IMpress: Large Integer Multiplication Expression Rewriting for FPGA HLS

Ecenur Ustun[1], Ismail San[2,1], Jiaqi Yin[3], Cunxi Yu[3], Zhiru Zhang[1]
[1]School of Electrical and Computer Engineering, Cornell University, Ithaca, NY, USA
[2]Department of Electrical and Electronics Engineering, Eskişehir Technical University, Eskişehir, Turkey
[3]Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT, USA
{eu49, zhiruz}@cornell.edu

***Abstract*—Large integer multiplication is becoming a major challenge for FPGA-based acceleration of many cryptographic applications. Existing techniques for decomposing and optimizing large integer multiplication bring about nontrivial trade-offs between different resource types as well as performance. In this work, we regard determining the level and order of multiplication decomposition as a phase ordering problem, which is a notable problem in compiler optimization. Our framework, IMpress, leverages equality saturation to automatically produce a wide range of equivalent integer multiplication expressions corresponding to various hardware implementations. We devise constrained and multi-objective extraction techniques to automatically choose the optimal expressions based on the resource requirements of a given application. IMpress automatically translates extracted integer multiplication expressions into behavioral descriptions in C++ and initiates FPGA compilation through high-level synthesis. IMpress offers significant control over resource utilization and balance, and it increases the maximum number of instances of cryptographic applications on FPGA.**

I. Introduction

Recent years have seen an increasing number of FPGA-based accelerators to improve the performance and energy efficiency of various cryptographic algorithms and applications such as RSA [1], [2], [3], elliptic-curve cryptography (ECC) [4], [5], and homomorphic encryption (HE) [6], [7], [8], [9], [10]. These cryptographic applications have high arithmetic intensity due to the extensive use of large integer multiplications that take on operands with hundreds or thousands of bits. However, it remains challenging to efficiently implement large integer multiplication on FPGAs due to the lack of efficient libraries and compiler support.

There are a number of techniques for generating high-performance and energy-efficient integer multipliers on FPGAs, such as Karatsuba decomposition [11], [12], [13], [14], [15] and number-theoretic transform (NTT) [16], [12], [17]. Each decomposition technique brings about a trade-off between different resource types (LUTs and DSP blocks) and performance. For instance, Karatsuba reduces the number of partial products at the cost of more addition operations, which results in a DSP-LUT trade-off. Deeper levels of decomposition intensify this trade-off. Given the area and performance requirements of an application, determining the optimal level and order of decomposition is a nontrivial task, which has not been addressed by the previous work.

Determining the optimal level and order of decomposition rules for large integer multiplication can be considered as a phase ordering problem, which is frequently encountered in compiler optimizations. A certain decomposition rule may lead to lower quality-of-results (QoR) at a given level. However, being followed by other decomposition rules in the subsequent levels, one can eventually achieve an integer multiplication implementation with a high quality. In other words, a transformation which is locally suboptimal may lead to globally optimal results based on a single objective or Pareto-optimal results based on multiple objectives, after being followed by other transformations, or vice versa.

Equality saturation is a promising solution to addressing some of the phase ordering problems in compiler optimization [18], [19]. Given an input program, equality saturation constructs an *e-graph*, a graph-based data structure, by repeatedly applying a set of rewrite rules that do not change the functionality of the program. After reaching saturation (or timeout), the resulting e-graph will represent, often in a compact way, a large set of equivalent expressions of the input program. The rewrite can be a compiler transformation or in our case a decomposition rule for integer multiplication. Since these rewrites only add information to the e-graph, careful phase ordering is not required.

In this work we propose *IMpress*, an integer multiplication expression rewriting framework that exploits the expressive power of equality saturation to optimize the FPGA implementation of large integer multiplications. We use multiplication decomposition rules at various bitwidths and DSP tiling patterns at low bitwidths as our rewrite rules for equality saturation. Bringing together a wide range of arithmetic transformations and resource mapping patterns, e-graphs generated by IMpress contain extensive ways to rewrite an integer multiplication. Consequently, one major challenge is extracting the optimal expression(s) from the e-graph. Although the e-graph is constructed to be as compact as possible, it contains a tremendous number of expressions which makes optimal extraction a nontrivial task. Previous efforts have developed heuristics and exact algorithms for extracting the optimal expression for a single objective [19], [20], [21], [22], [23]. However, our problem copes with multiple objectives such as

978-1-6654-8332-2/22/$31.00

different resource types on an FPGA. A recent study developed an iterative feedback-driven expansion and contraction approach to produce Pareto-optimal expressions for multiple objectives [24]. IMpress, on the other hand, offers an exact integer linear programming (ILP) formulation which supports both constrained and multi-objective optimization, giving the users the flexibility to choose the most suitable strategy for their application requirements.

IMpress translates an extracted custom integer multiplication expression into a behavioral description, which is then synthesized into hardware using high-level synthesis (HLS). HLS allows faster hardware development through faster design cycles and hardware customization opportunities at a high abstraction [25], [26], [27]. HLS tools offer users the option to choose the hardware resource type to use for a specific operator. However, existing HLS tools do not provide the option to set an upper limit to a specific resource type, especially LUT. Through expression rewriting, IMpress allows users to balance the utilization of different resource types at the HLS level. IMpress extends its equality saturation rules beyond arithmetic decompositions to DSP tiling patterns. With both coarse-grained transformations at the arithmetic level and fine-grained transformations at the DSP block level, IMpress offers a significantly richer design space and consequently better support for a wider range of applications. Moreover, our approach can be coupled with the allocation and scheduling strategies recently proposed by Langhammer *et al.* [11] to generate an architecture that meets certain throughput goals.
Our main technical contributions include:

- IMpress is the first work to optimize integer multiplication on FPGAs by performing equality saturation and producing various equivalent expressions with different hardware costs while avoiding the phase ordering problem.
- IMpress builds a cost model to accurately estimate the hardware cost of multiplication expressions and offers constrained and multi-objective extraction tailored to optimizing resource utilization based on application requirements.
- IMpress translates an extracted multiplication expression into C++ code for HLS, followed by downstream FPGA design stages. IMpress offers significant control over resource utilization and balance, and it increases the maximum number of instances of cryptographic applications on FPGA.

II. Preliminaries

A. Large Integer Multiplication

There are several popular decomposition methods that optimize the performance of large multiplications [28]. Schoolbook decomposition uses the basic textbook method to decompose each operand into two parts and performs four partial products followed by a summation. Karatsuba decomposes operands as in the schoolbook approach, but uses a clever rewriting to reduce the number of partial products from 4 to 3 [29], [30]. Toom-Cook multiplication is a generalization of Karatsuba in which operands can be split into 2 or more parts and splitting into more parts results in asymptotically faster implementations [31]. Comba multiplication extends classical approaches with a scheduling strategy over the summation of partial products [32]. At significantly high bitwidths, such as a million bits, asymptotically faster approaches such as fast Fourier transform (FFT) are used [33].

B. Equality Saturation

Expression rewriting [34] is a search-based approach used in theorem proving [35], [36] and program optimization [37], [18]. *Rewrite rules* encode equivalences between different expressions and are potential transformations that can be applied to a given specification during expression rewriting. Traditional optimization approaches destructively modify patterns in the original specification by sequentially selecting a path from an exponential number of choices. Such methods often lead to suboptimal results due to the suboptimal graph substitution heuristics aimed at overcoming the exponential growth in enumerating the possible rewriting solutions.

Equality saturation [18], [38], [39] performs non-destructive rewriting efficiently and overcomes the limitations of traditional rewriting engines, including the phase ordering problem [19]. The basic data structure used in equality saturation is an e-graph [40], [41]. An e-graph represents a congruence relation over expressions and it builds upon many equivalence relations over e-classes and e-nodes. Each e-node, denoted by a function symbol, represents an expression. Each e-class, which can contain one or more e-nodes, represents equivalent expressions. An e-graph is basically a set of e-classes.

Equality saturation is used to optimize an input specification as follows. First, an e-graph is constructed from the original specification and iteratively updated by applying rewrite rules until saturation or a timeout. Then, the best expression is extracted from the e-graph. Current state-of-the-art open-source equality saturation tool provides an extractor [19]. It is based on a heuristic which selects the lowest-cost e-node in each e-class from the leaf e-classes to the root e-class.

III. Motivational Example

We conduct a case study on 256-bit integer multiplication to motivate equality saturation and multi-objective extraction. Specifically, we list three implementations of 256-bit integer multiplication in Table I, namely the default HLS implementation, the schoolbook decomposition, and the Karatsuba decomposition. These three implementations are Pareto-optimal because among these solutions, DSP utilization cannot be reduced without sacrificing LUTs, and vice versa.

If we consider further decomposing all 128-bit partial products using Karatsuba decomposition, our new set of implementations becomes the list in Table II. One of such implementations, i.e., Schoolbook+Karatsuba, is illustrated in Fig. 1. This implementation does not yield a Pareto-optimal solution because it is dominated by the 1-level Karatsuba implementation in terms of both DSP and LUT utilization. In other words, although schoolbook decomposition alone yields a Pareto-optimal solution in the first level of decomposition at 256 bits, when followed by Karatsuba decomposition at

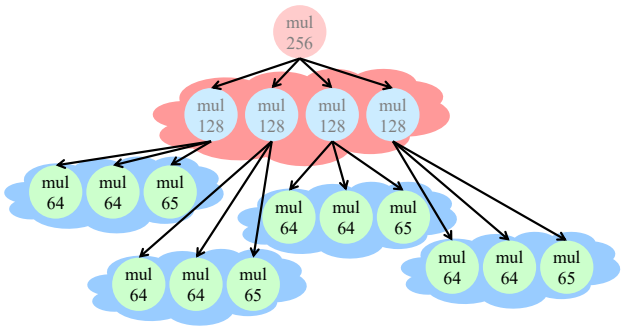

Fig. 1: Schoolbook followed by Karatsuba decomposition of 256-bit integer multiplication. Each node is a partial product implemented using DSPs and LUTs. Cloud shapes represent the summation of the partial products and are implemented using LUTs.

128 bits, it no longer yields a Pareto-optimal solution. Due to such scenarios where suboptimal solutions originate from locally optimal decisions, or vice versa, it is nontrivial to find the optimal combination of different decompositions.

This example also illustrates the trade-off between two FPGA resources, i.e., DSPs and LUTs. Such a trade-off becomes essential when a very complex design is being mapped to an FPGA device. If we extract an optimal expression based on only one resource type, we may end up exceeding the utilization limit for the other resource type. In Sections V-E2 and V-E1, we show that a good balance between these two resource types achieved through our multi-objective extraction strategy increases the maximum number of instances of a given design on an FPGA device.

IV. Approach

IMpress finds the most efficient implementation(s) of a given bit-accurate integer multiplication on an FPGA by (1) constructing an e-graph and performing equality saturation (Section IV-A), (2) extracting the optimal expression(s) from the saturated e-graph (Section IV-B), (3) transforming the optimal expression(s) to HLS C++ code and integrating them into the full application for FPGA synthesis (Section IV-C). IMpress can optimize multiplications with powers-of-two sizes which are typically used in cryptographic applications. Such applications use bitwidths ranging from 512 to a million bits depending on the algorithm and scheme.

TABLE I: Design points with up to one level of decomposition.

Implementation	DSP	LUT
HLS Default	225	226
Schoolbook	200	4,803
Karatsuba	164	5,050

TABLE II: Some design points with up to two levels of decomposition.

Implementation	DSP	LUT
HLS Default	225	226
Schoolbook	200	4,803
Karatsuba	164	5,050
Schoolbook + Karatsuba	192	6,026
Karatsuba + Karatsuba	144	6,141

A. Equality Saturation for Large Integer Multiplication

Given an input program and a set of rewrite rules, equality saturation constructs an e-graph of all equivalent expressions. IMpress takes a bit-accurate integer multiplication as input and defines a set of rewrite rules using multiplication decomposition and resource mapping patterns. As shown in Eq. (1), k rewrite rules constitute the set R. The number of rewrite rules k for each bit-accurate multiplication is listed in Table IV. A rewrite rule, $\langle l_i, r_i \rangle$, transforms the left-hand expression l_i to the right-hand expression r_i. Rewrite rules are applied in each iteration until the e-graph is saturated or a timeout is reached.

Our first set of rewrite rules (Eq. (2)) transform an i-bit integer multiplication mul_i to its schoolbook decomposition $schoolbook(mul_i)$. Schoolbook method is given in Eq. (7). In the last line, we substitute the addition of the first and last term with the zero-cost concatenation operation $\&$. A graphical representation of this rule is given in Fig. 2, where each node is a hardware operator and its children are the operands to it.

$$R = \{\langle l_i, r_i \rangle |\ i \in \{0, 1, ..., k-1\}\} \quad (1)$$

$$\langle mul_i, schoolbook(mul_i) \rangle \in R, i \in \{32, 64, ..., 512, ...\} \quad (2)$$

$$\langle mul_i, karatsuba(mul_i) \rangle \in R, i \in \{32, 64, ..., 512, ...\} \quad (3)$$

$$\langle mul_i, mul_{i-1} \rangle \in R, i \in \{17, 33, ..., 257, ...\} \quad (4)$$

$$\langle mul_{32}, tiling_i(mul_{32}) \rangle \in R, i \in \{0, 1, 2, 3, 4\} \quad (5)$$

$$\langle mul_{16}, tiling_i(mul_{16}) \rangle \in R, i \in \{0, 1\} \quad (6)$$

$$a = a_h 2^{n/2} + a_l \qquad b = b_h 2^{n/2} + b_l \quad (7)$$
$$a \cdot b = a_h b_h 2^n + (a_h b_l + a_l b_h) 2^{n/2} + a_l b_l$$
$$= ((a_h b_h) \& (a_l b_l)) + (a_h b_l + a_l b_h) \ll n/2$$

Our second set of rewrite rules (Eq. (3)) transform an i-bit integer multiplication mul_i to its Karatsuba decomposition $karatsuba(mul_i)$. Karatsuba uses Eq. (8) to reduce the number of partial products in the schoolbook decomposition from 4 to 3, at the cost of more addition/subtraction operations. A graphical representation of this rule is given in Fig. 3.

$$a_h b_l + a_l b_h = (a_h + a_l)(b_h + b_l) - a_h b_h - a_l b_l \quad (8)$$

Karatsuba decomposition of an n-bit multiplication produces an (n/2+1)-bit multiplier. In order to enable further decomposition of this odd-bit multiplier using either schoolbook or Karatsuba, we add another set of rewrite rules (Eq. (4)) which decompose an odd-bit multiplication mul_i into an even-bit multiplication mul_{i-1}. Decomposition of an odd-bit multiplication is given in Eq. (9), where each operand is first split into two parts :1 (i.e., most significant n-1 bits) and 0:0 (i.e., the least significant bit), and then the multiplication is constructed from the partial products of these parts.

$$a = 2a_{:1} + a_{0:0} \qquad b = 2b_{:1} + b_{0:0} \quad (9)$$
$$a \cdot b =$$
$$((a_{:1} b_{:1}) \ll 2 + (a_{:1} b_{0:0} + a_{0:0} b_{:1}) \ll 1)_{:1} \& (a_{0:0} \wedge b_{0:0})$$

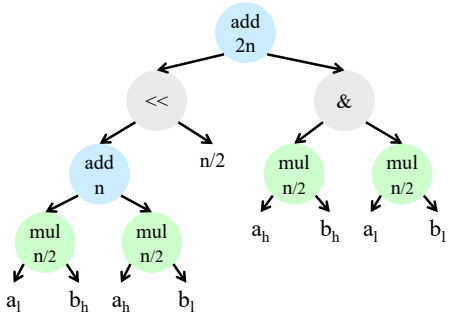

Fig. 2: Schoolbook decomposition of integer multiplication $a \times b$ as a hardware-friendly rewrite rule. Both a and b are n-bit integers. The number at the bottom of add/mul nodes is the operation bitwidth.

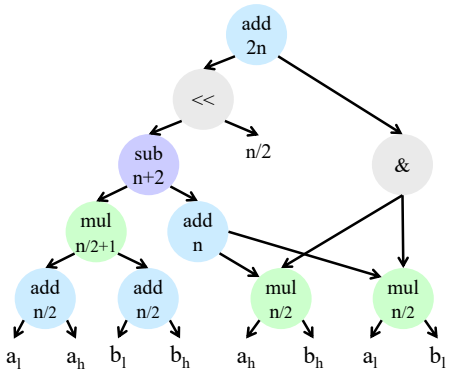

Fig. 3: Karatsuba decomposition of integer multiplication $a \times b$ as a hardware-friendly rewrite rule. Both a and b are n-bit integers. The number at the bottom of add/sub/mul nodes is the operation bitwidth.

Our last sets of rewrite rules are based on DSP block tiling. In Eq. (5), a 32-bit multiplication mul_{32} can be implemented with five tiling patterns $tiling_i(mul_{32})$ with DSP block utilization i ranging from 0 to 4. The example in Fig. 4 represents an implementation of 32-bit multiplication using 3 DSP blocks, corresponding to $tiling_3$ in Eq. (5). In Eq. (6), a 16-bit multiplication mul_{16} can be implemented with two tiling patterns $tiling_i(mul_{16})$ with either 0 or 1 DSP block. Each of our rewrite rules brings about a DSP-LUT trade-off.

Equality saturation is typically performed over syntactic rewrites. However, IMpress deals with bit-accurate operations and needs to distinguish the cost of different-bitwidth operators of the same type. Therefore, as part of our equality

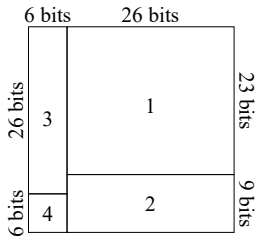

Fig. 4: Tiling of 32-bit integer multiplication. For example, assuming a 26×23 DSP block size, each of the regions 1, 2, and 3 can be mapped to a DSP block and region 4 can be implemented with LUTs.

TABLE III: Primitive operators used by IMpress in equality saturation, expression extraction, and code generation for a 1024-bit multiplication.

Operator Type	Bitwidth
add	16, 17, 32, 33, 34, 64, 65, 66, 128, 129, 130 256, 257, 258, 512, 513, 514, 1024, 1026
sub	34, 66, 130, 258, 514, 1026
mul	16, 32, 64, 128, 256, 512, 1024
and	1, 16, 32, 64, 128, 256, 512

saturation framework, we define bit-accurate operators carrying both the operation type and the bitwidth information such as add128. This allows us to rewrite a multiplication at any level and build accurate hardware cost models which in turn helps determine the optimal rewriting scheme. We identify the primitive operators used by our rewrite rules, create a library of those, and pre-characterize their hardware cost for the extraction (Section IV-B) and code generation (Section IV-C) phases. Pre-characterization includes assigning a LUT cost and a DSP cost to each bit-accurate operator and penalizing certain operators that cause timing failures by assigning large costs to them. Table III lists our bit-accurate primitive operators that are used to rewrite a 1024-bit multiplication. These are synthesized individually on the FPGA device and their costs are parsed from post-implementation reports.

The compactness of an e-graph stems from shared sub-expressions. The fact that we have different rewrite rules having common sub-expressions boosts the compactness of our e-graphs. Fig. 5 is a relatively small e-graph we can obtain for 32-bit integer multiplication. The expression rooted at the blue-shaded "&" is shared by our schoolbook and Karatsuba rewrite rules and the expressions rooted at the blue-shaded "add16" are shared between odd multiplication and its transformation into even multiplication. We also have sharing within a rewrite rule, as illustrated by the blue-shaded "≫".

B. Optimal Expression Extraction

Given an e-graph, our goal is to extract the optimal expression, or a set of expressions, which satisfy our objective. In Fig. 5, nodes of an expression, which corresponds to a Karatsuba decomposition, are highlighted. Extraction can be thought of as selecting e-nodes from e-classes such that each e-class in the final expression contains one e-node. IMpress supports both constrained single-objective extraction and multi-objective extraction. The former approach, as described in Section IV-B1, can be used to minimize LUTs while setting an upper bound on DSPs. The latter, as described in Section IV-B2, can be used to find Pareto-optimal solutions for the co-minimization of LUTs and DSPs. We assume an additive cost model where the cost of an expression is the summation of the cost of the e-nodes that remain after extraction.

1) Constrained Single-Objective Optimization: We formulate the extraction of an optimal expression using integer linear programming (ILP). Our ILP formulation assigns a decision variable xn_i to each e-node i, indicating whether the e-node is present in the final expression or not. Each e-class j is assigned

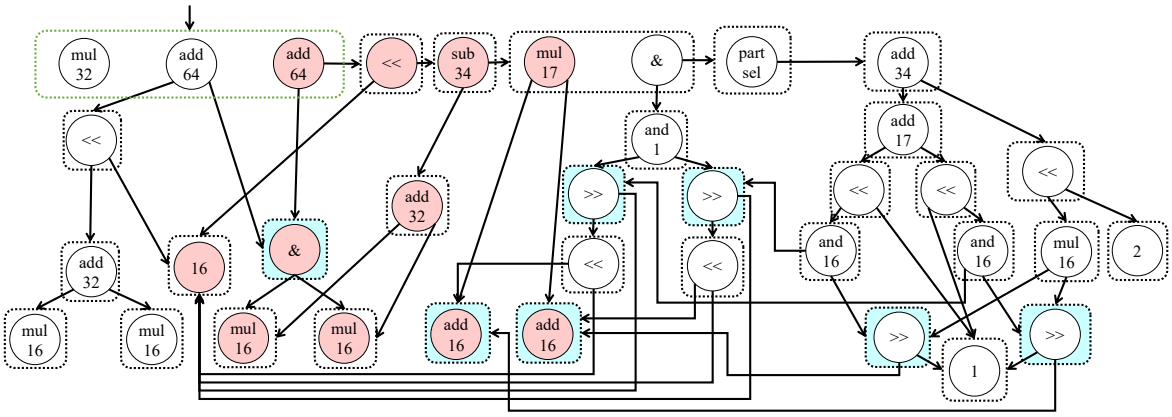

Fig. 5: E-Graph of 32-bit integer multiplication based on rewrite rules (2) and (3) for $i = 32$ and rule (4) for $i = 17$. Each node is an e-node corresponding to an operator or a constant. Each dashed rectangle is an e-class. Each e-class can contain multiple e-nodes that satisfy the equivalence relation. The root e-class, which expresses the input program, is highlighted in green. Nodes of an example expression are highlighted in rose pink. Root e-classes of the sub-expressions that are shared across different rewrite rules are highlighted in blue.

a decision variable xc_j to drive the selection of e-nodes from e-classes. These variables are used to construct the linear constraints and the linear objective in our ILP formulation.

Constraints (15) and (16) ensure that the decision variables xn_i and xc_j are binary variables. $xn_i = 1$ indicates that e-node i is present in the final expression while $xn_i = 0$ indicates otherwise. Similarly, $xc_j = 1$ indicates that e-class j is used to extract the final expression while $xc_j = 0$ indicates otherwise. Constraint (14) ensures that the final expression is drawn from the root e-class. Constraint (11) ensures that if an e-node is selected for the final expression, then all of its children e-classes must be used. If $xn_i = 1$, then xc_j for each and every $j \in children_i$ can only be 1. If $xn_i = 0$, then xc_j can be either 0 or 1. In other words, the fact that e-class j is used does not imply that e-node i is selected since e-classes can be shared across the e-graph. Constraint (12) implies that if an e-class is used, then one of its e-nodes must be in the final expression. If an e-class is not used, then none of its e-nodes exist in the final expression. Constraint (13) sets an upper bound on DSP consumption in the final expression. dsp_i is the DSP consumption of e-node i. The upper bound dsp_limit is to be specified by the user based on their application requirements. Finally, objective (10) minimizes LUT consumption in the final expression. lut_i is the LUT consumption of e-node i.

$$\text{minimize} \sum_{i \in enodes} lut_i \cdot xn_i \tag{10}$$

$$xn_i - xc_j \leq 0, \forall i \in enodes, \forall j \in children_i \tag{11}$$

$$xc_i - \sum_{j \in enodes_i} xn_j = 0, \forall i \in eclasses \tag{12}$$

$$\sum_{i \in enodes} dsp_i \cdot xn_i \leq dsp\_limit \tag{13}$$

$$xc_{root\_eclass} = 1 \tag{14}$$

$$xn_i \in \{0, 1\}, \forall i \in enodes \tag{15}$$

$$xc_j \in \{0, 1\}, \forall j \in eclasses \tag{16}$$

2) Multi-Objective Optimization: In this step, we extract Pareto-optimal expressions from a saturated e-graph by applying the ε-constraint method [42] to our previous ILP formulation. The idea behind the ε-constraint method is to transform all objective functions except one to additional constraints as $f_i <= \varepsilon$ and so leave a single objective. A single run of this method produces a single weakly Pareto-optimal solution. In order to obtain the Pareto frontier, one can re-run the ILP solver with all possible dsp_limit values. To find the lower bound of dsp_limit, we can solve our ILP formulation after omitting constraint (13) and replacing objective (10) with objective (17). To find the upper bound of dsp_limit, objective (10) is replaced with objective (18).

$$\text{minimize} \sum_{i \in enodes} dsp_i \cdot xn_i \tag{17}$$

$$\text{maximize} \sum_{i \in enodes} dsp_i \cdot xn_i \tag{18}$$

C. Code Generation and Synthesizing the Application

We translate rewritten integer multiplication expressions to C++ code for HLS. We do so by searching for rewrite patterns from lower to higher bitwidths in the expression graph, and replacing them with the corresponding fused multiplication operation. As an example, we first search for Fig. 2 for n=32 and replace it with a fused node *mul32* with label *schoolbook*. Once all rewrite patterns corresponding to 32-bit integer multiplication are searched for, we proceed with n=64, n=128, ..., until the input bitwidth. We use an implementation of the VF2 algorithm [43] provided by Networkx [44] to perform subgraph isomorphism that matches rewrite patterns with subgraphs in the expression graph. When all patterns are searched for, we instantiate C++ functions that perform the matched patterns. The last step is to insert the C++ code into the full application, followed by FPGA design stages HLS, logic synthesis, placement, and routing.

V. Evaluation

Section V-A shows the scalability of our equality saturation framework with increasing multiplication bitwidths. Section V-B evaluates the performance of IMpress for a single objective. Section V-C evaluates the performance of IMpress for multiple objectives. Section V-D measures the cost model accuracy of IMpress, which is used to extract optimal integer multiplication expressions. Section V-E evaluates the effectiveness of IMpress in two cryptographic applications. Our framework is applicable to any FPGA device. In our evaluations, we target Alveo U250 part xcu250-figd2104-2L-e at 300 MHz due to their increasing adoption in cryptography. For FPGA synthesis, we use AMD Xilinx Vitis HLS and Vivado v2020.2. All integer multiplications are pipelined in HLS with an initiation interval of 1.

We use an open-source framework called egg to perform equality saturation [19]. egg allows users to provide their own set of syntactic rewrite rules and provides a single-objective extractor as described in Section II-B. We perform two types of rewriting verification. First, we perform translation validation on our extracted expression. This ensures that the input integer multiplication and the extracted expression are equivalent for the given set of rewrite rules. Translation validation is provided by egg. Second, we perform C-simulation and C/RTL co-simulation after integrating the HLS implementation of our custom integer multiplier into the full application.

A. Scalability Analysis

We evaluate the scalability of the equality saturation framework in Table IV in terms of the time it takes to saturate the e-graph, the number of e-node, the number of e-classes, and the number of expressions that can be extracted from the e-graph. With a few rewrite rules, a tremendous number of expressions can be represented in a graph as small as hundreds of nodes. As we increase the multiplication bitwidth and thereby the number of rewrite rules, the growth in the e-graph size is significantly slower than the growth in the number of expressions. Although we evaluate IMpress for multiplications up to 2048 bits, the framework can easily be extended to higher bitwidths and supported by other decomposition techniques. For instance, NTT can be coupled with IMpress to optimize million-bit multiplications as shown in Section V-E1.

B. ILP vs Existing Heuristic for a Single Objective

We evaluate the effectiveness of our ILP formulation versus egg's bottom-up heuristic in optimizing a single objective. The objective is defined to be the weighted sum of LUTs and DSPs. The weights are determined based on the available resources on the given FPGA device.

$$objective = w \times \#DSPs + \#LUTs \tag{19}$$
$$w = \frac{\#Available\ LUTs}{\#Available\ DSPs}$$

For different integer multiplication bitwidths, Table V lists the post-placement costs of the expressions found by egg's extractor versus expressions found by IMpress using ILP formulation with CPLEX. For the same set of inputs, Table V also lists the time it takes to perform the extraction. At very low bitwidths, i.e., 64 bits, egg's bottom-up heuristic and IMpress have the same performance in terms of both final cost and runtime. As the bitwidth increases, IMpress produces higher-quality expressions within a comparable time frame. Our ILP formulation finds lower-cost implementations because egg's heuristic is based on the assumption that a minimum-cost expression originates from the minimum-cost subexpressions, which is not a valid assumption in the existence of common subexpressions in the e-graph [21]. ILP formulation will take considerably longer to optimize for bitwidths higher than 1024 due to the exponential increase in the number of variables and constraints. However, as shown in Section V-E1, the efficiency of the FFT/NTT algorithms at extremely large bitwidths can be coupled with the flexibility of IMpress at 2048 or lower bits to create design points with optimized resource and performance.

C. Expression Rewriting with Multiple Objectives

Table VI lists costs of different implementations of 1024-bit multiplication, including the default HLS implementation, different levels of Karatsuba decomposition, and IMpress. IMpress-1 is obtained using the single objective strategy from the previous section. IMpress-2 is obtained through DSP-constrained LUT minimization such that the percentage utilization of both LUTs and DSPs is balanced at around 7%.

TABLE IV: Scalability of our equality saturation framework for different input bitwidths and sets of rewrite rules.

Input Size	Rewrite Rules	Sat. Time (s)	E-Nodes	E-Classes	Expr.s
64	10	0.01	252	194	1.08e6
128	13	0.06	1,171	878	1.37e24
256	16	0.29	5,546	4,078	3.55e96
512	19	1.51	26,769	19,426	1.58e386
1,024	22	8.60	130,936	94,218	6.31e1544
2,048	25	49.19	645,935	462,342	1.59e6179

TABLE V: Comparison of single-objective extraction techniques.

Bitwidth	Post-Placement Cost		Runtime (s)	
	egg	IMpress	egg	IMpress
64	1,823	1,823	0.01	0.01
128	6,698	6,466	0.02	0.03
256	25,359	22,399	0.11	0.09
512	85,117	78,434	0.63	0.50
1,024	278,265	264,610	2.72	4.07
2,048	1,101,986	883,931	18.26	45.33

TABLE VI: Resource utilization, latency, and frequency of 1024-bit multiplication when synthesized using the default HLS implementation, Karatsuba decompositions, and IMpress.

Method	DSP	LUT	Latency (cycles)	Freq (MHz)
HLS Default	NA	NA	NA	NA
3-level Karat.	1,476	71,893	17	256
4-level Karat.	1,296	88,986	21	305
5-level Karat.	972	126,193	21	302
6-level Karat.	729	165,618	24	298
IMpress-1	810	150,400	23	250
IMpress-2	945	129,338	22	295

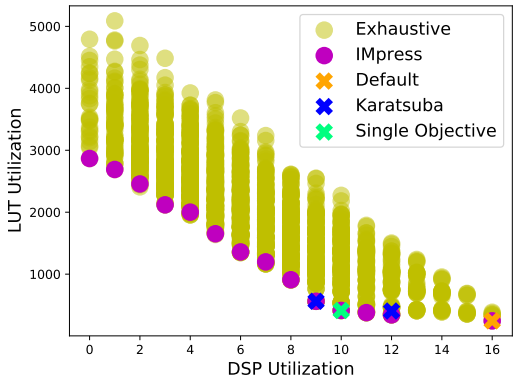

Fig. 6: Post-placement LUT and DSP utilization of 64-bit integer multiplication expressions extracted by different approaches.

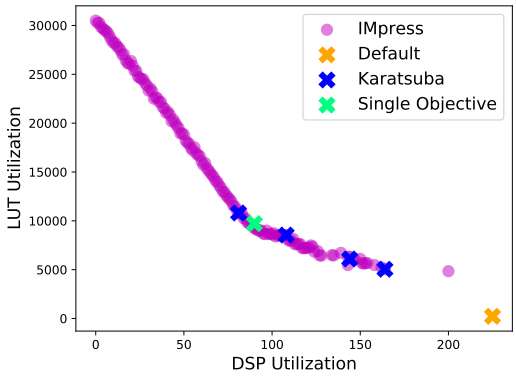

Fig. 7: Post-placement LUT and DSP utilization of 256-bit integer multiplication expressions extracted by different approaches.

We measure Pareto-efficiency of IMpress in optimizing multiple objectives. For this analysis we target 64-bit integer multiplication, because as previously shown in Table IV, it becomes intractable to exhaustively collect and synthesize all expressions at higher bitwidths. As identified based on the exhaustively collected results shown in Fig. 6, there are 14 expressions constituting the Pareto frontier. IMpress generates 12 of these expressions. The 2 non-Pareto-optimal expressions found by IMpress (at DSP=4 and DSP=7) have the same DSP utilization and only 2% higher LUT utilization compared to the corresponding Pareto-optimal expressions.

Fig. 7 shows the post-implementation cost of all 256-bit multiplication expressions found by our multi-objective extractor. The expression found by IMpress using the single objective strategy from the previous section is marked in green. The figure also includes the expressions corresponding to the default HLS implementation and different levels of Karatsuba. This study illustrates the flexibility IMpress offers in controlling and balancing the utilization of different resource types.

D. Cost Model Accuracy

Due to optimizations performed during logic synthesis and placement, our cost model is not expected to be fully accurate compared to post-implementation results. We measure the accuracy of our DSP and LUT estimations based on R2, Pearson, and Spearman accuracy metrics. R2, the coefficient of determination, represents how well the variation in the output can be explained based on the inputs to the model. The Pearson coefficient represents the linear correlation between estimates and actual values. Spearman's rank correlation coefficient represents the monotonic relationship between estimates and actual values. Coefficients that are close to 1 represent high accuracy. Accuracy score of DSP estimation is 1 for all metrics, regardless of the bitwidth. This is essential for constrained optimization where constraints are based on DSP utilization. R2 score in LUT estimation, which is 0.94 at 64 bits, gradually decreases as bitwidth increases. For instance, it becomes 0.92 at 128 bits. Pearson and Spearman correlation scores are 1 for both LUT and DSP estimation regardless of the bitwidth. Since we perform LUT minimization in both constrained and multi-objective settings, our high LUT correlation scores translate to successful optimization in IMpress at any bitwidth.

E. Cryptographic Applications

We evaluate the effectiveness of IMpress in two cryptographic applications, namely integer-based HE and RSA. We show that IMpress helps resolve placement issues, and compared to other decomposition approaches it allows more instances of these applications to fit on a given FPGA device.

1) Integer-Based HE: Fully homomorphic encryption (FHE) allows computation of arbitrary functions on encrypted data. A solution to this problem has been first proposed by Gentry [45], [46]. Some FHE schemes are based on ring learning with errors [47], [48] which require polynomial multiplications over rings, and some FHE schemes are based on very large (million-bit) integer arithmetic [49], [50], [51].

The Schönhage–Strassen algorithm is a fast multiplication algorithm for very large integer operands. It performs better than the classical methods for operands with more than 2^{17} bits [52]. We implement an NTT-based Schönhage–Strassen multiplication algorithm for million-bit integers using NTT/INTT implementations proposed by Navas *et al.* [53]. Due to resource limitations, most of the FPGA-based implementations of NTT use prime numbers less than or equal to 64 bits since the prime number determines the multiplication size [17], [54]. In order to reduce the number of NTT iterations and improve the overall latency, we select a large prime in the NTT computation. Our resulting million-bit multiplication implementation includes 512 and 1024-bit multipliers.

We explore million-bit multiplication implementations in Table VII. The default HLS implementation fails placement due to resource overutilization. K(n,m) represents Karatsuba decomposition applied to 512-bit multiplication producing n-bit leaf multiplications and 1024-bit multiplication producing m-bit leaf multiplications. The maximum number of instances of the million-bit design that can fit U250 is maximum 3 using Karatsuba decompositions. IMpress succeeds in fitting 4 instances of the design by constraining DSP utilization of 512-bit multiplication to 307, 1024-bit multiplication to 1228, and minimizing their LUT utilization. U250 consists of 4 SLR regions. In an FHE application, let us assume that we are

TABLE VII: Resource utilization, latency, and frequency of million-bit HE benchmark when synthesized using the default HLS implementation, Karatsuba decompositions, and IMpress.

Method	DSP (DSP%)	LUT (LUT%)	Latency (cycles)	Freq (MHz)
HLS Default	NA	NA	NA	NA
K(64,128)	4,681 (38.09)	274,224 (15.87)	12,452,003	221
K(64,64)	4,274 (34.78)	304,140 (17.60)	13,222,078	219
K(32,64)	3,950 (32.15)	339,999 (19.68)	13,123,773	212
K(32,32)	3,227 (26.26)	408,301 (23.63)	12,927,158	239
K(16,32)	2,984 (24.28)	439,775 (25.45)	13,222,078	239
K(16,16)	2,431 (19.78)	511,916 (29.62)	13,811,922	229
IMpress	3,071 (24.99)	381,697 (22.09)	12,746,927	225

TABLE VIII: Resource utilization, latency, and frequency of RSA-512 when synthesized using the Vitis library implementations, Karatsuba decompositions, and IMpress.

Method	DSP (DSP%)	LUT (LUT%)	Latency (cycles)	Freq (MHz)
Vitis-1	0 (0)	17,055 (0.99)	39,565	238
Vitis-2	4,530 (36.87)	41,882 (2.42)	3,638	18
1-level Karat	3,424 (27.86)	131,086 (7.59)	4,646	229
2-level Karat	3,072 (25.00)	156,339 (9.05)	5,192	223
3-level Karat	2,352 (19.14)	223,964 (12.96)	4,772	241
4-level Karat	1,782 (14.50)	291,507 (16.87)	5,150	236
IMpress	2,030 (16.52)	247,706 (14.33)	5,024	231

constrained to fit one million-bit multiplication design in one SLR. As Table VII suggests, IMpress can fit the logic into one SLR, while none of the Karatsuba decompositions can achieve this. The latency and frequency results of IMpress are comparable to Karatsuba results.

2) RSA: Modular exponentiation is computationally complex and heavily used in public-key cryptography, e.g., RSA and Diffie-Hellman key exchange. Its efficiency depends heavily on the modular multiplication unit. Montgomery modular multiplication is a fast method and it mainly contains three integer multiplications. Modulus size determines the complexity of the multiplication. We use the HLS implementation of RSA-512 provided by the AMD Xilinx Vitis HLS Security Library [55]. Their implementation is indicated as Vitis-1 in Table VIII. We obtain Vitis-2 by modifying their Montgomery unit to leverage 256-bit integer multiplications for improved latency.

Vitis-1 gives a considerably high latency compared to other implementations. Vitis-2 gives a very low frequency (18 MHz) due to its 256-bit multiplication unit. The maximum number of instances of RSA-512 that can fit U250 is maximum 5 using Karatsuba decompositions. IMpress succeeds in fitting 6 instances of RSA-512 by constraining DSP utilization of 256-bit multiplication to 93 and minimizing LUTs. In the classical approaches, DSPs are the limiting factors at the early decomposition levels and LUTs become the limiting factor as the level is increased. IMpress, on the other hand, effectively balances the utilization of both resources, leading to a more efficient use of the FPGA device. The latency and frequency results of IMpress are comparable to Karatsuba results.

VI. Related Work

Optimization of large integer multiplication is an important problem in FPGA-based acceleration of cryptographic applications. Rafferty *et al.* lay out a comprehensive study of classical methods in [12]. Langhammer *et al.* propose an efficient multiplication architecture leveraging Karatsuba decomposition along with allocation and scheduling strategies for improved throughput [11]. Chow *et al.* leverage Karatsuba in their large modular multiplier [1]. Vitali *et al.* use Karatsuba and Comba methods in their HLS-based multiplier generator for better throughput [13]. Kumm *et al.* propose Karatsuba tiling for rectangular multipliers [14]. Another line of research on multiplication decomposition studies the relation between the specification of the multiplication operation and its physical implementation on FPGA devices using hardened DSP blocks or soft logic [56], [57], [58], [59], [60], [61].

Equality saturation is a very promising approach for avoiding the phase ordering problem [18]. An efficient open-source equality saturation tool called egg is recently developed [19]. We are seeing an increasing use of it across various domains including linear algebra [21], tensor computations [23], [62], digital signal processing [22], floating-point arithmetic [63], 3D CAD [20], and fabrication [24]. Some of them use ILP to extract expressions for a single objective [21], [23]. Heuristics for multi-objective extraction are developed using strategies such as iterative pruning and genetic algorithms [24], [63]. There are other rewriting systems that bring together architectural and algorithmic optimizations for digital signal processing [64] or focus on graph substitutions in DNNs [65]. There are also tools for rewriting floating-point arithmetic expressions for FPGA HLS using formal approaches and targeting the area-accuracy trade-off [66].

VII. Conclusion

Ever increasing security requirements and complexity of cryptographic algorithms necessitate synthesizing large integer multiplication units on limited FPGA resources. We propose IMpress, an integer multiplication expression rewriting framework, which produces multiplication expressions with optimal hardware cost. Through equality saturation and multi-objective extraction, IMpress generates expressions that balance the utilization of different resource types and helps resolve placement issues. Furthermore, while other decomposition techniques such as Karatsuba may not efficiently avoid timing-degrading operators due to the LUT overhead of performing another level of decomposition, IMpress can handle timing in a resource-efficient manner.

Acknowledgment

This work was supported in part by NSF Awards #1723715, #2019306, #2008144, #2019336, and research gifts from AMD Xilinx and Intel. We thank Professor Pavel Panchekha, Professor Zachary Tatlock, Max Willsey, Alexa VanHattum, and Oliver Flatt for the insightful discussions, Max Willsey also for his assistance with the egg tool, and anonymous reviewers for their valuable feedback on earlier versions of this manuscript.

REFERENCES

[1] G. C. Chow, K. Eguro, W. Luk, and P. Leong, "A Karatsuba-Based Montgomery Multiplier," *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2010.

[2] D. D. Chen, G. X. Yao, R. C. Cheung, D. Pao, and C. K. Koç, "Parameter Space for the Architecture of FFT-Based Montgomery Modular Multiplication," *IEEE Trans. on Computers (TC)*, 2015.

[3] B. H. K. Chen, P. Y. S. Cheung, P. Y. K. Cheung, and Y.-K. Kwok, "An Efficient Architecture for Zero Overhead Data En-/Decryption using Reconfigurable Cryptographic Engine," *Int'l Conf. on Field Programmable Technology (FPT)*, 2015.

[4] M. A. Mehrabi, C. Doche, and A. Jolfaei, "Elliptic Curve Cryptography Point Multiplication Core for Hardware Security Module," *IEEE Trans. on Computers (TC)*, 2020.

[5] M. M. Islam, M. S. Hossain, M. Shahjalal, M. K. Hasan, and Y. M. Jang, "Area-Time Efficient Hardware Implementation of Modular Multiplication for Elliptic Curve Cryptography," *IEEE Access*, 2020.

[6] S. Gener, P. Newton, D. Tan, S. Richelson, G. Lemieux, and P. Brisk, "An FPGA-based Programmable Vector Engine for Fast Fully Homomorphic Encryption over the Torus," *Secure and Private Systems for Machine Learning (SPSL)*, 2021.

[7] T. Ye, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Performance Modeling and FPGA Acceleration of Homomorphic Encrypted Convolution," *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2021.

[8] V. Migliore, C. Seguin, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, G. Gogniat, and R. Tessier, "A High-Speed Accelerator for Homomorphic Encryption Using the Karatsuba Algorithm," *ACM Trans. on Embedded Computing Systems (TECS)*, 2017.

[9] A. C. Mert, E. Öztürk, and E. Savaş, "Design and Implementation of Encryption/Decryption Architectures for BFV Homomorphic Encryption Scheme," *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 2020.

[10] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar, "Hardware Architecture of a Number Theoretic Transform for a Bootstrappable RNS-based Homomorphic Encryption Scheme," *Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2020.

[11] M. Langhammer and B. Pasca, "Folded Integer Multiplication for FPGAs," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[12] C. Rafferty, M. O'Neill, and N. Hanley, "Evaluation of Large Integer Multiplication Methods on Hardware," *IEEE Trans. on Computers (TC)*, 2017.

[13] E. Vitali, D. Gadioli, F. Ferrandi, and G. Palermo, "Parametric Throughput Oriented Large Integer Multipliers for High Level Synthesis," *Design, Automation, and Test in Europe (DATE)*, 2021.

[14] M. Kumm, O. Gustafsson, F. de Dinechin, J. Kappauf, and P. Zipf, "Karatsuba with Rectangular Multipliers for FPGAs," *Symp. on Computer Arithmetic (ARITH)*, 2018.

[15] I. San and N. At, "On Increasing the Computational Efficiency of Long Integer Multiplication on FPGA," *Int'l Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2012.

[16] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Optimised Multiplication Architectures for Accelerating Fully Homomorphic Encryption," *IEEE Trans. on Computers (TC)*, 2016.

[17] X. Feng and S. Li, "Design of an Area-Efficient Million-Bit Integer Multiplier Using Double Modulus NTT," *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 2017.

[18] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality Saturation: A New Approach to Optimization," *Symp. on Principles of Programming Languages (POPL)*, 2009.

[19] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and Extensible Equality Saturation," *Symp. on Principles of Programming Languages (POPL)*, 2021.

[20] C. Nandi, M. Willsey, A. Anderson, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock, "Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations," *Conf. on Programming Language Design and Implementation (PLDI)*, 2020.

[21] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu, "SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra," *Proc. VLDB Endow.*, 2020.

[22] A. VanHattum, R. Nigam, V. T. Lee, J. Bornholt, and A. Sampson, "Vectorization for Digital Signal Processors via Equality Saturation," *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[23] Y. Yang, P. Phothilimthana, Y. Wang, M. Willsey, S. Roy, and J. Pienaar, "Equality Saturation for Tensor Graph Superoptimization," *Proceedings of Machine Learning and Systems*, 2021.

[24] H. Zhao, M. Willsey, A. Zhu, C. Nandi, Z. Tatlock, J. Solomon, and A. Schulz, "Co-Optimization of Design and Fabrication Plans for Carpentry," *arXiv preprint arXiv:2107.12265*, 2021.

[25] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2011.

[26] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2016.

[27] Y.-H. Lai, E. Ustun, S. Xiang, Z. Fang, H. Rong, and Z. Zhang, "Programming and Synthesis for Software-Defined FPGA Acceleration: Status and Future Prospects," *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 2021.

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

[29] A. A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Soviet Physics Doklady*, 1963.

[30] M. Scott, "Missing a Trick: Karatsuba Variations," *Cryptography and Communications*, 2018.

[31] S. A. Cook and S. O. Aanderaa, "On the Minimum Computation Time of Functions," *Transactions of the American Mathematical Society*, 1969.

[32] P. G. Comba, "Exponentiation Cryptosystems on the IBM PC," *IBM Systems Journal*, 1990.

[33] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, 1965.

[34] N. Dershowitz, "A Taste of Rewrite Systems," *Functional Programming, Concurrency, Simulation and Automated Reasoning*, 1993.

[35] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A Theorem Prover for Program Checking," *Journal of the ACM (JACM)*, 2005.

[36] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[37] R. Joshi, G. Nelson, and K. Randall, "Denali: A Goal-Directed Superoptimizer," *ACM SIGPLAN Notices*, 2002.

[38] M. Stepp, R. Tate, and S. Lerner, "Equality-Based Translation Validator for LLVM," *Int'l Conf. on Computer Aided Verification (CAV)*, 2011.

[39] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically Improving Accuracy for Floating Point Expressions," *ACM SIGPLAN Notices*, 2015.

[40] C. G. Nelson, "Techniques for Program Verification," *Stanford University*, 1980.

[41] R. Nieuwenhuis and A. Oliveras, "Proof-Producing Congruence Closure," *Int'l Conf. on Rewriting Techniques and Applications (RTA)*, 2005.

[42] Y. Y. Haimes, L. S. Lasdon, and D. A. Wismer, "On a Bicriterion Formulation of the Problems of Integrated System Identification and System Optimization," *IEEE Trans. on Systems, Man, and Cybernetics (TSMC)*, 1971.

[43] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 2004.

[44] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," *Python in Science Conference (SciPy)*, 2008.

[45] C. Gentry, "A Fully Homomorphic Encryption Scheme," *Stanford University*, 2009.

[46] ——, "Computing Arbitrary Functions of Encrypted Data," *Communications of the ACM*, 2010.

[47] V. Lyubashevsky, C. Peikert, and O. Regev, "On Ideal Lattices and Learning with Errors over Rings," *Journal of the ACM (JACM)*, 2013.

[48] Z. Brakerski and V. Vaikuntanathan, "Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages," *Cryptology Conf. (CRYPTO)*, 2011.

[49] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully Homomorphic Encryption over the Integers," *Int'l Conf. on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2010.

[50] C. Gentry and S. Halevi, "Implementing Gentry's Fully-Homomorphic Encryption Scheme," *Int'l Conf. on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2011.
[51] H. V. L. Pereira, "Bootstrapping Fully Homomorphic Encryption over the Integers in Less than One Second." *Int'l Conf. on Public-Key Cryptography*, 2020.
[52] L. C. C. Garcıa, "Can Schönhage Multiplication Speed Up the RSA Decryption or Encryption?" *Technische Universität Darmstadt*, 2007.
[53] J. A. Navas, B. Dutertre, and I. A. Mason, "Verification of an Optimized NTT Algorithm," *Working Conf. on Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2020.
[54] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating Fully Homomorphic Encryption in Hardware," *IEEE Trans. on Computers (TC)*, 2014.
[55] Xilinx, "Vitis Security Library," *https://xilinx.github.io/Vitis_Libraries/security/2020.2/index.html*, 2020.
[56] S. Srinath and K. Compton, "Automatic Generation of High-Performance Multipliers for FPGAs with Asymmetric Multiplier Blocks," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2010.
[57] H. Parandeh-Afshar and P. Ienne, "Measuring and Reducing the Performance Gap between Embedded and Soft Multipliers on FPGAs," *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2011.
[58] D. B. Roy, D. Mukhopadhyay, M. Izumi, and J. Takahashi, "Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves," *Design Automation Conf. (DAC)*, 2014.
[59] B. Ronak and S. A. Fahmy, "Efficient Mapping of Mathematical Expressions into DSP Blocks," *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2014.
[60] M. Kumm, J. Kappauf, M. Istoan, and P. Zipf, "Resource Optimal Design of Large Multipliers for FPGAs," *Symp. on Computer Arithmetic (ARITH)*, 2017.
[61] E. Ustun, C. Deng, D. Pal, Z. Li, and Z. Zhang, "Accurate Operation Delay Prediction for FPGA HLS Using Graph Neural Networks," *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2020.
[62] G. H. Smith, A. Liu, S. Lyubomirsky, S. Davidson, J. McMahan, M. Taylor, L. Ceze, and Z. Tatlock, "Pure Tensor Program Rewriting via Access Patterns (Representation Pearl)," *Int.'l Symp. on Machine Programming*, 2021.
[63] B. Saiki, O. Flatt, C. Nandi, P. Panchekha, and Z. Tatlock, "Combining Precision Tuning and Rewriting," *Symp. on Computer Arithmetic (ARITH)*, 2021.
[64] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, "SPIRAL: Extreme Performance Portability," *Proceedings of the IEEE*, 2018.
[65] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions," *Symp. on Operating Systems Principles (SOSP)*, 2019.
[66] X. Gao, S. Bayliss, and G. A. Constantinides, "SOAP: Structural Optimization of Arithmetic Expressions for High-Level Synthesis," *Int'l Conf. on Field Programmable Technology (FPT)*, 2013.