Efficient Operator Sharing Modulo Scheduling for Sum-Product Network Inference on FPGAs*

Hanna Kruppe[1 [0000−0001−7784−1164]], Lukas Sommer[1 [0000−0003−1918−3911]], Lukas Weber[1 [0000−0003−0621−1151]], Julian Oppermann[1 [0000−0002−8073−720X]], Cristian Axenie[2], and Andreas Koch[1 [0000−0002−1164−3082]]

[1] Embedded Systems and Applications Group
Technical University Darmstadt, Germany
{kruppe, sommer, weber, oppermann, koch}@esa.tu-darmstadt.de
[2] Intelligent Cloud Technologies Laboratory
Huawei Munich Research Center, Germany
cristian.axenie@huawei.com

Abstract. Probabilistic models are receiving increasing attention as a complementary alternative to more widespread machine learning approaches such as neural networks. One particularly interesting class of models, so-called *Sum-Product Networks* (SPNs), combine the expressiveness of probabilistic models with tractable inference, making them an interesting candidate for use in real-world applications.
Previously, inference in SPNs has successfully been accelerated by fully pipelined FPGA-based hardware. However, with these approaches, the maximum size of the SPN for FPGA acceleration has effectively been limited by the fully spatial mapping of arithmetic operations into hardware and the number of available resources in the FPGA.
In this work, we present an extended and specialized modulo scheduling algorithm based on Integer Linear Programming (ILP) for time-multiplexed sharing of hardware arithmetic operators in the SPN inference accelerator. In addition and in order to scale the scheduling to large SPN graphs, we combine the scheduling algorithm with a graph-partitioning heuristic, exploiting the graph structure of SPNs.
The combination of heuristic graph partitioning and ILP-based scheduling allows generating pipelined accelerators with the best possible initiation interval, while limiting the resource utilization to pre-set bounds. The evaluation discusses the effect different parameters have on convergence time and solution quality. A performance comparison shows that the FPGA improves the inference throughput over a comparable CPU- and GPU platform by a factor (geo.-mean) of 4.4x and 1.7x, respectively.

Keywords: FPGA · Machine Learning · Probabilistic Model · Sum-Product Network · Modulo Scheduling · Graph Partitioning

* The authors would like to thank Xilinx Inc. for supporting their work by donations of hard- and software. Calculations for this research were conducted on the Lichtenberg high performance computer of TU Darmstadt. This research was partially funded by the German Federal Ministry for Education and Research (BMBF) with the funding ID ZN 01|S17050.

1 Introduction

Probabilistic models are receiving increasing attention from both academia and industry, as a complementary alternative to more widespread machine learning approaches such as (deep) neural networks (NNs). Probabilistic models can handle the uncertainty found in real-world scenarios better [17] and are also, in contrast to NNs, able to express uncertainty over their output.

While many probabilistic models quickly become intractable for larger use cases, so-called *Sum-Product Networks* (SPNs) provide efficient inference for a wide range of probabilistic queries in different real-world use cases. Similar to neural networks, for which both accelerated inference and training have been implemented on reconfigurable architectures, SPNs lend themselves to accelerated inference on FPGAs [11, 19]. Key to the efficient computation of probabilistic queries in prior work was the pipelining of batches of queries. This task is further complicated by the fact that the probability values computed in the SPN require expensive floating-point arithmetic and in general cannot be quantized to integer values as it is done for neural network inference. So, despite successful efforts to realize the necessary probabilistic arithmetic efficiently with specialized hardware operators [20,23], prior approaches are constrained by the fully spatial mapping of operations and the available FPGA resources, effectively limiting the maximum size of the SPN that can be mapped to the physical resources on the target FPGA. A possible solution to overcome this limitation is to map multiple *operations* to the same hardware arithmetic *operator*, so that operators are time-shared. In order to retain as much performance as possible, this time-sharing of operators needs to be combined with efficient pipelining, requiring a resource-aware modulo scheduler [12].

Our main contribution is a scheduling algorithm specialized for the automatic mapping of SPNs to a pipelined FPGA accelerator. Our approach extends an existing Integer Linear Programming (ILP) formulation [21] to also optimize the size of the multiplexers used to realize the time-sharing of operators, a crucial factor for the operating frequency of the whole accelerator. Beyond that and in order to be able to handle large SPN graphs during scheduling, a divide-and-conquer heuristic leveraging the special graph structure of SPNs is presented.

2 Background

SPNs [15,17] are a relatively young class of probabilistic models. Similar to other probabilistic graphical models (PGM), SPNs are able to efficiently handle real-world uncertainties, such as missing feature values, and express uncertainty over their output. They are used in several domains [17] including, but not limited to, image classification and reconstruction, image segmentation, robotics, and natural language processing.

Sum-Product Networks capture the joint probability distribution over a number of variables as a directed acyclic graph. As shown in the example in Fig. 1, the graph consists of three different types of nodes, namely weighted sum-nodes

(red), product nodes (green) and nodes representing univariate distributions (orange), where the latter can only occur as leaf nodes.

The graph structure of Sum-Product Networks, including the parameters such as weights and distribution parameters, can either be hand-crafted, completely learned from data (e.g., [10]) or can be generated and refined through learning of parameters (e.g., [14]).

Semantically, the product nodes in the graph correspond to factorizations of independent variables. As variables in a joint probability distribution are not independent in general, the weighted sum nodes come into play. They represent mixtures of distributions and, through clustering, expose independencies for factorization. If, after repeated mixture and factorization, only a single variable remains, the univariate distributions of these variables are captured by the leaf nodes. In this work, based on the approach for Mixed Sum-Product Networks by Molina et al. [10], univariate distributions of discrete variables are represented as histograms.

In contrast to many other probabilistic graphical models, inference in Sum-Product Networks is tractable, even for large graphs with many variables [13]. Enabled by the graph structure capturing the joint probability, a wide range of probabilistic queries, including conditional probability and most-probable explanation (MPE), can be computed efficiently by evaluating the SPN graph bottom-up (starting at the univariate distributions at the leaf nodes) one or multiple times (linear w.r.t. to the graph size). This work focuses on the efficient evaluation of a batch of queries and generation of pipelined FPGA accelerators under resource constraints.

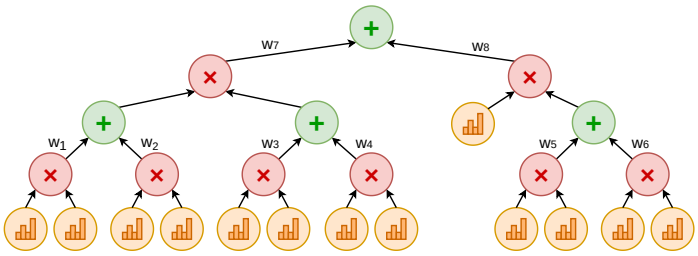


Fig. 1. Example of a Sum-Product Network graph.

3 Modulo Scheduling of SPN Inference

The core of our work is a resource-aware modulo scheduler tailored for SPN inference computations, which are described as acyclic data-flow graphs (DFGs). Operations in these data flow graphs include additions, multiplications, histogram evaluations, and constant weights. Among the operators realizing these operations, only adders and multipliers are subject to operator sharing. Other oper-

ations require only very few resources to realize in hardware and there are few opportunities to share these operators.

The throughput of a shared, modulo-scheduled datapath is chiefly determined by the initiation interval (II), the duration between starting successive overlapped computations in the datapath. As recurrences in the DFG significantly constrain the achievable II, the fact that we only need to support acyclic DFGs simplifies some typical challenges of modulo scheduling: without recurrences, the resource-constrained minimum II [16] is not just a lower bound on feasible IIs, but always equals the minimal feasible II. This allows us to determine the smallest II and smallest number of operators for a given hardware resource budget and SPN up-front before scheduling begins, rather than repeatedly attempting scheduling with different candidate IIs, as in most other applications of modulo scheduling.

To illustrate why the resource-constrained minimum II is always feasible, consider a variant of ASAP scheduling that starts operations once all their inputs are ready, but delays these start times as necessary to avoid over-subscription of operators. The resulting schedule will most likely be sub-optimal, but without recurrences that would impose additional *upper* limits on the start times of operations, such a schedule will always exist.

Based on similar considerations, we developed a divide-and-conquer heuristic for scheduling and binding, detailed in Section 3.2. Once II and available operators have been determined, this heuristic partitions the DFG and the available operators to produce a set of smaller scheduling and binding problems, whose solutions can be combined into a solution for the whole problem.

These sub-problems are then translated to ILP instances with an objective that attempts to reduce multiplexing overhead, detailed in Section 3.1. The bindings – the mapping of each DFG operation to a specific physical operator – heavily influence the amount of multiplexing necessary to realize the sharing of the operators, and this multiplexing can, in turn, limit the maximum operating frequency of the accelerator. As schedule and bindings influence and constrain each other, we consider them together in a joint optimization problem, rather than computing one before the other. After schedules and bindings for each sub-problem have been found by an off-the-shelf ILP solver, the results are combined into overall schedule and bindings by the heuristic component.

The heuristic combination of solutions for sub-graphs obtained by expensive, high-quality scheduling algorithms has been proposed before [5,6]. In our context, it provides a simple way to trade scheduling effort for solution quality, and offers a practical way of optimizing for a different objective than usual (reducing multiplexing overhead) without having to develop new heuristics specifically for this purpose.

3.1 ILP Extension for Multiplexer Reduction

We extended an ILP formulation of modulo scheduling and binding proposed by Šůcha and Hanzálek [21]. Out of the several variants presented there, we use the formulation for general processing time and multiple operator types (see

sections 4 and 5 there). This formulation prohibits over-subscription of shared operators by encoding the bindings in binary decision variables $\hat{z}_{iv}$ which are constrained such that $\hat{z}_{iv} = 1$ if and only if the operation identified by i should be bound to the v^{th} suitable operator (assuming some arbitrary but consistent numbering of the operator instances). We reuse these decision variables to also model connections between operators, as a proxy for multiplexing overhead.

Formally, each operation $i \in O$ is associated with a type of operator q such as adders, multipliers, histograms, and constant weights. For those operator types subject to sharing (in our case, adders and multipliers), there is a limited number m_q of operator instances, while the other types are *unlimited* – they are instantiated once per operation requiring them.

The baseline ILP formulation as presented by Šůcha and Hanzálek assumes all operator types are subject to sharing. It is simple to adapt the formulation to support unlimited operators: we can just omit decision variables and constraints related to bindings of operations implemented by unlimited operators[3] and leave only start time constraints in place. Due to space limitations, we do not show the ILP formulation here with these minor modifications applied.

The multiplexer at input port p of a shared operator v needs to select among all the physical locations in the datapath which produce the p^{th} input operand to any of the operations bound to v. In our accelerator's datapath, these values can be sourced from the output ports of other operators – whether they are themselves shared or not – as well as from shift registers inserted to buffer intermediate results for several cycles between being produced and consumed. Modeling the latter in the ILP formulation requires significant additional complexity: Sittel et al. [18] measured the register area by the maximum lifetime of intermediate results that can share registers, while multiplexer width is determined by the number of *distinct* lifetimes among intermediate results that could share a connection, which is far more difficult to linearize. This extra complexity is likely not justified in our context, as we combine the ILP formulation with a heuristic and thus will not obtain globally optimal solutions in any case.

Instead, we model only the presence or absence of connections between operator output ports and the input ports of shared operators. These connections are induced by the data flow edges $(i \to j) \in E$ and the bindings, encoded in the ILP by binary decision variables $\hat{z}_{iv}$ and $\hat{z}_{jv}$. The shared operators are identified by their type q and an index v from 1 to m_q. We also need to distinguish the different input ports p of the operators (typically, the operators are binary and thus $p \in \{1, 2\}$). Thus, for all q', v', p identifying a shared operator input port, there are binary variables $c^r_{q'v'p}$ for r ranging over the possible sources of a connection. These sources are the shared operator instances (q, v) as well as the operations $i \in O$ which are unlimited, i.e., *not* subject to operator sharing. Each such variable should be 1 if the result of operator r needs to be connected to port p of the shared operator (q', v').

Note that a single connection suffices for multiple edges $(i_1 \to j_1), \ldots, (i_n \to j_n) \in E$ if all the $i_1 \ldots i_n$ are mapped to the same shared operator, all the

[3] Specifically, variables $\hat{x}_{ij}, \hat{y}_{ij}, \hat{z}_{iv}$ and all constraints mentioning them.

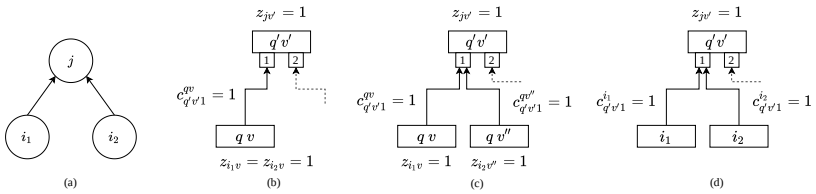


Fig. 2. Dataflow graph (a) and the impact of binding two operations to the same (b) or different (c) shared operator. When operations are not subject to operator sharing (d), multiple connections are necessary regardless of bindings.

$j_1 \dots j_n$ are mapped to the same shared operator, and each value is routed to the same input port of that shared operator. This is shown in Fig. 2 (b), while (c) shows different bindings that prevent sharing. When the sources of the data flow edges are not subject to operator sharing (d), separate connections are always required.

To encode these considerations into the ILP, we add constraints for every data flow edge $(i \to j) \in E$ whose destination j is subject to operator sharing, as those edges are the cause of the connections we model. Let q be the operator type of i and q' of j. When i is also subject to operator sharing, then it is bound to some shared operator (q, v) which will be connected to the input port p of the operator (q', v') which j is bound to. Hence, we add the following set of constraints:

$$c^{qv}_{q'v'p} \geq \hat{z}_{iv} + \hat{z}_{jv'} - 1 \qquad \begin{matrix} \forall v = 1, \dots, m_q \\ \forall v' = 1, \dots, m_{q'} \end{matrix} \tag{1}$$

Otherwise, if j requires an unlimited operator type, we simply have a connection from i to whatever operator (q', v') the operation j is bound to. In that case, we add the following set of constraints:

$$c^{i}_{q'v'p} \geq \hat{z}_{jv'} \qquad \forall v' = 1, \dots, m_{q'} \tag{2}$$

Constraints (1) and (2) only ensure that a connection indicator is set to 1 if the corresponding connection is required, but not the inverse implication. This modeling is correct within our formulation: we only use the values of these decision variables for the objective function (unlike other decision variables, which yield the schedule and bindings), and objective functions suitable for our purpose (reducing multiplexers or connection density) cannot be improved by setting more $c^{r}_{q'v'p}$ to 1 than necessary.

For the objective function, we first and foremost minimize the total connections between operators:

$$\min \sum_{r,q',v',p} c^{r}_{q'v'p} \tag{3}$$

This objective was proposed by Cong and Xu [4] in the context of choosing operator and register bindings for an already-determined schedule. As they noted, this objective is only an approximation of the real (non-linear) hardware cost, but minimizing it correlates with minimizing the number of multiplexer inputs, so it is a reasonable way to address the need for a linear objective function. They could only evaluate it on relatively small examples, but found it to be effective at least in those cases.

Objective (3) is combined with the classical sum-of-start-times objective (as in [21]) in a strictly hierarchical multi-objective optimization problem. In our throughput-oriented accelerator, the schedule length – the overall latency from inputs to final result of a single computation in the shared datapath – only has a small effect on the number and size of the aforementioned shift registers buffering intermediate results. The hardware resource cost of these registers is negligible, while large multiplexers can negatively affect the maximum frequency, so we prioritize multiplexer reduction over schedule length reduction.

Overall, our proposed ILP formulation consists of these two objectives and the constraints of the formulation by Šůcha and Hanzálek [21] – not repeated here due to the page limit – plus our constraints (1) and (2).

3.2 Divide-and-Conquer Heuristic

In this section, we present an algorithm for decomposing a modulo scheduling and binding problem on an acyclic DFG into smaller sub-problems whose solutions can be combined into a solution for the original problem. As presented here, the algorithm works for any acyclic DFG and any partitioning, although our implementation (Section 4) and evaluation is limited to DFGs that are trees, since most SPN learning algorithms only produce trees.

For now, assume some arbitrary partitioning of the DFG is given. We first partition the modulo reservation table (MRT) [8] – a data structure organizing the operations by the operator they are bound to and the time step modulo II in which they are scheduled – to match the DFG partitioning. Each available time step (modulo the II) on each available operator is exclusively assigned to one of the DFG partitions: only operations from that part of the DFG will be permitted to use that operator in that time step. By assigning each partition at least as many operator time slices as there are operations requiring such an operator in the partition, we ensure that each of the sub-problems is feasible in isolation. In addition, the exclusive assignment avoids conflicting bindings between the solutions of each sub-problem: no two operations from different partitions can use the same operator at the same time.

Fig. 3 shows such a partitioning of DFG and MRT, along with a solution for each partition. Note that the two available adders are each fully assigned to one or the other partition, while the multiplier is split up: node C can only be scheduled in even cycles while node D can only be scheduled in odd cycles. Although the schedules are correct with respect to each partition, the data flow edge C $\rightarrow$ E was ignored: the result of C is only available by cycle 10 (start time six plus latency of four), while E was scheduled to start in cycle five.

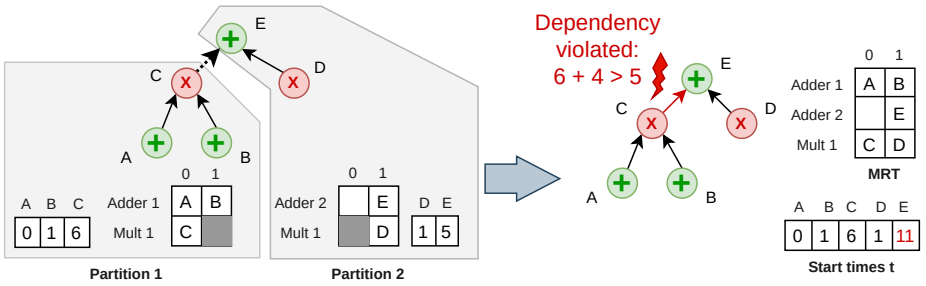


Fig. 3. Example of scheduling with graph partitioning (II=2, all operators have a latency of four cycles). Left: partitioning and partial solutions (MRT, operation start times). Right: overall solution after merging the partial solutions, including the adaptation of E's schedule to preserve dependencies.

To repair such inconsistencies arising from data flow between partitions, we delay the start time of affected operations when we combine partial schedules. Specifically, we inspect all edges between the different partitions, and if the destination operation starts before the source operation finishes, we increase the start time of the destination operation by the smallest multiple of the II that fixes this problem. After the adjustment has been made, successors of the delayed operation may face a similar problem and we also delay their start times as necessary until all start time constraints are satisfied.

Note that a smaller delay may work sometimes, but would place the operation in a different MRT cell, which may not be available. For simplicity, we always use a multiple of the II, as shown in Fig. 3: while all inputs to operation E are ready by cycle 10, our algorithm schedules it for time step 11 since that is the earliest start time compatible with the MRT chosen previously.

After repairing the start time constraints, the start times and bindings of each partial solutions can be combined without further changes to produce a valid schedule and bindings for the entire DFG. As each operation belongs to exactly one partition, the start time and binding of each operation is uniquely determined, start time constraints are now satisfied, and the up-front partitioning of the MRT ensures no operator is over-subscribed in the combined solution. As the bindings are combined without changes, the effort expended by the ILP solver trying to optimize the bindings within each partition carries over into the overall solution.

To ensure that each partition only uses the parts of the MRT assigned to it, we need to modify the ILP construction slightly. We create *virtual* operations that occupy partially-available operators in the time steps assigned to other partitions, substituting constants for the ILP decision variables relating to the virtual operations. This increases the size of the ILP, which is quadratic in the number of operations even without our extensions. We limit this increase by partitioning the MRT such that every partition has at most two operators partially assigned to it, and where possible prefer to exclusively assign operators

to a single partition. This is the case for Adder 2 in Fig. 3, which is assigned entirely to Partition 2 even though there is only one addition in that partition.

4 Implementation

We extended an existing toolflow for generating SPN inference accelerators [20] implemented in Chisel, replacing the fully spatial datapath and ASAP scheduling used there with a modulo-scheduled, operator-shared datapath. To realize the schedule-dependent sharing of operators in the new datapath, we use a local state machine per operator, which tracks the current cycle modulo II, and translates this local state to the control signals of multiplexers for selecting the current inputs to the operator. Due to space constraints, we must leave the presentation of the overall accelerator architecture to the prior work [20].

During scheduling, the multiple sub-problems generated by the divide-and-conquer approach are solved in parallel by launching multiple single-threaded ILP solver instances in a thread pool.

Graph partitioning is performed recursively, applying balanced 1-cuts repeatedly until the number of operations subject to operator sharing (which influences the size and difficulty of the ILP) falls below a user-specified threshold. We call this parameter the *split threshold* S and will evaluate its impact in Section 5. This approach works well in our domain, because most algorithms for learning SPNs produce trees rather than general directed acyclic graphs, but the latter could also be supported by using a more general graph partitioning method.

As in prior work, the open-source TaPaSCo framework [7] is used to integrate the core accelerator (load unit, store unit, data path, and controller) into a platform-specific SoC design, and provides a software API for interacting with the accelerator from the host CPU. In contrast to prior work [20], we also target MPSoC systems with *shared* memory between the host CPU and FPGA. However, the current version of TaPaSCo does not yet support *cache-coherent* shared memory between host and accelerator. To avoid the costs of copying input and output buffers, we use a custom user-space mappable memory buffer to make the input and output data available to both CPU and FPGA, rather than using the TaPaSCo-provided APIs for host-accelerator data transfers. This buffer is marked as cacheable, and the CPU cache is flushed explicitly and invalidated before launching inference jobs on the accelerator.

5 Evaluation

Our evaluation comprises two parts: an evaluation of the proposed ILP formulation and graph partitioning-based heuristic on a range of SPNs, FPGA platforms, IIs and resource constraints; and a case study on real hardware platforms suitable for embedded computing, comparing operator-shared accelerators on a Xilinx UltraScale+ MPSoC device to an Nvidia Jetson Xavier device.

In both parts of the evaluation, we use the customized floating-point formats and operators developed for SPN inference in prior work [20], which represent the

probabilities on a linear scale, and were found to be more resource-efficient than a log-scale representation of probabilities in most cases. For the SPNs already investigated in the prior work, we use the format parameters as reported there. For the other SPNs, we use a format with 10 exponent bits and 26 mantissa bits, as it has the largest exponent range, and therefore is least vulnerable to overflow and underflow out of the currently implemented formats.

5.1 Scheduler Evaluation

For the scheduler evaluation, we target Digilent PYNQ-Z1, AVNET Ultra96-V2, and Xilinx VC709 boards. We used 14 out of 16 SPNs used in prior work [20], excluding NIPS5 and MSNBC 300 for being too small to benefit from operator sharing on any of the target platforms. To these, we add another SPN over binary data (DNA) and three large-scale artificial examples that were randomly generated to serve as stress tests: fully spatial realizations of these SPNs would require 1499, 2249 and 2699 floating-point adders and multipliers respectively, far larger than practical for ILP-based modulo scheduling.

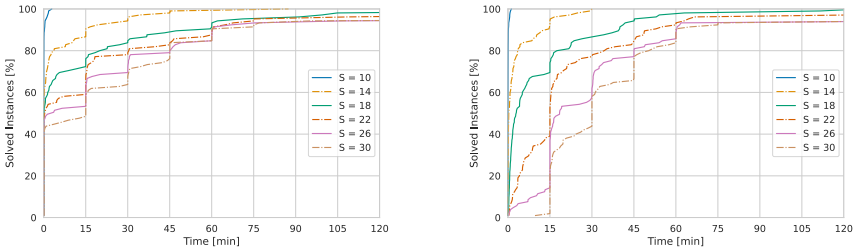
Experimental Setup Each of these 18 SPNs is tested against the resource model of each target platform to determine the resource-constrained minimum II. In many cases, this results in an accelerator design that would be severely memory-bound and could use a larger II – allowing more sharing and thus requiring fewer FPGA resources – without loss of performance. Thus, we also compute an alternative II (per SPN and platform) that would balance computational throughput with memory bandwidth requirement. Out of these $18 \times 3 \times 2$ candidate {SPN, Platform, II} triples, the 35 unique combinations with II from two to seven (inclusive) are used.

For each of the 35 {SPN, Platform, II} combinations, we perform scheduling and binding for three different resource constraints: the minimum number of operators possible, that minimum scaled up by a factor of 1.25 (rounded), and the largest number of operators that will fit on the device. These 105 scheduling tasks capture a wide range of DFG sizes and available number of operators.

Each scheduling task is solved by constructing a single large ILP instance as well as by our proposed divide-and-conquer heuristic with varying granularity for the graph partitioning step. In each case, we compare the baseline ILP formulation [21] to our proposed extension (Section 3.1).

Experiments were performed with Gurobi 8.1 as ILP solver, on systems equipped with two 12-core Intel Xeon E5-2680 v3 CPUs and 64 GiB of RAM. Each scheduler run was given access to four cores and 16 GiB of RAM, and wall clock running time was limited to two hours each. For each individual subproblem generated by graph partitioning, the ILP solver was given a time limit of 15 minutes.

For the split threshold S controlling the granularity of the graph partitioning, we evaluate $S \in \{1, 5, 10, 14, 18, 22, 26, 30\}$ – using 1 as naive baseline, 5 and 10 as very fast but low-quality variants, and equidistant values from 10 to 30 to



(a) Baseline ILP Formulation (b) Proposed ILP Formulation

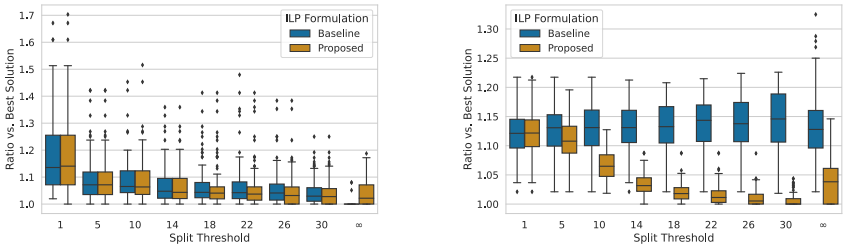
Fig. 4. Scheduler runtime profile for different split thresholds (S) with the baseline and proposed ILP formulation, plotting scheduler run time against the cumulative percentage of instances solved within that time frame.

explore the trade off between running time and solution quality as sub-graphs become more complex.

Scheduler Runtime Attempting to schedule the entire DFG by a single ILP is impractical on many of the instances in our benchmark suite. With the baseline ILP formulation, the solver finds a solution for just 57 out of 105 instances, of which only 30 are proven to be optimal solutions, while the other instances run out of time or memory while solving the ILP, or already while creating the constraints. Results are even worse with our proposed extension of the ILP: only 31 solutions are found, and none of them could be proven optimal.

In contrast, fine-granular graph partitioning ($S \leq 10$) enables heuristic scheduling within a minute on almost all examples, with only a few exceptions taking slightly longer. As Fig. 4 shows, run times rapidly increase as the graph partition gets more coarse. Curiously, although a significant number of instances are solved almost instantaneously with the baseline ILP formulation, with the extended ILP formulation, we observe fewer outliers that take exceptionally long to schedule. With the extended ILP formulation, the configuration $S = 14$ schedules most examples in 15 minutes, and all within 30 minutes. Even with $S = 30$, the majority of instances are successfully scheduled within one hour, but too many exceed the two hour time limit (especially with the baseline ILP formulation) to claim that larger values of S are always beneficial.

Solution Quality For lack of a clear baseline to compare the scheduling algorithms against, we resort to scoring the different scheduler variants by how well their solutions for each instance score relative to the best solution found by all of the variants evaluated. We compare both the schedule length (datapath latency) achieved and the multiplexer size (as encoded in the ILP objective) achieved by each variant. We record this ratio for every instance and report the distribu-



(a) Schedule length objective (b) Multiplexer size objective

Fig. 5. Solution quality of different scheduler variants w.r.t. to schedule length (5a) and multiplexer size (5b) objective. Split threshold $= \infty$ is the case of a single ILP for the whole graph. Values closer to 1 are better.

Table 1. Number of solutions found by each scheduler variant.

ILP-Form.	Split Threshold								
	1	5	10	14	18	22	26	30	∞
Baseline	105	105	105	105	103	101	99	99	57
Proposed	105	105	105	105	104	101	98	98	31

tion of these ratios in Fig. 5a and Fig. 5b (standard box plots with whiskers at $Q_1 - 1.5 \cdot \text{IQR}$ and $Q_3 + 1.5 \cdot \text{IQR}$).

Note that many variants, especially those with a single large ILP for the entire problem, did not find solutions for all of the 105 scheduling tasks. The number of solutions found within two hours is listed in Table 1.

Generally, coarser partitioning (larger S) yields better results – at the cost of longer running time, as seen above. The divide-and-conquer heuristic combined with our proposed ILP formulation improves multiplexer sizes, and prioritizing this objective does not have a negative impact on the schedule length. However, the improvements beyond $S = 14$ are marginal and may not justify the significantly longer running times.

While the baseline ILP without graph partitioning ($S = \infty$) achieves best-in-class schedule lengths on most instances where the ILP solver finds any solution, the third quartiles show that the heuristic schedulers with $S \geq 14$ get within 10% of the schedule length achieved by the ILP solver in the majority of cases, and occasionally obtain even better results.

5.2 Hardware Evaluation

Out of the 35 {SPN, Platform, II} combinations used in the scheduler evaluation, 13 target the AVNET Ultra96-V2 device. We generate accelerators for these configurations, using the heuristic scheduler with our proposed ILP formulation and $S = 14$. This configuration gives acceptable results, while also reliably finishing

Table 2. SPN graph properties, scheduling results, FPGA resource utilization on Ultra96-v2 platform and comparison of inference throughput. FPGA utilization is given as percent of the overall available resources. Best throughput results highlighted bold.

Benchmark	II	SL	Add	Mult.	Freq. [MHz]	Resource Util. [%] LUT	Reg.	CLB	DSP	Throughput [samples/µs] Xavier CPU	Xavier GPU	FPGA
Accidents	2	81	27	217	275	61.30	38.04	92.23	60.56	7.53	17.61	**39.85**
Audio	4	88	12	275	280	51.94	30.91	77.65	76.67	4.05	16.82	**35.01**
DNA	3	73	2	363	260	66.09	38.49	96.92	67.22	1.51	15.22	**25.95**
Netflix	2	73	11	231	260	62.07	37.82	92.85	64.44	3.44	26.63	**44.31**
Plants	3	140	14	256	280	53.58	35.85	86.63	95.56	4.21	30.23	**48.98**
NIPS20	2	47	7	56	350	38.00	20.33	57.03	15.56	27.55	19.42	**30.01**
NIPS30	2	65	10	87	345	49.57	24.94	71.52	24.44	15.95	15.82	**24.67**
NIPS40	3	74	16	122	350	51.52	26.62	76.17	22.78	10.27	12.61	**21.39**
NIPS50	4	80	16	143	340	57.95	26.29	78.72	20.00	7.79	11.59	**17.74**
NIPS60	4	77	13	156	350	57.85	27.74	81.89	21.67	5.23	10.28	**14.86**
NIPS70	5	88	14	180	205	62.52	28.36	84.17	20.83	3.09	9.36	**14.00**
NIPS80	2	85	32	265	245	83.16	43.53	99.34	74.72	3.20	6.41	**12.48**
NIPS80	5	93	32	265	245	68.49	34.18	93.84	30.28	3.20	6.41	**12.39**

in half an hour, which is a typical time frame for FPGA implementation of the entire accelerator for the target device.

We performed a design space exploration using the development version of the open-source framework TaPaSCo[4] and Vivado version 2019.2 to determine the highest possible frequency and corresponding FPGA resource utilization. Results are reported in Table 2, with resource utilization given as percentage of the total number of available resources (70,560 LUT, 141,120 Reg., 8,820 CLB, 360 DSP).

With these maximum frequencies, we compare the performance of the FPGA accelerators with the CPU and GPU implementations of the same inference computations running on another SoC suitable for embedded and edge AI computation, namely an Nvidia Jetson AGX Xavier SoC, having ARM CPU cores and an integrated 512-core Volta-class GPU. Similar to prior work [20], optimized C++ (single-threaded) or CUDA code is generated from the SPN description using an automated toolflow and then compiled by the respective compiler available on the Jetson Xavier System (NVCC version 10.0, GCC version 7.5.0). Just as in the Ultra96 used for FPGA performance measurements, CPU and GPU on the Jetson AGX Xavier share the same physical memory, which removes the need for expensive host-accelerator data transfers.

Table 2 lists the throughput achieved by the CPU, GPU and FPGA implementations. Each measurement is averaged over five runs. Our accelerators achieve better throughput than the implementations on CPU (geo.-mean speedup 4.4x) and GPU (geo.-mean speedup 1.7x) on the Xavier device.

Avoiding data transfers between CPU and GPU, respectively CPU and FPGA, has significant impact: it allows these embedded SoCs to achieve performance

[4] https://github.com/esa-tu-darmstadt/tapasco

much closer to the more powerful workstations evaluated in prior work [20] than one would expect from comparing hardware specifications. On several benchmarks, the Xavier GPU implementation even achieves significantly higher throughput than the discrete Nvidia 1080Ti GPU used in prior work, primarily because the latter needs to transfer all input data and results over PCIe.

Note that the two FPGA accelerators for NIPS80 have vastly different datapath throughput (II = 2 versus 5) and resource (DSP) requirements. They achieve essentially the same end-to-end performance because the larger II = 2 configuration is limited by memory bandwidth, while the smaller II = 5 configuration was selected to match the memory bandwidth.

6 Related Work

Two key components of our work are the use of graph partitioning to accelerate modulo scheduling and the optimization of operator bindings. The discussion in this section focuses on works related to these aspects. Please note that many other approaches to heuristic modulo scheduling exist [1, 3, 24].

Compared to scheduling in compilation flows for neural networks on FPGAs, our approach works on a much finer level of granularity. As outlined in the survey by Venieris et al. [22], scheduling in compilation flows for neural networks typically happens on the granularity of coarse-grained neural network operations, such as matrix multiplication, convolution, or even entire layers, whereas our scheduler operates on individual arithmetic operations.

6.1 Graph Transformations For Modulo Scheduling

Fan et al. [6] previously used graph partitioning to decompose large modulo scheduling tasks into multiple sub-problems. Due to recurrences, solutions to the sub-problems can not necessarily be combined into a valid solution to the whole problem. To address this, they perform scheduling of sub-graphs sequentially and back-track when later sub-graphs cannot be scheduled due to conflicts arising from previous decisions. As a consequence, this approach fails to schedule some examples even with a relatively fine-grained partitioning (ca. eight operations per sub-graph).

Dai and Zhang [5] used strongly connected components (SCCs) to partition the graph. As this partitioning does not split recurrences, partial solutions can always be combined into a full schedule. They demonstrate that this often accelerates scheduling significantly, though it is less effective when a single SCC encompasses most of the graph.

6.2 Optimization of Bindings

There are numerous works optimizing the bindings alongside the schedule as one of the key factors affecting the physical realization of operator sharing. Cong and Xu [4] perform this in a separate step after scheduling using a heuristic based

on min-cost network flows. LegUp [2] is a more recent example of binding as a separate phase after scheduling, focusing on balancing multiplexer sizes.

Other works are more closely related to our approach of integrated scheduling and binding. The aforementioned work by Fan et al. [6] focuses on ASIC implementations, while Memik et al. [9] target FPGA architectures. More recently and most closely related to our ILP formulation, Sittel et al. [18] incorporated the operator bindings into an ILP-based modulo scheduler to optimize the area required for registers holding intermediate values.

7 Conclusion & Outlook

In this paper, we presented an ILP formulation for modulo scheduling and binding of SPN inference computations, and a divide-and-conquer heuristic that makes the ILP-based approach practical for use on large SPNs by graph partitioning and combination of partial solutions.

This heuristic can schedule very large examples in minutes, while finding the optimal II by construction and making only minor sacrifices in schedule length – within 10% of the best known solution for most instances. In addition, our extended ILP formulation also reduces datapath multiplexing significantly, compared to scheduling approaches that only target the schedule length.

Using this scheduling algorithm, we generate SPN inference accelerators on an embedded FPGA-CPU hybrid SoC, where a fully spatial realization of the datapath would exceed the available hardware resources. These FPGA accelerators achieve higher throughput than CPU and GPU implementations on an Nvidia Jetson Xavier SoC in our benchmarks, with geometric mean speed-up of 4.4x over CPU and 1.7x over GPU.

The properties of acyclic data flow graphs that enable our divide-and-conquer heuristic also suggest other approaches to heuristic scheduling that hold promise for improving the running time and/or solution quality further. Since a feasible suboptimal solution is easy to find, local search approaches such as simulated annealing could be used as well, which allow specifying non-linear constraints and objectives directly.

References

1. Canis, A., Brown, S.D., Anderson, J.H.: Modulo SDC scheduling with recurrence minimization in high-level synthesis. In: Int. Conf. on Field Programmable Logic and Applications (FPL) (2014)
2. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Czajkowski, T., Brown, S.D., Anderson, J.H.: LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. ACM Transactions on Embedded Computing Systems (TECS) **13**(2) (2013)
3. Codina, J.M., Llosa, J., González, A.: A comparative study of modulo scheduling techniques. In: Int. Conf. on Supercomputing (ICS '02) (2002)
4. Cong, J., Xu, J.: Simultaneous FU and register binding based on network flow method. In: Design, Automation and Test in Europe (2008)

5. Dai, S., Zhang, Z.: Improving scalability of exact modulo scheduling with specialized conflict-driven learning. In: Design Automation Conf. (2019)
6. Fan, K., Kudlur, M., Park, H., Mahlke, S.: Cost sensitive modulo scheduling in a loop accelerator synthesis system. In: IEEE/ACM Int. Symp. on Microarchitecture (MICRO'05) (2005)
7. Heinz, C., Hofmann, J., Korinth, J., Sommer, L., Weber, L., Koch, A.: The TaPaSCo Open-Source Toolflow. Journal of Signal Processing Systems (May 2021). https://doi.org/10.1007/s11265-021-01640-8
8. Lam, M.: Software pipelining: An effective scheduling technique for VLIW machines. In: Programming Language Design and Implementation (PLDI) (1988)
9. Memik, S.O., Memik, G., Jafari, R., Kursun, E.: Global resource sharing for synthesis of control data flow graphs on FPGAs. In: Design Automation Conf. (2003)
10. Molina, A., Vergari, A., Di Mauro, N., Natarajan, S., Esposito, F., Kersting, K.: Mixed sum-product networks: A deep architecture for hybrid domains. In: Thirty-second AAAI Conf. on artificial intelligence (2018)
11. Ober, M., Hofmann, J., Sommer, L., Weber, L., Koch, A.: High-throughput multi-threaded sum-product network inference in the reconfigurable cloud. In: Workshop on Heterogeneous High-performance Reconfigurable Computing (2019)
12. Oppermann, J., Sittel, P., Kumm, M., Reuter-Oppermann, M., Koch, A., Sinnen, O.: Design-space exploration with multi-objective resource-aware modulo scheduling. In: Conf. on Parallel and Distributed Computing (Euro-Par) (2019)
13. Peharz, R., Tschiatschek, S., Pernkopf, F., Domingos, P.: On theoretical properties of sum-product networks. In: Artificial Intelligence and Statistics (2015)
14. Peharz, R., Vergari, A., Stelzner, K., Molina, A., Shao, X., Trapp, M., Kersting, K., Ghahramani, Z.: Random sum-product networks: A simple but effective approach to probabilistic deep learning. In: Proceedings of UAI (2019)
15. Poon, H., Domingos, P.: Sum-product networks: A new deep architecture. In: IEEE Int. Conf. on Computer Vision Workshops) (2011)
16. Rau, B.R.: Iterative modulo scheduling. Int. J. Parallel Program. **24**(1) (1996)
17. Sánchez-Cauce, R., París, I., Díez, F.J.: Sum-product networks: A survey. IEEE Transactions on Pattern Analysis and Machine Intelligence (2021)
18. Sittel, P., Kumm, M., Oppermann, J., Möller, K., Zipf, P., Koch, A.: ILP-based modulo scheduling and binding for register minimization. In: Int. Conf. on Field Programmable Logic and Applications (FPL) (2018)
19. Sommer, L., Oppermann, J., Molina, A., Binnig, C., Kersting, K., Koch, A.: Automatic mapping of the sum-product network inference problem to FPGA-based accelerators. In: IEEE Int. Conf. on Computer Design (ICCD) (2018)
20. Sommer, L., Weber, L., Kumm, M., Koch, A.: Comparison of arithmetic number formats for inference in sum-product networks on FPGAs. In: Int. Symp. on Field-Programmable Custom Computing Machines (FCCM) (2020)
21. Šůcha, P., Hanzálek, Z.: A cyclic scheduling problem with an undetermined number of parallel identical processors. Comput. Optim. Appl. (2011)
22. Venieris, S.I., Kouris, A., Bouganis, C.S.: Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions. ACM Comput. Surv. **51**(3) (2018)
23. Weber, L., Sommer, L., Oppermann, J., Molina, A., Kersting, K., Koch, A.: Resource-efficient logarithmic number scale arithmetic for SPN inference on FPGAs. In: Int. Conf. on Field-Programmable Technology (FPT) (2019)
24. Zhang, Z., Liu, B.: SDC-based modulo scheduling for pipeline synthesis. In: IEEE/ACM Int. Conf. on Computer-Aided Design (2013)