Efficient Data-Stream Management for Shared-Memory Many-Core Systems

Nuno Neves, Pedro Tomás and Nuno Roma

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Rua Alves Redol, 9, 1000-029 Lisboa - Portugal

Email: {nuno.neves,pedro.tomas,nuno.roma}@inesc-id.pt

***Abstract*—The design of most high-performance and heterogeneous processing platforms is usually solely focused on the computational part, while neglecting the power/performance impact of the data-management infrastructures. Moreover, such systems often struggle to achieve their potential performance when applications require fetching data with complex memory access patterns. To overcome these issues, a energy-efficient stream-based data-management infrastructure is herein proposed, relying on a novel tree-based descriptor specification. Such descriptors are decoded by a Descriptor Tree Controller (DTC) architecture, which allows simple and efficient management of arbitrarily complex memory access patterns. Moreover, a Stream Management Engine (SME) ensures energy-efficient data-reutilization through the application of automatic stream rerouting, splitting and merging techniques. The obtained results show that the proposed DTC architecture is capable of a highly efficient complex data-pattern generation, while significantly reducing the size occupied by the pattern description, when compared with state-of-the-art approaches. By also enabling the deployment of data-reuse techniques, a reduction of up to 85× in the number of accesses to the main shared memory is achieved, resulting in a decrease as high as 475× in the observed energy consumption.**

I. Introduction

The ever increasing demand for computational processing power at a significantly low energy consumption has pushed the research for alternative heterogeneous and often specialized processing structures. However, such architectures are usually designed by focusing on the individual computational blocks, often neglecting the power/performance impact of the inherent data transfers and general data indexing. In fact, most solutions mainly focus on increasing the throughput of the data-processing system, while relying on conventional cache structures to avoid the usually high memory access latencies. However, such structures do not always work well when the applications require streaming data with arbitrarily complex memory access patterns.

Several approaches can be adopted in order to minimize the power/performance impact of the data-management subsystem. In particular, besides aggressive data prefetcher schemes associated with high-energy consuming memory/cache hierarchies with complex shared bus protocols, more efficient and sophisticated stream-based communication schemes can be deployed. That way, instead of having each system's Processing Element (PE) to concurrently perform main memory requests, resulting in a tremendous pressure in the memory and bus subsystems and, in turn, in increased energy consumptions, it is possible to exploit the data-access pattern of each individual processing block, in order to smoothly and transparently buffer and stream the data to the corresponding PEs. Hence, most of the contention usually observed in shared communication structures can be eliminated. Moreover, by relying on streaming approaches, and by decoupling the data communication and processing structures, it is possible to hide data transfers behind computation with the use of intermediate (small) buffering memories, which can be preloaded by the data streaming structures. This presents a rather interesting outcome since, even without directly relying on aggressive prefetching schemes and structures, it is still possible to exploit the same advantages, such as handling of complex data-patterns [1] or reducing the energy consumption of the data-management infrastructure [2].

Regarding stream-based approaches, several techniques have been proposed to improve the throughput of the data-streaming and management infrastructure. Park [3] tackled the data fetching from an external memory to a Field-Programmable Gate Array (FPGA) in the context of stream-computing. Acknowledging that proper data re-utilization mechanisms are fundamental in FPGA-based systems, the work was further extended to support the scheduling of simple data operations (e.g. stripping, splitting or merging) in the context of multi-processor systems [4]. Meanwhile, Hussain [5] proposed a Programmable Pattern-based Memory Controller (PPMC) that supports regular 1D, 2D and 3D data-fetching mechanisms, such as scatter-gather and strided accesses with programmable tiling. An efficient scheduler and an intelligent memory manager were subsequently integrated in this data-fetching controller, to facilitate elaborated data movement and other computational tasks.

However, while the PPMC represents a step forward towards the streaming of complex patterns, it was designed for moving large and regular data-chunks and falls short with irregular or arbitrarily complex indexing. To partially tackle such a more ambitious challenge, the Hotstream framework [6] relies on PE-coupled dedicated pattern-programmable Data-Fetch Controllers (DFCs), capable of also deploying data-reuse techniques. Nevertheless, although such a system can greatly increase the system's throughput, it still partially relies on shared communication structures, thus still incurring in increased pressure in the memory subsystem (although mitigated by application of the data-reuse techniques). Moreover, although the considered programmable approach eases

the description of complex, but still regular, data-patterns, it also struggles with irregular and arbitrarily complex patterns. Hence, a special attention must be given to the generation of memory-access pattern structures, in order to tackle these persisting issues.

Most of the above mentioned techniques, such as data-reorganization and data-reuse schemes, combined with efficient stream-based infrastructures, are still insufficient to provide the aimed system throughput and energy savings. When irregular indexing or complex data patterns are considered, it is still clear that the adoption of more sophisticated techniques still has to be highly exploited for further optimizations both at the level of performance/throughput and at the level of energy consumption of the data-management subsystem.

Accordingly, a novel energy-efficient stream-based data-management infrastructure is herein proposed, based on a novel tree-based data-pattern descriptor specification. This specification relies on a specially devised descriptor to define a tree-like hierarchical organization, which significantly eases the description of arbitrarily complex memory access patterns. To decode such descriptors, a Descriptor Tree Controller (DTC) architecture is also proposed, capable of efficiently handling the memory address generation and data indexing. When combined with dedicated wrappers, the proposed DTC architecture provides an efficient memory access and stream generation, while also being used to promote data-reuse techniques through on-the-fly stream manipulation operations, such as stream merging and splitting, which is achieved relying on stream-oriented First-In First-Out (FIFO) memories. Such facilities are comprised in a Stream Management Engine (SME), which also includes a dedicated communication infrastructure with broadcast capabilities. The obtained experimental results demonstrate the address generation efficiency of the proposed DTC architecture when compared to state-of-the-art architectures. Furthermore, the combined data reuse and manipulation facilities of the proposed SME allow reducing the number of memory accesses by as much as 85×, which in turn results in a reduction of 475× of the memory-access-related energy consumption.

II. Streaming of Complex Data Patterns

Independently of their application domain, many algorithms are characterized by memory access patterns represented by a *n*-dimensional affine function [7]. For a 2D pattern, the affine function is typically written as $y(i,j) = \mathit{offset} + i + \mathit{stride} \times j$, where the current memory address (y) is calculated based on an initial OFFSET, two increment variables (i and j) and a STRIDE multiplication factor. Since such representation allows indexing most regular access patterns, the 2D specification is commonly used by Direct Memory Access (DMA) controllers or other similar data-fetch controllers, such as the PPMC [5]. However, by only relying on such simplistic representation, data-fetch controllers often need to combine several descriptors to represent complex patterns [8].

Although this solution allows the representation of more complex (but still regular) memory access patterns (e.g. 3D, blocked, tiled patterns), it still struggles when trying to describe higher levels of pattern complexity (e.g. diagonal, zig-zag, diamond patterns), requiring very large descriptor lists and a very complex control, which not only increases the size of the descriptors, but also the hardware structure that supports it [6]. Moreover, there is usually a certain amount of overhead associated to the switching between descriptors, which further degrades the memory address generation rate and efficiency. To circumvent these limitations, the new descriptor definition that is herein proposed not only allows increasing the dimensionality of the described pattern from the usual 2D to a 3D representation, but also introduces a novel tree-like hierarchical organization of such descriptors, which significantly eases the description of arbitrarily complex data-patterns and reduces the number of required descriptors.

A. 3D Data-Pattern Descriptor Tree

As the complexity and irregularity of a given pattern increases, the number of 2D descriptors that are required to represent it also increases in the same proportion, leading to long and hardly efficient descriptor chains. To circumvent such drawback, the new tree-like descriptor structure that is now proposed is based on the adoption of a greater dimensionality (from 2D to 3D) of memory access pattern description, allowing a consequent simplification and reduction of the number of required descriptors for complex patterns. In fact, the third dimension that is now introduced provides the addition to the above described pattern representation function of a new SPAN multiplication factor weighted by a new increment variable (k).

Hence, each 3D memory access pattern is formally described by the affine function:

$$\begin{aligned} y(i,j,k) = \mathit{offset} + i + \mathit{stride} \times j + \mathit{span} \times k, \\ \text{with } i \in \{0,...,hsize-1\} \\ j \in \{0,...,vsize-1\} \\ k \in \{0,...,dsize-1\} \end{aligned} \quad (1)$$

As depicted in the example of Fig. 1, this memory access pattern is represented by the tuple {OFFSET, HSIZE, STRIDE, VSIZE, SPAN, DSIZE}, specifying the starting address of the first memory block (OFFSET), the size of each contiguous block (HSIZE), the starting position of the next contiguous block with relation to the previous (STRIDE), the number of repetitions of the two previous parameters (VSIZE), the starting of the next 2D pattern in relation to the previous (SPAN), and the number of repetitions of the four previous parameters (DSIZE).

Despite the significant advantages that are offered by this newly proposed descriptor (capable of individually representing more complex patterns), the adoption of conventional

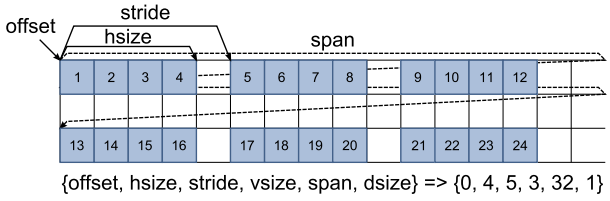


Fig. 1. 3D memory access pattern description example, with the order in which the data positions are accessed. The *vsize* value indicates the number of repetitions of the contiguous block of size *hsize*, separated by *stride* positions. The *dsize* value indicates the number of repetitions of the pattern generated by the *hsize*, *stride* and *vsize* values, separated by *span* positions.

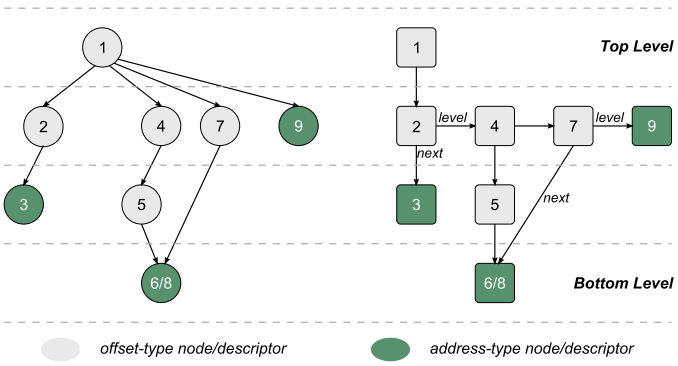


Fig. 2. Proposed descriptor tree-based hierarchical organization, shown both in tree representation (left) and with the corresponding referencing between the descriptors (right). The number associated to each node represents the order in which the descriptors are iterated. Note that the descriptor at the bottom level is referenced twice, which means it is solved with two different relative offset patterns.

cascaded lists of descriptors to represent highly complex data patterns still poses difficult challenges in terms of the attained efficiency. To also circumvent such adversity, the proposed descriptor organization relies on a tree-based hierarchical scheme (depicted in Fig. 2), in which multiple parent-child relations can be established between descriptors, representing dependencies between different descriptor levels. For such purpose, the considered tree hierarchy distinguishes the descriptors between *offset*-type descriptors (used to calculate relative offsets) and *address*-type descriptors, which generate memory addresses based on such relative offsets. Moreover, *address*-type descriptors are characterized by not having any depending-child descriptors in the tree hierarchy.

In order to deploy the proposed tree hierarchy, a double-referencing type representation (usually used to represent trees in programming languages) was used. Hence, each descriptor has a reference to a child descriptor (NEXT) and a reference to a descriptor that shares the same parent descriptor (LEVEL). This way, a descriptor with multiple child descriptors only references one of them, which in turn references another of the parent descriptors' child (and so on), as depicted in Fig. 2. With such a definition, each descriptor is defined as the tuple {OFFSET, {HSIZE, STRIDE, VSIZE, SPAN, DSIZE}, LEVEL, NEXT} and is paired with a unique identifier and the corresponding data type of the addressed memory elements.

By adopting such a chained structure, the resolution of the addressed positions is obtained by traversing this tree structure in child-priority order (see Fig. 2). Hence, for each iteration of a given descriptor, its child descriptor (referenced by the NEXT field) should be completely solved once. Subsequently, when the first is itself completely solved, the procedure will solve the following descriptor that shares the same parent descriptor with it (referenced by the LEVEL field). Fig. 3 illustrates this resolution procedure applied to an example descriptor tree.

B. Promotion of Automatic Data Reutilization

The proposed descriptor structure was devised to also target an efficient management and manipulation of the flowing data-streams through the SME, as it will be seen in Section III. For such purpose, each data element/block that composes a given data-stream is conveniently tagged with a *stream identifier*,

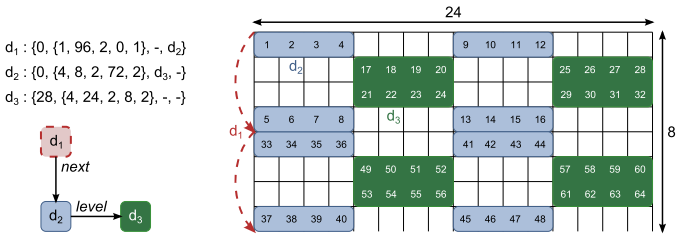


Fig. 3. Example of a data-pattern descriptor tree. Note the order in which blocks are accessed. d_1 calculates a relative offset for both d_2 and d_3, which generate two distinct patterns in relation to such offset.

unique to its corresponding stream, and with a *serial number* indicating its position/order in the stream. Such ordering mechanism of the streamed data makes each individual stream analogous to a contiguous memory block of indefinite size, which, in turn, allows applying the proposed descriptors in order to extract a sub-patterns from the flowing data streams. Accordingly, based on the proposed descriptor specification, it is possible to apply run-time stream manipulation operations (e.g. stream splitting and merging), as well as rerouting operations over the flowing streams between the PEs and memory.

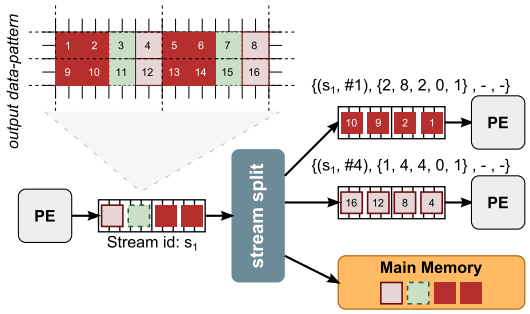
In the implementation of such operations, the OFFSET field of the output stream descriptor is replaced with the {ID, SERIAL} tuple, specifying a stream identifier and a starting serial number (analogous to a memory offset). In accordance, any of these operations can be simply applied by flowing the input streams through a descriptor solving structure and by extracting the data blocks according to the output stream descriptor pattern and stream identifiers.

With this approach, different stream operations can be easily implemented at the SME, thus significantly alleviating the addressing of the shared memory device (which would have to support such operations if the offered data reutilization facility was not directly provided at the SME level). Fig. 4(a) depicts an example where a subset of the data flowing from a PE to memory is extracted and redirected to a different PE (useful in data-dependent applications). Fig. 4(b) shows another example, where several streams generated at different PEs are interleaved in a single stream and sent to another PE (useful in reduction operations).

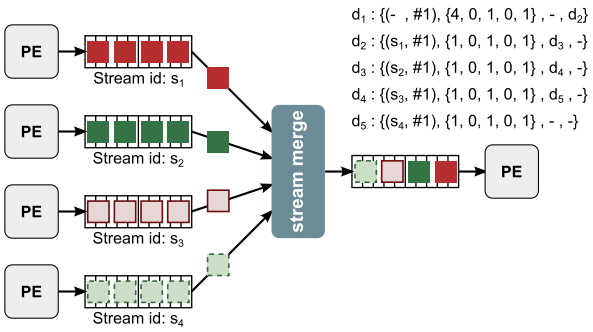
In accordance, such techniques promote the reutilization of data, by keeping it in the SME as long as further computation may be required, mitigating one of the most recurring concerns in many-core systems which is the pressure on the main memory system, due to the large number of PEs performing data transfers.

III. Stream Management Engine

The herein proposed Stream Management Engine (SME) was specially devised in order to minimize the number of main memory accesses and to promote data reutilization inside the multi-processor system, not only to maximize the data throughput to the PEs, but also to decrease the energy consumption associated with repeated memory accesses. Such data reutilization techniques allow keeping data in local and intermediate memories and buffers as long as they are necessary for subsequent computations by other PEs, minimizing the need for continuously accessing the main memory to store



(a) Stream Splitting.



(b) Stream Merging.

Fig. 4. Implementation of common stream manipulation operations with the proposed descriptors. Notice that in (a) while the stream is being stored in memory, part of it is being redirected to other PEs. On the other hand, in (b) four different streams are being merged in a new stream in a interleaved pattern.

partial results and make them available between the different PEs of the system. Furthermore, by adopting such a stream-based communication paradigm, it is also possible to mitigate most of the inherent contention in shared communication infrastructures, by making use of data acquisition modules that directly address the main memory and distribute the data to the appropriate PEs in parallel, instead of allowing them to concurrently request data from the main memory.

The proposed SME, depicted in Fig. 5, is composed of a set of controlling structures that allow the efficient generation and management of data streams and their communication to the system's PEs. In particular, it comprises: *i)* two memory access controllers, namely the Data-Stream Controller (DSC) and the Memory Store Controller (MSC), both based on the same Descriptor Tree Controller (DTC) architecture (illustrated in see Fig. 6 and subsequently described in Section III-A), capable of interpreting the proposed data-pattern descriptors described in Section II to load/store data from/to the main shared memory; *ii)* Stream FIFO memory modules, distributed across the communication infrastructure, able to buffer multiple data streams; *iii)* a stream-based communication bus with broadcast functionalities, capable of simultaneously streaming data to any of the accelerator's PEs; *iv)* Sub-Stream Generator (SSG) controllers, also based on the DTC architecture presented in Fig. 6, capable of splitting, merging and rerouting data streams coming from the PEs and the main memory, and redirecting them to their appropriate destination; and *v)* a Stream Management Controller (SMC), which manages the SME control-flow and controls and monitors the PEs execution through a simple backbone communication bus.

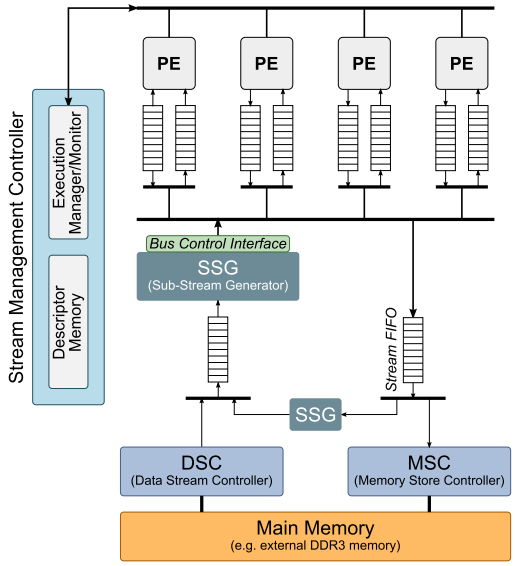


Fig. 5. Stream Management Engine architecture overview.

A. Descriptor Tree Controller Architecture

In order to efficiently address the memory access pattern described by the proposed descriptor tree specification, the DTC architecture must use the least number of clock cycles (per memory address) as possible. For such purpose, the proposed DTC architecture was conveniently divided in two parallel sub-modules: *i)* a control unit, herein denoted as Tree Solver Unit (TSU), responsible for iterating over the descriptor tree; and *ii)* an Address Generation Unit (AGU), responsible for generating the memory addresses according to the described pattern and starting at the offset address defined by its parent descriptor. These two units communicate over a register bank and operate completely in parallel. Hence, while the AGU is executing a given *address*-type descriptor, the TSU iterates over the tree to calculate the relative offset for a subsequent descriptor. A minimal local scratchpad descriptor memory is used to store the descriptors being processed.

The AGU functional unit, depicted in Fig. 6, iterates over a descriptor according to the addressing function defined in Eq. 1. In order to keep the architecture footprint as low as possible, it is solely based on adders. Hence, the AGU is simply connected to a control and a status register bank and comprises three parallel functional blocks, each composed of an adder and specific operand selection and control logic. The *stride control* block is responsible for incrementing an `inc` variable, representing the current contiguous data block, and for generating the required multiplication factors, by successively adding the `stride` and `span` descriptor fields to the `voffset` and `doffset` values, respectively. The `voffset` and `doffset` are intermediate values initialized with the current descriptor `offset` field. The *offset control* block calculates the actual memory address based on the `voffset`, `doffset` or `offset` values and the `inc`, `stride` or `span` variables, depending on the current descriptor state. The *count control* block is used to calculate the current descriptor iteration state, by incrementing the `i`, `j` and `k` values (refer to Eq. 1). Such values are limited by the `hsize`, `vsize` and `dsize` descriptor fields, respectively. In accordance, a set of

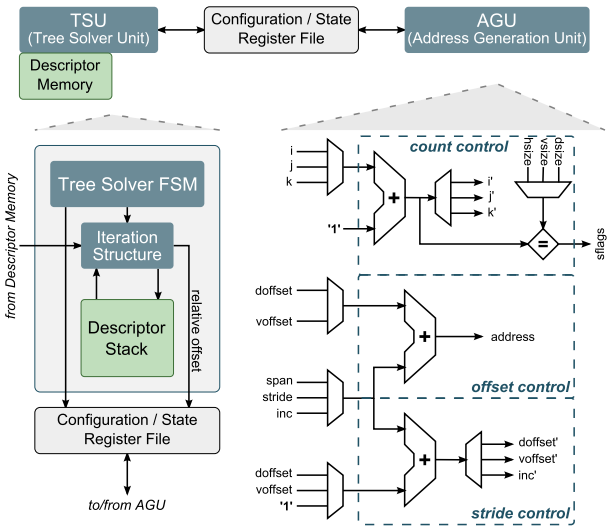


Fig. 6. Descriptor Tree Controller and its functional units architecture.

control flags (`sflags`) is generated at the end of the logic path of the *count control* block, to represent the iteration state of the descriptor and it is used to control all three functional blocks in terms of operand selection and reset logic. This way, each functional block performs one iteration per clock cycle, involving the computation of the current memory address, the multiplication factors for the next iteration and the next descriptor state, together with its corresponding control flags.

The TSU control unit, also represented in Fig. 6, is composed of: *i)* a Finite-State Machine (FSM) that deploys the descriptor tree solving procedure (see Section II-A); *ii)* an iteration structure, composed by the same three-adder topology used in the AGU, used to perform single iterations over the tree's *offset*-type descriptors (see Fig. 1); and *iii)* a descriptor stack, to store the *address*-type descriptors state.

The TSU is initiated upon the reception of an offset and a descriptor identifier (from the SMC), iterating over the descriptor tree in the child-priority order (described in Section II-A). This way, *offset*-type descriptors are iterated once by the TSU and pushed into the stack, as the TSU is going down through the tree's hierarchy. This approach eases the operation of the TSU since the order in which the descriptor states are pushed into the stack is the reverse order in which they are needed when the TSU is going up the tree's hierarchy.

As referred in Section II, the proposed DTC architecture was defined in order to ensure not only an efficient generation of the memory addresses, but also as a flexible means to implement a vast set of stream manipulations. Hence, three different wrappers were devised, based on the same DTC architecture:

1) Data-Stream Controller: implements a dedicated wrapper to the proposed DTC architecture to issue memory read operations to the main memory and to generate the resulting data-streams. The included memory access logic makes use of the memory addresses generated by the DTC to issue memory read requests. The obtained data is tagged with ordered serial numbers and with a stream-unique identifier. The generated data-stream is sent to an output Stream FIFO.

2) Sub-Stream Generator: makes use of a slightly different version of the DTC to generate the sub-streams, which main differences concern the specific descriptor modifications described in Section II-B, such as the replacement of the OFFSET descriptor field by the {ID, SERIAL} tuple. Hence, instead of generating a memory address, the modified architecture generates serial numbers, coupled with stream identifiers. The SSG also includes a logic block that monitors incoming streams and extracts data blocks according to the stream unique identifier and the last serial number generated by the DTC.

3) Memory Store Controller: makes use of the original DTC architecture to generate memory write operations. Specific monitoring logic is used to capture on-the-fly the incoming streams that should be stored in the main memory. The involved memory access logic makes use of the memory addresses generated by the DTC architecture to issue the memory write operations.

B. Streaming Infrastructure

In order to communicate, buffer and accommodate the streamed data as long as possible inside the SME, a number of Stream FIFOs are included in the streaming infrastructure. Formally, each of the system PEs will have associated two (input and output) configurable-size FIFOs. Similarly, both the DSC and the MSC are connected to the communication bus through a Stream FIFO. This way, all the coexisting data-streams are transfered between FIFOs, mitigating the usual communication latencies by hiding data communication behind data processing. The devised Stream FIFOs are implemented with a common double pointer control structure and the stored data comprises a streaming data-block, together with its serial number and its corresponding stream unique identifier. Fig. 5 depicts the location of each Stream FIFO in the SME.

Although the proposed architecture can be easily be adapted to any stream-based communication infrastructure, the communication of data-streams between the SME Stream FIFOs is herein achieved by a specially devised bus interconnection, which provides single-cycle communication between a master and a number of peripherals, featuring two independent unidirectional channels: a one-to-many channel and a many-to-one channel. The first channel routes data signals from the master to the peripherals in one of two ways: *i)* establishing a direct connection between the master and one of the peripherals, by using a decoder driven by the identification of one of the peripherals; or *ii)* making use of a broadcast mask to simultaneously connect to a set of the peripherals. The second channel routes data signals from each of the peripherals to the master. The channel is managed by a round-robin arbiter, driven by request and acknowledge signals, from and to the peripherals, respectively.

C. Stream Management Controller

The SME is managed by a specially devised Stream Management Controller (SMC) responsible for monitoring and controlling the PEs execution and the data-stream flow, maintaining a central descriptor memory and assigning descriptors to the DSC, MSC and SSGs.

In order to monitor and control the PEs execution, the SMC maintains a set of status registers indicating the current state

TABLE I. RESOURCE USAGE FOR EACH COMPONENT OF THE SME

	Available Resources	DTC	DTC w/ wrappers	Stream Interconnect (2-16 PEs)	1K Stream FIFO	SMC & Backbone (2-16 PEs)
Slices	75,900	498	557	8 - 83	21	324 - 399
LUTs	303,600	952	1046	11 - 202	57	692 - 883
Registers	607,200	692	717	3 - 16	29	426 - 439
BRAM	3,090	7	7	0	2	1
Static Power*	-			207.42		
Dynamic Power*[1]	-	22.78	25.2	0.12-2.02	7.33	8.37-10.16

* Power consumption values displayed in mW [1] @100 MHz

TABLE II. ADDRESS GENERATION RATE AND DESCRIPTOR SIZE (IN BYTES) FOR THE PROPOSED DTC AND THE RELATED WORK.

Pattern Type	Pattern Length (# words)	Proposed DTC		DFC (from [6])		AXI DMA [8]	
		Size	Addr/cycle	Size	Addr/cycle	Size	Addr/cycle
Linear	1024	16	1	24	1	32	0.96
Tiled	128×72[1]	32	1	40	0.99	32	1
Diagonal	1024×1024	128	1	44	1	65k	1
Zig-Zag	8×8	208	1	48 (132*)	0.36 (0.71*)	480	0.63
Cross	1024×1024	48	1	132	0.89	228k	1

* Values obtained after loop unrolling [1] Within a memory block of 512×512

of the each PE. Moreover, a backbone communication bus, implementing the same protocol as the above described streaming interconnection is used by the SMC to send execution configuration commands to the PEs and to receive execution status results from them. The latter is not only used to monitor their execution but also to obtain control-relevant application results. This is particularly useful in applications with varying streaming patterns along the time - example: diamond search algorithm, used in the video encoding motion estimation step, where the data-pattern (i.e. the descriptor) varies depending on the search direction in the frame [9].

Although the SMC could easily be paired with a dedicated scheduler to manage the complete SME, it is out of the scope of this work. Instead, a static execution queue is used to control the execution flow (i.e. starting the execution of the PEs and configuring the streaming interconnection with the correct broadcast destination for stream redirection).

IV. EXPERIMENTAL EVALUATION

To evaluate the proposed tree-based descriptor specification, its ability to promote data re-utilization and also the devisedSME, it was prototyped in a Xilinx VC707 Evaluation Kit, comprising a XC7VX485T Virtex-7 FPGA and a 1GB DDR3 SODIMM 800MHz/1600Mbps memory module (MT8JTF12864HZ-1G6G1). The Synthesis and Place&Route procedures were performed using Xilinx ISE 14.5. The power consumption of each of the system's components was estimated with the Xilinx Power Estimation toolchain and the DDR3 memory power consumption was calculated according to the vendor's guidelines and estimation tool [10]. Accurate clock cycle simulations were performed with the Xilinx iSim simulator. The obtained results for address generation and efficiency were compared with the most relevant related work, namely with a Xilinx AXI DMA engine [8], whose functionalities are equivalent to those of the PPMC [5] and with the Hotstream framework [6].

A. Hardware Resource Overhead

Although the subsequent parameters are completely configurable, for this experimental evaluation 128-bit wide descriptors were considered, i.e. all the descriptor's fields are 16-bit wide, except for the OFFSET field and the NEXT and LEVEL references, which are 32- and 8-bit wide, respectively. The SMC features a 255×128-bit memory used to store the application descriptors. Also, the local scratchpad memory of each of the DTC-based controllers is 16×128 bits in size. This particular scratchpad memory size assumes that, by default, no descriptor tree requires more than 16 descriptors. For the Stream FIFOs, it was considered a size of 1024 stream elements (which include a 32-bit data block, a stream unique identifier and the data block serial number).

The SME (depicted in Fig. 5) implementation results obtained for the Virtex-7 FPGA prototyping device are detailed in Table I. The presented results show that despite the complexity of the proposed operations, in what concerns descriptor solving and stream manipulation, the devised hardware modules incur in a very low resource overhead. In fact, the entire SME hardware structure (for a 16 PE configuration) requires only 4.5% of the FPGA resources. Also, the efficiency of the SME structure itself (not accounting for the main memory power consumption) is demonstrated by the estimated power consumption values, which are below 355 mW. Finally, as can also be concluded by analyzing the values on Table I, the SME hardware structure shows to be highly scalable, requiring only 5% of additional hardware resources when increasing the number of PEs from 2 to 16.

B. Data-pattern generation efficiency

In order to properly evaluate the proposed DTC descriptor solving efficiency and compare it with the considered state-of-the-art PPMC [5] and Hotstream [6] solutions, a representative evaluation benchmark was performed. To allow comparison with [5] and [6], the same set of data patterns that were considered in [6] is herein used, namely: *Linear*; *Tiled*; *Diagonal*; *Zig-Zag*; and *Greek Cross*. Fig. 7 depicts the considered patterns and their corresponding descriptor specifications, whereas Table II presents the DTC efficiency in solving tree-based descriptors. For comparison purposes, the table also shows the efficiency of state-of-art approaches (taken from [6]) when solving the corresponding descriptors.

By analyzing Table II, it is clear that the proposed DTC architecture provides a steady one-address per cycle generation rate, whereas the considered state-of-the-art pattern generation structures show a significant performance degradation for high complex patterns. This is rather important since the address generation rate typically constraints the system throughput, specially in memory-bound applications. Another advantage of the proposed system, when compared with the state-of-art approaches, regards the memory requirements for storing the actual data access pattern. As it can be concluded from Table II, the proposed tree-based pattern descriptor leads to an overall reduction in the memory size, achieving up to $2.7\times$ and $4800\times$ memory savings for the *Greek Cross* pattern, when comparing with the HotStream [6] and with the PPMC [5] (through the AXI DMA [8]).

It can also be ascertained from Table II that in some of the considered patterns the Hotstream DFC requires less memory occupancy for its pattern description code. Although it does not require significantly less memory, it must be kept in mind that,

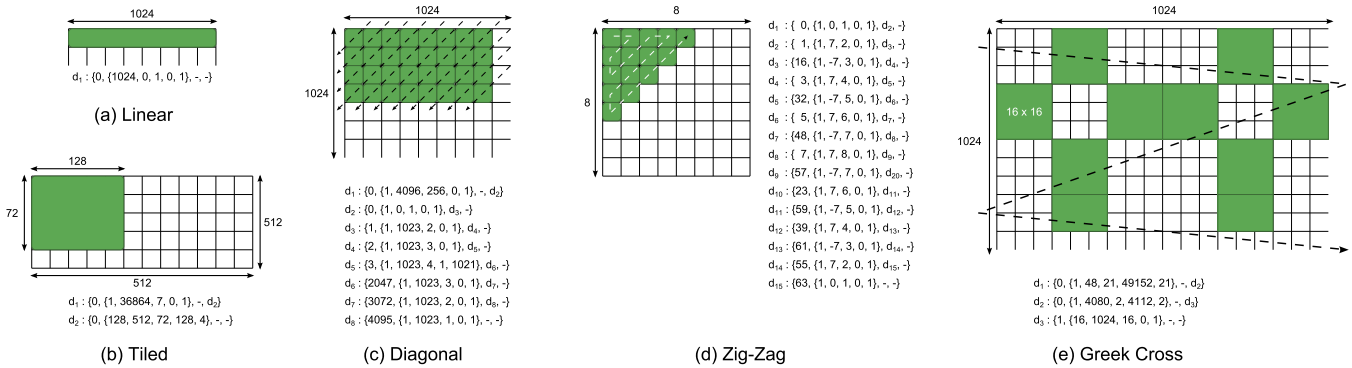


Fig. 7. Considered data-patterns and the corresponding descriptor trees.

in what concerns the main memory address generation, the Hotstream framework deploys two DFC controllers per PE [6], while in the SME there are only two (the DSC and MSC) independently of the number of PEs. This way, with a slightly higher memory size that is required in some cases to store the proposed descriptor tree specification, it is possible to achieve a higher pattern generation rate (for any described pattern), with significantly fewer data-fetching hardware structures. As such, not only is most of the FPGA fabric left for computing structures but it also allows reducing the total energy consumption of the data-management infrastructure.

C. Case Study A: Matrix Multiplication

In order to further demonstrate the proposed SME capabilities, specially in what concerns promoting data reutilization, a block matrix multiplication case study was specially devised. For comparison purposes a shared memory communication paradigm is used as a baseline, and compared with the proposed SME, with different levels of available parallelism, in terms of number of memory accesses and in memory-related energy consumption.

The benchmark application performs the multiplication of two 4096×4096 matrices (A and B), divided in 32×32 sub-blocks. In order to do so, a 128:1 reduction step is required to accumulate intermediate sub-block results (matrix C). Accordingly, the computing infrastructure is composed of a number of PEs, each one comprising: *i)* a pipelined multiply-add core, based on Xilinx IP cores; *ii)* a local 128Kb scratchpad memory, capable of storing a single column of 32x32 blocks; and *iii)* two 1K Stream FIFOs, one for data input and one for data output and result reutilization.

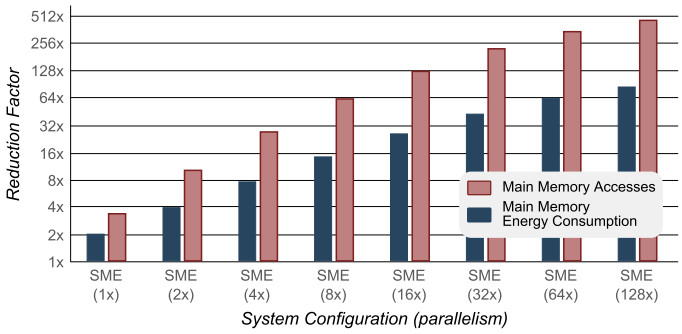


Fig. 8. Reduction factor observed in memory access and memory-related energy consumption for the blocked matrix multiplication benchmark.

The application has an inherent parallelism which can be exploited by calculating each matrix C sub-block column in parallel. This achieved by performing the multiplication of each sub-block column from matrix B with all the sub-block rows from matrix A and then performing reduction steps for each matrix C sub-block. By taking advantage of the Stream Interconnect broadcast capabilities and of PE local memories, it is possible to initially stream a sub-block column from matrix B to each of the PEs (which is then stored in the local memory) and then broadcast the entire matrix A to every PE. Moreover, by reusing the partial data stored in the PE output Stream FIFO, it is possible to eliminate the otherwise time-consuming reduction steps by accumulating the partial sub-block results and only stream them at the end of the computation. With this approach, matrix B is streamed only once (one sub-block column per PE), whereas matrix A is streamed a number of times depending on the number available PEs. Hence, to exploit the maximum achievable parallelism, the computation can be distributed by 128 PEs (since there are 128 sub-blocks columns in matrix B). If fewer numbers of PEs are used, each will have to process more than one sub-block column, and receive the entire matrix A that many times.

In Fig. 8, it is presented a graph of the memory access reduction factor of the SME, depending on the level of available parallelism, when compared to the considered shared memory configuration. As it can be ascertained from the figure, when using the maximum available parallelism, it is possible to greatly reduce the number of memory accesses by as much as 85×. Not only does this reduce the pressure on the external DDR3 memory (possibly allowing for a reduction in operating frequency), but it also allows for a significant reduction in the memory energy consumption (see Fig. 8). From the figure it is also possible to observe the energy efficiency of the proposed data-reuse techniques, since even with no parallelism the memory-related energy consumption is reduced 3×. Moreover, when using the maximum available parallelism it is possible to achieve an energy consumption reduction by as much as 475×.

D. Case Study B: Biological Sequence Alignment

The proposed 3D tree-based descriptor specification was devised not only to ease the description of arbitrarily complex data-access patterns, but also to allow run-time stream manipulation operations. In order to demonstrate such capabilities, a second case study based on a biological sequence alignment application [11] is presented. The application calculates the

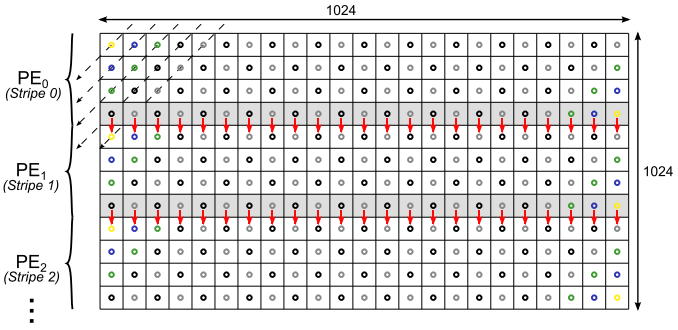


Fig. 9. Biological sequence alignment diagonal striped computing pattern. Dashed arrows represent the data flow direction; full arrows represent the data dependencies between stripes. The highlighted data must be extracted from the corresponding stream and sent to the PE computing the subsequent stripe.

alignment score between a reference and a query sequence (with 1024 elements each), by making use of a substitution score matrix between each of the sequences' elements. The score of matching a pair of elements from each sequence is stored in a corresponding matrix cell and is calculated based on the scores of three other cells (left, top and top-left).

In order to overcome such data dependencies the computation can be performed in diagonal order of the matrix cells [12]. However, the address calculation for such a memory access pattern presents a considerable overhead to an otherwise straightforward set of matrix cell computations. Hence, by making use of a *Diagonal* data-pattern descriptor similar to the one presented in Fig. 7(c), with the proposed SME, it is possible to completely eliminate that overhead.

The computation can also be parallelized, which can be done be dividing the score matrix in stripes that are computed by different PEs. This approach results in a data-dependency between the PEs (see Fig. 9), which, when considering a shared memory communication paradigm, requires performing subsequent writes and reads to the memory. Such an issue is easily solved with the SME, by making use of the SSGs in order to extract the required data from the output stream of a PE and send it to another PE that depends on that data for its computation.

In order to demonstrate both the efficient pattern generation and the data-manipulation capabilities of the SME infrastructure, both a 16-PE SME setup and 16-PE shared memory communication setup where designed in order to process two sequences (query and reference) each composed of 1024 elements. Although the obtained results show only a small reduction in the number of memory accesses by 1.13×, since only a small amount of data can be reused, a 6.5× processing speedup is observed, resulting from the elimination of the memory address generation overhead required in the shared memory setup.

V. Conclusion

In this manuscript, an energy-efficient stream-based data-management infrastructure was proposed. The devised Stream Management Engine (SME) structure relies on a 3D tree-based descriptor specification, capable of easily describing any arbitrarily complex access pattern. In order to decode such descriptors, a DTC architecture was specially devised that is capable of one-address per clock cycle pattern generation efficiency. The DTC architecture and the proposed tree-based descriptor specification are also used by a SSG controller, which is capable of extracting sub-stream patterns from on-the-fly data-streams. Such functionality is used to deploy a number of techniques for data-reutilization, such as stream splitting, merging and rerouting operations. The obtained esults demonstrate that combined data-reuse and manipulation facilities of the proposed SME allow reducing the number of memory accesses by as much as 85x, which in turn results in a 475× reduction in energy consumption related to memory accesses.

Acknowledgment

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project Threads (ref. PTDC/EEA-ELC/117329/2010) and project UID/CEC/50021/2013.

References

[1] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 247–259.

[2] Y. Guo, P. Narayanan, M. A. Bennaser, S. Chheda, and C. A. Moritz, "Energy-efficient hardware data prefetching," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 2, pp. 250–263, 2011.

[3] J. Park and P. Diniz, "Synthesis of pipelined memory access controllers for streamed data applications on fpga-based computing engines," in *Proceedings of the 14th international symposium on Systems synthesis*. ACM, 2001, pp. 221–226.

[4] J. Park and P. C. Diniz, "Data reorganization and prefetching of pointer-based data structures," *IEEE Design and Test of Computers*, vol. 28, no. 4, pp. 38–47, 2011.

[5] T. Hussain, M. Shafiq *et al.*, "PPMC: a programmable pattern based memory controller," in *Proceedings of the 8th international conference on Reconfigurable Computing: architectures, tools and applications*, ser. ARC'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 89–101.

[6] S. Paiágua, F. Pratas, P. Tomás, N. Roma, and R. Chaves, "Hotstream: Efficient data streaming of complex patterns to multiple accelerating kernels," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on*. IEEE, 2013, pp. 17–24.

[7] S. Ghosh, M. Martonosi *et al.*, "Cache miss equations: An analytical representation of cache misses," in *In Proceedings of the 1997 ACM International Conference on Supercomputing*. ACM Press, 1997, pp. 317–324.

[8] "LogiCORE IP AXI DMA v6.03a," Xilinx, Tech. Rep. PG021, 2012.

[9] S. Zhu and K.-K. Ma, "A new diamond search algorithm for fast block-matching motion estimation," *IEEE Transactions on Image Processing*, vol. 9, no. 2, pp. 287–290, 2000.

[10] "TN-41-01: Calculating Memory System Power for DDR3," Micron Technology, Inc., Tech. Rep., 2007.

[11] N. Neves, N. Sebastiao, D. Matos, P. Tomas, P. Flores, and N. Roma, "Multicore SIMD ASIP for Next-Generation Sequencing and Alignment Biochip Platforms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, July 2014.

[12] A. Wozniak, "Using video-oriented instructions to speed up sequence comparison," *Computer applications in the biosciences: CABIOS*, vol. 13, no. 2, pp. 145–150, 1997.