**Journal of Signal Processing Systems manuscript No.**
(will be inserted by the editor)

# Design flow for GPU and multicore execution of dynamic dataflow programs

**J. Boutellier, T. Nyländen**

June 15, 2016

**Abstract** Dataflow programming has received increasing attention in the age of multicore and heterogeneous computing. Modular and concurrent dataflow program descriptions enable highly automated approaches for design space exploration, optimization and deployment of applications. A great advance in dataflow programming has been the recent introduction of the RVC-CAL language. Having been standardized by the ISO, the RVC-CAL dataflow language provides a solid basis for the development of tools, design methodologies and design flows. This paper proposes a novel design flow for mapping RVC-CAL dataflow programs to parallel and heterogeneous execution platforms. Through the proposed design flow the programmer can describe an application in the RVC-CAL language and map it to multi- and many-core platforms, as well as GPUs, for efficient execution. The functionality and efficiency of the proposed approach is demonstrated by a parallel implementation of a video processing application and a run-time reconfigurable filter for telecommunications. Experiments are performed on GPU and multicore platforms with up to 16 cores, and the results show that for high-performance applications the proposed design flow provides up to 4× higher throughput than the state-of-the-art approach in multicore execution of RVC-CAL programs.

**Keywords** dataflow computing · design automation · signal processing · parallel processing

Center for Machine Vision and Signal Analysis
University of Oulu, Finland

## 1 Introduction

Parallel and heterogeneous computing has spread from desktop computers to embedded systems and mobile consumer products, such as smartphones and tablets. Unfortunately, the progress of programming languages and tools has not managed to keep up with the advances of hardware.

Dataflow is a well-known programming formalism, especially suitable for describing concurrent applications. It provides a formally defined basis for mapping a program to several parallel processing cores. However, most variants of the dataflow programming paradigm (such as [18,19]) only define the rules of communication between different program components, but do not specify the programming language for writing applications.

RVC-CAL is an ISO-standardized language [22] for dataflow programming under the *dataflow process networks* (DPN) [19] model of computation. RVC-CAL is based on the CAL actor language [11] and is at the moment best known for its use in platform-agnostic description of video decoders [22]. The main advantages of RVC-CAL over mainstream programming languages are its high level of abstraction, inherent concurrency and modularity that together form an excellent basis for platform independent description of programs for execution on multicore platforms. Moreover, unlike some other popular [18,5] dataflow formalisms, RVC-CAL allows describing *dynamic* applications, i.e. ones that have data-dependent behavior.

Since several years, RVC-CAL programming tools have provided the possibility to compile programs written in the RVC-CAL language to multithreaded workstation executables [30], enabling their deployment to Windows, Linux, and OS-X -based systems.

This paper proposes a novel approach for mapping high-performance applications written in the RVC-CAL language to heterogeneous platforms that consist of general-purpose cores and GPUs. In detail, the contributions of this work are:

- A novel, freely available[1] design flow for dataflow programming
- Efficient mapping of dataflow programs to GPUs and multicores
- Benchmarks that show up to 4× throughput improvement over the state-of-the-art RVC-CAL multicore performance
- The first design flow for RVC-CAL that enables efficient use of GPUs

This article presents a unified and extended description of the proposed design flow, which has been addressed in two of our previous works [8,6].

The rest of the paper is organized as follows: Section 2 describes the RVC-CAL language and gives an overview of related work; Section 3 describes the proposed design flow; Section 4 introduces the conducted experimental evaluation; Section 5 summarizes the experiments and Section 6 concludes the paper.

## 2 Background

### 2.1 The RVC-CAL dataflow language

In the RVC-CAL language, programs are described as sets of stateful *actors*. Similar to other dataflow formalisms, communication between actors happens solely over lossless directed FIFO (First-In-First-Out) channels that bear *tokens* of a pre-defined size. FIFO-based communication rules out the possibility of, for example, using global variables that can cause a variety of synchronization related issues on multicore platforms.

The interface between an actor and a FIFO channel is called a *port* that may have an input direction in which case the actor is said to *consume* tokens from a port, or an output direction in which case it *produces* tokens. The internal structure of an RVC-CAL actor is a finite state machine (FSM), whose state transitions are called *actions*. When an action is *fired* (executed), it consumes a pre-defined number of tokens from the actor's input ports, performs computations, and produces data to its output ports.

The FSM structure of an RVC-CAL actor allows *data dependent* execution of actions. Data dependence enables description of adaptive and dynamic applications where the processing path of tokens can change at run time, as for example in video decoders [22].

The RVC-CAL language is platform independent, which enables it to be used for both software and hardware description. The *Open RVC-CAL Compiler* (Orcc) [31] implements target independence of the language by providing a variety of *backends* that enable synthesizing an RVC-CAL program into Verilog [3], LLVM assembly [15] and multithreaded C [30].

### 2.2 Related work

**Multicore execution** of RVC-CAL (or CAL) programs has been widely studied. Early works such as [7] and [20] concentrated on design space exploration and simulation, accompanied with first measurements on multicore processors [2]. Multicore code generation from the Orcc compiler was first described in [30]. Later, results based on the same framework were reported in [4], [29] and [10].

Besides desktop multicores, works on executing RVC-CAL programs on embedded multicores have been presented. In [4], an RVC-CAL video decoder was executed on 4 cores of a Freescale P4080 processor; in [32] on multiple soft-core processors on FPGA; in [9] on a multicore DSP, and in [14] on the Epiphany architecture.

C++ based dataflow programming of multicores is realized, for example, by the XKaapi runtime system [13], the CnC programming model [23] and the Distributed Application Layer [25]. In these frameworks, programs are executed under a dataflow (or Kahn Process Networks (KPN) [17]) based execution environment.

**GPU programming** has taken considerable advances from the times when shader languages were used for the purpose: Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) being the most popular alternatives. These two frameworks provide an efficient, yet a rather low-level access to GPUs that may reduce programmer productivity. Approaches that require less programmer effort are, for example, Renderscript and C++ AMP. However, as it has been discovered in the case of OpenACC [16], a higher level of abstraction may also reduce performance. At the same time as high-level programming approaches are developed, both CUDA and OpenCL are acquiring new features. For example, dynamic parallelism and shared virtual memory introduced in the OpenCL 2.0 standard can be seen as significant improvements from both performance and productivity aspects.

Besides our previous article [8], GPU programming based on the RVC-CAL language has been proposed in the work of Lund et al. [21]. Although the objective

[1] https://github.com/orcc, http://www.dal.ethz.ch/

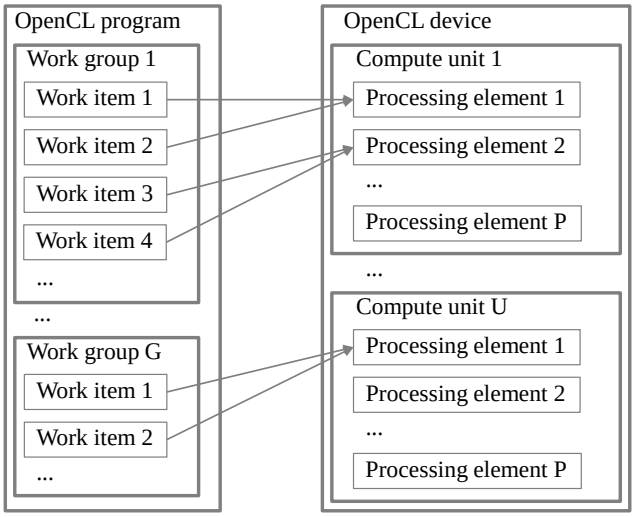


**Fig. 1** Mapping of an OpenCL program to an OpenCL compatible device.

is the same, Lund et al. have approached the problem by taking RVC-CAL application descriptions that have previously been designed for execution on a general purpose processor, and translating those for execution on OpenCL devices. The approach of the proposed work is different: we argue that RVC-CAL can very well be used to program OpenCL devices if the programmer uses suitable language constructs, which has been confirmed with experimental validation that shows speedup gain of up to 42× [8].

The work described in this paper builds on top of the Distributed Application Layer (DAL) [25] framework. DAL applications are written in C++ under the Kahn Process Networks (KPN) [17] model of computation. The framework allows running one or more applications on a many-core platform and includes an OpenCL code generator, which has been shown to be remarkably efficient [26]. The proposed work uses the OpenCL and multicore code generators of DAL, and on the other hand provides a dataflow language programming interface for DAL.

### 2.3 OpenCL concepts

Fig. 1 depicts the mapping of a generic OpenCL program to a generic OpenCL device and explains some key concepts of the language. A computing system may contain one or more OpenCL *devices* (usually GPUs) that have a specific number of *compute units* (1, 2, ..., U in Fig. 1). The compute units, then, have a fixed number (P) of *processing elements*.

An OpenCL program maps to an OpenCL device such that each processing element is responsible for executing one or more *work items*, each of which is independently executable with respect to other work items. Work items form *work groups* such that each compute unit can be responsible for executing one or more work groups. [12]

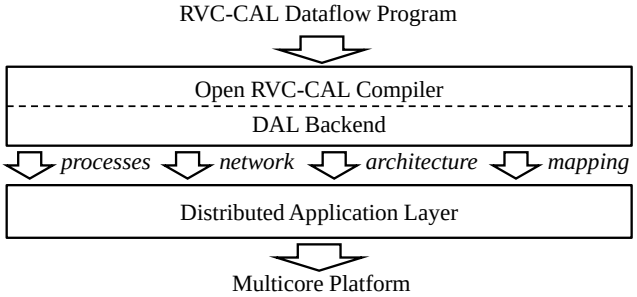


**Fig. 2** The proposed design flow that essentially consists of the Orcc dataflow design environment (upper box) and the DAL runtime framework (lower box).

## 3 The proposed design flow

This paper proposes a design flow for mapping RVC-CAL applications to multicore processors and GPUs. The design flow is based on two major software frameworks, the Orcc compiler and the Distributed Application Layer [25] as shown in Fig. 2. DAL is a framework for mapping one or more streaming applications to multi- and manycore platforms, such as generic Linux-based workstations, as well as Intel Xeon Phi, Intel SCC and GPUs. In addition to the wide multicore support, DAL also provides fault tolerance through allocation of spare cores, and minimal operating system features by supporting execution of multiple simultaneous applications.

The model of computation assumed by RVC-CAL is DPN [19] that enables actors to base their behavior on token values and token availability, which makes some RVC-CAL actors non-deterministic. In contrast, the model of computation assumed by DAL is KPN [17] that does not allow processes (equivalents of actors) to query for token availability, but requires that tokens are removed from the FIFO once their value has been read (a *destructive read*). A consequence of destructive reads is that DAL processes block if a FIFO buffer does not have enough data for a read. On the other hand, KPN guarantees deterministic behavior of a process network.

The fact that a DAL program cannot read data from a FIFO non-destructively has two implications: a) implementation of data dependent processing in actors needs special consideration (discussed in Section 3.1), and b) there is a set of non-KPN compatible RVC-CAL actors that cannot be translated to DAL processes. Such incompatibility is essentially caused by *token availability*

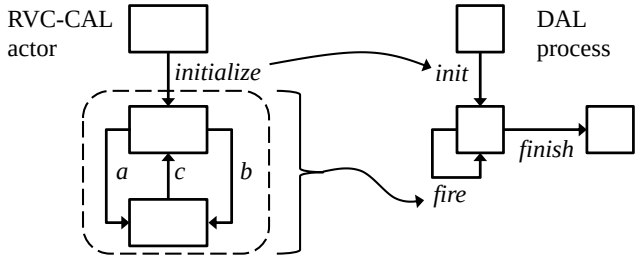


**Fig. 3** Mapping an RVC-CAL actor that consists of actions $a$, $b$ and $c$ to a DAL process, where the actions are encapsulated inside the *fire* function.

*dependent actor behavior*, which e.g. in [19] is exemplified with a two-input nondeterminate merge actor: if a token is available on either of such an actor's inputs, the token immediately becomes consumed. Such behavior is time dependent and incompatible with the deterministic nature of KPN, and has been observed in 17% of RVC-CAL actors [27]. Interested readers should refer to [27] for a more elaborate discussion.

Interoperability between RVC-CAL and DAL is achieved in the *DAL Backend* [8,6] of Orcc, the software component that enables the proposed design flow. As DAL assumes that processes (actors) are written in C/C++, the DAL Backend is based on the C backend [28] of Orcc. The DAL process network description and the process-to-core mapping files can be generated from Orcc without any transformations. In contrast, the procedure of translating RVC-CAL actors to DAL processes requires consideration of several issues that are explained in the following subsection.

### 3.1 Translating actors to processes

DAL processes consist of three procedures: `init` that is responsible for process setup; `fire` that performs the actual processing; and `finish` that terminates the process. RVC-CAL actors, on the other hand, have an optional `initialize` action, and one or more processing actions that are arbitrated by the actor's FSM structure.

For each actor in the RVC-CAL program, the DAL Backend maps the contents of the `initialize` action to the `init` procedure of the respective DAL process, and the rest of the actions together with the FSM structure to the `fire` procedure. The `finish` procedure is left empty except for termination of possible file stream accesses. An example of this translation with an actor that has processing actions $a$, $b$ and $c$ is shown in Fig. 3.

The DAL Backend implements non-destructive FIFO reads to the generated processes by instantiating a *token prefetching* mechanism for each actor input port that is detected to require non-destructive read access. The detection is performed automatically and for the sake of performance it is important to incorporate the prefetching mechanism only to those ports where it is needed. Prefetching is in practice a port-specific variable that keeps the latest token accessible until a destructive read is performed.

Detection of non-KPN compatible RVC-CAL actors that cannot be transformed to DAL processes is performed with an actor classifier [27] that is a part of the Orcc compiler. Non-KPN compatibility can either be caused due to actor nondeterminism (producing varying output for the same input) or determinate actors that cannot be expressed as Kahn processes [27].

The DAL framework does not support *broadcast FIFOs*, i.e. FIFOs that have one writer and multiple readers. Hence, applications that contain broadcast FIFOs need to undergo a transformation before the DAL Backend can continue.

### 3.2 Describing data level parallelism in RVC-CAL

Mapping of a concurrent dataflow program to a multi-core platform is mostly a matter of actor-to-core mapping and load balancing. In contrast, it is not so straightforward to make a dataflow program utilize the full performance potential of a GPU.

GPUs have been performance-optimized to execute programs in a single program, multiple data (SPMD) fashion. For efficient SPMD-style execution, the application needs to provide a high degree of data level parallelism. The RVC-CAL language, as most dataflow formalisms, can however most naturally express *task level* parallelism, where distinct tasks are executed concurrently. Intuitively, one could describe data parallelism in dataflow by replicating an actor into multiple parallel instances. However, as GPUs require thousands of parallel tasks for efficient execution such replication should not be done in the application description, but rather automatically by the compiler.

To this end, suitable language structures need to be identified for expressing data level parallelism. In the presented design flow this is achieved by the RVC-CAL `foreach` construct, whose iterations the code generator will then convert into independent computations *if the iterations are independent of each other*. An example of this is the actor `Gauss` in Fig. 4 where each `foreach` iteration (indexed by `j`) consumes a dedicated sample from the actor input port and produces dedicated samples to the output ports. If the designer decides not to place an actor such as in Fig. 4 to GPU execution, but rather run it on a sequential processor, the iterations of the `foreach` loop are executed sequentially. Similar to

```
actor Gauss() uint(size=8) in1 ==>
uint(size=8) out1, uint(size=8) out2:

  initialize ==> out2:[ delay ] repeat SIZE
  var
    uint(size=8) delay[SIZE]
  end

  action in1:[ rbuf ] repeat SIZE ==>
  out1:[ wbuf1 ] repeat SIZE, out2:[ wbuf2 ] repeat SIZE
  var
    uint(size=8) wbuf1[SIZE], uint(size=8) wbuf2[SIZE],
    int idx, int tmpgauss
  do
    foreach int j in 0 .. SIZE-1
    do
      if (j < 2+2*WIDTH || j+2+2*WIDTH > SIZE) then
        idx := 2+2*WIDTH; else idx := j; end

      tmpgauss := 0;
      foreach int k in -2 .. 2
      do
        tmpgauss := tmpgauss
          + gaussian[(k+2)*5+0] * rbuf[idx-2+k*WIDTH]
          + gaussian[(k+2)*5+1] * rbuf[idx-1+k*WIDTH]
          + gaussian[(k+2)*5+2] * rbuf[idx+k*WIDTH]
          + gaussian[(k+2)*5+3] * rbuf[idx+1+k*WIDTH]
          + gaussian[(k+2)*5+4] * rbuf[idx+2+k*WIDTH];
      end
      wbuf1[idx] := tmpgauss >> 8; wbuf2[idx] := tmpgauss >> 8;
    end
  end
end
```

**Fig. 4** The RVC-CAL actor Gauss that has the input port `in1`, and output ports `out1` and `out2` that each have a constant data rate of `SIZE`.

the approach of DAL [26], iterations of the outermost (`foreach`) loop are mapped to distinct OpenCL work items.

Data supply for parallel execution of the `Gauss` actor is ensured by the RVC-CAL keyword `repeat` that requires a fixed number of input tokens to be available and accessible through the token vector `rbuf` before computations specified between the `do` and `end` keywords can begin. Likewise, the underlying model of computation ensures that sufficient buffer space will be available for the output vectors `wbuf1` and `wbuf2`.

Especially desktop GPUs may require large quantities of data to be transferred between the CPU cores and the GPU for efficient operation. For example, in [26] data buffers of 32 MB were used. On the other hand, optimizing the application performance often requires device-specific settings for work group size and work item count (using OpenCL concepts) and hence these should be decoupled from the application specification.

In this work the FIFO buffers *within* the GPU and *between* the GPU and CPU cores have a hierarchical structure as shown in Fig. 5. The topmost part a) depicts the whole memory block of such a GPU-mapped FIFO, which consists of identical sub-buffers b) that each have the buffer size that is visible to the RVC-CAL application. Taking Fig. 4 as an example, if the constant `SIZE` would be of size 76800 and the total size of the buffer would be 300 kilobytes, the buffer of Fig. 5 a) would consist of 4 sub-buffers. Finally, Fig. 5 c) shows how the buffer visible to the application (e.g. of size 76800) is divided to individual work items for processing. Hence, if the number of work items is set to 768, each work item is responsible for processing 100 samples.

The FIFO buffers are allocated by DAL via OpenCL API calls. If both of two actors are running on an OpenCL device, FIFOs between them are allocated from the device memory. Likewise, if both of two actors are running on the host, FIFOs between them are allocated from the host memory. In cases where one actor is running on the host, and another one on the device, FIFOs between them are allocated by DAL following the scheme of *triple buffering* [24].

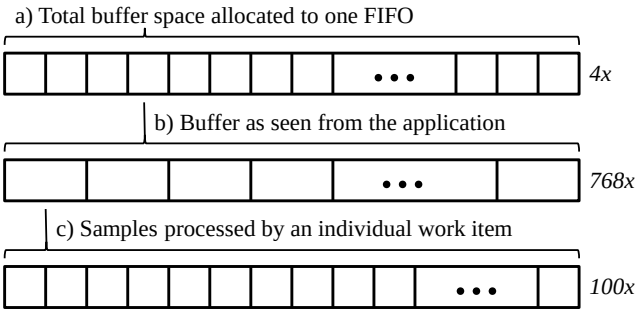


**Fig. 5** The proposed hierarchy for data buffering. The top row indicates the total memory space allocated for the FIFO that is divided into sub-buffers (here, 4). Each sub-buffer (e.g. size of a frame) is processed by the actor in one firing. Finally, each sub-buffer is divided to a number of OpenCL work items (here, 100) for processing.

### 3.3 Developing an application using the design flow

With the proposed design flow, a programmer writes her/his program using the Orcc compiler. Orcc provides an Eclipse-based graphical user interface for creating an actor network and for editing the RVC-CAL code of individual actors (as in Fig. 4). For each RVC-CAL actor network, Orcc enables creating *run configurations* that in the case of using the DAL Backend also allow mapping each actor to either a GPU device or to a CPU core, which is done manually by the programmer using the Orcc (Eclipse) dialog window for actor to processing mapping. The computing platform, in contrast, is defined outside the Orcc environment using the XML format specified by DAL. Orcc also generates a default configuration for the number of work items for each GPU-mapped actor, which the programmer can later on customize if needed.

## 4 Experiments

In this section the proposed design flow is experimentally evaluated. Throughout the experiments, two applications were used: parallelized video motion detection, and reconfigurable predistortion filtering. The experimental results were acquired by executing the applications on three workstations that are given in Table 1. The experiments consist of two parts: a) multicore measurement of application throughput as a function of buffer size, with several processor mappings, and b) GPU throughput measurement.

In multicore measurements the performance of the proposed design flow was compared against the standard multicore code generation approach [30] that is part of the Orcc compiler. In all the comparisons the actor-to-core mappings were static and identical between the proposed approach and the reference [30].

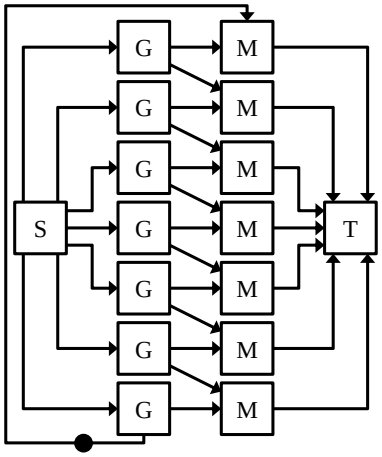


**Fig. 6** The data flow graph of the 7-way parallelized motion detection application.

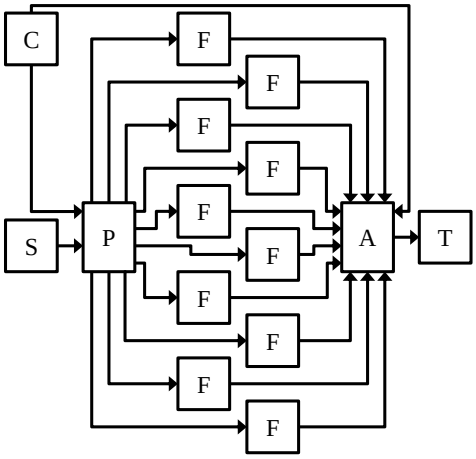


**Fig. 7** The data flow graph of the predistortion application.

### 4.1 Application use cases

Experiments were performed on two different applications. The first application was parallelized real-time motion detection on a video stream (Fig. 6). The motion detection application consisted of three phases, of which in the first one the video stream is processed with a noise-mitigating 5x5 pixel Gaussian filter, followed by subtraction of consecutive frames and 5-pixel median filtering.

In multicore experiments Gaussian filtering was performed in a dedicated actor, whereas median filtering and frame subtraction were performed in the same actor to achieve load balance. Moreover, in the multicore experiments the motion detection application was parallelized to 1,2,...,7 separate processing paths such that each processing path was dedicated to processing of one frame. The 7-way parallelized application is depicted in Fig. 6, where S is the source actor, T is the sink,

**Table 1** Platforms used for experiments. Xeon and Opteron platforms are used for multicore scaling experiments, whereas the i7 platform is used for the GPU experiment.

| Tag | CPUs | SMT | Logical cores | GPU |
|---|---|---|---|---|
| Xeon | 2 × Intel Xeon E5-2650 v2 (2.6 GHz) | ×2 | 32 | n/a |
| Opteron | 4 × AMD Opteron 8378 (2.4 GHz) | n/a | 16 | n/a |
| i7 | Intel Core i7-4770 (3.4 GHz) | ×2 | 8 | NVidia GeForce GTX 750 Ti |

G stands for Gaussian filtering and M for combined median filtering and frame subtraction.

For the GPU experiment frame subtraction and median filtering were divided to separate actors, as having them in the same actor would have required the use of OpenCL *barriers* that are not supported in the current design flow [8]. The actors executed on the GPU were Gaussian filtering, median filtering and frame subtraction, whereas source and sink executed on dedicated workstation cores. In contrast to the multicore experiments, no actor level parallelization was done. Instead, the desired parallelism was extracted from *foreach* constructs of each actor, as was shown in Fig. 4.

The second application use case was a configurable digital predistortion filter (Fig. 7) that is used in telecommunications transmitters to mitigate non-linearities of transceiver components [1]. The filter has a Parallel Hammerstein structure with up to 10 filter branches, each having 10 FIR filter taps and operating on complex values that were implemented with pairs of single-precision floats. The filter was designed to be run-time reconfigurable such that it can change the number of active filter branches at any time to find a suitable operating point for varying channel conditions and to minimize the power dissipation of digital filtering. In Fig. 7, C is the graph configuration actor, P presents computation of polynomials, F for complex-valued FIR filtering, and A for complex-valued signal addition. Each connection presents a pair of FIFO buffers for the real and imaginary components of the signal. Meaningful GPU execution of predistortion was not possible, as the DAL framework currently restricts GPU execution to actors that have fixed data rates [26].

### 4.2 Multicore throughput measurements

The motion detection application was instantiated as seven different versions: a sequential one and six parallel implementations that had 2,3,...,7 parallel processing paths. The throughput of the sequential version was measured with two different actor-to-core mappings: 1) every actor on one core, and with 2) a separate core for each of the four actors. Hence, the sequential version gave throughput results for core counts 1 and 4. The parallelized versions were mapped such that there always was one actor per core, which resulted in the 2-way parallelized version to occupy 6 cores and the 7-way parallelized version to occupy 16 cores. The video frame size in the experiments was 320x240 (grayscale), which required a FIFO size of 256 kB since one FIFO buffer contained initial data (black dot in Fig. 6) worth of one frame to implement frame subtraction. Hence, FIFO size of 256 kB was used throughout the motion detection experiments. Fig. 8 shows the throughput of the motion detection application on the Xeon platform. The curve shows the average throughput for 8 iterations and the vertical bars depict the standard deviation of the measurements. Respective measurements for the Opteron platform are shown in Fig. 9.

The predistortion application posed no limitations to communication buffer sizes, which enabled measurements with various buffer sizes. The performance of the predistortion application was measured on 1, 7, 10 and 15 cores, and for each core count the number of actors was 15. The predistortion application consisted of 10 FIR filter branches that were assigned to dedicated cores on the 15-core mapping. On the 10-core mapping each core was responsible for two FIR branches, and on the 7-core mapping there were five branches per core. The five non-FIR actors were always assigned to dedicated cores in the 7-, 10- and 15-core mappings. The number of branches used was allowed to be reconfigured for every 64 kilosamples. The number of active branches varied equally between 2 and 10. Fig. 10 shows the throughput of the predistortion application on the two platforms as a function of core count and fifo buffer size. Average throughput and standard deviation have been computed from 10 iterations.

### 4.3 GPU throughput measurement

GPU performance provided by the proposed design flow was measured on the i7 platform (See Table 1) that was equipped with the NVidia GTX 750Ti, a mid-range desktop GPU. The Gaussian computation, median computation and thresholding actors were mapped to the GPU, and each of them was set to be processed with a global work size of 76800, and the division of work items

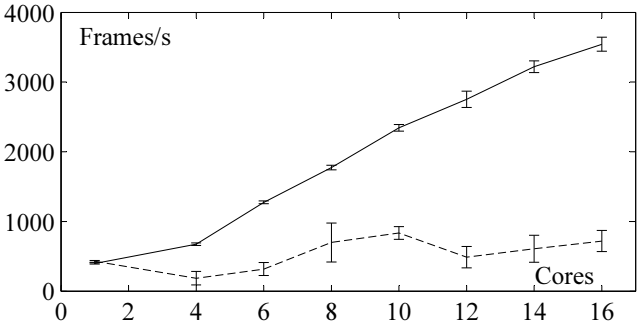


**Fig. 8** Throughput of motion detection on the Xeon platform. Solid line = proposed; dashed line = Orcc multithreaded.

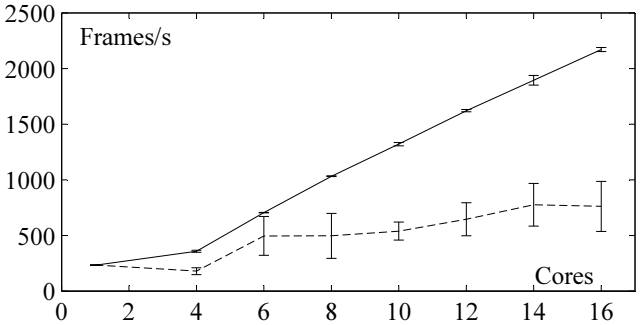


**Fig. 9** Throughput of motion detection on the Opteron platform. Solid line = proposed; dashed line = Orcc multithreaded.

to work groups was left to the OpenCL drivers. Performance was measured with a wide variety of GPU buffer sizes between 300 kilobytes and 4.7 Megabytes, but no measurable performance differences were observed due to buffer size changes. On average, the measured throughput was 5170 frames per second (fps) with a standard deviation of 198 fps. This performance was measured from the complete application and includes SSD disk reads and writes, computations on CPU cores, CPU-GPU-CPU data transfers and GPU processing time.

### 4.4 Analysis of results

As Fig. 8 and Fig. 9 show, the proposed methodology is fully capable of utilizing the available cores on the Xeon and Opteron multicore platforms, yielding almost linear speedup as a function of core count. In contrast, the reference design flow is capable of producing only little speedup with considerable variance between executions.

The more elaborate experiments conducted on the predistortion application (Fig. 10) show the speedup as a function of *communication buffer size* for core counts of 1, 7, 10 and 15. Each of the subfigures a) through h) show that the proposed design flow is capable of increasing predistortion throughput as the number of cores increases, starting from 6 Msps (Mega samples per second) for 1 Xeon core, to almost 30 mega samples per second on 15 cores. In analyzing the speedup, it is necessary to keep in mind that in the predistortion application three cores are always reserved to the source, sink and configuration actors that do not contribute to throughput. Similarly, due to its dynamic behavior, only six out of ten FIR filters are simultaneously in operation on average. Finally, similar to the motion detection application, the reference design flow can only yield little or no speed-up as the number of cores is increased.

The GPU experiment on motion detection shows that offloading computations to a GPU can provide even better performance than a rather high number (e.g. 16) of general purpose cores: on the i7 and the GTX 750Ti the performance goes beyond 5000 fps. Unfortunately, it was not possible to compare the GPU performance against any reference, since the only other design flow for GPU acceleration of RVC-CAL programs [21] has not yet been publicly released.

It is worthwhile to point out that as Fig. 10a) and Fig. 10b) show, the reference design flow still provides higher throughput in single core execution regardless of communication buffer size.

As a summary, our experiments show that the proposed design flow truly enables efficient execution of dataflow programs on multicore platforms and GPUs, and clearly outperforms the state-of-the-art.

## 5 Discussion

The motion detection application shows that the proposed design flow clearly outperforms the state-of-the-art approach on multicore platforms when the number of cores used is more than one. At best, the proposed design flow provides almost 5× higher throughput, and always a considerably smaller standard deviation. With the dynamic predistortion application the proposed approach outperforms the state-of-the-art when the number of cores used is more than one. At most, the throughput advantage is 2.5× higher and the standard deviation is of a magnitude smaller when compared to the reference approach. Acknowledging that the reference design flow [30] yields higher throughput on single-core and with small buffer sizes, it is still clear that for execution on numerous cores the proposed design flow is superior.

The DAL framework used by the proposed design flow, and the multithreaded code generated by Orcc have substantial differences in inter-process communication, which partially explain the performance gap: a) the DAL framework maps processes to threads [25] that are put to sleep when communication is blocked and awakened on arrival of data, whereas Orcc actors need to poll

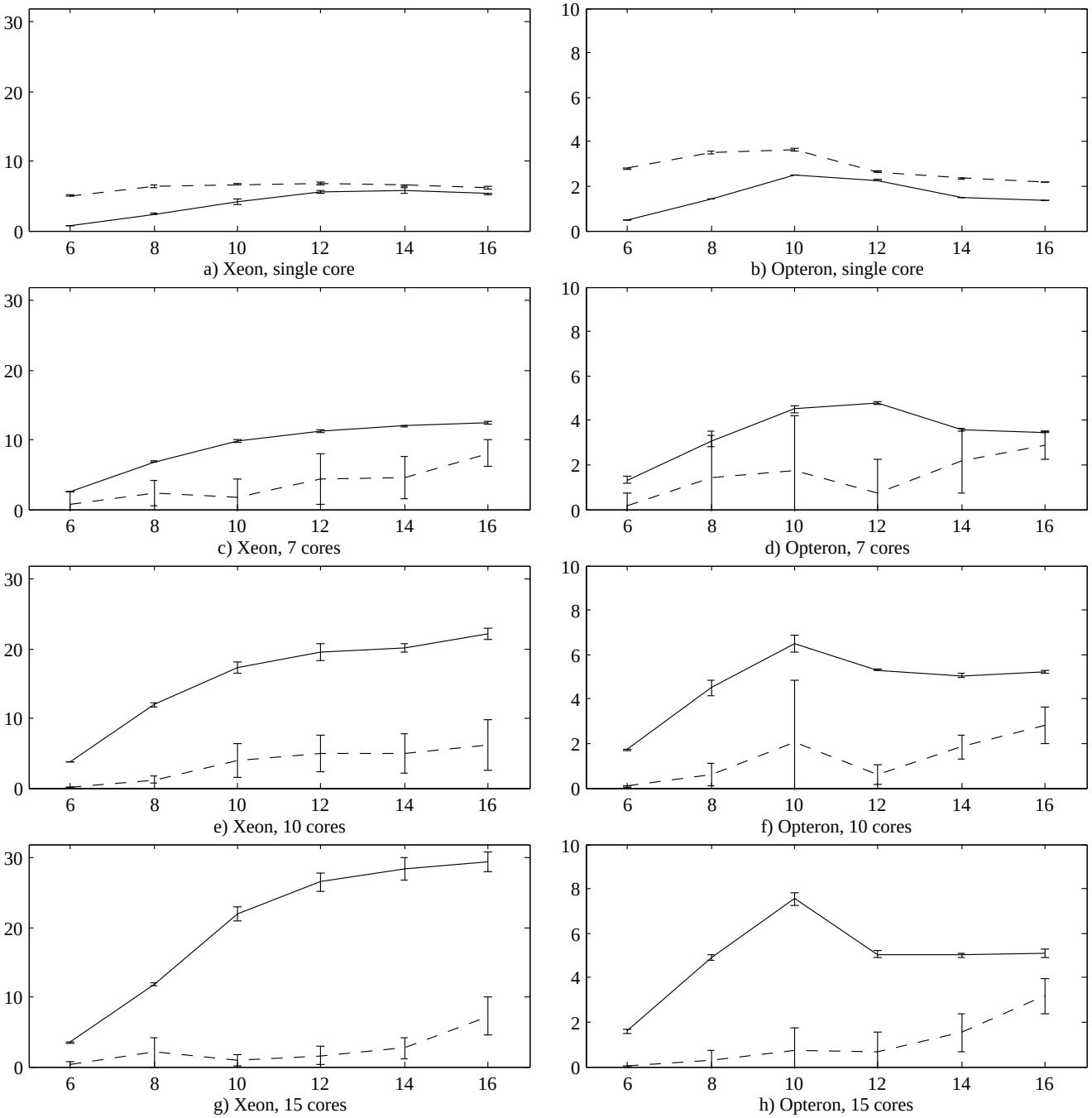


**Fig. 10** Throughput of predistortion. The X axis is $log_2$ of buffer size, Y axis is throughput in Megasamples / s. Solid line = proposed; dashed line = Orcc multithreaded.

input FIFOs for token availability; b) DAL copies data to/from FIFOs in blocks whereas Orcc-generated code performs FIFO access in the granularity of individual tokens. A full analysis on the reasons of performance differences and possibilities for unifying the advantages of both approaches are however outside the scope of this work and present a clear direction for further studies.

## 6 Conclusion

This work presented a novel design flow for executing dynamic dataflow applications on multicore platforms and GPUs. Experimental evaluation on two applications and three platforms revealed that the proposed approach clearly outperforms the state-of-the-art approach when high performance applications are considered.

## 7 Acknowledgements

The authors thank anonymous reviewers for their constructive comments. This work was funded by the Academy of Finland project UNICODE.

## References

1. Abdelaziz, M., Ghazi, A., Anttila, L., Boutellier, J., Lähteensuo, T., Lu, X., Cavallaro, J. R., Bhattacharyya, S. S., Juntti, M., and Valkama, M. (2013). Mobile transmitter digital predistortion: Feasibility analysis, algorithms and design exploration. In *Proc. Asilomar Conference on Signals, Systems and Computers*, pages 2046–2053.
2. Amer, I., Lucarz, C., Roquier, G., Mattavelli, M., Raulet, M., Nezan, J.-F., and Déforges, O. (2009). Reconfigurable video coding on multicore. *IEEE Signal Processing Magazine*, 26(6):113–123.
3. Bezati, E., Casale Brunet, S., Mattavelli, M., and Janneck, J. (2013). Synthesis and optimization of high-level stream programs. In *Proc. Electronic System Level Synthesis Conference*, pages 1–6.
4. Bezati, E., Thavot, R., Roquier, G., and Mattavelli, M. (2014). High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms. *Journal of Real-Time Image Processing*, 9(1):251–262.
5. Bilsen, G., Engels, M., Lauwereins, R., and Peperstraete, J. (1996). Cycle-static dataflow. *IEEE Transactions on signal processing*, 44(2):397–408.
6. Boutellier, J. and Ghazi, A. (2015). Multicore execution of dynamic dataflow programs on the Distributed Application Layer. In *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 893–897.
7. Boutellier, J., Martin Gomez, V., Lucarz, C., Silvén, S., and Mattavelli, M. (2009). Multiprocessor scheduling of dataflow models within the reconfigurable video coding framework. In *Proc. Conference on Design and Architectures for Signal and Image Processing*.
8. Boutellier, J. and Nyländen, T. (2015). Programming graphics processing units in the RVC-CAL dataflow language. In *Proc. IEEE Workshop on Signal Processing Systems*, pages 1–6.
9. Chavarrias, M., Pescador, F., Garrido, M. J., Juarez, E., and Sanz, C. (2015). A multicore DSP HEVC decoder using an actor-based dataflow model. In *Proc. IEEE International Conference on Consumer Electronics*, pages 370–371.
10. Chavarrias, M., Pescador, F., Juarez, E., and Garrido, M. J. (2014). An automatic tool for the static distribution of actors in RVC-CAL based multicore designs. In *Proc. Conference on Design of Circuits and Integrated Circuits*, pages 1–6.
11. Eker, J. and Janneck, J. W. (2003). CAL language report. Technical Report UCB/ERL M03/48, UC Berkeley.
12. Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., and Schaa, D. (2012). *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition*. Morgan Kaufmann.
13. Gautier, T., Lima, J. V. F., Maillard, N., and Raffin, B. (2013). XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proc. IEEE International Symposium on Parallel Distributed Processing*, pages 1299–1308.
14. Gebrewahid, E., Yang, M., Cedersjö, G., Abdin, Z. U., Gaspes, V., Janneck, J. W., and Svensson, B. (2014). Realizing efficient execution of dataflow actors on manycores. In *Proc. IEEE International Conference on Embedded and Ubiquitous Computing*, pages 321–328.
15. Gorin, J., Yviquel, H., Prêteux, F., and Raulet, M. (2011). Just-in-time adaptive decoder engine: A universal video decoder based on MPEG RVC. In *Proc. ACM International Conference on Multimedia*, pages 711–714.
16. Hoshino, T., Maruyama, N., Matsuoka, S., and Takaki, R. (2013). CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 136–143.
17. Kahn, G. (1974). The semantics of a simple language for parallel programming. In Rosenfeld, J. L., editor, *Information processing*, pages 471–475, Stockholm, Sweden. North Holland, Amsterdam.
18. Lee, E. A. and Messerschmitt, D. G. (1987). Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245.
19. Lee, E. A. and Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801.
20. Lucarz, C., Roquier, G., and Mattavelli, M. (2010). High level design space exploration of RVC codec specifications for multi-core heterogeneous platforms. In *Proc. Conference on Design and Architectures for Signal and Image Processing*, pages 191–198.
21. Lund, W., Kanur, S., Ersfolk, J., Tsiopoulos, L., Lilius, J., Haldin, J., and Falk, U. (2015). Execution of dataflow process networks on OpenCL platforms. In *Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 618–625.
22. Mattavelli, M., Amer, I., and Raulet, M. (2010). The Reconfigurable Video Coding standard [standards in a nutshell]. *IEEE Signal Processing Magazine*, 27(3):159–167.
23. Sbîrlea, A., Zou, Y., Budimlíc, Z., Cong, J., and Sarkar, V. (2012). Mapping a data-flow programming model onto heterogeneous platforms. In *Proc. ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, pages 61–70.
24. Scherer, T. (2013). Executing process networks on heterogeneous platforms using OpenCL. Master's thesis, ETH Zürich.
25. Schor, L., Bacivarov, I., Rai, D., Yang, H., Kang, S.-H., and Thiele, L. (2012). Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proc. International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 71–80.
26. Schor, L., Tretter, A., Scherer, T., and Thiele, L. (2013). Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL. In *IEEE Symposium on Embedded Systems for Real-time Multimedia*, pages 41–50.
27. Tretter, A., Boutellier, J., Guthrie, J., Schor, L., and Thiele, L. (2015). Executing dataflow actors as Kahn processes. In *International Conference on Embedded Software (EmSoft)*, pages 105–114.
28. Wipliez, M., Roquier, G., and Nezan, J.-F. (2011). Software code generation for the RVC-CAL language. *Journal of Signal Processing Systems*, 63(2):203–213.
29. Yviquel, H., Casseau, E., Raulet, M., Jääskelainen, P., and Takala, J. (2013a). Towards run-time actor mapping of dynamic dataflow programs onto multi-core platforms. In *Proc. International Symposium on Image and Signal Processing and Analysis*, pages 732–737.
30. Yviquel, H., Casseau, E., Wipliez, M., and Raulet, M. (2011). Efficient multicore scheduling of dataflow process networks. In *Proc. IEEE Workshop on Signal Processing Systems*, pages 198–203.

31. Yviquel, H., Lorence, A., Jerbi, K., Cocherel, G., Sanchez, A., and Raulet, M. (2013b). Orcc: Multimedia development made easy. In *Proc. ACM International Conference on Multimedia*, pages 863–866.
32. Yviquel, H., Sanchez, A., Jääskeläinen, P., Takala, J., Raulet, M., and Casseau, E. (2015). Embedded multi-core systems dedicated to dynamic dataflow programs. *Journal of Signal Processing Systems*, 80(1):121–136.