Optimizing FPGA-based vector product designs

Daniel Benyamin[1], Wayne Luk[2], and John Villasenor[1]

[1]Electrical Engineering Department, UCLA, Los Angeles, CA, 90095
{benyamin,villa}@icsl.ucla.edu
[2]Department of Computing, Imperial College
180 Queen's Gate, London, England SW7 2BZ

Abstract

This paper presents a method, called multiple constant multiplier trees (MCMTs), for producing optimized reconfigurable hardware implementations of vector products. An algorithm for generating MCMTs has been developed and implemented, which is based on a novel representation of common subexpressions in constant data patterns. Our optimization framework covers a wider solution space than previous approaches; it also supports exploitation of full and partial run-time reconfiguration as well as technology-specific constraints, such as fanout limits and routing. We demonstrate that while distributed arithmetic techniques require storage size exponential in the number of coefficients, the resource utilization of MCMTs usually grows linearly with problem size. MCMTs have been implemented in Xilinx 4000 and Virtex FPGAs, and their size and speed efficiency are confirmed in comparisons with Xilinx LogiCore and ASIC implementations of FIR filter designs. Preliminary results show that the size of MCMT circuits is less than half of that of comparable distributed arithmetic cores.

1 Introduction

The vector dot product (or "sum of products") is one of the most common algorithmic kernels in signal processing. Its use ranges from filters and correlators to complete two-dimensional image transforms such as the Discrete Cosine Transform (DCT).

The MOJAVE Project at UCLA [1] has shown that FPGAs can be highly effective in computing two-dimensional correlations using 1-bit adder trees. Specifically, the authors indicate an important way of leveraging the reconfigurability of FPGAs: an FPGA circuit designer can afford to implement circuits based on parameters that rarely or never change; these circuits can be designed very efficiently due to the availability of *a priori* knowledge. Variable-operand circuits are preferred over constant-operand circuits in ASIC designs, due to the high non-recurring engineering cost required to change the constant operands. Indeed, constant-operand circuits only find their way into ASICs when they are widely applicable; the 8x8 matrix of DCT coefficients is one such example.

Unless the problem size is small, however, implementing effective constant-operand FPGA circuits by hand can be very tedious. This paper presents a hardware implementation of vector products called multiple constant multiplier trees (MCMTs). The core of the MCMT algorithm is our novel representation of common subexpressions contained in constant data patterns, which provides for several optimization capabilities not present in previous design strategies. First, we adopt a global optimization strategy to cover a wide solution space, thus generating solutions not considered by previous approaches. Second, our method is able to exploit the run-time reconfigurability of FPGAs, which can profoundly reduce the effective circuit size. Third, technology-specific constraints, such as fanout limits and routing, can be taken into account by our optimization procedure.

2 Overview and Related Work

A number of hardware implementations exist for computing vector products. There are straightforward designs, where the vector product is computed using constant coefficient multipliers (KCMs) and adders. Other techniques, such as distributed arithmetic (DA), directly compute vector products in a less obvious manner. In section 8 we provide comparisons of MCMTs to DA and other traditional architectures.

Our approach is to break down multiple constants to the bit level and represent them in a manner that

is efficient for optimizing the required number of additions. While multiplications with a constant value are commonly performed using shifts and additions, research on the joint optimization of multiple constants is relatively new. Potkonjak [4] presents a generalized system named multiple constant multipliers (MCMs). This approach can be applied to any problem where an unknown is multiplied by more than one constant value, and can optimize for both the number of additions and the number of required shifts. Feher [6] concentrates on generating vector product circuits for FPGAs, and applies the techniques to the two-dimensional DCT. Feher's work implements only bit-serial designs, which not only simplifies the problem of high fanout nodes, but, due to the necessary parallel-serial conversion, is only of interest when the application can afford a low throughput. Furthermore, Feher's algorithm utilizes a pairwise greedy approach, which provides poor results for many problem cases. Additional work includes recoding for multiple coefficients [5] and optimizations for DSP architectures [3]. These techniques, however, do not exploit the full potential of FPGAs, especially in regards to run-time reconfiguration (RTR).

3 Constant Multiplier Trees

For this paper we will apply the MCMT technique to problems involving the vector product,

$$Y = \sum_{k=0}^{K-1} A_k X_k \tag{1}$$

where X is an unknown vector, but the vector A is known at compile time. Although we develop a methodology based on this simple vector product, the same techniques are applicable to any multiple constant multiplication problem.

We begin by decomposing the vector product in a manner that takes advantage of our *a priori* knowledge of the constant vector A. Specifically, we write

$$A_k = \sum_{l=0}^{L-1} a_{lk} 2^l,$$

where a_{lk} is a single bit and L is the precision of the coefficients. We assume for simplicity that A_k is a positive integer, but the derivation can be applied to signed fixed-point systems as well. Substituting this into (1) gives

$$Y = \sum_{k=0}^{K-1} X_k \left(\sum_{l=0}^{L-1} a_{lk} 2^l \right),$$

and after exchanging the order of summations, we have

$$Y = \sum_{l=0}^{L-1} \left(\sum_{k=0}^{K-1} X_k a_{lk} \right) 2^l. \tag{2}$$

Note that $a_{L-1,k}$ denotes the most significant bit of the coefficient A_k.

There are some important features of (2) that should be noted. First, no hardware multipliers are needed, since a_{lk} takes on values 0 or 1, leaving only shifts to implement the 2^l multiplication. Second, the decomposition of A into bit-planes by equation (2) allows for simple exploitation of the symmetries and common subexpressions contained in the coefficient data. Moreover, since a_{lk} is known at compile time, we perform only the additions that are necessary during run time.

In order to generate trees from a vector product, it is useful to represent (2) in the following manner:

$$\begin{bmatrix} Y_0 \\ Y_1 \\ \dots \\ Y_{L-1} \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,K-1} \\ a_{1,0} & \dots & & \\ \dots & & \dots & \\ a_{L-1,0} & & & a_{L-1,K-1} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ \dots \\ X_{K-1} \end{bmatrix},$$

with the desired result

$$Y = Y_0 + Y_1 2^1 + \dots + Y_{L-1} 2^{L-1}.$$

Denote the coefficient matrix of a_{lk} elements for output Y as $\mathbf{a}_Y$. This matrix will be the basis of all optimizations and transformations in this paper. Note that $\mathbf{a}_Y$ consists of binary data only, with the top row containing the least significant bits of the K elements of A, and the bottom row containing their most significant bits.

The product of each row of $\mathbf{a}_Y$ with X is computed using a binary tree; if a row of $\mathbf{a}_Y$ contains one or more 0's, then the corresponding tree will be a pruned tree. Regardless, the L trees Y_0, Y_1,...Y_{L-1} are summed together using one final adder tree combined with shifts to produce the value of Y. For simplicity, our figures show a linearized version of this final tree; in practice a binary tree will often be used [3].

As an example used throughout this section, suppose we wish to compute

$$Y = X_0 + 2X_1 + 3X_2 + 4X_3 + 5X_4 + 6X_5 + 7X_6 + 8X_7.$$

In this case A=[1 2 3 4 5 6 7 8], and $\mathbf{a}_Y$ is

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3}$$

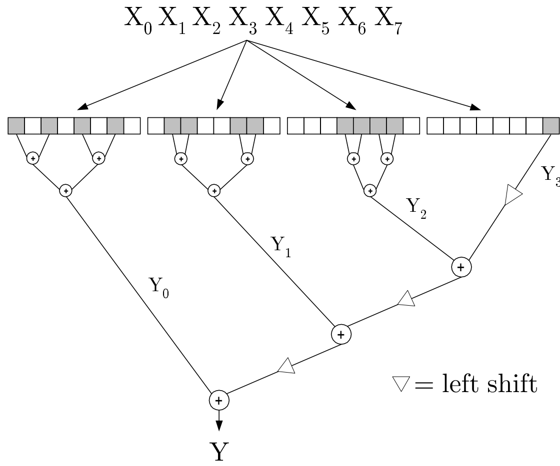


Figure 1: An unoptimized adder tree for computing a vector product. Shaded regions represent 1's in a coefficient's bitplane.

Figure 1 illustrates the unoptimized tree structure to compute this expression for an arbitrary vector X. The coefficients for the four subtrees from left to right correspond to the rows of (3) from top to bottom.

The leaves of Figure 1 are registers to hold the vector X; as illustrated there are four copies of X, one for each bitplane of coefficient data. The tree is constructed such that the shift-adds occur in order of significance: recall that the data's most significant bits, in our representation, are located at the bottom row of $\mathbf{a}_Y$. This row is shifted left one bit and added to the next row above it, with the process repeating for each row. Thus, Figure 1 computes

$$Y = X_0 + X_2 + X_4 + X_6 + \\ 2(X_1 + X_2 + X_5 + X_6 + \\ 2\,(X_3 + X_4 + X_5 + X_6 + 2\,(X_7))).$$

Note that the tree is completely specified by the data contained in $\mathbf{a}_Y$; this allows us to optimize the tree structure simply by applying transformations to the matrix $\mathbf{a}_Y$.

4 Optimizations

MCMTs rely heavily on common subexpression elimination (CSE) to optimize the adder trees. CSE is implemented easily in trees, since a common subexpression is a subtree common to two or more different trees, and is "eliminated" by sharing one subtree's output and removing the others. While non-trivial for random logic, the formulation of $\mathbf{a}_Y$ allows for the following, systematic approach.

4.1 Sharing common subexpressions

Since the four sets of eight leaves in Figure 1 contain identical values, a hardware vector multiplier would combine the four subtrees (corresponding to Y_0, Y_1, Y_2, and Y_3) into a single tree. A close inspection of Figure 1 reveals redundancies in the subtrees, such as the vertex connected to X_5 and X_6 in both Y_1 and Y_2; clearly, only one adder is necessary, and thus we label $X_5 + X_6$ as a common subexpression. This can be seen in the matrix $\mathbf{a}_Y$ in Figure 2, with the four 1's marked χ_2. Since adders are used to compute each row, whenever two or more 1's match, there is a possibility of a common subexpression.

In general, a subexpression χ is a set of locations of 1's in $\mathbf{a}_Y$, where the row and column size is at least two (since an adder needs at least two operands). A formal definition of subexpressions is now provided, concluding with the definition of a *valid* group of subexpressions.

Define the *dot product* of two sets A and B as

$$A \bullet B = \{(a, b) : a \in A \textit{ and } b \in B\}\,.$$

Given that $\mathbf{a}_Y$ is of size $L \times K$, define C to be the set of columns of $\mathbf{a}_Y$,

$$C = \{0, 1, ..., K-1\}\,,$$

and R to be the set of rows of $\mathbf{a}_Y$,

$$R = \{0, 1, ..., L-1\}\,.$$

Given that 2^R and 2^C are respectively the *power sets* of R and C, the i-th common subexpression χ_i of $\mathbf{a}_Y$ is defined as $\chi_i = R_i \bullet C_i$, where R_i and C_i are sets that meet the following conditions:

1. $R_i \in 2^R$ and $C_i \in 2^C$
2. $a_{r,c} a_{r',c'} = 1$ for all $r, r' \in R_i$ and $c, c' \in C_i$
3. $|R_i| \geq 2$ and $|C_i| \geq 2$

In Figure 2, for example, R_1={0,1} and C_1={2,6}.

Two example CSEs are illustrated in Figure 2. Both χ_1 and χ_2 contain four elements, and $|R| = |C| = 2$ for both as well. Note also that $\chi_1 \bigcap \chi_2 = \{(1, 6)\}$. In general, intersecting χ's produce valid circuits, but in this case it does not since $a_{1,6}$ would be added twice. Thus, we must alter the definition of intersection slightly in order to guarantee a correct result.

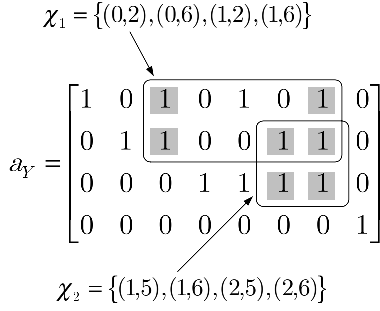


Figure 2: An example of an invalid set, since the subexpressions χ_1 and χ_2 overlap.

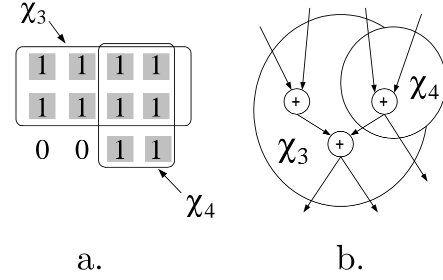


Figure 3: (a) An example of a valid set of common subexpressions. (b) The corresponding hardware implementation, with χ_4 as a subtree of χ_3.

Specifically, we define the intersection of two common subexpressions as

$$\chi_i \overset{*}{\bigcap} \chi_j = \begin{cases} R_{ij} \bullet C_{ij} & \text{if } C_i \subseteq C_j \vee C_j \subseteq C_i \\ \emptyset & \text{otherwise} \end{cases}$$

where $R_{ij} = R_i \bigcap R_j$ and $C_{ij} = C_i \bigcap C_j$. Of course, if $R_{ij} = \emptyset$ then $\chi_i \bigcap^* \chi_j = \emptyset$, which means

$$\chi_i \overset{*}{\bigcap} \chi_j \subseteq \chi_i \bigcap \chi_j. \tag{4}$$

A set of common subexpressions is *valid* if each pair of χ's in the set achieves equality in (4). For example, the two common subexpressions in Figure 2 do not form a valid set since $\chi_1 \bigcap \chi_2 = \{(1,6)\} \neq \emptyset = \chi_1 \bigcap^* \chi_2$.

In other words, $\chi_1 \bigcap^* \chi_2$ is equivalent to $\chi_1 \bigcap \chi_2$ if all of the columns of one subexpression is contained in the other; this condition corresponds to one χ's expressed as a *subtree* tree of another. An example of this case is given in Figure 3a, where $\chi_3 \bigcap \chi_4 = \{(0,2),(0,3),(1,2),(1,3)\} = \chi_3 \bigcap^* \chi_4$, and is thus a valid set. The columns of χ_3 are $C_3 = \{0,1,2,3\}$ and the columns of χ_4 are $C_4 = \{2,3\}$. Clearly, the columns of χ_4 are contained in the columns of χ_3, which dictates that χ_4 becomes a subtree of χ_3, as illustrated in Figure 3b.

4.2 Size of MCMTs

At this point we are able to evaluate the number of additions/subtractions (or simply, the number of vertices) required in an MCMT, a metric which is device independent.

Given a valid set of common subexpressions for $\mathbf{a}_Y$, χ_1, χ_2,...,χ_Z, and the number of 1's in $\mathbf{a}_Y$ as I, then the number of vertices in the MCMT for $\mathbf{a}_Y$ is

$$\begin{aligned} size_1 &= (I-1) - \sum_{i=1}^{Z} (|R_i|-1)(|C_i|-1) + \\ & \sum_{i=1}^{Z} \sum_{\substack{j=1 \\ j \neq i}}^{Z} (|R_{ij}|-1)(|C_{ij}|-1) \end{aligned} \tag{5}$$

for all i, j such that $\chi_i \bigcap^* \chi_j \neq \emptyset$. The double summation in (5) accounts for the overlap between pairs of χ's. The notation $size_1$ indicates the size of the tree for one problem instance; we will discuss multiple problem instances in a later section.

If we choose to reduce the problem represented by Figure 2 with either χ_1 or χ_2, the size of the circuit becomes:

$$(13-1) - (2-1)(2-1) + 0 = 11,$$

which is one less vertex than $\mathbf{a}_Y$ without CSE. An example of a tree reduced by sharing χ_1 is given in Figure 4, where the shaded vertex has an outdegree of 2. Thus, we have eliminated one vertex between leaves X_2 and X_6.

4.3 Shifting coefficients

The optimizations of the previous section show how vertices in the adder tree can be eliminated. We now discuss a transformation to the matrix $\mathbf{a}_Y$ which may increase the effectiveness of the optimization techniques.

For an arbitrary number x, it is possible to write

$$x = 2^s \left(2^{-s} x\right)$$

where s is considered a shift value. This implies replacing a value x by a shifted version, or equivalently, shifting the column of $\mathbf{a}_Y$ that represents x by an integer

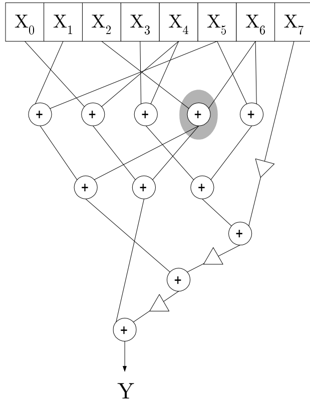


Figure 4: The tree structure with one removed vertex. The shaded vertex is reused in two subexpressions.

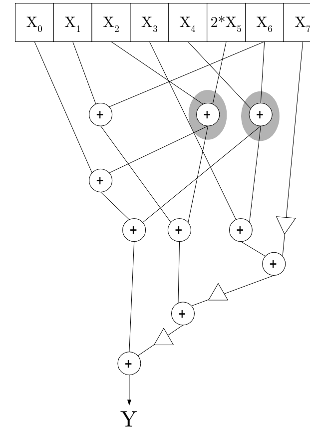


Figure 5: The final tree structure with two removed vertices, as indicated by the two shaded verticies.

amount s. For example, 6 can be written $.5 \times (2 \times 6)$, where $s = -1$. Thus, our previous example transforms $\mathbf{a}_Y$ into

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

where the sixth column is shifted up one place. Clearly, this changes the arrangement and availability of common subexpressions in $\mathbf{a}_Y$. After shifting the column corresponding to the coefficient 6, we can have the valid set χ_1={(0,2),(0,5),(1,2),(1,5)} and χ_2={(0,4),(0,6),(2,4),(2,6)}. This results in two removed nodes, as shown in Figure 5. The resulting circuit now has only 10 additions; note that the original expression has 7 additions, so only three additional adders gives us the vector multiplier.

4.4 Technology Mapping

Clearly, common subexpression elimination comes at a price when hardware implementation is concerned. Every time an adder is eliminated, the fanout of another adder is increased. The shaded nodes in Figure 5, for example, each drive two adders. Not only does increased fanout slow circuits due to increased gate capacitance, the fixed routing architecture in FPGAs makes high fanout nets even less attractive. For example, in the Xilinx XC4000X series FPGA there are two direct connect lines and 8 single length lines between CLBs. Since the two types of routing exhibit drastically different routing delays, if the fanout of an adder is increased to exceed the number of direct connect lines, the propagation delay is affected in a grossly non-linear fashion.

Without expressing the technology specific fanout characteristics, an optimizing algorithm would favor high fanout circuits due to the reduced number of adders. Fortunately, the fanout of a node in an MCMT is simply the number of rows of the χ that covers it. Thus, our optimization strategy may decide to exclude χ's whose row dimension $|R|$ is greater than a device's fanout constraints.

5 The MCMT Problem

We are now ready to formulate the multiple constant multiplier tree problem: Given a matrix $\mathbf{a}_Y$, with applied shifts, pick a set of common subexpressions such that the set is valid and the cost in (5) is minimized.

This formulation of MCMTs allows for a search strategy that is more capable than previous techniques. We have already noted that pairwise matching misses many major opportunities for optimization. This is due to MCMT's lack of an *optimal substructure*, where optimal decisions at one step may have a negative impact on the final result. Thus, any greedy algorithm may fail to produce optimal results. Moreover, the algorithms in [4] operate on the entire bit vector of a constant. Our definition of χ's does not impose any such restriction: a common subexpression may be any portion of any number of constants. It is interesting to note the similarity of the MCMT prob-

lem to the weighted set covering problem, which is known to be NP-complete [7].

The algorithm that we have developed is based on Simulated Annealing, and is described by the following pseudocode:

```
Represent constants in binary or canonical signed digit
(CSD) formats
Uniquely label all possible χ's
Define SOLUTION to be a set of χ's, starting with an
empty set
Define t = START_TEMP
While ITERATION < MAX_ITERATIONS {
    Randomly pick a χ from the set of all possible χ's
    and place into the SOLUTION set
    If the SOLUTION set is a valid set {
        Determine the size from (5)
        If size < previous_size or
        If X < exp((previous_size − size)/t), where X
        is a uniformly distributed random variable {
            Save the new SOLUTION set
        }
    Else
        Restore the previous SOLUTION set
    }
Reduce t
Increment ITERATION
}
```

In our current implementation, the rate at which t is reduced is a function of the number of common subexpressions found.

6 Multiple Independent Problems

We now turn our attention to applications where more than one vector product is required, and the multiple products can take place independently. A common example of multiple independent vector products is the matrix product. Consider the $N \times N$ matrix B and the $N \times 1$ vector Z,

$$\begin{aligned} Y &= AZ \\ &= \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,N-1} \\ a_{1,0} & \cdots & & \\ \cdots & & \cdots & \\ a_{N-1,0} & & & a_{N-1,N-1} \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ \cdots \\ z_{N-1} \end{bmatrix} \\ &= \begin{bmatrix} Y_0 \\ Y_1 \\ \cdots \\ Y_{N-1} \end{bmatrix}, \end{aligned}$$

where each Y_i is the result of a vector product as in (1); note that there is no dependence between the Y_i's. Because of this independence, there is no communication between each vector multiplier, which allows for run-time reconfiguration of the hardware with minimal overhead. Before proceeding to the topic of RTR, however, we need to add support for multiple independent problems in the preceding MCMT representation.

For each vector product $Y_i = \sum_{k=0}^{N-1} a_{i,k} z_k$, formulate each matrix $\mathbf{a}_{Y_i}$ as before, and then compose the block matrix

$$\mathbf{a}_Y = \begin{bmatrix} \mathbf{a}_{Y_0} \\ \mathbf{a}_{Y_1} \\ \cdots \\ \mathbf{a}_{Y_{N-1}} \end{bmatrix}.$$

For example, consider the product

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \\ Z_5 \\ Z_6 \\ Z_7 \end{bmatrix}$$

The matrix $\mathbf{a}_Y$ would then be

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Given this representation, we can apply the optimizations of the previous section to share χ's across the multiple vector products, with the number of vertices given by

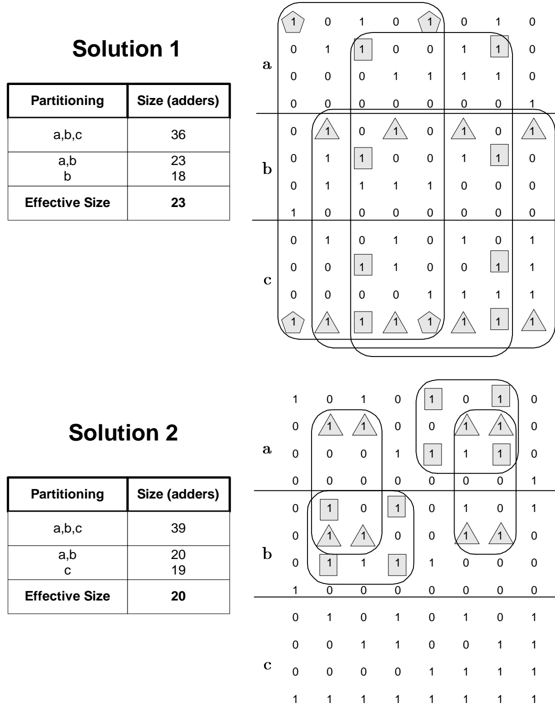
$$size_N = size_1 - (N-1)$$

for $N > 1$ independent problem instances.

7 Run-time Reconfiguration

Reconfiguring an FPGA at run time allows a designer to effectively implement circuits larger than the device itself by time-division multiplexing. This necessitates, however, the overhead of reconfiguration time and storage to allow communication between successive configurations. Two different forms of RTR are provided by FPGAs today: full reconfiguration and partial reconfiguration.

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \end{bmatrix} =$$



Solution 1

Partitioning	Size (adders)
a,b,c	36
a,b b	23 18
Effective Size	23

Solution 2

Partitioning	Size (adders)
a,b,c	39
a,b c	20 19
Effective Size	20

Figure 6: Two possible solutions to a three vector product. With no reconfigurations, solution 1 is smaller, using 36 adders. Solution 2 is smaller, however, if **a** and **b** exist alone in a configuration; such a partitioning requires just 20 adds.

For full reconfiguration, the entire FPGA is reset and reconfigured with the desired circuit. This process is usually quite lengthy, so we strive to maximize the amount of computation performed by a single configuration. Accordingly, the effective size of a fully RTR circuit is the maximum of any of its configurations. Thus we can run the MCMT algorithm until each configuration meets size constraints, or conversely, force the algorithm minimize the number of configurations. Clearly, however, subexpressions common to two different configurations cannot be eliminated. In fact, they should be combined into one configuration.

As an example, Figure 6 shows three coefficient vectors **a**, **b**, and **c**. The $\mathbf{a}_Y$ matrix is formed as before, and we proceed to run the MCMT algorithm. If the FPGA is not run-time reconfigured, then solution 1 requires 36 additions whereas solution 2 requires 39. However, by placing the coefficient vector **a** and **b** in

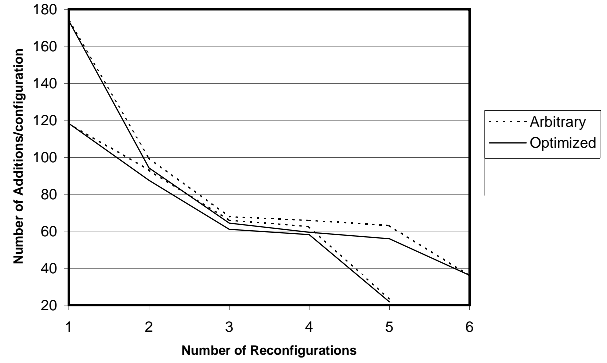


Figure 7: Arbitrary versus optimized RTR partitioning for 5 and 6 independent vector products, 8-bit precision, 8 coefficients each.

one configuration and **c** in another, we are able to obtain significantly smaller circuits; in Figure 6, solution 2 yields a smaller configuration for the chosen run-time partitioning, with only 20 additions. Note how the common subexpressions in solution 2 do not span multiple configurations. Thus, when determining the partitioning of multiple independent problems, the MCMT algorithm will group vectors into configurations according to the distribution of common subexpressions: the more χs shared by a set of vectors, the more likely they will coexist in one configuration.

Figure 7 illustrates how the partitioning selection affects the circuit size of both a 5 vector product and 6 vector product system. The dashed curve represents arbitrary partitioning of the circuit, whereas the solid curve represents an optimized partitioning choice based on the reasoning above. Clearly, run-time reconfiguration has a dramatic impact on circuit size, and an optimized partitioning scheme reduces the circuit size even further at no extra cost. Note the plateaus in the graph, caused by the reduced probability of sharing circuit resources.

The requirements of a *partially* reconfigured design, however, are quite different. For these designs, the reconfiguration time is a function of the number of circuit elements that need to be altered between successive configurations. Thus, we strive not to find minimal size partitions, but rather the best *sequence* of the N problem instances.

We propose a method of approaching the partial reconfiguration problem in the following manner. Given N matrices $\mathbf{a}_{Y_0}, \mathbf{a}_{Y_1}, \ldots, \mathbf{a}_{Y_{N-1}}$, each of size $L \times K$, we wish to reconfigure the FPGA to be each instance just once, with the minimal amount of reconfiguration

time[1]. We can rephrase this into the classic traveling salesman problem [7], where the salesman must visit each of N cities once before returning to the first, and with the shortest possible route. We measure the *distance* between two configurations as

$$d\left(a_{Y_i}, a_{Y_j}\right) = \sum_{l=0}^{L-1} \sum_{k=0}^{K-1} 1 - \left[\delta\left(a_i\left(l,k\right) - a_j\left(l,k\right)\right)\right] \quad (6)$$

where the l,k-th element of $\mathbf{a}_{Y_i}$ is denoted $a_i(l,k)$. Thus the distance between two configurations is a measure of the number of adders that differ between the leaves of their corresponding MCMTs. Clearly, we wish to minimize the total distance

$$\sum_{i=0}^{N-1} \sum_{\substack{j=0 \\ j \neq i}}^{N-1} d\left(\mathbf{a}_{Y_i}, \mathbf{a}_{Y_j}\right).$$

The optimal reconfiguration sequence can be determined using this cost metric and an appropriate traveling salesman algorithm. A given reconfiguration sequence, however, places a constraint similar to what we have previously encountered in fully reconfigured designs: subexpressions common to two different configurations cannot be eliminated if they are not consecutive configurations in the chosen reconfiguration path.

8 Results and Comparisons

This section provides some performance results of MCMT circuits, with appropriate comparisons against traditional approaches. Although the size estimates concern Xilinx 4000 series FPGAs, the same reasoning can be applied to other technologies.

Distributed arithmetic

DA is a popular method of implementing vector products in hardware. Indeed, DA is the design style of choice for Xilinx's LogiCORE FIR filters. Synthesizing a DA circuit is straightforward: pre-compute all combinations of the constant summation over A, store the values in a LUT, and use X as address bits. This mechanism makes DA a particularly attractive choice for LUT-based FPGAs. Unfortunately, storing these summations can become quite unwieldy. The size of fully parallel DA LUT storage in bits is [2]

$$2^K K\left(N + \lceil \log_2 K \rceil\right),$$

where N is the maximum bit width of X_k, and K is the number of coefficients.

Since the required memory size grows exponentially with K, implementing non-symmetric filters with a large number of taps becomes impractical with DA. Clearly, the data inside the LUT can be compacted since the words at the lower addresses contain fewer significant bits than the words at higher addresses.

As an illustration, the maximum LUT size for a single XC4000 CLB is 32 bits, so the LUT size of a parallel DA circuit in CLBs is simply

$$\left\lceil \frac{2^K K\left(N + \lceil \log_2 K \rceil\right)}{32} \right\rceil.$$

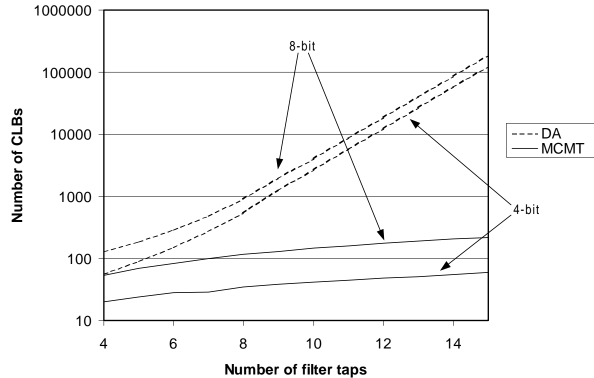


Figure 8: Comparison of DA and MCMT CLB usage for FIR filters. 4-bit and 8-bit wordlengths are illustrated.

Figure 8 shows an estimate of the number of CLBs used by the two implementations while varying the number of FIR filter taps, for both 4-bit and 8-bit wordlengths. Note how the DA circuit size for small filters is dominated by an adder tree to sum the K LUTs. Since the size of the MCMT circuit depends on the coefficients, the MCMT curves in Figure 8 represent an average of 100 randomly generated coefficient vectors. We choose simple ripple-carry adders in the adder trees to minimize CLB usage.

Constant coefficient multipliers

One can always design vector product explicitly with multipliers and adders, where the multipliers are

[1] Of course, partial reconfiguration times vary between different technologies. For simplicity, we assume that changing one circuit element and/or associated routing consumes one unit of time.

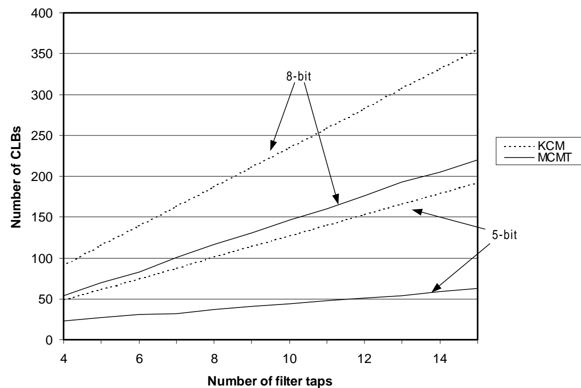


Figure 9: Comparison of KCM and MCMT CLB usage for FIR filters. 5-bit and 8-bit wordlengths are illustrated.

implemented as fixed data in look-up tables. The resource utilization of these multipliers, called KCMs, is technology dependent. For purposes of comparison, we again choose the Xilinx 4000 series FPGA. A comparison between KCM-based designs and MCMTs is shown in Figure 9 for 5-bit and 8-bit wordlengths. For each vector product of length K, there are K multipliers and $K-1$ adders; the number of CLBs required for each KCM multiplier is given in [8].

FPGA design examples

To gain insight in technology mapping and circuit performance, we choose to implement several FIR filter examples and compare the results with other design approaches. The MCMT algorithm takes the filter characteristics and produces tree topologies that minimize the total number of additions, with no adder exceeding a fanout of two and with no pipeline stage exceeding the delay of one adder. The output of the algorithm is a listing of the optimized set of χ's. The adder trees are then described in VHDL and synthesized for Xilinx XC4000 and Virtex devices; the tree design and VHDL coding are currently done by hand. The adders are simple ripple-carry designs, but do take advantage of the fast carry chains present in these FPGAs. Table 1 lists the results for two designs, a "typical" FIR filter, and a sinc compensation FIR filter.

The MCMT circuit size, and to a lesser extent its speed, are dependent on the coefficient values, whereas those of the DA circuit do not. Thus the "typical" FIR filter listed is an average of 10 randomly chosen, non-symmetric coefficient vectors of length five. The

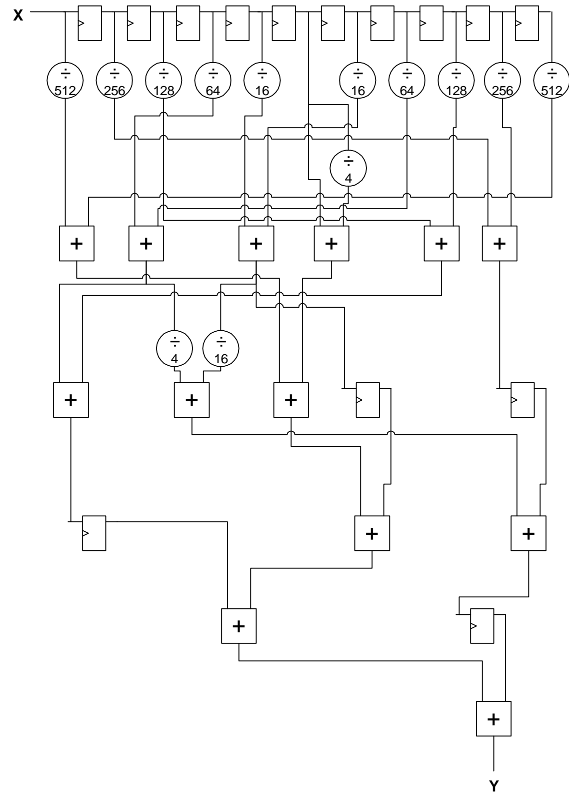


Figure 10: MCMT circuit for the sinc compensation filter. Input width is 14 bits.

performance figures for the DA implementation are obtained from [8]. Note that the MCMT circuit is pipelined at each level of the adder tree, which introduces finite latency in the filter; this latency depends on the topology produced by the algorithm.

The second design implements a symmetric 11-tap FIR filter for sinc compensation in a D/A converter, and has previously been fabricated using a 1.2-μm CMOS process [9]. This ASIC implements a transpose form FIR using canonical signed digit (CSD) multipliers, and achieves a sample rate of 112 MHz. Figure 10 illustrates the MCMT implementation. As shown in Table 1, the DA implementation is more than twice the size, and nearly half the speed of the MCMT; this is due to many symmetries and CSE in the coefficient values. The MCMT design differs from the ASIC only in its tree structure, and achieves 104 MHz on a Xilinx Virtex FPGA.

Design	Implementation Style	Technology	Size	Speed
Typical FIR 5-tap non-symmetric, 8b wordlength	Xilinx LogiCORE Parallel DA FIR	XC4036XL-1 FPGA	165 CLBs	75 MHz
	MCMT Pipelined Ripple-Carry Adder Tree	XC4036XL-1 FPGA XCV300-4 FPGA	78 CLBs 80 SLICEs	84 MHz 100 MHz
Sinc Compensation FIR 11-tap symmetric, 14b wordlength	Transposed Form FIR with CSD multipliers	1.2um CMOS ASIC	12500 transistors	112 MHz
	Xilinx LogiCORE Parallel DA FIR	XC4036XL-1 FPGA	516 CLBs	47 MHz
	MCMT Pipelined Ripple-Carry Adder Tree	XC4036XL-1 FPGA XCV300-4 FPGA	220 CLBs 242 SLICEs	74 MHz 104 MHz

Table 1: Comparison of different designs, implementations, and technologies.

9 Concluding Remarks

We have presented a novel design methodology and its implementation for computing vector products in hardware. Our optimization procedure covers a wider solution space than previous approaches, and can take advantage of run-time reconfigurability and technology-specific features of advanced FPGAs. In the future we plan to explore partial reconfiguration for MCMT designs still further, and express better control over circuit routability. Continued work on mapping various applications to new architectures, such as Xilinx's Virtex parts, should provide greater improvements in circuit utilization and performance.

Acknowledgments

This work is supported by DARPA/ITO under contract number DABT 63-95-C-0102 and by the UK Engineering and Physical Sciences Research Council (Grant number GR/24366, GR/54356 and GR/59658).

References

[1] J. Villasenor, B. Schoner, K. Chia, C. Zapata, H.J. Kim, C. Jones, S. Lansing and W. Mangione-Smith, "Configurable computing solutions for automatic target recognition," in *Proc. FCCM96*, 1996, pp. 70–79.

[2] A. Sinha and M. Mehendale, "Improving area efficiency of FIR filters implemented using distributed arithmetic," in *Proc. Eleventh International Conference on VLSI Design*, 1997, pp. 104–109.

[3] M. Mehendale, G. Venkatesh and S.D. Sherlekar, "Optimized code generation of multiplication-free linear transforms," in *33rd Design Automation Conference*, 1996, pp. 41–46.

[4] 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*, vol. 15, no. 2, 1996, pp. 151–165.

[5] A.G. Dempster and M.D. Macleod, "Use of minimum-adder multiplier blocks in FIR digital filters," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 42, no. 9, 1995, pp. 569–577.

[6] B. Fehér, "Efficient synthesis of distributed vector multipliers," in *Proc. Euromicro'93, 19th Symposium on Microprocessing and Microprogramming*, vol. 38, no. 1–5, 1993, pp. 345–350.

[7] T.H. Cormen et. al., *Introduction to Algorithms*, Cambridge: MIT Press, 1990.

[8] *CORE Solutions Data Book*, Xilinx, Inc., San Jose, 1998.

[9] R. Jain, P.T. Yang and T. Yoshino, "FIRGEN: a computer-aided design system for high performance FIR filter integrated circuits," *IEEE Transactions on Signal Processing*, vol. 39, no. 7, 1991, pp. 1655–1668.