Stream Oriented Modular Architecture with Polymorphic Processing Engines

Andriy Gorobets, Frederico Pratas, Nuno Roma and Pedro Tomás
INESC-ID, IST, Universidade de Lisboa
Email: {andriy.gorobets,fcpp}@ist.utl.pt,{Nuno.Roma,Pedro.Tomas}@inesc-id.pt

***Abstract*—Stream computing has shown to be an effective technique to decouple communication from computation in many application domains. It provides an efficient mitigation of bandwidth restrictions, by reducing the amount of memory accesses and by maximizing the available computational resources, potentiating the parallel processing using multiple execution engines. However, it frequently implies significant development costs, since efficient stream-based architectures are usually attained through application-specific full-custom processors, often tightened to the application at hand. To circumvent this limitation, a modular stream computing architecture aiming generic and high performance applications is presented. The proposed architecture, designed for reconfigurable hardware, is composed of modular processing engines that can be customized by the end-user in many ways, such as in terms of their number, type and precision of the functional units. Furthermore, these processing engines are designed as programmable cores, allowing the execution of a wide set of applications using the same configuration. From the conducted evaluation by using a series of benchmark case studies, it was observed that the proposed architecture achieves competitive results when compared with alternative solutions.**

Keywords—Stream Computing, Polymorphic Engine, Reconfigurable Hardware

I. INTRODUCTION

Nowadays integrated circuit technologies provide high computational power on a single chip. This allows to increase the amount of hardware to support intensive data parallel processing. Despite such computational potential, it is often difficult to keep all the available computational resources busy, either due to data dependencies, or simply because the application is not able to exploit all the available computational units, resulting in practical inefficiencies. Moreover, the memory and interconnections bandwidth is frequently a limiting factor. Thus, it is important to choose the appropriate architecture, as well as the programming approach, that maximizes the usage of the on-chip computational power.

Traditional general purpose processors adopt a computing-in-time paradigm, which focuses on low level organization to efficiently solve instruction dependencies and improve serial applications. However, this requires a huge amount of control logic. General purpose processors have also evolved in the direction of increasing its processing capacity, by increasing the amount of logic dedicated to arithmetic computations. Current general purpose processors already implement data parallel processing techniques such as vector processing and functional parallelism in multi-core processors. However, such conventional processing models are unable to maximally exploit the parallelism made available by several data intensive and highly demanding applications. While data prefetching mechanisms can help minimizing the limitations for memory-bounded applications, it is difficult to efficiently use all the available computing resources. In such cases, the adoption of dataflow processing is recognized as a viable approach.

The dataflow execution model applies a computing in space paradigm, where data is streamed through processing elements. As a consequence, a dataflow application is organized as a sequence of arithmetic kernels, each representing a data operation. An important characteristic of this model is the data locality of all intermediate results. Usually, only the initial inputs or final results must be provided from/to an external data storage. Overall, this model explores the locality and parallelism that is present in applications. Data locality greatly reduces the external memory bandwidth requirements, thus increasing functional units occupation and reducing energy consumption. Since most computationally demanding applications present high levels of data and functional parallelism, the streaming model shows to be an attractive solution.

While many attempts to develop highly efficient stream-based architectures have been made (e.g., [4, 9]), they are typically constrained to either a single or a very reduced set of applications. As a result, these solutions are not very flexible (with low or no programmability), and require long development time. Furthermore, many applications not only require processing efficiency but also programmability [2]. Attempts to build programmable stream-based architectures have also been made, e.g., [5, 3, 6]. However, many of these architectures are similar to that of modern Graphics Processing Units (GPUs), that rely on the SIMD approach, allowing the execution of the same instruction simultaneously on a vast number of arithmetic functional units. This raises important constraints when the kernel applied to the stream of data limits processing coalescence. Moreover those solutions rely on an external processor which organizes data into ordered streams and sends them to the stream processor. This poses important limitations on the flexibility for some applications that require complex memory data access patterns and/or reuse [8]. Furthermore, typical stream-based programmable architectures do not allow using custom-based arithmetic, resulting in decreased operating frequency, nor using special functional units leading certain operations to be performed in software. Finally, in many cases, the architectures are focused in high performance computing system with unlimited power and area constraints, which poses important constraints for embedded systems. While attempts to build configurable or even reconfigurable architectures have been made (e.g., [10, 11]), they have limited programmability, making application developing a challenge. During the past few years new embedded systems have appeared, where reconfigurable logic, used to accelerate the most demanding parts of the applications, coexists on a single chip with a general purpose CPU (e.g., Xilinx Zynq-7000 EPP, which includes a dual-core ARM Cortex-A9 CPU). This

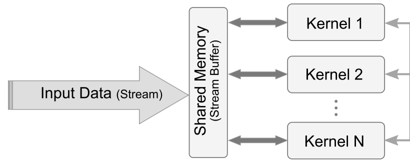

Fig. 1: Overview of main architectural blocks.

reduces the communication time between the CPU and the reconfigurable accelerators, as well as the power consumption.

In this work, a modular and fully programmable stream-based architecture for high efficiency computing purposes is proposed. A possible embodiment of such an architecture is also presented, along with a set of tests in order to assess its efficiency. The proposed architecture supports some level of customization, depending on the target application, in order to minimize hardware resources. On the other hand, for a given fixed configuration, the offered programmability allows the execution of a vast set of different applications depending on the number and type of involved functional units, without requiring hardware reconfiguration.

II. Proposed Modular Streaming Architecture

The proposed architecture takes the basic principles proposed in [10]. It consists of a modular programmable stream-oriented processor that can work either autonomously or with a host processor, playing the role of an accelerator. Figure 1 depicts a generic overview of the main architectural blocks: i) a custom number of independent Kernel modules that can be either fully custom modules or programmable cores; and ii) a Local Shared Memory, which serves as buffer for input and output streams and also for storing intermediate results.

A. Kernel

Kernels operate on streams i.e., they receive the data from an incoming input stream, execute a number of operations on it and deliver the results. Results are either stored in a memory buffer for further processing by the next Kernel, or sent back as the output of the processor.

Herein, we focus on a programmable microarchitecture with support for Kernels that can be customized in terms of quantity and functionality. This modular infrastructure supports the same architecture (in terms of working principles, structure, and programming model) independently of the application needs, being only constrained by resource availability. Each programmable Kernel is a multi-thread processor, with multiple instruction dispatchers sharing the same instruction set and the same functional units. This reduces the hardware resources for control and maximizes the usage of computational power.

Figure 2 shows the block diagram of the proposed programmable Kernel that is proposed in this work. It is composed of N *Dispatchers*, responsible for executing the control path of the application, a number of shared *Functional Units* that carry out the operations, a *Data Stream Manager* responsible for managing data streams between Shared Memory and Dispatchers, and various *Interconnection Networks* to efficiently transfer data between the different units.

B. Dispatcher

Each Dispatcher is responsible for handling the correct operation of the control-flow model. It consists of a fully programmable general purpose processor, with the reduced Instruction Set of the Microblaze softcore. The Dispatcher's microarchitecture consists of four pipeline stages: Instruction Fetch, Instruction Decode and Operand Fetch (ID&OF), Execute and Write Back, here part of the execution may be carried by the external pipelined Functional Units (see Fig. 3).

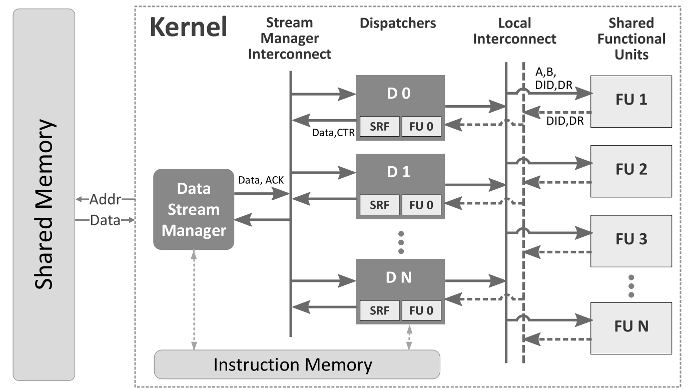

Fig. 2: Block diagram of the devised Kernel definition.

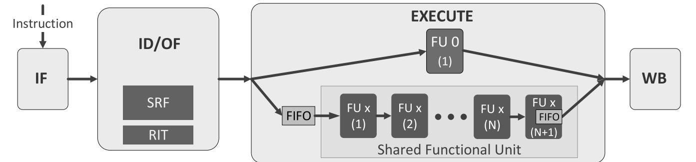

Fig. 3: Superpipelined Dispatcher architecture.

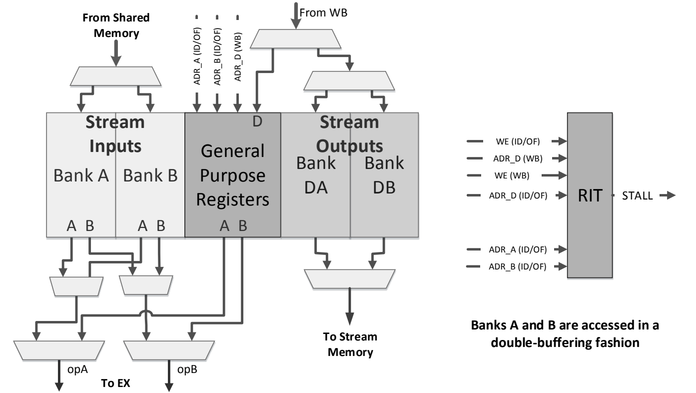

Fig. 4: Stream Register File and the Register Invalidation Table are accessed on the ID&OF stage and Write Back stage.

On the first stage, the instruction to execute is fetched from a shared Instruction Memory (outside the Dispatcher). On the second stage, the instruction is decoded and the operands are fetched from the Stream Register File (SRF), which serves for data input/output and to store temporary values while operating over streaming data (see Fig.4). From the Dispatcher's point of view, these accesses are done in a uniform way. To accomplish that, the decoding of standard instructions was modified to support selection of source and destination streaming operands (general purpose register *vs* stream register).

The ID&OF stage also includes a special block, the Register Invalidation Table (RIT), responsible for solving data hazards (see Fig.4). When an operation is issued for execution, the position on the RIT corresponding to the destination register is set, invalidating it. Then, only when the result is written to the SRF during the write back stage, can this value be validated. As such, when the operands are fetched, in the

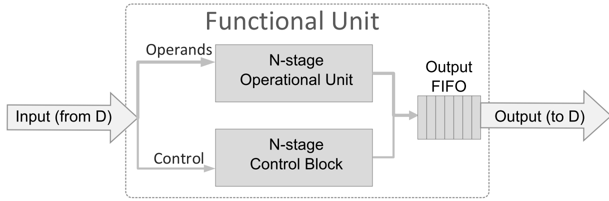

Fig. 5: Block diagram of the devised Polymorphic FU structure.

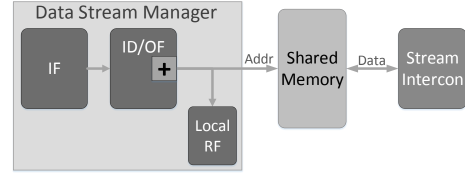

Fig. 6: Data Stream Manager block diagram.

ID&OF stage, the RIT has to be consulted to check if all the operands are valid; if they are not, the Dispatcher stalls.

Each instruction can either correspond to a data processing operation or a control operation. In the first case, the operation is executed on the respective *Functional Unit* (FU), usually external to the Dispatcher. All Dispatchers also contain an internal FU (FU 0), on the Execute stage, to execute simple operations, such as logic and fixed point arithmetic (addition/subtraction) for control flow operations and reductions. Moreover, it requires much less hardware resources than shared FUs, such as those operating on floating point numbers. As such, it is advantageous to keep it inside the Dispatcher. When an operation is to be executed on an external FU, it is placed in a FIFO buffer after fetching its operands. The FIFO has two main advantages: i) when the target FU is busy the Dispatcher does not have to stall, as long as the FIFO is not full; and ii) if a Dispatcher stalls and there is some valid data in the FIFO, it is still issued to the respective FU.

Finally, the Write Back stage manages the results of the different FUs to the SRF. Results can come from one of three sources: Jump Control Unit for branch instructions, FU0 and any of the external FUs. Data elements coming from the external FUs have priority over internal data giving priority to older instructions and avoiding stalling the shared FUs. When a conflict occurs, the Dispatcher stalls for one cycle, allowing external data to be written to the SRF.

C. Functional Unit

Each Kernel may include several independent FUs, which are defined as polymorphic blocks that may contain already existing, or custom operational units. As shown in Figure 5, each FU contains an *Operational Unit* (OU), with several pipeline stages, a control block, and an output FIFO.

Input data coming from the Dispatchers is divided between operands, which are fed to the OU, and control signals. The control signals are used by the unit itself, and to generate stream control data required by the WB engine. When the operation is finished, both the stream control data and the results are concatenated and written into the output FIFO. As in the Dispatcher, this FIFO plays an important buffering role, allowing the FU to continue operating even when the WB stage is not able to consume the data, as long as the FIFO is not full. This allows minimizing the number of stalls of the FU.

Kernels also contain a local memory shared by the Dispatchers, used for intermediate results and auxiliary constants. This memory adopts the same interface of a generic FU.

D. Local Interconnect

Two unidirectional and independent networks were specifically developed for the interconnection between the Dispatchers and the several FUs, namely for ISSUE and WB. Both networks support a generic number of masters and slaves, and both operate in a crossbar switch mode allowing data from different masters to be simultaneously transferred, as long as the destination slave is not the same. In both cases, a low latency high bandwidth crossbar switch is used to minimize the impact of this network. The operations issued from Dispatchers to the FU's mainly consists of: two operands, the sender's Dispatcher ID and the destination register address. Conversely, during WB operation data consists of the result, the destination address of the SRF the Dispatcher ID and FU ID.

III. Data Streaming

To enable stream computing, a data streaming framework was also integrated into the system, which is based on the structure proposed in [10], [8]. This section describes the main blocks responsible for the stream management inside the Kernel, comprehending the Data Stream Manager (DSM), SRF and Interconnect (see Fig.2).

A. Data Stream Manager

The DSM is the unit responsible for handling all data transfers between the local buffer memory and the Dispatchers. It mainly consists of a programmable core with an instruction set very similar to one used to program the Dispatchers. There are two main types of instructions: i) internal, to control the execution flow and calculate the data addresses to fetch the data; and ii) external, to load data to the Dispatcher's SRF. The DSM adopts a pipelined architecture with two stages for internal instructions, and three stages for external ones. It contains a set of general purpose registers, to store results of internal instructions. Figure 6 illustrates the devised architecture of the DSM. The first stage consists of the instruction fetch, while the second stage decodes the instruction, fetches the operands from the local Register File (RF), and executes the operation. After that, for internal instructions the result is written to the local RF and a new instruction is fetched. For external instructions, a third stage is still used for memory accesses. In case of a stream loading operation, the data is then sent to the Dispatcher through the Interconnect.

The implemented DSM instruction set features additional instructions to leverage the implementation and manipulation of complex data patterns, thus providing an efficient means to accelerate and mitigate the memory accesses. In particular, the new *Load Pattern* instruction gives support for the execution of regular memory accesses, with the ability to consider a user defined `stride` and `number of transfers`. As a consequence, this instruction allows the replacement of a significant amount of control related instructions and signals, being able to transfer one operand from up to two independent data vectors on each clock cycle, without any control overhead. A corresponding *Store Pattern* instruction was also introduced in the instruction set, offering similar capabilities to transfer the outgoing processed stream.

The execution flow of the DSM depends on the status of the SRF input and output banks. In some cases, each

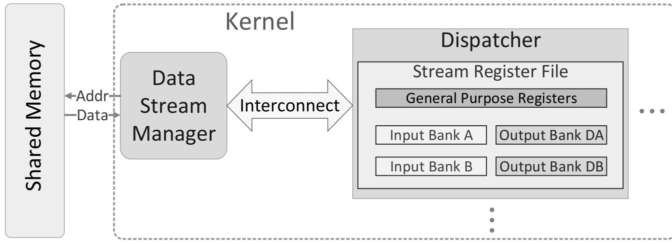

Fig. 7: Structure of the Stream Register File.

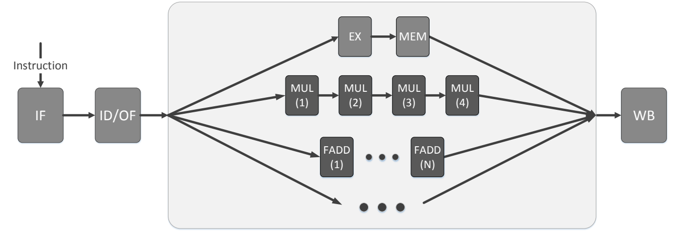

Fig. 8: General overview of the baseline superpipelined architecture.

Kernel may feature two DSMs, to keep the loading and storing transactions independent thus increasing efficiency. This also improves scalability, in terms of the number of Dispatchers.

B. Stream Register File

The SRF is divided in five register banks (see Figures III-B and 7). One bank of general purpose registers is used to store auxiliary variables and intermediate results. There are four FIFO fashioned banks for buffering inputs (A, B) and outputs (DA, DB). Operations on data from a given input bank are associated to its counterpart output bank in a paired way (A/DA, B/DB). Thus, banks A and B support the implementation of a double buffering transfer strategy. This mechanism allows the DSM and the Dispatchers to operate independently: while the DSM loads/writes data to/from one bank pair, the Dispatcher processes the data from another. As a result, the Dispatcher is alleviated from executing load or store operations when operating on streams. Instead, it simply fetches the operands from its register file. Furthermore, the status signals from the input and output banks are used to control the DSM flow and the Interconnect arbitration.

C. Stream Manager Interconnect

To connect the Data Stream Manager to the Dispatchers, a high bandwidth, bidirectional network infrastructure supporting a generic number of masters (M) and one or two slaves was developed. When two Stream Managers co-exist inside the same Kernel, the Interconnect operates in a crossbar switch mode, where the Dispatchers play the role of masters. As such, they are the peers that initiate each transaction. Accordingly, each Dispatcher issues a read/write operation when new operands have to be loaded/outputted to/from the SRF. Different applications may require different types of arbitration. Thus, data may be transferred to the Dispatchers either individually or as a broadcast.

IV. Experimental Results

To evaluate the performance of the proposed architecture, the whole computational structure was prototyped and evaluated in a Virtex-7 FPGA device (XC7VX485T-2). For such purpose, a set of benchmarking case studies were considered in this evaluation. This section presents a comprehensive description of those case-studies and a discussion of the results.

1) Reference Processor: To assess the attained performance, the proposed solution was compared with a conventional General Purpose Processor (GPP). A modified version of the MB-Lite processor [7] was adopted as baseline, consisting on a 5-stage pipelined embedded architecture implementing the Microblaze ISA. As shown in Fig. 8, the baseline architecture includes a superpipelined execution stage, with a set of internal FUs, in contrast with the proposed architecture, where most FUs are external to the Dispatcher modules and shared among them, which imposes two implicit overhead cycles: one for issuing the operation to the FU and another for transferring the results back to the Dispatcher. To make the comparison fair, the number of pipeline stages for the baseline internal FUs is set to be equal to the one used by the proposed processor. Moreover, while the baseline processor extensively adopts forwarding techniques to reduce the number of execution stalls during the execution, the proposed Dispatcher simply stalls until the dependency is solved. Nevertheless, the proposed architecture still offers significant advantages in terms of the computational performance when compared with conventional GPPs.

2) Case Study: Single Precision Inner Product: The vector dot product is a frequent operation in many domains. It consists of multiplications and accumulations over two data arrays. A data parallel approach is used herein, where a set of D Dispatchers in a Kernel processes $\frac{1}{D}$ part of the data. Hence, the scalability of the system mainly depends on the capability of the DSM to provide the input data. The load pattern instruction that is executed on the DSM is in charge of loading the data without interruptions, such that it can provide Dispatchers with interleaved values from each vector. This prefetching of the input data greatly simplifies the programming and execution of the Kernel, providing a significant increase of its overall efficiency, since the Dispatchers input banks receive the elements from both vectors in order.

The dot product example reinforces the usefulness and advantages of the application of double buffering techniques on the SRF. The Dispatcher repeatedly executes only three instructions (i.e., FMUL, FADD, BRANCH), as long as there is any data available to be processed. The loading of the operands from the input stream buffer is completely handled in background. Just like in the baseline architecture, the adopted pipelined multiplier structure forces the accumulation instruction to wait until it finishes. Nevertheless, the accumulation instruction does not introduce any stall in the Dispatcher since it does not create any dependency.

On the other hand, the baseline architecture requires two extra instructions to load the operands in each iteration, and a third one for loop control. Hence, considering that the input data is fetched from a cache with L elements per line and with a miss penalty of N cycles (corresponding to the number of cycles to bring data from the outer memory level to the L1), and by taking into account that the two input vectors are separately stored in the input data buffer, each line provides L operands from each vector. Hence, since it is necessary to read data from both vectors for each iteration, two cache misses will occur every L iterations.

Figure 9 depicts the obtained speed-up for different number of Dispatchers and considering that the baseline and the proposed processors operate at the same frequency. Furthermore, different organizations of the reference processor cache were

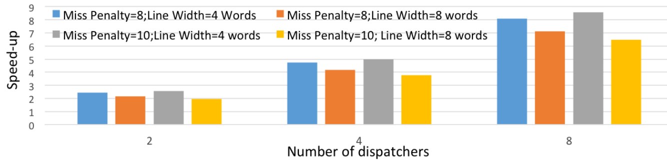

Fig. 9: Speed-up for different number of Dispatchers and different cache characteristics.

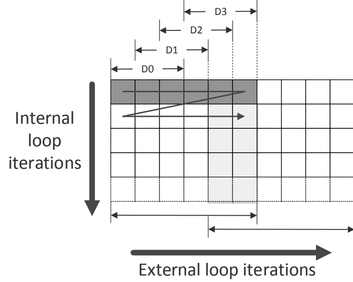

Fig. 10: Data management for the 2D convolution operation

assumed. In particular, it was considered an L1 data cache in the reference processor with a miss penalty (to access L2) of 8 and 10 clock cycles, with a block size of 4 or 8 32-bits words. Naturally, these rather optimistic cache organizations greatly benefit the baseline (in detriment of the proposed architecture). Nevertheless, the proposed architecture was still able to clearly gain, in terms of the offered performance.

The distribution of the input data by all Dispatchers implies a final reduction operation that must be carried by a single Dispatcher. The reduction Dispatcher reads the partial results, produced by other Dispatchers, from the intra-kernel memory. As a result, the overhead of the final accumulation grows with the number of Dispatchers, as shown in Fig. 9, where speed-up decays with the number of Dispatchers.

This case study used three shared FUs: a floating point multiplier with 8 clock cycles of latency; a floating point adder with 11 cycles of latency; and a small intra-kernel memory. The resource usage for different Kernel configurations are presented in Table I. The resources allocated to the Kernel grow proportionally with the number of Dispatchers.

3) Case Study: Image Convolution: The 2D convolution operation is extensively used by many image processing applications. Usually, a small filter matrix sweeps through the input image matrix to execute the convolution operation, which consists in multiplying all positions of the filter by the correspondent positions of the input matrix, with a final accumulation of the result. For each location of the filter relatively to the input matrix, one output value is produced. Two main issues have to be taken into account: i) the best way to exploit the parallelism, and ii) the reuse of data, in order to minimize the number of accesses to the output matrix and, in turn the required bandwidth.

Figure 10 depicts the parallelization approach and respective data patterns taken in order to reuse the input data both horizontally and vertically. The problem is distributed such that each Dispatcher processes one column of the output matrix. Each iteration the DSM sends part of the input line to be processed. To reduce contention on the DSM Interconnect and increase data reutilization, the DSM broadcasts the operands to all Dispatchers. This increases the scalability of the system. From the DSM point of view data is streamed in columns of

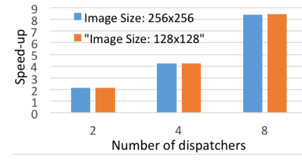

Fig. 11: Filter of 3x3

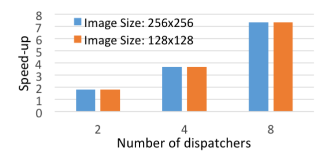

Fig. 12: Filter of 5x5

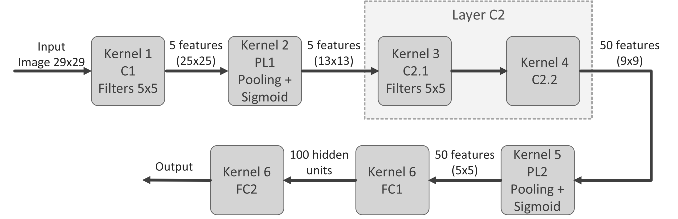

Fig. 13: Conceptual data flow for the CNN case study.

(D+K-1) elements, (with filter size equal to KxK), which are controlled by two for loops. On each iteration, each Dispatcher processes its K elements, ignoring the rest.

Broadcasting of the input data implies that all the Dispatchers are ready to receive new data. This imposes an implicit synchronization between Dispatchers. Nevertheless, as stated before, the Dispatchers execution is carried in parallel with the data transfers, due to the double-buffering mechanism in place. Thus, the adopted structure of the SRF allows to implicitly synchronize the Dispatchers, by using status signals from the input and output banks.

The results obtained for a variable number of Dispatchers are shown in Fig. 11. The presented results do not take into account any cache misses that may occur on the baseline processor, the optimistic baseline results assume that the image either fits entirely in the L1 cache or it is loaded by using a perfect data prefetching mechanism. Naturally, the speed-up of the proposed architecture would be substantially higher if neither of the previous conditions are met and the data would first have to be loaded from L2 or global memory.

The high performance that is achieved for this example is obtained due to two main factors. Firstly, the existence of multiple Dispatchers in the Kernel allows reusing the input data in both the horizontal and vertical directions. Also, the broadcasting of the data is particularly beneficial to the proposed architecture, since it avoids the occurrence of contention in the Interconnect and starving situations.

It is worth noting that larger filters may require partial caching of the filter data. This can be efficiently accomplished either by adjusting the size of the SRF; or by using an intra-kernel local memory, interfaced as a general purpose FU and shared among the Dispatchers. An example of the results obtained with a 5x5 filter is shown in Fig. 12.

For this case study two shared FUs were used: an integer multiplier with 6 cycles of latency, and a small intra-kernel memory. The accumulations were performed in the FU0 of each Dispatcher. Table II presents the amount of resources used in this case study. Once again, the used resources are proportional to the number of Dispatchers. Furthermore, since the two case studies mainly differ in the type of multiplier and adder structures, the differences in resources between them are minimal (excluding the number of DSPs).

4) Case Study: Convolutional Neural Network: The last case study presented executes a Convolutional Neural Network

TABLE I: Resources used for different number of Dispatchers for the inner product case study.

	Available	2 D's	4 D's	8 D's
#LUTs	303600	6558	11144	20523
#Registers	607200	7624	12724	22867
#Block RAM	1030	18	19	21
#DSP	2800	8	8	8
Max. frequency [MHz]	-	176	152	134

TABLE II: Resources used for different number of Dispatchers for the image convolution case study.

	Available	2 D's	4 D's	8 D's
#LUTs	303600	5724	10226	19407
#Registers	607200	6445	11526	21663
#Block RAM	1030	18	19	21
#DSP	2800	4	4	4
Max. frequency [MHz]	-	176	152	134

(CNN) topology for handwritten digit recognition similar to one presented in [1]. This example is interesting because it allows to build a multi-Kernel system deploying the previously introduced Kernels as basic blocks. The conceptual data flow is depicted in Fig. 13 where the layers of the CNN are assigned to respective Kernels as follows: i) *C1 (K1)*: first convolution layer, executes the convolution on a single input image (29x29) using 5 convolutional filters of 5x5, it results in 5 feature maps of 25x25; ii) *PL1 (K2)*: max pooling layer followed by sigmoid function, executes max pooling on each input map, reducing it to half of the original size, and applies a sigmoid function on each element, it outputs 5 feature maps of 13x13; iii) *C2 (K3&K4)*: second convolution layer, which is applied to 5 input features using 50 filters (5x5x5) – *K3* executes multiple regular convolutions on each input feature, while *K4* accumulates groups of 5 maps into a single feature, by adding the respective elements – this layer produces 50 features of 9x9; iv) *PL2 (K5)*: a second max pooling layer with sigmoid function, which returns 50 maps of 5x5; v) *FC1 (K6)* a fully connected layer that computes 100 dot products across all output elements of *PL2* using the respective weights from data memory, it produces 100 output elements; and vi)FC2 (K6): a second fully connected layer that performs 10 dot products across 100 input elements and its respective weights, producing 10 outputs, one per decimal digit.

Table III shows the number of Dispatchers allocated to each Kernel, while Table IV shows the amount of hardware resources used by the system. The size of each Kernel takes into account the demand of the respective layer in order to increase the overall performance. *K3* is the heaviest Kernel with 8 Dispatchers. It executes most convolutions. Also *K6* has 8 Dispatchers since it has to execute 100 dot products with vectors of 1250 elements. The remaining Kernels are small and do not contribute significantly to the total execution time.

In order to compare the obtained results with the baseline, we considered a multiprocessor system where each processor replaces one Kernel. This is the same approach followed in the previous case studies. We also verified that generally, a Kernel with N Dispatchers executes a program approximately N times faster than the baseline (near to linear speed-up).

Using the configuration presented in Table III, the achieved speedup for the proposed system is approximately 7x. This can be explained by the load distribution among the Kernels. Since the speed-up tends to be dominated by the largest Kernels which compute the slowest layers.

TABLE III: Dispatchers allocated to each Kernel

K	*K1*	*K2*	*K3*	*K4*	*K5*	*K6*
#Dispatchers	4	2	8	2	2	8

TABLE IV: Resources used for the CNN case study.

	Available	Used
#LUTs	303600	66212
#Registers	607200	74187
#Block Ram	1030	115
#DSP	2800	24
Max. frequency [MHz]	-	134

V. CONCLUSIONS

With the increasing interest of the market in the low power devices, and the increasing interest of the community in data intensive applications, solutions such as the one proposed here will have a strong impact in the near future. In this paper we propose a modular and fully programmable stream-based architecture that contrasts with other dedicated structures that can hardly be modified to different application domains. We have shown with several sound case studies how the modular nature of this architecture promotes an easy adaptation and customization of its processing structure to the targeted application domain, as well as an efficient utilization of the required hardware resources. It was also shown that the performance of the proposed data-stream architecture scales almost linearly when multiple dispatchers are implemented in the kernel. For multi-kernel applications the performance depends on load distribution, allowing to achieve even higher efficiency.

ACKNOWLEDGMENTS

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects Threads (ref. PTDC/EEA-ELC/117329/2010), P2HCS (ref. PTDC/EEI-ELC/3152/2012) and project PEst-OE/EEI/LA0021/2013.

REFERENCES

[1] Kumar Chellapilla, Sidd Puri, Patrice Simard, et al. High performance convolutional neural networks for document processing. In *IWFHR*, 2006.

[2] William J Dally, Ujval J Kapasi, Brucek Khailany, et al. Stream processors: Progammability and efficiency. *Queue*, 2(1):52, 2004.

[3] William J Dally, Francois Labonte, Abhishek Das, et al. Merrimac: Supercomputing with streams. In *ICS*, page 35. ACM, 2003.

[4] Amir Hormati, Manjunath Kudlur, Scott Mahlke, et al. Optimus: efficient realization of streaming applications on FPGAs. In *CASES*, pages 41–50. ACM, 2008.

[5] Ujval J Kapasi, William J Dally, Scott Rixner, et al. The Imagine stream processor. In *ICCD*, pages 282–288. IEEE, 2002.

[6] Brucek K Khailany, Ted Williams, Jim Lin, et al. A programmable 512 GOPS stream processor for signal, image, and video processing. *JSSC*, 43(1):202–213, 2008.

[7] T. Kranenburg and R. van Leuken. MB-LITE: A robust, light-weight soft-core implementation of the MicroBlaze architecture. In *DATE*, pages 997–1000, March 2010.

[8] Sérgio Paiágua, Frederico Pratas, Pedro Tomás, et al. Hotstream: Efficient data streaming of complex patterns to multiple accelerating kernels. In *SBAC-PAD*, pages 17–24. IEEE, 2013.

[9] Oliver Pell, Oskar Mencer, Kuen H. Tsoi, et al. Maximum performance computing with dataflow engines. In *HPRC*, pages 747–774. Springer, 2013.

[10] Frederico Pratas, Pedro Tomás, Pedro Trancoso, et al. Energy efficient stream-based configurable architecture for embedded platforms. In *SAMOS*, pages 193–200. IEEE, 2012.

[11] Ying Wang, Xuegong Zhou, Lingli Wang, et al. SPREAD: A streaming-based partially reconfigurable architecture and programming model. *IEEE Trans. VLSI Syst.*, 21(12):2179–2192, 2013.