2013 25th International Symposium on Computer Architecture and High Performance Computing

HotStream: Efficient Data Streaming of Complex Patterns to Multiple Accelerating Kernels

Sérgio Paiágua, Frederico Pratas, Pedro Tomás, Nuno Roma, Ricardo Chaves
INESC-ID / IST, Technical University of Lisbon, Portugal
Email: sergio.paiagua@ist.utl.pt, {fcpp, Pedro.Tomas, Ricardo.Chaves, Nuno.Roma}@inesc-id.pt

***Abstract*—Designing accelerating kernels is a comprehensive task that requires efficient coupling of hardware and software. In particular, the structures responsible for handling data transfers in multi-core accelerator-based systems play a crucial role in the resulting performance. This paper proposes a data streaming accelerator framework that provides efficient data management facilities that are easily tailored for any application and data pattern. This is achieved through an innovative and fully programmable data management structure, implemented with two granularity levels. The obtained results show that the proposed framework is capable of efficient address generation and data fetch for complex streaming data patterns, while significantly reducing the size occupied by the pattern description. A large matrices multiplication case-study, based on a streaming architecture with four sub-block multiplication cores, demonstrates that, by enabling data re-use, the proposed framework increases the available bandwidth by 4.2×, resulting in a performance speedup of 2.1×. Furthermore, it reduces the Host memory requirements and its intervention by more than 40×.**

***Keywords*—Stream Computing, Programmable Data (pre-)fetch Controller, Many-Core Heterogeneous Architectures**

I. Introduction

One of the most critical aspects in the development of custom multi-core accelerators is in the handling of data transfers between the various Processing Elements (PEs). The architecture of the memory subsystem and of the communication data channels has a significant impact on the memory bandwidth available to the PEs, and therefore on the overall system performance. In fact, an efficient management of the data transfers is important not only due to PEs having different processing characteristics and capabilities, but also because applications often present distinct memory footprints and bandwidth requirements.

While traditional solutions (such as hierarchical cache structures) try to reduce the data access latency, they do not allow exploiting all levels of available data parallelism. As a result, there was an increasing interest in stream computation models, which focuses on decoupling communication from computation, by exposing an additional level of concurrency. This type of concurrency is especially important in hardware accelerators, due to the slow communication channels (*e.g.*, buses) typically used to connect with the host device. Moreover, while regular streaming patterns are easy to handle, complex memory accesses require more radical strategies to avoid long memory access times and to keep a high system performance. Also, when data streams produced by a given kernel are consumed by several other kernels and at different paces, intermediate buffering is required. This further increases the pressure on the memory subsystem. To minimize the impact of these problems, dedicated Address Generation Units (AGUs) can be employed, which (pre-)fetch the data with the specific pattern required by the target application. Moreover, data reuse mechanisms can reduce the number of effective memory accesses by sharing some of the streams through alternative channels or by rearranging a stream before it is consumed by the next kernel.

The HotStream accelerator framework herein proposed consists of a Host Interface Bridge (HIB), and a Multi-Core Processing Engine (MCPE), which is capable of simultaneously processing an arbitrary number of stream-based kernels. These two major elements provide pattern-based data accesses with two granularity levels: a coarse-grained data access from the Host to the MCPE, to maximize the transmission efficiency; and a fine-grained data access between PEs within the MCPE, to maximize data reuse. The latter makes use of an innovative Data Fetch Controller (DFC), which extracts the data streams from an address-based shared memory, by using access patterns of arbitrary complexity. Moreover, instead of accomplishing the pattern description by traditional descriptor-based methods, the proposed framework relies on an innovative micro-coded approach, supported on a compact but rich Instruction Set Architecture (ISA). This allows efficiently describing data streams with complex memory access patterns, without compromising the address issuing rate.

The evaluation of the proposed framework shows that the embedded DFCs offer significant memory savings on the pattern description code. Compared to the existing related art, the proposed solution can achieve code size reductions above $1500\times$, with identical address generation rates. Considering block-based matrix multiplication as case study, experimental results suggest that the data-reuse offered by the proposed HotStream framework allow achieving a $2\times$ speed-up, relatively to the state of the art implementation. Moreover, the proposed solution is able to reduce the Host intervention in the acceleration by up to $45\times$, while requiring significantly less Host buffering.

II. Related Work

The popularity increase of stream-computing models have led to the development of specialized architectures that tackle the efficient fetching and management of data streams. Examples such as the IMAGINE stream processor [1][2] and the MERRICAM stream-based supercomputer [3][4], which are based on clusters of PEs and Stream Register Files (SRF), offer simple data pre-fetch mechanisms which, in the case of the IMAGINE processor, transfer entire streams between the SRF and an off-chip SDRAM. As stated by the authors, only

978-1-4799-2927-6/13 $26.00 © 2013 IEEE
DOI 10.1109/SBAC-PAD.2013.17

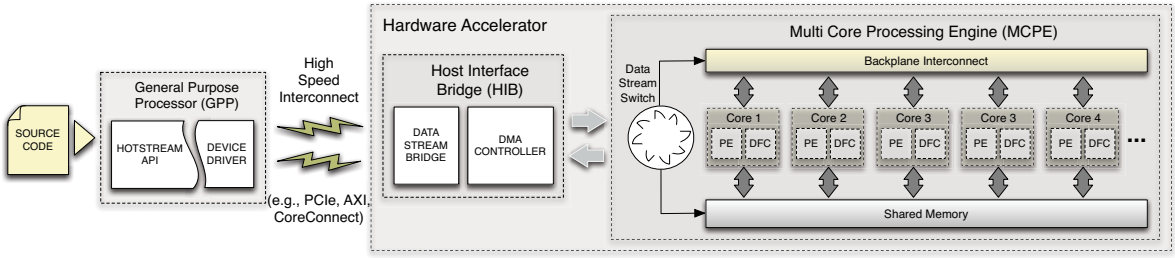

Fig. 1: Structure and organization overview of the HotStream framework

50% of the optimal performance is achieved, which motivates the development of more efficient data management structures.

Having also identified this bottleneck, other researchers have focused on improving the generation of data streams. After demonstrating that dataflow computing can lead to significant performance improvements in a wide range of applications, Pell et al [5] developed the MaxCompiler, which maps the computing kernels to an FPGA. To generate the streams, a set of commands is provided, which instructs the developed tool-chain to automatically generate simple 1D, 2D or 3D data patterns. However, more complex patterns can only be described by using multiple commands, which is both time-consuming and results in a large configuration overhead. The same shortcomings are experienced by the Programmable Pattern-based Memory Controller (PPMC) [6], which eases the programming of regular 1D, 2D or 3D patterns through a set of function calls, integrated in an API. Again, this solution falls short when long and/or complex patterns must be described.

The above data fetching solutions are actually very similar to modern DMA engines. For example, the Xilinx AXI DMA controller offers independent read and write channels that provide high-bandwidth communication between memory and PEs [7]. With its scatter-gather capabilities and the support for 2D transfers, it can actually be used as a pattern-generator. The configuration is done by setting up a chain of descriptors that are then read by the engine, making this a rather similar solution to the PPMC [6]. Moreover, multichannel support is offered through stream identifiers that accompany the data.

Both the PPMC and the AXI DMA are designed for moving large and regular data chunks and can fall short with complex access patterns.In contrast, the DFC herein proposed is capable of handling arbitrary patterns of varying complexity without significant penalties. Also, since the pattern description is not descriptor-based, there is essentially no limit to the length of the pattern to be generated.

III. HotStream Framework

The proposed HotStream framework, depicted in Fig. 1, is a comprehensive solution for the development of stream-based architectures, composed of a *software* layer and a *hardware* layer. The software layer integrates: *i*) a convenient API that allows the programmer to specify any arbitrarily complex streaming patterns, as well as *ii*) a Device Driver to map the user-specified memory buffers, allocated on the user space, to the physical address space. This device driver also serves as the data transfer peer, on the software side, that assures appropriate integration mechanisms with the hardware layer.

The hardware layer is composed of: *i*) the Host Interface Bridge (HIB), to handle data transfers between the host processor and the accelerator; and *ii*) the Multi-Core Processing Engine (MCPE), to manage the data streams between the PEs within the accelerator. The proposed architecture is designed to be fully scalable and adaptable (by supporting a variable number of data streams and PEs), as well as flexible enough to support applications with different and arbitrarily complex streaming patterns with minimal effort. Moreover, one of its main features is the support for efficient data fetch and reuse within the accelerator architecture. This is achieved by providing two distinct levels of data access patterns. The first level is implemented within the HIB and allows simpler patterns of a more coarse-grained nature like those supported by the PPMC [6]. The second and more fine-grained level of granularity is implemented within the MCPE and supports more complex streaming patterns.

A. Host Interface Bridge

Copying data from the Host processor memory system to the MCPE is a complex procedure. It requires the intervention of: *i*) the device driver, on the host side; and *ii*) a specific hardware structure, the HIB, that is able to autonomously issue data transfers (read/write requests) between the main memory of the host and the MCPE. The HIB mainly consists of two modules: the Data Stream Bridge (DSB), responsible for interfacing with the host GPP (e.g., using the PCIe protocol), and the Direct Memory Access (DMA) controller, which manages the transfer of data between the host GPP and the hardware accelerator using *coarse-grained data access patterns*.

Despite the implementation efficiency of the entities involved in a single data transaction, an unavoidable overhead is expected, which limits the effective channel bandwidth. While transferring large data chunks lead to a more efficient data channel utilization, complex streaming patterns often require accessing data that is not laid out linearly in the Host memory, but spread over a regular pattern of contiguous blocks separated by non-unit strides. In such cases, transferring the smallest data chunk that composes each set of contiguous blocks results in a waste of bandwidth. Thus, the proposed solution implements coarse-grained patterned data transfers between the Host and the MCPE, such that the HIB transfers only useful data but in large data chunks. This is accomplished by setting up the DMA controller to transfer each of the contiguous data blocks that constitute the pattern, according to the regions mapped by the device driver. The set up phase consists of creating scatter-gather DMA descriptors, arranged in a chain, and defining the starting position and size of the memory blocks to be read from or written to.

As the number of contiguous blocks described by a given

pattern increases, the number of required descriptors also increases in the same proportion. Hence, in order to minimize the impact of this increase, the simple and traditional DMA engine could be replaced by a more efficient alternative capable of performing, at least, the most regular 2D memory accesses, *i.e.*, each memory transaction can be described by the tuple {OFFSET, HSIZE, STRIDE, VSIZE}, specifying the starting address of the first memory block, the size of each contiguous block, the starting position of the next contiguous block with relation to the previous, and the number of repetitions of the two previous parameters, respectively. This reduces the total number of descriptors needed to describe a given pattern.

While the size and nature of the patterns applied to the data transfers between the Host and the MCPE are only limited by the available space to store the descriptors, these should not be too fine-grained in order to avoid a detrimental impact on throughput. Therefore, in the proposed HotStream Framework, the API provided to the programmer includes a special call to *gather* an arbitrary sequence of data segments stored across the memory space into one, larger and contiguous buffer that can then be transferred at once. This gathering operation takes a non-negligible time to complete, thus it is only useful when the incurred penalty does not exceed the overheads of transferring the smaller non-contiguous individual data chunks.

B. Multi-Core Processing Engine

The MCPE is where the actual computation takes place and is designed to support multiple independent and heterogeneous cores that collaboratively execute the multiple streaming kernels. Moreover, each kernel can span several Cores, to further exploit data parallelism. As depicted in Fig. 1, it consists of: *i*) multiple Cores, each composed of a PE, responsible for the computation, and one or more Data Fetch Controllers (DFCs), responsible for data management; *ii*) a high-speed Backplane Interconnection, able to dynamically route the data streams between the Cores, promoting the required data reusage schemes; *iii*) a shared memory, which allows rearranging the stream access patterns and data reutilization; and *iv*) a Data Stream Switch (DSS), to route the data streams coming from the host to either the backplane or to the shared memory. Accordingly, the data streams transferred from the host via the DSS or those produced by an individual Core can be routed to other Cores in the MCPE via the backplane interconnection or stored in the shared memory for later reuse.

The high-speed stream-oriented Backplane Interconnection must ensure that each Core can communicate with any other with a minimum routing delay. In addition, multiple connections may need to be active at any given time. Taking into account these requirements, a high-speed interconnection network is required. Sophisticated Network-on-Chip (NoC) solutions are likely to provide higher system scalability and better support for heterogeneity among the Cores in terms of data interfaces. However, one must also ensure that the amount of hardware resources required by the interconnection network is minimal, saving space for extra computing Cores.

The shared memory is particularly important in applications that require multiple access patterns or when data reusage is exploited between the PEs. Whenever a stream needs to be rearranged before it is consumed by another Core or streamed back to the Host machine, it can be buffered on the shared memory. As such, the shared memory must be accessible by all the Cores. A simple work-conserving round-robin arbitration mechanism makes sure that all read and write requests are served with equal priority and no starvation occurs. In addition, being an address-based element in an otherwise stream-oriented architecture, reading and writing operations require the inbound or outbound data streams to be accompanied by a stream of addresses. The generation of these addresses is carried out by the DFC unit, within each Core (further detailed in Section IV). These generate *fine-grained data access patterns* through small programmable units directly coupled with the PEs (one for each outbound or inbound stream). It results in pattern descriptions using much less memory space than the typical descriptor-based data-fetching mechanisms in the state-of-the-art approaches, such as the PPMC [6].

IV. Data Fetch Controllers and Shared Memory

The DFCs are undoubtedly the central and the most important elements of the MCPE. These units are responsible for single-handedly extracting the data from the (address-based) shared memory with arbitrarily complex patterns, and for forming the data streams that are presented to each Core's PE, while the latter remains completely oblivious as to the origin of the data that it is consuming. In particular, each DFC is responsible for generating the corresponding read and write data transactions, according to the defined streaming pattern.

Each DFC has its own instruction memory that is programmed from the Host machine through a compact but complete ISA, by using a custom assembler with syntax validation. This instruction memory can be dynamically updated and its size represents the only limitation to the complexity of the considered pattern. However, as long-running patterns can be described by common loop structures, the size of the instruction data is kept relatively small and independent of the extension of the pattern. In addition, the DFCs are optimized to take advantage of the features provided by the considered bus protocol (*e.g.* AMBA AXI), such as burst commands to minimize the inevitable overheads.

In order to handle these tasks, the DFCs incorporate two fundamental blocks that operate together: *i*) the Address Generation Core (AGC); and *ii*) a custom small-footprint 16-bit microcontroller (Micro16). The AGC is a small specialized processor that autonomously generates addresses in a linear, 2D or 3D fashion. On the other hand, the Micro16 microcontroller is capable of generating combinations of linear, 2D and 3D patterns that are sequentially requested to the AGC in order to construct more complex stream patterns. The two units interact via a small and shared register file, the External Register File (ERF). This way, while the AGC is generating a sequence of addresses, the microcontroller concurrently modifies the ERF with all the required parameters for the next regular pattern. The combination of these two units makes it possible to describe complex patterns, without compromising the address generation rate.

A. Address Generation Core (AGC)

The AGC effectively emulates traditional nested loops, as found on most programming languages, by specifying, for each loop level, the number of iterations to be executed. Each level

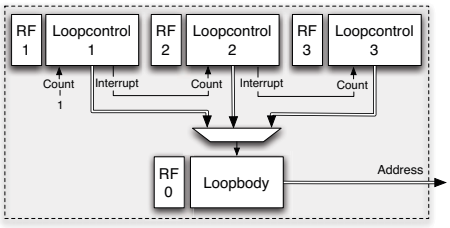

Fig. 2: AGC in a 3-level nested loop configuration.

is implemented through a *Loopcontrol* unit that independently counts down from a starting value to zero, generating an interrupt upon completion. In addition, each of these *Loopcontrol* units holds the necessary parameters to determine the starting address of the AGC during the next iteration of the loop level. By combining multiple *Loopcontrol* units in a daisy chain structure and by routing the interrupt signal of the innermost levels to the enable input of the outermost ones, an N-level nested loop can be designed. Figure 2 illustrates the required configuration to implement a 3-level loop. It should be noted that, regardless of the involved *Loopcontrol* units, the AGC is able to automatically configure all the necessary internal connections, based on a Generic VHDL parameter.

The body of the loop is emulated by the *Loopbody* unit, which generates one address per clock cycle based on three basic configuration parameters: *increment*, *multiplication*, and *initial* value. This trio makes it possible to generate any affine linear access pattern, *i.e.*, patterns of the type $y_n{=}y_{n-1}{\times}m{+}i$, which represent the great majority of the indexing needed by most scientific applications [8]. Hence, the *Loopbody* address generation is controlled by the associated *Loopcontrol* units, which interrupt the former whenever the iteration limit in any of the nested loop levels is reached. This results in a two clock cycle delay to compute the next starting address.

Considering that any delay in the data fetching procedure may potentially slow down multiple PEs, it is of the utmost importance for the address generation to be essentially continuous. While this is a reasonable requirement when the supported patterns are restricted to 2D or even 3D sequential accesses, supporting arbitrarily complex patterns with changing starting positions requires a more sophisticated approach. Therefore, for more complex patterns the configuration of the AGC relies on a double-buffered scheme, which is accomplished by duplicating the configuration registers used by the *Loopbody* and *Loopcontrol* units. With this architecture, the Micro16 is able to compute and configure the loop parameters for the following portions of the pattern, concurrently with the AGC execution, *i.e.*, without interrupting the address generation.

B. Micro16 microcontroller

The Micro16 is a custom microcontroller designed to configure and control the AGC. While the overall architecture follows a single-cycle RISC, it also comprises customized features aimed at easing the integration with the AGC. One such feature is the incorporation of an ERF, which is composed of the local register files from the *Loopcontrol* and *Loopbody* units within the AGC. Despite its external nature, any of the registers can be used as sources and destinations during ALU-based operations, with no additional latency. Additionally, two control bits (*Wait* and *Done*) are provided, to inform the Micro16 whether the ERF is ready to be modified and to report the AGC that the new parameters are ready to be used.

In light of the custom nature of the machine code defined by the Micro16 ISA, an assembler was developed to assist its programming. This tool offers syntax validation, label support, and overflow checking (when loading immediate constants either in base-10 or base-16). While the performance of the DFC is not directly influenced by these features, they greatly facilitate the pattern specification process, thereby increasing the ease-of-use of the HotStream framework, which is undoubtedly a key aspect of the proposed system. By taking advantage of such user-friendly assembly language, configuring a pattern is just a matter of populating the ERF with the relevant parameters and using the custom interface instructions to start and stop the address generation.

C. Access to the Shared Memory

The buffering capabilities of the HotStream framework are ensured by a single large-capacity shared memory, usually implemented by an external DRAM. This leads to complex timing characteristics, as the need for periodic refreshing and the cost of charging of data lines result in variable access times for arbitrary memory positions. To maximize bandwidth, modern Double Data Rate (DDR) memories offer special access modes to maximize the achievable data throughput. These are essentially burst-based accesses that retrieve a fixed number of sequential data beats from the DDR with one single command. Exploiting burst-based accesses to the DDR memory is, therefore, paramount to get the most out of the available memory bandwidth. Accordingly, the address stream that is generated by the developed AGC was shaped with this particular objective in mind.

As an arbitrary number of cores may be requesting data from the shared memory, a scalable and robust arbitration solution is also required. This may be achieved by adopting any of the currently available industrial-standard interfaces, specifically targeted at high-performance systems, e.g., the AXI4 specification, maintained by ARM, or the CoreConnect bus architecture, defined by IBM.

To comply with the specific bus architecture adopted for each particular application, a *Bus Master Controller* (BMC) was developed and integrated in the streaming framework. The purpose of this unit is to perform the conversion between the stream-based interface used by the DFC and the Memory-Mapped (MM) protocol used by the bus interface (see Fig. 3). To avoid placing additional pressure on the shared memory, each bus controller features a Stream-to-MM and an MM-to-Stream interface. These independent channels are arbitrated internally so that only one stream accesses the bus at a time.

This BMC also comprises convenient *write* and *read* control units, featuring appropriate internal buffers that enable the issuing of incremental burst-based transactions and, in the case of the latter, perform data pre-fetching by requesting data that will be buffered until the Core is ready to consume it (see Fig. 3). To accomplish this, both the *write* and *read* control units consume the address stream up until the point that an increment pattern is broken. In the particular case of the write channel, no data can be output until it is actually marked valid by the Core. Thus, a *synchronizer* block is also used, which forces the data and address stream to flow at the same pace.

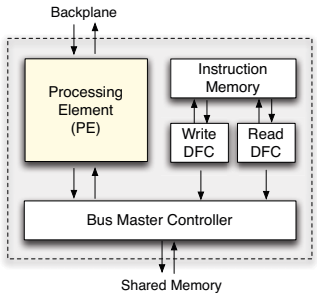

Fig. 3: Core internal structure, comprising the PE and the co-located BMC

V. HotStream Architecture Evaluation

The proposed HotStream framework represents a generic design that can be implemented in different environments. To evaluate the proposed framework, a Virtex 7 FPGA connected to a PC through a PCI Express interface was chosen as a prototyping vehicle. This configuration includes a high-throughput interface between the Host processor and the MCPE in the form of a PCI Express $\times 8$ connection, and a high-performance DDR 3 memory device, with 512 MB of capacity and a peak bandwidth of 12.8 GB/s. The HIB module was implemented by a Xilinx AXI DMA IP core and the Backplane Interconnect was implemented by means of a high-speed network in a crossbar topology, where all the network interfaces use the AXI4 protocol. The obtained results were compared with the most relevant related art, namely the PPMC [6]. However, since the PPMC implementation was not publicly available, the Xilinx AXI DMA engine was used as baseline, as its functionalities are identical to the PPMC [6].

A. Resources Overhead

Considering the strong focus on scalability that was given to the proposed streaming framework, it is paramount that its core elements do not significantly impact the overall resource usage. This goal was achieved by designing the DFC with a low area footprint in mind, as this is bound to be the most replicated unit in this framework. Table I summarizes the resource utilization by the key elements composing the framework when implemented on a XC7VX485T Virtex-7 FPGA. It is important to note that the DFC is fully configurable with relation to the number of *Loopcontrol* units used. Thus, its resource occupation varies with the chosen configuration, as shown in Table I where DFC configurations with 1 to 3 *Loopcontrol* units are presented.

It is important to note that, while the resource utilization of the Backplane Interconnect (implemented in a Crossbar topology) seems rather high, it represents a worst-case scenario, configured to support full connectivity and 16 independent nodes. On the other hand, each DFC accounts for only 1.6% of the total resources available in the device, which ensures the addressed scalability goal. While competitive for FPGA-based designs, the maximum operating frequency of the DFC is limited by the simple pipelined nature of the used microcontroller. By adopting a more aggressively optimized architecture, higher processing frequencies can be achieved.

TABLE I: Resource usage for each component in the MCPE (16 cores)

	Available Resources	DFC (1-3 Loopcontrol units)	Streaming Bus (AXI)	Backplane Interconnect	HIB
Slices	75,900	1,014 - 1,216	3,273	4,875	6,848
LUTs	303,600	1,743 - 2,225	5,305	8,882	16,208
Regs	607,200	1,553 - 2141	4,922	8,656	13,288
DSPs	2,800	4	0	0	0
BRAM	3,090	1	0	0	6
Max.Freq.		160 MHz	167 MHz	146 MHz	136 MHz

B. Stream Generation Efficiency

Given the relation between the complexity and nature of the considered patterns and the address generation rate and size of the pattern descriptor, a proper evaluation of the proposed DFC and overall framework can only be achieved through a representative benchmark. Therefore, five distinct patterns of varying complexity were considered: *Linear*; *Tiled*; *Diagonal*; *Zig-Zag*; and *Greek Cross*. While the first two are usually found in a wide range of applications, the remaining three are somewhat more exotic in nature. Nevertheless, they are still of great importance in the context of stream-computing: the *Diagonal* access pattern is extensively used by the Smith-Waterman algorithm for DNA sequences alignment [9]; the *Zig-Zag* scanning is a key element in the entropy encoding of the AC coefficients in the JPEG and MPEG standards [10]; and the *Greek Cross* is often used by a vast class of diamond search motion estimation algorithms adopted in video encoding [11]. Figure 4 shows a representation of the described access patterns, including their size and evolution over time, as well as the pseudo-code for their generation using the proposed API.

The considered metrics for this evaluation are the size of code to describe each pattern and the address generation rate, defined as the average number of addresses generated per clock cycle. As stated above, the AXI DMA engine is used as the baseline for this comparison, representing the characteristics of most descriptor-based pattern generation mechanisms that have been proposed in the related art. Table II depicts the obtained results. It can be concluded that the proposed controller achieves a similar address generation rate but with significantly lower code-memory requirements. Moreover, the related art does not offer any form of scalability, as the size of the descriptor increases with the length of the pattern. This is particularly emphasized for the *Diagonal* and *Cross* patterns: for an 1024×1024 pattern, the conventional DMA descriptor occupies about $1500\times$ more memory than the proposed DFC approach. For larger matrices this discrepancy will be even greater. Due to the larger code size, the conventional DMA approach requires either a significantly larger internal memory or an external processor to dynamically generate the patterns, which further increases the required hardware resources and decreases the attained performance.

For some cases such as the *Zig-Zag* access pattern, the execution time of the pattern description code in the DFC cannot be entirely overlapped with the address generation,

TABLE II: Address generation rate and descriptor size of the considered access patterns (the pattern length results from the parameterization in Fig. 4)

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

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

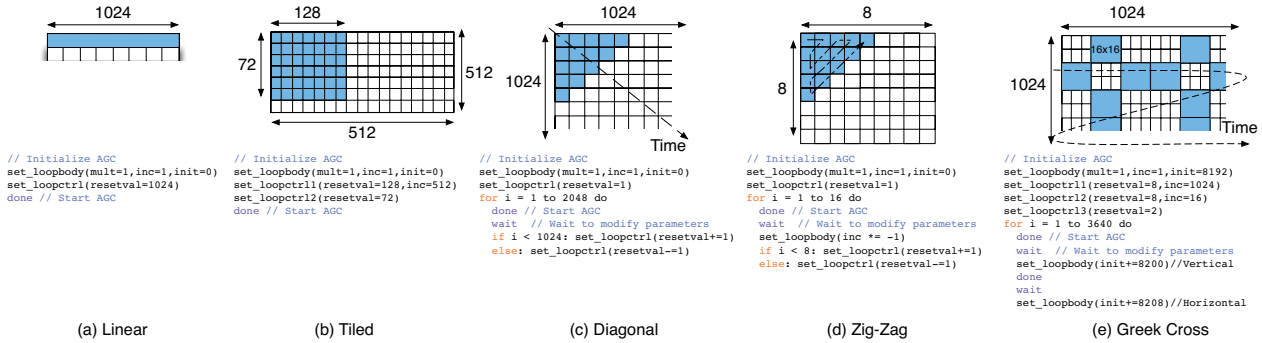

Fig. 4: Access patterns, with varying complexity degrees, adopted for the DFC evaluation

resulting in a low address generation rate. To circumvent this problem, the loop that sets the AGC parameters for each diagonal can be unrolled, at the cost of a slightly larger code size. This technique effectively provides a duplication of the address generation rate for the *Zig-Zag* pattern (values marked with an * in the table).

Naturally, the actual performance gains that can be achieved by accelerating a given data streaming application with the proposed framework depend not only on the amount of parallelism that can be exploited, but also on the involved computational complexity (*i.e.*, the number of operations performed on a single data element). The latter is especially important, as it effectively defines the amount of data reuse that can take place within the framework. In order to provide an insight of the speed-up magnitudes that can be expected by utilizing the HotStream framework, the following section presents a case study for the particular case of the block-based matrix multiplication. While this is a relatively simple example, the discussed concepts are general enough to be applied to more complex applications.

VI. Case Study: Matrix Multiplication

In this section, a block-based matrix multiplication example is used to evaluate and compare the proposed framework with other usual approaches based on hardware accelerators. The proposed framework provides efficient data streaming mechanisms between the host and the accelerating hardware, as well as extensive data (re-)usage and (pre-)fetching capabilities within the MCPE. This increases the effective data bandwidth available to each processing core and maximizes the processing throughput. In contrast, traditional implementations use a host GPP or a conventional DMA engine to centralize the data management, at a detrimental cost of being limited by the data bandwidth of the underlying communication interface between the host and the accelerating hardware.

Block-based matrix multiplication, is typically used because it improves data locality and allows operating on matrices much greater than would be possible if the multiplication was performed in a single step. The considered implementation is divided into two steps: *i)* the multiplication of the sub-blocks; and *ii)* the accumulation (reduction) step, to compose the final matrix with the computed partial sub-matrices [12].

Three different approaches were considered for this evaluation. The first, hereinafter denoted as *conventional*, simply streams the matrix data over the PCI Express link into the accelerator, where a single matrix multiplication core is

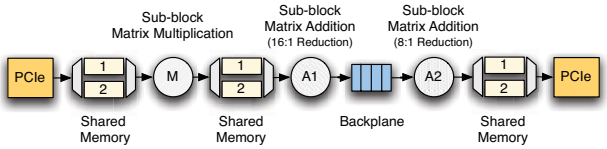

Fig. 5: HotStream implementation of the block multiplication algorithm, with 3 Kernels and the multiple concurrent data streams; double buffering is used on the shared memory to overlap communication with computation

consuming the incoming data and producing the results that are streamed back to the Host. The second approach, which we refer to as *conventional+buffering*, features an additional memory in the accelerator, which is large enough to buffer one of the input matrices, so that it can be reused over the course of the entire computation. Finally, the third approach makes use of the HotStream framework, which maximizes the data re-usage by including reduction (accumulation) modules on the MCPE that run concurrently with the multiplication cores (see Figure 5), as well as overlapping the data communication with the computation by employing double-buffering techniques. It should be noted that to implementing a 4096×4096 matrix multiplication with 32×32 sub-blocks, a 128:1 reduction step is required. This is achieved by using two addition kernels: one to perform a 16:1 reduction and the other to perform an 8:1 reduction. Thus, while the conventional approaches perform the reduction step on the host CPU, the proposed framework only streams the final resulting matrix back to the Host.

In addition to these three basic implementations, corresponding parallel versions were considered, by replicating the structure depicted in Fig. 5. The level of exploited parallelism is limited either by the available hardware resources or by the data bandwidth capacity of the communication channels.

A. Computing Cores

Since the main focus of this case study is on the HotStream framework and not on the matrix multiplication cores, off-the-shelf Xilinx IP Cores were used to implement both the matrix multiplication and the accumulation cores. To make the comparison fair, the same multiplication units are used in the conventional solutions. These soft cores run at a conservative frequency of 100 MHz, support matrices of up to 32×32, and output a 2 byte matrix element per clock cycle, after a significant initial latency (although it is effectively hidden by the large data set that composes the stream). Therefore, a constant rate of 100 MOps (Million Operations per Second) is maintained by each core.

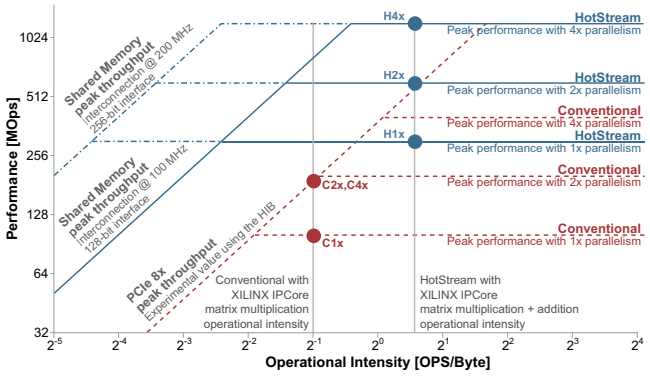

Fig. 6: Roofline model for the matrix multiplication example: Cx and Hx denote the actual performance of the Conventional and HotStream implementations, respectively; while the conventional solutions C2× and C4×, with 2× and 4× parallelism, respectively, are limited by the PCIe link (communication-bounded), all other implementations are computation-bounded

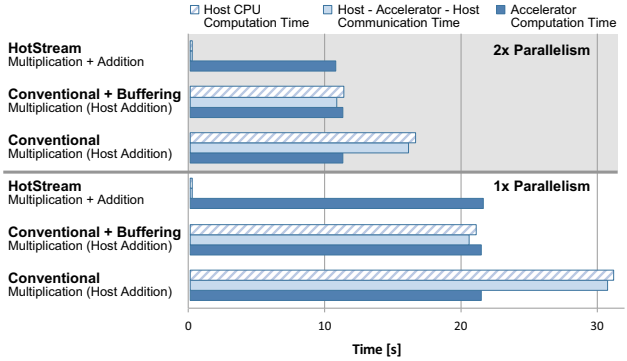

Fig. 7: Processing time taken on each step of the matrix multiplication algorithm for the considered implementations

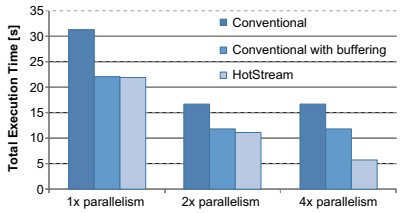

Fig. 8: Core scalability of the three matrix multiplication implementations

B. Roofline Model

To evaluate the available design space and correlate the exploited processing performance with the throughput of the involved communication channels, the Roofline model [13] was applied to the considered case-study. Figure 6 depicts the peak performance of the conventional implementations, using parallelism levels of $1\times$, $2\times$ and $4\times$, which result from using 1, 2 or 4 Multiplication Cores, respectively, to implement the matrix multiplication kernel (data parallelism). For the $1\times$ parallelism level, the conventional solution is limited by the matrix multiplication XILINX IP Core performance. By increasing the number of cores to 2 or 4, $2\times$ or $4\times$ parallelism can be achieved. However, these implementations become limited by the PCIe, thus resulting in a performance of only 190 MOps, i.e, a speed-up of only 1.9 with $4\times$ parallelism.

It is possible to increase the communication roofline and overcome the previous limitation by simply enabling data re-use within the accelerator. Accordingly, by just applying the proposed HotStream framework to implement the same kernels, a speed-up of about 2.1 is achieved, corresponding to a performance of 400 MOps. However, the HotStream framework also allows to easily develop more aggressive solutions that use addition kernels to perform the sub-block accumulation on the accelerator. This has the advantage of increasing the accelerator operational intensity from 0.5 operations per byte (OPS/Byte) to 1.5 OPS/Byte. By using the Xilinx IP Cores to also implement sub-block matrix addition, it is possible to observe a peak performance of 300 MOps for the dataflow illustrated in Fig. 5. Furthermore, this value can still be increased if data parallelism is added to this implementation: by replicating each of the kernels in Fig. 5 $4\times$, a peak performance of 1200 MOps is observed (see the HotStream roofline with $4\times$ parallelism in Fig. 6).

C. Performance and Memory Usage

Another key advantage of the HotStream framework when compared to conventional solutions is the reduction of data traffic on the PCI Express link. These savings can be rather significant for large data sets, allowing the application to run faster while reducing the intervention of the Host GPP. In fact, as the amount of transferred data over the PCI Express connection decreases, so does the number of DMA descriptors that must be setup by the Host. While this operation can be overlapped with the actual data transfer, it wastes valuable Host processing time. Moreover, the accelerator computation time is also improved when utilizing the HotStream framework, as the processing Cores benefit from the increased throughput of the shared memory, when compared to the PCI Express link. Thus, the overall throughput is less likely to become communication bound, as it occurs in the conventional solutions with a parallelism level of $2\times$ or greater. Figure 7 depicts the execution times stripped into different communication/computation domains for the considered matrix multiplication implementations, using a 4096×4096 matrix. To achieve a comparable performance to what is possible with the proposed HotStream framework, the *conventional* implementations must be able to completely overlap communication with computation. However, this comes at the cost of an higher intervention of the Host GPP, to manage the communication with the accelerator. Figure 7 clearly shows that the HotStream implementation is able to reduce the Host processor occupation by $45\times$ for a 4096×4096 matrix multiplication.

The three bars on the left of Fig. 8 represent single-kernel configurations of the three considered implementations. In this particular setup, the total execution time of the *conventional+buffering* implementation is very similar to the one provided by the HotStream implementation (whose Host processing time is not considered, since it can be overlapped with the computation). The slight performance gap that is observed in the *conventional+buffering* implementation is mainly due to the overhead introduced by the need to perform the reduction step in the Host. For the considered 4096×4096 matrix multiplication, this step corresponds to more than 2×10^9 operations, which take about 486 ms to execute in a state of the art Intel Core i7-3770K@3.5 GHz (8 MB of L3 cache). Furthermore, as was concluded from the conducted Roofline analysis (see

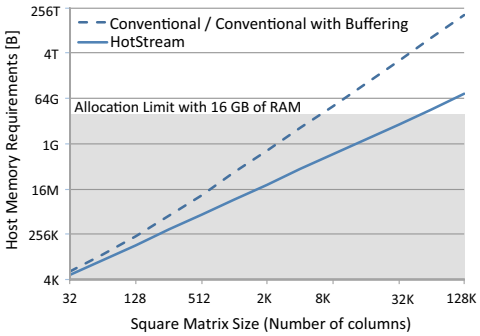

Fig. 9: Host memory requirements for matrix multiplication implementations

subsection VI-B), the HotStream implementation is never communication bounded, whereas the *conventional* approaches become so at $2\times$ parallelism. This is well demonstrated in the remaining columns of the graph, where almost linear speed-up values are achieved with the proposed framework and not with the conventional ones.

The last fundamental parameter that was considered refers to the memory that is required in the host to store the matrices and intermediate results. This is a very important scalability measure, as it effectively determines the maximum matrix size that can be handled by the accelerator. It is expected that all implementations (except the HotStream solution) fall short in this domain. In particular, the *conventional* solutions require a Host buffer that is several times larger than the whole matrix size, in order to hold all the block-based intermediate results of the multiplication operation. As depicted in Fig. 9, for 4096×4096 matrices, the *conventional* implementations require storing the full input and output matrices, as well as an additional 4GB buffer to store the intermediate results. In contrast with the HotStream implementation, the matrix size is only limited by the total capacity of the Host memory, which must be capable of holding the three full matrices, *i.e.*, 96MB, leading to a $42\times$ memory requirement reduction. Naturally, to enable data re-use within the accelerator, the shared memory must be large enough to hold three sub-blocks of the input and output matrices, as well as a small subset of extra sub-blocks, that guarantee the overlapping of the communication and computation. This is easily achieved with the 512MB of DDR memory available on the Virtex 7 FPGA board.

Finally, the scaling of the operation for larger matrices can be considered. The *conventional* implementations are not able to multiply 8192×8192 matrices, since a buffer of about 32 GB would be needed. In contrast, less than 1 GB is required with the proposed framework (storing of the input and output matrices), clearly demonstrating the ability of the HotStream framework to scale and deal with large data sets.

VII. Conclusions

This paper proposes the HotStream framework, featuring a novel architecture for the development of efficient and high-performance stream-based accelerators. When compared to existing pattern generation mechanisms, such as the PPMC [6], the HotStream framework proposes programmable Data Fetch Controller structures to implement fine-grained pattern descriptions. It provides considerable gains in terms of the hardware resources, as well as in what concerns the storage requirements of the pattern description code. This is achieved without any imposed compromise in the address issuing rate, while still allowing for a compact and simple pattern description code.

While the HotStream framework was designed to be platform independent, it was prototyped in a Virtex 7 development board, to properly evaluate the proposed solution. A block-based matrix multiplication, operating over large matrix sizes, was adopted as the evaluation case-study. Experimental results show that conventional solutions that do not exploit data-reuse can be easily constrained by the PCIe communication link. On the other hand, by using the proposed framework, it is possible to increase the core available bandwidth by $4.2\times$ thus leading to a 2.1 performance speedup in a $4\times$ data parallelism approach. Furthermore, by easing the implementation of more complex solutions, further speed-ups can still be achieved. In particular, it allows to implement the matrix reduction step directly on the accelerator and to alleviate the computational requirements of the Host processor. The final solution allows for a reduction in Host Memory requirements by $42\times$, consequently allowing for processing much larger matrices.

VIII. Acknowledgements

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under projects HELIX (ref. PTDC/EEA-ELC/113999/2009), Threads (ref. PTDC/EEA-ELC/117329/2010), P2HCS (ref. PTDC/EEI-ELC/3152/2012) and PEst-OE/EEI/LA0021/2013.

References

[1] U. Kapasi, W. Dally *et al.*, "The imagine stream processor," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE, 2002.

[2] U. Kapasi, S. Rixner *et al.*, "Programmable stream processors," *Computer*, vol. 36, no. 8, pp. 54–62, 2003.

[3] W. J. Dally, F. Labonte *et al.*, "Merrimac: Supercomputing with streams," in *ACM/IEEE Conference on Supercomputing*, 2003.

[4] M. Erez, J. Ahn *et al.*, "Analysis and performance results of a molecular modeling application on merrimac," in *ACM/IEEE International Conference on Supercomputing*, 2004, pp. 42–42.

[5] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *Computing in Science Engineering*, vol. 14, no. 4, pp. 98–103, 2012.

[6] T. Hussain, M. Shafiq *et al.*, "PPMC: a programmable pattern based memory controller," in *8th International Conference on Reconfigurable Computing*, 2012.

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

[8] S. Ghosh, M. Martonosi *et al.*, "Cache miss equations: An analytical representation of cache misses," in *ACM International Conference on Supercomputing*, 1997.

[9] R. K. Karanam, A. Ravindran, and A. Mukherjee, "A stream chip-multiprocessor for bioinformatics," *SIGARCH Comput. Archit. News*, vol. 36, no. 2, pp. 2–9, May 2008.

[10] G. Wallace, "The JPEG still picture compression standard," *Consumer Electronics, IEEE Transactions on*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

[11] 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.

[12] M. D. Lam, E. E. Rothberg *et al.*, "The cache performance and optimizations of blocked algorithms," *SIGPLAN Not.*, vol. 26, no. 4, pp. 63–74, Apr. 1991.

[13] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 99, no. RapidPosts, p. 1, 2013.