Parametric Throughput Oriented Large Integer Multipliers for High Level Synthesis

Emanuele Vitali, Davide Gadioli, Fabrizio Ferrandi, Gianluca Palermo
Dipartimento di Elettronica Informazione e Bioingegneria - Politecnico di Milano

Abstract—**The multiplication of large integers represents a significant computation effort in some cryptographic techniques. The use of dedicated hardware is an appealing solution to improve performances or efficiency. We propose a methodology to generate throughput oriented hardware accelerators for large integers multiplication leveraging High-Level Synthesis. The proposed micro-architectural template combines Karatsuba and Comba algorithms to control the extra-functional properties of the generated multiplier. The goal is to enable the end user to explore a wide range of possibilities, in terms of performance and resource utilization, without requiring them to know implementation and synthesis details. Experimental results show the large flexibility of the generated architectures and that the generated Pareto-set of multipliers can outperform some state-of-the-art RTL design.**

I. Introduction

THE increasing concern for security and safety [1] led to a more widespread usage of cryptographic techniques and an increase of their complexity. The multiplication between large integers is a common operation in this contest. As an example, the work in [2] uses the Pailler cryptosystem to perform data aggregation without decryption at intermediate hops in a network. However, this operation requires a significant computation effort, promoting the usage of efficient and optimized hardware component implementations to improve its performance, even if they have limited customization opportunities. The latter aspect is of paramount importance, when the component usage may have different requirements in terms of energy efficiency and performance. Therefore, researchers have spent a significant effort to investigate hardware accelerators for the multiplication of large numbers [3]–[7].

In this paper, we propose a methodology to generate a throughput oriented hardware accelerator for large integers multiplication. By customizing high-level parameters, the users can search for the best compromise between resource utilization and performance, according to their requirements. We target Field Programmable Gate Arrays (FPGA), to exploit their flexibility.

The main idea behind this paper is to follow the approach of autotuning libraries such as ATLAS, MKL, SPIRAL adopted in High Performance Computing (HPC) [8]–[11]. These libraries expose a single interface, while they tune the actual implementation for the underlying architecture and input data. If we focus on software, the GMP library [12] contains a collection of optimized algorithms. The optimal one is chosen at runtime according to the underlying architecture and operands size. While software multiplications focus on the execution time, a hardware implementation must consider multiple extra-functional properties, such as area consumption and energy efficiency. An interesting work has been proposed by Rafferty et al. [6], where they provides an extensive comparison of different techniques to perform integer multiplications with arbitrary size and inspired us for this work. However, properly programming FPGA devices with RTL, as focused in [6], requires experienced programmers. Since our objective is to provide the possibility to perform large unsigned multiplications to a large audience, we will target a High-Level Synthesis (HLS) oriented approach.

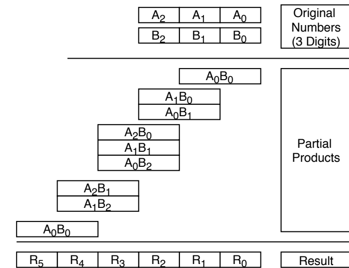


Fig. 1: Algorithm of the Comba Multiplication

We may summarize the contributions of this paper as follows: (I) we propose a parametric approach to generate, using HLS, large integer multipliers; (II) we propose a novel strategy to combine well-known multiplication algorithms; (III) we provide the user the possibility to explore different levels of resource utilization and performances for a single multiplier. Finally, to guarantee the results replicability and to reuse the proposed approach, the library code has been made available[1].

II. Target Class of Multiplication Algorithms

The basic multiplication method, named schoolbook, replicates the paper and pen method that is commonly used. Its drawback is that it requires a very large amount of load and store to update the result when we consider large integers. For this reason, literature investigates alternative multiplication methods. Among all of them, this work targets the intermediate size of operands, where the non-optimality of the direct approach starts to be a problem, and before the size of the operands is too big to justify the NTT approach.

a) Comba Multiplication: This multiplication algorithm, proposed by Comba [13], aims at reducing the number of load-store required to compute the results, without altering the complexity. Figure 1 shows the algorithm of this multiplication technique. It computes partial products between the digits of the operands, and the order of the partial products allows to build directly the final result by shifting and summing the computed partial product with the result. The complexity of the operation

[1] https://gitlab.com/me2x/hls_multipliers

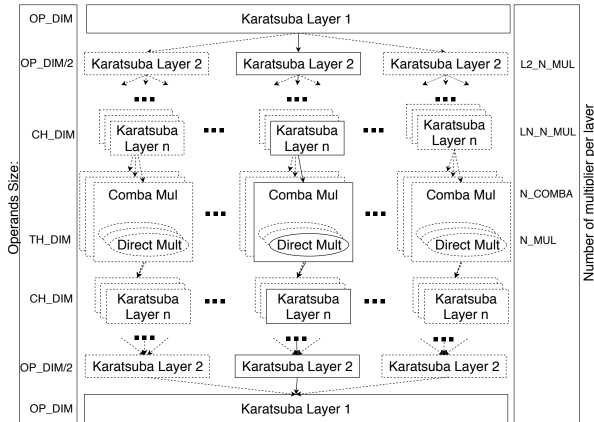


Fig. 2: Architecture Template of the proposed methodology. The components with dotted lines are optional and their instantiation depends from the high-level parameters configuration.

does not change.This algorithm is appealing when there are limited hardware resources [4].

b) Karatsuba Multiplication: This multiplication algorithm, proposed by Karatsuba [14], aims at reducing the complexity of the operation. It splits the operands into two smaller terms and computes the final result as a polynomial multiplication and it can be applied recursively. Therefore, if the complexity of the operation using the previous algorithms is O(n^2), the complexity of the Karatsuba algorithm is O($n^{log_2 3}$). It replaces one n bit multiplication with three $n/2$ bit multiplications and 4 additions. However, recursively applying the algorithm requires a high number of hardware resources [5].

c) Karatsuba-Comba Multiplication: The Karatsuba algorithm is an appealing approach to implement a large integer multiplication, due to the lower complexity of the operation. Therefore, to mitigate the resource requirements, researches investigated the possibility to combine the Karatsuba and the Comba algorithms [6], [15]. Previous works investigate both software implementation [15] and hardware implementation [6]. In particular, the latter focuses on a hardware implementation that performs a single iteration of the Karatsuba algorithm, while it uses the Comba algorithm to compute the three internal multiplications. Experimental results demonstrate that this solution has the lowest latency for operands between 512 and 16384 bits [6].

III. Proposed Methodology

This section describes the proposed methodology and its implementation. In particular, we propose to use a parametric high-level architecture description and to rely on a High Level Synthesis framework for the actual hardware implementation of the multiplier. At first, we describe the generated architecture template and we extract the high-level parameters. Then, we describe how these high-level parameters change the generation of the multiplier. Since we are targeting HLS, we aim at exploiting the DSPs available on the FPGA board. For this reason we will not decompose the operation to the bit-level.

a) Architecture Template: The proposed methodology provides an architectural template that combines the multiplication algorithms described in Section II, allowing the exploration

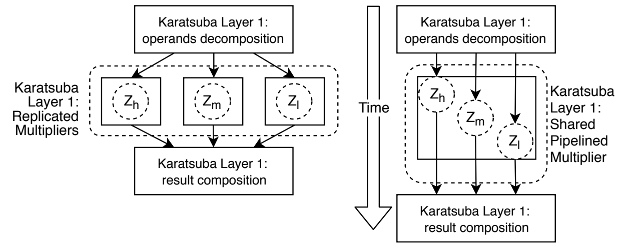


Fig. 3: Karatsuba multiplication layer with replicated (left) and shared (right) multipliers for the inner multiplications.

of a wide range of performance-resources trade-offs. The idea is to use a *divide et impera* approach. We use recursion to reduce the operands bit-width to increase multiplication efficiency, while we use sharing techniques to limit the required amount of logic in the computation. Figure 2 shows the architecture template and highlights the sizes of the operands across the different layers. We split the multiplication algorithm into three phases: (1) Karatsuba Operands Decomposition: we recursively apply the Karatsuba algorithm to reduce the complexity of the multiplication. Every recursion step is named "layer". (2) Products Evaluation using the Comba Algorithm. (3) Karatsuba Result Composition: we recursively apply the sum and shifts required by the Karatsuba algorithm to obtain the result, from the inner layer to the outer.

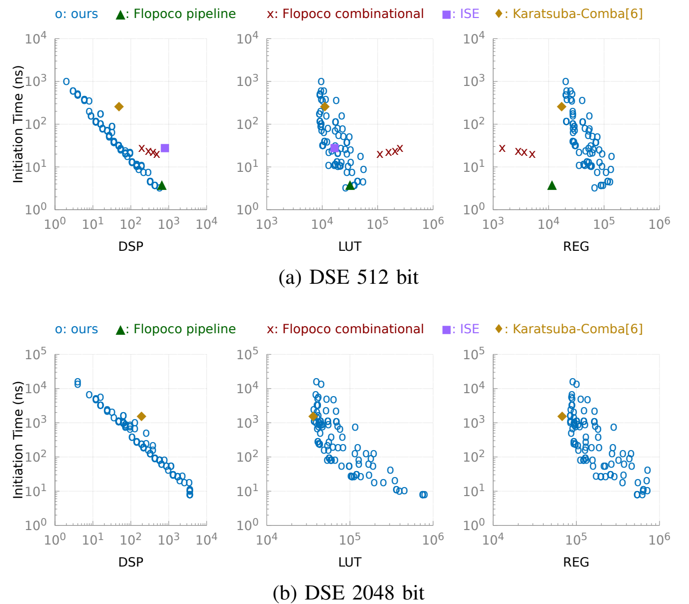
In the "Operand Decomposition" phase, at every layer, the dimension of the data is halved and the three multiplication are instantiated, following the traditional approach for the Karatsuba algorithm. We add, at every layer, a choice that allows the programmer to force the re-use of the inner multiplier to perform all the three multiplications needed by the Karatsuba algorithm in a pipelined fashion. This introduces the possibility to replicate or share resources, for each Karatsuba layer, as depicted in Figure 3. In particular, the dotted circles represent the required multiplications (z_h, z_m and z_l), while the squares represent the instantiated multipliers. On the left side of Figure 3, there is the traditional approach of the Karatsuba multiplication, where all the three multipliers are instantiated. On the right side of Figure 3, we represent the pipelined architecture for a Karatsuba layer, where a single multiplier is created and the three multiplications are pipelined. The idea of forcing the reuse of inner resources enables the trade-off between performance and resource usage. Therefore, we can use this mechanism to balance the initiation intervals of the different layers, to prevent idle inner layers.

When the bit size of the intermediate multiplication terms (i.e. z_h, z_m, z_l) reaches a given threshold, a last layer of Karatsuba instantiates its three multiplications using the Comba algorithm. This is the second phase of the multiplication. Even at this layer, the user can specify the number of instantiated Comba multipliers (1 or 3). We implement the Comba algorithm in a slightly different way with respect to the standard way with one single multiplier that evaluates all the partial products serially. This choice is done to enable parametrization inside this component. In particular, we divide the multiplication into four steps: (1) We split the two operands in digits, according to the size of the Direct multiplication. Each digit is stored in a different variable to enable parallel access. (2)

We compute the partial products independently. According to the number of Direct multipliers, we can modulate the latency of this phase. If this parameter is equal to the number of multiplications, they are computed in parallel. If this parameter is equal to one, it computes the products serially. Otherwise, there is some degree of parallelism in the computation of these products. We use Direct Multipliers because they are mapped directly on DSP. (3) We compute the sums needed to generate the partial products (the sum of all the columns in Figure 1) into separate variables. (4) We reconstruct the final result. We use only sums, shifts and masks to reduce the hardware complexity. This approach allows all the internal variables to be written and read only once, thus leaving all the scheduling decision to the HLS tool and enabling pipelined and/or parallel approaches, if enough resources are allowed for allocation. Finally, in the last phase, we combine the partial results of each Karatsuba layer.

Using this approach, we can expose to the designer, two categories of high-level parameters. The first one is related to the dimensions of the operands, while the second category controls the resource reuse. In particular, the parameters that fall in the first category are: *OP_DIM* (the bit size of the multiplier operands), *CH_DIM* (the bit size threshold of the last Karatsuba layer) *TH_DIM* (the bit size of the direct multiplication, i.e. the digit bit-width in the Comba algorithm). To generate a balanced multiplier, these parameters must be a power of two. The second category of parameters addresses resource utilization. In particular, the high-level parameters that fall in this category are: *L2_N_MUL* (the number of OP_DIM/2 bits Karatsuba Multiplier to instantiate, 1 or 3). *LN_N_MUL* (the number of Karatsuba Multiplier to instantiate inside the *N-th* layer, 1 or 3. The number of these parameters is tied to the number of Karatsuba layers). *N_COMBA* (the number of Comba multipliers in each innermost Karatsuba layer component, 1 or 3) and *N_MUL* (the maximum number of Direct multipliers allowed in the implementation of each Comba). We designed the approach to be agnostic to the HLS engine. However, in the current implementation we use Vivado HLS pragmas to enforce resource-related parameters. It is possible to port the methodology in a different HLS engine by changing this implementation detail.

b) Methodology Implementation: This work aims at generating throughput oriented large integer multipliers, with a wide range of extra-functional properties, in terms of Initiation Time (i.e. the inverse of the throughput) and resource usage. In this way, the end-users can generate the most suitable multiplier according to their requirements. The recursive nature of Karatsuba is a good match for recursive functions. However, they are not supported by most HLS tools. Therefore, instead of using complex code generators, we propose to use C++ templates to solve the recursion at compile-time, generating the actual code in the multiplier declaration. While its usage in the application code is consistent, by changing the template arguments, it is possible to drastically change its hardware implementation, leading to different extra-functional behaviors. Therefore, the high-level parameters described before with the architecture template definition are implemented using variadic C++ templates. To the best of our knowledge, [6] proposes the first implementation of the Karatsuba-Comba multiplier. In particular, it investigates a hardware design that uses a single layer of Karatsuba, to reduce the operands bit-width, and three Comba multipliers to perform the inner products. However, its implementation does not follow a pipeline approach and it exposes limited flexibility. On the contrary, our proposed approach is throughput oriented, and it enables a greater level of flexibility since it enables the reuse of pipelined components. Indeed, by changing high-level parameters, it is possible to set the number of Karatsuba layers. Moreover, it is possible to define for each layer the reuse policies to limit the area of the multiplier, leading to unexplored multiplier architectures.



(a) DSE 512 bit

(b) DSE 2048 bit

Fig. 4: Initiation time of different multipliers, by varying the number of used resources and the operand size. Each row represents a different operand size, while each column represents a different resource. No points mean no multipliers have been generated for that bitwidth size from that methodology.

The actual parametrized interface of the whole library is:

```
multiplier<OP_DIM,CH_DIM,TH_DIM,N_COMBA,N_MUL,...>
  (ap_uint<OP_DIM>A, ap_uint<OP_DIM>B, ap_uint<2*OP_DIM>OUT)
```

Where the variadic template is needed to manage the variable number of *LN_N_MUL* parameters.

IV. Experimental Results

This section evaluates the benefits of the proposed approach. At first, we describe the Design of Experiment that we used to generate the multiplier implementations. Then, we analyze their extra-functional properties in terms of Initiation Time and resource usage at post-place stage. Finally, we compare them with state-of-the-art multipliers. In particular, we consider RTL implementations [6] and several instances generated by FloPoCo [16]. To perform these tasks, we use the tool Vivado HLS 2018.2 and Vivado 2018.2, on a Virtex7 xc7vx980t.

a) Design of Experiments: The methodology aims at generating multipliers with a wide trade-off space between performance and resource usage, given the target operands size. Table I reports for all the considered operands dimensions (512 and 2048 bits) the values of the explored parameters

TABLE I: High-level parameters values explored in the DSEs. OP: operand dimension, LM: number of sub-multiplier per layer, NC: number of Comba, NM: number of Direct. Frequency is in MHz.

OP	CH_DIM	TH_DIM	LM	NC	NM	Freq
2048	1024 512 256 128	128 64 32	1 3	1 3	1 2 4	100
512	512 256 128 64	64 32 16	1 3	1 3	1 2 4	250

that influence the architecture. We performed a DSE for all of these bit-widths, combining in a full factorial DoE these parameters. The idea is to explore up to five Karatsuba layers with different sharing policies and characteristics of direct multipliers. The Design Space dimensions are 270 candidates for 512 bits multiplier and 540 for the 2048. We select the target frequency to compare fairly with the existing solutions [6]. Once the DSE is completed, we performed Pareto filtering to remove the dominated solutions.

b) Design Space Analysis: Figure 4 shows the result of the Design Space Exploration of multipliers. For each operand size, we report the trade-off between the Initiation Time and the target resource. The Initiation Time is the time elapsed between the start of two pipelined operations, i.e. the inverse value of the Throughput. In particular, the first column shows the DSP consumption, the second the LUT usage, and the third shows the Registers usage. All the reported plots use a logarithmic scale. If we focus only on the multipliers generated by the proposed approach, we can notice how the Initiation Time spans over 3 orders of magnitude, for all the operands size. Moreover, there is a strong correlation between Initiation Time and the number of DSP, where the lowest Spearman correlation coefficient among the different operand size is -0.96, with a p-value smaller than 0.0001. Even if we expected these results, it shows how the methodology can provide to the end-user the multiplier that best fits its requirements, in terms of performance and area utilization. For example, in all the operand sizes that we analyzed, the end-user can always choose between a fast multiplier (Initiation Time lower than $10ns$) and a small one (less than 10 DSP).

Comparison with state-of-the-art multipliers: FloPoCo is a VHDL generator of arithmetic cores. It mainly targets small operands, however it can generate large integer multipliers. Moreover, the library supports the generation of throughput-oriented pipelined components. We compared our multipliers with FloPoCo version 5.0. We generate different multipliers by changing the amount of available DSP, including one without restrictions. When the tool fails to generate a pipelined component, for example when we limit the amount of DSP, it will fall back to a combinational component. Rafferty et al. [6] present two optimal hand-optimized multiplier designs targeting the 512-2048 bit-width, the first belonging to the Xilinx ISE library and the second is a contribution. In particular, this work demonstrates that the ISE instantiated multiplier and the Karatsuba-Comba design have the best throughput according to the size of the operands (ISE up to 512 bit, Karatsuba-Comba above). Figure 4 reports the result of the DSE. A deep exploration of resource parameters is difficult using FloPoCo. The program is pretty slow in generating the RTL. The time required for 512 bit multipliers is a couple of days and it was unable to produce any 2048 bit multiplier after two weeks of computation. This is the reason why Figure 4b has no FloPoCo solution. These times are several orders of magnitude higher by the minutes required by the proposed methodology to obtain the RTL. FloPoCo has in average a worst DSP and LUT utilization compared to the ones generated by the proposed methodology, while it has a better Register consumption, the ISE multiplier is dominated in all but Registers and also the Karatsuba-Comba from [6] is unable to reach the same throughput of the proposed methodology at a similar DSP cost. However, we can notice that all of these methodologies perform better in the utilization of LUT and Registers. This behavior is due to the HLS procedure that is unable to reach the same optimization level when it translates high-level languages into hardware description. However, all these multipliers are not flexible and do not allow any performance-resource trade-off exploration. Indeed, we tried to perform a similar DSE with FloPoCo by constraining the DSP usage However, it uses LUT to replace DSP to generate architectures with a similar level of throughput, thus being unable to span on the large pareto front created by our methodology.

V. Conclusions

This work proposes a methodology to generate a tunable large integer multipliers using HLS. It uses a parametric Karatsuba-Comba multiplication template to instantiate throughput oriented multipliers. From experimental results, we can notice how the proposed methodology provides a wide flexibility so that the end-user can select the most suitable multiplier, according to the application requirements.

References

[1] M. Duranton, K. De Bosschere, C. Gamrat, J. Maebe, H. Munk, and O. Zendra, "The HiPEAC vision 2017," 2017.

[2] R. Lu, X. Liang, X. Li, X. Lin, and X. Shen, "Eppa: An efficient and privacy-preserving aggregation scheme for secure smart grid communications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 1621–1631, Sep. 2012.

[3] G. C. T. Chow, K. Eguro, W. Luk, and P. Leong, "A Karatsuba-based Montgomery multiplier," in *FPL 2010*, Aug 2010, pp. 434–437.

[4] L. Malina and J. Hajny, "Accelerated modular arithmetic for low-performance devices," in *TSP 2011*. IEEE, 2011.

[5] N. Nedjah and L. de Macedo Mourelle, "A review of modular multiplication methods ands respective hardware implementation," *Informatica*, vol. 30, 2006.

[6] C. Rafferty, M. O'Neill, and N. Hanley, "Evaluation of large integer multiplication methods on hardware," *IEEE Transactions on Computers*, vol. 66, pp. 1369–1382, Aug 2017.

[7] 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)*. IEEE, 2018, pp. 13–20.

[8] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, pp. 3–35, 2001.

[9] Intel. Math kernel library.

[10] M. Frigo and S. G. Johnson, "FFTW: an adaptive software architecture for the FFT," in *ICASSP*, 1998.

[11] J. M. F. Moura *et al.*, "SPIRAL: Automatic implementation of signal processing algorithms," in *High Performance Extreme Computing*, 2000.

[12] T. Granlund and G. D. Team, *GNU MP 6.0 Multiple Precision Arithmetic Library*. United Kingdom: Samurai Media Limited, 2015.

[13] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Syst. J.*, vol. 29, pp. 526–538, Oct. 1990.

[14] A. Karatsuba, "Multiplication of multidigit numbers on automata," in *Soviet physics doklady*, vol. 7, 1963, pp. 595–596.

[15] J. Großschädl, R. M. Avanzi, E. Savaş, and S. Tillich, "Energy-efficient software implementation of long integer modular arithmetic," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, pp. 75–90.

[16] F. de Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," in *FPL 2009*. IEEE, 2009, pp. 250–255.