SkyCastle: A Resource-Aware Multi-Loop Scheduler for High-Level Synthesis

Julian Oppermann*, Lukas Sommer*, Lukas Weber*,
Melanie Reuter-Oppermann†, Andreas Koch* and Oliver Sinnen‡
*Embedded Systems and Applications Group, Technische Universität Darmstadt, Germany.
{oppermann, sommer, weber, koch}@esa.tu-darmstadt.de
†Discrete Optimization and Logistics Group, Karlsruhe Institute of Technology, Germany. melanie.reuter@kit.edu
‡Parallel and Reconfigurable Computing Lab, University of Auckland, New Zealand. o.sinnen@auckland.ac.nz

***Abstract*—A common optimisation problem in the high-level synthesis (HLS) of FPGA-based accelerators is to find a microarchitecture that maximises the performance while keeping the utilisation of the device's low-level resources below certain limits.**

We propose to tackle it directly *as part of* the HLS scheduler. To that end, we formalise a general, integrated scheduling and allocation problem for HLS kernels, and present *SkyCastle*, a novel resource-aware multi-loop scheduler using integer linear programming to solve it for a subclass of kernels composed of multiple, nested loops. In order to demonstrate the practical applicability of the approach, we model the scheduler in such a way as to be plug-in compatible with the Xilinx Vivado HLS engine, allowing the computed solutions to be fed back into its synthesis flow.

We evaluate SkyCastle for three non-trivial kernels from the machine learning, signal processing, and physical simulation domains, on two FPGA devices. Additionally, we investigate the *replication* of slightly slower, but smaller accelerators as a means to further boost the overall performance. In contrast to Vivado HLS' default settings, which aim at maximum performance but may fail in later synthesis steps, the solutions computed by our scheduler always result in synthesisable designs.

I. Introduction

Modern field-programmable gate arrays (FPGA) have become large enough to accommodate far more functionality than one simple computational kernel, opening up new opportunities and challenges for designers. For example, when using all available resources, complex multi-phase kernels can be implemented within a single accelerator to reduce the number of context switches [1], [2]. On the other hand, it is also reasonable to partition the resources, e.g. to replicate an accelerator for parallel processing [3], or to share one device among different groups in a research project [4]. In all of the aforementioned situations, the question is usually the same: *How to obtain the best performance within the given resource constraints?*

High-level synthesis (HLS) tools are an ideal starting point to tackle this optimisation problem, as they can construct microarchitectures with different trade-offs for the accelerator's performance and resource demand from the *same* algorithmic specification. This work targets HLS tools that accept C/C++ code as input. We argue that the most influential control knob in this context is the amount of *pipelining* used in the microarchitecture.

Listing 1. Sum-Product Network example

```
double spn(...)          { /* 10 FP mul, 1 FP add */ }
double spn_marginal(...) { /*  8 FP mul, 1 FP add */ }

double top(char i1, char i2, char i3, char i4) {
  // most probable explanation for "i5"
  char maxClause = -1;  double maxProb = -1.0;
  MPE: for (char x = 0; x < 0xFF; x += 4) {
         double p0 = spn(i1, i2, i3, i4, x);
         double p1 = spn(i1, i2, i3, i4, x+1);
         double p2 = spn(i1, i2, i3, i4, x+2);
         double p3 = spn(i1, i2, i3, i4, x+3);
         maxProb   = ... // max(maxProb, p0, p1, p2, p3);
         maxClause = ... // argument value for i5 that
                         // yielded new value for maxProb
  }
  double pM = spn_marginal(i2, i3, i4, maxClause);
  return maxProb / pM;
}
```

Loop pipelining results in the partial overlapping of subsequent loop iterations, and is enabled by modulo schedulers: Given a control-data-flow graph (CDFG) that represents the computation of one loop iteration, a modulo scheduler computes a schedule that can be repeated after a certain number of time steps, called the *initiation interval* (II). A smaller II results in more overlapping of iterations and in consequence, in a shorter execution time for the whole loop, but also requires more resources as less operator sharing is possible.

Pipelining is also applicable to functions, where it results in an overlapping evaluation of the function's body for different sets of arguments. The same trade-off considerations and scheduling techniques apply to both forms of pipelining, though.

As a motivational example, consider the excerpt from the inference process in a Sum-Product Network (see also Section V-A1) in Listing 1. We instruct Xilinx Vivado HLS to pipeline the loop labeled `MPE`, which automatically pipelines the function `spn` as well. The function `spn_marginal` will be inlined automatically by the HLS frontend. Vivado HLS attempts, and succeeds, to construct the maximum performance version of this kernel with II=1 for the loop and the function. However, as this results in a fully-spatial microarchitecture, each operation in the computation requires their own operator. When targeting the popular ZedBoard, such a design requires 499 DSP slices, which exceeds the available 220 slices by a large margin. Finding the lowest-latency version that still fits on the device requires considering a) the degree of pipelining applied to function spn, b) the number of spn-instances, c) the amount of pipelining for loop MPE (which depends on a) and b)), and lastly, d) the operator allocation for the top-level function, which influences c) as well as the latency of the non-pipelined computation at the end of top. Here, the fastest solution is to pipeline spn and MPE with II=4, allocate two multipliers, one adder, one divider, three floating-point comparators and four instances of spn inside function top.

This paper makes the following key contributions. First, we provide the formal definition of an integrated scheduling and allocation problem that models these issues in general for HLS kernels containing arbitrarily nested loops and functions. Secondly, we present *SkyCastle*, a resource-aware multi-loop scheduler capable of solving the problem for a subclass of kernels composed of multiple, nested loops in a single top-level function.

Both the proposed problem definition and the scheduler apply to, or can be easily adapted to, any HLS flow. However, in order do demonstrate the practical applicability of the approach, we tailored the scheduler to be *plug-in* compatible with the Vivado HLS engine. To that end, we faithfully extract the actual scheduling and allocation problems faced by Vivado HLS from its intermediate representation. Afterwards, we feed the directives required to control pipelining and the operator allocation according to the solutions determined by our scheduler back to the synthesis flow.

Vivado HLS' default settings aim at maximum performance but may fail in later synthesis steps due to resource demands that exceed the capacity on the target device. Even with our proof-of-concept implementation, we are able to guide Vivado HLS to generate synthesisable microarchitectures for three complex kernels on two FPGA devices. On the larger device, we also explore partitioning the available resources in order to enable the replication of slightly slower, but smaller accelerators as a means to further boost the overall performance. The multi-accelerator solution easily outperforms the *theoretical* maximum-performance, single-accelerator design, which is actually unsynthesisable for two of our three case studies.

II. Related work

We discuss the related work with regards to the kind of exploration used to discover solution candidates to answer the initially stated research question.

A. As part of the HLS scheduler

The most direct way to solve the problem is to model it inside the HLS scheduler. This requires considering the highly interdependent problems of scheduling and (operator) allocation together, but has two main benefits: First, the resulting schedules are guaranteed to be feasible because they were computed by an actual scheduler that considers all nuances of the problem, such as tight inter-iteration dependences that might require more operators than the theoretical lower bound. Secondly, no external exploration is needed.

Recently, Oppermann et al. [5] established a framework to handle low-level resource constraints in modulo schedulers based on integer linear programming, such as the formulation by Eichenberger and Davidson [6] and the Moovac formulation [7], by making the operator allocation variable. They then showed how the extended schedulers can be used to efficiently compute different Pareto-optimal solutions with respect to the two conflicting objectives of maximising the throughput vs. minimising the resource demand. By itself, however, their approach is not sufficient to tackle the problem stated for this work, because it only modulo-schedules *individual* loops under the assumption of an independent operator allocation, instead of more complex multi-loop kernels. However, our proposed scheduler builds upon their framework and can be seen as a significant extension of their ideas to suit a more practical context.

B. Pipelining-focussed exploration

The next category is comprised of approaches that control the amount of pipelining in a complex kernel by determining target IIs for its pipelined parts, e.g. stages in a pipelined streaming application [8], stateless actors in a synchronous data-flow graph [9], [10], or loops arranged in a directed, acyclic graph [11]. Common aspects in these works are a) the use of a performance model to choose the IIs, b) the approximation of latencies of the individual parts, and c) the derivation of the operator allocation from the II, without checking the feasibility.

Differences exist in the chosen objectives. Li et al. [11] tackle a problem very similar to ours: minimise the overall latency of a kernel, subject to low-level resource constraints, and consider the benefits of slightly slower, but better replicable implementations.

Cong et al. [9], [10] and Kudlur et al. [8] attempt to minimise the required resources to fulfil an externally given throughput constraint, and, in consequence, would need some kind of exploration to find the highest throughput that still satisfies given resource constraints. Note, though, that these approaches employ more elaborate models of generated microarchitectures than we do. For example, the cost-sensitive modulo scheduler [12] used in [8] considers the different bitwidths of operations as well as the required interconnects and register storage, but crucially, performs the *allocation* of functional units before scheduling.

C. General design-space exploration

General design-space exploration approaches form the last (and largest) category, whose representatives may be model-based analysis tools [13], [14], integrated in an HLS flow [15], [16], or consider the HLS tool as a black box and emit directives to control the microarchitecture generation [17]. These approaches usually consider other techniques besides pipelining, such as loop unrolling, function inlining, or partitioning of arrays. Most tools aim to explore a diverse set of solutions to let the (human) designer choose from. A notable exception is the work of Prost-Boucle et al. [16], which describes an autonomous flow that successively applies transformations to improve the kernel's latency while obeying low-level resource constraints. However, internally, the allocation of operators precedes the scheduling phase.

III. Multi-Loop Scheduling Problem

Given an HLS kernel in a structured programming language, composed of multiple, optionally pipelined, loops and functions, we want to minimise the latency of one activation of the kernel's unique top-level function, subject to resource constraints in terms of the low-level FPGA resources, e.g. look-up tables (LUT) or DSP slices.

A. Overview

Figure 1 outlines the multi-loop scheduling problem (MLSP). We have a set of dependence graphs that each correspond to the body of a loop in the kernel, derived e.g. from a CDFG representation inside the HLS tool. The non-loop parts of functions are treated uniformly as single-iteration loops at the outermost level. Our goal in the *scheduling* part of the problem is to compute start times for each operation, and to determine a feasible initiation interval for graphs originating from pipelined parts of the kernel.

The operations in the graphs require operators, which occupy a specific amount of the FPGA's resources. HLS tools may *share* operators among several operations if the resource demand of the operator is higher than the cost of the additional multiplexing logic. Determining the number of operators of each type constitutes the *allocation* part of problem, and has a strong influence on the scheduling result.

We introduce the concept of an *allocation domain*, which provides the operators for a subset of the graphs. All graphs in an allocation domain share these operators, but assume exclusive access to them. This means the parts of the computation represented by any pair of graphs in the same allocation domain will execute sequentially at runtime. In contrast, graphs in different allocation domains can execute in parallel due to their independent sets of operators.

Figure 1 also presents the canonical examples for these concepts, inspired by Vivado HLS, which implements operator sharing at the function level. Here, a function in the kernel is an allocation domain. The loops in the function are the graphs that use and share the allocation domain's operators. Nested loops are represented by special operations that reference another graph in the same allocation domain. Lastly, function calls in any of the loops reference another graph embedded in its own allocation domain, which needs to be instantiated as a special operator type in the surrounding allocation domain. We will implicitly assume theses correspondences for the rest of this paper, and name the special operations and operators accordingly in order to keep the following problem definition as intuitive as possible. Note, however, that the underlying modelling ideas apply to other resource sharing strategies as well, e.g. sharing only within the same loop level.

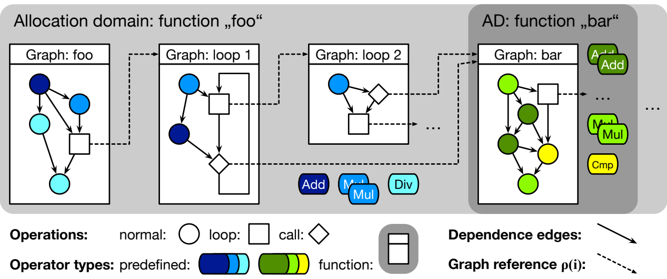

Figure 1. An example instance of the multi-loop scheduling problem

B. Formal definition

The target device is abstracted to its low-level resource types R, and the number of elements $\bar{N}_{\mathsf{r}} \in \mathbb{N}$ it provides for each $\mathsf{r} \in R$.

Let G denote the set of dependence graphs, and $\mathsf{G}^{\mathrm{pl}} \subseteq \mathsf{G}$ the set of pipelined graphs. The graph corresponding to the kernel's top-level function is identified as $\mathsf{g}^{\mathrm{top}}$. The set of operator types Q is statically partitioned into shared (Q^{Sh}) and exclusive (Q^{Ex}) operator types, and orthogonally, into predefined (Q^{Pd}) and function (Q^{Fu}) operator types. The set $\mathcal{AD}$ specifies the allocation domains. Each graph $\mathsf{g} \in \mathsf{G}$ and each operator type $\mathbf{q} \in Q$ is a member of exactly one allocation domain $A = (\mathsf{G}_A, Q_A) \in \mathcal{AD}$. Let the function α represent the mapping of a graph or an operator type back to its allocation domain. We use the notation Q_A^{Xx} for the set of operator types of a certain kind Xx in an allocation domain A.

A graph $\mathsf{g} \in \mathsf{G}$ is defined by its sets of operations O_{g} and dependence edges $E_{\mathsf{g}} = \{(i \to j)\} \subseteq O_{\mathsf{g}} \times O_{\mathsf{g}}$.

An operation $i \in O_{\mathsf{g}}$ has a start time t_i and a latency l_i. The latency models how many time steps after t_i the operation's result is available. We need to defer the actual definition of the latency until later, due to the recursive nature of the problem. We distinguish normal ($O_{\mathsf{g}}^{\mathrm{No}}$), loop ($O_{\mathsf{g}}^{\mathrm{Lo}}$) and call ($O_{\mathsf{g}}^{\mathrm{Ca}}$) operations. The normal operations $i \in O_{\mathsf{g}}^{\mathrm{No}}$ are mapped to a predefined operator type $\mathbf{q} \in Q_{\alpha(\mathsf{g})}^{\mathrm{Pd}}$ by the function σ. The loop operations $i \in O_{\mathsf{g}}^{\mathrm{Lo}}$ reference another graph $\rho(i) \in \mathsf{G}_{\alpha(\mathsf{g})}$, specified by the function ρ, in the same allocation domain. Lastly, the call operations $i \in O_{\mathsf{g}}^{\mathrm{Ca}}$ reference both a graph $\rho(i) \in \mathsf{G} \setminus \mathsf{G}_{\alpha(\mathsf{g})}$ and a function operator type $\sigma(i) \in Q_A^{\mathrm{Fu}}$. We group operations using the same operator type together as $O_{\mathsf{g}}^{\mathbf{q}} = \{i \in O_{\mathsf{g}}^{\mathrm{No}} \cup O_{\mathsf{g}}^{\mathrm{Lo}} : \sigma(i) = \mathbf{q}\}$.

Each dependence edge $(i \to j)$ models a precedence relationship between the operations $i, j \in O_{\mathsf{g}}$. The edge distance d_{ij} expresses how many iterations later the precedence has to be satisfied. We call edges with a non-zero-distance *backedges*. The graph may contain cycles that include at least one backedge.

We define the graph's schedule length T_{g} as the latest finishing time among its operations, formally $T_{\mathsf{g}} = \max_{i \in O_{\mathsf{g}}}(t_i + l_i)$. In case $\mathsf{g} \in \mathsf{G}^{\mathrm{pl}}$, we introduce II_{g} to model g's initiation interval. We assume to have a constant known trip count c_{g}.

An operator type $\mathbf{q} \in Q$ has a blocking time $b_{\mathbf{q}}$ and a resource demand $n_{\mathbf{q},\mathsf{r}}$ regarding each of the device's resource types $\mathsf{r} \in R$. The blocking time specifies the minimum number of time steps after which the operator can accept new inputs. Again, we need to defer the actual definition until later. If $\mathbf{q} \in Q^{\mathrm{Fu}}$, we let the function ρ map $\mathbf{q}$ to the referenced graph.

The allocation $a_{\mathbf{q}}$ represents the number of operator instances of type $\mathbf{q}$ in the associated allocation domain $\alpha(\mathbf{q})$. For each allocation domain A, we define its demand $n_{A,\mathsf{r}}$ of a resource type $\mathsf{r} \in R$ as $n_{A,\mathsf{r}} = \sum_{\mathbf{q} \in Q_A} a_{\mathbf{q}} \cdot n_{\mathbf{q},\mathsf{r}}$.

We can now revisit the deferred definitions. An operation $i \in O_{\mathrm{g}}$ for a graph g derives its latency either as a parameter $L_{\sigma(i)}$ from its associated, predefined operator type $\sigma(i)$, or from the scheduling result of the graph it references. Here, the latency is equivalent to the sequential, respectively overlapping, execution of $c_{\rho(i)}$ iterations. Recall that function bodies are treated as single-iteration loops.

$$l_i = \begin{cases} L_{\sigma(i)} & i \in O^{\mathrm{No}}_{\mathrm{g}} \\ c_{\rho(i)} \cdot T_{\rho(i)} & i \in O^{\mathrm{Lo}}_{\mathrm{g}} \wedge \rho(i) \notin \mathrm{G}^{\mathrm{pl}} \\ (c_{\rho(i)} - 1) \cdot \mathrm{II}_{\rho(i)} + T_{\rho(i)} & i \in O^{\mathrm{Lo}}_{\mathrm{g}} \wedge \rho(i) \in \mathrm{G}^{\mathrm{pl}} \\ T_{\rho(i)} & i \in O^{\mathrm{Ca}}_{\mathrm{g}} \end{cases}$$

The parameter $B_{\mathbf{q}}$ specifies the blocking time for each predefined operator types $\mathbf{q} \in Q$. Function operators derive it from the scheduling solution of the referenced graph $\rho(\mathbf{q})$.

$$b_{\mathbf{q}} = \begin{cases} B_{\mathbf{q}} & \mathbf{q} \in Q^{\mathrm{Pd}} \\ T_{\rho(\mathbf{q})} & \mathbf{q} \in Q^{\mathrm{Fu}} \wedge \rho(\mathbf{q}) \notin \mathrm{G}^{\mathrm{pl}} \\ \mathrm{II}_{\rho(\mathbf{q})} & \mathbf{q} \in Q^{\mathrm{Fu}} \wedge \rho(\mathbf{q}) \in \mathrm{G}^{\mathrm{pl}} \end{cases}$$

Lastly, the resource demand for a $\mathbf{q} \in Q$ comes either from parameters $N_{\mathbf{q},\mathsf{r}}$ for predefined operator types, or is derived from allocation domain $\alpha(\rho(\mathbf{q}))$ in which the referenced graph is embedded.

$$n_{\mathbf{q},\mathsf{r}} = \begin{cases} N_{\mathbf{q},\mathsf{r}} & \mathbf{q} \in Q^{\mathrm{Pd}} \\ n_{\alpha(\rho(\mathbf{q})),\mathsf{r}} & \mathbf{q} \in Q^{\mathrm{Fu}} \end{cases} \quad \forall \mathsf{r} \in R$$

To summarise, a solution to the problem consists of a schedule for each graph, an II for each pipelined graph, and the allocation in each allocation domain:

$$\begin{aligned} &t_i && \forall i \in O_{\mathrm{g}},\ \forall \mathrm{g} \in \mathrm{G} && \text{(schedule)} \\ &\mathrm{II}_{\mathrm{g}} && \forall \mathrm{g} \in \mathrm{G}^{\mathrm{pl}} && \text{(IIs)} \\ &a_{\mathbf{q}} && \forall \mathbf{q} \in Q_A,\ \forall A \in \mathcal{AD} && \text{(allocation)} \end{aligned}$$

A feasible solution must honour all precedence constraints expressed by the dependence edges (1), ensure that no operator type is oversubscribed at any time (2), and obey the given resource constraints for the outermost allocation domain (3).

$$\forall (i \to j) \in E_{\mathrm{g}},\ \forall \mathrm{g} \in \mathrm{G} : \quad \begin{cases} t_i + l_i \le t_j & \mathrm{g} \notin \mathrm{G}^{\mathrm{pl}} \\ t_i + l_i \le t_j - d_{ij} \cdot \mathrm{II}_{\mathrm{g}} & \mathrm{g} \in \mathrm{G}^{\mathrm{pl}} \end{cases} \tag{1}$$

$$\forall \mathbf{q} \in Q_A,\ \forall \mathrm{g} \in \mathrm{G}_A,\ \forall A \in \mathcal{AD} : \quad \begin{cases} |\{i \in O^{\mathbf{q}}_{\mathrm{g}} : t_i \le x < t_i + b_{\mathbf{q}}\}| & \\ \quad \le a_{\mathbf{q}} \quad \forall x \in [0, T_{\mathrm{g}}] & \mathrm{g} \notin \mathrm{G}^{\mathrm{pl}} \\ |\{i \in O^{\mathbf{q}}_{\mathrm{g}} : x \in \{(t_i + \beta) \bmod \mathrm{II}_{\mathrm{g}} : 0 \le \beta < b_{\mathbf{q}}\}\}| & \\ \quad \le a_{\mathbf{q}} \quad \forall x \in [0, \mathrm{II}_{\mathrm{g}} - 1] & \mathrm{g} \in \mathrm{G}^{\mathrm{pl}} \end{cases} \tag{2}$$

$$\forall \mathsf{r} \in R : \quad n_{\alpha(\mathrm{g}^{\mathrm{top}}),\mathsf{r}} \le \bar{N}_{\mathsf{r}} \tag{3}$$

As per our problem statement, the objective is to **minimise** $T_{\mathrm{g}^{\mathrm{top}}}$.

C. Compatibility with Vivado HLS

Vivado HLS imposes two additional restrictions on the nesting of pipelined graphs.

$$O^{\mathrm{Lo}}_{\mathrm{g}} = \emptyset \quad \forall \mathrm{g} \in \mathrm{G}^{\mathrm{pl}} \tag{4}$$

$$b_{\sigma(i)} \mid \mathrm{II}_{\mathrm{g}} \quad \forall i \in O^{\mathrm{Ca}}_{\mathrm{g}},\ \forall \mathrm{g} \in \mathrm{G}^{\mathrm{pl}} \tag{5}$$

First, pipelined loops cannot contain other loops (4). Secondly, the blocking times of all function operator types used in a pipelined graph must divide the graph's II (5). Note that all operator types predefined in Vivado HLS' library are fully pipelined, i.e. they have a blocking time of 1.

In the implementation of SkyCastle, we handle the chaining of combinatorial operations and the accesses to on-chip memory and AXI ports in a way that faithfully reproduces Vivado HLS' behaviour. As a concession to the clarity of this paper, these implementation details are omitted here.

IV. Resource-aware multi-loop scheduler

In general, the MLS problem is not a linear optimisation problem. However, with SkyCastle, we propose a solution approach using integer linear programming (ILP) that is capable of handling a realistic, non-trivial subclass of kernels, as will be demonstrated in Section V. Currently, kernels need to be legal for Vivado HLS compilation, and may contain multiple, optionally pipelined loops that may call multiple pipelined functions.

The basic idea is to solve the problem hierarchically, i.e. one allocation domain at a time, instead of modelling it all at once. This requires that we precompute a set of solutions for the nested allocation domains. The selection of a particular solution for each function operator type is then modelled as part of the SkyCastle ILP, alongside all scheduling problems for the loops in the current function.

A. Precomputing solutions

Due to the preconditions stated above, function operator types are leaf nodes in the problem structure, i.e. they contain neither loop nor call operations. In consequence, we can use the iterative exploration methods presented in [5], using the resource-aware version of the formulation by Eichenberger and Davidson [6], to compute a set of solutions $\mathcal{S}_{\mathbf{q}}$ for $\mathbf{q} \in Q^{\text{Fu}}$.

A solution $S \in \mathcal{S}_{\mathbf{q}}$ is computed by minimising the resource demand for a candidate interval $\text{II}_{\mathbf{q}}^{S}$, then fixing the resulting allocation, and finally minimising the latency. S is characterised by its blocking time $B_{\mathbf{q}}^{S} = \text{II}_{\mathbf{q}}^{S}$, latency $L_{\mathbf{q}}^{S}$, and resource demand $N_{\mathbf{q},\mathsf{r}}^{S}$ for each low-level resource type $\mathsf{r} \in R$.

Per definition in [5], the set $\mathcal{S}_{\mathbf{q}}$ contains only solutions that are Pareto-optimal with regards to the II and the resource demand. However, in order to prevent trivially infeasible MLSP instances due to the blocking time constraints (5), we compute additional solutions to ensure that $\mathcal{S}_{\mathbf{q}}$ and $\mathcal{S}_{\mathbf{q}'}$ contain solutions for the same set of IIs if $\mathbf{q}$ and $\mathbf{q}' \in Q^{\text{Fu}}$ occur together in at least one pipelined graph.

B. Bounds

A priori to constructing the ILP, bounds (6)–(9) are computed from the problem parameters and the precomputed solutions. Due to the page limit, we refer the reader to the more detailed discussion of bounds in previous work [5], [7]. In general, we extended the bound definitions to handle the hierarchical nature of our problem, e.g. when computing the minimum latency $T_{\mathrm{g}}^{\perp}$ of a graph, we use lower bound approximations for the latencies of loop and call operations. Note that the typical definition of the lower bound interval $\text{II}_{\mathrm{g}}^{\perp}$ needs to include the maximum blocking time of all operator types used in g as a third component.

$$a_{\mathbf{q}}^{\perp} \leq a_{\mathbf{q}} \leq a_{\mathbf{q}}^{\top} \quad \forall \mathbf{q} \in Q_A^{\text{Sh}} \tag{6}$$

$$T_{\mathrm{g}}^{\perp} \leq T_{\mathrm{g}} \leq T_{\mathrm{g}}^{\top} \quad \forall \mathrm{g} \in \mathrm{G}_A \tag{7}$$

$$\text{II}_{\mathrm{g}}^{\perp} \leq \text{II}_{\mathrm{g}} \leq \text{II}_{\mathrm{g}}^{\top} \quad \forall \mathrm{g} \in \mathrm{G}_A^{\text{pl}} \tag{8}$$

$$B_{\mathbf{q}}^{\perp} \leq b_{\mathbf{q}} \leq B_{\mathbf{q}}^{\top} \quad \forall \mathbf{q} \in Q_A^{\text{Fu}} \tag{9}$$

C. SkyCastle ILP formulation

We now develop an ILP formulation for one allocation domain A of the MLSP. To that end, we combine modelling techniques previously presented in [5] and [7] with the novel capability to handle the multi-variant loop operations and function operator types. As a side product, this work also extends the Moovac formulation [7] to support operator types with blocking times > 1.

We introduce the formulation part by part in the following sections. The complete ILP consists of the objective (23), subject to the constraints (24)–(44) and the domain constraints (10)–(22).

1) Decision variables: Our formulation uses the decision variables defined in (10)–(16) to model the corresponding components in the problem definition in Section III-B. Note that we only introduce variables where needed, and implicitly treat the remainder of the problem components as parameters. For example, exclusive operators have a constant allocation of $a_{\mathbf{q}} = \sum_{\mathrm{g} \in \mathrm{G}_A} |O_{\mathrm{g}}^{\mathbf{q}}|$, and only function operator types require a variable blocking time $b_{\mathbf{q}}$, whereas we have $b_{\mathbf{q}} = 1$ for all predefined types.

$$n_{A,\mathsf{r}} \in \mathbb{N}_0 \quad \forall \mathsf{r} \in R \tag{10}$$

$$a_{\mathbf{q}} \in \mathbb{N} \quad \forall \mathbf{q} \in Q_A^{\text{Sh}} \tag{11}$$

$$b_{\mathbf{q}} \in \mathbb{N} \quad \forall \mathbf{q} \in Q_A^{\text{Fu}} \tag{12}$$

$$t_i \in \mathbb{N}_0 \quad \forall i \in O_{\mathrm{g}},\ \forall \mathrm{g} \in \mathrm{G}_A \tag{13}$$

$$l_i \in \mathbb{N}_0 \quad \forall i \in O_{\mathrm{g}}^{\text{Lo}} \cup O_{\mathrm{g}}^{\text{Ca}},\ \forall \mathrm{g} \in \mathrm{G}_A \tag{14}$$

$$T_{\mathrm{g}} \in \mathbb{N}_0 \quad \forall \mathrm{g} \in \mathrm{G}_A \tag{15}$$

$$\text{II}_{\mathrm{g}} \in \mathbb{N}_0 \quad \forall \mathrm{g} \in \mathrm{G}_A^{\text{pl}} \tag{16}$$

Our formulation uses the following additional, internal decision variables.

$$s_{\mathbf{q}}^{S} \in \{0,1\} \quad \forall S \in \mathcal{S}_{\mathbf{q}},\ \forall \mathbf{q} \in Q_A^{\text{Fu}} \tag{17}$$

$$l_{\mathbf{q}} \in \mathbb{N}_0 \quad \forall \mathbf{q} \in Q_A^{\text{Fu}} \tag{18}$$

$$\dot{n}_{\mathbf{q},\mathsf{r}} \in \mathbb{N}_0 \quad \forall \mathbf{q} \in Q_A^{\text{Fu}},\ \forall \mathsf{r} \in R \tag{19}$$

$$\boldsymbol{\text{II}}_{\mathrm{g},x} \in \{0,1\} \quad \forall x \in [\boldsymbol{\text{II}}_{\mathrm{g}}^{\perp}, \boldsymbol{\text{II}}_{\mathrm{g}}^{\top}],\ \forall \mathbf{g} \in \mathbf{G}_A^{\text{pl}} \tag{20}$$

$$\begin{aligned} y_i \in \mathbb{N}_0,\ m_i \in \mathbb{N}_0 \quad & \forall i \in O_{\mathrm{g}} : i \notin O_{\mathrm{g}}^{\text{Lo}} \wedge \sigma(i) \in Q_A^{\text{Sh}} \\ & \forall \mathbf{g} \in \mathbf{G}_A^{\text{pl}} \end{aligned} \tag{21}$$

$$\begin{aligned} \mu_{ij}^{1}, \mu_{ij}^{2}, \mu_{ij}^{3}, \mu_{ij}^{4}, \quad & \forall i,j \in O_{\mathrm{g}}^{\mathbf{q}} : i < j,\ \forall \mathbf{q} \in Q_A^{\text{Sh}} \\ \xi_{ij} \in \{0,1\} \quad & \forall \mathbf{g} \in \mathbf{G}_A \end{aligned} \tag{22}$$

(17) model that a precomputed solution is selected for a function operator type, whose variable latency is stored in (18). (19) represent the accumulated resource demand for all allocated instances of a function operator type. (20) correspond to a particular value of a graph's II. (21) are used to decompose the start time of an operation i in a pipelined graph g as $t_i = y_i * \text{II}_{\mathrm{g}} + m_i$. We call m_i the modulo slot, i.e. the congruence class modulo the graph's II. (22) help to govern the maximum concurrent use of the shared operator types. As we only need to define these variables for unique pairs of operations, we assume the presence of an arbitrary total order $<$ among the operations.

2) Objective: We consider a tuple of objectives (23), and optimise lexicographically. The primary objective is, as in the general MLSP, to minimise $T_{\mathrm{g}^{\text{top}}}$, the latency of the top-level graph. As a practical consideration, we seek to find the most resource-efficient solution in case multiple solutions achieving the shortest possible latency exist. To that end, the secondary objective is to minimise the accumulated, weighted resource demand as in [5].

$$\textbf{min} \left(T_{\mathrm{g}^{\text{top}}}, \quad \frac{1}{|R|} \sum_{\mathsf{r} \in R} \frac{n_{A,\mathsf{r}}}{\bar{N}_{\mathsf{r}}} \right) \tag{23}$$

3) Resource constraints: (24) model the MLSP's resource constraints (3) for a resource type $\mathsf{r} \in R$. Using the $\dot{n}_{\mathbf{q},\mathsf{r}}$-variables here avoids the product of decision variables.

$$n_{A,\mathsf{r}} = \sum_{\mathbf{q} \in Q_A^{\text{Pd}}} a_{\mathbf{q}} \cdot n_{\mathbf{q},\mathsf{r}} + \sum_{\mathbf{q} \in Q_A^{\text{Fu}}} \dot{n}_{\mathbf{q},\mathsf{r}} \leq \bar{N}_{\mathsf{r}} \tag{24}$$

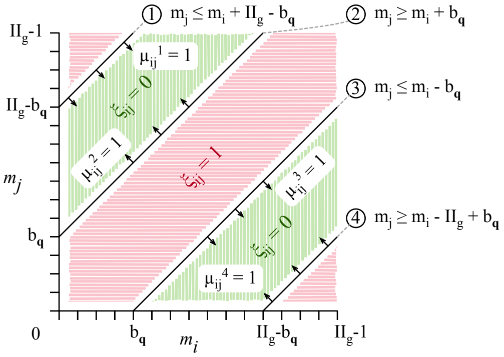
① $m_j \le m_i + \mathrm{II}_g - b_{\mathbf{q}}$
② $m_j \ge m_i + b_{\mathbf{q}}$
③ $m_j \le m_i - b_{\mathbf{q}}$
④ $m_j \ge m_i - \mathrm{II}_g + b_{\mathbf{q}}$
II_g-1
II_g-$b_{\mathbf{q}}$
$b_{\mathbf{q}}$
m_j
0
$b_{\mathbf{q}}$
m_i
II_g-$b_{\mathbf{q}}$
II_g-1
$\mu_{ij}{}^1 = 1$
$\mu_{ij}{}^2 = 1$
$\mu_{ij}{}^3 = 1$
$\mu_{ij}{}^4 = 1$
$\xi_{ij} = 0$
$\xi_{ij} = 1$
$\xi_{ij} = 0$

Figure 2. Interaction of the m, μ and ξ variables in the modelling of the operator constraints

4) Allocation constraints: In order to satisfy constraint (2) in the MLSP, we must ensure that that no more than $a_{\mathbf{q}}$ instances of a shared operator type $\mathbf{q} \in Q_A^{\mathrm{Sh}}$ are used at any time by the operations of a graph $\mathrm{g} \in \mathrm{G}_A$. We say two operations i, j are in conflict, indicated by $\xi_{ij} = 1$, if they cannot use the same **q**-instance due to overlapping blocking times. As this relation is symmetric, we only encode it for $i < j$. (25) express that we need to allocate at least one **q**-instance more than the maximum number of conflicts any operation is part of.

$$\sum_{j \in O_{\mathrm{g}}^{\mathbf{q}} : i<j} \xi_{ij} + \sum_{j \in O_{\mathrm{g}}^{\mathbf{q}} : i>j} \xi_{ji} \le a_{\mathbf{q}} - 1 \quad \forall i \in O_{\mathrm{g}}^{\mathbf{q}} \tag{25}$$

5) Selection of solutions: (26)-(28) are indicator constraints [18] that bind the latency, blocking time and resource demand of a function operator type $\mathbf{q} \in Q_A^{\mathrm{Fu}}$ to the respective values of a precomputed solution. (29) enforce that at least one solution is picked.

$$s_{\mathbf{q}}^S = 1 \rightarrow l_{\mathbf{q}} = L_{\mathbf{q}}^S \quad \forall S \in \mathcal{S}_{\mathbf{q}} \tag{26}$$
$$s_{\mathbf{q}}^S = 1 \rightarrow b_{\mathbf{q}} = B_{\mathbf{q}}^S \quad \forall S \in \mathcal{S}_{\mathbf{q}} \tag{27}$$
$$s_{\mathbf{q}}^S = 1 \rightarrow \dot{n}_{\mathbf{q},\mathrm{r}} = a_{\mathbf{q}} \cdot N_{\mathbf{q},\mathrm{r}}^S \quad \forall S \in \mathcal{S}_{\mathbf{q}},\ \forall \mathrm{r} \in R \tag{28}$$
$$\sum_{S \in \mathcal{S}_{\mathbf{q}}} s_{\mathbf{q}}^S = 1 \tag{29}$$

6) Variable latencies: (30)–(32) propagate the variable latencies of the loop and call operations in every graph $\mathrm{g} \in \mathrm{G}_A$. (33) define the graph's latency.

$$l_i = c_{\rho(i)} \cdot T_{\rho(i)} \quad \forall i \in O_{\mathrm{g}}^{\mathrm{Lo}} \wedge \mathrm{g} \notin \mathrm{G}^{\mathrm{pl}} \tag{30}$$
$$l_i = (c_{\rho(i)} - 1) \cdot \mathrm{II}_{\rho(i)} + T_{\rho(i)} \quad \forall i \in O_{\mathrm{g}}^{\mathrm{Lo}} \wedge \mathrm{g} \in \mathrm{G}^{\mathrm{pl}} \tag{31}$$
$$l_i = l_{\sigma(i)} \quad \forall i \in O_{\mathrm{g}}^{\mathrm{Ca}} \tag{32}$$
$$t_i + l_i \le T_{\mathrm{g}} \quad \forall i \in O_{\mathrm{g}} \tag{33}$$

7) Scheduling problems: Every pipelined graph $\mathrm{g} \in \mathrm{G}_A^{\mathrm{pl}}$ implies one modulo scheduling problem. We adopt the already linear precedence constraints (1) in the MLSP as (34).

$$t_i + l_i \le t_j + d_{ij} \cdot \mathrm{II}_{\mathrm{g}} \quad \forall (i \rightarrow j) \in E_{\mathrm{g}} \tag{34}$$

(35) define the binary variable $\mathrm{II}_{\mathrm{g},x}$ to be 1 iff the value of II_{g} is x. Using these variables and multiple indicator constraints (36), we linearise the decomposition of an operation's start time into a multiple of the II and the modulo slot. (37) models Vivado HLS' constraint (5) regarding the blocking times of function operator types: For each solution S, we determine a set of viable IIs for g that restrict the feasible values for II_{g} if S is selected.

$$\sum_{\mathrm{II}_{\mathrm{g}}^{\perp} \le x \le \mathrm{II}_{\mathrm{g}}^{\top}} \mathrm{II}_{\mathrm{g},x} = 1 \quad \sum_{\mathrm{II}_{\mathrm{g}}^{\perp} \le x \le \mathrm{II}_{\mathrm{g}}^{\top}} x \cdot \mathrm{II}_{\mathrm{g},x} = \mathrm{II}_{\mathrm{g}} \tag{35}$$
$$\mathrm{II}_{\mathrm{g},x} = 1 \rightarrow t_i = y_i \cdot x + m_i \quad \forall x \in [\mathrm{II}_{\mathrm{g}}^{\perp}, \mathrm{II}_{\mathrm{g}}^{\top}]$$
$$\forall i \in O_{\mathrm{g}}^{\mathbf{q}},\ \forall \mathbf{q} \in Q_A^{\mathrm{Sh}} \tag{36}$$
$$\sum_{x \in [\mathrm{II}_{\mathrm{g}}^{\perp}, \mathrm{II}_{\mathrm{g}}^{\top}] : B_{\mathbf{q}}^S \mid x} \mathrm{II}_{\mathrm{g},x} \ge s_{\mathbf{q}}^S \quad \forall S \in \mathcal{S}_{\mathbf{q}},\ \forall \mathbf{q} \in Q_A^{\mathrm{Fu}} \tag{37}$$

The following bounds help to restrict the ILP solution space further: (38) encode the II-dependent minimum allocation [5] for the shared operator types. (39) mandate that II_{g} is greater or equal to the longest blocking time of any selected function operator type.

$$a_{\mathbf{q}} \cdot x \ge |O_{\mathrm{g}}^{\mathbf{q}}| \cdot B_{\mathbf{q}}^{\perp} \cdot \mathrm{II}_{\mathrm{g},x} \quad \forall x \in [\mathrm{II}_{\mathrm{g}}^{\perp}, \mathrm{II}_{\mathrm{g}}^{\top}],\ \forall \mathbf{q} \in Q_A^{\mathrm{Sh}} \tag{38}$$
$$\mathrm{II}_{\mathrm{g}} \ge s_{\mathbf{q}}^S \cdot B_{\mathbf{q}}^S \quad \forall S \in \mathcal{S}_{\mathbf{q}},\ \forall \mathbf{q} \in Q_A^{\mathrm{Fu}} \tag{39}$$

Lastly, (40)–(44) define the conflict variables for each unique pair of operations $i, j \in O_{\mathrm{g}}^{\mathbf{q}} : i < j$, using the same shared type $\mathbf{q} \in Q_A^{\mathrm{Sh}}$. Figure 2 illustrates the space of possible modulo slot assignments for i and j. The green areas enclose non-conflicting assignments, i.e. both operations can use the same operator, and we set $\xi_{ij} = 0$ per definition. All other assignments in the red areas result in overlapping blocking times in either the same or adjacent iterations, which prevent i and j from sharing one **q**-instance. The conflict is expressed as $\xi_{ij} = 1$.

The different areas are bounded by four inequalities between m_i and m_j, as visualised by the lines in Figure 2. The four binary μ_{ij}-variables correspond to these inequalities, and are defined by constraints (40)–(43) to be equal to 1 if the respective inequality is satisfied. Our implementation uses additional constraints (not shown here) that bind the variables to 0 if the negation of the respective inequality is fulfilled. We then define the conflict variable ξ_{ij} by simply counting the number of satisfied inequalities in (44).

$$\mu_{ij}^1 = 1 \rightarrow \ m_j \le m_i + \mathrm{II}_{\mathrm{g}} - b_{\mathbf{q}} \tag{40}$$
$$\mu_{ij}^2 = 1 \rightarrow \ m_j \ge m_i + b_{\mathbf{q}} \tag{41}$$
$$\mu_{ij}^3 = 1 \rightarrow \ m_j \le m_i - b_{\mathbf{q}} \tag{42}$$
$$\mu_{ij}^4 = 1 \rightarrow \ m_j \ge m_i - \mathrm{II}_{\mathrm{g}} + b_{\mathbf{q}} \tag{43}$$
$$3 - \xi_{ij} \le \mu_{ij}^1 + \mu_{ij}^2 + \mu_{ij}^3 + \mu_{ij}^4 \le 3 \tag{44}$$

Note that for the common case of $b_{\mathbf{q}} = 1$, inequalities ① and ④ are always satisfied. Therefore, the associated μ-variables and their definitions are dropped, and (44) are simplified to $1 - \xi_{ij} \le \mu_{ij}^2 + \mu_{ij}^3 \le 1$.

We use non-modulo equivalents of (34), (41)–(42) and (44) to model the resource-constrained scheduling problems imposed by the non-pipelined graphs.

V. Case studies

A. Kernels

In the following sections, we introduce our three kernels SPN, FFT and LULESH. Figure 3 illustrates the nesting structure of the underlying MLS problems. The dependence graphs itself are not shown, however we outline which shared operator types and memory ports are used, and how many users they have.

1) SPN: Sum-Product Networks (SPN) [19] are a relatively young type of deep machine-learning models from the class of *Probabilistic Graphical Models* (PGM), for which inference has been successfully accelerated in prior work [20]. An SPN captures the joint probability distribution of its input variables in the form of a directed, acyclic graph. The graph comprises three different kinds of nodes: Weighted sums, products and, as leaf nodes, univariate distributions, which can for example be modelled as histograms [21].

The SPN kernel, as outlined in Listing 1, combines three inference processes for an example SPN from the NIPS corpus [22]: Assuming given values for the input variables i_1 to i_4 we are interested in finding the most probable explanation for the missing input feature i_5 in a first step. For this purpose, we iterate over all 256 possible values of i_5 and evaluate the SPN (loop `MPE`, which has been manually unrolled by a factor 4 and is pipelined). After that, we marginalise [19] input variable i_1, and compute a conditional probability using the most probable explanation for i_5. SPN is a small kernel, but not memory-bound, and therefore is well suitable to demonstrate the benefits of accelerator replication.

2) FFT: Our second kernel, FFT, is the `fft/transpose` benchmark from MachSuite [23]. One invocation processes a 512 byte chunk of input. We wrapped the `FFT8` macro in a function `fft8`, and disabled inlining for it as well as for the `twiddles8` function. The top-level function contains 11 loops in total, out of which three loops are pipelined and call either one of both of the functions. FFT therefore challenges the scheduler to obey the blocking time constraint (5).

3) LULESH: Our LULESH kernel represents one iteration in the `CalcFBHourglassForceForElems` function from the serial version of the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics proxy application [24]. In order to make the code compatible with Vivado HLS, we hardcoded dynamic array sizes to the default values in the application. We replaced the cubic root function by the power of $\frac{1}{3}$, as not even one `cbrt` operator would fit on the XC7Z020 device together with the minimal allocation of the other operator types. In order to obtain a best-effort HLS version of the code, we inlined the `CalcElemFBHourglassForce` function and restructured the loops in it. Additionally, we extracted common functionality into new functions `calcHM` and `calcHxx`. The resulting three loops and two functions are all pipelined. This kernel contains the most complex allocation problem in our case studies, as non-trivial computations in the loops compete with the variable allocation and solution selection of function operators.

B. Experimental setup

Our current SkyCastle implementation considers look-up tables (LUT), flip-flops (FF), block RAM (BRAM) and DSP slices, i.e. the typical low-level resource types on Xilinx devices. We target the ZedBoard (XC7Z020: 53,200 LUT; 106,400 FF; 280 BRAM; 220 DSP) at 100 MHz and the VCU108 evaluation board (XCVU095: 537,600; 1,075,200; 3,456; 768) at 200 MHz, and compose bitstreams for complete SoC designs, comprised of one or more accelerators, with TaPaSCo 2019.6 [3] and Vivado 2018.3.

In order to accommodate TaPaSCo's SoC template, as well as to give the logic synthesis tools some headroom, we make 85% (ZedBoard) respectively 70% (VCU108, more complex template due to PCIe interface) of the resources available to the allocation of operators during scheduling. The MLSP instances are extracted from Vivado HLS 2018.3 operating with medium effort levels for scheduling and binding, and using a target cycle time of 4 ns (VCU108) or 8 ns (ZedBoard) without clock uncertainty. We marked loops and functions for pipelining without specifying the II.

SkyCastle uses the Gurobi 8.1 ILP solver, which was allowed to use up to 8 threads and 16 GB RAM per kernel. The experiments were performed on 2×12-core Intel Xeon E5-2680 v3 systems running at 2.8 GHz with 64 GiB RAM. We set time limits of 15 min for the primary objective (minimisation of the latency, cf. (23)), and 5 min for the secondary objective (subsequent minimisation of the resource demand). The same limits were in place for the computation of the solutions for the function operator types, but please recall that the primary objective is to minimise the resource demand here. If the solver is unable to prove optimality within the time limit, we accept the feasible solution, and record the optimality gap, relative to the solver's best lower bound.

For the larger VCU108 board, we explore the possible replication of kernels by scheduling with $\frac{\bar{N}_\mathsf{r}}{k}$ available elements of a resource type $\mathsf{r} \in R$, for $1 \leq k \leq 8$.

For the configurations labeled "SC-xk", we computed a solution adhering to the resource constraints under replication factor k with SkyCastle, emitted pipeline and allocation directives accordingly, and ran Vivado HLS again with them. The configuration labeled "VHLS" denotes the baseline maximum performance that Vivado HLS constructs without the SkyCastle optimisation.

C. Results

Table I summarises the high- and low-level synthesis results. Column "Latency" shows the latency (in cycles) of one activation of the kernel's top-level function, as reported by Vivado HLS. SkyCastle's estimation of the Vivado HLS' cycle count (not shown) is very precise and differs by at most 2.4%, which shows that we model enough of Vivado HLS' scheduling peculiarities to meaningfully tackle the problem. The next column "Util." extracts the utilisation of DSP slices, which were always the scarcest resource type in our evaluation, from the HLS report. SkyCastle's estimation of DSP slices is almost perfect, and is off by at most three, because Vivado

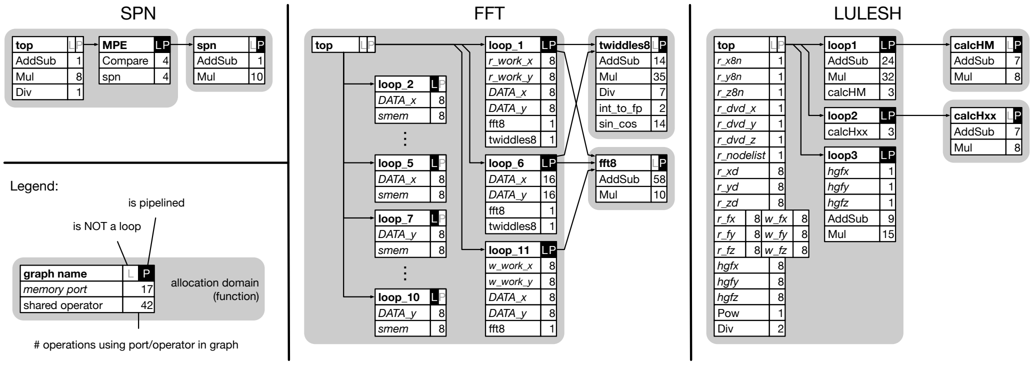

Figure 3. MLS problem structures for the case studies

HLS appears to ignore the allocation directives for the combined floating-point ADD/SUB core in some situations. The estimation error for LUTs and FFs is below 10%, but reaches up to 70% for BRAMs. The reason for the high deviation in the latter case is that the majority of BRAM is used by components that are not operators themselves, and thus do not occur in MLSP. However, as mentioned above, the BRAM utilisation was never crucial in our experiments.

The remaining columns characterise results of composing a bitstream comprised of "# Acc."-many accelerators. Most importantly, column "Freq." shows SkyCastle accomplished its mission: While neither FFT nor LULESH fit on the devices with the default VHLS flow, we computed synthesisable configurations for up to four replica. For both kernels, the scheduler determined $k = 5$ to be infeasible even with maximum resource sharing. SPN does fit once one the larger device with the default flow, but this configuration cannot be replicated. Again, all SkyCastle configurations yielded working *multi-accelerator* designs. The last column, "Throughp." states the theoretical throughput achievable with each multi-accelerator design, calculated as $\left(\frac{\text{\# Acc.}}{\text{Latency}} \cdot \text{Freq.}\right)$. When viewed together with the column "Latency", the benefits of scheduling for better replicability become apparent. For all three kernels, SC-x2 already yields a *better* throughput than the *maximum performance* VHLS configuration. SPN reaches its theoretical peak performance with a 7-way accelerator, whereas the other kernels profit from any additional replication.

The biggest challenge for SkyCastle was to schedule LULESH for the 2-way accelerator design. The feasible solution had an optimality gap of 5.2% after optimising the primary objective for 15 min, and a gap of 0.09% remained after 5 min spent on the secondary objective. In all other cases, the ILP solver either returned optimal solutions, or the remaining optimality gap was in the same ballpark as the inaccuracies in the latency estimation. Note that we computed the solutions for the function operator types only once per exploration of the replication factors. Altogether, using the aforementioned time limits, the entire process took 51 minutes for LULESH targeting the VCU108, and well below 20 minutes for the other configurations.

Table I
SCHEDULING AND SYSTEM COMPOSITION RESULTS

Kernel	Board	Config	HLS		Composition		
			Latency [Cyc.]	Util. [%]	# Acc.	Freq. [MHz]	Throughp. [1/µs, theo.]
SPN	ZedBoard	VHLS	175	226.8	1	*failed*	
		SC-x1	366	76.8	1	100	0.55
	VCU108	VHLS	212	65.0	1	200	0.94
					2	*failed*	
		SC-x1	277	36.3	1	200	0.72
		SC-x2	278	33.5	2	200	1.44
		SC-x3	402	22.0	3	200	1.49
		SC-x4	408	16.9	4	200	1.96
		SC-x5	659	12.6	5	200	1.52
		SC-x6	663	10.8	6	200	1.81
		SC-x7	665	9.4	7	200	2.11
		SC-x8	787	8.3	8	200	2.03
FFT	ZedBoard	VHLS	4479	883.2	1	*failed*	
		SC-x1	5534	82.7	1	100	0.02
	VCU108	VHLS	4682	247.5	1	*failed*	
		SC-x1	4700	64.7	1	155	0.03
		SC-x2	4918	34.2	2	159	0.06
		SC-x3	5721	23.3	3	194	0.10
		SC-x4	6641	17.3	4	187	0.11
LULESH	ZedBoard	VHLS	533	528.2	1	*failed*	
		SC-x1	656	82.7	1	100	0.15
	VCU108	VHLS	610	150.4	1	*failed*	
		SC-x1	622	69.3	1	200	0.32
		SC-x2	681	34.4	2	200	0.59
		SC-x3	745	22.8	3	200	0.81
		SC-x4	863	17.7	4	200	0.93

VI. CONCLUSION AND OUTLOOK

We formalised a novel, general scheduling and allocation model for the common problem of minimising the latency of a complex HLS kernel subject to low-level resource constraints. This model is the foundation for SkyCastle, our proposed resource-aware multi-loop scheduler, which currently handles a subset of kernels compatible with Xilinx Vivado HLS.

In the future, we plan to investigate improvements or alternatives to the precomputation of solutions for the function operators, which we believe will allow us to treat an arbitrary nesting structure in a uniform way. Also, our approach would benefit tremendously from a vendor-supported, high-level synthesis counterpart to the XDL interface [25], as we currently can only feed the II and the operator allocation back to Vivado HLS in the form of directives. Should such an interface become available in the future, SkyCastle could be easily adapted to replace the built-in scheduler.

Acknowledgements

The experiments for this research were conducted on the Lichtenberg high-performance computing cluster at TU Darmstadt. The authors would like to thank Xilinx, Inc. for supporting their work by hardware and software donations.

References

[1] L. Solis-Vasquez and A. Koch, "A case study in using opencl on fpgas: Creating an open-source accelerator of the autodock molecular docking software," in *Fifth International Workshop on FPGAs for Software Programmers (FSP)*, 2018.

[2] Z. Jin and H. Finkel, "Evaluating LULESH kernels on opencl FPGA," in *Applied Reconfigurable Computing - 15th International Symposium, ARC 2019, Darmstadt, Germany, April 9-11, 2019, Proceedings*, ser. Lecture Notes in Computer Science, C. Hochberger, B. Nelson, A. Koch, R. F. Woods, and P. C. Diniz, Eds., vol. 11444. Springer, 2019, pp. 199–213. [Online]. Available: https://doi.org/10.1007/978-3-030-17227-5_15

[3] J. Korinth, J. Hofmann, C. Heinz, and A. Koch, "The tapasco open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems," in *Applied Reconfigurable Computing - 15th International Symposium, ARC 2019, Darmstadt, Germany, April 9-11, 2019, Proceedings*, ser. Lecture Notes in Computer Science, C. Hochberger, B. Nelson, A. Koch, R. F. Woods, and P. C. Diniz, Eds., vol. 11444. Springer, 2019, pp. 214–229. [Online]. Available: https://doi.org/10.1007/978-3-030-17227-5_16

[4] H. Wang, P. Thiagaraj, and O. Sinnen, "Combining multiple optimised FPGA-based pulsar search modules using OpenCL," *Journal of Astronomical Instrumentation*, 2019.

[5] J. Oppermann, P. Sittel, M. Kumm, M. Reuter-Oppermann, A. Koch, and O. Sinnen, "Design-space exploration with multi-objective resource-aware modulo scheduling," in *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*, ser. Lecture Notes in Computer Science, R. Yahyapour, Ed., vol. 11725. Springer, 2019, pp. 170–183. [Online]. Available: https://doi.org/10.1007/978-3-030-29400-7_13

[6] A. E. Eichenberger and E. S. Davidson, "Efficient Formulation for Optimal Modulo Schedulers," in *Proc. of the ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation, Las Vegas, USA*, 1997.

[7] J. Oppermann, M. Reuter-Oppermann, L. Sommer, A. Koch, and O. Sinnen, "Exact and practical modulo scheduling for high-level synthesis," *TRETS*, vol. 12, no. 2, pp. 8:1–8:26, 2019. [Online]. Available: https://doi.org/10.1145/3317670

[8] M. Kudlur, K. Fan, and S. A. Mahlke, "Streamroller: : automatic synthesis of prescribed throughput accelerator pipelines," in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2006, Seoul, Korea, October 22-25, 2006*, R. A. Bergamaschi and K. Choi, Eds. ACM, 2006, pp. 270–275. [Online]. Available: https://doi.org/10.1145/1176254.1176321

[9] J. Cong, M. Huang, B. Liu, P. Zhang, and Y. Zou, "Combining module selection and replication for throughput-driven streaming programs," in *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, W. Rosenstiel and L. Thiele, Eds. IEEE, 2012, pp. 1018–1023. [Online]. Available: https://doi.org/10.1109/DATE.2012.6176645

[10] J. Cong, M. Huang, and P. Zhang, "Combining computation and communication optimizations in system synthesis for streaming applications," in *The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014*, V. Betz and G. A. Constantinides, Eds. ACM, 2014, pp. 213–222. [Online]. Available: https://doi.org/10.1145/2554688.2554771

[11] P. Li, P. Zhang, L. Pouchet, and J. Cong, "Resource-aware throughput optimization for high-level synthesis," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*, G. A. Constantinides and D. Chen, Eds. ACM, 2015, pp. 200–209. [Online]. Available: https://doi.org/10.1145/2684746.2689065

[12] K. Fan, M. Kudlur, H. Park, and S. A. Mahlke, "Cost sensitive modulo scheduling in a loop accelerator synthesis system," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-38 2005), 12-16 November 2005, Barcelona, Spain*. IEEE Computer Society, 2005, pp. 219–232. [Online]. Available: https://doi.org/10.1109/MICRO.2005.17

[13] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: a high-level performance analysis tool for fpga-based accelerators," in *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*. ACM, 2016, pp. 136:1–136:6. [Online]. Available: https://doi.org/10.1145/2897937.2898040

[14] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017*, S. Parameswaran, Ed. IEEE, 2017, pp. 430–437. [Online]. Available: https://doi.org/10.1109/ICCAD.2017.8203809

[15] F. Ferrandi, P. L. Lanzi, D. Loiacono, C. Pilato, and D. Sciuto, "A multi-objective genetic algorithm for design space exploration in high-level synthesis," in *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2008, 7-9 April 2008, Montpellier, France*. IEEE Computer Society, 2008, pp. 417–422. [Online]. Available: https://doi.org/10.1109/ISVLSI.2008.73

[16] A. Prost-Boucle, O. Muller, and F. Rousseau, "Fast and standalone design space exploration for high-level synthesis under resource constraints," *Journal of Systems Architecture - Embedded Systems Design*, vol. 60, no. 1, pp. 79–93, 2014. [Online]. Available: https://doi.org/10.1016/j.sysarc.2013.10.002

[17] B. C. Schäfer, "Probabilistic multiknob high-level synthesis design space exploration acceleration," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 394–406, 2016. [Online]. Available: https://doi.org/10.1109/TCAD.2015.2472007

[18] Gurobi Optimization, LLC., "Gurobi documentation: Constraints," 2019. [Online]. Available: https://www.gurobi.com/documentation/8.1/refman/constraints.html

[19] H. Poon and P. Domingos, "Sum-Product Networks: a New Deep Architecture," *Proc. of UAI*, 2011.

[20] L. Sommer, J. Oppermann, A. Molina, C. Binnig, K. Kersting, and A. Koch, "Automatic mapping of the sum-product network inference problem to fpga-based accelerators," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct 2018, pp. 350–357.

[21] A. Molina, A. Vergari, N. Di Mauro, S. Natarajan, F. Esposito, and K. Kersting, "Mixed sum-product networks: A deep architecture for hybrid domains," 2018.

[22] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[23] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. M. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*. IEEE Computer Society, 2014, pp. 110–119. [Online]. Available: https://doi.org/10.1109/IISWC.2014.6983050

[24] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[25] B. E. Nelson, "Third party CAD tools for FPGA design - A survey of the current landscape," in *Applied Reconfigurable Computing - 15th International Symposium, ARC 2019, Darmstadt, Germany, April 9-11, 2019, Proceedings*, ser. Lecture Notes in Computer Science, C. Hochberger, B. Nelson, A. Koch, R. F. Woods, and P. C. Diniz, Eds., vol. 11444. Springer, 2019, pp. 353–367. [Online]. Available: https://doi.org/10.1007/978-3-030-17227-5_25