Karatsuba with Rectangular Multipliers for FPGAs

Martin Kumm*, Oscar Gustafsson†, Florent de Dinechin‡, Johannes Kappauf*, and Peter Zipf*
*University of Kassel, Digital Technology Group, Germany
†Linköping University, Division of Computer Engineering, Department of Electrical Engineering, Sweden
‡Univ Lyon, INSA Lyon, Inria, CITI, France

kumm@uni-kassel.de, oscar.gustafsson@liu.se, florent.de-dinechin@insa-lyon.fr,
johannes.kappauf@student.uni-kassel.de, zipf@uni-kassel.de

***Abstract*—This work presents an extension of Karatsuba's method to efficiently use rectangular multipliers as a base for larger multipliers. The rectangular multipliers that motivate this work are the embedded 18×25-bit signed multipliers found in the DSP blocks of recent Xilinx FPGAs: The traditional Karatsuba approach must under-use them as square 18×18 ones. This work shows that rectangular multipliers can be efficiently exploited in a modified Karatsuba method if their input word sizes have a large greatest common divider. In the Xilinx FPGA case, this can be obtained by using the embedded multipliers as 16×24 unsigned and as 17×25 signed ones. The obtained architectures are implemented with due detail to architectural features such as the pre-adders and post-adders available in Xilinx DSP blocks. They are synthesized and compared with traditional Karatsuba, but also with (non-Karatsuba) state-of-the-art tiling techniques that make use of the full rectangular multipliers. The proposed technique improves resource consumption and performance for multipliers of numbers larger than 64 bits.**

I. Introduction

Karatsuba's method [1] was the first method for computing products with sub-quadratic complexity. It trades multiplications for additions, which are cheaper. It will be exposed in detail in Section II. In its original formulation, Karatsuba's algorithm is only expressed in terms of *square* multipliers, i.e., multipliers whose two inputs have the same number of digits. However, the multiplier resources available in recent Xilinx FPGAs are not square: they typically consist of a 18×25-bit signed multiplier which can be used for 17×24-bit unsigned multiplication [2]. The Karatsuba method is therefore inefficient on such FPGAs [3], [4]: to apply it, one either needs to under-use the DSP blocks (use them as 17×17-bit multipliers), or complement them with (slower) logic to turn them into 24×24-bit multipliers. The first option is preferred, as it can exploit the optional pre-adder and post-adder/accumulator that surround the multiplier in a DSP block [3], [4].

This article attempts to reconcile Karatsuba's method with FPGA hardware whose DSP blocks are rectangular. It first slightly generalizes the classical Karatsuba formulation. It then shows how rectangular multipliers can be used under certain constraints on their aspect ratio.

This problem corresponds to sparse polynomial multiplication, so the suggested algorithms also find use here. Previous work in this area primarily focus on asymptotically efficient results or comparisons for polynomials with a random sparseness [5], [6]. In [6], the authors mention regarding Karatsuba's method for sparse polynomial multiplication: "In the sparse case this phenomenon hardly occurs, and it is commonly admitted that this approach is useless". The current work shows that there are use cases, at least when the sparseness is regular, and that significant savings can be obtained compared to a straightforward evaluation. The authors are not aware of any explicit results for multiplication of sparse polynomials with a regular sparseness.

Back to FPGAs, the proposed technique is able to save a smaller proportion of the multipliers compared to classical Karatsuba using only square multipliers. However, since each rectangular multiplier is used almost to its full potential, the proposed technique saves resources and improves latency for large multipliers (above 64×64 bits). This is confirmed by actual synthesis results on recent Xilinx devices. Very large integer multipliers could be an important component for the booming field of fully homomorphic encryption (see for instance multipliers from 944×944 up to 2560×2560 bits in [7]).

II. Classical Karatsuba Method

A. Two-Part Splitting

Consider a large multiplication of size $2W \times 2W$ bits. It can be split into smaller multiplications of size $W \times W$ each by splitting the arguments into smaller segments of size W, leading to

$$A \times B = (a_1 2^W + a_0)(b_1 2^W + b_0) \tag{1}$$

$$= \underbrace{a_1 b_1}_{=\text{M4}} 2^{2W} + (\underbrace{a_1 b_0}_{=\text{M3}} + \underbrace{a_0 b_1}_{=\text{M2}}) 2^W + \underbrace{a_0 b_0}_{=\text{M1}} . \tag{2}$$

The basic idea of Karatsuba's algorithm is based on the identity:

$$a_1 b_0 + a_0 b_1 = (a_0 + a_1)(b_0 + b_1) - \underbrace{a_0 b_0}_{=\text{M1}} - \underbrace{a_1 b_1}_{=\text{M4}} \tag{3}$$

Hence, by reusing the sub-products M1 and M4, we can compute the sum of M2 and M3 by a single multiplication: this reduces the number of smaller multiplications from four to three. The price to pay are two new additions computing $a_0 + a_1$ and $b_0 + b_1$ (called pre-additions in the following),

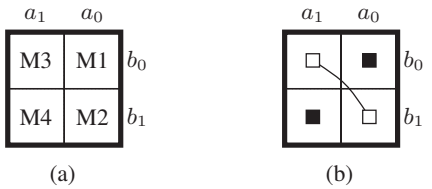


Fig. 1: Graphical tiling representation of (2)

and two additional negative terms in the final sum (called post-additions). Besides, the new product is a $(W+1)\times(W+1)$-bit one, hence requires a slightly larger multiplier than each of the $W \times W$-bit products it replaces.

Karatsuba's technique has been used in software for fast multiple-precision arithmetic. Equation (2) is applied recursively, and the recursion typically stops when the smaller multipliers match the capabilities of the hardware multipliers offered by the processor.

The present work addresses the design of large multipliers for FPGAs [3], [4], [8], [9]. It is closer to hardware multiplier design [10], [11]: the relevant metrics here are area and delay. Area-wise, Karatsuba reduces the cost for large enough values of W. Exactly how large is technology-dependent, and we will only address here FPGA technology with embedded hard multipliers.

Delay-wise, post-additions may be performed in constant time using carry-save arithmetic. Equivalently, they can be merged with negligible overhead in the final bit array compression. Pre-additions, on the other hand, have a significant timing overhead due to the need for carry propagation. Therefore, Karatsuba will typically improve area, but degrade timing.

An alternative form of (2) is

$$a_1b_0 + a_0b_1 = (a_1 - a_0)(b_1 - b_0) + M1 + M4\ . \quad (4)$$

It uses pre-subtractions, which have the same hardware cost as pre-additions. However, the product is now a product of signed numbers. This will therefore be the preferred formulation on FPGAs, whose DSP blocks are capable of signed multiplication. The extra bit added by the pre-subtractions is now a sign bit, allowing us to exploit the sign bit inputs of the DSP blocks that would otherwise be wasted.

B. Terminology and Graphical Representation

Throughout this article, we will use a tile-based graphical representation introduced in [4]. For instance, Fig. 1 illustrates the representation of (2). A multiplication is represented as a rectangle. Decomposing the inputs A and B into subwords splits this rectangle with vertical and horizontal lines respectively. Each of the sub-products appears as a smaller rectangle called a *tile* throughout this article. Fig. 1(a) shows square tiles, while Fig. 4 shows multiplications decomposed in rectangular tiles. When two multipliers are paired by Karatsuba, they are linked on the figure (Fig. 1(b)). Such a link means that the sum of the two linked tiles will be computed by a single multiplication, reusing the sub-products corresponding to tiles with black squares. The size of this multiplication is 1-bit larger than the tile size in both dimensions (and more if Karatsuba is applied recursively).

C. Square K-Part Splitting

The inputs to a large multiplication may also be split into K segments of W bits [12]:

$$A \times B = \Big(\sum_{i=0}^{K-1} 2^{iW} a_i\Big)\Big(\sum_{j=0}^{K-1} 2^{jW} b_j\Big) \quad (5)$$

$$= \sum_{i,j} 2^{(i+j)W} a_i b_j \quad (6)$$

In (6), any expression $a_ib_j + a_jb_i$ may be replaced as follows:

$$a_ib_j + a_jb_i = (a_i + a_j)(b_j + b_i) - a_jb_j - a_ib_i \quad (7)$$

which again computes the sum of two tile subproducts using only one multiplier. The novelty is that subproducts may be reused more than one time, which further improves the efficiency of the method.

Take for example, a $3W \times 3W$ multiplier where the arguments are split into three parts

$$A \times B = (a_2 2^{2W} + a_1 2^W + a_0)(b_2 2^{2W} + b_1 2^W + b_0) \quad (8)$$

$$= a_0b_0 + (a_0b_1 + a_1b_0)2^W + (a_0b_2 + a_2b_0 + a_1b_1)2^{2W} + (a_1b_2 + a_2b_1)2^{3W} + a_2b_2 2^{4W} \quad (9)$$

$$= a_0b_0 + ((a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1)2^W + ((a_0 + a_2)(b_0 + b_2) - a_0b_0 - a_2b_2 + a_1b_1)2^{2W} + ((a_1 + a_2)(b_1 + b_2) - a_1b_1 - a_2b_2)2^{3W} + a_2b_2 2^{4W}\ . \quad (10)$$

Here, (7) could be used three times, which reduces the multiplier count from $3 \times 3 = 9$ to only six.

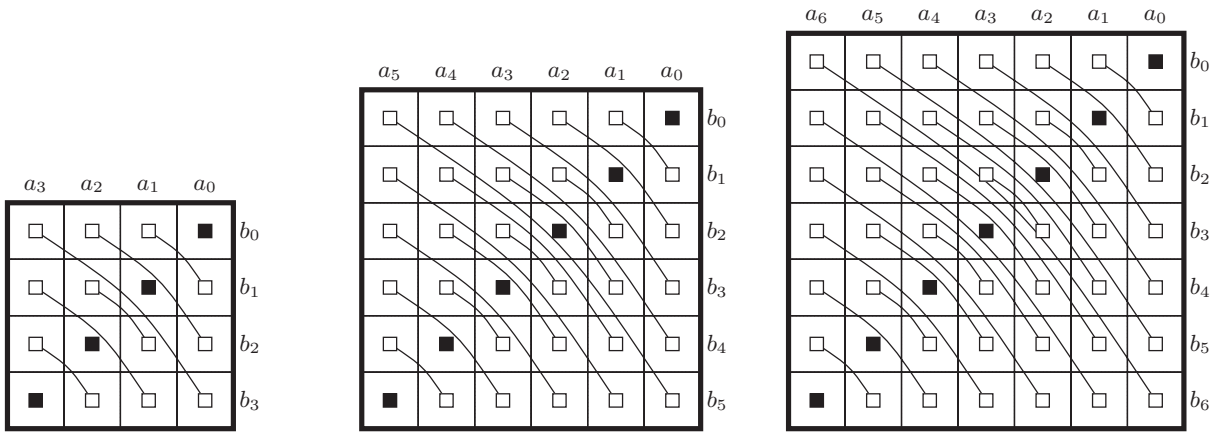
The K-Part splitting is a straightforward application of rule (7) on as many partial products as possible. Figure 2 shows examples of 4-part, 6-part and 7-part splitting.

In general, K-Part splitting saves a triangle of $K(K-1)/2$ multiplications. For instance, in the 4-part splitting of a $4W \times 4W$ multiplier shown on Fig. 2(a), six of the 16 required $W \times W$ bit multiplications can be saved. This corresponds to the number of tile pairs on the figure.

D. Recursive Karatsuba

The multiplication schemes above can also be applied recursively (it was actually the original formulation [1]), i.e., each multiplier is split into smaller multiplications where each of the smaller multiplication is itself split into smaller multiplications, etc.

Consider the $4W \times 4W$ multiplication again. It can be realized by using three multiplications of up to $(2W + 1) \times (2W + 1)$ using the 2-part splitting in (2). Each of these $(2W + 1) \times (2W + 1)$ multiplications can then be realized by using three $(W+1)\times(W+1)$ multiplications by applying



(a) 68×68 ($4W \times 4W$, $W = 17$) (b) 102×102 ($6W \times 6W$, $W = 17$) (c) 119×119 ($7W \times 7W$, $W = 17$)

Fig. 2: Tilings using the conventional Karatsuba with square 17×17 bit multipliers

another 2-part splitting. This leads to a total of only nine $(W+1) \times (W+1)$ multiplications (compared to ten in the 4-part splitting). However, this comes at the price of additional critical path delay or latency, as two pre-additions have to be performed in sequence.

In other words, in hardware, the choice between K-part splitting or its recursive application exposes a trade-off between latency and number of multipliers.

III. Karatsuba with Rectangular Multipliers

A. Generalized Square Karatsuba

Karatsuba works by pairing two multipliers $a_i b_j$ and $a_j b_i$ which have the same weight 2^w in the sum defining the large multiplication. On a tiling representation such as Figure 2, this means they are aligned on a NW-SE diagonal.

A first observation is that the pattern that results from Karatsuba can be arbitrary shifted on the rectangle. In other words it is possible to pair any tile that share the same weight: as soon as $i + j = k + \ell = s$, the complete product involves $2^{sW}(a_i b_j + a_k b_\ell)$, and it is therefore possible to use the rewriting

$$a_i b_j + a_k b_\ell = (a_i + a_k)(b_j + b_\ell) - a_i b_\ell - a_k b_j \qquad (11)$$

or

$$a_i b_j + a_k b_\ell = (a_i - a_k)(b_j - b_\ell) + a_i b_\ell + a_k b_j \qquad (12)$$

assuming $a_i b_\ell$ and $a_k b_j$ are already computed. Note that these formulae generalize (7) and (4).

Graphically, this can be viewed as applying Karatsuba on a sub-square of the large multiplier. This generalization is in general less interesting than the classical K-part decomposition, because the latter exposes more reuse of the diagonal subproducts. However it could be used to apply Karatsuba technique to non-square large multipliers: rectangular ones, or truncated multipliers, for instance.

This generalization is also key to understanding how the Karatsuba technique can reuse rectangular sub-products.

B. Weight-aligning Rectangular Tiles

In order to use (11) or (12) when the products are rectangular ones, they need to have the same binary weight. The trick is therefore to find splittings of the two inputs that expose such alignment.

Consider the splitting of argument A in W_A-bit chunks, and argument B into W_B-bit ones. The weight of $a_i b_j$ is now $2^{W_A i + W_B j}$. Therefore, for $a_i b_j$ and $a_k b_\ell$ to share the same weight, we must have $W_A i + W_B j = W_A k + W_B \ell$ for $i \neq k$ and $j \neq \ell$. We can rewrite this as

$$W_A N - W_B M = 0 \text{ with } N, M \in \mathbb{N} \; . \qquad (13)$$

Now, the smallest values of N and M to satisfy (13) are obtained by $M = W_A/W$ and $N = W_B/W$ where W is the greatest common divisor (GCD) of W_A and W_B, i.e.,

$$W = \gcd(W_A, W_B) \; .$$

Hence, the first common weight that appears is 2^{WNM}. Thus, the smallest multiplier for which Karatsuba can be applied is of size $WM(N+1) \times WN(M+1)$.

For example, consider a tile exactly matching the Xilinx DSP block in unsigned mode: $W_A = 17$ and $W_B = 24$. As W_A and W_B are relatively prime, we have $W = 1$, $M = 17$ and $N = 24$. Relation (13) can only be satisfied for N being multiples of 24 and M being multiples of 17, which is only useful for very large multipliers starting from $17 \cdot (24+1) \times 24 \cdot (17+1) = 425 \times 432$ bits.

However, if we accept to use the slightly smaller tile size 16×24 (which means under-using the DSP blocks, but much less than if we use them as 17×17), then we get a much larger GCD $W = \gcd(16, 24) = 8$, with $M = 16/8 = 2$ and $N = 24/8 = 3$. The minimum large multiplier for which we may apply Karatsuba is now $8 \cdot 2 \cdot (3+1) \times 8 \cdot 3 \cdot (2+1) = 64 \times 72$. This specific example is shown in Fig. 3(a). On this figure, we made the choice that the index of each A chunks is $2i$, and the index of each B chunk is $3j$, which makes it easier to spot the situations where $2i + 3j = 2k + 3\ell$.

Before we further look into specific implementation issues, we will first analyze different large multiplier sizes and their corresponding amount of small multipliers by using their polynomial representation.

IV. Polynomial Interpretation

Fast multiplication algorithms are often described as polynomial multiplication [12], [13]. For instance (8) can be read as the product of two polynomials in the variable $x = 2^W$. The product of polynomials of similar orders corresponds to large multipliers that are approximately square. Let us consider a few potentially useful cases.

A large multiplier of size $W(N+1)M \times W(M+1)N$ corresponds to the following polynomial multiplication of an $N+1$-term M-sparse polynomial by an $M+1$-term N-sparse polynomial, both in $x = 2^W$:

$$\left(\sum_{j=0}^{N} a_{Mj}x^{Mj}\right)\left(\sum_{k=0}^{M} b_{Nk}x^{Nk}\right) = \sum_{i=0}^{2MN} c_i x^i. \tag{14}$$

For example, a large multiplier of size $8W \times 9W$, constructed from $2W \times 3W$ tiles, can be expressed as

$$\left(a_0 + a_2x^2 + a_4x^4 + a_6x^6\right)\left(b_0 + b_3x^3 + b_6x^6\right) = \sum_{i=0}^{12} c_i x^i. \tag{15}$$

The advantage of this representation is that we can identify the left-hand side and the right-hand side of (15), which provides the values of the polynomial coefficients:

$$\begin{aligned}
c_0 &= a_0b_0\\
c_1 &= 0\\
c_2 &= a_2b_0\\
c_3 &= a_0b_3\\
c_4 &= a_4b_0\\
c_5 &= a_2b_3\\
c_6 &= a_0b_6 + a_6b_0\\
c_7 &= a_4b_3\\
c_8 &= a_2b_6\\
c_9 &= a_6b_3\\
c_{10} &= a_4b_6\\
c_{11} &= 0\\
c_{12} &= a_6b_6
\end{aligned} \tag{16}$$

which shows where Karatsuba can be applied: one multiplier can be saved for c_6 on this example, with the Karatsuba identity $a_0b_6 + a_6b_0 = (a_0 - a_6)(b_0 - b_6) + c_0 + c_{12}$. This product requires eleven multipliers instead of twelve for the direct evaluation. The polynomial point of view also tells us that this is the optimal value, since there are eleven non-zero c_i coefficients.

In general, such an algorithm can be derived that requires $MN + M + N$ multipliers, which is one less compared to the direct evaluation.

Another case is constructing a large multiplier of size $2WNM \times 2WMN$. Here, the polynomial multiplication can be written as

$$\left(\sum_{j=0}^{2N-1} a_{Mj}x^{Mj}\right)\left(\sum_{k=0}^{2M-1} b_{Nk}x^{Nk}\right) = \left(A_0(x) + A_1(x)x^{MN}\right)\left(B_0(x) + B_1(x)x^{MN}\right), \tag{17}$$

where

$$\begin{aligned}
A_0(x) &= \sum_{j=0}^{N-1} a_{Mj}x^{Mj}\\
A_1(x) &= \sum_{j=N}^{2N-1} a_{Mj}x^{Mj-MN}\\
B_0(x) &= \sum_{k=0}^{M-1} b_{Nk}x^{Nk}\\
B_1(x) &= \sum_{k=M}^{2M-1} b_{Nk}x^{Nk-MN}.
\end{aligned} \tag{18}$$

Using a polynomial version of (2), (17) can be expressed using only three polynomial multiplications, each of which requires MN multiplier tiles, so in total $3MN$ instead of $4MN$ multiplier tiles.

The last considered case is a large multiplier of size $W(2N+1)M \times W(2M+1)N$. This can be written similar to (17), but with

$$\begin{aligned}
A_1(x) &= \sum_{j=N}^{2N} a_{Mj}x^{Mj-MN}\\
B_1(x) &= \sum_{k=M}^{2M} b_{Nk}b^{Nk-MN},
\end{aligned} \tag{19}$$

i.e., one more term in each of the $A_1(x)$ and $B_1(x)$ polynomials. Now, again, three polynomial multiplications can be computed based on (2). The multiplication $A_0(x)B_0(x)$ requires MN multipliers as above. However, for $(A_0(x) + A_1(x))(B_0(x) + B_1(x))$ and $A_1(x)B_1(x)$, it is now possible to use the approach for $W(N+1)M \times W(M+1)N$ multipliers leading to $MN+M+N$ multipliers. Besides, the product $a_{2MN}b_{2MN}$ appears in both polynomial multiplications, and, hence, only must be computed once. In total, a $W(2N+1)M \times W(2M+1)N$ multiplication require $3MN + 2M + 2N - 1$ rectangular multipliers instead of $4MN + 2M + 2N + 1$ for direct evaluation.

More complexity results for rectangular multipliers are summarized in Table I. The lower bounds are based on the number of non-zero coefficients in the resulting polynomial. It should be noted that the lower bounds can always be obtained, but sometimes the pre- and post-computation will become more complex. See, e.g., three-term multiplication with five square multipliers [14]. It requires division by three, but this operation is quite cheap on FPGAs [15]. This remains to be studied in detail.

TABLE I: Number of rectangular $WM \times WN$ tiles for different multiplier sizes. N, M relative prime.

Case	Large multiplier size ($\times W$)	Number of tiles of size $WM \times WN$		
		Tiling	Karatsuba	Lower bound
a	$NM \times MN$	MN	-	MN
b	$(N+1)M \times (M+1)N$	$MN+M+N+1$	$MN+M+N$	$MN+M+N$
c	$2NM \times 2MN$	$4MN$	$3MN$	$3MN$
d	$(2N+1)M \times (2M+1)N$	$4MN+2M+2N+1$	$3MN+2M+2N-1$	$3MN+M+N$

TABLE II: Karatsuba cases for M and N when mapped to Xilinx DSP blocks

M	N	W	Tile size	Case	Large mult. size	Small mult.
2	3	8	16×24	b	64×72	11
3	4	6	18×24	b	90×96	19
2	3	8	16×24	c	96×96	18
3	5	5	15×25	b	90×100	23
2	3	8	16×24	d	112×120	27
3	4	6	18×24	c	132×132	36
3	5	5	15×25	c	150×150	45
3	4	6	18×24	d	160×168	49
3	5	5	15×25	d	165×175	60
17	24	1	17×24	b	425×432	449

V. Design of Concrete Multipliers

In the following, we consider the design of concrete multipliers targeting recent Xilinx FPGAs with DSP blocks providing a 18×25 signed multiplication.

A. Discussion on Other Possible Rectangular Tilings

A natural question to answer is: are there other possibilities of rectangular tilings like the 16×24 tiling discussed in Section III-B that would lend themselves to Karatsuba pairing?

We considered the following tiles:

- 18×24 (with one line of the multiplication performed using logic resources). This leads to $W = \gcd(18, 24) = 6$, $M = 3$ and $N = 4$.
- 15×25 (under-using the DSP in one dimension, and again with one line of the multiplication performed using logic resources). This leads to $W = 5$, $M = 3$ and $N = 5$.

Figure 3 shows the smallest possible multipliers on which Karatsuba can be applied with these tiles. A detailed comparison of the resulting number of small multipliers when using one of the different options for M, N and W and the cases of Table I is given in Table II, sorted by the resulting large multiplier size. We conclude that Karatsuba opportunities are higher for the 16×24 configuration, and select this tile for actual evaluation in the rest of this work.

Besides, this configuration is also the simplest one to use as each tile uses just one DSP block without additional logic. As the result from the pre-addition grows in word size by one bit, it would require a 17×25 (unsigned) multiplication. However, it is simple to circumvent this by selecting the alternative form given in (12), which leads to subtractions in the pre-processing. The multiplier is now 17×25 signed, and this fits very well the 18×25 signed multiplier available in the DSP block.

TABLE III: Operation counts for similar multiplier sizes

Square				Rectangular				
	Karatsuba				Tiling	Karatsuba		
Size	Mult	Pre-add	Post-add	Size	Mult = Post-add	Mult	Pre-add	Post-add
51×51	6	6	6	48×48	6	6	0	6
68×68	10	12	22	64×72	12	11	2	13
102×102	21	30	51	96×96	24	18	5	30
119×119	28	42	70	112×120	35	27	7	43

B. Results on Concrete Multipliers

We designed several multipliers using the classical Karatsuba using square tiles, the conventional tiling (without any reduction) using rectangular tiles as well as the proposed Karatsuba method using rectangular tiles. The size of the square tiles was selected to be 17×17 and the rectangular tiles was selected to be 16×24 as discussed above. The sizes of the large multipliers were selected to be as close to each other as possible to achieve a direct comparison. The resulting tilings are shown in Fig. 4 and their concrete coefficients c_i are given in appendix. Table III compares the operation counts in terms of their used amount of small multipliers, pre-adders and post-adders for all three methods.

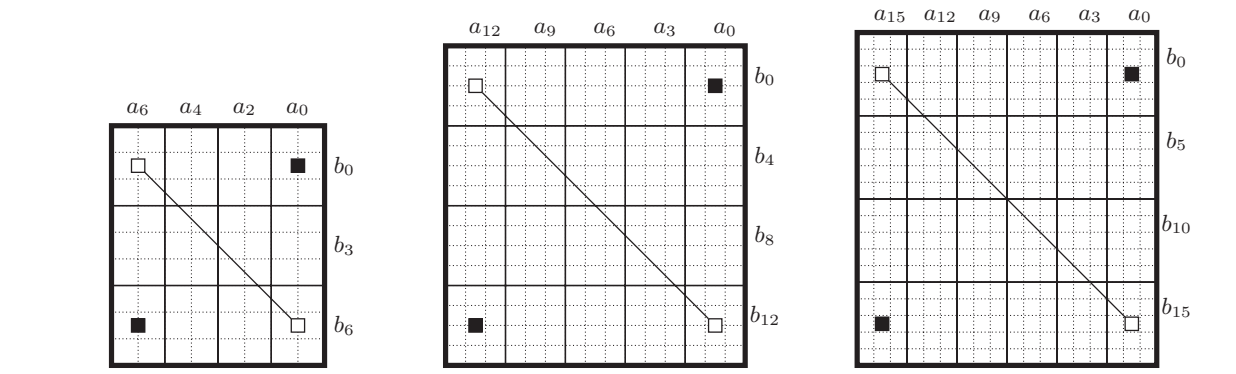
The proposed Karatsuba method using rectangular tiles requires the least multipliers except in the 68×68 case, where classic Karatsuba uses one small multiplier less.

In terms of adders, the proposed method also significantly improves the numbers of pre- and post-adders. The main reason for this is of course the larger tile size, which leads to fewer sub-products, hence pre-adders.

Note that for the larger multipliers using the proposed method, many of the pre-adders can be shared as they compute the same values. For the 96×96, only 5 out of 12 pre-additions compute different values. For the 112×120 multipliers, only 7 pre-additions are required out of 16 (see appendix). For the traditional Karatsuba, no sharing is possible as each pre-addition only appears once.

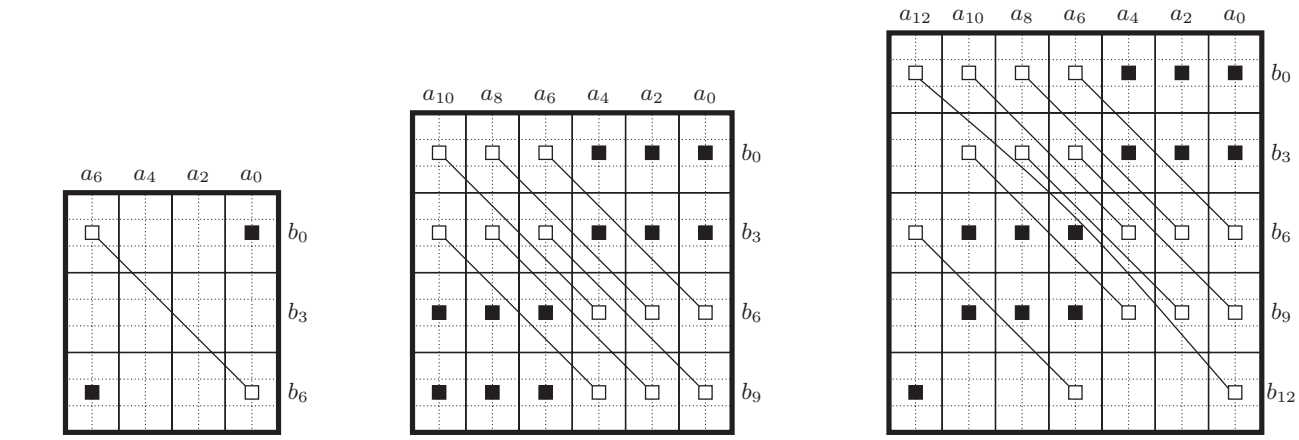
VI. Synthesis Results

Table III shows the expected trade-off between multipliers and adders used for pre- and post-processing. To be able to compare the actual performance and resource consumption (including pipeline resources etc.), we designed a VHDL code generator for the multipliers using the different methods



(a) $M = 2$, $N = 3$, $W = 8$ (tile size: 16×24, pattern size: 64×72)
(b) $M = 3$, $N = 4$, $W = 6$ (tile size: 18×24, pattern size: 90×96)
(c) $M = 3, N = 5, W = 5$ (tile size: 15×25, pattern size: 90×100)

Fig. 3: Minimal Karatsuba patterns with various rectangular tiles matching Xilinx DSP blocks.



(a) 64×72 ($8W \times 9W$, $W = 8$)
(b) 96×96 ($12W \times 12W$, $W = 8$)
(c) 112×120 ($14W \times 15W$, $W = 8$), see Appendix C

Fig. 4: Rectangular Karatsuba tilings selected for synthesis. They mostly use 16×24 tiles, replicating the pattern of Fig. 3(a)

discussed above. It was integrated into the FloPoCo open-source code generator [16]. FloPoCo also provides a state-of-the-art framework to create and optimize compressor trees [17], [18]. It was used to implement all the post-adders within a single compressor tree. For each Karatsuba sub-product, one of the pre-adders was implemented using the pre-adder within the DSP block, while the other was implemented using logic. This further reduces about half of the pre-adders of Table III that are not shared.

Table IV shows synthesis results on Xilinx Virtex-6 using ISE. Results for Kintex-7 using Vivado are similar. As expected from Table III, the tiling approach typically requires the least logic resources in terms of look-up tables (LUTs) but the most DSP resources. Using the conventional Karatsuba with square multipliers helps reducing the DSP count but at the cost of a highly increased LUT count: more than 50% compared to the tiling approach. The proposed rectangular Karatsuba reduces the DSP count compared to the tiling approach with only a slight increase in logic resources. As a consequence, the latency (number of cycles) is also reduced.

VII. Conclusion and Future Work

In this work, an extension of Karatsuba's method was presented that exploits rectangular-shaped multipliers. Necessary conditions were derived showing that Karatsuba's method can be applied when the word lengths of the two inputs of these multipliers have a large GCD. Different large multipliers were designed for the general case, showing the benefits of the proposed extension compared to the conventional tiling. Based on that, several concrete multipliers were implemented for recent Xilinx FPGAs providing 18×25-bit signed multipliers in their DSP blocks. With this technique, a significant reduction in arithmetic operations (multipliers and adders) is possible. Synthesis experiments showed that this also predictively translates to a reduction of DSP blocks and logic resources.

The proposed technique is probably not useful when targeting ASIC or software, although it could find applications in the multiplication of polynomials with very specific sparseness properties.

Future work will be directed towards the further exploration

TABLE IV: Actual synthesis results obtained for Virtex 6 FPGA (xc6vlx760-ff1760) using Xilinx ISE 13.4, post place&route

Square					Rectangular								
	Karatsuba					Tiling				Karatsuba			
Size	DSPs	LUTs	Cycles	$f_{\max}$ [MHz]	Size	DSPs	LUTs	Cycles	$f_{\max}$ [MHz]	DPSs	LUTs	Cycles	$f_{\max}$ [MHz]
68×68	10	1405	11	215.1	64×72	12	764	10	217.5	11	867	10	247.0
102×102	21	2524	13	192.0	96×96	24	1586	13	215.1	18	2032	14	195.1
119×119	28	3438	15	192.6	112×120	35	2293	16	218.9	27	2292	14	190.1

of possible large multiplier sizes, and exploring more formulae [12]. Another goal is a fully automated design flow that selects the best fitting tile, integrated in a generic multiplier optimization framework [3], [9].

REFERENCES

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

[2] *UltraScale Architecture DSP Slice User Guide (v1.6)*, Xilinx Corporation, 2018.

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

[4] B. Pasca, "High-performance floating-point computing on reconfigurable circuits," Ph.D. dissertation, École Normale Supérieure de Lyon, 2012.

[5] G. I. Malaschonok and E. Satina, "Fast multiplication and sparse structures," *Programming and Computer Software*, vol. 30, no. 2, pp. 105–109, 2004.

[6] J. Van Der Hoeven and G. Lecerf, "On the bit-complexity of sparse polynomial and series multiplication," *Journal of Symbolic Computation*, vol. 50, pp. 227–254, 2013.

[7] 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*. Springer, 2013, pp. 226–237.

[8] 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.

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

[10] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

[11] M. K. Jaiswal and R. C. C. Cheung, "VLSI Implementation of Double-Precision Floating-Point Multiplier Using Karatsuba Technique," *Circuits, Systems, and Signal Processing*, vol. 32, no. 1, pp. 15–27, Jul. 2012.

[12] P. L. Montgomery, "Five, six, and seven-term Karatsuba-like formulae," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 362–369, Mar. 2005.

[13] R. Brent and P. Zimmermann, *Modern Computer Arithmetic*. Cambridge University Press, 2010.

[14] M. Bodrato and A. Zanoni, "Integer and polynomial multiplication: Towards optimal toom-cook matrices," in *Symbolic and Algebraic Computation*. ACM, 2007, pp. 17–24.

[15] H. F. Ugurdag, F. de Dinechin, Y. S. Gener, S. Gren, and L.-S. Didier, "Hardware division by small integer constants," *IEEE Transactions on Computers*, vol. 66, no. 12, pp. 2097–2110, 2017.

[16] F. de Dinechin and B. Pasca, "Custom arithmetic datapath design for FPGAs using the FloPoCo core generator," *IEEE Design & Test of Computers*, no. 99, pp. 1–1, 2012.

[17] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic core generation using bit heaps," in *Field Programmable Logic and Application (FPL)*. IEEE, 2013, pp. 1–8.

[18] M. Kumm and J. Kappauf, "Advanced compressor tree synthesis for FPGAs," *IEEE Transactions on Computers*, pp. 1–1, 2018.

APPENDIX

A. Coefficients for $8W \times 9W = 64 \times 72$

$$
\begin{aligned}
c_0 &= a_0b_0 \\
c_1 &= 0 \\
c_2 &= a_2b_0 \\
c_3 &= a_0b_3 \\
c_4 &= a_4b_0 \\
c_5 &= a_2b_3 \\
c_6 &= a_0b_6 + a_6b_0 = (a_0 - a_6)(b_6 - b_0) + c_0 + c_{12} \\
c_7 &= a_4b_3 \\
c_8 &= a_2b_6 \\
c_9 &= a_6b_3 \\
c_{10} &= a_4b_6 \\
c_{11} &= 0 \\
c_{12} &= a_6b_6
\end{aligned}
$$

B. Coefficients for $12W \times 12W = 96 \times 96$

$$
\begin{aligned}
c_0 &= a_0b_0 \\
c_1 &= 0 \\
c_2 &= a_2b_0 \\
c_3 &= a_0b_3 \\
c_4 &= a_4b_0 \\
c_5 &= a_2b_3 \\
c_6 &= a_0b_6 + a_6b_0 = (a_0 - a_6)(b_6 - b_0) + c_{12} + c_0 \\
c_7 &= a_4b_3 \\
c_8 &= a_2b_6 + a_8b_0 = (a_2 - a_8)(b_6 - b_0) + c_2 + c_{14} \\
c_9 &= a_0b_9 + a_6b_3 = (a_0 - a_6)(b_9 - b_3) + c_3 + c_{15} \\
c_{10} &= a_4b_6 + a_{10}b_0 = (a_4 - a_{10})(b_6 - b_0) + c_4 + c_{16} \\
c_{11} &= a_2b_9 + a_8b_3 = (a_2 - a_8)(b_9 - b_3) + c_5 + c_{17} \\
c_{12} &= a_6b_6 \\
c_{13} &= a_4b_9 + a_{10}b_3 = (a_4 - a_{10})(b_9 - b_3) + c_7 + c_{19} \\
c_{14} &= a_8b_6 \\
c_{15} &= a_6b_9 \\
c_{16} &= a_{10}b_6 \\
c_{17} &= a_8b_9 \\
c_{18} &= 0 \\
c_{19} &= a_{10}b_9
\end{aligned}
$$

C. Coefficients for $14W \times 15W = 112 \times 120$

Terms which are conflicting each other are marked with double underlines. Terms in square brackets are not realized due to conflicts.

$$
\begin{aligned}
c_0 &= a_0 b_0 \\
c_1 &= 0 \\
c_2 &= a_2 b_0 \\
c_3 &= a_0 b_3 \\
c_4 &= a_4 b_0 \\
c_5 &= a_2 b_3 \\
c_6 &= a_0 b_6 + a_6 b_0 = (a_0 - a_6)(b_6 - b_0) + a_6 b_6 + a_0 b_0 \\
c_7 &= a_4 b_3 \\
c_8 &= \underline{a_2 b_6} + a_8 b_0 = (a_2 - a_8)(b_6 - b_0) + a_2 b_0 + \underline{a_8 b_6} \\
c_9 &= a_0 b_9 + \underline{a_6 b_3} = (a_0 - a_6)(b_9 - b_3) + a_0 b_3 + \underline{a_6 b_9} \\
c_{10} &= \underline{a_4 b_6} + a_{10} b_0 = (a_4 - a_{10})(b_6 - b_0) + a_4 b_0 + \underline{a_{10} b_6} \\
c_{11} &= a_2 b_9 + a_8 b_3 = (a_2 - a_8)(b_9 - b_3) + a_2 b_3 + a_8 b_9 \\
c_{12} &= a_0 b_{12} + a_{12} b_0 + a_6 b_6 \\
c_{13} &= a_4 b_9 + a_{10} b_3 = (a_4 - a_{10})(b_9 - b_3) + a_4 b_3 + a_{10} b_9 \\
c_{14} &= a_2 b_{12} + \underline{a_8 b_6} \left[= (a_2 - a_8)(b_{12} - b_6) + \underline{a_2 b_6} + a_8 b_{12} \right] \\
c_{15} &= \underline{a_6 b_9} + a_{12} b_3 \left[= (a_6 - a_{12})(b_9 - b_3) + \underline{a_6 b_3} + a_{12} b_9 \right] \\
c_{16} &= a_4 b_{12} + \underline{a_{10} b_6} \left[= (a_4 - a_{10})(b_{12} - b_6) + \underline{a_4 b_6} + a_{10} b_{12} \right] \\
c_{17} &= a_8 b_9 \\
c_{18} &= a_6 b_{12} + a_{12} b_6 = (a_6 - a_{12})(b_{12} - b_6) + a_6 b_6 + a_{12} b_{12} \\
c_{19} &= a_{10} b_9 \\
c_{20} &= a_8 b_{12} \\
c_{21} &= a_{12} b_9 \\
c_{22} &= a_{10} b_{12} \\
c_{23} &= 0 \\
c_{24} &= a_{12} b_{12}
\end{aligned}
$$