FPGA Acceleration of Probabilistic Sentential Decision Diagrams with High-level Synthesis

YOUNG-KYU CHOI, Inha University, South Korea and University of California, Los Angeles
CARLOS SANTILLANA, YUJIA SHEN, ADNAN DARWICHE, and JASON CONG,
University of California, Los Angeles

Probabilistic Sentential Decision Diagrams (PSDDs) provide efficient methods for modeling and reasoning with probability distributions in the presence of massive logical constraints. PSDDs can also be synthesized from graphical models such as Bayesian networks (BNs) therefore offering a new set of tools for performing inference on these models (in time linear in the PSDD size). Despite these favorable characteristics of PSDDs, we have found multiple challenges in PSDD's FPGA acceleration. Problems include limited parallelism, data dependency, and small pipeline iterations. In this article, we propose several optimization techniques to solve these issues with novel pipeline scheduling and parallelization schemes. We designed the PSDD kernel with a high-level synthesis (HLS) tool for ease of implementation and verified it on the Xilinx Alveo U250 board. Experimental results show that our methods improve the baseline FPGA HLS implementation performance by 2,200X and the multicore CPU implementation by 20X. The proposed design also outperforms state-of-the-art BN and Sum Product Network (SPN) accelerators that store the graph information in memory.

CCS Concepts: • **Computer systems organization** → **High-level language architectures**; **Reconfigurable computing;**

Additional Key Words and Phrases: PSDD, HLS, FPGA

ACM Reference format:
Young-kyu Choi, Carlos Santillana, Yujia Shen, Adnan Darwiche, and Jason Cong. 2023. FPGA Acceleration of Probabilistic Sentential Decision Diagrams with High-level Synthesis. *ACM Trans. Reconfigurable Technol. Syst.* 16, 2, Article 18 (March 2023), 22 pages.
https://doi.org/10.1145/3561514

1 INTRODUCTION

Probabilistic Sentential Decision Diagrams (**PSDDs**) were recently proposed to model distributions over structured probability spaces that are defined by massive logical constraints [22, 36]. Traditionally, probability distributions are modeled using graphical models such as **Bayesian**

This research is supported by Inha University Research Grant, **National Research Foundation** (**NRF**) Grant funded by Korea **Ministry of Science and ICT** (**MSIT**) (2022R1F1A1074521), US NSF Grant on RTML: Large: Acceleration to Graph-Based Machine Learning (CCF-1937599), and Xilinx **Heterogeneous Accelerated Compute Cluster** (**HACC**) Program.
Authors' addresses: Y.-k. Choi (corresponding author), Inha University, 100 Inha-ro Hitech 1012, Incheon, South Korea, 22212 and University of California, Los Angeles, 404 Westwood Plaza, Los Angeles 90095, California; email: ykc@inha.ac.kr; C. Santillana, Y. Shen, A. Darwiche, and J. Cong, University of California, Los Angeles, 404 Westwood Plaza, Los Angeles 90095, California; emails: {csantillana21, syj0614}@gmail.com, {darwiche, cong}@cs.ucla.edu.

1936-7406/2023/03-ART18 $15.00
https://doi.org/10.1145/3561514

networks (**BNs**) [15, 23, 26, 28]. A BN employs a **directed acyclic graph** (**DAG**) to capture dependencies among random variables. In the presence of massive logical constraints, which naturally arise in many domains, the DAG can become too highly-connected to allow efficient reasoning and learning in real-world applications. PSDDs, on the other hand, provide a tractable representation of probability distributions in this case because they are based on a sophisticated representation of logical constraints known as **Sentential Decision Diagrams** (**SDDs**), which generalize and can be exponentially smaller than **Ordered Binary Decision Diagrams** (**OBDDs**) [2, 16]. The effectiveness of PSDDs has been demonstrated in numerous real-world applications with massive logical constraints. As an example, a classical Naive Bayesian classifier of a board game trace requires 362,879 parameters, whereas a PSDD needs 1,793 parameters [10]. Other successful examples of PSDDs include learning user preferences [8], anomaly detection [10], and route distribution modeling [9, 36]; see [17] for a survey.

PSDDs are normally synthesized from a combination of data and symbolic knowledge in the form of logical constraints. For example, in [36], PSDDs represent a distribution over routes, which are modeled as paths in a graph. A path is encoded as a binary instantiation over the edge variables. The variable is set to 1 if the edge is used on the path and 0 otherwise. Some instantiations of the variables correspond to invalid paths in the graph (disconnected edges), and this leads to logical constraints. To learn a probability distribution over the valid paths, one can first construct a PSDD from the logical constraint among the edge variables. Then we can estimate the weights on the PSDD from the data that consists of routes that are more frequently taken by the travelers.

PSDDs can also be synthesized from graphical models such as Bayesian networks [35], positioning them as an inference tool for such models. Since PSDDs are synthesized, PSDDs can be very large and exhibit some strong properties in comparison to other tractable circuit representations such as Sum Product Networks [30] and Cutset Networks [31] (which are normally handcrafted or learned from data). All these models are variants of **Arithmetic Circuit** (**AC**) representations of probability distributions [14], which allow inference in time linear to the circuit size; see [18] for a recent survey of these circuit representations.

A **field-programmable gate array** (**FPGA**) is a high-performance, energy-efficient reconfigurable platform that has accelerated many probabilistic inference problems. Examples include: Sum Product Network [33, 37, 38], Bayesian Monte Carlo Makov Chain Model Inference [45], Bayesian Computing Machines [25], Bayesian Neural Networks [3], and Bayesian Inference with Arithmetic Circuits [19, 27, 43]. But to our knowledge, there is no previous work that accelerates PSDDs.

We accelerate PSDD on the Xilinx Alveo U250 FPGA Acceleration Card [41]. We implemented the two commonly used PSDD queries: the probability of the most probable explanation (MPE_p) query and the marginal (MAR) query. In order to reduce the development effort, we program in C and generate the bitstream using a **high-level synthesis** (**HLS**) tool [13, 40]. But we found that accelerating the PSDD kernel in HLS presents a unique set of challenges. As we construct a PSDD graph from a BN, the sparseness of the node connections leads to each node having only a small number of child nodes. This property increases the proportion of a loop's pipeline epilogue/prologue compared to the loop's length (details in Section 3). We could try the edge-centric processing scheme [32, 44] to solve this problem, but a straightforward implementation in HLS causes new dependency issues and worsens the **initiation interval** (**II**) of the pipelined loops (details in Section 3). Moreover, in order to perform a query on a real-world application with massive logical constraints, we process a large graph that may not fit on an FPGA. Thus, we have to make the **processing elements** (**PEs**) configurable for irregular tree connection patterns—but this complicates the parallelization process.

In this article, we present optimization techniques to solve these problems. We propose a novel HLS-based edge-centric processing scheme that achieves an II of 1 for a PSDD query with

dependency issues. The scheduling for this scheme is lightweight and can be done in a few seconds. Moreover, we exploit multiple levels of parallelism that can be applied even if the entire graph cannot fit on an FPGA. The proposed optimizations are fully compatible with HLS, and the design was verified on-board. Compared to related works whose performance largely depends on the size of the graph, our work better retains the performance for datasets with various sizes.

2 BACKGROUND

2.1 Probabilistic Sentential Decision Diagrams

Figure 1(a) shows an example PSDD, which is based on a Boolean circuit known as a SDD [16] that is annotated with probabilistic parameters. The PSDD is composed of fragments shown in Figure 2. In an internal fragment, the OR-gate can have an arbitrary number of inputs. Each child of the OR-gate is associated with a parameter α_i. The AND-gates have precisely two inputs each. Each left child p_i is called a *prime,* and each right child s_i is called a *sub*. Figure 1(a) highlights some PSDD fragments.

Each PSDD conforms to a tree of variables, called a *vtree* [29]. A vtree is a binary tree whose leave are the circuit variables (Figure 1(b)). The conformity is roughly as follows. For each internal fragment with primes p_i and subs s_i, there must exist a vtree node v where the variables of each prime p_i appear in the left child of v, and the variables of each sub s_i appear in the right child of v. For example, the root fragment of the PSDD in Figure 1(a) conforms to vtree node 3 in Figure 1(b). Each prime of this PSDD fragment, fragments 2 and 4, conforms to vtree node 1. Each sub, fragments 3 and 5, conforms to vtree node 5.

2.2 PSDD Semantics and Queries

A PSDD represents a probability distribution over the binary variables $\mathbf{X}$ that appear in the vtree. We use a bold letter $\mathbf{X}$ to represent a set of variables, and a normal letter X to represent a single variable.[1] The semantics of a PSDD can be defined by unfolding it into an AC through replacing each AND-gate with a multiplication operation and each OR-gate with a weighted sum. The weights are the parameters that annotate the children of the OR-gate. By properly setting the leaf literals of this AC for a given instantiation $\mathbf{x}$ of variables $\mathbf{X}$, the AC will evaluate to the probability of instantiation $\mathbf{x}$.

To evaluate the AC at a given variable instantiation $\mathbf{x}$, we set each leaf literal in the AC to 1 if it is *compatible* with instantiation $\mathbf{x}$, and to 0 otherwise. A positive X is compatible with an instantiation that sets $X = 1$, and it is not compatible with an instantiation that sets $X = 0$. The opposite is true for the negative literal $\neg X$. To evaluate the probability of an instantiation, e.g., $A = 0, B = 0, C = 0, D = 0$, we compute the value of each fragment in a bottom-up order. The following table shows the values of the literals:

Literals	A	$\neg A$	B	$\neg B$	C	$\neg C$	D	$\neg D$
Values	0	1	0	1	0	1	0	1.

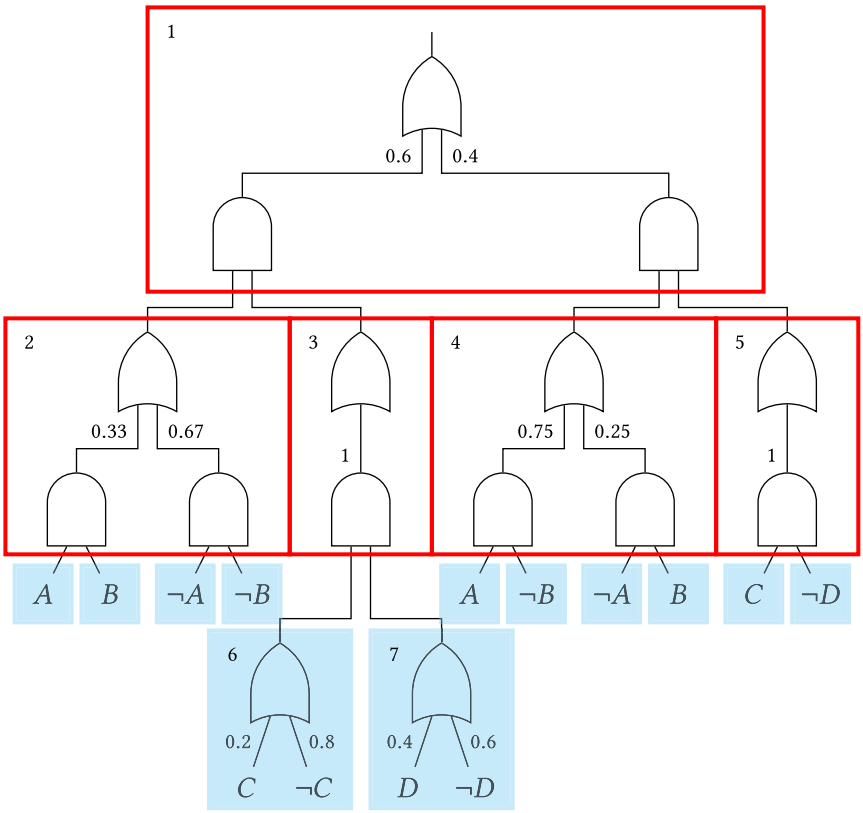
The value of each fragment is computed using the corresponding arithmetic operations. For example, the value of the internal fragment 2 is computed as

$$0.33 \times 0 \times 0 + 0.67 \times 1 \times 1 = 0.67,$$

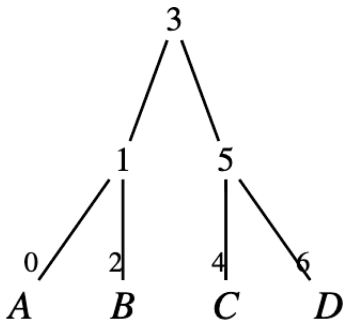
and the value of fragment 4 is

$$0.75 \times 0 \times 1 + 0.25 \times 1 \times 0 = 0.0.$$

[1] We abuse the notation X to also represent a positive literal of variable X.



(a) PSDD



(b) Vtree

Fig. 1. Figure 1(a) shows an example PSDD. Each fragment is boxed. Internal fragments are surrounded by an empty red box, and leaf fragment is surrounded by shaded boxes. Further, some fragments are indexed for reference purpose, and the index is indicated on the top left of the box. Figure 1(b) shows the vtree that the PSDD conforms to.

Given a variable instantiation, the value of each fragment represents the conditional probability $\Pr(\mathbf{x} \mid \{\mathbf{y}_0, \ldots, \mathbf{y}_n\}$, where the conditions $\{\mathbf{y}_0, \ldots, \mathbf{y}_n\}$ are called *contexts* [22]. The query $\mathbf{x}$ corresponds to the subset of the input instantiation whose variables appear beneath the fragment. If the input instantiation is $A = 0, B = 0, C = 0, D = 0$, the query for fragment 4 corresponds

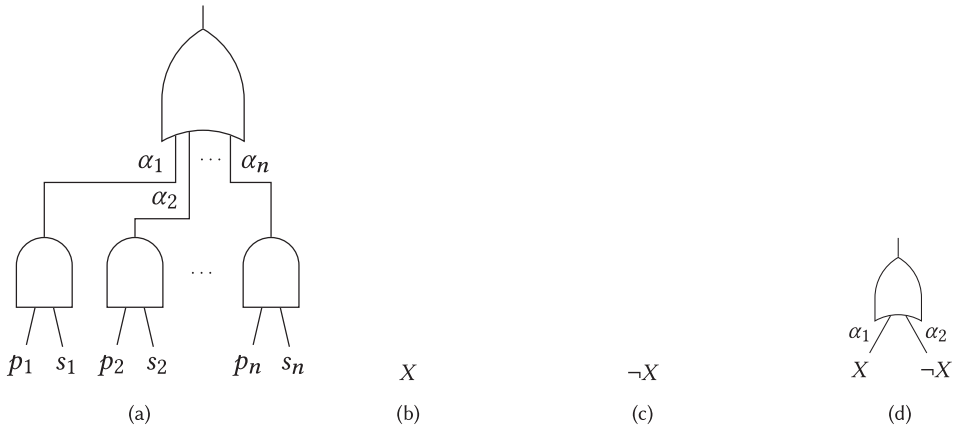


Fig. 2. Four types of PSDD Fragments: internal (2(a)) and boundary, which can be a positive literal (2(b)), a negative literal (2(c)), or a simple OR-gate (2(d)). A PSDD can also have leaf nodes representing false, but we omit these as they are not necessary for our acceleration.

to $A = 0, B = 0$. Roughly speaking, the context consists of a set of input instantiations whose probability computation depends on the fragment (please refer to [22] for details). The context for fragment 4 consists of instantiations $A = 1, B = 0, C = 1, D = 0$ and $A = 0, B = 1, C = 1, D = 0$. Then the value of fragment 4 equals to $\Pr(A = 0, B = 0 \mid \{\{A = 1, B = 0, C = 1, D = 0\}, \{A = 0, B = 1, C = 1, D = 0, \}\})$, which would have a probability of 0.0 (matches the value of the corresponding arithmetic operation).

The values of the rest of the labeled fragments are listed in the following:

Fragment ID	7	6	5	4	3	2	1
Value	0.60	0.80	0.00	0.00	0.48	0.67	0.19.

2.3 Most Probable Explanation and Marginal Queries

We have implemented the two commonly used probabilistic queries: the probability of the most probable explanation (MPE_p) and marginal (MAR) queries. The queries are discussed in detail in this section.

2.3.1 MPE_p. The *most probable explanation* (MPE) query finds the most likely instantiation $\mathbf{x}$ of variables $\mathbf{X}$ that is compatible with a given evidence. The evidence is described by an instantiation $\mathbf{e}$ over a subset of the PSDD variables $\mathbf{E} \subseteq \mathbf{X}$. For example, in natural language processing, we can predict the most likely sentence structure given an observed sentence. We could construct a model of the sentence that describes the probability distribution over sentences and grammatical structures. The MPE query can be invoked to find the most likely sentence structure that is compatible with the observed sentence.

The MPE query is computed by evaluating an MPE circuit, which is constructed similarly to the AC of a PSDD. The difference is that the max operation replaces the OR-gate of AC [4]. For example, the internal fragment in Figure 2(a) evaluates to

$$\max_i \{\alpha_i \times \texttt{value}(p_i) \times \texttt{value}(s_i)\}, \tag{1}$$

where $\texttt{value}(p_i)$ and $\texttt{value}(s_i)$ are the values of primes and subs for that fragment.

In this article, we explain the methodology for performing an MPE_p query—which computes the *probability* of the most likely instantiation by evaluating the MPE circuit. This is the most

compute-intensive part of the MPE query. The most likely instantiation can be found by recording the child node that provides the maximal probability for each node and performing a backward pass over the MPE circuit to recover the corresponding instantiation (see Section 12.3.2 in [15] for details). For simplicity, we will refer to *MPE$_p$ query* as the *MPE query* from now on.

Suppose $\mathbf{e}$ is an instantiation of variables $\mathbf{E} \subseteq \mathbf{X}$. We can compute the probability of an MPE query by evaluating the AC of a PSDD as discussed earlier. One difference is that both literals of a variable $Y \in \mathbf{X} \setminus \mathbf{E}$ will be set to 1 since they are both compatible with the instantiation $\mathbf{e}$. For example, given evidence $B = 1$, literal $\neg B$ is set to 0 and all other literals are set to 1. Evaluating the AC in Figure 1(a) under this setting yields the following values to fragments:

Fragment ID	7	6	5	4	3	2	1
Value	0.60	0.80	1.00	0.25	0.48	0.33	0.1.

Then, the probability of the most likely instantiation that is compatible with $B = 1$ is 0.1 (the value of Fragment 1).

2.3.2 MAR. The *marginal* query (MAR) calculates the probability of an instantiation over a subset of variables $\mathbf{E} \subseteq \mathbf{X}$ and is one of the most common probabilistic queries. For example, in medical diagnosis, the model describes a probability distribution that assigns a probability to every possible symptom and disease. The marginal query can be used to compute the probability of a particular symptom or the probability of a disease given a symptom (with Bayes conditioning).

For a MAR query, the internal fragment in Figure 2(a) evaluates to

$$\Sigma_i \{\alpha_i \times \texttt{value}(p_i) \times \texttt{value}(s_i)\}. \tag{2}$$

The difference with the MPE query in Equation (1) is that the summation operation (instead of the max operation) replaces the OR-gate of AC.

When evaluating the marginal probability at evidence $\mathbf{e}$, leaf literals are set in a similar way as the MPE query. Every leaf literal compatible with $\mathbf{e}$ is set to 1 and every leaf literal incompatible with $\mathbf{e}$ is set to 0. Consider again the PSDD in Figure 1(a). Given evidence $B = 1$, literal $\neg B$ is set to 0 and all other literals are set to 1. Evaluating the AC yields the following values for fragments:

Fragment ID	7	6	5	4	3	2	1
Value	1.00	1.00	1.00	0.25	1.00	0.33	0.29.

Thus the marginal probability of $B = 1$ is 0.29.

3 BASELINE HLS IMPLEMENTATION AND CHALLENGES

In this section, we present the baseline MPE query implementation in HLS and identify the challenges in the acceleration. We compute the output of each PSDD fragment (we will simply refer to this as a *node* from now on) in the outermost loop (line 1) of Figure 3. As explained in Equations (1) and (2), each node performs a maximum (MPE) or addition (MAR) operation for OR gates, and a multiplication for AND gates. We will refer to these operations as the *edges* for the rest of the article. Each node processes its edges in the innermost loop shown in line 3 of Figure 3. For the MPE query, we use the maximum operation; for the MAR query, we use the addition operation (line 6 of Figure 3). Since the MPE and MAR queries have a very similar computation pattern (Equations (1) and (2)), we employ the C ternary operator (which implies a mux in HLS) to choose between the result of the two different queries. The outermost loop in line 1 traverses through all the nodes in a bottom-up fashion.

We store α_i in the array `weight[]`, the value of the node probability for the prime $\texttt{value}(p_i)$ and the sub $\texttt{value}(s_i)$ in the array `prob[]`, the number of p_i and s_i for each node in the array

```
1 for(n = 0; n < NODE_NUM; n++){     //bottom-up traversal on all nodes
2   node_prob = 0;
3   for(c = 0; c < edge_num[n]; c++){            //edge traversal loop
4 #pragma HLS pipeline                           //HLS pipeline pragma
5     edge_prob = prob[prime[c]] * prob[sub[c]] * weight[c];
6     node_prob = (mar == 1) ? node_prob + edge_prob :    //add for MAR
                              {(edge_prob > node_prob) ?
                                 edge_prob : node_prob};  //max for MPE
7   }
8   prob[n] = node_prob;       //update prob after traversing all edges
9 }
```

Fig. 3. Baseline HLS implementation for PSDD acceleration.

edge_num[], and the node index of p_i and s_i in prime[] and sub[]. The node probability values are stored in an integer type.

The array prob[] is accessed very frequently—we need to fetch the probability for primes and subs (line 5) and update the result (line 8). Thus, prob[] is stored in the FPGA internal memory. Moreover, a considerable amount of memory is required to process large networks. The natural choice is to assign prob[] to **Ultra-RAM** (**URAM**) [42], which is the largest internal memory resource[2] in Alveo U250.

The most commonly used optimization techniques for accelerating an HLS kernel are pipelining and unrolling [40]. One could consider pipelining the loop in line 1 of Figure 3 and unrolling the loop in line 3, but edge_num[] is a variable that makes it difficult for HLS compilers to determine the unrolling factor. Another option is to pipeline the loop in line 3—the code after applying the pipeline compiler directive (#pragma HLS pipeline) is shown in Figure 3. We will refer to this code as the baseline implementation.

To test the baseline implementation, we utilize a PSDD dataset compiled from the Mastermind network using the method described in [35]. The Mastermind network is a commonly used Bayesian network that models the Mastermind game. This network exhibits a *local structure* [5] that can be exploited when compiling the PSDD. Unlike general PSDD graphs, each node in the Mastermind network has at most two children nodes because it is synthesized from a BN [35]. But we found that such sparsity in the network causes a severe negative effect on the performance of an HLS-based implementation.

The problem we faced was the low processing rate of the pipelined loop. Even after adding the pipeline pragma (line 4 of Figure 3), the averaged processing rate in the Mastermind dataset turned out to be only 0.06 nodes per cycle. This is because, in the Mastermind dataset, each node has only 1.1 prime and sub-child nodes on average. When the innermost loop is invoked, the loop requires 12 cycles of pipeline epilogue/prologue overhead cycles. If the averaged loop iteration is only 1.1, then the overhead dominates the time spent in actual computation.

To solve this problem, we refactor the code to be edge-centric [32, 44]. That is, we flatten the outer loop in line 1 of Figure 3 with the inner loop in line 3, and we iterate on the index of the operations. The modified HLS code is shown in Figure 4. Now there is only a single loop that traverses through all the edges in a bottom-up fashion (line 1 of Figure 4). Even if each parent node has only a few child nodes, the loop no longer suffers from the repetitive pipeline epilogue/prologue overhead cycles.

Although the new processing scheme has the potential to achieve a higher processing rate, the naive HLS implementation in Figure 4 has several new problems. One of them is the overhead of

[2]Alveo U250 has 45 MB of Ultra-RAM (URAM) and 12 MB of Block-RAM (BRAM).

```
1 for(e = 0; e < EDGE_NUM; e++){   //bottom-up traversal on all edges
2 #pragma HLS pipeline
3   node_prob = prob[parent[e]];
4   edge_prob = prob[prime[e]] * prob[sub[e]] * weight[e];
5   prob[parent[e]] = (mar[e] == 1) ? node_prob + edge_prob :   //MAR
                                  {(edge_prob > node_prob) ?
                                   edge_prob : node_prob};      //MPE
6 }
```

Fig. 4. Naive edge-centric processing in HLS.

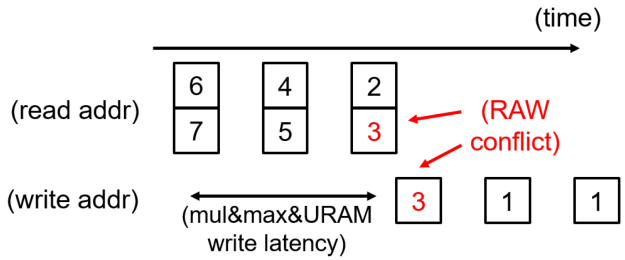


Fig. 5. A possible dependency problem for Figure 1(a) after adopting edge-centric processing in Figure 4.

storing more data. We need the index `parent` to keep track of the parent of each edge (lines 3 and 5 of Figure 4). This is a minor issue, and it can be easily solved with additional memory.

The next issue is the dependency problem. Reading probability, performing the multiplication and the max/add operations, and writing probability take several cycles of latency—for an integer variable, it took 12 cycles. This is a true dependency, because a probability written to a parent node may be read as a child prime/sub node probability in subsequent iterations. Forcing the HLS tool to ignore this dependency results in a **read-after-write** (**RAW**) hazard. An example is illustrated in Figure 5— for the PSDD in Figure 1(a), the output (node 2) after processing nodes 6 and 7 will be written 12 cycles later. The RAW hazard occurs if the probability for node 3 is read before the updated value is written. This challenge will be addressed in Section 4.1.

Moreover, the naive edge processing scheme has a local memory port limitation problem. Even if the dependency problem is somehow solved, the loop in Figure 4 cannot be pipelined to 1 because the probability (colored red) is read three times (from addresses `prime[e]`, `sub[e]`, and `parent[e]`) and written once (to address `parent[e]`) every iteration. We cannot solve this problem with the true dual-port mode [42] because `prime[e]` and `sub[e]` addresses may be different; the true dual-port mode only supports two independent addresses. The array partitioning technique [12, 40] also does not help because we cannot guarantee that the read addresses and the write address will always be different. We will discuss the solution to this local memory port problem in Section 4.2.

The last issue is the lack of adequate parallelism. To increase parallelism, we could consider partial unrolling [11] of the loop in line 1 of Figure 4. But such an approach also causes a dependency problem because the primes and subs of an iteration may be written in the next iteration. For a small graph, we can resolve this issue by exploiting the operation-level parallelism—that is, we could map the entire AC onto the FPGA similar to [37, 43]. But this is not feasible for Mastermind because we operate on a large graph with 42,558 nodes. It is possible to implement a part of the graph on the FPGA, but the irregularity of the graph makes it difficult to supply the operands to each OR/AND-gate without stalling. We will explain how to solve this problem in Section 5.

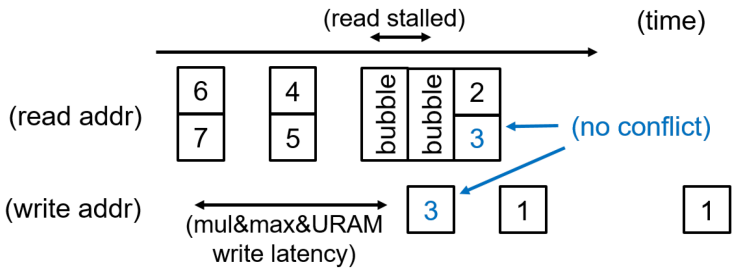


Fig. 6. Removing the inter-depth dependency problem in Figure 5 with bubble insertion.

4 PROCESSING RATE OPTIMIZATION IN HLS

In this section, we propose an HLS-based method to improve the processing rate and the II of the edge-centric processing scheme.

4.1 Depth-Batched Static Scheduling

We will describe how to solve the dependency issue (Section 3) that emerged from the edge-centric processing scheme. There are two approaches to this problem. One is a dynamic solution that detects nodes with data hazards. An example can be found in [44] which employs a complex mutex-based locking mechanism on the target node. But this increases the hardware cost. Instead, we take a static approach. This is a hardware-friendly solution exploiting the fact that the PSDD structure of a dataset is fixed, even for different queries. This characteristic allows us to explicitly manage the scheduling in an offline compilation.

A related static approach was proposed for **sum-product network (SPN)** in [24], where they formulate an **Integer Linear Programming** (**ILP**) problem for the modulo scheduling and the binding of arithmetic operators. In this article, we take a more lightweight approach to quickly determine the static schedule. We first perform a topological sorting of the nodes starting from the head node of PSDD tree, and we assign a tree level (depth of a tree) to each node. For the example in Figure 1(a), node 1 is assigned depth 0, nodes 2, 3, 4, and 5 are assigned depth 1, and nodes 6 and 7 are assigned depth 2. Then we batch-process all nodes in the same depth. Since we take a bottom-up approach, the nodes with the largest level (deepest depth) are first processed in a batch, and then the nodes in the upper level are batch-processed, and so on.

After this process, all the primes and subs writing to a common parent node will have the same depth and can be easily controlled to avoid the dependency problem (more details in the common parent clustering approach in Section 4.2). The only RAW hazard now remaining is the inter-depth RAW hazard—an example was illustrated in Figure 5 where node 3 (depth 1) is read before the probability write in-depth 2 is completed. This issue is resolved by adding bubbles (no-op) into the computation slots that have inter-depth dependency problems (Figure 6). PEs will wait until the conflicting write operation to node 3 is resolved. The computation pipeline is stalled as a result, but because we batch-process each depth, the proportion of the stall is small compared to the number of nodes processed in each depth. In the Mastermind dataset, the proportion of bubble cycles is only 0.11%.

Figure 7 presents the HLS kernel code after applying static scheduling. We pack the bubble instruction, the node index of primes/subs, and weights into the array `edge_schedule[]`. Then the array is read and decoded in the PEs (lines 6–7 of Figure 7). The bubble instruction stops the probability storage from being updated (lines 9–12 of Figure 7). Also, we add the "dependency inter false" pragma on array `prob[]` in line 4 of Figure 7 to inform the HLS tool that the dependency is

```
1  node_prob = 0;                          //initialize node probability
2  for(e = 0; e < EDGE_NUM; e++){   //bottom-up traversal on all edges
3  #pragma HLS pipeline
4  #pragma HLS dependency inter false var=prob
5                                          //programmer manages dependency
6    schedule = edge_schedule[e];   // reads static schedule from DRAM
7    {bubble, mar, node_end, parent, prime, sub, weight} = schedule;
                                                        // decode schedule
8    edge_prob = prob[prime] * prob[sub] * weight;
9    if( bubble == false ){         //no-op if there is a bubble instr
10     node_prob = (mar == 1) ? node_prob + edge_prob :          //MAR
                               {(edge_prob > node_prob) ?
                                 edge_prob : node_prob};         //MPE
11     prob[parent] = node_prob;                  //store updated prob
12   }
13   if( node_end == true ){ node_prob = 0; }     //init for next node
14 }
```

Fig. 7. The HLS code after applying the depth-batched static scheduling (Section 4.1) and the common parent clustering (Section 4.2) [blue variables are the decoded static schedule, red variables are the accesses to the probability array, and green directive allows a programmer to manage dependency].

now managed by the programmer. The static schedule is accessed once per edge per query,[3] so it is stored in the external DRAM and passed to the processing elements in a streaming fashion.

The order of processing is the same for the same dataset, so we can generate the static schedule offline and reuse it for various different PSDD queries. Since it is determined offline, the proposed solution does not increase the hardware cost. Also, whereas the ILP-based approach [24] typically takes tens of minutes to determine the schedule, our approach can be finished in a matter of a few seconds because assigning a depth to all nodes in a tree has a low complexity (more details to be presented in Sections 5.1 and 6.2).

In addition to resolving the dependency problem, the static scheduling scheme provides another benefit of reducing the probability of storage. Rather than allocating one node's probability to each address space in the array `prob[]`, we time-share the array. This is possible because the address space for nodes that will be no longer be accessed can be reused to store other nodes' probabilities. For the example in Figure 1(a), the probability for node 3 only needs to be stored in a memory space from the clock cycle when nodes 6 and 7 are processed to the clock cycle when node 1 is processed. The space allocated for node 3 can be used for other nodes in all other clock cycles. This technique reduces the node storage for the Mastermind dataset by 81%.

If the MPE and MAR queries are implemented on separate PEs, we can insert fewer bubbles to an MPE query because the max operation is simpler than addition (6 vs. 12 cycles latency). But instead, we apply the same static schedule for both types of queries. This is because we utilize the same pipeline architecture. We wanted to quickly switch between different types of queries without reprogramming the FPGA. This helps decrease the computation latency even if there is a diverse type of incoming queries.

4.2 Resolving Memory Port Limitation in HLS

As demonstrated in Section 3 and Figure 4, we cannot achieve an II of 1 in the naive edge processing scheme because of the internal memory port limitation problem. We will detail how to solve this problem in the HLS syntax.

[3]The schedule is reused in the query-level parallelism that will be explained in Section 5.2.

```
1  node_prob = 0;                          //initialize node probability
2  for(e = 0; e < EDGE_NUM; e++){   //bottom-up traversal on all edges
3  #pragma HLS pipeline
4  #pragma HLS dependency inter false var=p_prob, s_prob
5                                          //programmer manages dependency
6    schedule = edge_schedule[e];   // reads static schedule from DRAM
7    {bubble, mar, node_end, parent, prime, sub, weight,
                        parent_is_prime} = schedule; // decode schedule
8    edge_prob = p_prob[prime] * s_prob[sub] * weight;
9    if( bubble == false ){         //no-op if there is a bubble instr
10     node_prob = (mar == 1) ? node_prob + edge_prob :          //MAR
                               {(edge_prob > node_prob) ?
                                 edge_prob : node_prob};         //MPE
11     if(parent_is_prime == 1){ p_prob[parent] = node_prob;}  //store
12     else                     { s_prob[parent] = node_prob;}   //prob
13   }
14   if( node_end == true ){ node_prob = 0; }     //init for next node
15 }
```

Fig. 8. II reduction to 1 after applying the probability array separation technique (Section 4.2).

The first part of the solution consists of clustering the edges of a common parent. If we adjust the static schedule so that an edge that shares the same parent is processed immediately after one another, the updated probability value (the maximum value in MPE and the summation value in MAR) can be read from a temporary register. This can be observed in line 10 of Figure 7—a temporary register `node_prob` is read and written in the same line (the value is also written to the probability URAM in line 11). There is no RAW hazard because the addition to `node_prob` is serialized and can be done in a single cycle. When we process an edge that does not have the same parent as the previous edge, we can initialize `node_prob` using a flag from the static schedule (line 13). By adopting this technique, one probability array read is removed. This approach can naturally be combined with the depth-batched processing in Section 4.1 because the child of a common parent has the same depth in a tree.

The second part of the solution is the probability array separation. We exploit the fact that a node is either a prime node or a sub node, and it cannot be both. We separate the node probability storage for primes and subs. The revised HLS code is shown in Figure 8. One data is read from the prime node probability storage (`p_prob[]`), and one data is read from the sub node probability storage (`s_prob[]`). They are used for the computation in line 8. If the parent node is a prime, the output of each iteration is written to `p_prob[]` (line 11). It is a sub, the output is written to `s_prob[]` (line 12). With this code modification, `p_prob[]` and `s_prob[]` have one read and one write per iteration, and we can reduce the II to 1. This approach does not double the URAM consumption because there are approximately the same number of primes and subs, and the size of `p_prob[]` and `s_prob[]` are each about the half of `prob[]`.

5 PARALLELIZATION

The work in [37, 43] achieves a large operation-level parallelism by mapping the entire graph on the FPGA. But this approach has two limitations. First, the amount of available parallelism decreases when the graph size is small. Second, it is difficult to parallelize a large graph that cannot fit on the FPGA. The Mastermind dataset suffers from the second problem since it is composed of 42,558 nodes. Therefore, we need other types of parallelism to increase the throughput. In this section, we describe two solutions to overcome this problem.

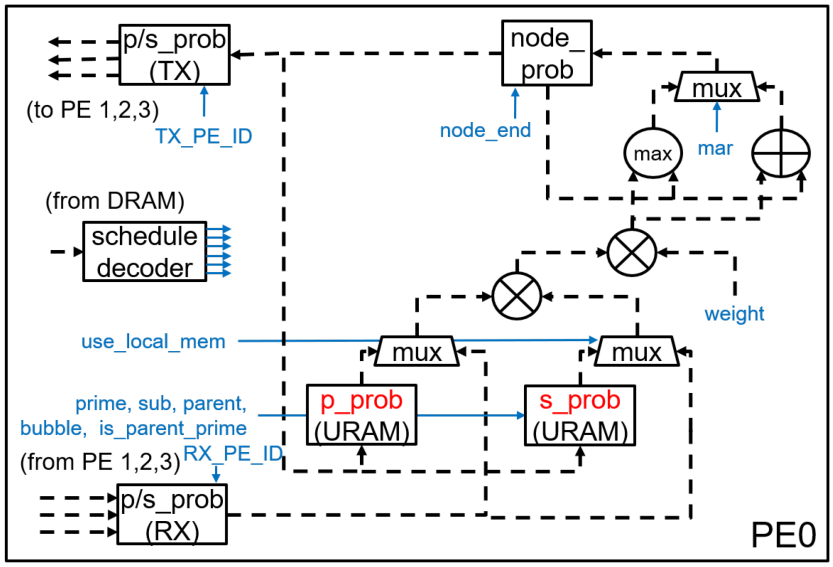


Fig. 9. PE0 architecture when the subtree-level parallel factor is four.

5.1 Subtree-level Parallelism

We increase the throughput of the PSDD accelerator by processing multiple subtrees in parallel. We split the prime and the sub probability memory (`p_prob` and `s_prob`) into multiple subtrees and attach OR/AND-gate operators to each memory. The architecture is illustrated in Figure 9. The address of the memory (prime, sub, parent) is read from the schedule decoder (the decoded control signals are colored in blue). The probability of the prime and sub nodes can be directly fed into the operators if the data exists in the local memory. If not, it will be read from the local memory of other PEs. Prime and sub node probability can be sent to other PEs as well. The data will be selected based on the schedule-decoded signal `use_local_mem`, and the target ID of the PEs are selected with signals `TX_PE_ID` and `RX_PE_ID` (these control signals were omitted in Figure 8 for simplicity). After we compute the AND gate (multiplication) with weights, we calculate a new temporary node probability `node_prob` with addition or max, depending on the type of the query (selected with the control signal `mar`). Register `node_prob` may be initialized by the signal `node_end` if the child nodes no longer share a common parent node (Section 4.2). Register `node_prob` will be written to `p_prob` or `s_prob` depending on the control signals `bubble` and `is_parent_prime`. They may be sent to other PEs if the result is read in other PEs.

Since each PE needs to read a separate schedule from DRAM, we attach a DRAM access module per PE. Alveo U250 has four DRAM channels, and thus we employed the subtree parallel factor of four.

Next, we need to determine how to assign the nodes to different PEs. Initially, we have equally partitioned the nodes in each level and assigned them to different PEs. However, we found that 68% of the instructions in the Mastermind's static schedule were either sending or receiving data from other PEs. Even though equal load balancing was achieved, the parent nodes in a different level were not guaranteed to be in the same PE.

We solved this problem with a graph coarsening technique. The new PE assignment process is shown in Algorithm 1. Given a PSDD graph, we first determine the tree depth of each node with topological sorting (line 3). Then we traverse in a bottom-up fashion (lines 4 and 5) and

ALGORITHM 1: PE assignment of nodes

```
1  Input: PSDD graph
2  Output: List of nodes assigned to each PE: PE_0, PE_1, ..., PE_S
3  Assign depth to all nodes
4  C_dMAX ← All nodes with the largest depth (dMAX)
5  for d = dMAX, ..., 1 do
6      C_{d-1} ← ∅
7      for each cluster C_d^i ∈ C_d do
8          coarsened ← false
9          size_limited ← false
10         for each cluster C_{d-1}^j ∈ C_{d-1} do
11             if Any nodes in Parent(C_d^i) exists in C_{d-1}^j then // share common parent nodes
12                 if |C_{d-1}^j ∪ C_d^i| < N/S then
13                     C_{d-1}^j ← C_{d-1}^j ∪ C_d^i
14                     coarsened ← true
15                 else // cannot be coarsened since it will exceed the cluster size limit
16                     size_limited ← true
17                 break
18         if coarsened == false then
19             if size_limited == false then
20                 Append {C_d^i ∪ Parent(C_d^i)} to C_{d-1}
21             else
22                 Append C_d^i to C_{d-1}
   // The below is a multi-way partitioning based on LPT scheduling
23 sort(C_0) // in a descending order of the cluster size
24 for C_0^0, C_0^1, ..., ∈ C_0 do
25     Insert all the nodes in C_0^i into one of PE_0, PE_1, ..., PE_S bins if the number of nodes in the bin is the
       smallest after insertion.
```

try to coarsen the cluster of nodes that share the same parent node (lines 6–22). For each cluster in-depth d (line 7), we look up the clusters in-depth $d-1$ for common parent nodes (lines 10 and 11). If found, the common parent node and the child nodes are grouped together (line 13). This process can naturally be combined with the common parent clustering method described in Section 4.2. The coarsening continues until the size of the cluster reaches the limit N/S (line 12), where N is the number of all nodes and S is the subtree level parallelism factor. If the size limit is reached, the cluster is appended to the next depth cluster vector without further coarsening (line 22). If a cluster can find no other clusters with a common parent node, the parent nodes are merged with the cluster and appended to the next depth cluster vector (line 20).

After obtaining a list of coarsened clusters for depth 0, we perform multi-way partitioning. Similar to the **largest processing time** (**LPT**) algorithm [20], we sort the clusters in a descending order of its size (line 23). The nodes in the sorted clusters are inserted into one of the S bins where the sum of nodes after insertion is the smallest (line 26). All nodes are assigned a PE after this process.

If the nodes with an edge connection are assigned different PEs, inter-PE communication of probability is needed. For the Mastermind dataset, we discovered that most of the communication

Table 1. Resource Consumption and Performance After Increasing Subtree Parallel Factor

Par.	Intercon.	LUT/FF/DSP/BRAM/URAM	CLK	GOPS
1	Full	2.3 K/4.4 K/3/3/16	300	0.86
2	Cross	4.3 K/8.3 K/6/9/16	300	1.7
4	bar	8.5 K/16 K/12/33/16	234	2.6
4	Chain+[21]	12 K/22 K/12/33/16	300	3.2

occurs near the top part of the tree using the proposed algorithm, and the proportion of inter-PE communication instruction is reduced to 2%. But we also noticed that the proportion of inter-PE communication worsens to 30% for certain datasets (more details in Section 6.2). It remains as a future work to further improve the node assignment strategy.

Table 1 shows the resource consumption for various configurations. The amount of computation resources (LUT and DSPs) grows almost proportionally as the subtree parallel factor increases. The URAM consumption, on the other hand, stays approximately the same because the number of nodes processed by each PE decreases as the subtree parallel factor increases. This can be confirmed in the table which reveals that the URAM consumption stays at 16 even though the parallel factor increases from 1 to 2 to 4. The BRAM consumption grows rapidly (3 to 9 to 33) with a larger subtree parallel factor because the number of connections increases quadratically with a full crossbar structure.

An important observation from Table 1 is the significant drop in the kernel clock frequency down to 234 MHz when the subtree parallel factor is 4. In Alveo U250, the four DRAM channels are located in a physically separated **Super Logic Region** (**SLR**) [41], and 16 (=4 × 4) inter-PE FIFOs' SLR crossings have a negative impact on the routing process.

To solve this problem, we changed the inter-PE communication architecture to a 1D chain (thus reducing the number of FIFOs crossing the SLR), and we added signal relay modules in the FIFOs using Autobridge [21]. The frequency is improved to 300 MHz as a result. Even though the inter-PE FIFOs are now partially shared among multiple PEs, the performance is not significantly degraded by the data congestion because of the small proportion of inter-PE communication in the Mastermind dataset. The cost of the improved frequency is the larger LUT consumption (from 8.5 K to 12 K). This is due to the inter-PE chain modules and the signal relay modules. We obtain a speedup of 3.7 (=3.2/0.86 GOPS) when using a subtree parallel factor of four.

5.2 Query-level Parallelism

The BRAM and the LUT consumption of the inter-PE interconnect for subtree-level parallelism increases rapidly as we add more PEs. Also, each PE requires a separate DRAM access module to fetch the static scheduling information.

In order to increase the parallelism even with a limited number of DRAM channels, we propose query-level parallelism. As the name suggests, we compute multiple queries in parallel. The static schedule from DRAM is shared among all different queries. The probability storage, AND/OR gates, max operations, and node_prob storage in Figure 9 are kept separate for each query (these variables are duplicated in the HLS code). Therefore, increasing the level of parallelism increases the LUT/FF/URAM usage of PEs, not the complexity of the inter-PE interconnect nor the DRAM channel usage.

Since the static schedule is shared among different queries, no modification is required for the schedule data. But we need to change the ordering of the literal values in the DRAM so that they can initialize the probability storage in parallel. The vector of input literal values for a query is

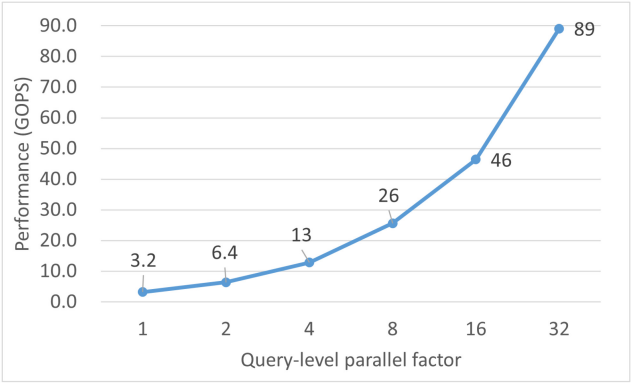


Fig. 10. Speedup after increasing query-level parallelism in the Mastermind dataset.

Table 2. Datasets used for Experiment

Dataset	FS-04	Mastermind	Students	Blockmap
# of nodes	52,789	42,558	6,921	3,548
# of leaf literal nodes	528	2328	687	1,218
# of edges	73,048	45,272	7,582	2,334
# of edges per node	1.4	1.1	1.2	1.0

originally placed next to the literal values of another query. After reading the literal values for multiple queries (that correspond to the query-level parallel factor) from the DRAM, the accelerator transposes the literal input data so that different queries' values for the same literal are placed adjacent to each other. This allows the probability storage to be initialized in a SIMD style (the initialization timing is the same and the value is different) at the beginning of the query computation.

Figure 10 presents the effect on the performance after increasing the query-level parallelism. It shows that the performance improves from 3.2 GOPS to 89 GOPS when we increase the query-level parallel factor from 1 to 32. The improvement is slightly sub-linear because it is more difficult to close the timing as the resource utilization is increased. The clock frequency dropped from 300 MHz to 273 MHz for the design with a 32 query-level parallel factor. The LUT/FF/URAM consumption of PEs and the BRAM consumption of the inter-PE chain increase approximately linearly with the query-level parallelism factor. The storage for transposing the literal values occupies about 16 K LUTs and 32 K FFs when the subtree-level parallel factor is 4 and the query-level parallel factor is 32 (the resource breakdown will be presented in Section 6.3).

6 EXPERIMENTAL RESULTS

6.1 Experimental Setup

For development, we used the Xilinx Vitis 2019.2 unified software platform [39]. The kernel was programmed in C++ and compiled with Vivado HLS 2019.2 [40]. The tests were run on-board, using the Xilinx Alveo U250 [41] platform. The probability is stored in 32b integers. The performance is measured in **Giga Operations per Seconds** (**GOPSs**).

In addition to the Mastermind dataset (which has been used for the optimization and parallelization process), we also tested our accelerator with three more datasets from [6]: FS-04 (friends and smokers), Students (and professors), and (random) Blockmap datasets (shown in Table 2). Each dataset is a grounded relational Bayesian network, which is a very challenging problem for

Table 3. Schedule Generation Time (in Seconds)

Dataset	FS-04	Mastermind	Students	Blockmap	AVG
Time	2.5	0.60	0.088	0.036	0.81

Table 4. Performance and Cumulative Speedup with Proposed Optimizations (Mastermind dataset)

	Base line	Proc. rate opt.	Subtree paral.	Query paral.
GOPS	0.041	0.86	3.2	89
Speedup	-	21X	3.7X	28X
Cum SpdUp	1.0X	21X	78X	2,200X

classical inference methods. Like Mastermind, these datasets exhibit a vast amount of local structure, and PSDD can exploit this characteristic and achieve exact, tractable inference. The datasets contain 3,548—52,789 nodes, and we cannot fit all the operators for the entire graph on the FPGA. The datasets have a sparse connection—having 1.0–1.4 edges per node (excluding leaf literal nodes). We also list the number of the leaf literal nodes and edges in the datasets.

6.2 Acceleration Results

The time required to generate the schedule for depth-batched processing (Section 4.1), common parent clustering (Section 4.2), and subtree-level parallelism (Section 5.1) is shown in Table 3. The scheduling can be performed in seconds due to its low complexity. Also, the scheduling only needs to be done once per dataset, and it can be reused across many different queries.

We have already analyzed the effect of increasing the subtree-level parallel factor and the query-level parallel factor in Sections 5.1 and 5.2, respectively. In this section, we present the cumulative effect of each optimization starting with the baseline HLS code in Figure 3. After applying the edge-centric processing with the static scheduling, the common parent clustering, and the probability array separation, the processing rate no longer suffers heavily from the long pipeline epilogue/prologue cycles (12 cycles) even though there are only a small number of child nodes (average:1.1). Also, we can achieve II of 1. The performance is improved by 21X (Table 4). The subtree-level parallelism of four improves the performance by 3.7X (refer to the explanation in Section 5.1), and the query-level parallelism of 32 leads to a speedup of 28X as observed in Figure 10. After applying all optimization steps, we achieved a cumulative speedup of 2,200X compared to the baseline code. The clock frequency of the final design is 273 MHz.

After applying all the proposed optimizations, we have measured the performance in various datasets. The result is presented in Table 5—the first row is calculated after considering the FPGA execution time only, and the second row is calculated after considering both the FPGA execution time and the PCIE transfer time (of the graph processing static schedule and the literal values of each query). Compared to other works that assume the entire graph can fit on the FPGA, our design architecture is scalable and retains a relatively high performance for datasets of various sizes (see Section 7 for quantitative comparison). This is because the proposed design reads the graph structure information from the DRAM and the performance does not depend on the size of the graph that is mapped to the FPGA.

The maximum FPGA-only performance is 89 GOPS (Mastermind), and the averaged FPGA-only performance is 59 GOPS. But there is some performance variance among the datasets. For the Blockmap dataset, the low performance is due to the high proportion (34%) of leaf literal nodes

Table 5. Performance in Various Datasets (in GOPS)

Dataset	FS-04	Mastermind	Students	Blockmap	AVG
Performance (FPGA only)	62	89	56	30	59
Performance (FPGA+PCIE)	61	77	45	17	50

Table 6. Performance Comparison Between 32-Thread Multicore CPU and Proposed FPGA Implementation (in GOPS)

Dataset	FS-04	Mastermind	Students	Blockmap	AVG
CPU	1.9	2.2	4.2	3.3	2.9
FPGA	62	89	56	30	59
Speedup	32X	41X	13X	9X	20X

(Table 2)—much of the execution time is spent on fetching and transposing the value of literal nodes rather than processing an edge. Apart from this factor, the performance difference is mostly due to the effectiveness of the subtree parallelization step. The amount of inter-PE communication (Section 5.1) is 2% for Mastermind, 23% for Students, 27% for Blockmap, and 30% for FS-04, and the data congestion due to the simple 1-D chain architecture is further degrading the performance. Moreover, the coarsening step explained in Section 5.1 introduces an unbalanced workload, which accounts for the rest of the performance difference. It remains as a future work to reduce these overheads without severely complicating the inter-PE communication architecture.

The performance drops to an average of 50 GOPS if we consider the PCIE transfer time in addition to the FPGA execution time. The static schedule is fetched from the DRAM and sent to the FPGA for each batch of queries, but it only needs to be transferred once through the PCIE because the same schedule is reused in the DRAM. The literal values for each query, on the other hand, are transferred through both the PCIE and the DRAM only once—they are not reused in the DRAM. This leads to a larger performance drop for the Blockmap dataset (30→17) compared to FS-04 dataset (62→61)—the Blockmap dataset has a larger proportion of literal nodes compared to the FS-04 dataset (34% vs. 1%, Table 2). This causes more time to be spent on transferring the literal values through the PCIE.

Table 6 compares the performance between a multicore CPU implementation and the proposed FPGA implementation. We use a two-socket Intel Xeon Gold 6244 server class node,[4] and we parallelize the loop that processes the queries with 32 OpenMP threads (the parallel factor is the same as the FPGA query-level parallelism). To avoid contention, `prob[]` has been separated for each OpenMP thread (the memory allocation time is excluded from the execution time). The CPU implementation has been optimized with O3 flag. The experimental result shows that the average performance is 2.9 GOPS—so even though we use 2X CPUs, the performance is 20X slower than the proposed FPGA implementation. The reason is mainly related to how fast the data can be supplied to the computation units. The proposed FPGA static scheduling provides operands to the OR/AND-gate in almost every cycle. In a CPU, this is not guaranteed since the data for thousands of nodes and multiple threads cannot fit into the L1/L2 cache (notice that, unlike the FPGA performance, the CPU performance is generally higher in smaller datasets). Also, the superior performance is attributed to the customized computation/memory pipeline.

We will present a quantitative comparison with related works in Section 7.

[4]Xeon 6244 and Alveo U250 are based on a comparable technology (14 nm vs. 16 nm) and have the same release year (2019).

Table 7. Post-PnR Resource Consumption on Alveo U250

	LUT	FF	DSP	BRAM	URAM
DRAM Access	40 K	50 K	12	21	0
PEs	42 K	66 K	256	0	256
Inter-PE Chain	33 K	76 K	0	684	0
Total Res. Cons.	115 K	193 K	268	705	256
Util. Ratio	7.7%	6.2%	2.2%	31%	20%

6.3 Accuracy and Resource Consumption

As mentioned in the experimental result, the bitwidth of the probability variables is 32b. After comparing with the MPE and the MAR results of the floating-point-based CPU implementation, we found that the **signal-to-quantization-noise ratio** (**SQNR**) is 66 dB–97 dB, with an average of 85 dB. We can conclude that there is almost no accuracy loss in employing the fixed-point arithmetic because this SQNR value exceeds the typically expected fixed-point representation accuracy (around 30 dB or more).

Table 7 details the post-PnR resource consumption of the PSDD acceleration kernel. We exclude the resource used in the static region and the DRAM controller. The table reveals that the PEs consume most of the DSPs (96%) for the computation. The PEs also require a large amount of URAM (256) to store the node probability. The inter-PE chain (introduced in Section 5.1 for the subtree-level parallelism) consumes several BRAMs (684) because of the FIFO buffers among the PEs. It also consumes a large portion of LUTs to implement its control circuits. About half of the LUTs/FFs in the DRAM access modules are used for transposing the literal values for the query-level parallelism (Section 5.2).

Among all the FPGA resources, the one with the highest utilization is BRAM. But even the highest utilization ratio is relatively small (31%). This is because we wanted to ease the PnR process by keeping the consumption under 50% for all resources. Thus the subtree-level parallelism and the query-level parallelism were limited to 4 and 32, respectively.

7 RELATED WORKS

There are several recent articles that accelerate graph applications (e.g., sparse matrix-vector multiplication) on an FPGA. HitGraph [44] is a high-performance edge-centric graph accelerator with several performance optimization techniques such as node buffer and data layout optimization. ThunderGP [7] is an HLS-based graph processing framework that automatically builds FPGA accelerators based on its high-level APIs. It supports several efficient memory access patterns such as scatter/gather, coalescing, and prefetching. However, these articles target general graph applications and cannot exploit unique characteristics that exist in graph models such as BN, SPN, or PSDD.

In the remainder of this section, we will review related FPGA acceleration works for the BN and the SPN.

The **Bayesian Computing Machine** (**BCM**) has been presented in [25]. It supports the sum-product algorithm and the max-sum algorithm. Their hardware is composed of a network of processors and memory connected through a switching crossbar. They propose an optimal scheduling of computation and memory units to minimize the execution time. This work has been implemented on the Berkeley Emulation Engine FPGA platform [1].

The work in [43] uses a high-level synthesis tool chain similar to our work. They accelerate the Arithmetic Circuit on the Xilinx Zynq embedded platform. They provide an option to reconfigure the network parameters with the data fetched through the AXI bus. The parallelism is achieved by

Table 8. Graph Size, Platform, Resource Consumption, and Performance Comparison with Related Works Implemented on FPGA

	HitGraph [44]	ThunderGP [7]	BCM [25]	[37]	[24]	Our work
Model	General	General	BN	SPN	SPN	PSDD
# of nodes	0.7 M–41 M	45K–21 M	N/A	11–287	63–2,699	3,548–52,789
Graph Info	DRAM	DRAM	DRAM	On-chip	N/A	DRAM
Num Repr.	N/A	32b int	float	double	[38]	32b int
Platform	VU5P	VCU1525	Virtex-5	VC709	Ultra96	U250
CLK (MHz)	200	250	N/A	200	205–350	273
LUT	380 K	1,100 K	N/A	78 K–346 K	27 K–59 K	115 K
DSP	62	150	N/A	60–1.5 K	54–342	268
GOPS	N/A	N/A	20	2.1–57	1.9–13	30–89
GTEPS	1.0–3.4	1.7–6.4	N/A	N/A	N/A	10–30

compiling the entire network onto the FPGA, but such an approach limits the network size to no more than 511 nodes. Also, their accelerator achieves a performance of only about 0.01 GFLOPS, possibly due to the data transfer overhead.

The work in [37] accelerates the SPN inference problem on a Virtex-7 FPGA. They map the SPN tree to a hardware datapath composed of pipelined functional units and shift registers. Similar to [43], the entire SPN tree is implemented on-chip. In a separate work [24], they present an architecture where the operators are time-shared—which makes it possible to process larger graphs. The number representation in a SPN graph can be efficiently optimized based on the histogram of the variables—readers are referred to [38] for an automated method that finds the best number representation.

The work in [34] converts a PSDD graph to a SPN graph by replacing AND with products and OR with sums. The resulting SPN graph is accelerated with a tree of customized processors, each with private registers. Their simulation result reports a peak performance of 11.6 operations/cycle.

Compared to these works, our work concentrates on developing an HLS-friendly optimization method to improve the processing rate and parallelism in PSDD graphs. A quantitative comparison is shown in Table 8. Since each work uses a different graph structure, number representation, and dataset, it is difficult to make a direct comparison. Instead, the performance is provided in either GOPS or **Giga Edges Traversed Per Second** (**GTEPS**). The table also presents the graph size, resource consumption, platform, and performance. The performance of [37] is estimated by multiplying the number of add/mul operations by the reported throughput (FPGA time only).

Our work can outperform accelerators that do not assume a particular graph model (e.g., [44] or [7]) because our processing engine can exploit PSDD-specific characteristics—such as each node having a small number of child nodes or being connected to prime and sub nodes. Our work also outperforms related BN [25] and SPN work [24] by 3.0X and 9.0X on average, respectively. These works read the graph information from memory similar to our work. It is unclear if our work significantly outperforms [37]—it is possible that the higher frequency (273 MHz vs. 200 MHz) may have been achieved due to more advanced FPGA technology (Alveo U250 vs. VC709). This may have led to the better performance (59 GOPS vs. 31 GOPS, on average). It is also difficult to make a direct comparison on the LUT/DSP consumption, because our work is operating on 32b integers. But it is still worth noting that [37] can only process relatively smaller graphs with the entire SPN tree being mapped to the FPGA operators—whereas our work can process larger graphs efficiently with the proposed static scheduling method. Also, our work has relatively less variance on the performance with a different graph size because of the processing rate optimizations in Section 4.

8 CONCLUSION

We presented an HLS-friendly accelerator design for PSDD. We found that changing the processing scheme from node-centric to edge-centric helps maintain a high processing rate even if there are only a few number of children per parent node. This led to a dependency problem, which was solved with static scheduling. The throughput of the PSDD pipeline was improved to II of 1 with common parent clustering and array separation techniques. We also proposed subtree-level and query-level parallelization methods that can be used to improve the computation speed of a large tree that cannot fit on an FPGA. Experimental results show that the optimizations improve the performance of the baseline implementation by 2,200X. The proposed architecture has a speedup of 20X over CPU implementation, and it outperforms the BN and SPN FPGA acceleration work that stores the graph information in the memory by 3.0X–9.0X. As a future work, we plan to further reduce the performance variance among different datasets with load balancing improvement and inter-PE communication reduction. This PSDD acceleration project has been open-sourced at https://github.com/carlossantillana/psdd/tree/alveo250.

ACKNOWLEDGMENTS

We thank Yuze Chi, Vidushi Dadu, Licheng Guo, Jason Lau, Michael Lo, Tony Nowatzki, and Yizhou Sun for the discussion and the help with the experiments. We also thank Marci Baun for proofreading this article.

REFERENCES

[1] Berkeley. 2008. *BEE3 (Berkeley Emulation Engine)*. Retrieved 24 July, 2022 from https://www.microsoft.com/en-us/research/project/bee3/.

[2] Simone Bova. 2016. SDDs are exponentially more succinct than OBDDs. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 929–935.

[3] Ruizhe Cai, Ao Ren, Ning Liu, Caiwen Ding, Luhao Wang, Xuehai Qian, Massoud Pedram, and Yanzhi Wang. 2018. VIBNN: Hardware acceleration of Bayesian neural networks. *ACM SIGPLAN Notices* 53, 2 (2018), 476–488.

[4] Hei Chan and Adnan Darwiche. 2006. On the robustness of most probable explanations. In *Proceedings of the 22nd Conference in Uncertainty in Artificial Intelligence*.

[5] Mark Chavira and Adnan Darwiche. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence* 172, 6 (2008), 772–799.

[6] Mark Chavira, Adnan Darwiche, and Manfred Jaeger. 2006. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning* 42, 1–2 (2006), 4–20.

[7] X. Chen, H. Tan, Y. Chen, B. He, W. Wong, and D. Chen. 2021. ThunderGP: HLS-based graph processing framework on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80.

[8] Arthur Choi, Guy Van den Broeck, and Adnan Darwiche. 2015. Tractable learning for structured probability spaces: A case study in learning preference distributions. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*. AAAI Press, 2861–2868.

[9] Arthur Choi, Yujia Shen, and Adnan Darwiche. 2017. Tractability in structured probability spaces. In *Proceedings of the Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*. 3477–3485.

[10] Arthur Choi, Nazgol Tavabi, and Adnan Darwiche. 2016. Structured features in naive bayes classification. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. AAAI Press, 3233–3240.

[11] Young-kyu Choi and Jason Cong. 2018. HLS-based optimization and design space exploration for applications with variable loop bounds. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 1–8.

[12] J. Cong, W. Jiang, B. Liu, and Y. Zou. 2011. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Transactions on Design Automation of Electronic Systems* 16, 2 (2011), 1–25.

[13] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[14] Adnan Darwiche. 2003. A differential approach to inference in Bayesian networks. *Journal of the ACM* 50, 3 (2003), 280–305.

[15] Adnan Darwiche. 2009. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press.

[16] Adnan Darwiche. 2011. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*. 819–826.
[17] Adnan Darwiche. 2020. Three modern roles for logic in AI. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 229–243.
[18] Adnan Darwiche. 2022. Tractable boolean and arithmetic circuits. In *Proceedings of the Neuro-symbolic Artificial Intelligence: The State of the Art*, Pascal Hitzler and Md Kamruzzaman Sarker (Eds.), Vol. 342, Frontiers in Artificial Intelligence and Applications. IOS Press, Chapter 6.
[19] Johannes Geist, Kristin Y. Rozier, and Johann Schumann. 2014. Runtime observer pairs and bayesian network reasoners on-board FPGAs: Flight-certifiable system health management for embedded systems. In *Proceedings of the International Conference on Runtime Verification*. Springer, 215–230.
[20] R. L. Graham. 1966. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal* 45, 9 (1966), 1563–1581.
[21] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong. 2021. AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.
[22] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. 2014. Probabilistic sentential decision diagrams. In *Proceedings of the 14th Conference on the Principles of Knowledge Representation and Reasoning*. AAAI Press.
[23] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press.
[24] Hanna Kruppe, Lukas Sommer, Lukas Weber, Julian Oppermann, Cristian Axenie, and Andreas Koch. 2021. *Efficient Operator Sharing Modulo Scheduling for Sum-Product Network Inference on FPGAs*. Retrieved 24 July, 2022 from https://www.esa.informatik.tu-darmstadt.de/assets/publications/materials/2021/2021_SAMOS_HK.pdf.
[25] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. 2010. High-throughput Bayesian computing machine with reconfigurable hardware. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 73–82.
[26] Kevin P. Murphy. 2012. *Machine Learning - A Probabilistic Perspective*. MIT Press.
[27] Xie Pan and Yu Jinsong. 2017. Diagnosis via arithmetic circuit compilation of Bayesian network and calculation on FPGA. In *Proceedings of the 13th IEEE International Conference on Electronic Measurement & Instruments*. 35–41.
[28] Judea Pearl. 1989. *Probabilistic Reasoning in Intelligent Systems - Networks of Plausible Inference*. Morgan Kaufmann.
[29] Knot Pipatsrisawat and Adnan Darwiche. 2008. New compilation languages based on structured decomposability. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*. AAAI Press, 517–522.
[30] Hoifung Poon and Pedro M. Domingos. 2011. Sum-product networks: A new deep architecture. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence*. AUAI Press, 337–346.
[31] Tahrima Rahman, Prasanna Kothalkar, and Vibhav Gogate. 2014. Cutset networks: A simple, tractable, and scalable approach for improving the accuracy of chow-liu trees. In *Proceedings of the Machine Learning and Knowledge Discovery in Databases - European Conference*. Springer, 630–645.
[32] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 472–488.
[33] N. Shah, L. Olascoaga, W. Meert, and M. Verhelst. 2020. Acceleration of probabilistic reasoning through custom processor architecture. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. 322–325.
[34] Nimish Shah, Laura I. Galindez Olascoaga, Wannes Meert, and Marian Verhelst. 2020. Acceleration of probabilistic reasoning through custom processor architecture. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 322–325.
[35] Yujia Shen, Arthur Choi, and Adnan Darwiche. 2016. Tractable operations for arithmetic circuits of probabilistic models. In *Proceedings of the Advances in Neural Information Processing Systems*. 3936–3944.
[36] Yujia Shen, Arthur Choi, and Adnan Darwiche. 2018. Conditional PSDDs: Modeling and learning with modular knowledge. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*. AAAI Press, 6433–6442.
[37] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. 2018. Automatic mapping of the sum-product network inference problem to FPGA-based accelerators. In *Proceedings of the IEEE 36th International Conference on Computer Design*. 350–357.
[38] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. 2020. Comparison of arithmetic number formats for inference in sum-product networks on FPGAs. In *Proceedings of the IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines*. 75–83.
[39] Xilinx. 2020. *Vitis Unified Software Platform*. Retrieved 24 July, 2022 from https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.
[40] Xilinx. 2020. *Vivado High-Level Synthesis (UG902)*. Retrieved 24 July, 2022 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug902-vivado-high-level-synthesis.pdf.

[41] Xilinx. 2021. *Alveo U250 Data Center Accelerator Card*. Retrieved 24 July, 2022 from https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.
[42] Xilinx. 2021. *UltraScale Architecture Memory Resources*. Retrieved 24 July, 2022 from https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf.
[43] S. Zermani, C. Dezan, H. Chenini, J. Diguet, and R. Euler. 2015. FPGA implementation of Bayesian network inference for an embedded diagnosis. In *Proceedings of the 2015 IEEE Conference on Prognostics and Health Management*. 1–10.
[44] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu. 2019. Hitgraph: High-throughput graph processing framework on FPGA. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2249–2264.
[45] Stephanie Zierke and Jason D. Bakos. 2010. FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods. *BMC Bioinformatics* 11, 1 (2010), 1–12.

Received 20 December 2021; revised 24 July 2022; accepted 19 August 2022