Multithread Accelerators on FPGAs: A Dataflow-Based Approach

Francesco Ratto
Università degli Studi di Cagliari, Italy

Stefano Esposito
Università degli Studi di Cagliari, Italy

Carlo Sau
Università degli Studi di Cagliari, Italy

Luigi Raffo
Università degli Studi di Cagliari, Italy

Francesca Palumbo
Università degli Studi di Sassari, Italy

Abstract

Multithreading is a well-known technique for general-purpose systems to deliver a substantial performance gain, raising resource efficiency by exploiting underutilization periods. With the increase of specialized hardware, resource efficiency became fundamental to master the introduced overhead of such kind of devices. In this work, we propose a model-based approach for designing specialized multithread hardware accelerators. This novel approach exploits dataflow models of applications and tagged tokens to let the resulting hardware support concurrent threads without the need to replicate the whole accelerator. Assessment is carried out over different versions of an accelerator for a compute-intensive step of modern video coding algorithms, under several feeding configurations. Results highlight that the proposed multithread accelerators achieve a valuable tradeoff: saving computational resources with respect to replicated parallel single-thread accelerators, while guaranteeing shorter waiting, response, and elaboration time than a unique single-thread accelerator multiplexed in time.

2012 ACM Subject Classification Computer systems organization → Data flow architectures; Computing methodologies → Concurrent algorithms; Hardware → Best practices for EDA

Keywords and phrases multithreading, dataflow, hardware acceleration, heterogeneous systems, tagged dataflow

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.6

Funding Prof. Palumbo is grateful to the University of Sassari that supported her studies on this topic through the "fondo di Ateneo per la ricerca 2020".

1 Introduction

With the end of Moore's Law and Dennard Scaling, high-level specification and hardware specialization have become fundamental to keep on improving performance [10]. Specialized hardware has already demonstrated its capability of generating performance and efficiency gains exploiting data and instructions specialization, parallelism, local memories and reduced overhead [7]. A popular solution for matching specialized-hardware performance with the flexibility of general-purpose computing are Heterogeneous Systems-on-chip, where multiple processors are integrated with reconfigurable logic and other components [4]. Here computational-intense tasks can be delegated to specialized hardware to improve performance and/or efficiency [9].

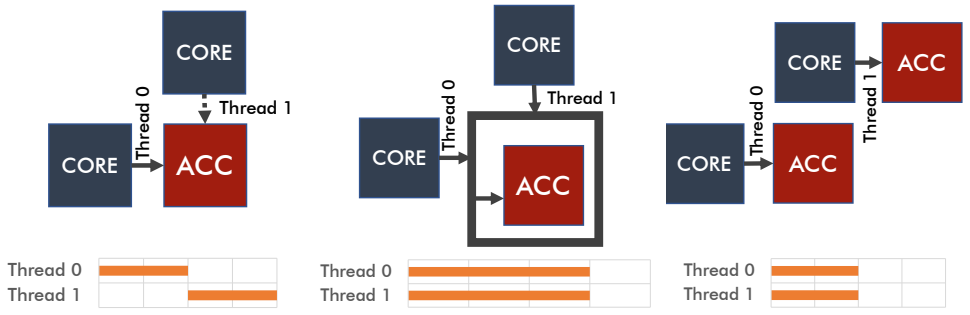
13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022).
Editors: Francesca Palumbo, João Bispo, and Stefano Cherubin; Article No. 6; pp. 6:1–6:14

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In particular, dealing with multiple host sources, being them local cores of a multicore cluster or remote processes from other connected devices, a single specialized hardware accelerator multiplexed in time (Fig. 1a) can be a bottleneck, lowering down the whole integrated-/connected-system performance. On the other hand, replicating the accelerator for each host (Fig. 1c) that can potentially delegate processing could easily end up in a huge waste of resources. The proposed solution is based on a single accelerator supporting multiple, potentially concurrent, threads (Fig. 1b). This solution can provide halfway benefits, delivering a tradeoff between performance and resource utilization.

Tagging tokens in dataflow models is exploited to differentiate threads flowing into the datapath: the state of the different threads is stored into multiple dedicated sequential resources, while combinational ones are shared among them. To allow the exchange of tagged tokens, FIFO channels supporting out-of-order access have been designed. With the defined design approach, a hardware accelerator implementing a video coding use case has been developed. The proposed solution has been tested on a Xilinx Artix-7 device, demonstrating that a significant performance gain can be obtained at the cost of a limited resource overhead. This paper is focused on presenting the approach to design the accelerator in Fig. 1b, while the integration with the host processor will come as future development.

The rest of this paper is organized as follows. An overview of the proposed solutions for accelerators multithreading is provided in Sect. 2, followed by our approach, which is described in Sect. 3. Then, experimental results are shown and discussed in Sect. 4, before concluding in Sect. 5 with some final remarks.



(a) Single-thread. **(b)** Multithread. **(c)** Replicated.

Figure 1 Schematic of the three possible configurations described, and a sketch of their time evolution when two threads have to be elaborated by the accelerator.

2 Related Work

Several solutions have been proposed for tackling the challenge of implementing multithread hardware accelerators on reconfigurable fabric.

Dynamic reconfiguration based accelerators

Some of them [20, 21, 17] exploit the dynamic partial reconfiguration feature of modern FPGAs. These works focus mainly on the interaction between the host processor and the reconfigurable accelerators, and on the management of the system architecture for loading partial bitstreams to configure the programmable logic. These approaches must be

integrated with other design flows for single-thread hardware accelerators to generate the partial bitstream for each function to be accelerated. The main performance limitations are the time and energy needed to load the partial bitstream for the successive task [20], or when a configuration miss occurs [21, 17], i.e. there is no available slot on the programmable logic already configured with the required bitstream. In our approach the hardware overhead for dynamic partial reconfiguration is not required, and a model-based architecture is adopted for developing the accelerator.

Software APIs based accelerators

Also High-level Synthesis has been exploited to design multithread accelerators [16, 6, 5]. These works focus on generating the RTL description of an accelerator starting from a C-based description integrated with widely used multithread software APIs, like CUDA [16], OpenCL [6], Pthread and OpenMP [5]. In these solutions a dedicated hardware accelerator is generated for each coarse-grained thread, so the number of threads must be known at compile time. The accelerators, which may execute the same or different functions, are synchronized to respect the original software behavior. In our work the maximum number of threads must be known at compile-time as well, but the resource overhead is mitigated through resource sharing.

Computing resource sharing accelerators

Another solution based on HLS is Nymble [11]. In this case a unique multithread accelerator is generated. The architecture of a Nyble accelerator is made up of a control unit, called Dynamic Stage Controller (DSC), and a datapath. To support multithreading, the DSC is extended with state replicas, while queues are inserted in the datapath for buffering pipelined results of unstoppable blocks, e.g. multicycle memory accesses.

Avoiding compute-units replication by adding the required hardware for multithreading support makes Nymble the most related to our work. However, there are some significant differences in the input specification and in the resulting accelerator. Nymble takes as input a sequential description of an application, while we use a dataflow one. Nymble generates an accelerator made up of a control unit and a datapath, while our accelerators are made up of a network of modules that exchange data through FIFO channels.

Advantages of Dataflow models

Modularity and parallelism make dataflows a widely adopted specification for both software [1] and hardware [15, 12] design. They turn out to be particularly suitable for describing low-power streaming applications to be accelerated [2, 8].

Other works have made use of token labeling and tagged-dataflow already. So far, however, the use has been limited to software-oriented solutions, e.g., for high-level parallel language definition [14], massive data processing [3] or loop optimization [18].

3 Approach and Architecture

In this section we present a novel model-based approach that, starting from the single-thread dataflow specification of an application and without explicit need of data synchronization, allows to design a corresponding multithread hardware accelerator through token labeling (Section 3.1) .

3.1 Tagging tokens for multithreading

Dataflow models [13] are basically networks of processing elements, the actors, which exchange chunks of data, the tokens, through point-to-point buffered communication channels managed in a First-In-First-Out (FIFO) manner. Execution of operations within actors, the actions, is token mediated, meaning that it only depends on tokens/free-space availability on incoming/outgoing FIFOs. Hardware accelerators can be derived from dataflows mapping each actor directly into a module, while FIFOs and tokens flow ensures the execution correctness without the need of a centralized control.

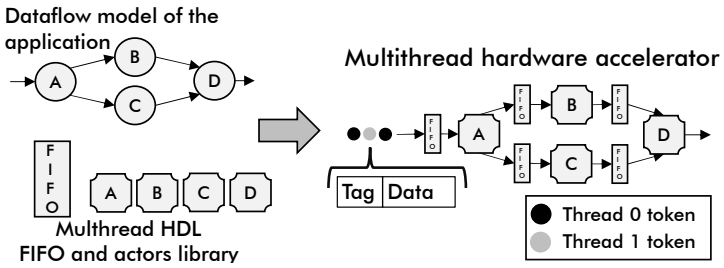
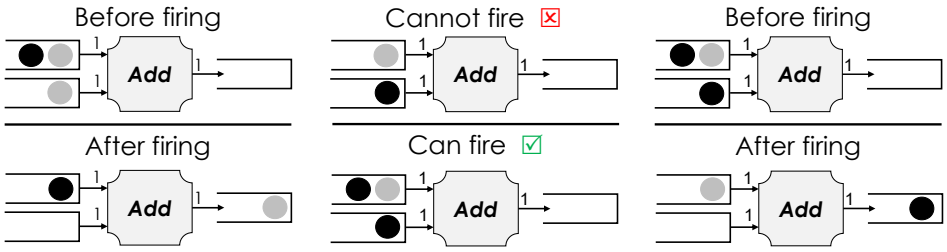


Figure 2 With the proposed approach, a dataflow model can be mapped into a multithread hardware accelerator using multithread actors and FIFOs.

To support hardware multithreading we added a tag to each token. The tag indicates which thread the token belongs to, and so it allows to differentiate tokens of different threads. Once tokens are tagged, FIFOs and actors have to meet a set of requirements to implement a multithread accelerator corresponding to the described application (see Fig. 2). These requirements are necessary to ensure the correct flow of tokens:

1. A firing actor has to tag the output tokens with the same tag of the input ones (Fig. 3a).
2. The firing rules have to be adjusted so that only tokens belonging to the same thread are able to fire the execution. Then, any actor must be enabled to fire only when matching token(s) are available in its input channel(s) (Fig. 3b).
3. FIFOs must provide semi-out-of-order read, letting the reading actors choose among the first token of each flow of execution. This feature is necessary to prevent deadlocks in actors with multiple input ports, as depicted in Fig. 3c.



(a) Requirement 1. **(b)** Requirement 2. **(c)** Requirement 3.

Figure 3 Representation of the three requirements using the *Add* actor, whose token rates are indicated on the arrows. In each image, the status of input and output buffers is depicted using different colors for tokens belonging to different threads.

In the rest of this section it is shown how the above requirements drive the design of multithread interfaces (Section 3.2), FIFOs (Section 3.3) and actors (Section 3.4).

3.2 Multithread FIFO Interface

To support tagged-token exchange, the first challenge to be faced is the FIFO read/write interface definition. In the proposed implementation, these interfaces are customizable using two parameters:

- `DATA_WIDTH`, the number of information bits in a token;
- `N_THREADS`, the number of supported threads, which determines the number of bits of the tag.

The `write_interface` and the `read_interface` are both made up of three corresponding signals:

- `din` and `dout` - They are used to transmit the content of a token (tag and data);
- `full` and `empty` - They represent the status of the FIFO and are N_THREADS-bit wide, as each bit represents the status on a specific thread (see Requirement 2);
- `write` and `read` - `write` is 1-bit wide, because a FIFO can determine the token's thread from its tag. `read`, instead, is N_THREADS-bit wide to allow the reading actor, that drives the signal, to choose which thread it wants to read from (see Requirement 3).

The resulting connection scheme between a FIFO and the surrounding actors is depicted in Fig. 4.

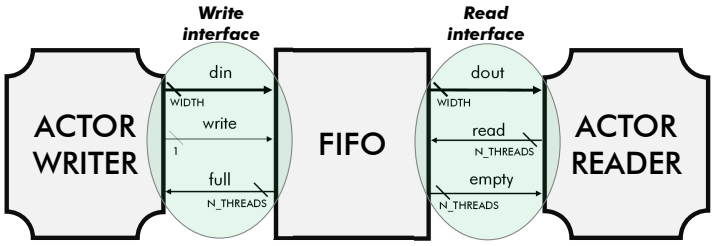


Figure 4 Schematic of the connection between a FIFO and two actors using the `write_interface` and the `read_interface`. `WIDTH` is the sum of `DATA_WIDTH` and the tag width, which is $log_2(N\_THREADS)$.

3.3 Multithread FIFO

To support tagged-dataflow computation, FIFOs must preserve the order of the tokens within each thread and allow out-of-order reading among tokens of different threads (see Requirement 3). Given these constraints, and adopting the interfaces described in Sect. 3.2, we developed two possible implementations of a multi-thread FIFO.

3.3.1 Separated-memory FIFO

It uses a dedicated memory for each thread. As shown in Fig. 5, this FIFO is composed of a bank of N_THREAD dual-ported `RAMs`, which allow simultaneous read/write operations. These RAMs are managed by a `Control Logic` that drives also the empty and full signals. Internally the `Control Logic` uses a set of registers, two for each thread, for storing pointers to the next read and the next write location. To evaluate the status of the FIFO an additional register for storing the latest performed operation is needed. Finally, there is a combinational module, the `Tag reader`, which forwards the data field of the token, while the tag is used to select the right memory to use by asserting its `wr_en`. When a token is read, `Tag writer` selects the right memory and appends the tag to complete the output token. Indeed, the tag is not stored but computed considering the read signal.

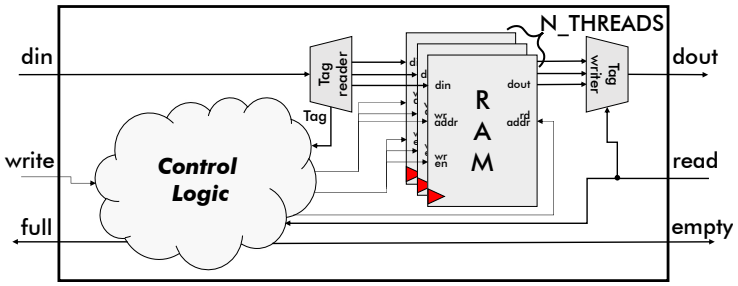


Figure 5 Schematic of the Separated-memory FIFO. In this FIFO each thread has a dedicated `RAM` where tokens are stored by the `Control Logic`.

3.3.2 Address-memory FIFO

it uses a shared `Data memory` for storing tokens of all the threads and an `Address memory` to keep track of the order. In this way the `Data memory` slots are thread agnostic and can be used without any tag restriction, but a complex `Control Logic` and an additional memory are needed, as shown in Fig. 6. The `Control Logic` uses more registers than in the previous implementation: one for storing the next location to be written, one for storing the last written location of the `Address memory`, one for each thread for storing the next location to read, one status register for storing which locations are currently used, and another one for each thread to count the contained tokens. These registers are necessary to manage the two memories and to evaluate the status of the FIFO. Note that the FIFO would become full for all the threads simultaneously, but still, the `empty` signal must be driven independently for each thread. As before, `Tag reader` separates tag and data field, while `Tag writer` picks up the requested data and appends the tag (not stored with the data).

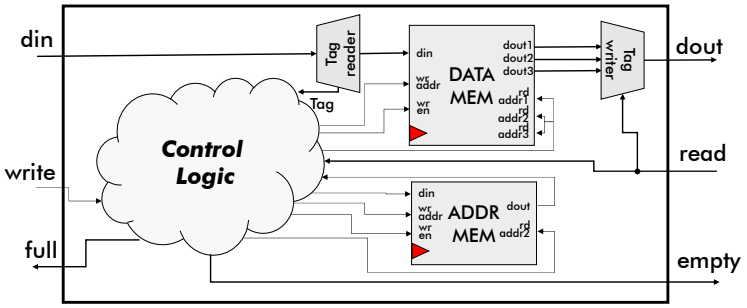


Figure 6 Schematic of the Address-memory FIFO. In this FIFO all the tokens are stored in the shared `Data memory`, managed by the `Control Logic` with the support of the `Address memory`.

3.4 Multithread Actor

To conclude this section, a hardware architecture for multithread dataflow actors is discussed. Let's start considering an actor having one input port and one output port. The proposed structure (Fig. 7) is based on the state-action model of a dataflow actor. The set of possible actions an actor is able to perform (`Actions`) and the logic to compute the next state (`State update`) are mapped into combinational logic, which is used to elaborate tokens of all the threads one at a time. Handling one token at a time, there is no need for hardware replication as the number of supported threads increases. On the contrary, the sequential logic used for storing the state must be replicated to keep track of the evolution of the system.

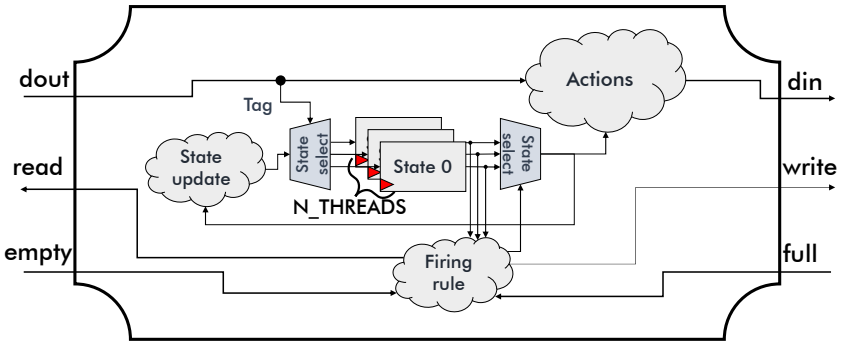


Figure 7 Schematic of an actor supporting tagged tokens. The combinational logic `Firing rule` must consider only matching tokens. Combinational logic, `Actions` and `State updates`, is shared, while the sequential one `State` is replicated for each supported thread.

The approach can be extended to actors with multiple ports without significant modifications. Any dataflow actor that has to read from two input ports to fire would wait until the empty signal is deasserted in both ports. In a tagged-token actor, a multi-bit empty signal is used to check the availability of a matching pair of tokens. The same occurs analogously for more than two ports.

In the given model, an actor could be able to fire for different threads. Since simultaneous multiple firing is not supported by actors, a priority scheme is needed. In this work, we opted for a static priority assignment, to keep the logic as simple as possible and avoid an excessive resource overhead.

4 Experimental Results

The proposed design approach allows a flexible time multiplexing of the computational resources of an accelerator. In fact, each actor can carry out its computation according to token availability while supporting multiple threads. In this section experimental results considering a video coding use case (Sect. 4.1) are presented and discussed both in terms resource utilization (Sect. 4.2) and execution analysis (Sect. 4.3).

4.1 Use Case and Setup

A video coding use case involving fractional pixel interpolation for the luma component, used during motion estimation and compensation phases of the HEVC codec, has been used for the experimental validation. The interpolator takes as input one image block and produces as output the same image block shifted by fractional pixels positions. It is implemented through two cascaded 8-tap digital filters, one for the horizontal direction and one for the vertical direction. Two different architectures have been developed for the interpolator:

- **Baseline**: the filter takes 1 pixel per cycle, performs 1 8-tap horizontal filtering, buffers 8 block lines, and performs 8-tap vertical filtering, producing 1 pixel per cycle on the output side.
- **Matrix**: the filter takes 8 pixels per cycle, performs 8 parallel 8-tap horizontal filterings, buffers 8 block lines, and performs 8 parallel 8-tap vertical filterings, producing 8 pixels per cycle on the output side.

The aim of the two versions is to assess the proposed multithread accelerator architectures with applications presenting a different degree of parallelism.

Starting from these two versions of the interpolator, different evaluation set-ups have been derived:

- **Single**: a single-thread accelerator that can execute exclusively 1 thread before starting the processing of the next one (Fig. 1a);
- **Tagged2, Tagged4**: the proposed multithread accelerator that can support respectively 2 or 4 threads (Fig. 1b);
- **Parallel2, Parallel4**: a multithread accelerator made replicating the Single one respectively 2 or 4 times (Fig. 1c).

Resource utilization (Sect. 4.2) is gathered leveraging post-synthesis reports retrieved using Vivado Xilinx, targeting an Artix-7 device (xc7a100t). Time performance (Sect. 4.3) are assessed through behavioral SystemVerilog simulations using Vivado Simulator.

4.2 Resources Analysis

4.2.1 FIFOs

The two types of FIFO, introduced in Sect 3.3, are interchangeable without modifications in the accelerator model nor in the actors. A design-space exploration varying the three parameters of the FIFOs (`N_THREADS`, `DEPTH` and `DATA_WIDTH`) has been performed to evaluate their resource utilization and, then, which of the two is preferable. Results are reported in Fig. 8.

The Separated-memory FIFO utilizes fewer LUTs and FFs than the Address-memory FIFO with any set of parameters, due to the simpler control logic. On the other hand, the latter needs fewer LUTRAMs compared with the Separated-memory FIFO that has `N_THREADS` · `DEPTH` slots, while the Address-memory FIFO has only `DEPTH` shared slots, independently from the number of supported threads. Also, the FFs overhead in Address-memory FIFO is independent of the `DATA_WIDTH`, but grows with `DEPTH`, as the size of most control-logic registers is linearly proportional to it. In the end, to choose between the two implementation one should consider design parameters as well as resource availability in the target device.

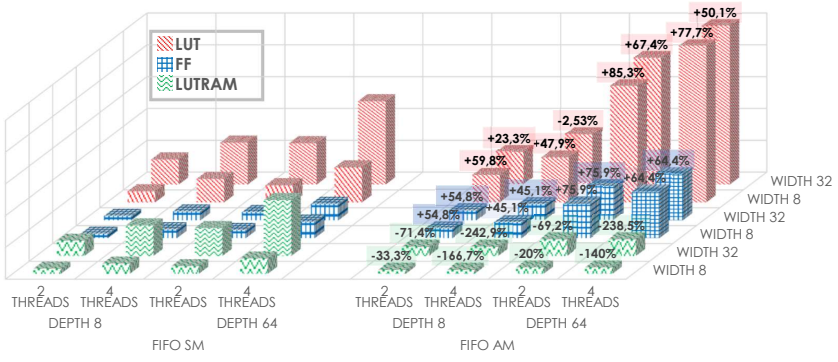


Figure 8 Resource-utilization comparison between the Separted-memory FIFO (on the left) and the Address-memory FIFO (on the right) varying design parameters: `N_THREADS`={2, 4}, `DEPTH`={8, 64}, `DATA_WIDTH`={8, 32}. On top of the right-side columns the percentage variation with respect to the corresponding left-side column is reported.

4.2.2 Actors

In the proposed approach, what affects most actors' resource overhead is the ratio between combinational and sequential logic used to implement them. Three actors have been selected to investigate this aspect:

- **Add:** it performs the sum of two input tokens → **combinational**;
- **Mul:** it performs the multiplication by a fixed stored coefficient → **combinational/sequential**;
- **Line-buffer:** it stores and forwards one row of the input block → **sequential**.

From Fig. 9 it can be seen that the Add actor is purely combinational, as only LUTs are used to synthesize it. In this case, the overhead to support multiple threads is limited. Moving to the Mul actor, it can be noticed that the overhead to support multiple threads on FFs is, as expected, equal to replicating the resource. However, on LUTRAMs there is no overhead at all: the target technology plays an important role in that. Indeed, a LUTRAM of the target board can implement a Single-Port 32x1-bit RAM and, in turn, store more data than it is actually required. So, being the LUTRAM more efficiently utilized, the extension to support multiple threads comes from free. On the line-buffer, which uses a larger memory for storing a row of input, a larger overhead occurs, especially for LUTRAMs and FFs.

As a general rule, we can state that the more an actor has a combinational behavior, the less the additional logic for supporting multithreading impacts on the overall resource utilization.

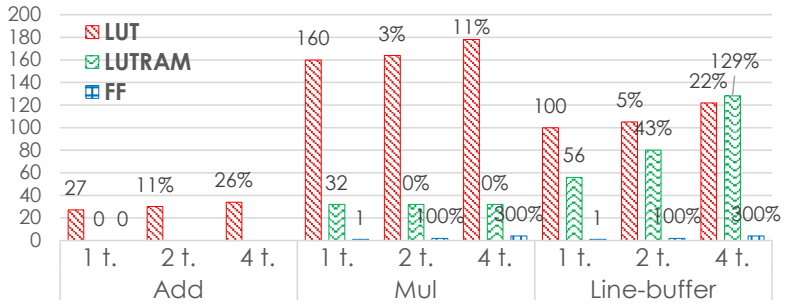


Figure 9 Resource utilization of Add, Mul, and Line-buffer actors supporting 1, 2, or 4 threads. On the 1-thread columns the absolute value is reported, on the others the variation with respect to that one.

4.2.3 Overall filters results

The architectures presented in Sect. 4.1 have been synthesized to evaluate the impact of the proposed approach on resource utilization. Separated-memory FIFOs with minimal size have been used. The minimal size that ensures reaching the end of the computation has been evaluated through a SystemC simulation of the system generated with CAPH [19], which automatically reports the maximal usage of each buffer.

As it can be seen from Fig. 10, the two versions, Baseline and Matrix, have a similar trend in resource utilization when multithreading is supported. This trend is coherent with what was observed on FIFOs and actors (Sect. 4.2.1 and 4.2.2). Moreover, DSP sharing can be noticed in the Tagged accelerators, which uses the same amount of the Single one and 75% less than the Parallel4. As DSPs are merely used for computation, this is completely consistent with the proposed approach. In any case, the Tagged accelerators proves to be the promised tradeoff among Single and Parallel implementations of the same accelerator, consistently with what is depicted in Fig. 1.

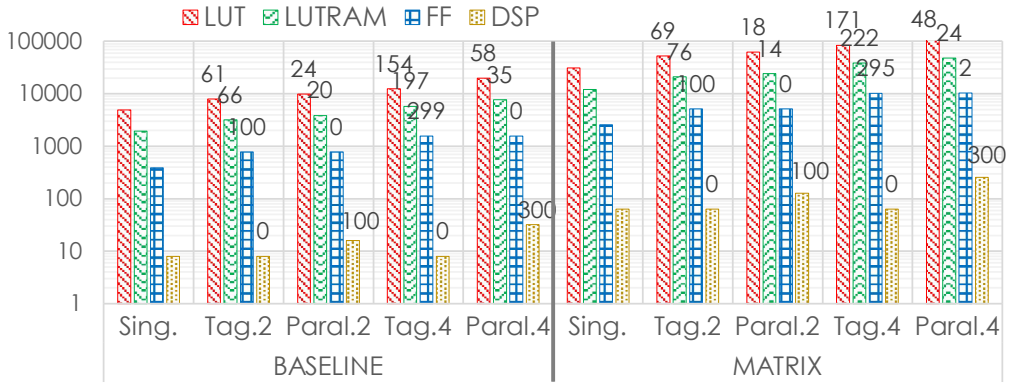


Figure 10 Resource utilization of all the accelerators described in Sect. 4.1. Data are reported on a logarithmic scale. On top of Tagged columns percentage variation referred to Single, and on top of Parallel columns percentage variation referred to the corresponding Tagged one.

4.3 Execution Analysis

Supporting multiple threads to run concurrently on an accelerator brings many benefits in terms of time performance. To evaluate them, the following time intervals are considered (please notice the color coding that is then used in the dissertation below):

- **Waiting time:** time between the request to access the accelerator and the first input token read (yellow dotted segment);
- **Response time:** time between the request to access the accelerator and the first output token written (sum of yellow segment and blue vertically-stripped segment);
- **Elaboration time:** time between the request to access the accelerator and the last output token written (sum of the three segments).

Time performance is strictly correlated to the considered scenario. Anyhow, we tried to select some significant cases to let the main pros and cons of the proposed solution come up.

4.3.1 Same-size

First, it is analyzed a case where the same computation, filtering a 16x16 image block, has to be carried out for each thread under execution. What will affect most the final results is the arrival time of requests to use the accelerator. A corner negative case for the Tagged accelerator happens when these requests do not overlap over time, i.e. a new request arrives when the previous thread has already ended the computation. Of course, in this scenario a Tagged accelerator cannot have any gain on a Single one. Nevertheless, there is not performance loss due to multithreading support, as they perform in the same way. In addition, the Tagged accelerator is not outperformed by the Parallel one.

A corner positive case considers requests that arrive almost simultaneously (they differ by one clock cycle). Going from Single to Tagged accelerators in Fig. 11, and observing average values with 2 threads (line Avg 2), it can be noticed that the waiting time is almost nullified, while the response time is reduced (up to -43% in the Matrix case).

On the Baseline accelerator with 2 threads (line Avg 2) the elaboration time grows in the Tagged accelerator, since sharing the logic and guaranteeing quick access tend to balance the elaboration time of each thread. But, if 4 concurrent threads are running (line Avg 4), even the average elaboration time is reduced. On the Matrix accelerator greater advantages both in response and elaboration time than on the Baseline filter can be noticed with 2 and

4 threads. Indeed, the pipeline composed of two filtering stages is not always completely fulfilled in the Matrix implementation, due to the end of line steps. This allows the Tagged accelerator to better use the available resources among different threads than the Single accelerator.

With requests arriving almost simultaneously, Parallel-accelerators average results are equal to Single `Thread 0` line in Fig. 11, as each thread can run independently on a dedicated accelerator. Parallel accelerators outperform Tagged ones, as the resource replication is fully used.

The obtained behavior shows how the proposed multithread approach can successfully exploit the full potential of the available resources, while a Single accelerator does not. It should be noticed that real cases lie between the corner negative and positive cases here described.

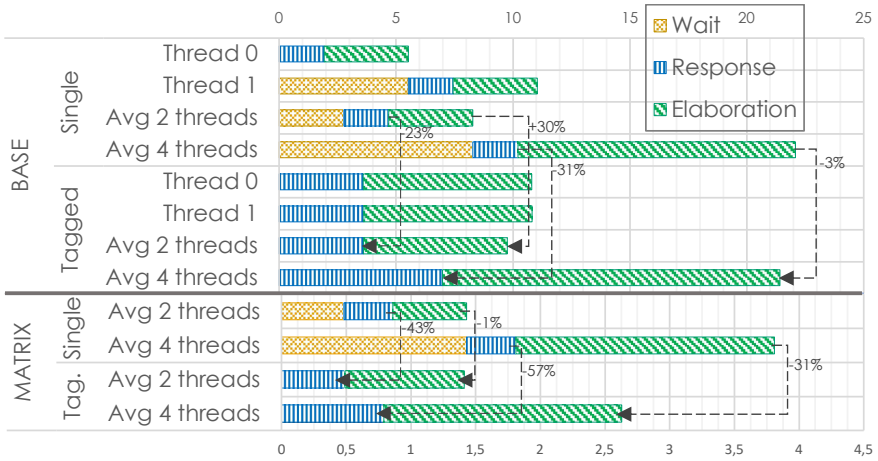


Figure 11 Time performance with same-size elaborations. `Thread 0` and `Thread 1` are the evolution of the threads when 2 of them are elaborated. `Avg 2 threads` and `Avg 4 threads` are the average time performance when respectively 2 or 4 threads are considered. Arrows show the percentage variation in the Tagged accelerators compared with the corresponding Single ones. Timescales are in us.

4.3.2 Different-size

When dealing with multiple threads, priority assignment may play a role. To investigate this aspect, scenarios where different threads have to carry out different computations are analyzed: elaborate 8x8 and 32x32 blocks when dealing with 2 threads; 8x8, 16x16, 32x32 and 64x64 blocks with 4 threads. The adopted scheduling policies for each configuration are the following: in the Parallel accelerators each thread can be assigned to an accelerator, so no scheduling is needed; in the Tagged accelerators each thread can be elaborated concurrently, a higher priority is assigned to earlier requests; in the Single accelerators a first-come-first-served scheduling policy is used.

In a corner negative case, with sequential requests to the accelerator, the behavior is equivalent with what is described in Sect. 4.3.1, so the Tagged and Parallel accelerators give no performance gain with respect to the Single ones. For the corner positive case, requests arriving almost concurrently in increasing- or decreasing- size order are considered.

From Fig. 12, looking at the results of the Single accelerators , it can be noticed a significant difference depending on which thread is elaborated first. Indeed, elaborating the heavier thread first causes the average waiting, response and elaboration time to increase, as lighter threads spend most of the time waiting for the heavier ones to end the computation.

On the other hand, in the Tagged accelerator the arrival order does not significantly affects time performance. Threads flow concurrently through the accelerator, letting the lighter ones end without waiting for the heavier ones. That happens because each actor has a throughput of one token per clock cycle and there is not token accumulation in any buffer. In the end, we obtained an average result which is in between the advantageous decreasing-size cases (up to -92%) and disadvantageous increasing-size cases (up to +13%) of the Single accelerator.

As before, results on the Matrix filter show a greater gain on the Tagged accelerators that fully exploit available computational resources during underutilization periods.

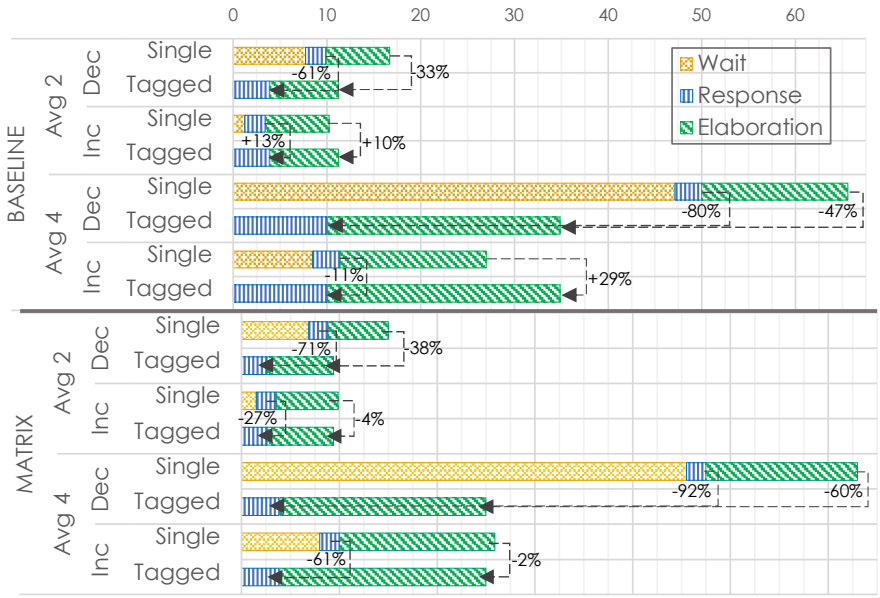


Figure 12 Time performance with different-size elaborations. `Avg 2` and `Avg 4` are the average time performance when 2 or 4 threads are elaborated in incrasing-(`Inc`) or decrasing-(`Dec`) size order. Arrows show the percentage variation in the Tagged accelerators compared with the corresponding Single ones. Timescales are in us.

5 Conclusion

Specialized hardware is crucial in modern electronics to meet performance requirements. In this work we proposed a novel model-based approach for designing multithread hardware accelerators. Following the dataflow paradigm, with additional tagged tokens, we designed a general HDL architecture for actors and two architectures for FIFOs supporting multithreading. Then, using this approach, we designed two complete architectures for a video codec use case with different degrees of parallelism.

Experimental results showed a limited resource overhead, thanks to the possibility of sharing combinational resources. Immediate access to the accelerator by multiple threads and more effective resource exploitation let the proposed accelerator outperform a single-thread accelerator in terms of waiting, response and elaboration times.

Future works will aim to investigate how other aspects of the approach, e.g. the adopted model-of-computation (static or dynamic dataflow), and further details, e.g. the priority management, impact on performance. As this work is mainly meant to demonstrate the feasibility of the proposed approach, future work will also address the points that limit its applicability. A complete host processor-accelerator environment, with proper Operating System support is under development. Also, the design automation through the integration within an HLS flow and a complete design methodology specification will be carried out to make the method effective and available in practice. This will support tackling not only the performance issues for this kind of specialized hardware, but also the design effort.

References

1 Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee. *Software synthesis from dataflow graphs*, volume 360. Springer Science & Business Media, 1996.

2 Nicola Carta, Carlo Sau, Danilo Pani, Francesca Palumbo, and Luigi Raffo. A coarse-grained reconfigurable approach for low-power spike sorting architectures. In *2013 6th International IEEE/EMBS Conference on Neural Engineering (NER)*, pages 439–442. IEEE, 2013.

3 Angelos Charalambidis, Nikolaos Papaspyrou, and Panos Rondogiannis. Tagged dataflow: a formal model for iterative map-reduce. In *EDBT/ICDT Workshops*, pages 29–36, 2014.

4 Yen-Kuang Chen and Sun-Yuan Kung. Trend and challenge on system-on-a-chip designs. *Journal of signal processing systems*, 53(1):217–229, 2008.

5 Jongsok Choi, Stephen Brown, and Jason Anderson. From software threads to parallel hardware in high-level synthesis for fpgas. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 270–277. IEEE, 2013.

6 Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From opencl to high-performance hardware on fpgas. In *22nd international conference on field programmable logic and applications (FPL)*, pages 531–534. IEEE, 2012.

7 William J Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57, 2020.

8 Tiziana Fanni, Lin Li, Timo Viitanen, Carlo Sau, Renjie Xie, Francesca Palumbo, Luigi Raffo, Heikki Huttunen, Jarmo Takala, and Shuvra S Bhattacharyya. Hardware design methodology using lightweight dataflow and its integration with low power techniques. *Journal of Systems Architecture*, 78:15–29, 2017.

9 Rajesh K Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & test of computers*, 10(3):29–41, 1993.

10 John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

11 Jens Huthmann, Julian Oppermann, and Andreas Koch. Automatic high-level synthesis of multi-threaded hardware accelerators. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2014.

12 Jörn W Janneck, Ian D Miller, David B Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet. Synthesizing hardware from dataflow programs. *Journal of Signal Processing Systems*, 63(2):241–249, 2011.

13 Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

14 Rishiyur S Nikhil et al. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on computers*, 39(3):300–318, 1990.

15 Francesca Palumbo, Danilo Pani, Emanuele Manca, Luigi Raffo, Marco Mattavelli, and Ghislain Roquier. RVC: A multi-decoder CAL composer tool. In *Proceedings of the 2010 Conference on Design & Architectures for Signal & Image Processing, DASIP 2010, Edinburgh, Scotland, UK, October 26-28, 2010, Electronic Chips & Systems design Initiative, ECSI*, pages 144–151. IEEE, 2010. doi:10.1109/DASIP.2010.5706258.

16 Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *2009 IEEE 7th Symposium on Application Specific Processors*, pages 35–42. IEEE, 2009.

17 Alfonso Rodrıguez, Juan Valverde, and Eduardo de la Torre. Design of opencl-compatible multithreaded hardware accelerators with dynamic support for embedded fpgas. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7. IEEE, 2015.

18 Leandro Santiago, Leandro AJ Marzulo, Brunno F Goldstein, Tiago AO Alves, and Felipe MG França. Stack-tagged dataflow. In *2014 International Symposium on Computer Architecture and High Performance Computing Workshop*, pages 78–83. IEEE, 2014.

19 Jocelyn Serot, Francois Berry, and Sameer Ahmed. Implementing stream-processing applications on fpgas: A dsl-based approach. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 130–137. IEEE, 2011.

20 Harald Simmler, Lorne Levinson, and Reinhard Männer. Multitasking on fpga coprocessors. In *International Workshop on Field Programmable Logic and Applications*, pages 121–130. Springer, 2000.

21 Ying Wang, Xuegong Zhou, Lingli Wang, Jian Yan, Wayne Luk, Chenglian Peng, and Jiarong Tong. Spread: A streaming-based partially reconfigurable architecture and programming model. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2179–2192, 2013.