Heuristics for the Design of Large Multipliers for FPGAs

Andreas Böttcher, Keanu Kullmann, Martin Kumm
Fulda, University of Applied Sciences, Faculty of Applied Computer Science, Fulda, Germany
andreas.boettcher@cs.hs-fulda.de, keanu.kullmann@cs.hs-fulda.de, martin.kumm@cs.hs-fulda.de

***Abstract*—This proposal presents a scalable methodology for the design of large multipliers by using tiling. Thereby, a multiplier of a given arbitrary size is partitioned into smaller DSP blocks or logic-based multipliers. This can be represented by tiling an area (defined by the size of the large multiplier) by using tiles of different shapes (defined by the small multipliers), each assigned with individual costs. Resource optimal solutions for this problem have been proposed for small multipliers by using integer linear programming (ILP) solvers, but the computational effort to solve the tiling problem for multipliers beyond 64x64 exceeds solving times of several days on current computers. Many applications like from cryptography require much larger multipliers. In addition, none of the previous methods exploit resource reductions from the well known Karatsuba scheme. Hence, it is first shown how the Karatsuba scheme can be included in the tiling optimization by considering it as a specific tile. Next, two fast and scalable tiling heuristics are presented to obtain good solutions in a reasonable time. Similar to previous work, the first heuristic is based on a greedy search. Based on that, the second heuristic improves these results by applying the idea of the beam search meta-heuristic. Both heuristics are capable to include the Karatsuba scheme, scale well to large multipliers and show significant improvements compared to state-of-the-art heuristics.**

***Index Terms*—large multiplier, tiling, heuristics, computer arithmetic, post quantum cryptography, Karatsuba multiplier**

I. Introduction

Due to the elementary nature of the multiplication operation, the effective implementation of the operator is of vital interest in computer arithmetic, since it greatly affects the implementation of derived operators and algorithms. Although the implementation of generic multipliers is a well studied matter, the implementation on FPGAs poses some distinct design challenges, since the features and limitations of the available hardware resources confine the structure of the specific realization. Modern FPGAs from vendors like Intel or Xilinx typically provide DSP-blocks that can perform a multiplication operation directly. For example the Intel Stratix 10 FPGAs feature DSP-Blocks that can be configured to perform a signed or unsigend 27×27 or two 18×18 multiplications, while the recent series of Xilinx FPGAs feature DSP-blocks that can perform a 25×18 signed or 24×17 unsigned multiplication. Since optimized fixed or floating point calculations might require arbitrary input sizes, the multiplier tiling methodology was devised to provide an automatic design process to assemble a multiplier of a given size from smaller sub-multiplier tiles, as they can be realized by the LUTs and DSP-blocks [1]–[4].

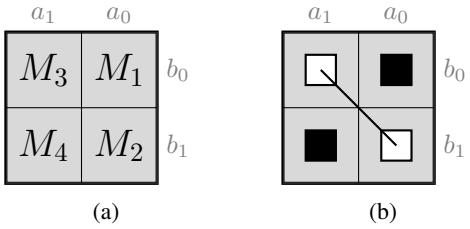


Fig. 1: Tiling representation of a $2W \times 2W$ multiplication using $W \times W$ multipliers (a) using schoolbook multiplication and (b) using Karatsuba

Take as an example a multiplication $A \times B$ with input vectors A and B of size $2W$. A and B can be split into two subwords of size W each and the multiplication becomes

$$\begin{aligned} A \times B &= \left(a_1 2^W + a_0\right)\left(b_1 2^W + b_0\right) \\ &= \underbrace{a_1 b_1}_{M_4} 2^{2W} + (\underbrace{a_1 b_0}_{M_3} + \underbrace{a_0 b_1}_{M_2}) 2^W + \underbrace{a_0 b_0}_{M_1} \end{aligned} \quad (1)$$

Hence, the $2W \times 2W$ bit multiplication can be replaced by four $W \times W$ multiplications $M_{1,..,4}$.

This can be graphically represented as quadratic area of size $2W \times 2W$ which has to be covered by tiles of size $W \times W$ as shown in Fig. 1. The tiling can be easily generalized to non-quadratic multipliers as well as non-quadratic tile shapes – each can have complex shapes (see Section II below). If several tile shapes are available having different cost we get a combinatorial optimization problem [4]. On FPGAs, these different tiles can represent DSP blocks or logic-based multiplications.

Different methods were proposed to obtain solutions for the multiplier tiling problem. In [2] and [3] heuristics were presented that cover the area with DSP-blocks up to a user-defined threshold and fills the remaining positions with logic-based multipliers. These have been either generic [2] or an optimized type of 3×3 [3] logic-based multiplier, that fits to the 6-input LUTs of modern FPGAs. Resource optimal solutions can be found with the integer linear programming (ILP) formulation described in [4]. The ILP approach yielded improvements of up to 47.5% in terms of LUT-usage for a given number of DSP-blocks in the design. However, due to the NP completeness of the ILP problem, this does not scale well to large multiplier sizes. For large multipliers like 64×64, the number of variables and equations rises dramatically which

leads to a high number of possible solutions and therefore long solving-times of hours or days and huge memory requirements of hundreds of GBytes.

Various applications like from cryptography demand larger multiplier sizes, but the computational and memory efforts are prohibitive to optimize larger multipliers with the ILP method on current computers in a reasonable time. Encryption methods like RSA or the fully homomorphic encryption scheme FHE examined in [5] rely on large multiplications in their implementation. In [5], architectures for computing multiplications up to 2556×19350000 are considered.

Apart from the tiling several methods were proposed to reduce its complexity. It was first demonstrated by Karatsuba, that multiplication can be realized with a sub-quadratic rise in complexity [6]. This also has been extensively used for FPGA designs [7], [8]. The complexity can be further reduced by applying discrete transforms or number theoretic transforms (NNTs) like used in the Schönhage-Strassen algorithm [9]. However, it has been shown recently that this only applies for very large multipliers with multiplicand sizes larger than 16,384 bits [7]. Below that, the Karatsuba scheme typically performs better. However, applying the Karatsuba scheme to the rectangular shape of the DSP blocks in current FPGAs is challenging as shown recently [8]. There remain cases which are not covered by DSP blocks that might be better filled by logic-based multipliers.

To summarize, there is a lack in good heuristics for the tiling optimization that scale well to large multiplier sizes and they can not make use of the complexity reductions of the Karatsuba scheme. The main contribution of this work is to close this gap by proposing heuristics for solving the tiling problem that 1) scale well to large multipliers 2) improve the state-of-the-art heuristics in terms of quality and 3) exploit the Karatsuba scheme including the rectangular case by treating the Karatsuba cases as a special set of tiles. We have a focus on Xilinx FPGAs in this work but believe that similar reductions are likely to be achieved for other FPGA vendors.

In the following, we will first start with an introduction to the multiplier tiling scheme, followed by a formal description of the corresponding optimization problem (Section II-B). Next, we introduce how to generally treat the Karatsuba scheme in the tiling problem (Section III). Then, we introduce a first heuristic that is based on a greedy search (Section IV). This greedy search is used as starting point for an improved heuristic that is based on the beam search meta-heuristic (Section V). Finally, experimental results are proposed and a conclusion is drawn.

II. Multiplier Tiling

A. The Tiling Rules

With the multiplier tiling methodology the desired multiplier is described geometrically as a rectangular shape, which size is defined by the word size of the multiplicands A and B. To design a multiplier from smaller sub-multipliers, this rectangular shape can be tiled by the shapes that describe the sub-multipliers (which do not necessarily have to be rectangular). Then, the position of each sub-multiplier determines the respective input bits of the sub-multiplier and the binary weight of its output. So, a multiplier that corresponds to a tile placed at coordinates (x, y) (which we define geometrically as the upper right corner) with dimensions (W_X, W_Y) has to be connected to the input $A(x, .., x + W_X - 1)$ and $B(y, .., y + W_Y - 1)$. The least significant bit (LSB) of the result can be determined by the sum of the coordinates $x + y$ while the most significant bit (MSB) is obtained by its furthest point from the origin $x + y + W_X + W_Y - 1$. Accordingly, the output vector width is defined by the sum of the tiles dimensions $W_X + W_Y$.

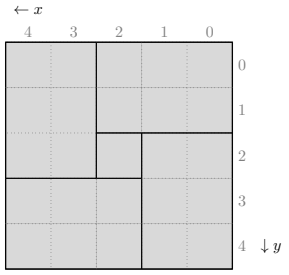


Fig. 2: Example of a 5×5 multiplier tiled with LUT-multipliers

The weighted output vectors can then be added with a compressor tree to calculate the final result. To provide valid solutions for a multiplier of a desired size, the tiling methodology requires that each field of the grid pattern is covered by exactly one sub-multiplier.

Figure 2 shows an example tiling of a 5×5-multiplier using 2×3 and 1×1 LUT-based multipliers. In accordance with the relationships presented above, the lower left 2×3-multiplier receives $A(2, .., 4)$ and $B(3, 4)$ as input and generates a 5-bit output vector that contributes with weight 5 to the compressor tree.

B. The Tiling Problem

The task of tiling a multiplier using different tiles can be formulated as a generic optimization problem as follows:

Multiplier Tiling Problem: Given a shape $M_{x,y} \in \{0, 1\}$ of the large multiplier and a 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 $m_s \in \mathcal{S}$.

For the tiling of multipliers for FPGAs, the cost cost_s is compiled of different resources, the LUTs and DSPs. The whole problem thus becomes a multi-objective optimization problem. There are several ways to treat this. One way is to weight the individual costs

$$\text{cost}_s = (1 - \alpha)\text{cost}^s_{\text{LUT}} + \alpha\,\text{cost}^s_{\text{DSP}} \tag{2}$$

and to select the $\alpha = 0 \ldots 1$ to find the desired trade-off. Another way is to limit one of the resources, typically the DSP resources are limited as these are most limited and of course most specific. Our tool flow supports both, hard thresholds as well as the optimization using weighted cost.

TABLE I: Propeties of available multiplier tiles (a) LUT-based multipliers (b) DSP-based multipliers

Shape	Tile area	$\text{cost}^s_{\text{LUT}}$	$\text{cost}^s_{\text{DSP}}$	LUT Efficiency
1×1	1	1.65	0	0.625
1×2	2	2.3	0	0.87
2×3	6	6.25	0	0.96
3×3	9	9.9	0	0.91
2×k	2k	1.65k + 2.3	0	$\frac{2k}{1.65k+2.3}$
24×17	408	26.65	1	15.30
super tiles (a–d)	816	27.3	2	29.89
super tiles (e–l)	816	37.7	2	21.64
2-part Karatsuba	1536	146	3	10.521
3-part Karatsuba	3456	360	6	9.6
4-part Karatsuba	6144	668	10	9.198

C. Available Multiplier Tiles

Table I gives an overview of the sub-multipliers together with their properties that can be used for the design of larger multipliers that have been proposed and used in previous work [2]–[4], [10]. The metrics used in the table have been applied for recent Xilinx FPGAs and are explained below.

The tile area denotes the number of coordinates that are actually covered. The LUT cost are obtained by

$$\text{cost}^s_{\text{LUT}} = \#\text{LUT}_s + 0.65w_s \tag{3}$$

where $\#\text{LUT}_s$ denotes the number of LUTs used to realize the multiplier while and w_s is the output word size. The term $0.65w_s$ approximates the LUT contribution of a single bit in the compressor tree that is used to add all the partial products. Its factor 0.65 was experimentally obtained in [4]. The term $\text{cost}^s_{\text{DSP}}$ denotes the number of DSPs used in the multiplier. The *efficiency* is a benefit-cost ratio by dividing the tile-area of the multiplier (i. e., the geometric area on the multiplier board) by the cost

$$E_s = \frac{\text{area}_s}{\text{cost}_s} \ . \tag{4}$$

The LUT efficiency in Table I assumes an α of zero in (2), which means that only LUTs are counted.

The upper part of Table I shows logic-based multipliers. Their shapes are also visualized in Fig. 3. Most of them try to utilize the 6-input LUT of the FPGA as much as possible, the smaller ones (like the 1×1) is inserted to be able to fill smaller gaps. The $2 \times k$ is one row of a Baugh-Wooley multiplier with an FPGA mapping proposed in [10].

Besides the logic-based multipliers, different DSP-based multipliers exist, which are given in the lower part of Table I. The 24×17 one is the Xilinx DSPE48E(1) in standard unsigned configuration. Several DSPs can be combined by using their post adder to build so-called super-tiles [1]. All efficient cases that can be built from two DSPs are shown in Fig. 4. The same has been used in previous work [4]. Their shapes come from the fact that the post adder require either a zero shift between two DSP tiles or a hard coded 17 bit shift. This requires that their manhattan distance is either 0 or 17, which can be easily checked by the reader. As their output word size depends on this shift, different cost values are obtained in Table I.

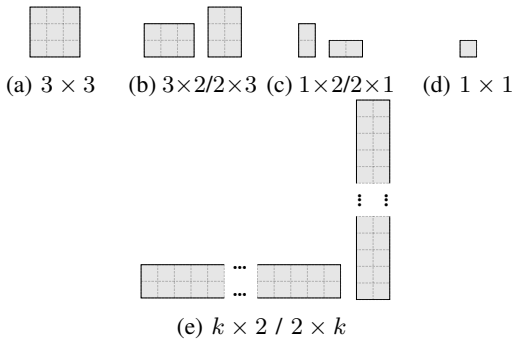


Fig. 3: Geometric shapes of the LUT-based tiles

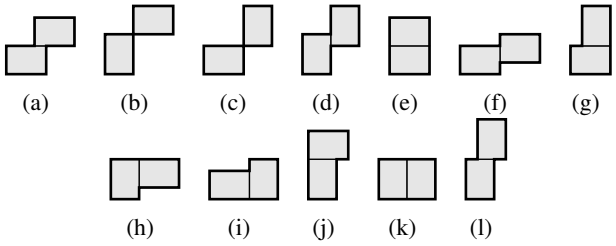


Fig. 4: All super-tiles consisting of two DSP blocks

The tiles denoted as n-part Karatsuba in Table I use the Karatsuba scheme. How to describe the Karatsuba scheme as a special tile is described in the following.

III. Karatsuba as a special tile

The main idea behind the Karatsuba scheme is to apply the mathematical identity

$$\underbrace{a_1b_0}_{M_3} + \underbrace{a_0b_1}_{M_2} = (a_0 + a_1)(b_0 + b_1) - \underbrace{a_0b_0}_{M_1} - \underbrace{a_1b_1}_{M_4} \tag{5}$$

to (1), which replaces two of the multiplications (M_2 and M_3) by a single one by subtracting two of the results obtained from other multiplications (M_1 and M_4). Thus, two multiplications and their subsequent addition of size $2W$ are replaced by one multiplication, two pre-additions of size W and two size $2W$ post-subtractions. The result of M_1 and M_4 can be reused since it is computed anyway in different parts of the formula.

To illustrate this, we use a notation shown in Fig. 1b. Here, the black squares show the multipliers that are used to eliminate one of the connected white squares.

The Karasuba scheme can be generalized by considering, that every pair of sub-multiplications a_ib_j and a_jb_i with equal weight 2^{i+j} can be substituted if the result of a_ib_i and a_jb_j is available. So (5) can be reformulated to [8]

$$a_ib_j + a_jb_i = (a_i + a_j)(b_j + b_i) - a_ib_i - a_jb_j \ . \tag{6}$$

Alternatively, (6) can be rewritten to a form where the pre-additions and post-subtractions are replaced by pre-subtractions and post-additions

$$a_ib_j + a_jb_i = (a_i - a_j)(b_j - b_i) + a_ib_i + a_jb_j \ . \tag{7}$$

This relation can be be applied several times, either recursively or by using the same pattern several times, called the n-part splitting. Although, the recursive scheme usually shows the best resource complexity, the n-part splitting is more attractive for hardware implementation as its word size and the circuit depth stays constant (for details see [8]). Thus, we focus on n-part splitting in the following but it would be straigh-forward to define tiles for the recursive scheme as well.

Karatsuba was traditionally only applied with quadratic sub-multipliers. It was recently shown in [8], that the conditions for the use of Karatsuba can also be satisfied when using rectangular multipliers, like the embedded DSP blocks of recent Xilinx FPGAs. For that, both tiles to be substituted have to share the same weight 2^{i+j}. When rectangular sub-multiplier of size $W_A \times W_B$ are used this criteria can be met if the multiples $N, M \in \mathbb{N}$ of the greatest common divisor (GCD) $W = \texttt{gcd}(W_A, W_B)$ result in the same weight in both dimensions

$$W_A N = W_B M \quad . \tag{8}$$

The multiples N, M can be calculated using the GCD W by $M = W_A/W$ and $N = W_B/W$. So the smallest multiplier composed of $W_A \times W_B$ sub-multipliers, where the Karatsuba scheme can be used is of size:

$$(W_A N + W_A) \times (W_B M + W_B) \tag{9}$$

For the 17×24 DSP multiplier targeted here, the GCD is 1 and the smallest possible multiplier size for which Karatsuba can be applied is $425{\times}432$, which is not very attractive. Here, the idea in [8] was to slightly under-utilizing the DSP as a $16{\times}24$ multiplication. This is advantageous, as the GCG now becomes 8, Karatsuba can be already applied on a 64×72 multiplier. This is illustrated in Fig. 5(a).

Using the DSPs as $17{\times}17$ multipliers is only advantageous for smaller multipliers since Karatsuba can be applied at every multiple of 17, starting at $34{\times}34$, but more area is wasted due to the under-utilisation.

The n-part splitting can also be applied for $n > 2$ using the rectangular Karatsuba scheme starting at 112×120, as illustrated in Fig. 5(b). The number of DSPs used for the Kartsuba scheme as a function of n can be described by the arithmetic series:

$$\#\texttt{DSP}(n) = \sum_{x=1}^{n} x = \frac{1}{2} n(n+1) \tag{10}$$

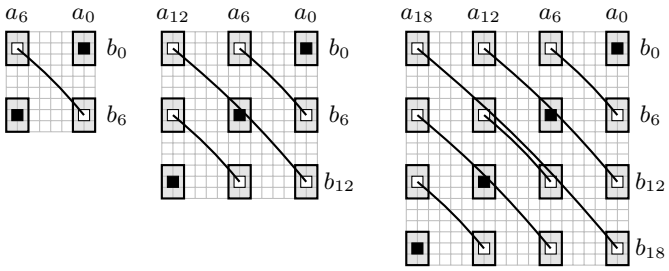
The relative reduction of DSPs saved by Karatsuba can be described using the quotient of $\#\texttt{DSP}(n)$ divided by the number of DSPs required without the application of Karatsuba.

$$\text{red}(n) = 1 - \frac{\#\texttt{DSP}(n)}{n^2} = \frac{1}{2}\left(1 - \frac{1}{n}\right) \tag{11}$$

For $n \to \infty$, the relative DSPs reduction converges to

$$\lim_{n \to \infty} \text{red}(n) = 0.5 \ , \tag{12}$$

so, up to half of the DSPs can be saved. It is generally desirable to use the largest possible n to fit the indented multiplier size.



(a) 64×72 (b) 112×120 (c) 160×168

Fig. 5: Karatsuba n-part splitting with $n = 2, 3, 4$ and 16x24 base multipliers

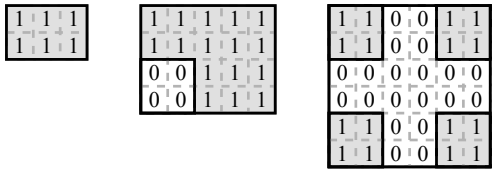


Fig. 6: Coverage data structure

Instances of the n-part splitting can be used for the tiling methodology by allowing tiles with an irregular shape having even unconnected areas. Hence, the tiles do not cover the complete area enclosed by its outer dimensions. This is modeled by a binary mask as illustrated in Fig. 6. The spaces in between the individual DSPs can be filled by other tiles like regular DSPs, LUT multipliers or further (possibly shifted) instances of Karatsuba. By doing so, the Karatsuba scheme can be integrated into any solver capable to solve the problem as defined in Section II-B. This includes the optimal ILP-based [8] as well as the novel heuristics which are presented next.

IV. Proposed Greedy Heuristic

The previous heuristics had a few limitations, such as not supporting variable $(2 \times k)$ as well as general irregular tiles like the Karatsuba tiles introduced above. The proposed heuristic aims to support all of those tiles while trying to beat the optimal ILP tiling algorithm in run time and previous heuristics in the total cost of the generated tilings.

The general idea is to tile the board by iteratively selecting the tile which delivers the best *effective efficiency*. With *effective efficiency* we mean the efficiency according to (4) when considering the effectively covered area in area$_s$. The *covered* area can be smaller that area$_s$ in the case when 1) the tile overlaps the border or 2) the size of the tile has to be reduced because it otherwise does not fit inside an area already covered. The algorithm starts at the origin 0,0 (which is the upper right corner in the figures) and aims to place the most efficient tile at this location. All tiles and orientations are evaluated in each iteration and the one with the best effective efficiency is selected. Then, the coordinate is updated to the next free location that is closest to the origin accordingly to the euclidean distance. This process is repeated until the complete area is tiled.

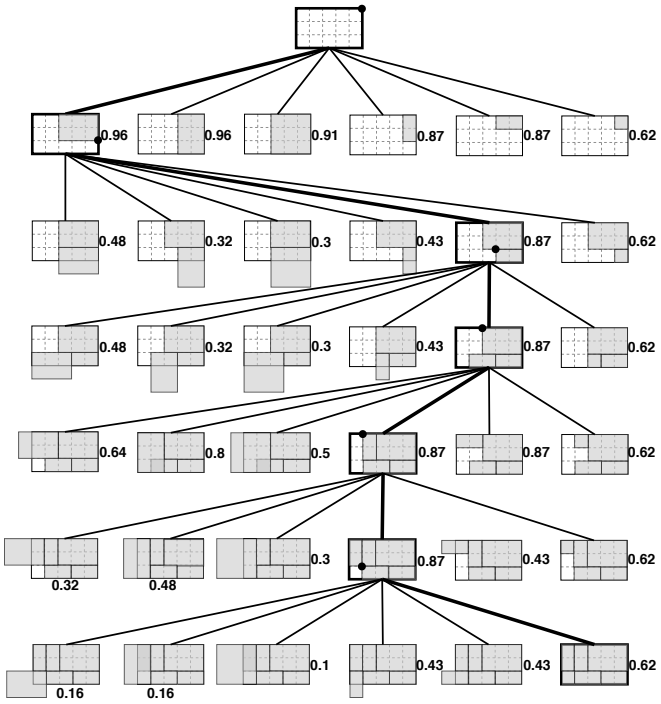


Fig. 7: Optimization tree using the greedy tiling heuristic

This search is illustrated as a tree in Fig. 7 for the example of a 5×3 multiplier that is tiled by a tiling set containing the first four tiles of Table I. In the first iteration (first row in Fig. 7), each tile achieves its efficiency as it does not overlap with the border. Hence, the first and best tile with $E = 0.96$ is selected. In the next iteration (second row), the best efficiency is obtained for the 2×1 tile. This continues until a complete cover is found which is shown in the last column of the last row.

The pseudo code for the proposed greedy heuristic is shown in Listing 1. The heuristic requires a list of all available tiles, that is sorted by the efficiency. Variable size tiles will be scaled when needed. This avoids having a too large list which would decrease the performance of the algorithm. The algorithm also requires a data structure that takes care of the current state of the area that needs to be tiled (`board` in Listing 1). Another key aspect is handling variable size and irregular tiles like the Karatsuba ones mentioned above.

The outer loop (line 5–19) will continue to attempt to place a tile until the complete area is tiled. The next free position to place a tile is obtained by `getNextPosition()`. In the inner loop (line 8–14), the heuristic tries to find the tile with the most efficiency for a certain position. The function `getEfficiency()` computes the effective efficiency. If the efficiency of the current tile is higher than the currently highest efficiency it is updated. After the inner loop identified the best tile, it will be placed in line 16 by calling `placeTile()`. This function will update the remaining untiled area and increases, if required, the number of used DSP blocks.

Algorithm 1: Proposed greedy tiling heuristic

```
1  greedy(board, tiles):
2  solution = dspList = []
3  cost_total = 0
4
5  while board not completely covered do
6    eff_best = -1.0
7    (x,y) = getNextPosition(board);
8    foreach tile in tiles do
9      eff_temp = getEfficiency(board, tile, x, y)
10     if eff_temp > eff_best do
11        eff_best = eff_temp
12        tile_best = tile
13     end
14   end
15
16   placeTile(board, tile_best, x, y)
17   cost_total += getLUTCost(tile_best)
18   add(solution, tile_best)
19 end
20
21 cost_total -= searchSuperTiles(dspList, solution)
22 return cost_total, solution
```

Placing non-rectangular tiles in an efficient way is not an easy task itself, it would basically require yet another heuristic that solves the problem that is underlying in the well known game "Tetris", which is itself NP complete [11]. As most super tiles are non-rectangular tiles the heuristic will run a second pass by `searchSuperTiles()` afterwards, similar to the previous heuristics. In this second pass, all DSP tiles are checked and possible super tiles are built while the remaining DSP tiles are added to the solution list directly. In case super tiles are found, the solution has to be updated and the cost have to be reduced by the achieved savings as well. In the end, the algorithm provides the approximate LUT costs and all required tiles and coordinates in a list. Not included in the proposed pseudo code are optimizations, such as discontinuing tiling checks as soon as a tile fits perfectly and thus none of the tiles after it in the list have a chance to beat its efficiency or skipping tiles that have no chance of fitting in the remaining subarea.

As naturally for a greedy algorithm a good decision from a local point of view may lead to worse solutions in later decisions. To overcome this problem, we present an improved heuristic in the following.

V. Proposed Beam Search Heuristic

Beam search is a meta-heuristic algorithm that tries to avoid problems that come from only following the path of local optimums. It requires a greedy solution and checks other solutions / branches in a certain range around it. While these solutions are not the local optimum, they may lead to a better total result. The proposed beam search starts with a solution obtained from the greedy heuristic. Then, for each decision, the B (the "beam width") locally best solutions are evaluated using a greedy depth search. Then, the solution with the best

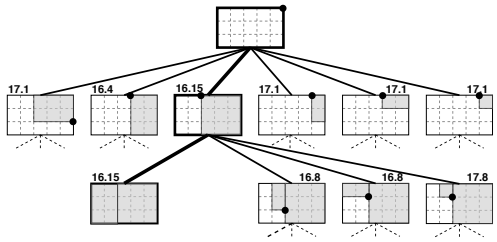


Fig. 8: Optimization tree using the beam search heuristic

total cost is selected and the same procedure is repeated in the next iteration.

To illustrate this, the search tree for the beam search heuristic solving the same problem as in Fig. 7 is shown in Fig. 8. Again, each possible tile is evaluated but the evaluation is based on the greedy search that follows after the selection of the tile. For the first tile in the first iteration (the one selected in Fig. 8), the total cost is 17.1 (which is the cost of the solution of Fig. 7). By also checking the B neighbors at the same levels (the example shows results for all elements, i.e., $B = 5$), it turns out that the third tile in the first iteration leads to a better total cost, even though the local efficiency was worse. Hence, this one is selected and the same procedure is repeated for the next tile. In this example, a complete cover is already found in the second iteration.

The beam width parameter B now allows to trade between search space (and thus run-time) and the optimization quality.

VI. Results

Several experiments were performed to evaluate the proposed heuristics including the Karatsuba scheme. We start with comparisons to mid-size multipliers (up to 64 bits) to compare with previous work. For those, we can not make use of the Karatsuba scheme as it is just too small to apply the smallest 2-part splitting (see Fig. 5). It is also the region where optimal results can be obtained by the ILP approach of [8]. However, it gives valuable insights regarding the the distance to the optimal objective value and its performance compared to previous heuristics [2], [3].

Then we go up to 1024 bits to compare the run time as well as the optimization quality. Here, the Karatsuba scheme can be fully exploited. Finally, we demonstrate the flexibility of our tool by performing a complete design space exploration of a 256×256 bit multiplier with and without the Karatsuba scheme.

The algorithms have been integrated into the `IntMultiplier` operator of the VHDL code generator framework FloPoCo [12] and are freely available. All experiments were performed on an Intel Xeon machine with an E5-2650v2 CPU @ 2.60GHz (8 cores, 16 Threads) and 128GB of RAM running Ubuntu 18.04.3 LTS. The tests involving optimal ILP tiling were utilizing the Gurobi ILP solver version 8.1.0. The synthesis experiments were performed with Xilinx Vivado 2019.1 targeting the Kintex7 FPGA and include place&route. Timing results were obtained by placing a wrapper with registers at inputs and outputs and considering the critical path of this wrapper.

TABLE II: Optimization results for optimal (ILP) and the proposed greedy and beam search heuristics

24 x 24 (single precision floating point)					
#DSP	2	1	0		
optimal (ILP) [4]	31.2	179.95	502.8		
prop. greedy	31.2	179.95	502.8		
prop. beam search	31.2	179.95	502.8		
32 x 32 (unsigned)					
#DSP	4	3	2	1	0
optimal (ILP) [4]	57.85	119.2	256.8	567.95	881.6
prop. greedy	67.6	119.2	256.8	570.55	881.6
prop. beam search	62.4	119.2	256.8	567.95	881.6
53 x 53 (double percision floating point)					
#DSP	9	8	7	6	5
optimal (ILP) [4]	144.3	164.45	307	450.5	759.7
prop. greedy	234.65	324.9	362.55	451.15	782.75
prop. beam search	180.1	228.5	307	450.5	761.9
64 x 64 (unsigned)					
#DSP	11	10	9	8	7
optimal (ILP) [4]	198.25	354.8	570.7	862.5	1192.35
prop. greedy	218.4	427.2	692.05	988.55	1250.5
prop. beam search	208	427.2	641.4	939.1	1209

A. Comparison to Optimal ILP on Mid-Size Multipliers

The motivation to develop the presented heuristics was to provide an alternative for the optimal ILP approach presented in [4], while providing solutions close to optimal with a considerably shorter runtime of the algorithm. Table II shows a comparison of the objectives (LUT cost) generated by the heuristics in comparison to the optimal results in [4] for various sizes of multipliers and maximal allowed DSPs. It can be seen that the optimal results can be found for smaller multipliers. As expected the beam search heuristic is able to reach better or equal objectives than greedy.

Most results are close to the optimum with some exceptions. The optimum in case of the 53×53 multiplier is really hard to obtain. Its optimum contains a "windmill" like pattern (see [4]) similar to the one of Fig. 2 consisting of many efficient super tiles. Here, the proposed heuristics do not find the optimal pattern for the exploitation of the integrated adders in super-tiles. However, as shown in the next section, the proposed heuristics still outperform the previous heuristics.

B. Comparison to Previous Heuristics on Mid-Size Multipliers

A series of tests were performed to benchmark the novel heuristics with previous state-of-the-art heuristics presented in [2] and [3]. As those were available as VHDL source from previous FloPoCo versions, we performed synthesis experiments and compared the resulting slices, which also include the flip flops (FFs) for pipelining as well as the real cost for the compressor tree. Table III shows the slices for various numbers of DSPs compared to the previous heuristics. In most of the cases, significant savings can be obtained

TABLE III: Synthesis results compared to previous approaches

Mult.	Method	#DSP	Slices	Slice red.
24 × 24	[2]	1	83	
	prop. beam search	1	69	16.9%
	[2]	2	19	
	[3]	2	0	100%
	prop. beam search	2	0	100%
32 × 32	[2]	0	427	
	[3]	0	257	39.8%
	prop. beam search	0	288	32.5%
	[2]	1	279	
	[3]	1	257	7.9%
	prop. beam search	1	197	29.3%
	[2]	2	135	
	[3]	2	119	11.8%
	prop. beam search	2	123	8.9%
	[2]	3	95	
	[3]	3	115	−21.0%
	prop. beam search	3	74	22.1%
	[2]	4	58	
	[3]	4	15	74.1%
	prop. beam search	4	16	72.4%
53 × 53	[2]	5	411	
	prop. beam search	5	322	21.6%
	[2]	6	240	
	[3]	6	226	5.8%
	prop. beam search	6	226	5.8%
	[2]	7	322	
	prop. beam search	7	130	59.6%
	[2]	8	211	
	prop. beam search	8	103	51.1%
	[2]	9	230	
	[3]	9	174	24.3%
	prop. beam search	9	105	54.3%
64 × 64	[2]	7	713	
	prop. beam search	7	467	34.5%
	[2]	8	584	
	[3]	8	500	14.4%
	prop. beam search	8	374	35.9%
	[2]	9	497	
	prop. beam search	9	312	37.2%
	[2]	10	415	
	prop. beam search	10	252	39.2%
	[2]	11	415	
	[3]	11	234	43.6%
	prop. beam search	11	125	68.1%

with the beam search heuristic. The speed of the circuits are similar as they use pipelines of similar depth. The average clock frequencies from the available designs were 305 MHz, 255 MHz, and 283 MHz for [2], [3], and our designs.

C. Comparison to Previous Karatsuba Results

In [8] various arrangements of DSPs and rectangular Karatsuba schemes were proposed, that yielded considerable savings in utilized DSPs at multiplier sizes of 64×72, 96×96 and 112×120. By considering the rectangular Karatsuba arrangements as a special tile as proposed in Section III, it could be included into the ILP approach of [4]. With that, the results from [8] could be reproduced in terms of LUT and DSP cost, which proves the optimality of the previous results. In addition, the proposed greedy and beam search heuristics are both able to find these optimal results. It should be emphasized that in contrast to the hand-optimized designs of [8], this is now a fully automated tool flow that can be applied to arbitrary multiplier problems.

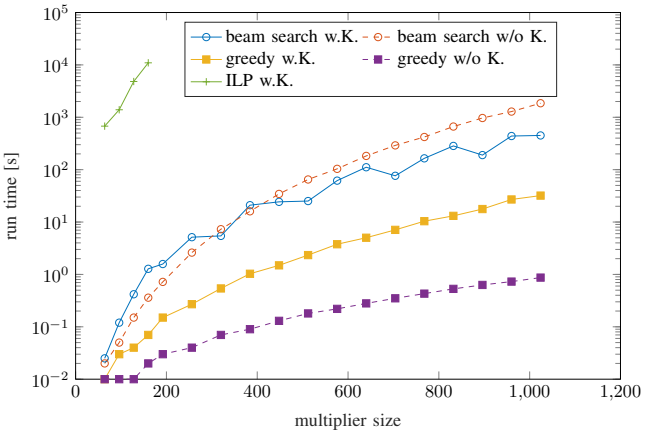


Fig. 9: Runtime of the presented methods with Karatsuba (w K.) and without Karatsuba (w/o K.)

D. Quality and Run Time for Large Multipliers

In this experiment, we performed optimizations for up to 1024×1024 bit multipliers. The optimal ILP approach [4] was able to find solutions of up to to 160×160 when using the proposed Karatsuba extension. Larger solutions could not be found due to the exhaustion of the 128GB available RAM.

Fig. 9 shows the run times of the different methods with and without Karatsuba at a logarithmic scale. While the largest ILP solution required 3h, the beam search and greedy heuristics required solving times of 1.28s and 0.07s, respectively. Fluctuations in solving times can be observed for Karatsuba at the points where larger Karatsuba patterns fit the area. While the greedy algorithm generally runs slower with Karatsuba patterns, beam search is faster in case of larger multipliers. In the 1024×1024 case, greedy requires 32s (0.9s) and beam search 450s (1855sec) with (without) Karatsuba. It has to be noted that for the large multipliers, the time to optimize the compressor tree (using the heuristic of [13]) requires more time than the tiling itself (not counted here). It took about 3h for the 1024×1024 case.

Fig. 10 shows the LUT and DSP cost for the same experiment. The use of the Karatsuba scheme increases the LUT costs due to the required extra adders while reducing the utilized DSPs. In the 1024×1024 case, the LUT cost using the greedy heuristic rises from 49354 to 133975 while the utilized DSPs sink from 2605 to 1471 by 56%. This is relatively close to the theoretical maximum of 50%. It can further be seen, that the LUT cost advantages of using beam search over greedy

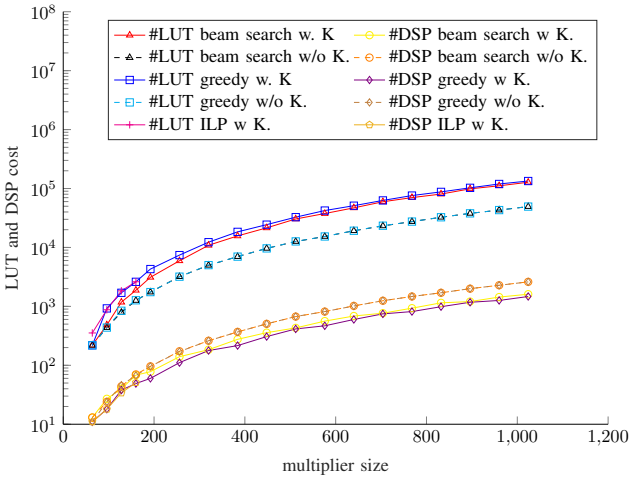


Fig. 10: DSP and LUT costs for various sizes with Karatsuba (w K.) and without Karatsuba (w/o K.)

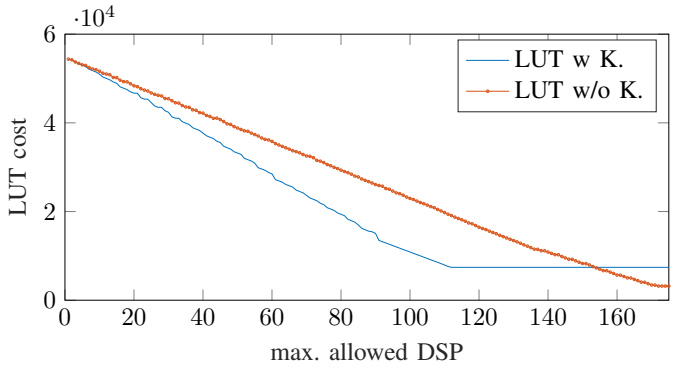


Fig. 11: LUT costs of a 256×256-multiplier as a function of the max. allowed DSP with and without Karatsuba

shrink with the multiplier size, as they are identical in the 1024×1024 case.

The application of Karatsuba does not generally result in higher LUT costs. A complete design exploration for the full trade-off between LUTs ans DSPs is given in Fig. 11. It can be observed that the Karatsuba scheme requires considerably less LUT resources for a given DSP count as long as the DSP resources are limited. Only when more than about 150 DSPs are allowed, further LUT resources can be saved by avoiding the overhead from Karatsuba pre/post adders. But the price to pay in terms of DSPs is large.

VII. Conclusion

It was demonstrated that the Karatsuba patterns can be integrated in the generic tiling methodology and solutions for medium sized multipliers can be found using ILP. Two novel heuristics based on the greedy and beam search meta-heuristics were presented, that overcome the performance and memory limitations of the ILP approach for large multipliers. Those were shown to generate results close to the optimal ILP formulation and offer considerable improvements over existing state-of-the-art heuristics. While beam search has advantages for mid-size multipliers, with run times of seconds the greedy heuristic provides a tool for the design of multipliers up to and beyond 1024×1024.

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)*. IEEE, 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 Comput. Archit. News*, vol. 38, no. 4, p. 73–79, Jan. 2011. [Online]. Available: https://doi.org/10.1145/1926367.1926380

[3] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic core generation using bit heaps," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sep. 2013, pp. 1–8.

[4] M. Kumm, J. Kappauf, M. Istoan, and P. Zipf, "Resource optimal design of large multipliers for FPGAs," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, July 2017, pp. 131–138.

[5] C. Moore, N. Hanley, J. McAllister, M. O'Neill, E. O'Sullivan, and X. Cao, "Targeting FPGA DSP slices for a large integer multiplier for integer based FHE," in *Financial Cryptography and Data Security*, A. A. Adams, M. Brenner, and M. Smith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 226–237.

[6] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," in *Soviet Physics Doklady*, vol. vol. 7, Jan 1963, p. 595.

[7] C. Rafferty, M. O'Neill, and N. Hanley, "Evaluation of Large Integer Multiplication Methods on Hardware," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1369–1382, 2017.

[8] M. Kumm, O. Gustafsson, F. de Dinechin, J. Kappauf, and P. Zipf, "Karatsuba with rectangular multipliers for FPGAs," in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, June 2018, pp. 13–20.

[9] A. Schönhage and V. Strassen, "Schnelle Multiplikation großer Zahlen," *Computing*, vol. 7, no. 3-4, pp. 281–292, Sep. 1971.

[10] 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.

[11] E. Demaine, S. Hohenberger, and D. Liben-Nowell, "Tetris is hard, even to approximate," in *9th International Computing and Combinatorics Conference (COCOON 2003)*, June 2003, p. 351–363.

[12] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2012.

[13] M. Kumm and J. Kappauf, "Advanced compressor tree synthesis for FPGAs," *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1078–1091, 2018.