LUXOR: An FPGA Logic Cell Architecture for Efficient Compressor Tree Implementations

SeyedRamin Rasoulinezhad[1], Siddhartha[1], Hao Zhou[2], Lingli Wang[2], David Boland[1], Philip H.W. Leong[1]
[1]School of Electrical and Information Engineering, The University of Sydney, Sydney, 2006, Australia
[2]State Key Lab of ASIC and System, Fudan University, Shanghai 201203, China
{seyedramin.rasoulinezhad, siddhartha.siddhartha, david.boland, philip.leong}@sydney.edu.au
{zhouhao, llwang}@fudan.edu.cn

ABSTRACT

We propose two tiers of modifications to FPGA logic cell architecture to deliver a variety of performance and utilization benefits with only minor area overheads. In the first tier, we augment existing commercial logic cell datapaths with a 6-input XOR gate in order to improve the expressiveness of each element, while maintaining backward compatibility. This new architecture is vendor-agnostic, and we refer to it as LUXOR. We also consider a secondary tier of vendor-specific modifications to both Xilinx and Intel FPGAs, which we refer to as X-LUXOR+ and I-LUXOR+ respectively. We demonstrate that compressor tree synthesis using generalized parallel counters (GPCs) is further improved with the proposed modifications. Using both the Intel adaptive logic module and the Xilinx slice at the 65nm technology node for a comparative study, it is shown that the silicon area overhead is less than 0.5% for LUXOR and 5–6% for LUXOR+, while the delay increments are 1–6% and 3–9% respectively. We demonstrate that LUXOR can deliver an average reduction of 13–19% in logic utilization on micro-benchmarks from a variety of domains.BNN benchmarks benefit the most with an average reduction of 37–47% in logic utilization, which is due to the highly-efficient mapping of the XnorPopcount operation on our proposed LUXOR+ logic cells.

ACM Reference Format:
SeyedRamin Rasoulinezhad[1], Siddhartha[1], Hao Zhou[2], Lingli Wang[2], David Boland[1], Philip H.W. Leong[1]. 2020. LUXOR: An FPGA Logic Cell Architecture for Efficient Compressor Tree Implementations. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373087.3375303

1 INTRODUCTION

The design of parallel computer arithmetic circuits is a well established field of research dating back to the works of Wallace [30], Dadda [4], Swartzlander [26], Verma [28], and others. In the context of field-programmable gate arrays (FPGAs), there has always been interest in specialized arithmetic primitives which improve performance over a wide range of application domains. One such primitive, Generalized Parallel Counters (GPCs), enables fast accumulation of compressor trees. Work from Parandeh-Afshar et. al. [21] motivated the use of GPCs on FPGAs, while Kumm et. al. [12] demonstrated software techniques that automate the design of optimal compressor tree implementations for FPGAs. However, modern FPGA lookup table (LUT) based architectures are not particularly efficient for implementation of compressor trees [19].

In this paper, we show that support for compressor trees in FPGAs could be significantly improved through minor modifications to the logic element (LE). This is beneficial for implementing low-precision and multi-operand operations. One example of interest is that compressor trees and GPCs can be used to accelerate the XnorPopcount operations within binarized neural networks (BNNs) [1], which forms the critical path of the model's execution. BNNs enable neural networks to be utilized in resource constrained applications and can be deployed efficiently on FPGAs [13, 34]; our optimizations would improve their performance further.

LUXOR is a portmanteau of the acronyms LUT and XOR. Its design is motivated by the observation that the Boolean XOR operation is very commonly found in optimized compressor trees. This is corroborated by Verma et. al. in [29], where they exploited the correlations between the operands of the XOR function to improve delay for ASIC implementations. Our goal is to utilize this insight in a similar vein, but optimized for FPGAs.

Our proposed changes provide a means to efficiently implement compressor trees using new area-optimized GPCs, which can all be applied to a large variety and/or important classes of applications. The contributions of this paper can be summarized as follows:

- A new logic element, LUXOR, that integrates a 6-input XOR gate with commercial FPGA logic elements. This architecture independent modification improves the implementation of XnorPopcount operation and the most commonly used GPC.
- LUXOR+, an amalgamation of LUXOR with further Intel (I-LUXOR+) and Xilinx (X-LUXOR+) architecture-specific optimizations to achieve further resource reduction. To the best of our knowledge, this leads to the most efficient reported logic element based GPC, called C06060606, which can be mapped to just a single Xilinx slice.
- A novel integer linear programming (ILP) formulation based on the flexible Ternary Adder approach proposed in [22] to optimally map compressor tree problems to LUXOR cells.
- Quantitative investigation of the benefits of LUXOR and LUXOR+ architectures using a set of more than 50 micro-benchmarks. Our results also show the positive benefits of

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02…$15.00
https://doi.org/10.1145/3373087.3375303

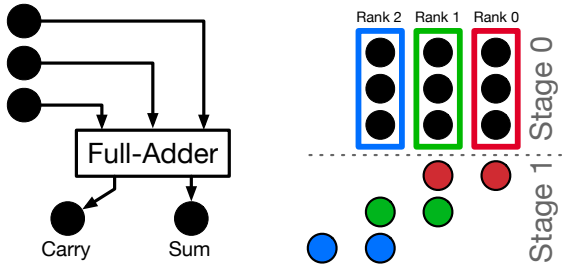
the proposed LUXOR and LUXOR+ enhancements in SMIC 65nm standard cell technology.
- The ILP-based compressor tree synthesizer, benchmarks and design files required to generate the results in this paper are open source to support reproducible research, and available at `github.com/raminrasoulinezhad/LUXOR_FPGA20`.

The remainder of the paper is organized as follows. In Section 2, we provide background on parallel counters, GPCs, compressors, and compressor trees. Our LUXOR and LUXOR+ enhancements are presented in Section 3, and the accompanying ILP formulation in Section 4. The experiment results are given in Section 5. Finally, we present conclusions in Section 6.

2 BACKGROUND

2.1 Parallel Counters

Parallel counters are digital circuits that simply count the number of asserted bits in the input, returning this value as a binary output. They can be specified in (p:q) notation, where p is the number of input bits, and q is the number of output bits used to express the result in binary notation. Half-adders (HA) and full-adders (FA) are commonly used parallel counters, denoted as (2:2) and (3:2) respectively. Parallel counters can also be expressed in dot notation [6] as shown for the full-adder in Figure 1a. We use this notation frequently in this paper to visualize various designs, and use the terms bits and dots interchangeably. Figure 1b shows how FAs can be used in parallel to implement a single stage of carry-save addition for a 3-bit (3b) 3-operand addition. Note that each FA takes inputs from a single-column, and hence, all input bits to a parallel counter have the same *rank*, *i.e.* they all have the same weight.



(a) FA takes in three bits (dots) and produces two outputs: sum and carry

(b) One stage of 3b carry-save addition of three operands using three FAs in parallel

Figure 1: (3:2) parallel counter, also known as a full-adder.

2.2 Generalized Parallel Counters

Generalized Parallel Counters, or GPCs, were first proposed by Meo [14] and subsequently shown by Parandeh-Afshar et. al. [17] to map efficiently to FPGAs. Unlike parallel counters, GPCs allow input bits to have different weights, which, in the dot notation, make the GPCs appear as multi-column counters. Figure 2 shows the dot notation of some previously published GPCs [22]. Mathematically, GPCs are written as a tuple: $(p_{n-1}, ..., p_1, p_0{:}q_{m-1}, ..., q_1, q_0)$, where p_i is the number of input bits in the i^{th} column, and q_j is the number of output bits in the j^{th} column. FPGA implementations can be classified as lookup table-based GPCs [10], or carry-chain-based GPCs [18]. As their names suggest, the "shape" of a GPC can have a profound impact on its hardware implementation on FPGAs, and subsequently its performance and efficiency in a compressor tree. Popular metrics to quantify the efficiency of a GPC include [20, 22]:

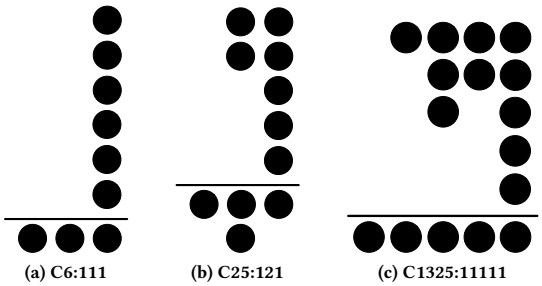


Figure 2: Three popular GPCs found in the literature

$$\text{GPC efficiency, E} = \frac{p-q}{k} \quad (1)$$

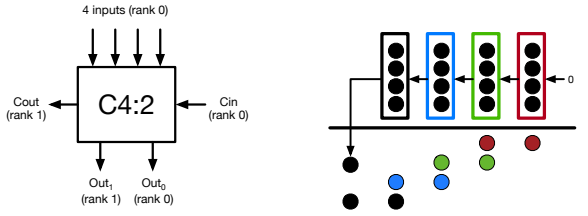
$$\text{Strength, S} = \frac{p}{q} \quad (2)$$

$$\text{Area-Performance Degree, APD} = \frac{(p-q)^2}{k*d} \quad (3)$$

$$\text{Arithmetic slack, A} = 1 - \frac{1+\sum_{i=0}^{m-1} 2^i p_i}{1+\sum_{i=0}^{n-1} 2^i q_i} \quad (4)$$

where p and q are the number of input and output bits to/from the GPC respectively, k is the area utilization (in LEs) of the GPC, and d is the critical path delay (in nanoseconds) of the GPC implementation. We tabulate the efficiency of each GPC studied in this work using these metrics later in this paper (Table 1 and Table 3).

2.3 Compressors



(a) Block diagram

(b) One stage of a four-operand 4b-addition using four (4:2) compressors

Figure 3: Simple (4:2) compressor example.

Compressors can be considered parallel counters, with one main difference: they have explicit carry-in (Cin) and carry-out (Cout) bits that can be connected to adjacent compressors in the same stage, as shown in Figure 3b. In contrast to carry-propagate adders, the carry-chains between compressors are not cascaded and hence reduce the critical path. Instead they are connected in a carry-save manner.

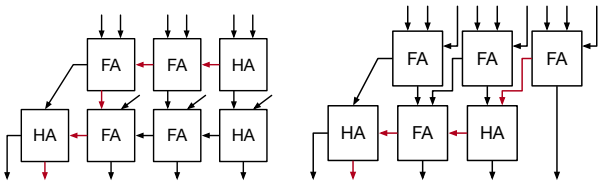
So the overall delay of the circuit scales much better (Figure 4). To the best of our knowledge, the (4:2) compressor (see Figure 3a) is the only FPGA-friendly [11]) design that targets Xilinx FPGAs, while no efficient compressors exist for Intel devices. Parandeh-Afshar et al. [19] addressed this issue by proposing configurable carry-chains as modifications to the Intel Adaptive Logic Module (ALM), supporting 6:2 and/or 7:2 compressors.

For brevity, we describe adders/compressors/GPCs with a simplified notation omitting commas. For example, we describe the GPC (6:1,1,1) as C6:111, the (4:2) compressor as C4:2, or the full adder (3:1,1) as C3:11.

2.4 Adder and Compressor Trees

For multi-operand addition, we can build adder trees by chaining multiple ripple-carry adders (RCA). Figure 4a shows addition of 3×3b operands. The carry-out from each FA/HA propagates to the next FA, which results in a long critical path along the carry-chain (shown in red). While the RCA has a small area footprint, this long delay is undesirable and can limit performance, especially for operands with large bitwidth.

The carry-save adder (CSA) [5] addresses this issue by treating the full-adder as a C3:11 compressor and breaking the carry-chain as shown in Figure 4b. By avoiding the carry chain, the delay is largely determined by the depth of the tree. However, the final stage must be reduced to the final answer using an RCA. Nevertheless, the CSA adder reduces the overall delay of the addition. For the example in Figure 4, the critical path delay has one less full-adder-delay.



(a) Ripple-carry adder. (b) Carry-save adder.

Figure 4: Examples of two types of adders.

This idea of breaking the carry-chain dependency up till the final RCA stage is the basis behind compressor trees. A compressor tree is simply a circuit that takes in a set of binary values (or dots) that represent multiple operands, and outputs the result as a sum and carry. Stage 0 in Figure 1b is a compressor tree that produces sum and carry bits as inputs into Stage 1, which are then evaluated by an RCA to produce the final result (see HA→FA→HA row in Figure 4b, which is the RCA stage). Compressor trees can be built using GPCs, compressors, or both, and efficient compressor tree design is an active area of research with large bodies of existing literature [11, 12, 18–22, 29].

The reader is encouraged to read [19] for a more detailed background on parallel counters, GPCs, compressors, and different methods of compressor tree implementations.

2.5 Xilinx FPGA Logic Elements

The Xilinx configurable logic block (CLB) [33] is composed of two slices, which are the basic unit of the FPGA's soft-fabric. Each slice is composed of four 6-input LEs, including 6-input LUT and additional circuitry such as registers and multiplexers, which give the slice its expressiveness. Figure 5a (in black) shows a quarter of the slice architecture (an 6-input LE and the corresponding circuits) found in the modern Xilinx UltraScale+ FPGAs. Another notable feature of the slice is the presence of a fast carry-chain between the LEs, which is often used to implement arithmetic circuits such as RCAs. The architectural modifications proposed in this work are at the *quarter slice* abstraction.

2.6 Intel FPGA Logic Elements

The main logic element in Intel FPGAs is the adaptive logic module (ALM) [8]. Figure 5b (in black) shows the ALM architecture of a modern Stratix-10 device. Each ALM is composed of a fracturable 6-input LUT, while primitives such as full-adders and multiplexers help to support higher-order boolean functions. Ten ALMs on Intel FPGAs are grouped to form a logic array block (LAB), which augments the ALMs with more primitives such as HyperFlex registers, local interconnect, and configurable carry-chains [8]. Our proposed modifications in this paper are at the *ALM* abstraction.

2.7 Related Work

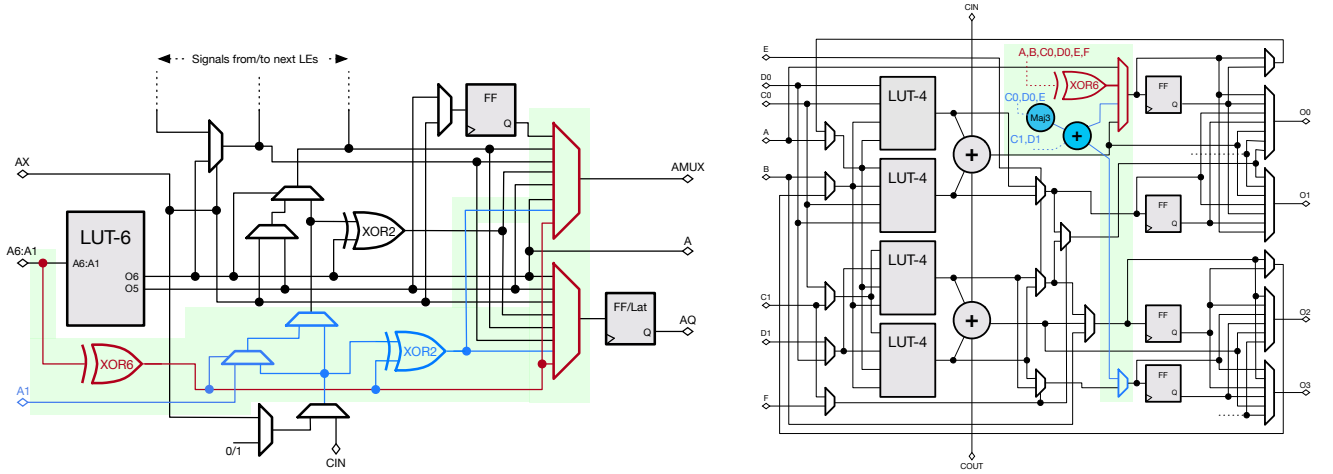
Parallel digital arithmetic circuits have been explored since the 1960s [4, 26, 30], but FPGA-based compressor trees were only popularized in the past two decades, primarily from work by Parandeh-Afshar et. al. [17, 19] and Kumm et. al. [11, 12]. In [19], the authors proposed architectural changes to the Intel ALM carry-chains such that large compressors like (6:2) and (7:2) can be efficiently mapped to single ALMs. Although their proposed compressor is very efficient, for modern applications such as BNN popcounting [13], these compressors would be significantly underutilized. Similarly, Kim et. al. [9] and Boutros et. al. [2] propose changes to the FPGA architecture, by adding sum-chain and extra carry chains respectively, specifically for modern deep neural network applications, which do not necessarily benefit general-purpose compressor trees. Our proposed changes are motivated by insight into modern GPC-based compressor tree designs, and benefit a larger suite of old and new benchmarks.

3 FPGA LOGIC CELL ENHANCEMENTS

In this section, we describe in detail the proposed hardware architecture modifications that further improve the performance of GPCs on FPGAs. We focus our efforts on improving the design of the logic cell of FPGAs from the two major FPGA vendors, Intel and Xilinx. Our modifications are organized into two tiers: (1) A vendor-agnostic change to both Intel and Xilinx FPGA logic cells, and (2), a vendor-specific modification on top that further optimizes performance. We refer to these logic cell design tiers as LUXOR, and LUXOR+ respectively. Both LUXOR and LUXOR+ are backward-compatible and retains pin-interchangeability, *i.e.* any existing design maps equally well to these new architectures.

3.1 LUXOR

Our first proposed modification is to add a 6-input XOR gate (XOR6) to both Intel and Xilinx FPGA cells. The XOR6 is parallel to the LUT and re-uses its inputs and output path as shown in Figure 5.



(a) Modifications to Xilinx UltraScale+ LE. A slice is composed of four LEs.

(b) Modifications to the Intel Stratix-10 ALM. Each ALM has 8-inputs and a fracturable LUT6 [8].

Figure 5: Basic logic element (LE) for Xilinx and Intel FPGA architectures. LUXOR modifications are highlighted in red, while vendor-specific LUXOR+ modifications are colored blue. Some signals are omitted for simplicity.

This modification is motivated by the observation that the C6:111 GPC is dominant in modern FPGA-based compressor tree designs. To quantify that claim, we analyzed optimal solutions of compressor trees from a set of 50+ micro-benchmarks that are commonly found in various domains (*e.g.* popcounting, multi-operand addition, FIR filters, etc) using efficient GPCs and compressors for Xilinx architecture from reference [22]. Figure 6 shows a histogram of the percentage count and cost (in LEs) for all GPCs across all solutions. Due to its compression efficiency, C6:111 is used more than a third of the time, and as a result, most of the hardware is dedicated towards its implementation. In modern FPGAs, the C6:111 maps to 3 LUTs, but by providing an explicit XOR6 datapath inside each logic cell, we can bring that cost down to 2 LEs. This is done by mapping the first output bit to the XOR6 rather than using a separate LE. Hence, LUXOR can deliver a resource utilization reduction for the most commonly-used GPC of up to 33%.

Another very useful feature of the LUXOR design is its applicability to binarized neural networks (BNNs). In BNNs, the core computational workload is generated by the convolution layers, which are reduced via a XnorPopcount [24] operation for the binary case. Consider the XnorPopcount operation between three binary activations (x_0, x_1, and x_2) and their corresponding binary weights (w_0, w_1, and w_2). The required computation is:

$$\text{Sum} = (w_0 \overline{\oplus} x_0) \oplus (w_1 \overline{\oplus} x_1) \oplus (w_2 \overline{\oplus} x_2)$$

$$\text{Carry, C} = (w_0 \overline{\oplus} x_0) \cdot (w_1 \overline{\oplus} x_1) + (w_2 \overline{\oplus} x_2)[(w_0 \overline{\oplus} x_0) \oplus (w_1 \overline{\oplus} x_1)]$$

where $\overline{\oplus}$ and $\oplus$ represent the XNOR and XOR operations respectively.

This XnorPopcount operation gets mapped to 2 LEs on modern FPGAs, as shown in Figure 7a – one LE to compute the sum bit, and the other to compute the carry bit. With LUXOR, however, this computation can be mapped to just a single logic element via a Boolean transformation, where the Sum bit (S) can now be

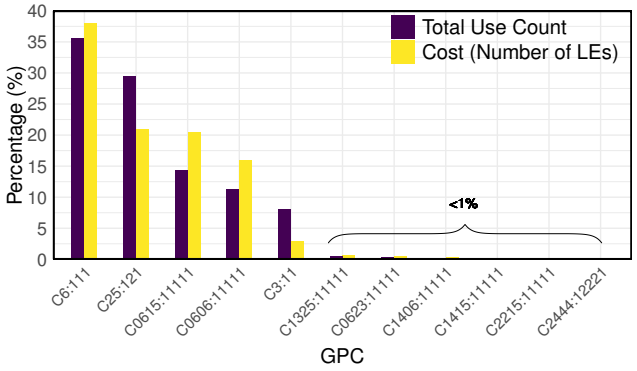


Figure 6: Total percentage count and cost of each GPC/compressor found in optimal solutions of compressor trees across 50+ Micro-Benchmarks from a variety of fields. The GPC/compressor list is according to [22]

expressed as:

$$\text{Sum} = (\overline{w_0} \oplus x_0) \oplus (\overline{w_1} \oplus x_1) \oplus (\overline{w_2} \oplus x_2)$$

which is essentially a XOR6 function where the complement of the weights are used. The LUT-6 implements the carry logic in this case, and both outputs from a single Xilinx slice can now be used to compute the partial products of the binarized convolution layer (see Figure 7b). Finally, to compute the output activations of the convolution layer, all the partial sums have to be summed, which can be visualized as a tall two-column many-operand instruction of carry and sum bits, as shown in Figure 7c. This can be efficiently reduced using a compressor tree, which is also improved by our proposed LUXOR modifications.

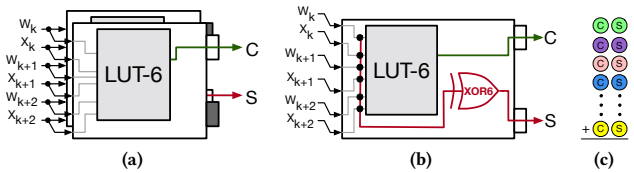


Figure 7: BNN implementation on Xilinx FPGAs: primary multiply and compressors of (a) XnorPopcount with 2 LEs, (b) XnorPopcount with 1 LUXOR LE. (c) Final two-column popcount to accumulate the partial sums (S), and carries (C)

3.2 LUXOR+

3.2.1 LUXOR+ for Xilinx FPGAs (X-LUXOR+). Reference [22] proposes the *atoms* (−06−, −14−, −22−), as primitives to construct slice-based GPCs. Atoms are 2-column-input GPCs which mapped well to half of a slice (2 LEs) and can be connected via fast in-slice carry-chains to form wider GPCs, called couple. Note, the first atom in a couple can also accept one extra input in the first rank, except −06− for structural reasons. For instance −06− and −22− atoms builds two couples as C0623:11111 and C2206:11111. All combinations of these three atoms as well as C1325:11111 (which is also a slice-based GPC but not decomposable) are listed in the baseline section of Table 1.

The blue datapath in Figure 5a highlights the proposed modification to the Xilinx FPGA slice. It involves modification of the carry chain datapath, introducing additional logic to allow the output from the XOR6 gate to be propagated into the carry-chain. This allows us to improve the implementation of slice-based GPCs. This enables us to map atom −06− to a quarter slice and consequently offers new set of slice-based GPCs such as C06060606:111111111, which can be mapped to just a single slice. This particular GPC has a very high compression efficiency of 3.75, which is more than any other existing GPCs in the literature. The X-LUXOR+ portion of Table 1 summarizes the characteristics of the new GPCs for Xilinx FPGAs.

We provide a simple illustration of the impact of our X-LUXOR and X-LUXOR+ optimizations in Figure 8. The penultimate (red) column can be implemented with a C6:111 compressor, requiring 2 LEs (instead of 3 in the unmodified case) in X-LUXOR. X-LUXOR+ is able to use the C06060606:111111111 GPC, which further reduces resource usage. In general, X-LUXOR has the greatest impact on tall-skinny compressor trees, which require significant use of C6:111, and hence has greater gains for wide compressor trees.

3.2.2 LUXOR+ for Intel FPGAs (I-LUXOR+). Note that in Figure 6, the C25:121 GPC, originally suggested in [22], is also a very efficient. Figure 9 shows that it can be implemented using two sets of two 5-shared-input functions, occupying 2 ALMs. I-LUXOR+ introduces a majority circuit and full-adder to the ALM datapath, called MajFA (blue in Figure 5b), to explicitly implement S1 and C1 while S0 and C0 can be implemented in parallel with two 5-input LUT which shares the inputs in a ALM. This modification captures C25:121 in a single ALM instead of two. In summary, I-LUXOR+ reduces the cost of two highly used GPCs, C6:111 and C25:121, by one LUT (33% and 50% respectively).

Table 1: Slice-based GPCs for Xilinx FPGAs. N.B. X-LUXOR+ area overhead is not considered in computing E, S, A.

	GPCs	p	q	LUTs	E	S	A
Baseline [11, 22]	C0606:11111	12	5	4	1.75	2.40	0.031
	C1415:11111	11	5	4	1.50	2.20	0.000
	C2215:11111	10	5	4	1.25	2.00	0.000
	C0615:11111	12	5	4	1.75	2.40	0.000
	C1423:11111	10	5	4	1.25	2.00	0.000
	C2223:11111	9	5	4	1.00	1.80	0.000
	C0623:11111	11	5	4	1.50	2.20	0.000
	C1406:11111	11	5	4	1.50	2.20	0.031
	C2206:11111	10	5	4	1.25	2.00	0.031
	C1325:11111	11	5	4	1.50	2.20	0.063
X-LUXOR+	C06060606:111111111	24	9	4	**3.75**	**2.67**	0.002
	C140606:1111111	17	7	4	2.50	2.43	0.008
	C220606:1111111	16	7	4	2.25	2.29	0.008
	C060606:1111111	18	7	4	2.75	2.57	0.008
	C060615:1111111	18	7	4	2.75	2.57	0.000
	C060623:1111111	17	7	4	2.50	2.43	0.000
	C061406:1111111	17	7	4	2.50	2.43	0.008
	C062206:1111111	16	7	4	2.25	2.29	0.008

4 ILP-BASED COMPRESSOR TREE SYNTHESIS

Many commonly used arithmetic operations such as multiplications, multiply-add, or digital filters can be expressed compactly as compressor tree hardware implementations. However, realizing efficient compressor trees is a non-trivial task that typically requires software automation. Methods to do efficient compressor tree synthesis include heuristics-guided search [16, 22, 25], integer linear programs (ILP) [12, 29], or hybrid approaches [10]. We opt for the ILP method in this work, and use ideas from [12] and [22] as inspiration. Our goal is to quantify the effect of our proposed LUXOR/LUXOR+ modifications on efficient compressor tree synthesis for commonly-used arithmetic operations in modern applications. Figure 10 encapsulates the workflow of our ILP formulation, and we detail each building block shown in the figure. Table 2 serves as a reference for all the variables used in this section. Note that, for clarity, all variable names in Table 2 are local to this section, and should not be confused with nomenclature in other sections.

4.1 Objective

There are two key metrics that quantify the effectiveness of a compressor tree implementation on FPGAs: area utilization in LUTs and the critical path delay, which is strongly correlated to the number of stages in the compressor tree. Hence, the objective function to an ILP program should be described in a way that minimizes these two metrics for each input micro-benchmark. To minimize the area cost, the objective function can be written as follows:

$$\min \sum_{s=0}^{St-1} \sum_{c=0}^{C-1} \sum_{t=0}^{T-1} V_t R_{s,t,c}$$

To model the number of stages in the objective function, the authors in [12] add the number of stages (St) as a heuristic to the

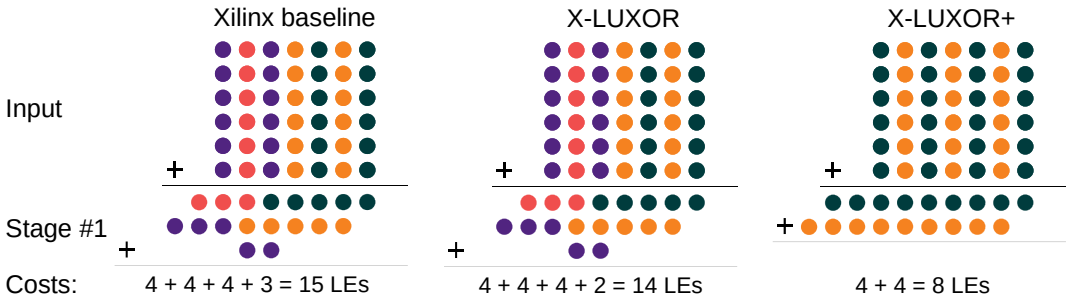


Figure 8: Example compressor tree for a 6-operand 7-bit addition using Xilinx baseline, X-LUXOR, and X-LUXOR+

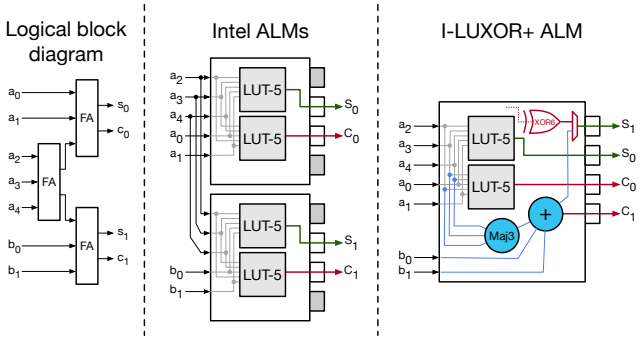


Figure 9: Efficient implementation of C25:121 GPC

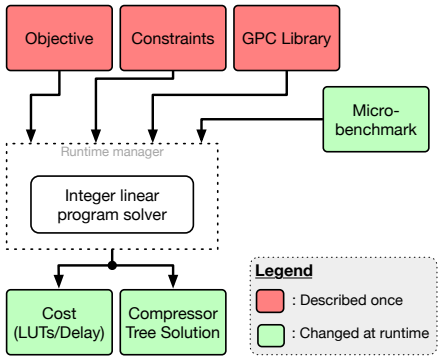


Figure 10: Flowchart of ILP-based compressor tree synthesis

cost function. However, we found this optimization strategy to be slow for difficult problems, and in some cases, the solver returns a solution that takes more stages than required. To tackle this issue, we design a runtime manager that improves the speed of the optimization process.

4.2 Runtime manager and solver

Instead of modeling St as a heuristic in the objective function, we rely on an iterative approach where we query the solver to find an optimal solution within a fixed maximum stage limit, St_{max}. This limit is relaxed incrementally until a feasible solution is found. In practice, we found that the solver was able to determine infeasibility within a few seconds, whilst being able to find a feasible integer solution within a few minutes. This iterative approach was also recently used by Kumm et. al. [10] by combining the ILP optimality search with heuristics to guide the solver. We use the IBM CPLEX v12.9 [3] ILP solver (under academic license), and design a Python3-based interface for the runtime manager using the PuLP package [15].

Table 2: Variables used in the ILP model

Var	Description
St	Number of stages in model
C	Maximum number of columns in model
X_c	Number of bits in column c of benchmark
T	Total number of compressors used
I_t	Total number of columns consumed by compressor t
V_t	Cost (in LUTs) of compressor t
$M_{t,c}$	Number of bits consumed by compressor t in column c
O_t	Total number of columns output by compressor t
$K_{t,c}$	Number of bits output by compressor t in column c
$N_{s,c}$	Number of bits in stage s of column c
$C_{s,c}$	Number of carry-bits in stage s of column c
$R_{s,t,c}$	Number of compressor t used in column c of stage s

4.3 Constraints

Since the the input stage captures the input shape of the benchmark, we set constraints on the input stage as follows:

$$N_{0,c} = X_c \qquad \text{for c = 0,1,2,...,C-1}$$

For subsequent stages, there are two constraints required to guide the solver towards a feasible compressor tree architecture, such that input/output requirements of each stage are met:

$$\sum_{t=0}^{T-1} \sum_{c'=0}^{O_t-1} M_{t,c'} * R_{s-1,t,c-c'} \geq N_{s-1,c} \qquad \text{for c = 0,1,2,...,C-1}$$

$$\sum_{t=0}^{T-1} \sum_{c'=0}^{I_t-1} K_{t,c'} * R_{s-1,t,c-c'} = N_{s,c} \qquad \text{for s = 1,2,3,...,St-1}$$

The first constraint ensures that all bits in each column of every stage are used as inputs by compressors in the next stage. The second constraint ensures that the number of bits produced by the

compressors in the previous stage matches the number of input bits in the following stage. Both these constraints can also be found in [12].

In each stage, the number of carry-bits in each column are computed in (5), where the division by two is due to the increase in the column's radix.

$$C_{s,c} = \left\lfloor \frac{C_{s,c-1} + N_{s,c-1}}{2} \right\rfloor \tag{5}$$

This can be formulated as an ILP constraint as follows:

$$C_{s,c} + 0.999 \geq \frac{1}{2}(C_{s,c-1} + N_{s,c-1}) \qquad \text{for c = 0,1,2,...,C-1}$$
$$C_{s,c} \leq \frac{1}{2}(C_{s,c-1} + N_{s,c-1}) \qquad \text{for s = 0,2,3,...,St-1}$$
$$C_{s,0} = 0$$

Note that the number of input carry-bits into the first column is always set to 0.

When solving the model iteratively, as described above, the constraints on the final stage guide the solver to converge to the solution. In [22], the author proposes a novel ragged carry-propagate architecture for the final accumulation stage for Xilinx FPGAs. This architecture reduces the overall number of stages required, and hence, we opt for this strategy on Xilinx FPGAs. Unlike [22], where the author uses a heuristic solver, we model the ragged carry-propagate adder into our model for the final stage as three constraints:

$$N_{s,c} + C_{s,c} \leq 5$$
$$C_{s,c} \leq 2 \qquad \text{for c = 0,1,2,...,C-1 and s = St-1}$$
$$N_{s,c} \leq 4$$

Finally, since Intel FPGAs cannot benefit from the ragged carry-propagate adder, we model the ILP constraints for Intel FPGAs as shown in [12] :

$$N_{s,c} \leq 3 \qquad \text{for c = 0,1,2,...,C-1 and s = St-1}$$

4.4 GPC/Compressor library

4.4.1 Xilinx compressor set. When targeting Xilinx architectures for our baseline, we use the GPC/compressor set defined by Preußer [22], who pruned a set from Kumm and Zipf [11]. For our LUXOR experiments, we reduce the cost of C6:111 GPCs from 3 to 2 logic elements, as described in Section 3.2.1. For LUXOR+, in addition to the smaller version of C6:111, we add all the new slice-based GPCs described in Table 1 to our model. We denote these results as X-LUXOR and X-LUXOR+ respectively.

4.4.2 Intel compressor set. When targeting Intel architectures, our baseline compressor set is based on a GPC set proposed by Parandeh-Afshar et al. [20], augmented with the C25:121 compressor from [22]. Since this GPC set is large, to minimise run-time of our ILP, we pruned this set using the GPC selection approach and metric described by Preußer [22].

Parandeh-Afshar et al. [20] have gathered a group of LUT-based and arithmetic-based GPCs for Intel architectures. In the first three

Table 3: Comparison of different GPCs proposed in [22] and new GPCs supported by I-LUXOR and I-LUXOR+

GPCs		*S*	*A*	Delay	LUTs	APD
[20]	C6:111	2	0.13	0.38	3	7.9
	C15:111	2	0	0.38	3	7.9
	C23:111	1.67	0	0.38	2	5.3
[22]	C25:121	1.75	0	0.38	2	11.8
Ours	C6:111	2	0.13	0.39*	2	10.95*
	C25:121	1.75	0	0.39*	1	21.9*

*Area/delay overheads for I-LUXOR+ are included (Section 5).

line of Table 3, we show the efficiency and compression metrics of our selected GPCs according to the *APD* (Equation 3) metric, which measures the efficiency of a GPC taking into account delay and resource usage. We also considered the delay itself, since some of the proposed GPCs, such as C7:111, offer slightly better *S* (compression rates) but their reported delay is 3.5× greater. In addition, we included C3:11 and C25:121 in the baseline GPC set for Intel architecture.

Similar to our X-LUXOR experiments with Xilinx architectures, we reduce the cost of C6:111 GPCs from 3 to 2 logic elements for I-LUXOR. For I-LUXOR+, as well as using the upgraded version of C6:111, we reduce the cost of C25:121 from 2 to 1 LE as described in Section 3.2.2. We also comment that the effect of the I-LUXOR and I-LUXOR+ enhancements are highlighted by the metrics, as demonstrated by the last two rows of Table3. Due to the lower logic element cost, the *APD* of both GPCs show significant improvement.

4.5 Micro-Benchmarks

To evaluate the improvements of our proposed architectures, we use different basic operations that are commonly found in various domains in three categories: 1) Low-rank inputs including popcount and two-column count (based on [22], but with additional input sizes) 2) High-rank inputs including multi-addition [12], 3-MAC operation (described below), and a FIR-3 filter from [20], and 3) BNN XnorPopcount operation for various input sizes, where the filter sizes are taken from the networks in [1, 7]. These three categories highlight the benefits and limitations of LUXOR and LUXOR+ architectures, as the chosen operations appears in various applications, especially digital signal processing and neural networks which are the most important concerns of new FPGA architectures [2, 23].

4.5.1 3-MAC operation. The 3–MAC operation is modeled according to the following equation:

$$\text{3-MAC}_{N\times(N-bit)} = \sum_{i=0}^{2} A_{i(N-bits)} \times B_{i(N-bits)} \tag{6}$$

Note that since there are 3 pairs of inputs, instead of computing partial products then and summing their results, we can select partial products of the same rank and perform a primary compression. The cost of this step is included in our result. The resulting tree forms the input to the compressor. We repeat this for different input widths (*N*).

Table 4: ASIC results for the Intel Stratix-10 ALM architecture

		Intel	I-LUXOR	Intel+MajFA	I-LUXOR+
Area	um^2	1680	1687	1715	1767
	ratio	1	1.00	1.02	1.05
Delay	*ns*	1.42	1.44	1.49	1.46
	ratio	1	1.01	1.05	1.03

Table 5: ASIC results for the Xilinx UltraScale+ slice architecture

		Xilinx	X-LUXOR	X-LUXOR+
Area	um^2	6045	6002	6361
	ratio	1.00	0.99	1.06
Delay	*ns*	0.84	0.89	0.92
	ratio	1.00	1.06	1.09

5 RESULTS

In this section we present results from experiments undertaken to evaluate the performance of the LUXOR and LUXOR+ architectural enhancements.

5.1 ASIC Modeling: Delay and Area Overheads

We model state-of-the-art Intel Stratix-10 ALM unit [8], and Xilinx UltraScale+ slice [32, 33] according to their respective data sheet descriptions. For the ASIC metric analysis, we synthesize our Verilog models using SMIC 65-nm technology standard cell by Synopsis Design Compiler 2013.12. Post synthesis results are reported and the synthesis strategy was set to "Timing Optimization" since it usually leads to a better $Area \times Delay$ product. We note that while our approach to estimating area and delay overheads using standard cells may differ slightly from an commercial full custom layout, in either case the overhead is minimal.

Table 4 gives the post-synthesis area and timing results for the Intel baseline, I-LUXOR, Intel+MajFA and I-LUXOR+ modifications to the ALM. From the table, it can be seen that the delay increase of I-LUXOR is about 1% while the area increase is less than 0.5%. This demonstrates that there is little overhead associated with adding a 6-input XOR gate to the ALM unit. In contrast, adding MajFA circuits will increase the area and delay by 2% and 5% respectively (see description in Section 3.2.2). The full I-LUXOR+ implementation, has 3% and 5% delay and area overhead respectively. We believe that the unexpectedly large increase in area compared to the individual effect of each modification arises from the performance-driven synthesis optimization. For measuring the critical path, we removed the multiplexers connecting the ALM's outputs to its input, and thus it is measured from: an input, through a LUT and two-coupled full adders (FAs) to an output multiplexer.

In a Xilinx slice, the critical path is from an input, passing through the first LUT (A) and four carry-chain circuits, and ending with the last output multiplexer. This path is also the critical path after applying LUXOR(+) for both architectures.

The synthesized Xilinx baseline slice model has an area of 6045 um^2. We compare the reported critical path with that from the Virtex-5 datasheet, which was a device that was also manufactured with the similar 65 *nm* process. Reference [31] reports the critical path from an input, through four carry circuits to the output (T_{ITO}) as 0.67, 0.77, or 0.90-*ns* for three different speed grades. Comparing these values with our value of 0.84 *ns* from Table 5, consistency with our synthesis results was verified. The same table shows that X-LUXOR has similar area utilization and a 6% increase in delay, while X-LUXOR+ has 6% area and 9% delay overheads.

Since the routing delay strongly contributes to the total delay, the LUXOR(+) delay advantages are diluted in practice. Although the X-LUXOR+ overheads are notable, because of the significant resource and performance benefits, new trade-offs are offered. For example, partially upgrading the LEs to LUXOR(+) architectures is another option. Also, with more effort in layout and buffer sizing, area and delay overheads can be recovered/balanced.

At a higher level of abstraction, LUXOR(+) does not require any I/O scheme modifications. However, they increase the logic implementation density leading to higher connectivity per LE/ALM. Thus, routing limitations may slow down LUXOR(+) enhancements. LUXOR(+) adds to the input load which also slows down the LE. This was not measured directly but taken into account in the LE measurements.

Table 6: A comparison of solutions from our ILP-based synthesis compared with those reported in [22]

Test cases	[1]H1/H2/H3[22]		Our ILP Solver					
			Baseline		X-LUXOR		X-LUXOR+	
	[2]LE	[2]Stage	LE	Stage	LE	Stage	LE	Stage
S128	101/102/101	4/3/4	100	3	79	3	78	3
S256	209/209/209	4/4/4	195	4	159	4	154	4
S512	418/422/418	5/5/5	380	5	319	5	312	4
D128	178/205/178	5/4/5	168	4	156	4	150	4
D256	360/417/360	6/5/6	328	5	315	5	298	4
D512	721/839/721	7/6/7	709	5	631	5	586	5

[1]Heuristics used in [22]: Efficiency/Strength/Product, reported in that order.
[2]LE = logic elements (LUTs), Stage = # of compressor tree stages

5.2 Benchmark Performance

The effect of our ILP approach on resource utilization in logic elements is affected by the choice of primitives in the primary stage (if applicable), compression tree stages, and the last stage (final ternary adder in Intel or the equivalent relaxed ternary adder for Xilinx architectures as proposed in [22]). Table 6 compares our technique with that of [22] for X-LUXOR and X-LUXOR+ where test cases are popcount and double column popcount operations indicated respectively by S and D, concatenated with input size. As can be seen in the baseline column, our ILP approach uses fewer logic elements (LEs) and stages for all benchmark problems compared with the heuristic approach, since an optimal solution is found. While the X-LUXOR enhancement significantly reduces the number of LEs compared with the baseline, X-LUXOR+ achieves a further reduction in the number of stages.

Figure 11 shows the savings in LEs for Xilinx architectures over a larger micro-benchmark set, with the red star also indicating a reduction in number of stages by one. For low-rank inputs (i.e. popcount and two-column popcount), the C6:111 and C25:121 compressors are heavily used. X-LUXOR improves the resource efficiency

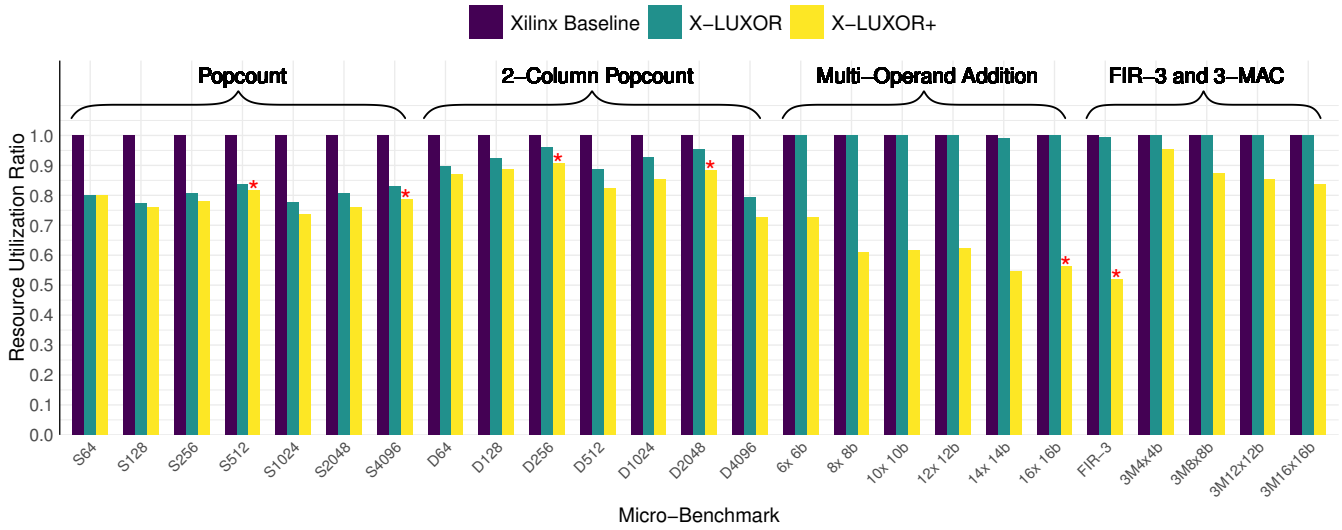


Figure 11: Resource reduction on Xilinx UltraScale+, X-LUXOR, and X-LUXOR+ architectures for various micro-benchmarks. The * indicates that the proposed solution required one less logic stage in the compression tree.

of C6:111 implementations and achieves the best savings for the 1024-input popcount problem at 22% reduction. Less improvement is seen for two-column popcount, as in the first stage, C25:121 has better arithmetic slack (A) while offering the same efficiency. This observation was also made in [1]. X-LUXOR+ offers a new set of the state of the art compression rate and compression efficiency. On average, X-LUXOR+ can reduce area utilization on the low-rank input popcount and two-column popcount benchmarks by 22% and 15% respectively.

For the high-rank benchmarks (multi-operand addition, FIR-3 and 3-MAC), the inputs are wide enough to benefit from the slice based GPCs. The C6:111 compressor is not significantly utilized. However, X-LUXOR+ offers higher compression rates and hence achieves 39% and 18% improvement in multi-addition and 3-MAC benchmarks respectively and in some cases the required number of stages is also reduced by one.

Figure 12 shows the same result for Intel I-LUXOR, and I-LUXOR+ architectures. More dramatic resource savings are apparent over Xilinx, particularly for low-rank problems using I-LUXOR+. Since I-LUXOR and I-LUXOR+ do not present new compressors, no reduction in number of stages is achieved. However, because the baseline offers no wide GPC, the resource reduction of I-LUXOR is more significant (averaging 24% and 17% for popcount and double popcount). I-LUXOR+ offers an enhanced C25:121 GPC which is the most efficient GPC for the Intel architecture. This leads to 35% and 39% resource savings for popcounting and two-column counting.

5.3 Performance on BNNs

Binarized neural networks offer a new challenge for FPGA architectures as 1-bit multiply-accumulate operations require XNOR and popcount operations to be efficient. As explained in Section 3.1 the first computation stage (Multiplication) should be merged with the early compression circuits, leading to an efficient implementation (as illustrated in Figure 7(a)). If the number of input pairs is N, $N/3$ fused units are required in the primary stage. LUXOR can implement this fused computation using a single LE rather than two LEs in the baseline architectures leading to $N/3$ fewer LE utilization. In addition, after implementation of the primary stage, a two-column counting problem with the height of $N/3$ is encountered.

As shown in Figure 13, these two optimizations lead to almost the same 34% resource reduction for LUXOR modification on both Xilinx and Intel architectures. Moreover, as described before, X-LUXOR+ cannot reduce the number of LEs significantly for low-rank inputs, and hence, the best area savings for BNNs plateaus at 37%. In the case when the input size is 3×3×256, the number of stages is reduced by one, which would give us a significant improvement in delay. In comparison, I-LUXOR+ reduces the number of LEs significantly at an average of 47%, but without reducing the number of stages.

6 CONCLUSION

This paper has discussed several low-cost FPGA logic cell modifications that can lead to significantly improved performance GPCs and the XnorPopcount operation. We then added these primitives to a set of state-of-the-art compressor tree primitives (adders, GPCs, and compressors) described in literature and built an ILP model for finding optimal solutions for FPGA-based compressor tree implementations for both Xilinx and Intel FPGAs. Using this ILP, we were able to show that our modifications lead to substantial performance gains. LUXOR is a vendor-agnostic modification, which augments each logic cell datapath with a dedicated 6-input XOR circuit, reduces the cost of the commonly used (C6:111) GPC from 3 LEs to just 2 and enables efficient XnorPopcount implementations.Over a benchmark set, this reduces the logic utilization cost of compressor trees by up to 36% (average 12–19%) on both Intel and Xilinx FPGAs, with a silicon area overhead of <0.5%. The architectural re-design is taken a step further with LUXOR+, which proposes carefully-crafted vendor-specific modifications. LUXOR+ requires

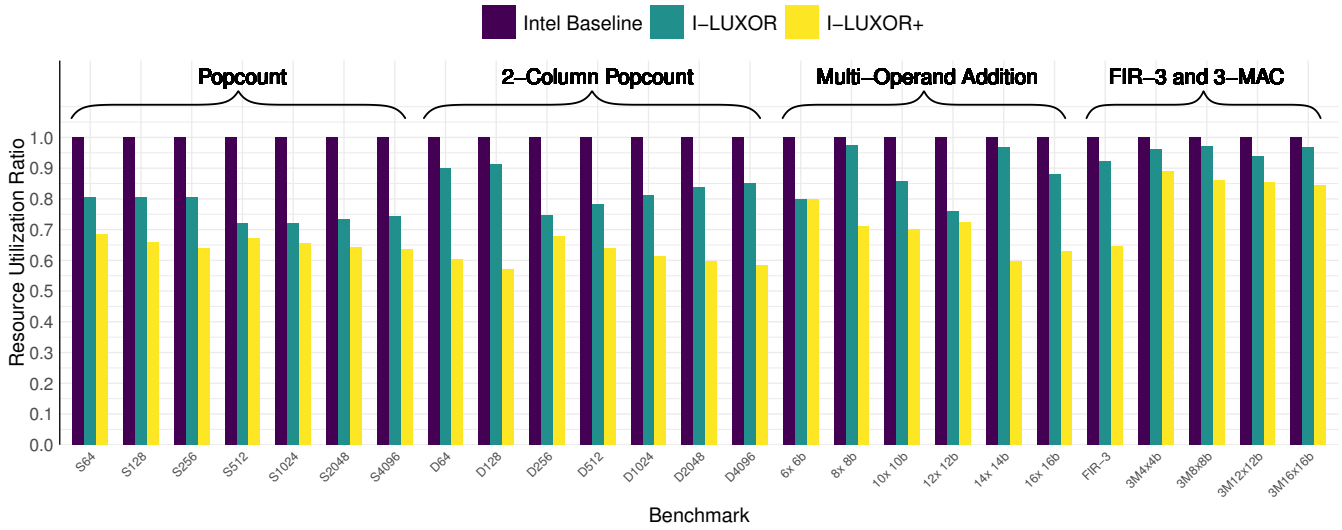


Figure 12: Resource reduction on Intel Stratix-10, I-LUXOR, and I-LUXOR+ architecture for our selected micro-benchmarks.

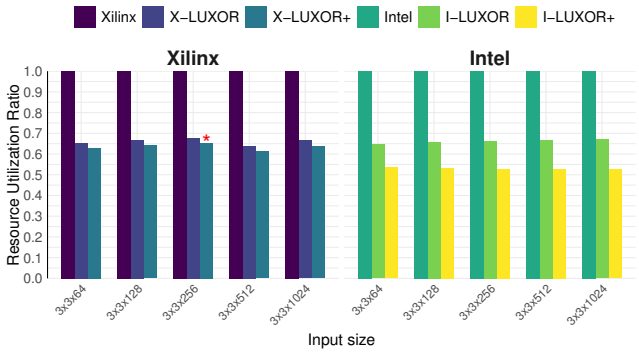


Figure 13: XnorPopcount micro-benchmarks found in BNN Convolution Layers in [7, 27]

an additional 3–6% silicon area, can improve our micro-benchmark results to up to 48% (average 26–34%).

REFERENCES

[1] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'Brien, Yaman Umuroglu, Miriam Leeser, and Kees A. Vissers. FINN-*R*: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *TRETS*, 11(3):16:1–16:23, 2018.

[2] Andrew Boutros, Mohamed Eldafrawy, Sadegh Yazdanshenas, and Vaughn Betz. Math doesn't have to be hard: Logic block architectures to enhance low-precision multiply-accumulate on FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays FPGA*, pages 94–103. ACM, 2019.

[3] IBM ILOG CPLEX. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

[4] Luigi Dadda. Some schemes for parallel multipliers. *Alta frequenza*, 34:349–356, 1965.

[5] J. G. Earle. Latched carry-save adder. *IBM Tech. Disc. Bull.*, 7(10):909–910, 1965.

[6] Milos D Ercegovac and Tomas Lang. *Digital arithmetic*. Elsevier, 2004.

[7] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems*, pages 4107–4115, 2016.

[8] Intel. UG-S10LAB Intel®Stratix®10 Logic Array Blocks and Adaptive Logic Modules User Guide, 9 2018.

[9] Jin Hee Kim, Jongeun Lee, and Jason Anderson. FPGA architecture enhancements for efficient BNN implementation. In *International Conference on Field-Programmable Technology, FPT*, pages 214–221, 2018.

[10] Martin Kumm and Johannes Kappauf. Advanced compressor tree synthesis for FPGAs. *IEEE Transactions on Computers*, 67(8):1078–1091, 2018.

[11] Martin Kumm and Peter Zipf. Efficient high speed compression trees on xilinx FPGAs. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Böblingen, Germany*, pages 171–182, 2014.

[12] Martin Kumm and Peter Zipf. Pipelined compressor tree optimization using integer linear programming. In *24th International Conference on Field Programmable Logic and Applications, FPL*, pages 1–8, 2014.

[13] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. FP-BNN: Binarized neural network on FPGA. *Neurocomputing*, 275:1072–1086, 2018.

[14] Angelo Raffaele Meo. Arithmetic networks and their minimization using a line of elementary units. *IEEE Transactions on Computers*, 100(3):258–280, 1975.

[15] Stuart Mitchell, Stuart Mitchell Consulting, and Iain Dunning. Pulp: A linear programming toolkit for python, 2011.

[16] Vojin G. Oklobdzija, David Villeger, and Simon S. Liu. A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach. *IEEE Transactions on computers*, 45(3):294–306, 1996.

[17] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. Efficient synthesis of compressor trees on FPGAs. In *2008 Asia and South Pacific Design Automation Conference*, pages 138–143. IEEE, 2008.

[18] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. Exploiting fast carry-chains of FPGAs for designing compressor trees. In *19th International Conference on Field Programmable Logic and Applications, FPL*, pages 242–249, 2009.

[19] Hadi Parandeh-Afshar, Philip Brisk, and Paolo Ienne. An FPGA logic cell and carry chain configurable as a 6:2 or 7:2 compressor. *TRETS*, 2(3):19:1–19:42, 2009.

[20] Hadi Parandeh-Afshar, Arkosnato Neogy, Philip Brisk, and Paolo Ienne. Compressor tree synthesis on commercial high-performance FPGAs. *TRETS*, 4(4):39:1–39:19, 2011.

[21] Hadi Parandeh-Afshar, Ajay Kumar Verma, Philip Brisk, and Paolo Ienne. Improving fpga performance for carry-save arithmetic. *IEEE transactions on very large scale integration (VLSI) systems*, 18(4):578–590, 2009.

[22] Thomas B. Preußer. Generic and universal parallel matrix summation with a flexible compression goal for xilinx fpgas. In *27th International Conference on Field Programmable Logic and Applications, FPL*, pages 1–7, 2017.

[23] SeyedRamin Rasoulinezhad, Hao Zhou, Lingli Wang, and Philip H. W. Leong. PIR-DSP: an FPGA DSP block architecture for multi-precision deep neural networks. In *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM*, pages 35–44, 2019.

[24] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision - ECCV*, pages 525–542, 2016.

[25] Paul F Stelling, Charles U Martel, Vojin G Oklobdzija, and R Ravi. Optimal circuits for parallel multipliers. *IEEE Transactions on Computers*, 47(3):273–285, 1998.
[26] Earl E Swartzlander. Parallel counters. *IEEE Transactions on computers*, 100(11):1021–1024, 1973.
[27] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays FPGA*, pages 65–74. ACM, 2017.
[28] Ajay K Verma, Philip Brisk, and Paolo Ienne. Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1761–1774, 2008.
[29] Ajay K. Verma and Paolo Ienne. Automatic synthesis of compressor trees: reevaluating large counters. In *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*, pages 443–448, 2007.
[30] Christopher S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. Electronic Computers*, 13(1):14–17, 1964.
[31] Xilinx. DS202 (v5.5) Virtex-5 FPGA Data Sheet:DC and Switching Characteristics, 6 2016.
[32] Xilinx. UG474 7 Series FPGAs Configurable Logic Block, 9 2016.
[33] Xilinx. UG574 UltraScale Architecture Configurable Logic Block, 2 2017.
[34] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani B. Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable FPGAs. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA*, pages 15–24, 2017.