = |2^{l_1}u + (-1)^{sg}2^{l_2}v|2^{-r} \tag{3}$$

with configuration $q = (l_1, l_2, r, sg)$, where $l_1, l_2, r \in \mathbb{N}_0$ are shift factors, the sign bit $sg \in \{0,1\}$ denotes whether an addition or subtraction is performed and u and v are positive, odd input arguments. A *valid configuration* q is a combination of l_1, l_2, r, and sg such that the result is a positive odd integer.

The greatest effort during MCM optimization is finding the numerical values of non-output nodes, i. e., the values of all u and v which are not in the coefficient set. Once all these node values are found, it is easy to determine the configuration q of the corresponding adder graph, e. g., using the optimal part of RAG-n [14] or H_{cub} [4]. Since the same can be applied for PAG optimization, it is appropriate to define a set X_s for each pipeline stage s, containing the node values of the corresponding stage. The pipeline sets for the PAG in Figure 1d are, e. g., $X_0 = \{1\}$, $X_1 = \{7, 9\}$, and $X_2 = \{11, 43, 65\}$. With this representation, we can formally define the PMCM problem:

Definition 1 (Pipelined MCM problem)**.** *Given a set of positive target constants $T = \{t_1, \ldots, t_M\}$ and the number of pipeline stages S, find sets $X_1, \ldots, X_{S-1} \subseteq \{1, 3, \ldots, x_{max}\}$ with minimal area cost such that for all $w \in X_s$ for $s = 1, \ldots, S$ there exists a valid $\mathcal{A}$-configuration q such that $w = \mathcal{A}_q(u, v)$ with $u, v \in X_{s-1}$, $X_0 = \{1\}$ and $X_S = \{\text{odd}(t) \mid t \in T \setminus \{0\}\}$, where* odd($t$) *is the absolute value of t divided by 2 until it is odd.*

The upper bound is usually chosen as $x_{\max} = 2^{b_{\max}+1}$ [4], where $b_{\max}$ is equal to the maximum bit width of T. The number of stages S has to be at least the largest minimal adder depth of the coefficients. The minimal AD of an integer x can be directly computed using

$$\text{AD}_{\min}(x) = \lceil \log_2(\text{nz}(x)) \rceil, \tag{4}$$

where nz(x) represents the minimal number of non-zero digits of x in canonic signed digit (CSD) representation [29]. As each additional pipeline stage introduces additional nodes in the PAG, it is very unlikely that there exists a graph with higher S but less cost. Therefore, we define S to be the minimum number of stages which is possible:

$$S := S_{\min} = \max_{t \in T} \text{AD}_{\min}(t) \tag{5}$$

The area cost of pipeline sets $X_{0 \ldots S}$ depends on the target architecture, and it will be discussed in the following sections.

FPGA cost of pipelined adder graphs

We use the area (i. e., the number of FPGA components) as cost measure, since the maximum speed results from the architecture and is given by the largest ripple carry adder and the routing delay which is hard to predict. Different cost metrics are used for evaluating the size of an FPGA circuit. Common metrics are counting the LUTs, flip-flops, slices (Xilinx devices), or logic elements (Altera

devices). As different FPGA components are used in this work, the smallest common piece of logic, consisting of LUT(s), FF(s), and full adder logic, is counted and referred to as basic logic element (BLE) in the following. Simplified block schematics of BLEs of the Virtex 4 and Virtex 6/7 architectures are shown in Figure 3. On Virtex 4, each BLE is half of an FPGA slice, and on Virtex 6/7, it is a quarter of a slice. While the Virtex 4 BLE provides a four-input LUT, a flip-flop and additional logic for building a full adder (AND gate, XOR gate, and a multiplexer), the BLE from the later generations provide a six-input LUT which can be used as two five-input LUTs with shared inputs, two flip-flops, and full adder logic (XOR gate and multiplexer).

Each BLE can realize a single full adder including the register for pipelining. Thus, for evaluating adder graphs, the number of full adders, or pure registers can be counted. A detailed cost model of common adder graphs mapped to FPGAs was recently presented in [33]. This model respects the replacement of full adders by simple wires if the two shifted numbers do not overlap at every bit position. However, for pipelined adders on FPGAs, full adder and pipeline register are realized in a single BLE resource. Hence, saving some full adders has no effect on BLE resources as the pipeline register is needed in any case. The number of BLEs of an $\mathcal{A}$-operation is equal to the output word size of the corresponding adder. If there is no right shift ($r = 0$), the output word size is identical to the sum of input word size and the word size needed to represent node value w. If a right shift $r > 0$ is used, the word size of the corresponding adder has r additional output bits. Even if these additional output bits are ignored after the right shift, they are needed to compute the carry-in for higher order bits to yield a correct result. Hence, the number of BLEs of each $\mathcal{A}$-operation in a pipelined adder graph can be summarized to

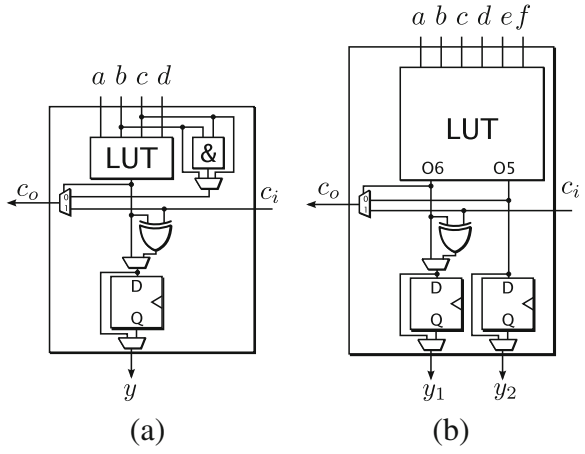

Figure 3 Simplified basic logic elements of Xilinx's Virtex families. (**a**) Virtex 4 BLE, (**b**) Virtex 6/7 BLE.

$$\text{cost}_{\mathcal{A}}(u, v, w) = B_x + \lceil \log_2(w) \rceil + r, \tag{6}$$

where B_x is the input word size and $w = \mathcal{A}_q(u, v)$.

ILP formulation for the pipelined adder graph problem

Before the ILP formulation for the PAG problem is described, some variables and auxiliary sets are introduced. There are two types of binary decision variables used in the formulation:

$$a_w^s = \begin{cases} 1 & \text{if } w \text{ is realized in pipeline stage } s \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

$$a_{(u,v,w)}^s = \begin{cases} 1 & \text{if } \mathcal{A}-\text{operation } w = \mathcal{A}_q(u, v) \\ & \text{is computed in pipeline stage } s \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Hence, there is one variable a_w^s for each stage s and each possible element w and one variable $a_{(u,v,w)}^s$ for each w and each possible pair (u,v), from which w can be computed with a single $\mathcal{A}$-operation.

To determine these combinations, we use the definition of the $\mathcal{A}_*(u, v)$ set [4], which contains all elements that can be computed from u and v using a single $\mathcal{A}$-operation:

$$\mathcal{A}_*(u, v) := \{\mathcal{A}_q(u, v) \mid q \text{ valid configuration}\} \tag{9}$$

For convenience, the $\mathcal{A}_*$ set is also defined for a set X:

$$\mathcal{A}_*(X) := \bigcup_{u,v \in X} \mathcal{A}_*(u, v) \tag{10}$$

Note that $\mathcal{A}_*(\{u, v\})$ is different from $\mathcal{A}_*(u, v)$ as the first one contains all combinations of u and v: $\mathcal{A}_*(\{u, v\}) = \mathcal{A}_*(u, u) \cup \mathcal{A}_*(u, v) \cup \mathcal{A}_*(v, v)$. With this set, we can now define the set $\mathcal{A}^s$ of all possible odd variables which may be used at pipeline stage s. It can be recursively defined by

$$\mathcal{A}^0 = \{1\} \tag{11}$$

$$\mathcal{A}^s = \mathcal{A}_*(\mathcal{A}^{s-1}) \ . \tag{12}$$

As not all elements of $\mathcal{A}^s$ are needed to compute the target coefficients, the set $\mathcal{S}^s \subseteq \mathcal{A}^s$ is defined which contains all *single* elements that may be used to compute the target coefficients T in the last stage. Furthermore, with $\mathcal{T}^s$ we denote the set of (u, v, w) *triplets* for which $w \in \mathcal{S}^s$ can be computed using u and v of the previous stage $s - 1$. Now, the sets $\mathcal{S}^s$ and $\mathcal{T}^s$ can be computed recursively, starting from the last stage S, where $\mathcal{S}^S$ is identical to the odd target coefficients excluding zero:

$$\mathcal{S}^S := \{\text{odd}(t) \mid t \in T \setminus \{0\}\} \tag{13}$$

$$\mathcal{T}^{s-1} := \{(u, v, w) : w = \mathcal{A}_q(u, v), u, v \in \mathcal{A}^{s-1}, u \leq v, w \in \mathcal{S}^s\} \tag{14}$$

$$\mathcal{S}^{s-1} := \{u, v : (u, v, w) \in \mathcal{T}^{s-1}\}. \tag{15}$$

Using these variables and auxiliary sets, we can define the ILP formulation for the PAG problem as follows:

ILP Formulation 1 : Pipelined adder graph optimization

$$\text{minimize} \quad \sum_{s=1}^{S} \sum_{(u,v,w)\in\mathcal{T}^s} \text{cost}_{\mathcal{A}}(u,v,w)\, a^s_{(u,v,w)}$$

subject to

$$\text{(C1)} \quad a^S_w = 1 \text{ for all } w \in \mathcal{S}^S$$

$$\text{(C2)} \quad a^s_w - \sum_{\forall (i,j,k)\in\mathcal{T}^s | k=w} a^s_{(i,j,k)} \leq 0 \text{ for all } w \in \mathcal{S}^s, s = 2\ldots S$$

$$\text{(C3)} \quad \left.\begin{array}{l} a^s_{(u,v,w)} - a^{s-1}_u \leq 0 \\ a^s_{(u,v,w)} - a^{s-1}_v \leq 0 \end{array}\right\} \text{for all } (u,v,w) \in \mathcal{T}^s, s = 2\ldots S$$

$$a^s_w \in \{0,1\},\ a^s_{(u,v,w)} \geq 0$$

The subject is to minimize the BLE cost of all required $\mathcal{A}$-operations. Constraint (C1) simply forces that all target coefficients are realized in the last pipeline stage. To realize an element a^s_w, one of the possible $\mathcal{A}$-operations to compute w must be available, which is constrained by (C2). The last constraint (C3) ensures that an $\mathcal{A}$-operation $w = \mathcal{A}_q(u,v)$ can only be realized in stage s when u and v are both available in stage $s-1$.

In the formulation, we use the relaxed condition $a^s_{(u,v,w)} \geq 0$ instead of requiring $a^s_{(u,v,w)} \in \{0,1\}$ which results in an MILP formulation. Using continuous variables $a^s_{(u,v,w)}$ instead of binary ones reduces the runtime of the optimization substantially. This is possible since we can construct a binary solution that is feasible and minimal from the relaxed solution.

Proof. Assume an optimal solution of ILP Formulation 1 is given. Due to (C3), all variables $a^s_{(u,v,w)}$ satisfy the condition

$$0 \leq a^s_{(u,v,w)} \leq 1\,, \tag{16}$$

but in the optimum solution of ILP Formulation 1, a variable $a^s_w = 1$ can exist with several variables $a^s_{(u,v,w)} > 0$. Let us define a set $\mathcal{T}'_w$, which contains all triplets (u,v,w) with $a^s_{(u,v,w)} > 0$. Then, there exist at least one triplet $(u',v',w) \in \mathcal{T}'_w$ with the least cost contribution:

$$(u',v',w) = \arg \min_{(u,v,w)\in\mathcal{T}'_w} \text{cost}_{\mathcal{A}}(u,v,w). \tag{17}$$

Now, a feasible binary solution of the formulation can be obtained by assigning new values $\hat{a}^s_{(u',v',w)} = 1$ and $\hat{a}^s_{(u,v,w)} = 0$ for all $(u,v,w) \in \mathcal{T}'_w \setminus \{(u',v',w)\}$. Since the contribution of the triplets in $\mathcal{T}'_w$ to the objective function is

$$\sum_{(u,v,w)\in\mathcal{T}'_w} \underbrace{\text{cost}_{\mathcal{A}}(u,v,w)}_{\geq \text{cost}_{\mathcal{A}}(u',v',w)} a^s_{(u,v,w)} \tag{18}$$

$$\geq \text{cost}_{\mathcal{A}}(u',v',w) \underbrace{\sum_{(u,v,w)\in\mathcal{T}'_w} a^s_{(u,v,w)}}_{\geq 1 \text{ because of (C2)}}$$

$$\geq \text{cost}_{\mathcal{A}}(u',v',w) \tag{19}$$

$$= \sum_{(u,v,w)\in\mathcal{T}'_w} \text{cost}_{\mathcal{A}}(u,v,w)\, \hat{a}^s_{(u,v,w)} \tag{20}$$

and since the objective function is minimized, the new solution is also minimal and the objective values are equal. If we do this construction for all variables $a^s_w = 1$, we obtain an optimum solution where all variables are binary. □

ILP Formulation 1 is very generic and is extendable into different directions. For optimizing MCM blocks on application specific integrated circuits (ASICs), the cost function in the objective can be easily divided into two cost functions: one for pipelined adders ($\text{cost}_{\mathcal{A}}$) and one for pure registers (cost_{R}) as both require different ASIC resources:

$$\text{cost}(u,v,w) = \begin{cases} \text{cost}_{\text{R}}(w) & \text{for } w = u \vee w = v \\ \text{cost}_{\mathcal{A}}(u,v,w) & \text{otherwise} \end{cases} \tag{21}$$

A modified cost function can also be used to influence the number of adders within a pipeline stage. Instead of placing pipeline registers after each stage of adders, they could be placed behind multiple adder stages (e.g., every second or third stage). This would reduce the register resources for applications where a lower speed is sufficient or a lower latency is required. For that, the cost function for pure registers (as for ASICs) can be set to zero in each stage in which no registers should be placed. In the implementation, these registers are exchanged by wires.

Another extension of the ILP formulation would be the support of dedicated shift registers which are provided by the latest Xilinx FPGA architectures. These shift registers are realized in a single FPGA LUT and provides up to 16 bits on Virtex 4 FPGAs (SRL16 primitive) and up to 32 bits on Virtex 5/6, Spartan 6 and 7 series FPGAs (SRLC32E primitive). In other words, up to 16 or 32 registers (plus one additional register at the output) can be realized at the same BLE cost like a single register. As long as $S < 17$ or $S < 33$, constraint (C3) in ILP Formulation 1 can be replaced by:

$$\text{(C3a)} \quad \left.\begin{array}{l} a^s_{(u,v,w)} - a^{s-1}_u \leq 0 \\ a^s_{(u,v,w)} - a^{s-1}_v \leq 0 \end{array}\right\} \begin{array}{l} \text{for all } (u,v,w) \in \mathcal{T}^s, \\ s = 2 \ldots S, w \neq u, w \neq v \end{array} \tag{22}$$

$$\text{(C3b)} \quad a^s_{(u,v,w)} - \sum_{s'=0}^{s-1} a^{s'}_u \leq 0 \text{ for all } (u,v,w) \in \mathcal{T}^s, s = 2 \ldots S, w = v \vee w = u. \tag{23}$$

While (C3a) is nearly identical to (C3), (C3b) allows the bypass of several stages using shift registers. Of course, these values are not accessible for intermediate stages.

Multiple constant multiplication using LUT multipliers

In this section, a LUT-based constant multiplier method is introduced, which is used to extend ILP Formulation 1.

LUT-based constant multiplication

Consider a number x which is represented in two's complement format using B_x bits:

$$x = -2^{B_x-1} x_{B_x-1} + \sum_{b=0}^{B_x-2} 2^b x_b \tag{24}$$

If this number is multiplied by a constant c_n with B_c bits, the resulting $B_c \times B_x$ multiplication can be divided into several smaller multiplications by rearranging the partial sum terms:

$$\begin{aligned} \underbrace{c_n \cdot x}_{B_c \times B_x \text{Mult.}} &= c_n \left(\sum_{b=0}^{B_x-2} 2^b x_b - 2^{B_x-1} x_{B_x-1} \right) \\ &= c_n \sum_{b=0}^{L-1} 2^b x_b + c_n \sum_{b=L}^{2L-1} 2^b x_b + \ldots \\ &\quad + c_n \left(\sum_{b=(K-1)L}^{KL-2} 2^b x_b - 2^{KL-1} x_{KL-1} \right) \\ &= \underbrace{c_n \sum_{b=0}^{L-1} 2^b x_b}_{B_c \times L \text{Mult.}} + 2^L \underbrace{c_n \sum_{b=0}^{L-1} 2^b x_{b+L}}_{B_c \times L \text{Mult.}} + \ldots \\ &\quad + \underbrace{2^{(K-1)L} c_n \left(\sum_{b=0}^{L-2} 2^b x_{b+(K-1)L} - 2^{L-1} x_{KL-1} \right)}_{B_c \times L \text{Mult.}} \end{aligned} \tag{25}$$

Now, $K = \left\lceil \frac{B_x}{L} \right\rceil$ multiplications of size $B_c \times L$ are necessary. In case that B_x is not divisible by L, the number x has to be sign extended to $B'_x = KL$ bits. The idea of Wirthlin [34] was to realize each $B_c \times L$ multiplier using L-input LUTs, where L has to be chosen to fit the FPGA LUT input size. The resulting multiplier structure for an example of a 12×4 multiplier using four-input LUTs [34] is shown in Figure 4.

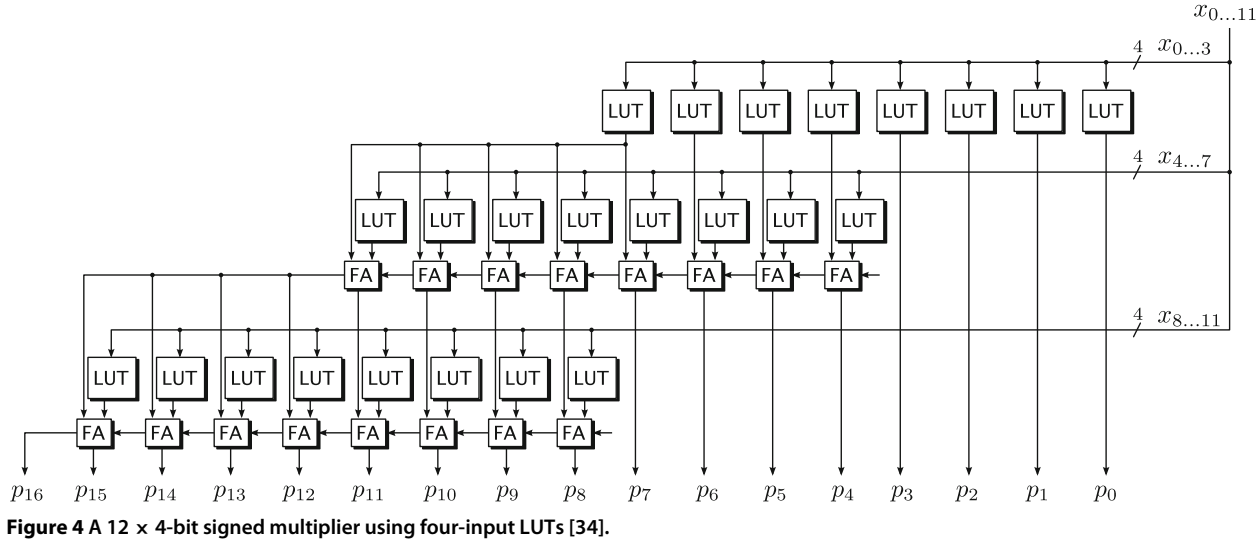

Figure 4 A 12 × 4-bit signed multiplier using four-input LUTs [34].

LUT minimization techniques

Several methods were proposed by Wirthlin to reduce the hardware complexity of the circuit. In the following cases, LUTs can be eliminated:

(a) Removal of constant LUTs, i. e., LUTs that are always '0' or '1' can be replaced by a constant.
(b) Removal of LUTs which are identical to one of the inputs, i. e., the output can be connected to the corresponding input.
(c) Removal of redundant LUTs, i. e., LUTs with identical content and identical inputs can be shared.

In particular, the removal of redundant LUTs can be used to reduce logic when multiple constants are involved [35]. It was shown that it is very likely that identical LUTs occur in MCM operations. For example, only 52 unique LUTs are required to be able to compute all signed products with constants from 1 to 2^{12} using four-input LUTs. Compared to adder graphs with minimal adder depth, the latency could be reduced by approximately 33% on average for an input word size of 8 bits. However, even for this small input word size, clock frequencies of about 100 MHz could be achieved which is far less than what is possible on the used Virtex 4 FPGA. Hence, in our work, pipelined realizations of LUT multipliers are investigated, as shown in Figure 5. In the first pipeline stage, the LUT content is looked up, and the results are shifted and added according to (25) using a pipelined adder tree. Note that some of the bit shifts can be moved toward the output of the adder tree to reduce the word size in each stage (not shown in Figure 5). No extra BLE has to be reserved for pure registers as the flip-flops of BLEs using LUTs, and full adders are used. But from the reduction methods, only the removal of redundant LUTs can be used as these can be shared together with the flip-flop, while the other methods still require a BLE for a single flip-flop.

ILP formulation for the combined pipelined adder/LUT graph optimization

ILP Formulation 1 of the PAG optimization is now extended to include LUT multipliers. Thus, the ILP solver should decide whether a target coefficient is realized using adders or using LUTs. Of course, if more than one LUT multiplier is used, the sharing of identical LUTs should be included in the cost model. For that, the set $\mathcal{L}^s_{v,w}$ is defined, which contains all LUTs of stage s which are needed to compute each bit of w from v and a new boolean decision variable is introduced:

$$l^s_{v,i} = \begin{cases} 1 & \text{if LUT with input node } v \text{ and LUT content } i \\ & \text{is available in stage } s \\ 0 & \text{otherwise} \end{cases} \tag{26}$$

If a LUT multiplier is inserted in the adder graph to compute w from node v, the variable $a^s_{(0,v,w)}$ is set to one (the case $u = 0$ does not exist in ILP Formulation 1). The corresponding $(0, v, w)$ triplets have to be added to $\mathcal{T}^s$. Now, ILP Formulation 1 can be extended by two constraints (C4) and (C5) (the remaining constraints remain nearly identical) and a modified objective function:

ILP Formulation 2 : Combined pipelined adder graph/LUT optimization

$$\text{minimize} \quad \sum_{s=1}^{S} \sum_{(u,v,w)\in\mathcal{T}^s} \left(\text{cost}(u,v,w) a^s_{(u,v,w)} + \sum_{i\in\mathcal{L}^s_{v,w}} \text{cost}_{\text{LUT}}\, l^s_{v,i} \right)$$

$$\text{with} \quad \text{cost}(u,v,w) = \begin{cases} \text{cost}_{\text{AT}}(u,v,w) & \text{for } u = 0 \\ \text{cost}_{\mathcal{A}}(u,v,w) & \text{otherwise} \end{cases}$$

subject to

$$\text{(C1)} \quad a^S_w = 1 \text{ for all } w \in \mathcal{S}^S$$

$$\text{(C2)} \quad a^s_w - \sum_{\forall(i,j,k)\in\mathcal{T}^s | k=w} a^s_{(i,j,k)} \le 0 \text{ for all} w \in \mathcal{S}^s, s = 2 \dots S$$

$$\text{(C3)} \quad \left.\begin{array}{r} a^s_{(u,v,w)} - a^{s-1}_u \le 0 \\ a^s_{(u,v,w)} - a^{s-1}_v \le 0 \end{array}\right\} \text{ for all } (u,v,w) \in \mathcal{T}^s, s = 2 \dots S \text{ with } u \neq 0$$

$$\text{(C4)} \quad a^s_{(0,v,w)} - a^{s-LD(w/v)}_v \le 0 \quad \begin{array}{l} \text{for all } (u,v,w) \in \mathcal{T}^s, s = 2 \dots S \\ \text{with } w \bmod v = 0,\ s \ge LD(w/v) \text{ and } u = 0 \end{array}$$

$$\text{(C5)} \quad a^s_{(0,v,w)} - l^s_{v,i} \le 0 \quad \begin{array}{l} \text{for all } (u,v,w) \in \mathcal{T}^s, i \in \mathcal{L}^s_{v,w}, s = 2 \dots S \\ \text{with } u = 0 \end{array}$$

$$a^s_w \in \{0,1\},\ a^s_{(u,v,w)} \ge 0,\ l^s_{v,i} \in \{0,1\}$$

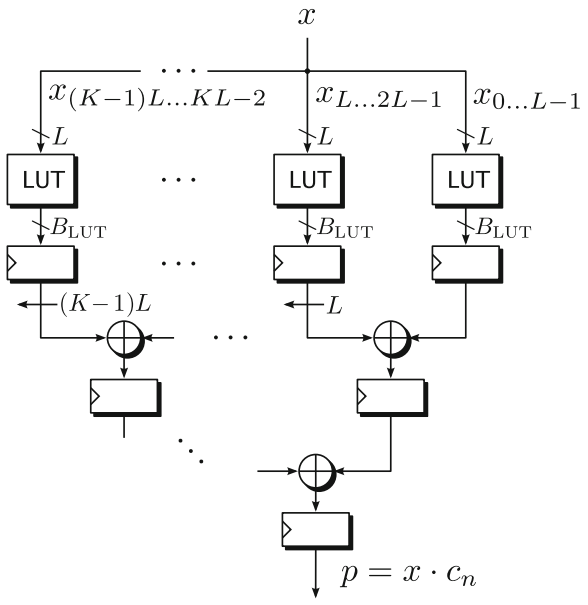

Figure 5 Pipelined LUT-based multiplier.

Two cost functions were added, $\mathrm{cost}_{\mathrm{LUT}}$ includes the cost of a single LUT and $\mathrm{cost}_{\mathrm{AT}}(u, v, w)$ corresponds to the cost of the adder tree which is necessary to compute the sum of the LUT outputs. The splitting of the cost function is necessary to respect the reduced cost for shared LUTs. For Virtex 4 or Virtex 6/7 using six-input LUTs, each LUT consumes one BLE ($\mathrm{cost}_{\mathrm{LUT}} = 1$). For Virtex 6/7 using five-input LUTs, the LUT costs can be approximated to $\mathrm{cost}_{\mathrm{LUT}} = 1/2$ as only one half of a BLE is used for a single LUT. The adder tree costs ($\mathrm{cost}_{\mathrm{AT}}(u, v, w)$) depend on v, w and the LUT multiplier pipeline depth, which is defined by the depth of the adder tree plus one for the LUT registers:

$$\mathrm{LD}(w/v) = \log_2\left(\lceil B_v/L \rceil\right) + 1 \tag{27}$$

The additional constraints (C4) and (C5) are defined to incorporate LUT realizations during optimization. Constraint (C4) is similar to (C3): If a LUT multiplier is used to compute w in stage s from v in stage $s - \mathrm{LD}(w/v)$, the corresponding node v must be available in that stage. Constraint (C5) ensures that the corresponding LUTs are available in the correct pipeline stage to compute w from v. In first experiments, we found only cases where LUT multipliers directly compute target coefficients of the last stage. Hence, we decided to reduce the search space by evaluating the constraints (C4) and (C5) only for the target constants of the last stage $s = S$.

Results

To evaluate the performance of the two ILP formulations, a benchmark set of folding matrices was designed using Matlab. The coefficients were computed using the `fspecial()` function except from the lowpass and highpass filters. These were obtained by the Parks-McClellan algorithm (`remez()` and `ftrans2()` functions). All real coefficients were quantized to a word size of B_c. The complete convolution matrices are given in Appendix 1; the filter parameters are summarized in Table 1 with their matrix size, word size, the required pipeline stages S using PMCM, parameters for their design, and the N_{uq} unique odd coefficients of their folding matrix.

The BILP solver of Matlab is slow compared to the CPLEX optimizer from IBM [46] and does not provide a solver for MILP problems. CPLEX provides a text file interface with a comfortable human readable syntax (LP file format [47]). Hence, Matlab was used to generate

Table 1 Parameters of the used benchmark filters

Filter type	Filter size	B_c	N_{uq}	S	Parameter	Unique odd coefficients
Gaussian	3 × 3	8	3	2	$\sigma = 0.5$	3 21 159
Gaussian	5 × 5	12	4	3		1 23 343 1,267
Laplacian	3 × 3	8	3	2	$\alpha = 0.2$	5 21 107
Unsharp	3 × 3	8	3	2	$\alpha = 0.2$	3 11 69
Unsharp	3 × 3	12	3	3		43 171 1,109
Lowpass	5 × 5	8	5	2		11 33 35 53 103
Lowpass	9 × 9	10	13	2	$f_{\mathrm{pass}} = 0.2$, $f_{\mathrm{stop}} = 0.4$	1 5 7 25 31 63 65 67 73 97 117 165 303
Lowpass	15 × 15	12	26	3		1 5 7 13 17 19 21 27 41 43 45 53 61 79 93 101
						103 113 133 137 199 331 333 613 1,097 1,197
Highpass	5 × 5	8	5	2	$f_{\mathrm{stop}} = 0$, $f_{\mathrm{pass}} = 0.2$	1 3 5 7 121
Highpass	9 × 9	10	6	2		1 3 5 7 11 125
Highpass	15 × 15	12	13	2		1 3 5 7 9 11 13 15 17 19 21 23 507

the MILP models as LP files, and CPLEX was used for solving them. Then, Matlab was used to read in the solution file of CPLEX in XML format [47] for generating synthesizable VHDL code. All MCM blocks were optimized for an input word size B_x of 8, 10, and 12 bits. The results are compared with the RPAG algorithm [9] and the LUT MCM method of [35], which was applied to the pipelined realization, as shown in Figure 5. The RPAG

Table 2 Optimization results in terms of the number of basic logic elements (BLE) for the previous methods RPAG [9] using *R* iterations, a LUT MCM block [35] including pipelining and the proposed methods for optimal pipelined adder graphs using ILP Formulation 1 (Optimal PAG) and optimal pipelined adder/LUT graphs using ILP Formulation 2 (Optimal PALG), including the coefficients realized by LUTs (LUT Coeffs)

				No. of BLE					
Filter type	Filter size	B_c	B_x	RPAG [9] ($R = 1$)	RPAG [9] ($R = 50$)	LUT MCM [35]	Optimal PAG	Optimal PALG	LUT Coeffs
Gaussian	3 × 3	8	8	63	58	56	58	56	All
Gaussian	5 × 5	12	8	125	125	77	111	77	All
Laplacian	3 × 3	8	8	79	61	54	61	54	All
Unsharp	3 × 3	8	8	56	56	51	56	51	All
Unsharp	3 × 3	12	8	112	107	64	91	64	All
Lowpass	5 × 5	8	8	98	98	91	98	91	All
Lowpass	9 × 9	10	8	235	221	192	221	192	All
Lowpass	15 × 15	12	8	480	475	371	≤478	371	All
Highpass	5 × 5	8	8	74	74	69	74	69	All
Highpass	9 × 9	10	8	85	85	83	85	83	All
Highpass	15 × 15	12	8	186	186	170	186	170	All
Gaussian	3 × 3	8	10	73	68	71	68	68	None
Gaussian	5 × 5	12	10	145	143	98	129	98	All
Laplacian	3 × 3	8	10	93	71	69	71	68	All
Unsharp	3 × 3	8	10	66	66	66	66	66	None
Unsharp	3 × 3	12	10	130	123	80	105	80	All
Lowpass	5 × 5	8	10	114	114	118	114	112	33
Lowpass	9 × 9	10	10	271	255	276	255	254	117
Lowpass	15 × 15	12	10	550	543	546	≤547	≤545	None
Highpass	5 × 5	8	10	88	88	95	88	87	121
Highpass	9 × 9	10	10	101	101	115	101	101	None
Highpass	15 × 15	12	10	218	218	254	218	216	23
Gaussian	3 × 3	8	12	83	78	143	78	78	None
Gaussian	5 × 5	12	12	165	161	203	147	147	None
Laplacian	3 × 3	8	12	107	81	142	81	81	None
Unsharp	3 × 3	8	12	76	76	135	76	76	None
Unsharp	3 × 3	12	12	148	139	173	119	119	None
Lowpass	5 × 5	8	12	130	130	242	130	130	None
Lowpass	9 × 9	10	12	307	289	589	289	289	None
Lowpass	15 × 15	12	12	620	611	1203	≤620	≤620	None
Highpass	5 × 5	8	12	102	102	192	102	102	None
Highpass	9 × 9	10	12	117	117	232	117	117	None
Highpass	15 × 15	12	12	250	250	523	250	250	None
			Average:	168.09	162.73	207.36	160.3	150.97	
	Improvement to RPAG ($R = 50$):			–	–	−27.43%	1.49%	7.23%	

algorithm is a greedy heuristic. As the best local choice in a greedy algorithm does not necessarily lead to the best global solution, it allows to randomly select one of the nth best decisions. Then, the best result out of several runs can be taken to improve the optimization. RPAG was configured for a single run ($R = 1$, pure greedy), and $R = 50$ runs per MCM instance where the locally first or second best decision was randomly selected. These results were considered as state-of-the-art reference. The optimization was performed for the Virtex 6/7 FPGA architecture, and CPLEX was configured with a computation time limit of 8 hours.

The results are summarized in Table 2 for RPAG ($R = 1$ and $R = 50$) and the LUT MCM method which are compared to the proposed pipelined adder graph (Optimal PAG) and pipelined adder/LUT graph (Optimal PALG) methods. An optimal solution could be found within the time limit for all cases, except for the lowpass 15×15 instances. Here, the best obtained result is shown as upper bound. All solutions are given as PAG in Appendix 2. The coefficients which has to be replaced by LUT multipliers to obtain the PALG solution are given in the last column of Table 2. Comparing RPAG with the optimal PAG solution, it is apparent that RPAG ($R = 50$) often finds an optimal solution (in 24 out of 32 cases). For the three cases where the timeout of the ILP solver occurred, RPAG finds slightly better solutions. Comparing the pipelined LUT MCM method with RPAG and optimal PAG, the LUT MCM method performs better for low-input word sizes as expected. For $B_x = 8$, it is always the best choice; for $B_x = 10$, it is sometimes the best; and for $B_x = 12$, it is never the best choice. This is different to the combined PALG optimization which found the best results in all cases where an optimal solution was obtained. For $B_x = 8$, all instances were pure LUT MCM realizations; for $B_x = 10$, there is a mixture of pure adder graphs, pure LUT realizations, and combinations of both; while for $B_x = 12$, the adder graph realizations dominate. An example is given in Figure 6. While the PAG in Figure 6a needs two elements in the second pipeline stage (1 and 7) to compute coefficient 121, this coefficient is realized with a two-stage LUT multiplier in Figure 6b, saving one element in stage two. An overall reduction of 1.5% and 7.2% of BLEs compared to RPAG could be achieved for the optimal PAG and PALG methods, respectively.

Table 3 Minimum, maximum, and average computation times of the optimization algorithms for the benchmark instances of Table 2

	RPAG [9] ($R = 1$) [s]	RPAG [9] ($R = 50$) [s]	LUT only [s]	Optimal PAG [s]	Optimal PALG [s]
Minimum	0.01	4.06	0.11	0.15	0.1
Maximum	0.94	57.6	1.21	28,800	28,800
Average	0.15	11.16	0.31	3,538.58	2,129.79

The minimum, maximum, and average computation times on an Intel Nehalem 2.93-GHz computer (Intel, Santa Clara, CA, USA) for all used optimization algorithms are given in Table 3. All heuristics take up to a few seconds; the optimal methods are very fast for the pipeline depth $S = 2$ cases in Table 1 but take hours for the three matrices with $S = 3$ (up to the time limit of 8 h = 28,800 s for the lowpass 15×15 instances). However, the optimization time is reasonable compared to the common FPGA development effort.

To validate the optimization results, synthesis experiments were performed for a Virtex 6 XC6VLX75T-2FF484 FPGA. For that, VHDL code generators were developed in Matlab for all the examined architectures. The synthesis was performed using Xilinx ISE v13.4. Unfortunately, the BLE usage cannot be directly obtained by counting the

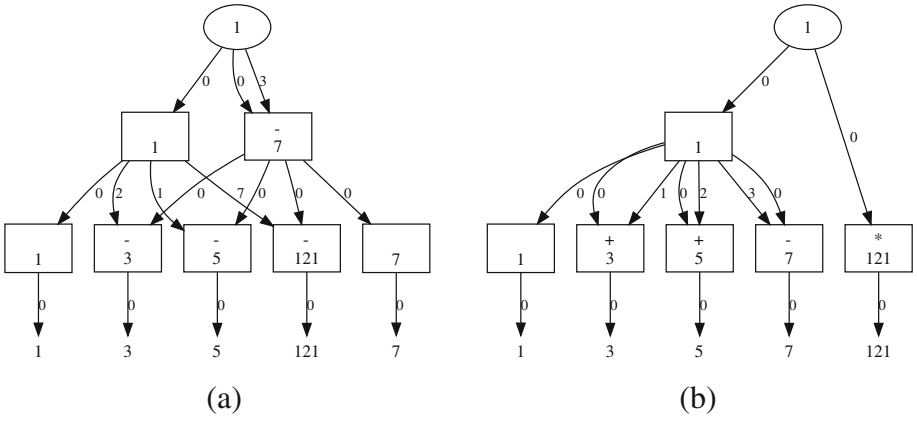

Figure 6 Example PAG/PALG of the highpass 5×5 instance with $B_{x=10}$ bits. (**a**) Optimal PAG, (**b**) optimal PALG.

FFs, LUTs, or slices reported by the tool, as a Virtex 6 may use more than one FF per BLE, a LUT may be unused, and many slices may not be fully used but can be used for other logic. Therefore, the resulting netlist after place&route was converted to an XDL file, and the XDL-Parser from the RapidSmith Project [48] was used to obtain the BLEs directly from the netlist. All synthesis results are listed in Table 4. It can be seen that in most cases the optimal PALG solution leads to the best synthesis result. However, there are some cases where the estimated cost model fails

Table 4 Synthesis results using the same benchmark instances as in Table 2 providing the actual BLEs as well as the maximum clock frequency f_{max} on a Virtex 6 FPGA

Filter type	Filter size	B_c	B_x	RPAG [9] ($R = 50$)		LUT MCM [35]		Optimal PAG		Optimal PALG	
				BLE	f_{max}	BLE	f_{max}	BLE	f_{max}	BLE	f_{max}
Gaussian	3 × 3	8	8	58	713.8	58	720.5	58	730.5	58	720.5
Gaussian	5 × 5	12	8	125	635.3	68	633.3	111	619.2	68	633.3
Laplacian	3 × 3	8	8	61	749.1	52	718.9	61	701.8	52	718.9
Unsharp	3 × 3	8	8	56	727.8	51	731.0	56	709.7	51	731.0
Unsharp	3 × 3	12	8	107	636.9	59	701.3	91	657.9	59	701.3
Lowpass	5 × 5	8	8	98	665.8	93	652.3	98	657.9	93	652.3
Lowpass	9 × 9	10	8	221	593.5	186	573.1	221	582.4	186	573.1
Lowpass	15 × 15	12	8	469	530.2	368	525.8	478	489.0	368	525.8
Highpass	5 × 5	8	8	74	605.3	67	726.2	74	711.7	67	726.2
Highpass	9 × 9	10	8	85	718.4	81	655.3	85	689.2	81	655.3
Highpass	15 × 15	12	8	184	613.9	166	630.5	183	598.4	166	630.5
Gaussian	3 × 3	8	10	64	713.3	70	776.4	68	745.7	64	666.7
Gaussian	5 × 5	12	10	139	653.6	82	690.6	129	637.8	82	690.6
Laplacian	3 × 3	8	10	71	682.6	59	745.2	71	656.6	59	745.2
Unsharp	3 × 3	8	10	66	729.9	60	769.8	66	712.3	66	712.3
Unsharp	3 × 3	12	10	118	682.1	69	690.1	103	605.3	69	690.1
Lowpass	5 × 5	8	10	110	640.6	106	627.0	114	646.8	111	647.3
Lowpass	9 × 9	10	10	255	557.4	225	574.4	251	602.1	252	528.3
Lowpass	15 × 15	12	10	539	526.0	443	414.8	546	514.7	529	470.8
Highpass	5 × 5	8	10	88	700.3	89	698.8	86	704.2	87	721.5
Highpass	9 × 9	10	10	97	616.5	91	644.8	97	677.5	97	677.5
Highpass	15 × 15	12	10	213	561.8	218	600.6	218	567.9	203	583.8
Gaussian	3 × 3	8	12	78	646.4	129	688.2	78	644.3	78	644.3
Gaussian	5 × 5	12	12	161	619.6	150	615.4	147	624.2	147	635.7
Laplacian	3 × 3	8	12	81	684.0	104	672.0	81	724.1	81	724.1
Unsharp	3 × 3	8	12	76	721.5	129	638.2	72	638.6	72	638.6
Unsharp	3 × 3	12	12	132	614.3	122	620.0	119	638.6	119	638.6
Lowpass	5 × 5	8	12	127	601.3	198	557.1	126	653.2	130	625.8
Lowpass	9 × 9	10	12	285	603.1	412	606.4	278	581.7	273	602.4
Lowpass	15 × 15	12	12	606	521.4	835	530.8	619	530.8	620	517.9
Highpass	5 × 5	8	12	102	669.8	145	647.7	94	714.3	94	714.3
Highpass	9 × 9	10	12	113	655.7	159	624.6	113	680.7	113	680.7
Highpass	15 × 15	12	12	238	591.4	388	593.5	239	539.7	250	590.7
			Average	160.52	641.9	167.64	645.28	158.52	642.08	146.82	648.94
	Improvement to RPAG ($R = 50$):			–	–	−4.4%	−0.5%	1.25%	0.03%	8.53%	1.09%

Table 5 Convolution matrices of the benchmark filters

Filter	Convolution matrix
Gaussian 3×3, $B_c = 8$ bits	$\begin{pmatrix} 3 & 21 & 3 \\ 21 & 159 & 21 \\ 3 & 21 & 3 \end{pmatrix}$
Gaussian 5×5, $B_c = 12$ bits	$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 46 & 343 & 46 & 0 \\ 1 & 343 & 2,534 & 343 & 1 \\ 0 & 46 & 343 & 46 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$
Laplacian 3×3, $B_c = 8$ bits	$\begin{pmatrix} 5 & 21 & 5 \\ 21 & -107 & 21 \\ 5 & 21 & 5 \end{pmatrix}$
Unsharp 3×3, $B_c = 8$ bits	$\begin{pmatrix} -3 & -11 & -3 \\ -11 & 69 & -11 \\ -3 & -11 & -3 \end{pmatrix}$
Unsharp 3×3, $B_c = 12$ bits	$\begin{pmatrix} -43 & -171 & -43 \\ -171 & 1109 & -171 \\ -43 & -171 & -43 \end{pmatrix}$
Lowpass 5×5, $B_c = 8$ bits	$\begin{pmatrix} 22 & 88 & 132 & 88 & 22 \\ 88 & 140 & 103 & 140 & 88 \\ 132 & 103 & 106 & 103 & 132 \\ 88 & 140 & 103 & 140 & 88 \\ 22 & 88 & 132 & 88 & 22 \end{pmatrix}$
Lowpass 9×9, $B_c = 10$ bits	$\begin{pmatrix} -1 & -7 & -25 & -50 & -62 & -50 & -25 & -7 & -1 \\ -7 & -25 & -10 & 73 & 130 & 73 & -10 & -25 & -7 \\ -25 & -10 & 117 & 165 & 126 & 165 & 117 & -10 & -25 \\ -50 & 73 & 165 & 194 & 303 & 194 & 165 & 73 & -50 \\ -62 & 130 & 126 & 303 & 268 & 303 & 126 & 130 & -62 \\ -50 & 73 & 165 & 194 & 303 & 194 & 165 & 73 & -50 \\ -25 & -10 & 117 & 165 & 126 & 165 & 117 & -10 & -25 \\ -7 & -25 & -10 & 73 & 130 & 73 & -10 & -25 & -7 \\ -1 & -7 & -25 & -50 & -62 & -50 & -25 & -7 & -1 \end{pmatrix}$
Lowpass 15×15, $B_c = 12$ bits	$\begin{pmatrix} 0 & 0 & 1 & 5 & 13 & 27 & 40 & 45 & 40 & 27 & 13 & 5 & 1 & 0 & 0 \\ 0 & 2 & 7 & 13 & 2 & -41 & -103 & -133 & -103 & -41 & 2 & 13 & 7 & 2 & 0 \\ 1 & 7 & 10 & -21 & -93 & -137 & -101 & -64 & -101 & -137 & -93 & -21 & 10 & 7 & 1 \\ 5 & 13 & -21 & -106 & -122 & -41 & -17 & -43 & -17 & -41 & -122 & -106 & -21 & 13 & 5 \\ 13 & 2 & -93 & -122 & -8 & 79 & 199 & 304 & 199 & 79 & -8 & -122 & -93 & 2 & 13 \\ 27 & -41 & -137 & -41 & 79 & 333 & 613 & 662 & 613 & 333 & 79 & -41 & -137 & -41 & 27 \\ 40 & -103 & -101 & -17 & 199 & 613 & 904 & 1,097 & 904 & 613 & 199 & -17 & -101 & -103 & 40 \\ 45 & -133 & -64 & -43 & 304 & 662 & 1,097 & 1,197 & 1,097 & 662 & 304 & -43 & -64 & -133 & 45 \\ 40 & -103 & -101 & -17 & 199 & 613 & 904 & 1,097 & 904 & 613 & 199 & -17 & -101 & -103 & 40 \\ 27 & -41 & -137 & -41 & 79 & 333 & 613 & 662 & 613 & 333 & 79 & -41 & -137 & -41 & 27 \\ 13 & 2 & -93 & -122 & -8 & 79 & 199 & 304 & 199 & 79 & -8 & -122 & -93 & 2 & 13 \\ 5 & 13 & -21 & -106 & -122 & -41 & -17 & -43 & -17 & -41 & -122 & -106 & -21 & 13 & 5 \\ 1 & 7 & 10 & -21 & -93 & -137 & -101 & -64 & -101 & -137 & -93 & -21 & 10 & 7 & 1 \\ 0 & 2 & 7 & 13 & 2 & -41 & -103 & -133 & -103 & -41 & 2 & 13 & 7 & 2 & 0 \\ 0 & 0 & 1 & 5 & 13 & 27 & 40 & 45 & 40 & 27 & 13 & 5 & 1 & 0 & 0 \end{pmatrix}$

Table 5 Convolution matrices of the benchmark filters *Continued*

Highpass 5 × 5, $B_c = 8$ bits	$\begin{pmatrix} -2 & -7 & -10 & -7 & -2 \\ -7 & -3 & 8 & -3 & -7 \\ -10 & 8 & 121 & 8 & -10 \\ -7 & -3 & 8 & -3 & -7 \\ -2 & -7 & -10 & -7 & -2 \end{pmatrix}$
Highpass 9 × 9, $B_c = 10$ bits	$\begin{pmatrix} 0 & -2 & -6 & -11 & -14 & -11 & -6 & -2 & 0 \\ -2 & -7 & -10 & -3 & 4 & -3 & -10 & -7 & -2 \\ -6 & -10 & -1 & -2 & -11 & -2 & -1 & -10 & -6 \\ -11 & -3 & -2 & -11 & -1 & -11 & -2 & -3 & -11 \\ -14 & 4 & -11 & -1 & 500 & -1 & -11 & 4 & -14 \\ -11 & -3 & -2 & -11 & -1 & -11 & -2 & -3 & -11 \\ -6 & -10 & -1 & -2 & -11 & -2 & -1 & -10 & -6 \\ -2 & -7 & -10 & -3 & 4 & -3 & -10 & -7 & -2 \\ 0 & -2 & -6 & -11 & -14 & -11 & -6 & -2 & 0 \end{pmatrix}$
Highpass 15 × 15, $B_c = 12$ bits	$\begin{pmatrix} 0 & 0 & 0 & -1 & -3 & -6 & -8 & -10 & -8 & -6 & -3 & -1 & 0 & 0 & 0 \\ 0 & 0 & -2 & -5 & -8 & -8 & -5 & -4 & -5 & -8 & -8 & -5 & -2 & 0 & 0 \\ 0 & -2 & -6 & -9 & -8 & -6 & -10 & -13 & -10 & -6 & -8 & -9 & -6 & -2 & 0 \\ -1 & -5 & -9 & -8 & -8 & -14 & -14 & -11 & -14 & -14 & -8 & -8 & -9 & -5 & -1 \\ -3 & -8 & -8 & -8 & -15 & -15 & -15 & -19 & -15 & -15 & -15 & -8 & -8 & -8 & -3 \\ -6 & -8 & -6 & -14 & -15 & -17 & -21 & -18 & -21 & -17 & -15 & -14 & -6 & -8 & -6 \\ -8 & -5 & -10 & -14 & -15 & -21 & -19 & -23 & -19 & -21 & -15 & -14 & -10 & -5 & -8 \\ -10 & -4 & -13 & -11 & -19 & -18 & -23 & 2,028 & -23 & -18 & -19 & -11 & -13 & -4 & -10 \\ -8 & -5 & -10 & -14 & -15 & -21 & -19 & -23 & -19 & -21 & -15 & -14 & -10 & -5 & -8 \\ -6 & -8 & -6 & -14 & -15 & -17 & -21 & -18 & -21 & -17 & -15 & -14 & -6 & -8 & -6 \\ -3 & -8 & -8 & -8 & -15 & -15 & -15 & -19 & -15 & -15 & -15 & -8 & -8 & -8 & -3 \\ -1 & -5 & -9 & -8 & -8 & -14 & -14 & -11 & -14 & -14 & -8 & -8 & -9 & -5 & -1 \\ 0 & -2 & -6 & -9 & -8 & -6 & -10 & -13 & -10 & -6 & -8 & -9 & -6 & -2 & 0 \\ 0 & 0 & -2 & -5 & -8 & -8 & -5 & -4 & -5 & -8 & -8 & -5 & -2 & 0 & 0 \\ 0 & 0 & 0 & -1 & -3 & -6 & -8 & -10 & -8 & -6 & -3 & -1 & 0 & 0 & 0 \end{pmatrix}$

to predict the synthesis. The following cases have been identified by examining the netlists with the Xilinx' FPGA Editor:

1. A single Virtex 6 BLE is able to realize two flip-flops (see Figure 3b), but sometimes, ISE uses one BLE to realize a single register and sometimes to realize two registers.
2. Sometimes, ISE maps a single flip-flop in a BLE that is also used for a full adder.

Hence, an overestimate was done in the cost model with the assumption that single flip-flops are mapped to a single BLE. Due to the large shifts in the LUT multiplier architecture (see Figure 5), the least significant bits in the adder tree are pure flip-flops. Thus, it is more likely that a LUT multiplier is overestimated and adders are used instead. However, the synthesis results show that a BLE reduction of 1.3% and 8.5% on average compared to RPAG could be achieved for the optimal PAG and PALG method, respectively. The speed of the different architectures is very similar. In 75.8% and 78.8% of the cases for optimal PAG and PALG, respectively, they were even faster than the maximum clock frequency of the embedded DSP48E1 multiplier (which is 600 MHz).

Conclusion and outlook

Two ILP formulations to optimize the multiple constant multiplication on FPGAs were presented and analyzed using synthesis experiments. The first one is a formulation of the PMCM problem, for which only heuristics exist [7,9]. It was shown that better results are achievable for the used low word size coefficients. For most

instances, the RPAG heuristic is also able to find an optimal solution (in 24 out of 32 cases). The second ILP formulation incorporates pipelined LUT-based multipliers. It was shown that this combined method outperforms the PMCM realization in particular for a low-input word size of 8 to 10 bits.

The used ILP solver CPLEX was able to find an optimal solution for most of the test instances within 8 h of computation time. If not, the best feasible solution was close to that of the RPAG heuristic. Synthesis experiments were performed for the Xilinx Virtex 6 architecture, showing that compact and fast multiple constant multipliers are obtained. A resource reduction of 8.5% was achieved compared to the state-of-the-art while having approximately the same speed. Thus, the proposed optimizations can be beneficially used for many

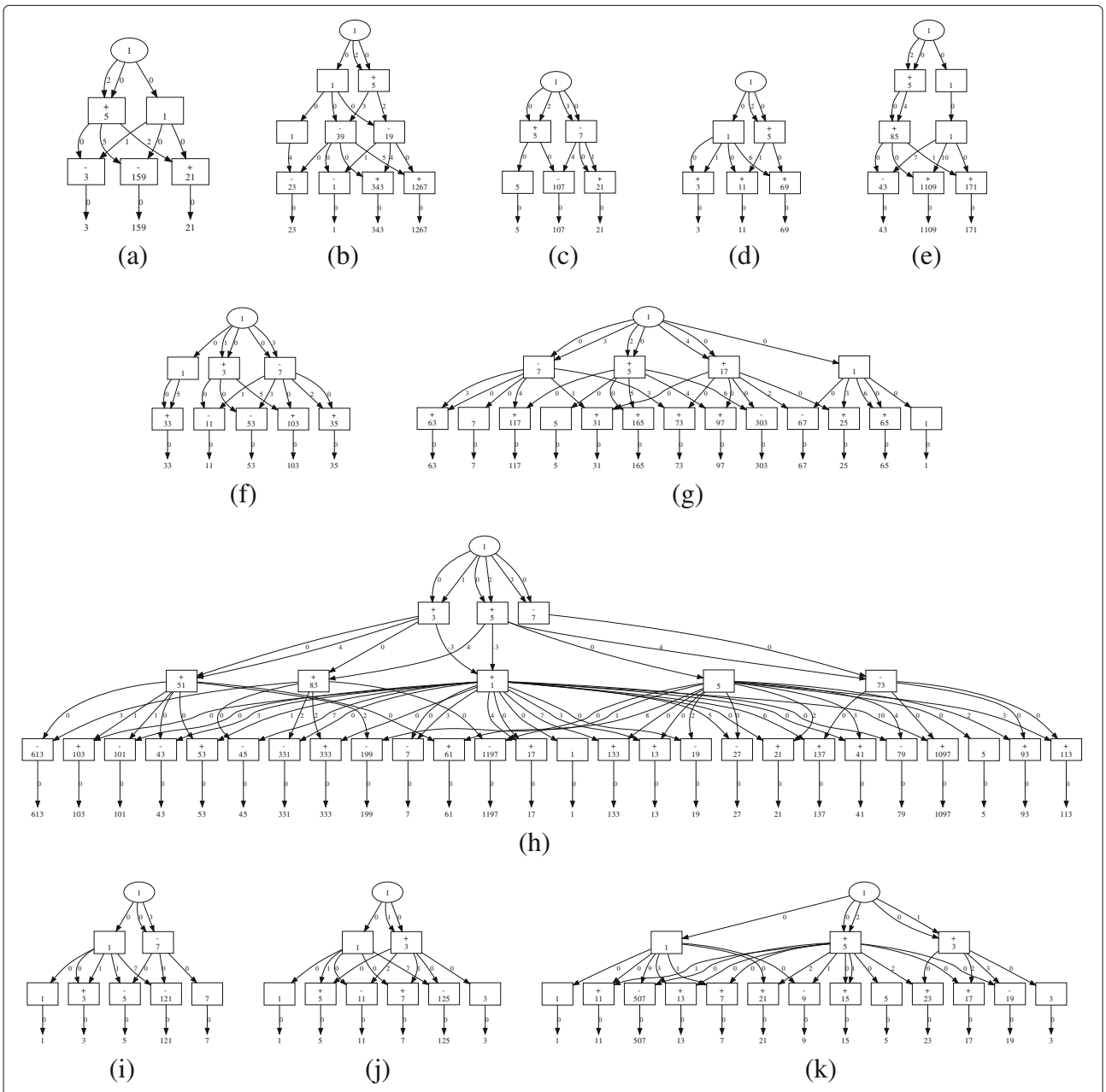

Figure 7 Pipelined adder graph solutions of the benchmark convolution matrices. (**a**) Gaussian 3×3 ($B_C = 8$ bits; $B_X = 8, 10, 12$ bits). (**b**) Gaussian 5×5 ($B_C = 12$ bits; $B_X = 8, 10, 12$ bits). (**c**) Laplacian 3×3 ($B_C = 8$ bits; $B_X = 8, 10, 12$ bits). (**d**) Unsharp 3×3 ($B_C = 8$ bits; $B_X = 8, 10, 12$ bits). (**e**) Unsharp 3×3 ($B_C = 12$ bits; $B_X = 8, 10, 12$ bits). (**f**) Lowpass 5×5 ($B_C = 8$ bits; $B_X = 8, 10, 12$ bits). (**g**) Lowpass 9×9 ($B_C = 10$ bits; $B_X = 8, 10, 12$ bits). (**h**) Lowpass 15×15 ($B_C = 12$ bits; $B_X = 8, 10, 12$ bits). (**i**) Highpass 5×5 ($B_C = 8$ bits; $B_X = 8, 10, 12$ bits). (**j**) Highpass 9×9 ($B_C = 10$ bits; $B_X = 8, 10, 12$ bits). (**k**) Highpass 15×15 ($B_C = 12$ bits; $B_X = 8, 10, 12$ bits).

real-time video processing applications which involve FIR filters.

Future work could be extended into different directions. The examinations about the mismatch between cost model and synthesis results could be used to improve the results. The slice flip-flops in adders which were not used by synthesis could be utilized to implement pure registers. Furthermore, pure registers could be forced to use all of the eight available slice flip-flops. In the ILP model, this could be respected by reducing the cost for registers, e. g., setting the cost function for one flip-flop to a half BLE instead of a full BLE. The physical realization can be done using low-level placement constraints. However, this could reduce the performance as fixed placements (even relative) may limit a timing driven place & route optimization.

Another extension could be the optimization with adder graphs containing ternary adders, i. e., adders with three inputs. Modern FPGAs provide methods to implement ternary adders with the same number of slices/ALMs as needed for a two-input adder with equal output word size but with a reduced speed due to a longer critical path [49,50]. The ILP formulations could be extended in that direction using quadruplets instead of the (u, v, w) triplets which allows the modeling of three inputs instead of the two inputs u and v. However, this would substantially increase the search space, so one has to investigate if a solution can be found in an acceptable runtime which is left open for future research.

Appendices

Appendix 1. Convolution matrices of the benchmark filters

The convolution matrices which were used as benchmark are listed in Table 5. They were used to produce the results, listed in Tables 2, 3, 4. Their properties are summarized in Table 1.

Appendix 2. Adder graphs results of PMCM optimization

The pipelined adder graph solutions of the benchmark convolution matrices of Table 5 are shown in Figure 7. Only the realization of unique odd coefficients is shown; their even representation can be realized by additional left shifts. All solutions were optimal (obtained by optimal PAG) except lowpass 15×15, from which the optimal solution is unknown (obtained by RPAG).

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Digital Technology Group, University of Kassel, Kassel 34121, Germany.
[2]Department of Electrical & Computer Engineering, Florida State University, Tallahassee, FL 32310-6046, USA.

Received: 31 January 2013 Accepted: 30 April 2013
Published: 25 May 2013

References

1. A Bovik, *The Essential Guide to Image Processing*. (Academic Press, Waltham, 2009)
2. M Kumm, P Zipf, in *International Conference on Electronics, Circuits and Systems (ICECS)*. Hybrid multiple constant multiplication for FPGAs (IEEE Piscataway, 2012), pp. 556–559
3. DR Bull, DH Horrocks, Primitive operator digital filters. IEEE Proc. Circuits, Devices Syst. **138**(3), 401–412 (1991)
4. Y Voronenko, M Püschel, Multiplierless multiple constant multiplication. ACM Trans. Algorithms (TALG). **3**(2), 1–38 (2007)
5. U Meyer-Baese, J Chen, CH Chang, AG Dempster, in *Asia Pacific Conference on Circuits and Systems (APCCAS)*. A comparison of pipelined RAG-n and DA FPGA-based multiplierless filters (IEEE Piscataway, 2006), pp. 1555–1558
6. S Mirzaei, R Kastner, A Hosangadi, Layout aware optimization of high speed fixed coefficient FIR filters for FPGAs. Int. J Reconfigurable Comput. **3**, 1–17 (2010)
7. U Meyer-Baese, G Botella, D Romero, M Kumm, in *SPIE Defense Security+Sensing, Volume 8401*. Optimization of high speed pipelining in FPGA-based FIR filter design using genetic algorithm (SPIE Baltimore, 2012), pp. 1–12
8. M Kumm, P Zipf, in *International Conference on Field Programmable Technology (ICFPT)*. High speed low complexity FPGA-based FIR filters using pipelined adder graphs (IEEE Piscataway, 2011), pp. 1–4
9. M Kumm, M Faust, P Zipf, CH Chang, in *International Symposium on Circuits and Systems (ISCAS)*. Pipelined adder graph optimization for high speed multiple constant multiplication (IEEE Piscataway, 2012), pp. 49–52A
10. M Kumm, K Liebisch, P Zipf, in *International Conference on Field Programmable Logic and Applications (FPL)*. Reduced Complexity Single and Multiple constant multiplication in Floating point precision (IEEE Piscataway, 2012), pp. 255–261
11. R Hartley, Subexpression sharing in filters using canonic signed digit multipliers. IEEE Trans. Circuits and Syst. II: Analog Digit. Signal Process. **43**(10), 677–688 (1996)
12. S Mirzaei, A Hosangadi, R Kastner, in *International Conference on Computer Design (ICCD)*. FPGA implementation of high speed FIR filters using add and shift method (IEEE Piscataway, 2006), pp. 308–313
13. M Imran, K Khursheed, M O'Nils, in *International Conference on Signals and Electronic Systems (ICSES)*. On the number representation in sub-expression sharing (IEEE Piscataway, 2010), pp. 17–20
14. AG Dempster, MD Macleod, Constant integer multiplication using minimum adders. IEE Proc. Circuits, Devices Syst. **141**(5), 407–413 (1994)
15. AG Dempster, MD Macleod, Use of minimum-adder multiplier blocks in FIR digital filters. IEEE Trans. Circuits Syst. II: Analog Digit. Signal Process. **42**(9), 569–577 (1995)
16. O Gustafsson, in *International Symposium on Circuits and Systems (ISCAS)*. A difference based adder graph heuristic for multiple constant multiplication problems (IEEE Piscataway, 2007), pp. 1097–1100
17. L Aksoy, E Günes, P Flores, Search algorithms for the multiple constant multiplications problem: exact and approximate. Microprocessors and Microsystems. **34**(5), 151–162 (2010)
18. P Flores, J Monteiro, E Costa, in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. An exact algorithm for the maximal sharing of partial terms in multiple constant multiplications (IEEE Computer Society Washington, 2005), pp. 13–16
19. A Yurdakul, G Dündar, Multiplierless realization of linear DSP transforms by using common two-term expressions. J. VLSI Signal Process. **22**, 163–172 (1999)
20. L Aksoy, E Costa, P Flores, J Monteiro, in *43rd ACM/IEEE Design Automation Conference (DAC)*. Optimization of area under a delay constraint in digital filter synthesis using SAT-based integer linear programming (IEEE Piscataway, 2006), pp. 669–674
21. L Aksoy, E Costa, P Flores, J Monteiro, in *44th ACM/IEEE Design Automation Conference (DAC)*. Optimization of Area in Digital FIR Filters Using Gate-Level Metrics (IEEE, 2007), pp. 420–423
22. L Aksoy, E da Costa, P Flores, J Monteiro, Exact and approximate algorithms for the optimization of area and delay in multiple constant

multiplications. IEEE Trans. Computer-Aided Design Integrated Circuits Syst. **27**(6), 1013–1026 (2008)
23. L Aksoy, E Gunes, P Flores, in *NORCHIP*. An Exact Breadth-First Search Algorithm for the Multiple Constant Multiplications Problem (IEEE Piscataway, 2008), pp. 41–46
24. O Gustafsson, in *42nd Asilomar Conference on Signals, Systems and Computers*. Towards optimal multiple constant multiplication: a hypergraph approach (IEEE Piscataway, 2008), pp. 1805–1809
25. O Gustafsson, Lower bounds for constant multiplication problems. IEEE Trans. Circuits Syst. II: Express Briefs. **54**(11), 974–978 (2007)
26. L Aksoy, E Costa, P Flores, J Monteiro, in *Proceedings of the 21st Edition of the Great Lakes Symposium on VLSI*. Design of low-power multiple constant multiplications using low-complexity minimum depth operations (ACM New York, 2011), pp. 79–84
27. A Dempster, S Dimirsoy, I Kale, in *International Symposium on Circuits and Systems (ISCAS)*. Designing multiplier blocks with low logic depth (IEEE Piscataway, 2002), pp. 773–776
28. K Johansson, Low Power, Low Power and Low Complexity Shift-and-Add Based Computations. PhD thesis, Linköping University, Department of Electrical Engineering, 2008
29. M Faust, CH Chang, in *International Symposium on Circuits and Systems (ISCAS)*. Minimal logic depth adder tree optimization for multiple constant multiplication (IEEE Piscataway, 2010), pp. 457–460
30. HJ Kang, IC Park, FIR Filter Synthesis Algorithms for Minimizing the Delay and the Number of Adders. IEEE Trans. Circuits Syst. II: Analog Digital Signal Process. **48**(8), 770–777 (2001)
31. L Aksoy, E Costa, P Flores, J Monteiro, in *European Conference on Circuit Theory and Design (ECCTD)*. Optimization of gate-level area in high throughput multiple constant multiplications (IEEE Piscataway, 2011), pp. 609–612
32. O Gustafsson, A Dempster, *On the use of multiple constant multiplication in polyphase FIR filters and filter banks* (IEEE, Piscataway, 2004), pp. 53–56
33. L Aksoy, E Costa, P Flores, J Monteiro, *Design of low-complexity digital finite impulse response filters on FPGAs* (IEEE, Piscataway, 2012), pp. 1197–1202
34. M Wirthlin, Constant coefficient multiplication using look-up tables. J. VLSI Signal Process. **36**, 7–15 (2004)
35. M Faust, CH Chang, in *International Symposium on Circuits and Systems (ISCAS)*. Bit-parallel multiple constant multiplication using look-up tables on FPGA (IEEE Piscataway, 2011), pp. 657–660
36. A Crosisier, DJ Esteban, ME Levilio, V Riso, Digital filter for PCM encoded signals. US Patent No. 3777130 (1973)
37. S Zohar, New hardware realizations of nonrecursive digital filters. IEEE Trans. Comput. **22**(4), 328–338 (1973)
38. A Peled, B Liu, A new hardware realization of digital filters. IEEE Trans. Acoustics, Speech Signal Process. **22**(6), 456–462 (1974)
39. SA White, Applications of distributed arithmetic to digital signal processing: a tutorial review. IEEE ASSP Mag. **6**(3), 4–19 (1989)
40. W Sen, T Bin, Z Jim, in *International Conference on Communications, Circuits and Systems (ICCCAS)*. Distributed arithmetic for FIR filter design on FPGA (IEEE Piscataway, 2007), pp. 620–623
41. P Meher, S Chandrasekaran, A Amira, FPGA realization of FIR filters by efficient and flexible systolization using distributed arithmetic. IEEE Trans. Signal Process. **56**(7), 3009–3017 (2008)
42. M Kumm, K Möller, P Zipf, in *International Symposium on Circuits and Systems (ISCAS)*. Reconfigurable FIR filter using distributed arithmetic on FPGAs. (accepted for publication in 2013)
43. D Bailey, *Design for Embedded Image Processing on FPGAs*. (Wiley-IEEE Press, New York, 2011)
44. A Willson, Desensitized half-band filters. IEEE Trans. Circuits and Syst I: Regular Papers. **57**, 152–167 (2010)
45. WS Lu, HP Wang, A Antoniou, Design of two-dimensional FIR digital filters by using the singular-value decomposition. IEEE Trans. Circuits and Syst. **37**, 35–4 (1990)
46. IBM Inc, IBM ILOG CPLEX Optimizer. http://www.ilog.com/products/cplex. Accessed 16 April 2013
47. IBM Inc, IBM ILOG CPLEX V12.1 - File Formats Supported by CPLEX (2009)
48. C Lavin, M Padilla, J Lamprecht, P Lundrigan, B Nelson, B Hutchings, in *International Conference on Field Programmable Logic and Applications (FPL)*. RapidSmith: do-it-yourself CAD tools for Xilinx FPGAs (IEEE, 2011), pp. 349–355
49. JM Simkins, BD Philofsky, Structures and methods for implementing ternary adders/subtractors in programmable logic devices. US Patent No 7274211, Xilinx Inc. (2006)
50. G Baeckler, M Langhammer, J Schleicher, R Yuan, Logic cell supporting addition of three binary words. US Patent No 7565388, Altera Coop. (2009)

doi:10.1186/1687-6180-2013-111
Cite this article as: Kumm *et al.*: **FIR filter optimization for video processing on FPGAs.** *EURASIP Journal on Advances in Signal Processing* 2013 **2013**:111.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com