Resource Optimal Design of Large Multipliers for FPGAs

Martin Kumm*, Johannes Kappauf*, Matei Istoan† and Peter Zipf*
*University of Kassel, Digital Technology Group, Germany
†University Lyon, Inria, INSA Lyon, CITI, France

kumm@uni-kassel.de, johannes.kappauf@student.uni-kassel.de,
matei.istoan@inria.fr, zipf@uni-kassel.de

***Abstract*—This work presents a resource optimal approach for the design of large multipliers for FPGAs. These are composed of smaller multipliers which can be DSP blocks or logic-based multipliers. A previously proposed multiplier tiling methodology is used to describe feasible solutions of the problem. The problem is then formulated as an integer linear programming (ILP) problem which can be solved by standard ILP solvers. It can be used to minimize the total implementation cost or to trade the LUT cost against the DSP cost. It is demonstrated that although the problem is NP-complete, optimal solutions can be found for most practical multiplier sizes up to 64×64. Synthesis experiments on relevant multiplier sizes show slice reductions of up to 47.5% compared to state-of-the-art heuristic approaches.**

***Keywords*-multiplier, FPGA, optimization, ILP**

I. Introduction

Multiplication is one of the basic building blocks inside many higher-order arithmetic operations like, e. g., division or function approximation and used in many applications. Therefore, it is of utmost importance to have the best possible implementation for the multiplication. Any improvement here translates to gains in any application that involves multiplications.

While most concepts known from computer arithmetic can be mapped to FPGAs, specialized multiplication architectures largely exploit the low-level properties of modern FPGAs [1]–[9]. A Baugh-Wooley multiplier that effectively uses the look-up-tables (LUTs) as well as the fast carry-chains of FPGAs was presented in [3]. The idea of using the 6-input LUTs of current generation devices for implementing 3×3-bit multiplications was proposed in [5] and further evaluated in [7]. A similar idea was followed in [9]. In the work described above, compressor trees are used to add the partial products from the LUT-based multipliers. For that, a couple of methods exist for the design and optimization of compressor trees [5], [10]–[12]. The compressor trees are avoided in the architectures which combine partial product generation with subsequent addition and make them fit in the resources available in a Xilinx FPGA slice [6]–[8]. This avoidance of compressor trees makes the multiplier more compact but is potentially slower or requires a higher latency than fast compressor trees.

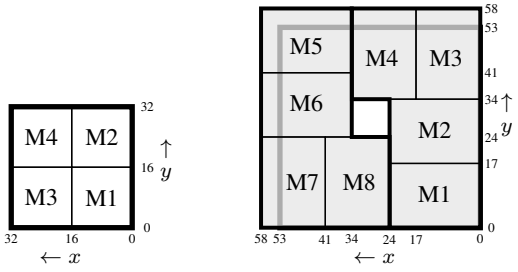
Aside from LUTs, FPGAs provide direct support for multiplications through the embedded multiplier blocks (DSPs). On recent Xilinx devices, the DSPs support signed multiplications of up to 25×18 bit (or 24×17 bit unsigned) using a single DSP block. On Intel FPGAs, their latest Stratix 10 device provides so-called variable precision DSP blocks which can be configured as two independent 18×18 multipliers or a single 27×27 multiplier (both signed or unsigned).

However, if the size of the required multiplier does not fit into the DSP block, it can be split into smaller ones with sizes tailored according to the available resources of the target device, i. e., the embedded multipliers/DSPs and specific logic-only multipliers [3]–[9]. The basic rules for this kind of splitting are simple and well known [13]. A more (resource) efficient splitting can be performed by the Karatsuba-Ofman algorithm [14]. Here, additional additions/subtractions are used to reduce the number of multipliers at the cost of a longer critical path. However, it was pointed out in [1] that the method is less suitable for non-square multipliers which is the case for the DSP blocks of many modern FPGAs. For this case, the authors in [1] introduced a graphical *multiplier tiling* methodology which was further refined in [2]. Large multipliers were also treated in [4]. However, they used a rather straight-forward splitting as well as simple counters and ternary adders in their compressor tree implementations.

The underlying tiling problem has attracted much attention in various other fields from mathematics/computer science (geometry, combinatorics, complexity theory) to industrial processes. This related work is reviewed in Section III. While the tiling methodology defines the rules for valid splittings of a large multiplier into smaller ones, it is highly ambiguous which splitting delivers the best results. The contribution of this work is a method that delivers a resource optimal multiplier tiling by using integer linear programming (ILP). In the following, a brief introduction to the underlying multiplier tiling methodology is given.

II. Multiplier Tiling

For the realization of a large multiplication using smaller multipliers, the operands can be divided into smaller words. Consider a large multiplication with operands A and B using



(a) 32×32 mult. example (b) 53×53 mult. [1]

Figure 1: Tilings for realizing different multipliers

X and Y bit, respectively, denoted as a $X \times Y$ multiplication in the following. By splitting A and B into two smaller words, where A_L and B_L denote the n least significant bits and A_H and B_H the remaining high significant bits, respectively, the multiplication can be represented as follows:

$$\begin{aligned} A \times B &= (A_\mathrm{H} 2^n + A_\mathrm{L})(B_\mathrm{H} 2^n + B_\mathrm{L}) \\ &= \underbrace{A_\mathrm{H} B_\mathrm{H}}_{\mathrm{M4}} 2^{2n} + \underbrace{A_\mathrm{H} B_\mathrm{L}}_{\mathrm{M3}} 2^n + \underbrace{A_\mathrm{L} B_\mathrm{H}}_{\mathrm{M2}} 2^n + \underbrace{A_\mathrm{L} B_\mathrm{L}}_{\mathrm{M1}} \end{aligned} \quad (1)$$

Hence, the multiplication is divided into four smaller multiplications (labeled M1...M4) and three additions. The result can be graphically represented as a rectangular board of size $X \times Y$, which is tiled by smaller rectangles representing the smaller multipliers [1]. The tiling of the example above is shown in Figure 1(a) for a 32×32 multiplication using $n = 16$ bit multipliers. It can be generalized such that any complete tiling of the board with non-overlapping tiles represents a valid solution [1]. This can be used as a design method which was introduced in [1] as *multiplier tiling* method and was further refined with an optimization heuristic in [2], [5]. In the FPGA context, the small multipliers are either realized using DSP blocks or as logic-based multipliers. The corresponding bit shifts that have to be performed before addition can be directly read out from this graphical representation. A multiplier placed at position (x, y) has to be shifted by $x + y$ bits to the left in the final sum. For example, the multiplier $M4$ in Figure 1(a) is located at coordinates $(16, 16)$, hence, its result has to be shifted by $16+16 = 32$ bits which is also the result obtained in (1).

Figure 1(b) shows a more complex example of an unsigned 53×53 bit multiplier as required in a double precision multiplication [2]. Here, the gray rectangles correspond to the 24×17-bit unsigned multiplications available in the Xilinx DSP48E(1) blocks while the white square is realized as logic-based multiplier. This example also shows how another feature of the target FPGA devices can be exploited: the internal post-adders contained in the DSP blocks, which can be used for summing-up the results of several multipliers. The multipliers M1...M4 as well as multipliers M5...M8 can be combined in such a way. They are indicated by the bold frames in Figure 1(b) and form so-called *super-tiles* [1].

From a timing point of view, all multiplications run in parallel and parallel compressor trees can be used to sum their results. Here, the resource reduction using super-tiles comes at the cost of either additional delay or latency (when pipelined). Hence, a trade-off between resources and delay/latency can be made by choosing the size of the super-tiles.

The considered problem is closely related to the generic tiling problem where a given shape (quadratic, rectangular, etc.) has to be tiled from a set of tiles with different shapes (quadratic, rectangular, polyomino, etc.). A review on related work is given in the following.

III. Related Work on the Tiling Problem

The tiling problem has attracted much attention in various fields from mathematics/computer science (geometry, combinatorics, complexity theory) to industrial processes. A very comprehensive survey on the tiling problem is given in [15], where it is considered as a *packing* and *cutting* problem. The works surveyed span over five decades and various fields, ranging from methods for solving the problem, to the analysis of its complexity and some possible applications.

In terms of complexity, variations of the tiling problem have been shown to be *NP-complete* [16] [17]. In these works, the problem consists of covering a square board with a given set of smaller square tiles. The tiles have letters on each corner and there are rules as to which tiles can be placed one next to the other. Tiling a multiplier using square multiplier blocks is a generalization of the problem of [16] and [17]. Hence, the problem can be regarded as NP-complete, too.

In [18] the authors use tiles that have colors along the edges for tiling a square. They propose the use of the tiling problem as the *master reduction*, a proof for establishing the NP-completeness of some combinatorial problem. In [19] it is shown that tilings using trominos and tetrominos (L-shapes tiles) are also NP-complete. The supertiles introduced above can be modeled as trominos.

It is shown in [20] that looking at tiling as a *partitioning problem* is also NP-complete. Their formulation is the partitioning of the unit square into a set of rectangles of given dimensions. The tiling of a multiplier is a harder problem: the total number of tiles to be used is not known in advance, and the sizes of the board and the tiles are integer numbers.

IV. Proposed Multiplier Tiling

Unfortunately, the tiling problems discussed above do not exactly fit to the multiplier tiling problem which is described more formally in the following.

A. Problem Formulation

Consider an $X \times Y$ multiplier which is represented as a rectangle of size $X \times Y$ and considered as the *large* multiplier M. Special case multipliers like truncated multipliers or squarers have a different shape as pointed out in [2]. To allow arbitrary shapes within the boundaries of $X \times Y$, the binary constants $M_{x,y}$ ($0 \leq x \leq X$, $0 \leq y \leq Y$) are defined which are true within the shape of the large multiplier.

To implement the large multiplier, we assume to have a set of *small* multipliers $\mathcal{S} = \{m_0, m_1, \ldots, m_{S-1}\}$. Each small multiplier m_s is represented as a tile with a different shape. A small multiplier can be a DSP block, a super-tile of DSP blocks or a LUT-based multiplier. As the shape can be different from a rectangular shape, the binary constants $m^s_{x,y}$ are defined to be true within the shape s of the small multiplier. Further, we assume that each small multiplier is associated with a cost value cost_s that aggregates the corresponding LUT or DSP costs. The related optimization problem can now be formulated as follows:

Multiplier Tiling Problem: Given a shape $M_{x,y}$ of the large multiplier and set $\mathcal{S}$ of small multipliers, each associated with cost$_s$, find a tiling with minimal cost such that each position (x, y) for which $M_{x,y} = 1$ is covered by exactly one small multiplier $m_s \in \mathcal{S}$.

This definition avoids overlaps of small multipliers which would lead to incorrect results but allows that a small multiplier may overlap with the border of the board. The small multipliers typically consist of different resources (LUTs/DSPs) which makes it difficult to assign a scalar cost value. As DSPs are very specialized and limited resources, it may be more practical to just constrain the number of DSP blocks and to minimize the remaining logic. This problem is referred to as *DSP constrained multiplier tiling*.

In the following, we will give an ILP formulation for the multiplier tiling problem which is then extended to the DSP constrained multiplier tiling problem.

B. ILP Formulation of the Multiplier Tiling

To describe a solution of the problem, the binary ILP variables $d^s_{x,y}$ are introduced which are true when a small multiplier with shape index s is located at position (x, y) on the board. With that, we can describe the ILP formulation for the tiling problem as follows:

$$\text{minimize} \sum_{s=0}^{S-1} \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} \text{cost}_s d^s_{x,y}$$

subject to

$$\left.\sum_{s=0}^{S-1} \sum_{x'=0}^{x} \sum_{y'=0}^{y} m^s_{x-x',y-y'} d^s_{x',y'} = 1 \right\} \begin{array}{l} \text{for } 0 \leq x \leq X, \\ 0 \leq y \leq Y \\ \text{with } M_{x,y} = 1 \end{array}$$

The objective is to minimize the costs of the used small multipliers which correspond to the sum of all $d^s_{x,y}$ variables weighted by their cost. The constraints in the ILP formulation ensure that the complete shape of $M_{x,y}$ is covered without overlap. To illustrate the constraints consider Figure 2, where a single small multiplier with shape index $s = 0$ is placed on a 6×6 board defined by $M_{0\ldots5,0\ldots5} = 1$. The shape of the small multiplier is described by the five nonzero constants $m^0_{0,0\ldots2} = m^0_{1,0\ldots1} = 1$, all other $m^0_{x,y}$ are zero. In the example, the small multiplier is placed at coordinate $(1, 2)$. As illustrated in Figure 2, there are exactly five coordinates, all lying in the area of the small multiplier, for which the left hand side of the ILP constraint is equal to one. In case no multiplier is located at (x, y), the result would be zero while in case that several multipliers overlap at coordinate (x, y), the result would be > 1. Both of these illegal cases are excluded by the constraints.

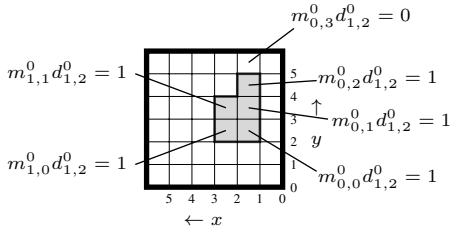


Figure 2: Example placement of a single non-rectangular multiplier tile

Due to the pure binary data type of the variables, the formulation belongs to the class of binary ILP (BILP) problems. It can be solved by any of the numerous open source or commercial ILP solvers.

C. DSP Constrained Multiplier Tiling

The extension of the ILP formulation above to the DSP constrained multiplier tiling is straightforward. Consider that D^s specifies the number of DSP blocks that are contained in a small multiplier of shape s. Then, the following constraint is sufficient to limit the number of DSP blocks to exactly #DSP:

$$\sum_{s=0}^{S-1} \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} D^s d^s_{x,y} = \#\text{DSP}$$

Similarly, a $\leq$ relation can be used to limit the DSP count to at most #DSP.

V. Considered Small Multipliers

A crucial step for the proposed methodology is the selection of the multipliers that are considered in the optimization (the set $\mathcal{S}$). These can be LUT-based or DSP-based (single DSPs or super-tiles). Their selection is strongly device dependent. Clearly, the optimization result can only be as good as these multipliers. We will focus in the following on the latest Xilinx FPGAs that provide six-input LUTs and DSP48E1 slices, namely the Virtex 6, Spartan 6 and the 7 series. However, it is straightforward to apply the same considerations to other FPGA vendors.

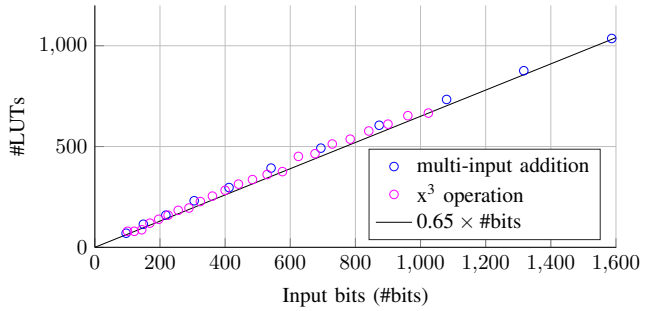


Figure 3: LUT requirements of a compressor tree for a given number of input bits to compress

A. Evaluation of Multiplier Cost

The cost contribution of a given multiplier is at least twofold: First, resources (LUTs/DSPs) are required to implement the multiplier itself. These costs are easy to obtain. Second, each multiplier contributes with a partial product. To get the final result, all partial products have to be added which requires additional resources. For that, the LUT costs that are associated with the compressor tree are required. As the compressor tree is typically obtained from a separate optimization process [10]–[12], without combining both optimization problems they can only be estimated. To do so, we performed several compressor tree optimization experiments. The compression was performed using a compressor tree algorithm based on the heuristic of [10] together with the advanced compressors of [12], which we integrated into the FloPoCo arithmetic core generator [21]. As the shape of the compressor tree can influence the cost, we decided to synthesize two cases: Equally weighted multi-input additions (FloPoCo operator `IntAdderTree`) and compressor trees obtained from an x^3 operation (FloPoCo operator `IntPower`). While the first has a *dot representation* [13] of a rectangular shape, the second has a more gaussian-like distribution as in the considered multiplier. The resulting LUT counts over the number of input bits are plotted in Figure 3.

It can be observed that the LUT cost per bit follows a fairly linear trend of about 0.65 LUTs per input bit of the compressor tree (straight line in Figure 3) which is independent of the completely different shapes of the dot representation. Hence, we weighted the cost for each produced bit by 0.65 in the objective.

B. Efficiency Metric

To rate the quality of a LUT-based multiplier, an *efficiency* metric is introduced. Similar to the efficiency metric that was used for compressor trees [5] it is defined for each shape s as a benefit-cost ratio by dividing the tile-area of the multiplier (i. e., the geometric area on the multiplier board) by the cost:

$$E_s = \frac{\text{area}_s}{\text{cost}_s} \tag{2}$$

Table I: LUT-based multipliers for Xilinx FPGAs

Shape	Tile area	Word size (w_s)	#LUT$_\text{m}$	Total cost (cost$_s$)	Efficiency (E_s)
1×1	1	1	1	1.65	0.625
1×2	2	2	1	2.3	0.87
2×3	6	5	3	6.25	0.96
3×3	9	6	6	9.9	0.91
$2\times k$	$2k$	$k+2$	$k+1$	$1.65k+2.3$	$\frac{2k}{1.65k+2.3}$

C. LUT-based Multipliers

The list of LUT-based multipliers used in this work is given in Table I. The number of LUTs to implement the multipliers is denoted as #LUT$_\text{m}$, the total costs are obtained by taking the output bits w_s into account as discussed above, using

$$\text{cost}_s = \#\text{LUT}_\text{m} + 0.65 w_s \ . \tag{3}$$

Note that the output word size may be smaller when the multiplier overlaps the border of the board which has to be considered in the cost function.

Some of these LUT-based multipliers were used in previous work. The idea to directly tabulating the results of a 3×3-bit multiplication into 6-input LUTs (LUT6) was proposed in [5]. As they produce six output bits, exactly six LUT6 are required. Later, also 2×3-bit and 1×4-bit multiplications were considered which can be mapped to only three and two LUT6, respectively, thanks to the capability of computing two 5-input LUTs within one LUT6 [7]. In addition to these, a 1×1 and a 1×2 multiplier are used in this work. The 1×1 multiplier corresponds to a single AND gate and consumes one LUT. It was added to fill possible gaps during tiling to guarantee a solution. The 1×2 multiplier can be mapped to a single LUT6. A 1×4 multiplier can be built from two 1×2 multipliers with the same cost and same efficiency. As a 1×2 multiplier is more generic, the 1×4 multiplier was not considered. In [9], a 4×2 multiplier was used in addition to 3×3 multipliers. As the 4×2 multiplier has a low efficiency of about $E=0.8$ and its shape can be covered by combining a 3×2 multiplier and a 1×2 multiplier with a higher efficiency, it was also not considered. Note that all multipliers in Table I can be also flipped, e. g., a 2×3 multiplier can be used as 3×2 multiplier, leading to additional shapes.

Besides the LUT-only realizations discussed so far, a multiplier can also exploit the fast carry chain capabilities of modern FPGAs. In the multiplier described in [3], it was proposed to implement two rows of a Baugh-Wooley multiplier in a single LUT stage which is followed by the carry-chain. The result is the sum of the two partial products which is already compressed into a single bit vector. Figure 4 shows the corresponding mapping to a Xilinx slice. The two AND gates in the LUTs compute a partial product each. Two

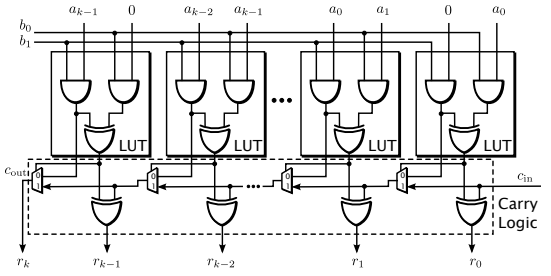


Figure 4: Slice mapping of $2 \times k$ multiplier [3], [7]

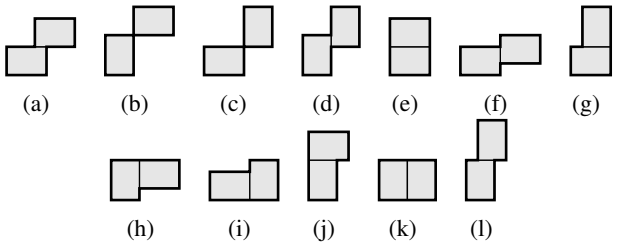


Figure 5: All super-tiles consisting of two DSP blocks

rows of partial products are added in the ripple-carry adder which is built from the XOR gates and the multiplexers as given in Figure 4. Its corresponding tile has a $2 \times k$ shape and its cost and efficiency depends on the row size k as shown in Table I. A relatively small k of 6 leads to an efficiency of $E = 0.98$ which is higher than all the other LUT-based multipliers. For $k \to \infty$ it reaches $E = 1.21$.

D. DSP-based Multipliers and Super-Tiles

The DSP block of the considered Xilinx FPGAs provides multipliers with up to 17×24 bit (unsigned) and pre/post-adders. They can be cascaded in such a way that the most significant bits (MSBs) of the product can be fed into the post adder of another DSP block, optionally shifted by 17 bit. Using that, the post adders can be exploited by constructing super-tiles [1]. Two DSP blocks can be aggregated into a super-tile when the sum of the (component wise) difference of the coordinates results in either 0 (no shift) or 17. Of course, this process can be repeated to built larger super-tiles. However, as the DSPs are connected in sequence the delay or latency (in case of pipelined DSPs) increases with the size of the super-tile. In our results, we use all possible super-tiles which comprise two DSP blocks, which turned out to be a good trade-off point. The resulting shapes are illustrated in Figure 5. The super-tiles in Figure 5(a)–5(d) have no shift and an output word size of $w_s = 17 + 24 + 1 = 42$, the remaining super-tiles include a 17 bit shift and produce an output word of size $w_s = 17 + 17 + 24 = 58$. To be able to find solutions where only a part of a DSP is used inside the board (like in pinwheel shaped tilings), we also included DSPs of size 4×4 to 17×17.

Table II: Optimization results for different number of DSP blocks starting from a DSP-only solution

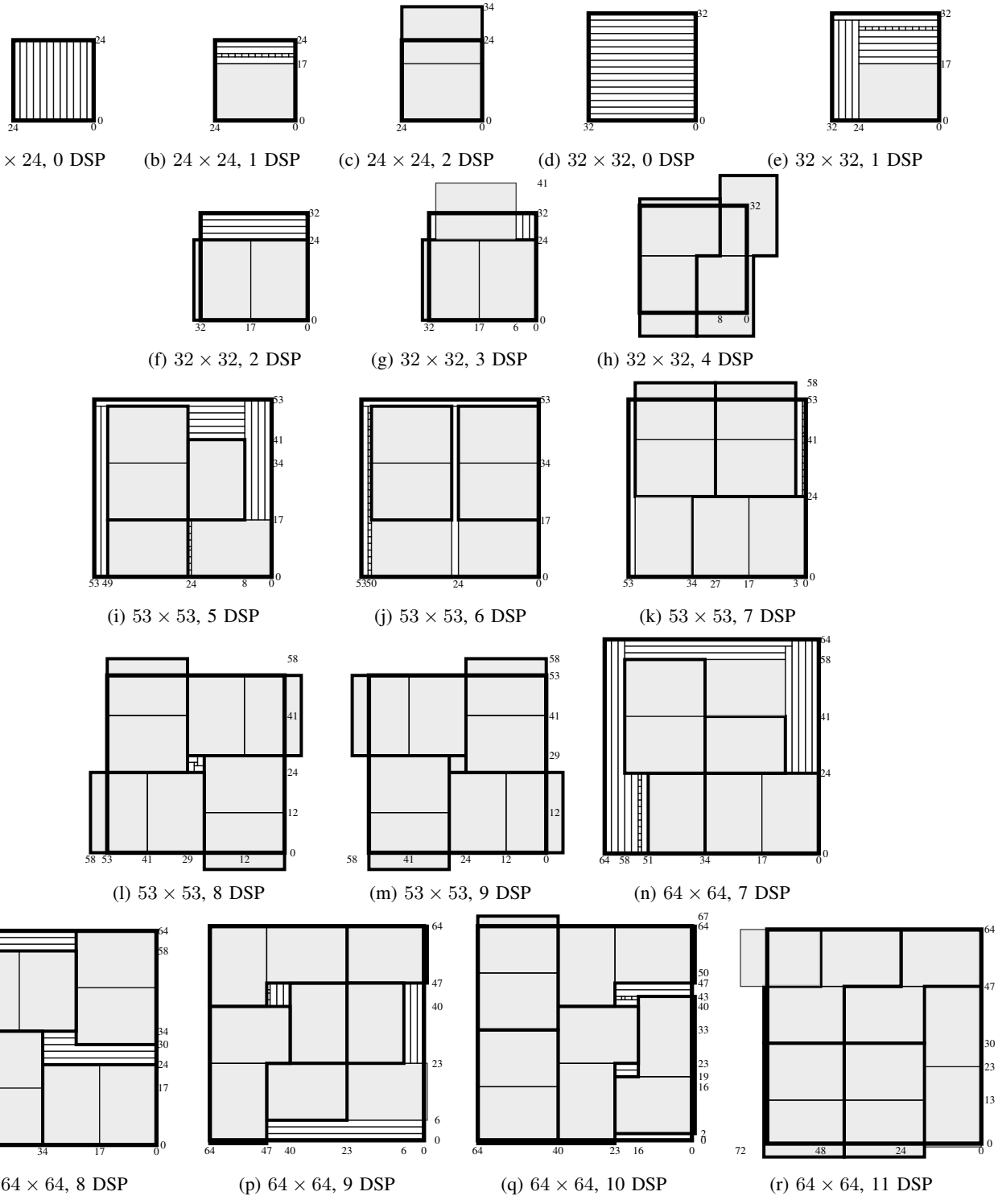
24×24 (single precision floating point)					
#DSP	2	1	0		
LUT cost	31.2	179.95	502.8		
ΔLUT	–	148.75	322.85		
CPU [s]	22.7	129	8		
32×32 (unsigned)					
#DSP	4	3	2	1	0
LUT cost	57.85	119.2	256.8	567.95	881.6
ΔLUT	–	61.35	137.6	311.15	313.65
CPU [s]	146	320	187	382	19
53×53 (double precision floating point)					
#DSP	9	8	7	6	5
LUT cost	144.3	164.45	307	450.5	759.7
ΔLUT	–	20.15	142.55	143.5	309.2
CPU [s]	1433	701	4331	2112	27215
64×64 (unsigned)					
#DSP	11	10	9	8	7
LUT cost	198.25	354.8	570.7	862.5	1192.35
ΔLUT	–	156.55	215.9	291.15	329.9
CPU [s]	43031	81149	21382	54001	TO

VI. Results

We performed several experiments by optimizing common multiplier sizes with varying numbers of DSP blocks, following the DSP constrained multiplier tiling approach of Section IV-C. The considered multiplier sizes were 24×24 and 53×53 which are required in single/double precision multipliers, respectively, as well as 32×32 and 64×64 unsigned integer multipliers.

As the most interesting solutions should utilize the DSPs as much and as efficient as possible, we evaluated the multipliers for maximum DSP count and gradually reduced the DSP count in up to five steps.

The experiments were performed on an Intel Xeon E5-2650v3 machine with 20 cores running at 2.30GHz. The ILP solver gurobi 6.0.0 (http://www.gurobi.com) was used (limited to 4 threads). The optimization results and optimization times are listed in Table II. Optimal solutions were found for all problems except the 64×64 case with 7 DSPs, where a timeout (TO) of 24 hours was exceeded. The resulting optimal tilings are given in Figure 6. Obviously, the proposed method allows to effectively trade between LUT and DSP resources. The row ΔLUT in Table II shows the LUT cost to replace one DSP. Clearly, for a high DSP count it is relatively cheap to replace a DSP block by LUTs. This can be explained by the overlaps that appear for larger DSP counts (see Figure 6). We made a separate experiment to tile a single DSP using LUT multipliers leading to LUT costs of 362.8. This limit is roughly approached for lower DSP counts. It is interesting to note that the LUT-only multipliers are identical to the multipliers proposed in [3]. Another interesting observation is that the LUT-multiplier used in



(a) 24×24, 0 DSP (b) 24×24, 1 DSP (c) 24×24, 2 DSP (d) 32×32, 0 DSP (e) 32×32, 1 DSP

(f) 32×32, 2 DSP (g) 32×32, 3 DSP (h) 32×32, 4 DSP

(i) 53×53, 5 DSP (j) 53×53, 6 DSP (k) 53×53, 7 DSP

(l) 53×53, 8 DSP (m) 53×53, 9 DSP (n) 64×64, 7 DSP

(o) 64×64, 8 DSP (p) 64×64, 9 DSP (q) 64×64, 10 DSP (r) 64×64, 11 DSP

Figure 6: Optimal tiling results of the considered multipliers. DSPs are shown in grey, LUT-Based multipliers are shown in white. DSPs that form super-tiles are surrounded by thick lines.

Table III: Comparison with previous approaches

Mult.	Method	#DSP	Area (geom.) logic mult.	Slices	Slice red.	f_{clk} [MHz]
24×24	[5]	1	216	65		212.4
	prop.	1	168	58	10.8%	287.4
	[5]	2	0	0		418.9
	prop.	2	0	0	0.0%	418.9
32×32	[2]	0	1024	339		275.8
	prop.	0	1024	276	18.6%	304.4
	[5]	1	648	205		192.8
	[2]	1	616	234		352.6
	prop.	1	616	180	12.2%	302.5
	[5]	2	288	94		270.1
	prop.	2	256	82	12.8%	338.0
	[5]	3	135	75		194.0
	[2]	3	176	75		426.6
	prop.	3	64	44	41.3%	314.5
	[5]	4	0	17		314.7
	[2]	4	40	38		379.4
	prop.	4	0	13	23.5%	181.7
53×53	[2]	5	1029	350		298.2
	prop.	5	769	295	15.7%	313.2
	[5]	6	468	196		214.1
	[2]	6	721	220		298.2
	prop.	6	361	180	8.2%	263.2
	[2]	7	313	223		378.9
	prop.	7	193	137	38.6%	290.2
	[2]	8	265	145		356.4
	prop.	8	25	81	44.1%	272.7
	[5]	9	162	125		195.6
	[2]	9	215	174		255.8
	prop.	9	0	72	42.4%	348.8
64×64	[2]	7	1504	614		245.0
	prop.	7	1191	430	30.0%	270.5
	[5]	8	1188	420		194.2
	[2]	8	1096	449		280.7
	prop.	8	652	348	17.1%	261.2
	[2]	9	864	413		262.9
	prop.	9	475	217	47.5%	249.6
	[2]	10	592	341		250.7
	prop.	10	187	179	47.5%	267.7
	[5]	11	270	196		162.8
	[2]	11	592	268		225.3
	prop.	11	0	108	44.9%	265.4

the hand-optimized 53×53 multiplier proposed in [1] (see Figure 1(b)) could be reduced from a 10×10 multiplier to a 5×5 as shown in Figure 6(l).

We compared our designs with two state-of-the-art multiplier tiling methodologies [2], [5]. The automatic heuristic results of [2] and [5] were obtained using FloPoCo 2.3.2 and FloPoCo 4.0, respectively [21]. In the generated circuits using the proposed method, each logic-based multiplier and DSP is followed by a pipeline stage whereas the super-tiles include two pipeline stages. A compressor tree algorithm based on the heuristic of [10] using GPCs from [12] was used in the proposed designs. The results are listed in Table III showing the geometric area of the remaining LUT-based multiplier on the board as a high-level measure as well as synthesis results obtained from Xilinx ISE 13.4 for a Virtex 6 FPGA (XC6VLX760). Registers were placed at inputs and the output to obtain realistic timing results. These registers are not counted in the slice resources. Column 'Slice red.' shows the percentage slice reduction compared to the best previous result with same DSP count. It can be observed that significant reductions of the geometric area of the remaining logic-based multiplier can be achieved which directly translates to significant slice reductions at a comparable speed.

VII. Conclusion and Outlook

An optimal ILP-based method was presented that is able to find resource optimal multiplier tilings for practical multiplier sizes. It can be used to effectively combine the resources available on an FPGA and allows to trade between logic resources and DSPs. Previous results could be significantly improved. Future work is directed towards the full integration of the proposed method into the core generator FloPoCo [21]. Further extensions could be directed to the limitation of the compressor tree depth to minimize its delay.

References

[1] F. de Dinechin and B. Pasca, "Large Multipliers with Fewer DSP Blocks," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*, 2009, pp. 250–255.

[2] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, "Multipliers for Floating-Point Double Precision and Beyond on FPGAs," *SIGARCH Computer Architecture News*, vol. 38, no. 4, pp. 73–79, Sep. 2010.

[3] H. Parandeh-Afshar and P. Ienne, "Measuring and Reducing the Performance Gap between Embedded and Soft Multipliers on FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2011, pp. 225–231.

[4] S. Gao, D. Al-Khalili, N. Chabini, and P. Langlois, "Asymmetric Large Size Multipliers with Optimised FPGA Resource Utilisation," *Computers & Digital Techniques, IET*, vol. 6, no. 6, pp. 372–383, 2012.

[5] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic Core Generation Using Bit Heaps," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*, 2013, pp. 1–8.

[6] E. G. Walters, "Partial-Product Generation and Addition for Multiplication in FPGAs with 6-Input LUTs," *Asilomar Conference on Signals, Systems and Computers*, pp. 1247–1251, 2014.

[7] M. Kumm, S. Abbas, and P. Zipf, "An Efficient Softcore Multiplier Architecture for Xilinx FPGAs," in *IEEE Symposium on Computer Arithmetic (ARITH)*, 2015, pp. 18–25.

[8] E. Walters, "Array Multipliers for High Throughput in Xilinx FPGAs with 6-Input LUTs," *Computers, MDPI*, vol. 5, no. 4, pp. 1–25, Sep. 2016.

[9] A. Kakacak, A. E. Guzel, O. Cihangir, S. Gören, and H. F. Uğurdağ, "Fast Multiplier Generator for FPGAs with LUT based Partial Product Generation and Column/Row Compression," *Integration, the VLSI Journal, Elsevier*, pp. 147–157, Dec. 2016.

[10] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Efficient Synthesis of Compressor Trees on FPGAs," in *Asia and South Pacific Design Automation Conference (ASPDAC)*. IEEE, 2008, pp. 138–143.

[11] H. Parandeh-Afshar, A. Neogy, P. Brisk, and P. Ienne, "Compressor Tree Synthesis on Commercial High-Performance FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 4, pp. 1–19, Dec. 2011.

[12] M. Kumm and P. Zipf, "Pipelined Compressor Tree Optimization Using Integer Linear Programming," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2014, pp. 1–8.

[13] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[14] A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet Physics Doklady*, vol. 7, p. 595, Jan. 1963.

[15] P. E. Sweeney and E. R. Paternoster, "Cutting and Packing Problems: A Categorized, Application-Orientated Research Bibliography," *Journal of the Operational Research Society*, vol. 43, no. 7, pp. 691–706, 1992.

[16] L. A. Levin, "Problems, Complete in "Average" Instance," in *ACM Symposium on Theory of Computing*, 1984, p. 465.

[17] ——, "Average Case Complete Problems," *SIAM Journal on Computing*, no. Chapter 26, pp. 106–106, 1987.

[18] P. van Emde Boas and M. W. P. Savelsbergh, "Bounded Tiling, an Alternative to Satisfiability?" in *Mathematical Research*, 1984, pp. 354–363.

[19] C. Moore and J. M. Robson, "Hard Tiling Problems with Simple Tiles," *Discrete & Computational Geometry*, vol. 26, no. 4, pp. 573–590, 2001.

[20] Beaumont, Boudet, Rastello, and Robert, "Partitioning a Square into Rectangles: NP-Completeness and Approximation Algorithms," *Algorithmica*, vol. 34, no. 3, pp. 217–239, 2002.

[21] F. de Dinechin. FloPoCo Project Website. [Online]. Available: http://flopoco.gforge.inria.fr