Latent text
behaviors. Thus, we define criteria for generating efficient parallel OpenCL code from RVC-CAL models. The generated code is finally executed on OpenCL-abstractive heterogeneous processor architectures. The contributions of this paper are therefore the following:

- We present a methodology that generates efficient software from RVC-CAL dataflow programs with the aim to improve end-to-end performance. In contrast to related approaches, the proposed methodology supports not only static actors, but even more generalized behaviors. In addition to this, our approach even exploits the fine-grained parallelism at the level of actions within actors (if available).
- To the best of our knowledge, we present the first code generation from RVC-CAL programs to SYCL. This allows us to compare the performance of OpenCL and SYCL.
- We validate our methodology by benchmarks on different OpenCL abstracted processor architectures. Based on the experimental results, we evaluate the methodology for performance in comparison with a pure multithreaded C++ approach and a well-known reference framework.

II. PRELIMINARIES

This section highlights the used programming languages, the employed hardware abstraction for heterogeneous architectures, and the existing related approaches for code generation.

A. Dataflow modeling using RVC-CAL

The RVC-CAL language is based on the CAL actor language (CAL) [3] developed for the specification of dataflow actors. It imposes some restrictions on CAL and only supports fully typed actors to enable efficient hardware and software generation. RVC-CAL is part of the standardized RVC framework of MPEG [4]. RVC-CAL supports the modeling of state-based dataflow actors with actions that consume and produce tokens. It incorporates specialized constructs that can be used for modeling different kinds of dataflow behaviors, offering parallelism at different levels. A sample actor modeled with RVC-CAL that contains all the considered language constructs for the proposed approach is shown in Listing 1.

An actor is declared with a finite set of input ports (input1, input2) and a finite set of output ports (output), separated with the identifier ‘===>’ (Lines 2-3). The state of an actor is defined by state variables (Line 5). The behavior of an actor is specified by a set of actions that are declared with a set of input and output ports (Lines 7-9, 11-18, 20-22). Each action upon execution can consume tokens from its declared input ports and produce tokens for its declared output ports. These tokens are designated by a, b and c. There are two ways to implement an action: The functionality can either be implemented in the output port access definition (Lines 8 and 21), or more descriptively in the action body (Lines 14-17). The listed actions are self-explanatory as they perform simple operations like addition, subtraction, etc. The repeat keyword associated with a port defines the number of tokens per execution that can be consumed/produced from an input/output port (Lines 12 and 21). An action can contain a guard (Line 13) that requires an additional condition for executing that action. Guard conditions can refer to state variables and the values of the consumed tokens.

The order in which actions are executed can be specified by an action schedule (Lines 24-28) and/or using a priority block (Lines 30-32). An action schedule is modeled as a finite state machine (FSM) where state transitions are triggered by the execution of actions. In a particular state, only the associated actions can be fired. For instance, in state s_start, action addtwiceandsub > add; can be fired. For such a state where more than one action can be fired, a priority block can be used to specify the priorities (Line 30-32).

The RVC framework also provides the XML based functional network language (FNL) to define network topologies for a set of RVC-CAL actors. The example networks are shown in Fig. 2 and Fig. 3.

B. OpenCL and SYCL

OpenCL is an open specification language designed for heterogeneous parallel computing on cross-vendor and heterogeneous architectures. The basic aim and the objective of OpenCL can be understood from two primary benefits it offers: First, OpenCL provides an abstract platform model that can be exploited for substantial acceleration in parallel computing. To this end, it supports both coarse-grained (task-level) as well

---

Listing 1: Example of a RVC-CAL actor.

```cpp
actor exampleActor()
int(size=8) input1,
int(size=8) input2 ===> int(size=9) output :
bool state_flag;
add: action
input1:[a],input2:[b] ===> output:[a+b]
end
addtwiceandsub: action
input1:[a] repeat 2, input2:[b] ===> output:[c]
guard b > 0
var int(size=9) c
state_flag := true;
end
forward: action
input1:[a],input2:[b] ===> output:[a,b] repeat 2
end
schedule fsm s_start :
s_start(forward) ===> s_working;
s_working(addtwiceandsub) ===> s_start;
s_working(add) ===> s_working;
end
priority
addtwiceandsub > add;
end
end
```
as fine-grained (data-level) parallelism. Second, it provides the ability to write vendor-neutral cross-platform applications. This is achieved by providing high-level abstractions to hide the lower-level details of implementations (drivers and the runtime system) as well as the memory consistency and execution models to allow cross-vendor development.

To further assist the development of OpenCL programs, a higher level abstraction layer has been built on top of OpenCL by SYCL [6]. SYCL features a single-source multiple compiler-passes (SMCP) design that allows the integration of kernels into the C++ host program. Consequently, this ability to embed kernels into the host program simplifies the development mainly because of two reasons: First, the communication between the host and the kernels is not required to be handled explicitly, as it is implicitly managed by the SYCL abstraction. Second, kernels are no longer needed to be written in OpenCL C (based on C99 standard). Instead, they can also be written in the host language (C++) that even allows the use of standard C++ libraries for better code development. Since a general discussion of OpenCL and SYCL is out of scope for this paper, we refer to [5], [6] for further details.

C. Related Approaches

Existing frameworks for automatic code generation of RVC-CAL programs can be classified in two different categories: First, there are frameworks that use dedicated software and hardware backends for generating C code and HDL code (VHDL/Verilog), respectively, and second, there are frameworks that employ specialized abstractions like OpenCL for generating code for different heterogeneous architectures. In this section, we discuss some of these frameworks.

1) Frameworks with Dedicated Backends: The Open DataFlow (OpenDF) framework [9] is built under the Eclipse environment that uses a software backend (CAL2C) [10] and a hardware backend [11] for the generation of C code and RTL descriptions (Verilog), respectively.

Another similar approach [8] presents a HW/SW co-design methodology built as an Eclipse plug-in on top of ORCC [7] and OpenForge1 for the generation of C code and Verilog code, respectively.

The work presented in [12] proposed the distributed application layer (DAL) [13] based design flow to execute RVC-CAL programs on multi-core platforms. A dedicated DAL backend based on the C backend of ORCC is proposed to translate RVC-CAL actors to DAL processes.

Limitations: Frameworks in this category generally use a multithreading concept at the abstraction of an OS, and thus offer limited utilization of parallelism at the coarse-grained level, usually restricted to devices like multi-core CPUs.

2) Frameworks with OpenCL Abstraction: The work presented in [14] provides an approach to translate RVC-CAL actors into programs running some of the computations on OpenCL. The methodology incorporates static analysis and is restricted to the synthesis of synchronous (static) behaviors. Similarly, an approach is presented in [15] to use the RVC-CAL language for programming OpenCL-abstracted GPUs. The basic idea relies on translating RVC-CAL programs to the DAL framework that allows the execution of static actors on OpenCL-compatible GPUs.

Independent of the used dataflow language, there exist other OpenCL based frameworks that execute dataflow behaviors on heterogeneous architectures. The framework presented in [16] introduces a design flow for executing applications specified as synchronous dataflow (SDF) graphs on heterogeneous systems using OpenCL.

Limitations: Frameworks in this category essentially support static behaviors, and thus only allow code generation of RVC-CAL programs restricted to static actors. Furthermore, these approaches mainly exploit parallelism limited to the concurrent execution of RVC-CAL actors.

III. PROBLEM AND APPROACH

A. RVC-CAL Code Generation

RVC-CAL is an actor oriented dataflow language that uses DPNs as model of parallel computation (MoC). As presented in Section II-A, modeled behaviors can potentially offer parallelism at the level of actors as well as at the level of actions within actors. These modeled behaviors can be efficiently executed on parallel architectures by exploiting the potential parallelism of these models. Therefore, in order to efficiently execute DPNs modeled with RVC-CAL on the target hardware, a generalized and systematic approach is needed that can automatically generate code based on the potential parallelism. The existing frameworks that support the automatic code generation of RVC-CAL DPNs, as presented in Section II-C, lack of a generalized approach: they only support the modeling of limited behaviors (static DPNs), and only exploit parallelism at the level of actors.

We therefore propose an approach that automatically generates software from RVC-CAL DPNs in a systematic manner based on the potential parallelism of modeled behaviors. In contrast to existing approaches, the proposed approach supports software synthesis from RVC-CAL actors which is neither restricted to static behaviors nor is it limited to the concurrent execution of actors. Instead, it even utilizes the fine-grained data-parallel computations at the level of actions within actors.

B. Proposed Design Flow

The overall methodology employs the multithreaded execution of RVC-CAL actors and the parallelization of their actions with SYCL or OpenCL. To this end, it is systematically organized into different levels of code generation and the final execution, as shown in Fig. 1.

Actor level. At this level, RVC-CAL actors and the FNL network description are taken into account where each actor is translated into a C++ class and a global scheduling routine is generated for scheduling actors, respectively. The global scheduling routine provides the scheduler at the actor level (discussed in detail in Section IV-C1). The actors are executed

1https://sourceforge.net/projects/openforge
concurrently by a fixed number of threads, generally equal to the available number of compute units. Each thread calls the global scheduling routine initially and when a scheduled actor terminates.

**Action level.** At this level, all actions of an actor are either translated into pure C++ code or to SYCL/OpenCL code for parallel execution. This decision is explained in detail in Section IV-A. If the criteria for parallelization with SYCL/OpenCL are not met, the actions are translated to C++ member functions and are executed without any further parallelization. Also, the initialization actions are not considered for parallelization because they are executed only once by the constructor.

**Execution of actions.** During execution, the global scheduler decides which actors to execute next in the calling threads. Each actor has its own local scheduling routine that manages the execution of actions based on the modeled behavior. Upon execution of an actor, its local scheduler is called. Depending on the generated code, the local scheduler either calls an action that is parallelized by SYCL/OpenCL or directly executes it in the thread. The local scheduler, however, does not distinguish between both cases. Each time a local scheduler terminates, the control flow returns to the global scheduler and the next actor is scheduled in the current thread. The local scheduling scheme is discussed in detail in Section IV-C2.

The focus of this work is on the efficient code generation of RVC-CAL DPNs based on the parallelism offered by modeled DPNs, mainly by using a combination of multithreaded C++ and the OpenCL/SYCL model. Thus, we do not detail how the generated code is mapped and executed on devices.

### IV. The Proposed Methodology

This section explains the proposed criteria for code generation (parallelization) and the scheme for execution (communication and scheduling).

#### A. Parallelization

As discussed, each RVC-CAL actor is translated into a C++ class. A translated C++ class of the actor shown in Listing 1 is shown in Listing 2. The generated class contains the member variables for the state variables (Line 5), an enumeration for the FSM (Lines 7-9), the variable holding the current state of the FSM (Line 11), the ports to which the input and output FIFOs can be attached (Lines 14-23) and a constructor routine (Lines 20-26).

The proposed concept of executing actions within actors in parallel mainly relies on the fine-grained (data-level) parallelism of actions using SYCL/OpenCL. Hence, the basic idea is to execute multiple instances (work-items/threads) of an action, operating on different data, in parallel. However, this can only be achieved if each work-item knows exactly where to read/write its input/output data. Based on this fundamental concept, the proposed criteria for deciding whether an actor (entirely modeled with RVC-CAL) can further be parallelized with SYCL/OpenCL or not is depicted in Algorithm 1.

For each actor in a network, the algorithm first checks whether an actor has state variables (Line 2). As the same state variable can be used by different actions, executing them in parallel without ensuring the correct access order can lead to non-deterministic implementation. Therefore, actors only without state variables are qualified for further investigation. Next, the algorithm checks whether an actor has an FSM or not (Line 3). If an actor has no FSM, the algorithm checks if there are actions with guard conditions (Line 4).
If there is no guard condition, for each action, a separate SYCL/OpenCL kernel and the corresponding host code is generated. Each kernel can be executed in a number of parallel instances (work-items) depending on the availability of tokens. However, in case an actor has a guard condition, the algorithm checks if the actor is synchronous (static) (Line 7). If an actor consumes/produces a fixed number of tokens in each firing, it can be parallelized even with guard conditions as each work-item would know exactly where to read/write its input/output data and each work-item can evaluate the guard conditions independently. In this case, a single SYCL/OpenCL kernel is created for the actor containing all actions. Hence, the guard conditions are evaluated in the kernel and the ready actions are executed.

If an actor has an FSM, the algorithm first checks if it consists of exactly one cycle (Line 11). In this case, the resulting action schedule is clear because in each state there is only one state transition possible. Therefore, a SYCL/OpenCL kernel is created for a complete cycle of the action schedule. Since, a complete cycle does not always have to be executed, thus, for each action a C++ version is also generated. However, if there are multiple state transitions possible in a single state of an FSM, no parallelization is possible because a scheduling decision is required before the execution of each action.

The decision on whether an actor can be parallelized further is not influenced by the priority block, mainly because it statically defines the priorities between actions. Therefore, the proposed algorithm does not consider the priority block as a part of the criteria for parallelization.

**Generated code - OpenCL.** The generated OpenCL kernel for the same add action is shown in Listing 4. The OpenCL version of the add action is similar to the SYCL one, except a different function primitive is used for retrieving the buffer indices (Lines 8-10). As part of the OpenCL specification, the host code is generated separately as shown in Listing 5. As in the SYCL version, first the number of parallel instances is calculated (Line 2) and assigned as the index space for the OpenCL call (Line 6). The OpenCL buffers are set as the arguments of the kernel (Lines 6-10) and the kernel is enqueued for execution by the function `ExecuteKernel` (Line 13). The object `ocl` stores all relevant object references like the OpenCL context, kernels, the index space and so on.

**B. Communication**

A general RVC-CAL dataflow program consists of a network of actors that communicate with each other via statically determined point-to-point buffers. To this end, we use bounded First-In-First-Out (FIFO) buffers, realized as cyclically addressed arrays.
Communication with the OpenCL device. The data communication between the host (FIFO buffers) and the SYCL/OpenCL device is carried out using SYCL/OpenCL buffers. Each time a kernel is enqueued for execution, SYCL/OpenCL buffers are created on-the-fly mainly because of two reasons: First, this allows us to create these buffers with the exact required size based on the computed number of parallel instances. Second, this allows us to consistently copy data from the FIFO buffers to the SYCL/OpenCL buffers in a proper aligned manner. Essentially, this design lets us to achieve a contiguous memory access pattern at the SYCL/OpenCL device. On the contrary, if we consider the same FIFO structure implementation at the device side, this would require additional offset and modulo operations for accommodating non-contiguous memory access patterns. Hence, using the proposed design does not only allow us to simplify the code generation, but potentially enables us to improve performance by avoiding additional overhead caused by non-contiguous memory accesses at the device. The SYCL/OpenCL buffers are removed once the SYCL/OpenCL call is finished and the data is copied back to the associated FIFO buffers.

C. Scheduling

As discussed, we use a two-level scheduling scheme, where at the first level, the global scheduler decides which actors to execute next, and at the second level, the local scheduler decides which action of the scheduled actor to execute next.

1) Global Scheduling: The global scheduler is typically executed by multiple threads in parallel and relies on an atomic data structure to keep track of whether an actor is active or not. As RVC-CAL does not enforce any termination criteria for the network, the global scheduler runs forever. It works in a round-robin fashion and selects the next actor in the list that is not running and has no pending SYCL/OpenCL calls. In case an actor has a pending call, i.e., all kernel instances are not completely executed, the next invocation of this actor is delayed to buffer up tokens and achieve more parallel instances. The list of actors provided to the global scheduler is sorted during the code generation in the order of source actors, source-plus-sink actors, and finally sink actors. When an actor is scheduled for execution, the local scheduler of this actor is called, where it is executed in the same thread of the global scheduler. After the local scheduler is terminated, the global scheduler sets the actor to an inactive state and selects the next actor for execution.

2) Local Scheduling: Each actor has its own local scheduler that manages the execution of actions. The decision on which action of the actor has to be executed next is based on the available tokens, the current state of the FSM, the priority block and the evaluation of the guard conditions. The local scheduling scheme is illustrated in Algorithm 2.

Based on the algorithm, the scheduler first selects the current state (Lines 1-2) and iterates through the list of actions that are executable in the current state (Line 3). This list is sorted according to the priorities of the actions to ensure that actions with higher priorities are tested first for two additional conditions. As a first condition, the local scheduler tests whether enough tokens are available and if the guard is
fulfilled (Line 4). If this evaluates to true, the second condition is considered, otherwise the next action is selected. In the second condition, the output buffers are tested for sufficient free space to store the output tokens (Line 5). Hence, if required free space is not available in the output buffers, the local scheduler terminates to delay the execution until sufficient free space is available (Line 11). If both conditions are evaluated to true, the action is executed, and the state is updated (Lines 6-8). The local scheduler also terminates when there is no executable action available (Line 15).

V. EXPERIMENTAL EVALUATION

We selected two benchmarks, namely a FIR filter and a ZigBee multi-token transmitter from the ORCC samples repository², mainly because of following reasons: First, the applications are entirely modeled with RVC-CAL code. Second, a C++ program can be generated from RVC-CAL code and can be executed to get the desired output. Finally, the applications feature different kinds of DPNs, where the first benchmark features only static actors with constant token rates per firing, and the second one also accommodates actors with more flexible behaviors offering multiple token rates.

For each selected benchmark modeled with RVC-CAL, three different software implementations are generated, namely the C++/SYCL based implementation, the C++/OpenCL based implementation and a pure C++ version that only uses multitreading for the concurrent execution of actors. The total execution time of the network to process the complete input data set, including initialization and termination of the program, is used as the comparison metric. Based on that, the execution of the pure C++ version and the implementation generated by the ORCC framework are compared against the SYCL/OpenCL based implementations. The data set used has a maximum of 37 million samples and the average of fifty repetitions is taken for each version.

A. Experimental Setup

We executed the benchmarks on the following hardware:

- Intel i5-7200U CPU
- NVIDIA GTX 950M GPU
- 8GB RAM

Unfortunately, the SYCL version cannot be evaluated on the GPU because NVIDIA devices do not support the standard portable intermediate representation (SPIR) that is required by SYCL. For the execution of the benchmarks we used the following software environment:

- ComputeCpp Version 1.0 conformant to SYCL 1.2.1
- Intel OpenCL SDK Version 7.0.0.2519
- Windows 10 Pro Version 1803 Build 17134.407

²https://github.com/orcc/orc-apps

TABLE I: Results for the FIR filter.

<table>
<thead>
<tr>
<th></th>
<th>C++ CPU</th>
<th>ORCC CPU</th>
<th>OpenCL GPU</th>
<th>OpenCL CPU</th>
<th>SYCL CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time [ms]</td>
<td>6458</td>
<td>5961</td>
<td>5921</td>
<td>7321</td>
<td>106032</td>
</tr>
<tr>
<td>Relative Performance [%]</td>
<td>100</td>
<td>108</td>
<td>109</td>
<td>88</td>
<td>0</td>
</tr>
</tbody>
</table>

B. Evaluation of the FIR Filter

The low level finite impulse response (FIR) filter consists of thirteen actors as shown in Fig. 2. The actions of the actors source and sink cannot be further parallelized with SYCL/OpenCL because they contain C code and, therefore, are not entirely modeled with RVC-CAL. Also, the actions of the delay actors cannot be parallelized because they contain state variables. Apart from that, the actions of all other actors are parallelized as they meet the criteria. The performance comparison of the considered implementations is shown in Table I.

Discussion. Although the parallelized actions only perform simple operations like addition and subtraction, the generated OpenCL version on the CPU shows the best performance compared with all other versions. Considering the fact that only few simple actions are parallelized, the OpenCL CPU version shows a promising performance, slightly better than the ORCC version and considerably better than the C++ version. The OpenCL version on the GPU understandably performs low because the communication overhead of OpenCL on GPU is higher than the performance gain of executing rather small kernels in parallel on many cores. In contrast to the OpenCL CPU where the host and the kernels reside on the same device, in the case of GPU, the data has to be transferred to the GPU and back to the main memory (host). This overhead therefore contributes in elevating the total execution time. The SYCL version, unfortunately, evokes a substantial decrease in performance, mainly because of the overhead induced by the implemented runtime of the abstraction layer. Based on our profiling and analysis, we discovered that the SYCL runtime consumes a lot of time on the host and therefore substantially elevates the execution time.

C. Evaluation of the ZigBee Multi-token Transmitter

The ZigBee multi-token transmitter consists of six actors as shown in Fig. 3. For this benchmark only the actions of the actors chipMapper and qpskMod can further be parallelized
because all other actors do not meet the proposed criteria. The actors headerAdd and pulseShape exhibit generalized behaviors with multiple token rates. Each actor contains an action schedule with more than one cycle and incorporates guard conditions for realizing multiple token rates. The performance comparison of the considered implementations is depicted in Table II.

**Discussion.** As the parallelized actions comparatively perform more complex operations, the generated implementations show promising results, although there is still room for improvement. The OpenCL CPU version again performs best of all other versions, in particular, about 14% faster than the ORCC version. With the increased complexity of parallelized actions, the OpenCL GPU version also performs considerably better than the ORCC version. However, the communication overhead caused by the data transfer between the GPU kernels and the host (main memory) still makes it slower than the OpenCL CPU. To further improve the performance, the actors can be merged to yield more complex actions. In particular, the static actors, namely the chipMapper and the qpskMod can be merged to design more complex actions to further improve the performance. The SYCL version although shows some improvement from the first benchmark, but still lags behind the counterpart versions.

VI. CONCLUSIONS AND FUTURE WORK

We propose an approach for automatically generating efficient parallel software from RVC-CAL programs based on the analysis of the parallelism of the modeled behaviors. To this end, we discussed criteria that take into account the RVC-CAL and the OpenCL specification, and specify a set of rules for generating software that exploits the potential parallelism. The approach is evaluated by benchmarks on different OpenCL abstracted hardware platforms. Based on the evaluation, the proposed approach has shown promising results with improved end-to-end performance. In particular, even with the selection of worst case applications i.e., only few simple actions could further be parallelized based on the proposed criteria, the proposed approach has improved the performance up to 14% in comparison with the reference approaches.

As a future work, we aim to optimize the criteria to analyze and merge actors to get more complex kernels. Also, the future work will focus on exploring schemes for efficiently mapping OpenCL executions on devices.

**References**


**Table II:** Results for the ZigBee multi-token transmitter.

<table>
<thead>
<tr>
<th></th>
<th>C++ CPU</th>
<th>ORCC CPU</th>
<th>OpenCL CPU</th>
<th>OpenCL GPU</th>
<th>SYCL CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time [ms]</td>
<td>4900</td>
<td>5405</td>
<td>4753</td>
<td>4839</td>
<td>14244</td>
</tr>
<tr>
<td>Relative Performance [%]</td>
<td>100</td>
<td>90</td>
<td>103</td>
<td>101</td>
<td>34</td>
</tr>
</tbody>
</table>


