Moorthy and Gopalakrishnan *Journal of Cloud Computing: Advances, Systems and Applications* (2017) 6:20
DOI 10.1186/s13677-017-0089-9

Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH **Open Access**

IO and data management for infrastructure as a service FPGA accelerators

Theepan Moorthy* and Sathish Gopalakrishnan

Abstract

We describe the design of a non-operating-system based embedded system to automate the management, reordering, and movement of data produced by FPGA accelerators within data centre environments. In upcoming cloud computing environments, where FPGA acceleration may be leveraged via Infrastructure as a Service (IaaS), end users will no longer have full access to the underlying hardware resources. We envision a partially reconfigurable FPGA region that end-users can access for their custom acceleration needs, and a static "template" region offered by the data centre to manage all Input/Output (IO) data requirements to the FPGA. Thus our low-level software controlled system allows for standard DDR access to off-chip memory, as well as DMA movement of data to and from SATA based SSDs, and access to Ethernet stream links. Two use cases of FPGA accelerators are presented as experimental examples to demonstrate the area and performance costs of integrating our data-management system alongside such accelerators. Comparisons are also made to fully custom data management solutions implemented solely in RTL Verilog to determine the tradeoffs in using our system in regards to development time, area, and performance. We find that for a class of accelerators in which the physical data rate of an IO channel is the limiting bottleneck to accelerator throughput, our solution offers drastically reduced logic development time spent on data management without any associated performance losses in doing so. However, for a class of applications where the IO channel is not the bottle-neck, our solution trades off increased area usage to save on design times and to maintain acceptable system throughput in the face of degraded IO throughput.

Keywords: Infrastructure as a service, FPGA acceleration in data centres, Heterogeneous computing

Introduction

Parallel and heterogeneous computing with the use of discrete device accelerators like GPUs, FPGAs, and even ASICs, are some of the recent system level attempts that have addressed an increasing demand for computational throughput [1, 2]. However, heterogeneous computing is challenging at both the systems level implementation and resource management aspects of deployment. Therefore, data centres are starting to fill a market need for computing horsepower—without end users having to tackle such challenges directly. This sort of market has been termed Infrastructure as a Service (IaaS) within the more general category of cloud computing services.

Within the IaaS domain, there have been recent attempts in literature to facilitate the integration of device accelerators into cloud computing services [3–6]. These works primarily focus on scaling the use of accelerators in IaaS in three different ways; firstly, by treating the accelerator as a virtualized hardware resource, secondly in the case of FPGAs, by making use of their dynamic reconfigurable capabilities directly without virtualization, or thirdly, by offloading greater portions of traditionally host CPU based code onto accelerator based soft or hard processor cores. The work in this article is complementary to these efforts to integrate accelerators into cloud computing services, but we focus on a singular aspect of this broader challenge. We deal with the systems level implementation of integrating accelerators. We draw attention to the fact that present day accelerators may rely on direct access to IO data channels to offer effective acceleration.

High Level Synthesis (HLS) has gained some traction in offering acceptable levels of performance relative to

*Correspondence: tmoorthy@ece.ubc.ca
The Department of Electrical and Computer Engineering, The University of British Columbia, Vancouver, BC, Canada

manual RTL design for FPGA based hardware accelerators. More specifically, HLS is presently capable of transforming task specifications in a high-level programming language (such as C or Java) into near optimal hardware implementations for the compute portions of hardware acceleration [7]. However, successful acceleration of the compute portions alone does not translate into the desired application level acceleration as well. The increases in computational throughput must be supported by a corresponding increase in Input/Output (IO) data throughput to achieve system level acceleration. Unfortunately, HLS at present offers very little support or design abstraction for implementing custom IO data transfers and their corresponding organization in the memory subsystem, leaving system designers to still fall back on manual Register Transfer Level (RTL) design for these tasks. In fact, managing ingress and egress data across multi-level on-chip and off-chip memory hierarchies is an open HLS problem [7].

To mitigate some of these existing limitations of HLS in virtualizing FPGA usage in the cloud, recent work has provided further innovation [8, 9]. Ma et al. in their work [8], have combined existing development on Domain Specific Languages and the use of FPGA overlays to offer a runtime interpreter solution to managing FPGA resources. However, because they have taken an overlay approach, as they have cited, they also must offer standard accelerator *interface* templates for any IO to the FPGA. Such IO interfaces would indeed provide a great amount of flexibility for FPGA accelerator usage, but would suffer on optimality for any specific accelerator's data access pattern. Chen et al. [9] in their related work on FPGA based cloud computing, go further to recognize that there may be miscellaneous peripherals such as Ethernet controllers and various memory controllers required by specific accelerators. As such they make reference to a context controller switch to handle varying IO sources, but do not elaborate on whether any optimization is being made on this IO interface on a per accelerator application case basis.

In IaaS, this issue of customized IO and memory throughput for accelerators is further exacerbated by the fact that an end-user of a cloud computing environment does not have access to modify the bare underlying hardware's IO in any way. This is due to security limitations in the case of FPGAs [3], and in the case of GPUs and ASICs once their IO controller data-paths are designed and deployed they are fixed. Thus even if an end-user is willing to commit the manual RTL development hours required to customize the IO interfaces for their accelerated application, their lack of control and visibility over the underlying physical IO channels would prohibit them from doing so. Moreover, it is sometimes desirable to *virtualize* accelerators and enable time-sharing, and allowing for significant user customization may not be possible in this scenario.

With such limitations in mind, if IaaS is to be a viable option for those seeking FPGA based acceleration, a flexible method to access IO data channels must be offered by the infrastructure itself. Furthermore, a level of abstraction must also be offered by any such method in order for the underlying infrastructure to be upgraded or changed and not require changes in the end user's applications as a result.

We seek to complement HLS based acceleration by abstracting the design efforts required to integrate IO and memory data channels. We do so by investigating the use of the existing embedded ecosystem on modern FPGAs to handle IO data channels in software. Although this approach may not offer the same performance when compared to automated custom configuration methods [10], and far less effective relative to manual RTL design, it still offers design times that are comparable to an HLS approach. More importantly, it allows for data transfers to be handled in software. Certain classes of FPGA devices today have an embedded ecosystem as hardened components readily available; making use of these embedded resources via software control abstracts away the exact details of the embedded processor and physical IO channels. This gives IaaS providers the freedom to carry out infrastructure upgrades without disrupting their customer workloads. And for the end user customers, this provides the option to harness any future hardware upgrades or changes in underlying IO technology at a software level should they wish to do so.

Towards demonstrating the value of an embedded IO processor in FPGA-based accelerators, we use two case studies to demonstrate the value of IO management in data-intensive applications as well as the possible performance implications of embedded IO processors. *Although we have focused on FPGAs as the substrate for implementing accelerators, we believe that the central idea of utilizing an embedded IO processor will be an attractive approach for other accelerator implementations as well.*

FPGAs as managed hardware in IaaS data centres

Whether FPGAs are integrated into data centres via proposed virtualized methods [3] or by more direct Open-CL based methods [11, 12] the FPGA fabric itself will need to consist of two separate regions. A static region of logic will exist in order for the FPGA to bootstrap itself and configure the minimum communication protocols necessary for it to interact with the host system. The second (dynamic) region will be an area of the fabric that is partially reconfigurable at runtime to allow for custom logic to be placed onto the device. This custom logic will then accelerate a specific compute-intensive task call of the application's algorithm, or run the application as a whole entirely on the FPGA.

The dynamic region of the FPGA will be accessible to the end user to configure via the FPGA vendor's partial reconfiguration tool flow. The static region's base "template", however, will not be accessible to the end user and must be provided for and maintained by the data centre host. The boundary (Fig. 1) between these two static and dynamic regions of the FPGA fabric is what this article concerns itself with.

We refer to the static region as a template above because this region is what defines the structure of external IO data channels that the FPGA end user sees and has access to. The static region accomplishes this by instantiating the necessary DRAM memory controllers, PCI express endpoint blocks, and any other specific IO controllers that it wishes to provide. It is worthy to note here that although the FPGA device itself may be physically wired to multiple IO channels on the Printed Circuit Board (PCB) that was manufactured for its data centre usage (Fig. 2), the static region itself does not necessarily need to instantiate a controller for all of the available IO channels. What this allows for is different price points within IaaS for a data centre to rent its FPGA resources at, while keeping their actual physical FPGA infrastructure uniform with a fixed cost. For example, a data centre may have all of its FPGA PCBs designed with four Ethernet ports available, but intentionally restrict an end user from only accessing one or two based on what level of access they have paid for. In Fig. 2 this is illustrated by depicting multiple IO protocols and channels that may be physically wired to FPGA boards, however, the actual IO resources that an end user has access to can easily be limited by a static region bitstream in the SDAccel flow. The static region bitstream will not have an IO controller for all of the available IO protocols such as Ethernet, SATA, JESD, or SERDES, unless the end-user has subscribed to a design flow that includes these options (at a cost).

Apart from the business side advantages that mandating a static region on the FPGA provides, a static region also allows physical IO channel upgrades in technology while offering minimal disruption to existing applications and customers. Consider that the data centre upgrades the solid state storage devices that its FPGA customers have access to from SATA 2 to SATA 3 SSDs. In this scenario, only the SATA controllers within the static template would need to be redesigned and deployed, while keeping the partially reconfigurable dynamic portion of FPGA logic unaffected. However, this begs the question as to whether this is feasible or even possible in the actual implementation of these static and dynamic regions. Given that there is to be very tightly linked and high throughput communication between these two regions, if a component within the static region changes even slightly will this not necessitate a redesign in the modules of the dynamic region that it interacts with. The answers to these questions rest completely on what the boundary logic between these two regions consists of.

The boundary region at the implementation level merely represents the set of architectural choices that are made to design and determine the type of interfaces that the static and dynamic regions will use to accomplish data transfers between them. This work contributes an embedded method to manage data channels that strays away from what conventional RTL design practices would dictate. We also put forth the novelty of using our embedded

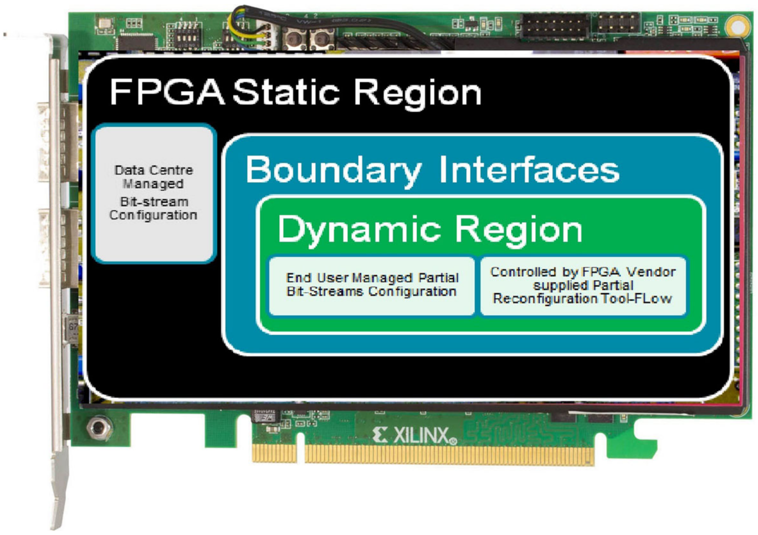
FPGA Static Region
Data Centre Managed Bit-stream Configuration
Boundary Interfaces
Dynamic Region
End User Managed Partial Bit-Streams Configuration
Controlled by FPGA Vendor supplied Partial Reconfiguration Tool-FLow
XILINX

Fig. 1 FPGA Fabric's Static, Boundary, and Dynamic Regions on a PCIe based Data Centre accelerator PCB card

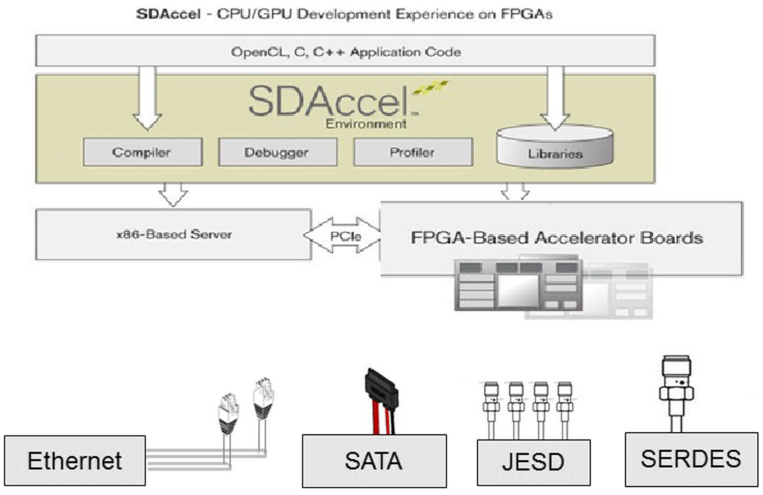
SDAccel - CPU/GPU Development Experience on FPGAs
OpenCL, C, C++ Application Code
SDAccel Environment
Compiler
Debugger
Profiler
Libraries
x86-Based Server
PCIe
FPGA-Based Accelerator Boards
Ethernet
SATA
JESD
SERDES

Fig. 2 Varying Physical IO Channels Directly Accessible by an FPGA within a Data Centre Host Environment, for an FPGA vendor's (Xilinx) software development flow

method in data centres where FPGAs may contribute as managed hardware resources within IaaS environments. This domain is an emerging and rapidly evolving field where end users are given increasingly less control over the bare-metal hardware that they seek to use. They must also accept certain levels of abstraction in their usage if they are to reap the economically cheaper rewards of scaling their FPGA acceleration based needs.

We explain our solution to IO data channel management in this environment by way of two use case examples that follow in "Bioinformatics application study" and "Video application study" sections. Towards demonstrating the value of our solution, in the first case we show that there are no performance penalties in adopting IaaS acceleration for the majority of data intensive or IO bound applications. In the second case, where IO performance loss does present itself as a problem, we successfully show that the acceleration itself can be scaled to compensate/offset IO performance loss at the application level if need be.

Bioinformatics application study

Application characteristics

Applications that require access to, or that produce, large volumes of data are presently loosely defined as "Big Data" applications. Big Data type applications, where FPGAs are utilized for acceleration, can be found in various application domains, ranging from Financial Computing, Bioinformatics, and conventional Data Center usages [13–15].

The particular application chosen for the experiments of this section falls under the Bioinformatics domain. However, the IO, memory, and data volume characteristics of this application allow for the results to be prevalent across most FPGA accelerated Big Data applications in general. Traditional, in house, computing-cluster based infrastructures for such applications may not be economical as their data needs continue to scale, and can be costly to maintain. The take away that we would like to stress in this section is that, in IO constrained cases of these applications, performance does not have to be sacrificed when moving to IaaS.

The underlying common denominator, from a hardware perspective, of such applications, is that they all require the use of a non-volatile storage device to handle the large data sets. The use of non-volatile devices, such as Solid State Drives (SSDs) or other flash memory based hardware, often necessitates a memory hierarchy within the system design process. A typical hierarchy will include off-chip flash memory for large data sets, followed by off-chip DRAM for medium sized data sets, and finally on-chip SRAM.

For the off-chip memory components, the physical IO data channel by which these components are accessed also adds a lot of variation to the system level design. For example, in the Financial Computing domain of applications, large volumes of off-chip data might be streaming in from multiple Ethernet channels, whereas in Data Centers the incoming data might be arriving over PCI-express links connected to SSD storage. This variation in the physical IO channels creates the need for multiple IO controllers that rely on different IO protocols for communication, which in turn results in varying interface requirements for each controller.

The FPGA application accelerators in this class will require the IO controllers to supply a high throughput of data that at least matches the targeted scale of acceleration throughput. This required acceleration throughput can often be much greater than the physical bandwidth of the underlying IO device, the throughput rate of the controller that is accessing the underlying device, or any combination of the two. This dynamic results in the high throughput potential of the accelerator being limited by low IO throughput at the systems level. For the system architects that have the financial resources to do so, overcoming this problem might be a possibility by investing in higher throughput IO channels (i.e. converting from 2nd generation SATA to 3rd generation, or increasing the number of lanes in a PCI-express link). However, cases, where even state of the art physical IO channels cannot match accelerator throughput requirements, are not rare [16].

The use case application chosen for this section demonstrates the operation of two different IO protocols, Ethernet and SATA, and more importantly illustrates the effort that is required to integrate their respective controllers into the FPGA-accelerated data path. In doing so, we find that indeed the limitations in controller throughput do become the bottleneck to system level performance. Within the latter part of this section, we move to demonstrating how such IO bottlenecks in the system can be exploited to create interfaces to these controllers that are software driven and less complex to design.

The DIALIGN algorithm

Bioinformatics algorithms generally lend themselves well to hardware acceleration due to their inherent amount of parallelization. DIALIGN [17] takes an input query sequence against another input reference sequence, and aligns/matches segments of the query sequence to the regions of the reference sequence that offer the highest (optimal) degree of similarity. The output of DIALIGN is a mapping of the query sequence coordinates to various reference sequence coordinates (i.e. regions).

The input data to DIALIGN is a series of 1-byte characters that represent either DNA or protein strands of query and reference sequences for comparison (cross comparisons between DNA strands and protein strands are never made). DNA representation requires only 4 distinct characters thus a 2-bit data representation would suffice, however, protein variations require greater than 16 characters to represent. Thus an 8-bit data representation is favoured to accommodate these two types of input sequences.

Acceleration in hardware is achieved by a systolic array architecture implemented by Boukerche et al. [18]. Their systolic array architecture (Fig. 3) stores each character of the query sequence within each PE, then streams the reference sequence across the chain of PEs to compute a matrix scoring scheme. Ideally, this architecture performs best when all of the characters of the query sequence can entirely fit into the total number of synthesizable PEs. Given that query-sequence sizes of interest today are in the mega characters range, this ideal scenario is not feasible even with relatively large FPGA devices. Thus several passes of the reference-sequence must be made across multiple partitions of the query-sequence.

DIALIGN implemented with fully RTL based IO interfaces

There are two IO channels that the DIALIGN application running on our FPGA platform relies on. The first is the Ethernet channel for external communication, and the second is the SATA channel for internal deep storage access.

Between the two IO channels, the Ethernet channel has the simpler interface to the accelerator. Here we are using the word interface, not in the sense of a standard protocol by which the ports of two or more communicating hardware modules must be connected, but in the more general sense of any necessary changes in frequency, data-width, or timing that must be performed in order to achieve a viable data-path between any two hardware components. The two primary components of interest in this article are the accelerator (or more specifically, the systolic array for this particular application) and an IO controller. The interface is then, the mechanism by which the IO controller transfers data to the native input or output ports of the accelerator. This mechanism, by definition, should at a minimum support two basic features—data width conversions and cross clock domain stability.

Our system was implemented on the Digilent XUP-V5 development board [19], which features a 1-Gbps physical Ethernet line, two SATA header/connectors, and the Xilinx XC5VLX110T Virtex 5 FPGA. Our system consumed 88% of the device logic with the accelerator and the (hardware) IO interfacing resources combined. With the accelerator synthesized to utilize 50 PEs, the system is capable of operation at 67.689 MHz. The SSD connected to our board is the Intel SSD_SA2_MH0_80G_1GN model, with a capacity of 80 GBs. It offers SATA 2 (3 Gbps) line rates, and states 70 and 250 MB/s sequential write and read bandwidths respectively under technical specifications by the manufacturer. A 1-gigabyte-reference and 200-character-query synthetic sequence were generated on the Host side for experimentation (Fig. 4).

The reference sequence is at most 1 byte per character, and at our synthesized accelerator operational speed of 67.689 MHz, this requires only 68 MB/s of Ethernet controller throughput to sustain the accelerator's maximum throughput level. Therefore, this required Ethernet throughput rate of 68 MB/s does not become a bottleneck to the system, and thus the Ethernet interface specifications will not be discussed further.

In comparison to the Ethernet controller the SATA controller, however, does pose significant system bottleneck issues.

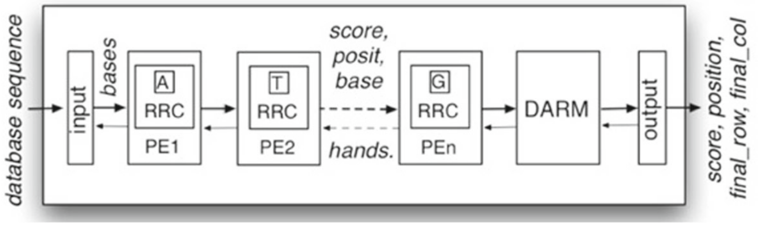

Fig. 3 Array Architecture for Single-Partition processing [18]

Intermediate data storage support between partitions

Given that partition switches across the PEs are necessary to support increasing lengths of query-sequences, the demand that this process places on intermediate storage requirements when gigabyte reference sequences are streamed is now described. Three partition switches across a systolic-array architecture of 50 PEs daisy chained together, effectively creates the logical equivalent of 200 PEs being used in a similar daisy chain fashion [20]. Thus, all of the data being produced by the last (50th PE) at each clock cycle of a single partition's operation must be collected and stored. This stored data will then be looped back and fed as contiguous input to the 1st PE when the next partition is ramped up, thereby creating the required illusion that 200 PEs are actually linked together (Fig. 5).

During the systolic-array based operation of the PEs, there are seven pieces of intermediate data that flow from one pipeline-stage to the next. The abstract Load/Store FIFO Blocks (Fig. 5) are implemented as 7 banks of 32-bit wide FIFOs (Fig. 6), with each bank capturing 1 of the 7 values of output from the terminal-PE. This flow of intermediate data generated by the DIALIGN accelerator is outputted or inputted at a significantly high throughput rate of 1896 MB/s. There are seven native accelerator ports in each direction that push and receive this intermediate data to and from external storage, with each port being 4-bytes wide. The accelerator operates at 67.689 MHz and at 28 bytes every clock cycle (assuming no stalls by the Ethernet controller) this produces the one-way throughput of 1896 MB/s.

The fourteen accelerator ports mentioned above are what feed the Store FIFO Block and receive data from the Load FIFO Block in Fig. 6. At the very bottom of Fig. 6, data heading downstream is received by the SATA controller ports. The native port-width of our particular SATA controller [21] is 16-bits. The controller clock runs at a frequency of 150 MHz such that its 2-byte input port then offers a bandwidth of 300 MB/s, which is the maximum bandwidth of SATA 2 devices. Observed SATA throughput, however, is a combination of the controller's implementation and the particular drive that it is connected to. With the Intel SSD used in our system, we experimentally observed maximum sequential write and read throughput rates of 66.324 and 272.964 MB/s respectively.

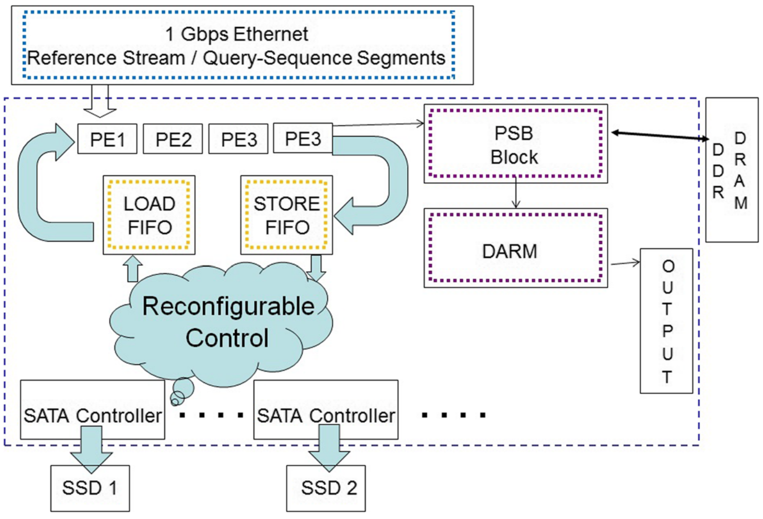

Fig. 4 Ethernet and SATA IO channels to the XUP 5 Development Board

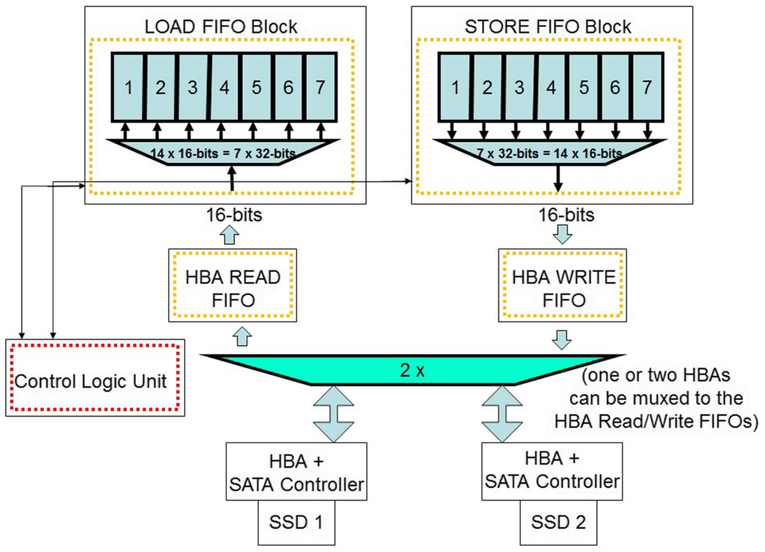

Fig. 5 FPGA based Storage Interfaces for Query-Sequence Partitioning

The accelerator to IO controller interface, for this channel, consists of all the data path and control units depicted in Fig. 6. It serves all three of the functions within our definition of an interface: data-width conversion, clock domain crossing, and flow control. Figure 6 is an intentionally simplified block diagram of this interface that hides the details of implementation. In this section, such details will be elaborated so that they may be fairly compared to alternative software interfaces in future sections.

Within the Store FIFO Block of Fig. 6, below the 7 FIFO segments, a multiplexer is depicted to convert the 224-bit (7x32-bit) data path from the accelerator to only 16-bits. This is an oversimplification, and in reality, a mux-control-FSM in conjunction with seven other 2-to-1 muxes are used to accomplish this task. The FSM also helps to perform another mandatory task of framing the data for SATA protocol writes. The SATA controller writes to disk in single sector segments of 512-bytes per sector. The 28-bytes of data produced by the accelerator at each cycle cannot be evenly packed into 512-byte frames. Every 18 cycles of accelerator operation, will only produce 504-bytes of data that must be packed into a 512-byte frame. The FSM packages the payload data with 4-bytes of Start of Frame and End of Frame encoding, such that a single frame can be successfully detected, and stripped of its payload data, during the return backward path of data flow.

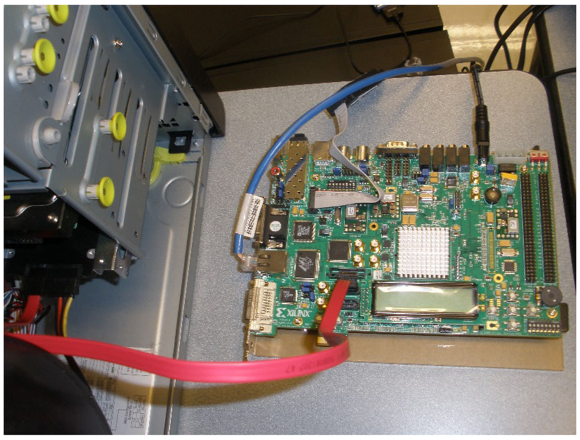

Fig. 6 Data flow to and from the Load/Store FIFOs to SSDs (Not all connections to the CLU are depicted)

Stuffing the remaining 8 bytes of a 512-byte sector with the first 8 bytes from another cycle of accelerator operation would create the additional overhead of realigning FIFO-bank lines when the sectors are eventually read back. Moreover, if a FIFO-bank line crosses two sectors, and the controller has not returned the 2nd sector yet, this would cause some individual FIFOs within the bank to become empty while others are not, and thus further exacerbate the synchronization issues that would have to be resolved.

Lastly, the mentioned mux-control-FSM is designed to operate at the faster SATA controller clock frequency of 150 MHz, such that 16-bits of output can be produced at the maximum bandwidth of 300 MB/s. Note that the 224-bits to 16-bits ratio, means that an accelerator running at even 1/14th of the SATA frequency is capable of providing enough data to sustain a throughput of 300 MB/s by this FSM.

To briefly recap the primary hardware components that went into this interface, there were two FSMs, one at the front end to perform data-width conversion and data framing, and a back end FSM to issue SATA commands and handle error recovery. Alongside these primary FSMs, non-trivial arrangements of FIFOs and muxes were utilized to assist with data packing and flow control. The most important aspect of these hardware considerations is that all of these components were designed with the architectural requirement of running at the native SATA controller frequency of 150 MHz, such that data bandwidth was never compromised due to interfacing. However, the much slower SATA write throughput limitation imposed by the actual physical SSD negates the over-provisioning of the interface logic. This then allows for other simpler (and slower) IO interfacing options to be considered while still achieving a comparable system level runtime.

The question that we are soon to resolve is that given that the SATA controller's actual end write throughput is much lower than its input-port bandwidth capacity, can we relax the aforementioned frequency requirements of the interface for a simpler architectural solution.

DIALIGN implemented with software IO interfaces

Adapting to software controlled IO interfaces, by definition requires at least one embedded processor to execute the software control. As such, the IO controllers in the system must also support communication over whatever bus standard that is supported by the chosen processor.

The development board that was used in the hardware interface experiments is held as a constant, and used within this section as well. As such, the Xilinx MicroBlaze [22] softcore processor is the embedded processor around which our software solution will be explored. The MicroBlaze ISA supports the following three bus protocols for external communication. They are

a) Local Memory Bus
(LMB, low latency direct access to on-chip memory block-ram modules),
b) Processor Local Bus
(PLB, a shared bus intended for processor peripherals and memory-mapped IO),
c) Xilinx Cache-Link
(XCL, similar to the PLB, however, it supports burst transactions and is exclusive to the processor's instruction and data caches).

The two IO controllers used for the DIALIGN application, with fully RTL interfaces, in "DIALIGN implemented with fully RTL based IO interfaces" section were the SIRC based Ethernet controller [23] and the corresponding Groundhog SATA controller [21]. Because the Ethernet controller makes use of only block-rams at its top level for data intake and delivery, it can be adapted to connect with the MicroBlaze LMB interface. However, the Groundhog SATA controller, which although allows for a lot of design flexibility to interface directly with its SATA command layer via custom hardware, is not capable of being connected directly to a PLB interface without significant design additions. As such, the SATA2 controller [24], another open source SATA controller that is more amenable to embedded solutions is adopted in this section. Both controllers are comparable in their SATA 4 KiB write tests, with the SATA2 controller performing slightly worse using comparable SSDs. Therefore, the alternative software solution being explored in this section is not unfairly biased in its favour via a better SATA controller.

The SATA2 controller makes available the following two bus protocols for communication with the core—PLB and Native Port Interface (NPI). The PLB has been previously introduced, however, the NPI is another protocol exclusive to Xilinx's DRAM controller that is being introduced here. For embedded systems' use, Xilinx makes available their Multi-Port Memory Controller (MPMC). Furthermore, the MPMC supports various interfaces on its ports, with two of them being PLB and NPI interfaces in this case. An NPI port, as its name implies, is meant to provide the closest matching signals and data-width to that of the physical DRAM being controlled. For example, the off-chip DRAM modules on the XUP5 development board have a 64-bit wide combined data path, and thus the NPI, in this case, would also accommodate a 64-bit port. This is in contrast to the PLB port, which would be set to a 32-bit port as dictated by the MicroBlaze ISA. Therefore, an MPMC instantiated with both PLB and NPI ports allows the MicroBlaze processor access to DRAM, and also allows a Direct Memory Access (DMA) module or peripheral access to native memory widths respectively (Fig. 7).

With the addition of a complementary NPI based DMA, the SATA2 core has direct access to DRAM with little MicroBlaze intervention; thereby not having the throughput to the SSD restricted by PLB limitations as well.

HW/SW partitioned architecture

Given the feasibility of interconnecting the Ethernet and SATA controllers utilized by the DIALIGN algorithm to standard bus protocols, the existing EDA tools offered by FPGA vendors for embedded systems design can be leveraged to manage data flow in software, while allowing for the computationally intensive accelerator portions to remain in hardware.

Newer interfaces such as AXI [25] exist to interconnect the programmable logic data-path to an embedded processor; however, due to the limitations on IP for the Virtex 5 development board, our architecture utilizes Fast Simplex Link (FSL) [26] based communication. Our

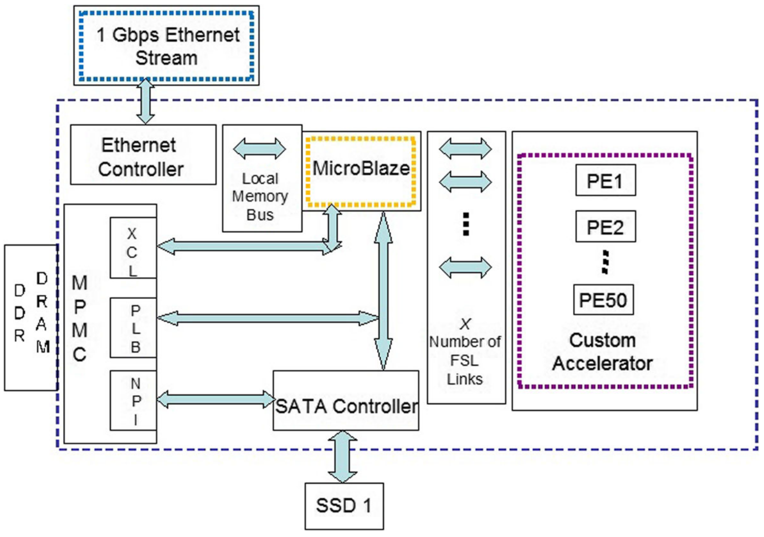

Fig. 7 Embedded MicroBlaze access to the SATA2 Core and DRAM [24]

architecture combines a varying amount of FSL channels, along with the Ethernet and SATA controllers, to form a flexible and easily controllable data-path for hardware acceleration (Fig. 8).

The number of FSL links used to connect the configurable accelerator to the embedded system depends on the needs of the application being accelerated. Each link, as mandated by the FSL interface, has configurable FIFOs built in. Although a minimal dual cycle latency for data is possible via the FSL links, the total bandwidth between the accelerator and the processor will be limited by the lower clock speed of the MicroBlaze processor. Nonetheless, the potential for ease of automated configuration is high due to the fact that the FSL lanes can be scaled according to the number of input and output ports that are required by the accelerator, and by the fact that the data-width of each port can be set accordingly as well.

The low aggregate bandwidth between the accelerator and the processor that comes with the use of FSL lanes does not introduce a new throughput bottleneck into the system for all applications. If the application already has an IO channel throughput bottleneck that is lower than the FSL channel rate, the ease of automation gained by using the FSL lanes will have no adverse effect on the overall system throughput.

Soft-processor data management

As depicted in Fig. 8 all input to the alignment accelerator is received over the Ethernet channel. During accelerator operation, one DNA character (a single byte) of the

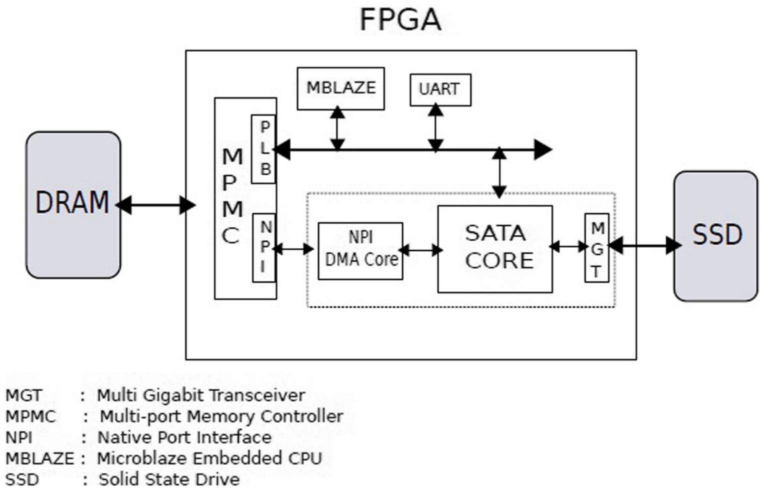
FPGA
DRAM
MPMC
PLB
NPI
MBLAZE
UART
NPI DMA Core
SATA CORE
MGT
SSD
MGT : Multi Gigabit Transceiver
MPMC : Multi-port Memory Controller
NPI : Native Port Interface
MBLAZE : Microblaze Embedded CPU
SSD : Solid State Drive

Fig. 8 Hardware/Software Partitioned Infrastructure

reference sequence is required per cycle. The accelerator runs at a maximum frequency of 67.689 MHz and thus requires 68 MB/s of input bandwidth to sustain operation at full throughput. Through experimentation, we have verified that the SIRC Ethernet controller can provide a stream bandwidth rate between 58.88 MB/s and 60.35 MB/s. However, this does not cause the SIRC stream to become the bottleneck of the system. The greater bottleneck is discussed below.

The SATA 2 (3 Gbps line rate) protocol allows for a maximum theoretical bandwidth of 300 MB/s. Real SSDs, however, offer much lower rates, especially on write bandwidth. The SSD used in our experiments revealed maximum sequential write rates of 66 MB/s. The DIALIGN accelerator produces seven 32-bit words per cycle of intermediate data at a clock rate of 67.689 MHz, or rather 1895 MB/s of write data.

The design goal of this section is to capture the 1895 MB/s of data produced by the accelerator logic, bring it into the soft-processor based embedded system domain, and feed it to the SSD controller. As long as this process can be successfully implemented without dropping below the physical SSD data write rate of 66 MB/s, the use of an embedded system for data management does not degrade system performance relative to custom RTL based data management. Given that a conventional MicroBlaze soft-core processor can be run at a frequency of 125 MHz and that it operates on 32-bit word operands, this provides for an upper bandwidth limit of 500 MB/s to work within. Realistic bus transfer rates along with actual DRAM write/read rates in implementation will be considered to determine the degradation to this 500 MB/s upper limit.

The first stage of data transfer is achieved via seven FSL links that give the MicroBlaze core access to each of the seven accelerator output operands using distinct FSL channel ID numbers in software. The Xilinx FSL link interface includes built in FIFOs on each channel. All seven FIFOs on the links are all set to the same depth. The accelerator concurrently writes to all FIFOs on each clock cycle of its operating frequency, however, the processor will read from the FIFOs in round robin fashion. Accelerator operation can be paused in response to any almost-full control signals from the FSL FIFOs.

The XUP5 development board used in our design uses DDR2 DRAM and is controlled by the Multi-Port Memory Controller (MPMC) IP [27]. Although the PLB [28] interface supports burst-mode data transfers to the MPMC, MicroBlaze does not support burst-transfers over the PLB. As a workaround, a data-cache can be incorporated into the MicroBlaze design, and then the Xilinx Cache Link (XCL) [22] can be utilized for burst-mode data transfers to DRAM. This is the optimization we have employed to achieve native 64-bit wide data writes into DRAM via write bursts issued by the MicroBlaze data cache. We also ensured that the "Use Cache Links for all Memory Accesses" parameter was set so that data is pushed onto the XCL even if the data-cache was disabled.

Software functions to read/write to the FSL channels are provided within Xilinx's SDK environment and 32-bit words of data can be accessed within single cycle speeds. Thus the processor running at 125 MHz can consume data from the FSL links at a bandwidth of 500 MB/s. From there, the data rate at which the data-cache-bursts write to the MPMC will be the first bandwidth funnel. The upper bound on throughput from the MicroBlaze to the MPMC is affected by several factors. From the source (MicroBlaze) side, data may not always be injected at the upper limit of 500 MB/s due to the fact that some MicroBlaze overhead cycles will be used to gather the data from multiple FSL channels. On the sink (MPMC) side, write throughput will depend on any other concurrent reads or writes occurring to the same physical DRAM (via the SATA controller) and how effectively the arbitration policy of the MPMC is executed. These complexities prohibit derivations of any practical analytical methods to infer reasonable throughput levels between the MicroBlaze and the MPMC.

After the data has been streamed by the MicroBlaze from the FSL channels into DRAM, the SATA2 core can begin to consume it from there. The modified SATA2 core [24] that we employ uses the Native Port Interface (NPI) [27] of the MPMC for its own internal DMA based transfers. The DMA setup and start commands are still issued in software by the MicroBlaze communicating with the core over the PLB. The SATA2 core's internal NPI based DMA engine can obtain data from the MPMC at a throughput of 240 MB/s. This rate has been experimentally verified, and also drops down to no less than 200 MB/s when the MicroBlaze is simultaneously issuing data writes to the MPMC via its XCL port. In this scenario, the MPMC internally performs round-robin arbitration between the XLC writes and the NPI reads. Nonetheless, even a 200 MB/s rate of data delivery to the SATA2 controller far exceeds the 66 MB/s write rate of the actual SSD. Thus these empirical results, which could not have been obtained in a practical analytical manner, establish that the system throughput is not degraded by this software solution.

In creating this embedded system based data-path, all of the custom logic depicted in Fig. 6 for the hardware interface in "DIALIGN implemented with fully RTL based IO interfaces" section is no longer necessary. FIFOs to collect the data from an accelerator at variably wide data-widths (up to sixteen 32-bit wide words) are automatically available within the FSL IP, transfer of that data into DRAM is then controlled in software, and finally the repacking of data words for the appropriate sector write frames into SSDs can also be handled in software.

Furthermore, should the underlying physical SATA-core be changed to support future SATA-3 generation devices or merely wider data-widths of the transceivers used even within SATA-2 devices, these changes can be supported by software. For example, Virtex 5 transceivers (GTP tiles) utilize 16-bit SERDES (Serializer-Deserializer) circuits for their SATA physical-layer links, however, Virtex 6 transceivers (GTX tiles) were upgraded to 32-bit SERDES transceivers. These changes propagate all the way up to the command layer of the SATA HBA controller and require the buffers or FIFOs that supply data to the controller to also be converted to matching 32-bit words. If our data-path was solely an RTL design ("DIALIGN implemented with fully RTL based IO interfaces" section), then the muxes, FIFOs, and more importantly the FSMs would all have to be accordingly modified to pack wider words. In the presently described embedded solution, not only can the 32-bit upgrade be supported entirely in software, but we can also maintain full backward compatibility with any legacy 16-bit SATA controllers, through 32-bit input-word conversions into half-word data writes to DRAM in software.

Resource utilization across interface solutions

In this section, the hardware utilization associated with the solely hardware, and the HW/SW partitioned interface solutions will be compared and discussed.

Table 1 lists the amounts and percentages of resources consumed on the Virtex 5, XUP5 development board [19], for both interface options. It can be observed that there is roughly a 20% overhead increase in LUT and DFF logic that is consumed when moving to a HW/SW partitioned solution. The majority of this extra logic can be attributed to not only the MicroBlaze softcore itself but also to the MPMC, its supporting busses, the FSL channels, and the extra DMA unit within the SATA2 controller itself. As expected for this application, the actual run time performance of the DIALIGN accelerator is equal between the custom HW and HW/SW interfaces, and thus is not reported in the table for comparison.

What is gained by this increase in hardware is the significant reduction in design time and debugging efforts. The custom hardware interfaces are very resource efficient, however, they can require 6 months of design to architect, implement, and verify. Furthermore, they cannot easily be reused when an IO controller update is required.

Video application study

Motion estimation implemented with fully RTL based IO interfaces

As a motivator for the case study described in this section the reader is urged to consider the system level challenges of a commercial application such as Netflix [29]. Although an application such as Netflix in its infancy may start out with a fully customizable computing cluster environment, as its customer base and data requirements expand, a full warehouse data-centre infrastructure is often inevitable [30]. In this scenario, customized acceleration for the encoding required for all standards of input video sources, and the output video stream resolutions produced, will be challenging. Therefore, cloud-based solutions are often sought to handle the scale and possible automation of the encoding workloads [31]. It is precisely this sort of environment that our solution aims to target. In this environment, the flexibility to scale the accelerators that handle the compute bound portions of the application's code can be leveraged to manage IO constraints *if*, and only if, the IO interface architecture is appropriately designed and matched.

Acceleration of Motion Estimation (ME), for the H.264 video encoding standard [32], on FPGAs uses only a memory controller for off-chip DRAM access as its single channel for IO data. We examine ME in this section because, from an IO perspective, it is fundamentally different from the previously covered DIALIGN application in that neither its physical IO channel device (DDR2 DRAM) nor its IO controller (the Multi-Port-Memory-Controller) are bottlenecks to accelerator performance. Nonetheless, ME is still a relatively IO dependent application. However, what will be demonstrated in this section is that the requirements of this application, and how these requirements are exploited, combine to prevent IO throughput from becoming an issue. We show that this form of application requirement exploitation can be achieved through the careful customization of its interface to the memory controller.

Table 1 DIALIGN alignment accelerator with custom HW interfaces vs. HW/SW IO interfaces

	Area[a]							
	LUTs		DFFs		LUTs %	DFFs %	Accelerator	MicroBlaze
	#(K)	%	#(K)	%	Increase	Increase	frequency (MHz)	frequency (MHz)
HW Interface	51.0	73.8	36.9	53.4			67.7	N/A
HW/SW Interface with Microblaze	64.5	93.3	52.0	75.2	19.5	21.8	67.7	125

[a]Xilinx's Virtex 5 devices use 4 DFFs & 4 6-input LUTs per Slice

External DRAM throughput requirements

A memory hierarchy (Fig. 9) is used to move portions of a large data set stored at the non-volatile level (i.e. a video file), into a medium data set stored in DRAM (video frames), and then finally into small data sets of on-chip buffering (sub-frame blocks) to support acceleration. It will soon be detailed that a throughput of only ˜620 MB/s from off-chip DRAM to on-chip block rams suffices for this application. Thus even using a modest DDR2 memory controller will not become a bottleneck based on the required 620 MB/s of read throughput (to supply data to the accelerator) and the required 125 MB/s of write throughput (to load video data coming in from the SSD at a speed of at most 60 fps).

A single pixel is often encoded as a single byte. Thus a single 1920 × 1088 High Definition (HD) frame occupies 2.09 MB of memory. Since on-chip FPGA memory is commonly limited to be between 2 to 6 MB, video frames must be encoded in partial segments sequentially. The 1920×1088 frame can be encoded as eight equal segments of 960×272 Sub-Frame (SF) portions at a time. The 960×272 SF allows for a more manageable 261.1 KB of pixels to be encoded within on-chip memory at a time.

The VBMSE algorithm [32, 33] requires the video frame that is to be encoded (the present frame) to be compared to at least four other (reference) frames, before conclusive calculations on its corresponding motion vectors are reached. Therefore in addition to the 1/8th portion of the present frame (PSF), four other such portions of reference frames (RSFs) must be held within on-chip memory as well (Fig. 10), which brings the total memory space thus far to 1.3 MB. Here, we introduce the design concept of double buffering—the first buffer is initially loaded with data to be processed, processing then commences, simultaneously the second buffer is loaded with the next set of sub-frame data, thereby overlapping memory access latency with processing time. This then brings the total required on-chip memory allocation to 2.6 MB for double buffering, which even modest FPGA devices can support.

FPGA devices with larger amounts of on-chip memory can opt to double buffer larger sub-frame portions, but we wish to note that doing so only reduces the required memory-controller throughput rate. This is due to the fact that for VBSME an increase in data to be processed is not linearly proportional to processing times. Therefore, double-buffering larger sub-frame segments allows for the acceleration to be more compute-bound than IO bound, and thus lowers the required external memory throughput.

The basic building blocks for which motion vectors are produced as outputs in motion estimation are 16×16 blocks of pixels, commonly referred to as a macro-block. Within the 960 × 272 sub-frame portion size that we are using in this work there will be 1020 macro-blocks that need to be encoded. For reasons that will be clarified in the next sections to come, the number of required clock cycles to encode a single macro-block against a single reference frame is 99 cycles, when the accelerator is scaled to its highest level of acceleration. We intentionally use the highest level of scaling here in these calculations to demonstrate the upper bound on the required external memory throughput. Next, recall that VBSME requires the present frame to be compared against four other reference frames when calculating the motion vectors for its macro-blocks. Therefore, the 99 cycles become 396 cycles in total per macro-block. We can now state the entire processing time of a sub-frame portion in cycles to be equal to 403 920 (1020 macro-blocks x 396 cycles per macro-block).

As detailed earlier in this section, the total data required to process a sub-frame against its four reference frames is 1.3 MB (1.3056 MB exactly, Fig. 10). The maximum

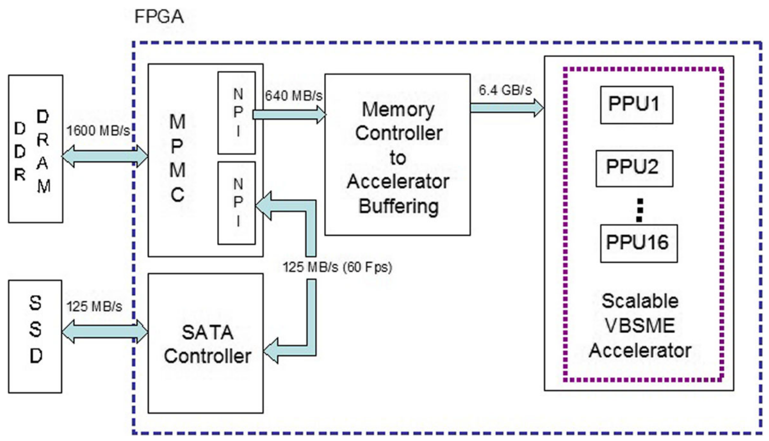

Fig. 9 VBSME Accelerator Memory Hierarchy at 16-PPUs Data Rate

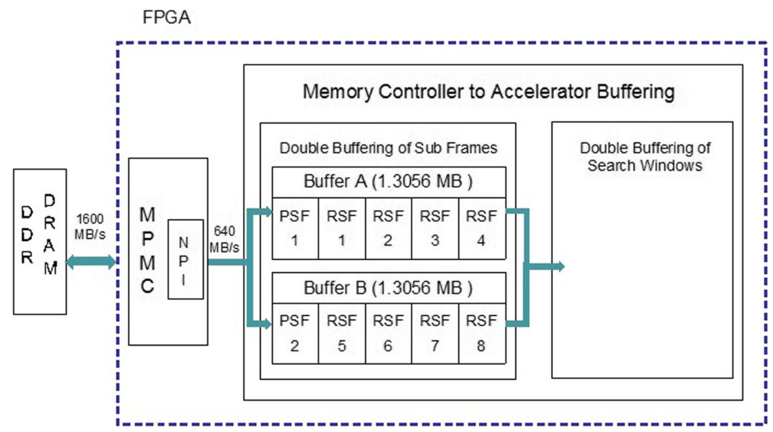

Fig. 10 Memory Controller to Accelerator Buffering

frequency of the implemented VBSME accelerator on our chosen FPGA device is 200 MHz. Thus the final required external memory throughput with the use of double buffering can now be derived as 646.5 MB/s (1.3056 MB / 403 920 cycles * 200 MHz).

VBSME accelerator operation

For the purposes of this article, it suffices to view the VBSME accelerator as a scalable black-box accelerator. The scalability of our black-box accelerator is measured in terms of the number of Pixel Processing Units (PPUs) that it is instantiated (synthesized) with. In our system implementation, the offered set of scaling levels is 1, 2, 4, 8, or 16 PPUs. Looking into this black-box slightly for the purpose of understanding its IO requirements, each PPU can be viewed as a single 16×16 two-dimensional array of Processing Elements (PEs) (Fig. 11). A single PPU requires two separate memory busses, each being 16-bytes wide. The necessity of this dual-bus architecture stems from the need to avoid any pipeline bubbles within the hardware architecture for VBSME [34]. These dual-buses are each independently connected to two physically separate Memory Partitions A and B that contain two vertically segregated logical partitions of the search-window memory space (Fig. 12).

Although each PPU requires, in total, an input path of 64-bytes across its two buses, fortunately, this relationship is not held when scaling the VBSME architecture to use multiple PPUs in parallel. The relationship between the number of bytes required when scaling upwards to n number of PPUs is $15 + n$ bytes on each bus [35]. Thus an instance of a VBSME accelerator scaled to 16-PPUs would only require 31-bytes on each bus. This derives from the fact that each additional PPU always shares 15-pixels in common with its preceding PPU (Fig. 13).

Search-window buffer scaling

Due to the fact that the input bus-width requirements of the VBSME accelerator vary when it scales, as discussed previously, the search-window buffers must also be scaled

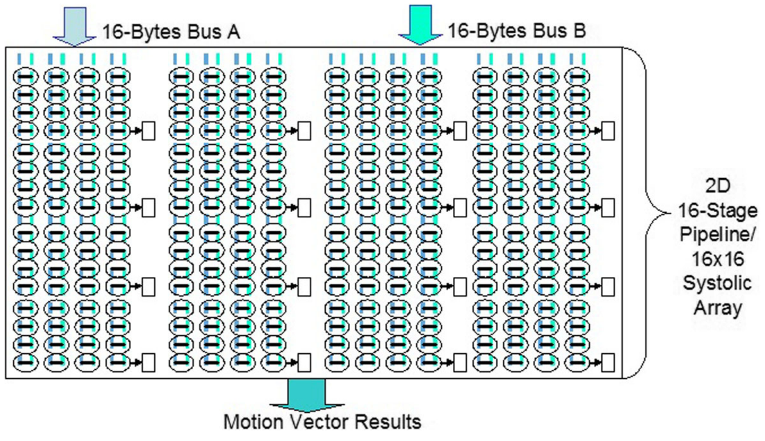

Fig. 11 PPU Black Box View, with 2 Input Buses [39]

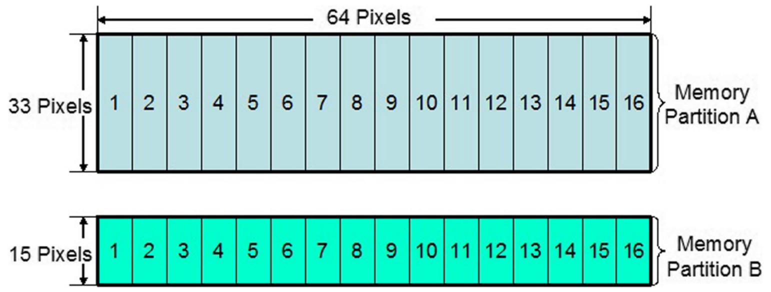

Fig. 12 Logical to Physical Segregation of Memory Partitions A and B [39]

(synthesized) in varying patterns of data organization to accommodate the scaling.

The VBSME accelerator processes a search window by accessing its rows (top to bottom) one cycle at a time. Accesses to the last 15 rows of the search window, coming from Memory Partition B (Fig. 12), are always overlapped with the 33 rows being accessed from Memory Partition A [34]. Thus 33 cycles are all that is required to finish a vertical sweep of the search-window. However, not all of the 64 columns of the pixels within a 64×48 search-window are read per each row access.

The number of columns ($15 + n$) that are read per row depends on the number of PPUs that the accelerator is scaled to. Once a vertical sweep of these columns is performed, a horizontal shift to the right is performed to start the next vertical sweep. The granularity of this horizontal-shift to the right is exactly equal to the number of PPUs being employed. Therefore, the number of vertical sweeps that are required to sweep a search window of width w, in a top-down left-to-right manner is given by $\lceil (w - (15 + n))/n \rceil$.

Thus in our implementation, the VBSME accelerator when scaled to 16-PPUs will require a total of 3 vertical sweeps to process the search window ($\lceil (64 - (15 + 16))/16 \rceil$). These 3 vertical sweeps of 33 cycles each, result in a total of 99 cycles needed to completely process the search window. As another example, if the VBSME accelerator is scaled down to use only a single PPU the number of required vertical sweeps is 48 ($\lceil (64 - (15 + 1))/1 \rceil$), resulting in a total of 1584 cycles required to complete the search.

Figures 14 and 15 represent the logical columns (Lx_A) of the horizontal shifts that are required when the VBSME accelerator is scaled to 16-PPUs and 8-PPUs respectively. In both cases, the widths of each Lx_A partition is equal to the number of PPUs being used. The number of vertical sweeps, and thus the total processing time, is also labeled in the figures for each case according to the previously explained formula.

In order to support the access of these logical partitions, which are in turn based on the granularity of the PPUs being scaled, different mappings of these logical partitions into a varying number of physical block-ram banks must be implemented. The number of block-ram banks required for this logical to physical translation of search window data is determined by $\lceil (15 + n)/n \rceil$ [35].

The logical to physical implementation of search window buffering for the 16-PPUs and 8-PPUs cases of scaling are represented in Figs. 16 and 17 respectively. As shown in the figures, the required hardware includes both block-rams and multiplexers to implement the required

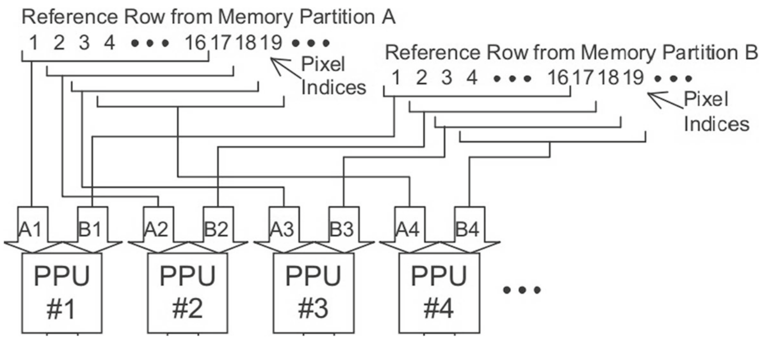

Fig. 13 Input Bus Pixels Shared Amongst PPUs [35]

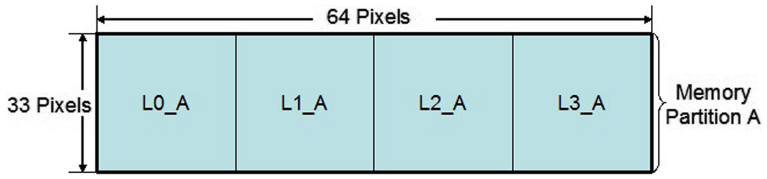

Fig. 14 Three Vertical Sweeps (99 cycles) for a 16-PPUs Case

data path flexibility during the various clock cycles of search window processing. An important component not depicted in the figures is the Control Logic Unit (CLU). These CLUs are implemented as Finite State Machines (FSMs) for each level of accelerator scaling, and perform the duties of generating the appropriate block-ram addresses and mux selection control signals.

A custom RTL implementation consisting of the above-mentioned block-rams, muxes, and FSMs must exist within the "Double Buffering of Search Windows" functional block of Fig. 10, within each of the search windows shown. This requires a significant amount of RTL development and debugging that must be performed for each level of VBSME accelerator scaling that is to be implemented.

In the following section, an embedded soft-core approach to memory organization that will obviate the need to redesign these hardware components for each level of scaling is presented.

Motion Estimation implemented with Software IO interfaces

Before any video compression algorithm can be run it is, of course, necessary for the raw video frames to be available within DRAM. From the details of the previous section, it is clear as to how the FPGA device itself within our proposed embedded system can directly access raw video data from a secondary storage device such as SATA SSDs. Once video data is held within DRAM, the 256 MB or more of DRAM capacity is enough to sustain the buffering of frames in a manner that does not render the SATA-core to DRAM bandwidth to be the bottleneck [35].

To support the VBSME algorithm introduced in "External DRAM throughput requirements" section in terms of physical design within the embedded system, only two main parameters need to be resolved. The first one being how many FSL channels need to be instantiated per each level of accelerator scaling, and secondly what the data-width of each channel needs to be. If this is done correctly, the manner in which the memory subsystem is implemented will be transparent to the accelerator logic. Apart from answering these two parameter questions, the data pattern by which pixels should be loaded into the FSL FIFOs is another important algorithmic challenge.

The number of FSL channels to be instantiated can be resolved via the following formula for VBSME scaling.

$x = 2 \times \lceil (15 + n)/4 \rceil$

Here x refers to the number of FSL channels that need to be instantiated, and n refers to the number of PPUs that are implemented according to the level of accelerator scaling. $15 + n$, are the number of pixels that need to be forwarded to the accelerator unit respective of its n units of scaling ("Search-window buffer scaling" section). Since each pixel is 8-bits in representation, a single 32-bit FSL channel can forward 4 pixels of information. The multiple of 2 derives from the accelerator requirement of having dual memory partitions ("VBSME accelerator operation" section).

If the number of pixels required per clock cycle by the accelerator (the numerator in the above equation) is exactly divisible by 4, then each FSL channel will be set to the maximum width of 4-bytes (32-bit words). However, if the quotient leaves a remainder when dividing by four then a modulus function can be applied to determine the data width of the last FSL channel. Since the FSL widths are configurable, this customization can be achieved.

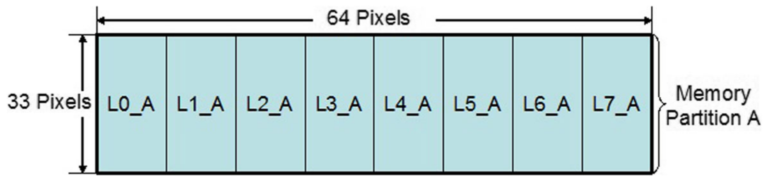

Fig. 15 Six Vertical Sweeps (198 cycles) for an 8-PPUs Case

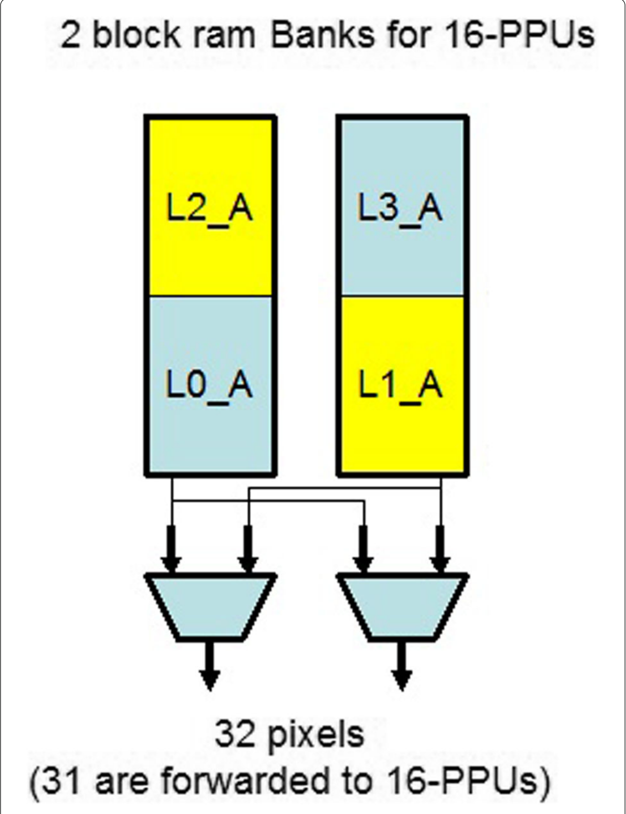

Fig. 16 Logical to Physical Memory Mapping for 16-PPUs

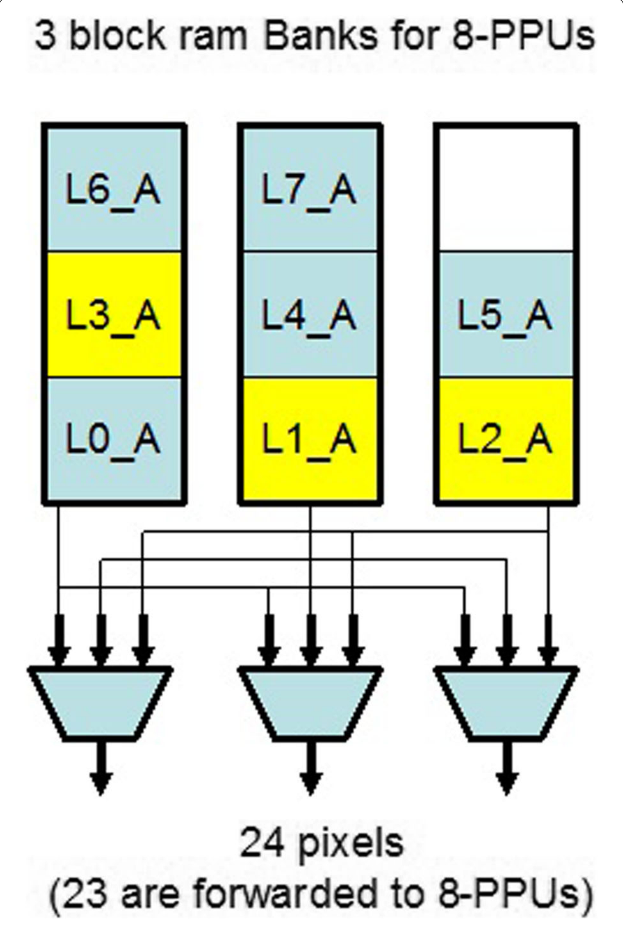

Fig. 17 Logical to Physical Memory Mapping for 8-PPUs

In "Search-window buffer scaling" section block-rams and muxes were introduced as the components that constituted the underlying memory substructure. In this section, we will detail the use of the built in FIFOs in the FSL channels as a transparent substitute for the custom RTL-designed components. In a pure block-rams based buffering system, memory addressing can be used to repetitively access data. However, if the block-rams are instantiated as FIFOs, once a specific memory-data read occurs, it can not be accessed again without it being re-pushed into the FIFO queue again. Thus, although a memory interface based on FIFOs is simpler to integrate and control, the continual loading of data into these FIFOs will follow a more complex pattern. This is the trade-off that is to be made, and this is also where we will leverage the flexibility of soft-core based transfers from DRAM to support the complex patterns of FIFO loading.

Once the number of FSL channels has been set according to the previous formula, columns of pixel data from the search window of interest must be initially transferred into these channels in a left to right manner. After this initial filling of the FIFOs within the FSL channels, the sweeps across the search window by the accelerator ("Search-window buffer scaling" section) are accomplished by horizontal pixel-column shifts in the search window that equate to the number of PPUs the accelerator is scaled to. In Figs. 14 and 15, this shift-width can be logically viewed as the partition width (e.g. block L0_A, L1_A, etc.). In the case of the accelerator scaled to 16 PPUs their partition widths will be 16-pixels, in the case of 8 PPUs, it will be 8-pixels and so forth.

When using the FSLs for pixel-column shifts, since their channel width is set to the maximum of being 4-pixels wide (to optimize data transfers), shifts that occur in multiples of 4 can be handled by relatively straight forward 32-bit memory word reads from DRAM. Such is the case for the accelerator scaling levels of 16, 8, and 4 PPUS. However, as the accelerator scaling falls to 2-PPUs or even a single PPU, the required FSL granularity of shifting becomes *1/2* FSL width and *1/4* FSL width respectively. Since the Microblaze ISA is byte addressable, word accesses from DRAM can be packed/re-ordered such that a finer granularity of shifting pixels is supported. At the micro-architectural level, Microblaze may make use of barrel shifting within its ALU to get the desired byte from a 32-bit data bus read. Therefore the Microblaze core was synthesized with its additional barrel-shift logic parameter enabled.

Four-pixel (32-bit word) reads from memory for a 2-PPUs accelerator are realigned before being pushed into the FSL channel FIFOs at various time steps, such that they hold the appropriately shifted pixel columns within them (Fig. 18).

For simplicity's sake Fig. 18 above shows only the logical to physical mapping of pixel-rows 1 to 33 in a Search Window (i.e. Physical Memory partition A in Figs. 14 and 16), the reader is urged to keep in mind that another separate set of 5 FSL channels are implemented for the pixel-rows 34 to 48 of partition B. In Fig. 18, a single time step refers to 33 cycles being completed, or in other words one vertical sweep (Search-window buffer scaling section) of the Search Window being completed.

Experimental results

Before we present the experimental results of the measured bandwidths in our soft-core controlled system, we would like to address why experiments are necessary in the first place. One could argue that analytical analysis alone could suffice to determine the feasibility of our system, since the bandwidths and operational frequencies of our embedded system components are already published within their respective technical data sheets. This is true in that such an analysis would result in an upper theoretical bound on what level of system performance is possible. However, this estimate is likely to be overly optimistic and may downplay the realistic result of accelerator throughput degradation in actual implementation.

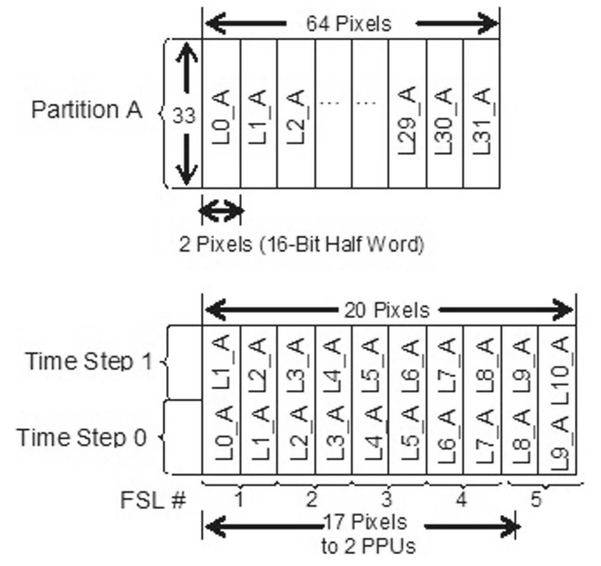

Fig. 18 Logical pixel columns mapped to 5 Physical FSL Channel FIFOs

Many of the embedded system components have non-deterministic latencies and throughput. These components included the MPMC (Multi-Port Memory Controller), the PLB and XCL buses to the MPMC, and even the Microblaze processor itself given the varying loads on the buses that it interacts with. All of these factors combine to make the throughput of an embedded system based data transfer mechanism highly variable in nature. This is even more pronounced when the accelerators that draw data from this system belong to varying application domains.

The Virtex 5 device used in the synthesis results for the VBSME accelerator is the XC5VLX330. This chip contains 51 840 Virtex 5 slices (each slice has four 6-input LUTS and FFs). Table 2 above presents the area and performance results of the VBSME accelerator when paired with custom RTL implementations of a memory substructure for each level of scaling; Table 3 then compares the area and performance of the accelerator paired with our soft-processor based system of memory data delivery.

Similar results for the DIALIGN DNA Alignment accelerator were previously listed in Table 1 (the table is reproduced below as Table 4 for reader convenience). Unlike the VBSME application, DIALIGN did not suffer any performance degradation in the adoption of a HW/SW MicroBlaze interface, and thus performance metrics are not listed in the table for comparison. The DIALIGN accelerator was implemented on the 5vlx110tff1136-1 Virtex 5 device, on the XUP 5 development board [19]. As seen in the table, moving to the easier development flow of a HW/SW solution does come at the cost of roughly a 20% increase resource utilization. This device is a relatively smaller chip with 17 280 Virtex 5 slices.

Total development hours for the embedded-system based design of data movement for both of these application accelerators was measured in weeks, and less than a month at a maximum including debug time. In comparison, custom RTL based interface design and memory management for these accelerators was closer to 6 months exclusive of debugging hours. Furthermore, the RTL designs cannot easily be reused when an IO controller update is required.

For the embedded-system the development time is quantifiable under two different steps. The first step, which is the actual configuration of the embedded system can take up to a week or two to design and test, to ensure that the FSLs are connected to the accelerator correctly. The 2nd step is the writing of the embedded software itself, to transfer data from the SSD to the accelerator. This resulted in roughly only 500 to 1000 lines of code, between the two applications, with the debug hours being greater than the code writing hours.

In contrast, the previous work on custom RTL interfaces [35] consisted of five different Verilog modules for

Table 2 VBSME accelerator area and performance results with a custom RTL-designed memory subsystem

# of PPUs	Area[a]				Performance		
	LUTs		DFFs		Target resolution	Freq	fps
	#(K)	%	#(K)	%		(MHz)	
1	8.71	4.20	3.42	1.65	640×480 (VGA)	200.6	28
2	18.5	8.92	5.49	2.65	800×608 (SVGA)	199.0	34
4	37.8	18.2	9.64	4.65	1024×768 (XVGA)	198.3	42
8	76.4	36.8	18.0	8.68	1920×1088 (HD Video)	198.3	31
16	154	74.3	34.6	16.7	1920×1088 (HD Video)	198.3	62

[a]Xilinx's Virtex 5 devices use 4 DFFs & 4 6-input LUTs per Slice

each level of memory subsystem scaling. Then, five other top-level Verilog modules were also required to accomplish the correct wiring of the scalable accelerator to the chosen memory subsystem module. Each of the Verilog memory subsystem modules contained sub-components previously described such as block rams, muxes, and the most time consuming of all—an FSM based subcomponent acting as the CLU. On average there were close to 600 lines of RTL across the five Verilog memory subsystem files.

For the DIALIGN accelerator use case example, the hardware interface to the SATA controller consisted of eight Verilog modules, each consisting of roughly 500 lines of RTL code. Since these lines of RTL code interact with a physical external device, often simulation alone is not enough to diagnose bugs during the debug process. Thus internal logic analyzers were required to pinpoint issues at run time. This level of hardware debugging requires multiple iterations to complete and suffers from high synthesis and place and route times between iterations. Visibility into simultaneous variables (i.e. registers) is also limited and confined to within usually 1024 clock-cycle sample windows at a time. Thus the verification process quickly becomes on the order of months before the system is functional as intended.

In contrast, the MicroBlaze HW/SW partitioned interface solutions can be implemented and verified within less than a month, inclusive of debugging hours, since debugging the interfaces in software does not require the FPGA device to be synthesized nor placed and routed for each debug iteration. It is also highly flexible towards future IO controller updates, while also retaining backwards compatibility in software. Above all of these factors that favour an embedded system approach—when custom RTL designed access to bare metal IO controllers in IaaS environments is not permitted at all, an embedded solution may be the only viable option to provide end customers with a flexible method to manage to their IO data.

The Microblaze processor chosen for the experiments was implemented using the minimum-area tool setting in XPS (Xilinx Platform Studio) and was clocked at 125 MHz. It also included barrel-shift logic as the only additional hardware component to its ALU. The VBSME accelerator using its stand-alone custom RTL memory substructure could be clocked at least at 198.3 MHz at 16-PPUs scaling, and at a maximum frequency of 200.6 MHz at the single PPU level of scaling. The required bandwidth to support its native operating frequency is far above what the FSL channels are capable of supporting (up to 500 MB/s). Thus the accelerator was underclocked down to 100 MHz, to run slower than the Microblaze and thus allowing for the Microblaze processor to keep up with data delivery demands. This results in a roughly 50% frame rate drop in performance across all the levels of scaling.

Table 3 VBSME accelerator area and performance results with MicroBlaze (125 MHz) based data delivery

# of PPUs	Area*						Performance	
	LUTs		DFFs		LUTs % Increase	DFFs % Increase	Freq (MHz)	fps
	#(K)	%	#(K)	%				
1	22.7	10.9	6.70	9.11	6.70	7.46	100	15
2	32.5	15.7	6.78	10.1	6.78	7.45	100	16
4	51.8	25.0	6.80	12.1	6.80	7.45	100	20
8	90.4	43.6	6.80	16.1	6.80	7.42	100	15
16	168	81.0	6.70	24.2	6.70	7.50	100	30

*Xilinx's Virtex 5 devices use 4 DFFs & 4 6-input LUTs per Slice

Table 4 DIALIGN alignment accelerator with custom HW interfaces vs. HW/SW IO interfaces

	Area[a]							
	LUTs		DFFs		LUTs % Increase	DFFs % Increase	Accelerator frequency (MHz)	MicroBlaze frequency (MHz)
	#(K)	%	#(K)	%				
HW interface	51.0	73.8	36.9	53.4			67.7	N/A
HW/SW interface with Microblaze	64.5	93.3	52.0	75.2	19.5	21.8	67.7	125

[a]Xilinx's Virtex 5 devices use 4 DFFs & 4 6-input LUTs per Slice

Discussion

For the VBSME accelerator system, its custom memory substructure was never the bottleneck of the system. The IO channel that its memory substructure relied on was DDR2 or greater device memory, which always supplied a surplus of memory bandwidth than what was required by the accelerator [35]. Thus, in this case, transitioning to a Microblaze soft-core data delivery method hurt its system performance by 50% or more (i.e. the Microblaze based FSL channels became a new IO bottleneck).

Despite the less than acceptable frame rates (< 26 fps) of Table 3 across all but one level of resolution scaling, an important piece of insight is still gained from the data within Tables 2 and 3 when analyzed together. In Table 2, the RTL designed interface, at the highest 1920×1088 HD target resolution, scaling to 16-PPUs results in a frame rate of 62 fps. The 62 fps rate, for most video applications, is beyond what is necessary (26̃30 fps). However, the same level of scaling, using a software IO interface in Table 3 still produces a useable frame rate of 30 fps. What this reveals is an important tradeoff—dialing up the computational performance of the accelerator can indeed compensate for limited IO throughput, and compensate well enough to achieve acceptable system performance.

At first, this revelation seems fairly counter-intuitive. One would expect that as the computational power of an accelerator is scaled up, this would then cause existing IO bottlenecks to be exacerbated further, resulting in unacceptable system performance. However, what we see in the frame rates of Table 3 is the opposite. As the accelerator performance is scaled higher from a single PPU up to 16 PPUs the frame rates increase correspondingly as well (except in the 8-PPUs row, which is an outlier in both Tables 2 and 3 that will be explained shortly). The reason for this lies in the architecture of the VBSME accelerator, and how it correlates a doubling in computational performance with only a single byte increase in the required input bandwidth. Based on this, one could reason that at some unknown level of accelerator scaling even a modest amount of IO throughput, such as that offered via a MicroBlaze solution, will produce the required frame rate for the system.

The 8-PPUs row is an outlier with respect to correlating increases in frame rates as the number of PPUs is increased. This is due to the fact that we are not measuring frame rate performance while holding the frame resolution as a constant. In other words the frame rate that we measure is against a targeted resolution per number of PPUs used. When video industry standards moved to 1080 HD video from previous VGA resolutions, the screen aspect ratio was widened. This widening of the frame means there are many more columns of macro-blocks that must be processed during VBSME. Thus the number of computations is significantly higher for HD video. As a result, even though we increase the number of PPUs to 8, the increase in HD resolution workload outweighs the extra PPUs and thus we do not see an increase in frame rate relative to the preceding row. However, when the HD resolution is held constant and the number of PPUs is further increased to 16, we see a significant doubling of the frame rate as expected.

For the DIALIGN DNA alignment accelerator, even with a custom memory/data delivery system, the slow SSD write rates of 66 MB/s were the IO bottleneck. In this case, transitioning to a Microblaze software controlled data infrastructure offered significant advantages in design time, testing, and future flexibility with no performance loss at all. However, the area overhead costs to include the Ethernet and SATA controllers as software driven microprocessor peripherals (versus custom FSM based controller logic) was just under 20% (for the LUTs used).

It should also be noted that the particular Virtex 5 device used did not contain a hardcore processor or memory controller, thus the MicroBlaze softcore and its required IP peripherals needed to consume programmable fabric resources in order to be implemented. This, however, is not the case with all of the modern FPGA devices on the market today. FPGA vendors have adopted the System-on-Chip model, where a hardened processor and memory-controller are well integrated with the programmable logic by default. Furthermore, volume of sales does not necessarily place such devices at higher price points either. In the case of such devices, the programmable logic overhead cost of 20% reported here would not exist, and the reduction in development time and effort could be gained without incurring the increased logic area penalty.

Present advances and shortcomings to hardware IO interfaces

On the path of making FPGA design more amenable to software developers, FPGA vendors have adopted software frameworks such as OpenCL [11, 36] for accelerator development. These frameworks, as a side effect of having raised the abstraction level in the hardware design process, have also made it easier to integrate certain IO devices.

As an example, the Ethernet channel that was used in this article's DIALIGN application was integrated using third party open-source SIRC cores that made the controller and a suitable interface for Ethernet access available. Today, FPGA vendors, such as Xilinx, have made Ethernet controllers directly available via their OpenCL development platforms (Fig. 19).

This step certainly solves the first problem of having at least a controller readily available to access the IO channel. However, the application requirements of how the IO data should be organized and buffered, or when back pressure should be asserted and released still falls into the user design space. These design choices falling into the user design space in itself is not a shortcoming. From an architectural point of view, requirements that will vary per application, should of course fall into that application user's design space. But the fact that no abstractions in software are available to manage the IO data stream within the application accelerator's design space is the real limitation.

The present means to emulate the same level of control that an embedded soft-core or hardened processor would have over the data within the OpenCL environment is as follows. The user must first create custom data structures such as arrays to contain the incoming IO data. For DIALIGN the query-sequence and reference-sequence Ethernet data would be segregated, for example, into two separate arrays. Then the control sequences by which the data stream is fed to these two arrays would also be written. Since all of this is still performed in software there is no drawback between our proposed solution and this framework as of yet.

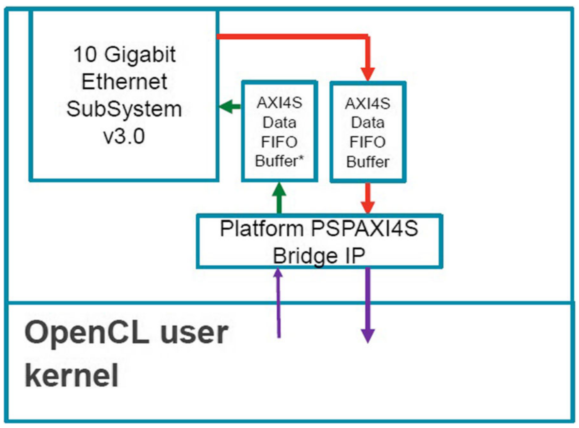

Fig. 19 FPGA vendor supplied Ethernet infrastructure

After all of the data organization and control has been programmed according to the available OpenCL APIs, the burden will completely fall on the High Level Synthesis (HLS) tool to implement the fine-grained data movement that is desired. And as discussed previously, having HLS efficiently manage data across varying clock-domains and interfaces is still an open-ended problem [7]. Due to this limitation, the fine-grained data pattern that was intended by the accelerator developer may not match what is actually implemented, thereby making throughput and latency worse off compared to manual implementation. This degradation, as shown in this article, may not affect the overall application performance. However, in scenarios where it does, an embedded core solution will offer better IO performance relative to HLS.

Using an embedded IO solution also offers other significant benefits to the FPGA vendor or the data centre that is offering FPGA nodes as Infrastructure as a Service (IaaS). The integration of future IO controllers yet to be released (such as SATA controllers), will not require their own individual FIFO and Bridge IP logic for integration to the accelerator design. Furthermore, these IO resources can be virtualized over multiple users' accelerators without comprising on security by allowing each user to access the IO controller directly.

The same arguments that we have made so far towards IO controllers also applies to the device DRAM memory controller as well. And in the context of device DRAM being shared by multiple accelerator users, the case for security over the bare metal controller is even stronger.

One of the more recent interfaces to FPGA vendor provided memory controllers is the AXI Stream interface [25]. This type of interface, allows custom accelerators and OpenCL based accelerators alike access to DRAM memory via accompanying AXI interconnect infrastructure (Fig. 20).

However, as can be seen in the architecture of Fig. 20, there is no inherent support for memory data buffering organization beyond relying on HLS or custom RTL development. As this article has shown, this organization and control of DRAM data onto on-chip buffers can significantly affect the application level performance.

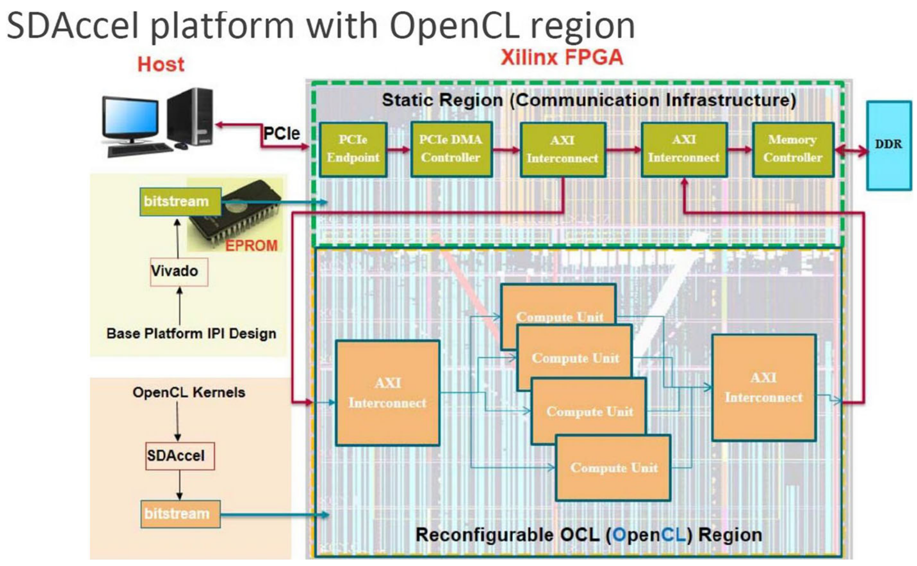

Fig. 20 FPGA vendor (Xilinx) supplied Memory-Controller infrastructure

Conclusion

This article has highlighted the need for standardized mechanisms to handle IO and data transfers to FPGA accelerators within data centers that offer their Infrastructure as a Service. This need arises from the fact that the end-user of such environments will never have full access to the underlying FPGA hardware. Thus data transfer methods that can be controlled in software while still offering performance levels equal to that of custom RTL interfaces are necessary.

To this end corporations such as Amazon (F1 Instances [37]) and Microsoft (Azure [38]) at present, for no extra cost, already offer FPGA "shells" as wrappers that developers can call to gain IO functionality on their cloud based FPGA instances. This shell is similar in concept to the static-region "template" described within this work. It is similar in that it too eases the pain of handling IO on FPGAs to the end-customer by providing pre-implemented IO controllers to standard interfaces such as PCIe, DDR memory, or even Ethernet in the case of Microsoft's Azure. However, this is where the similarity ends. What is proposed in this article goes another step beyond the abstraction of providing the IO controllers, we have presented a method by which the incoming data from the IO controllers can be effectively tailored to the performance needs of a particular accelerated algorithm. In contrast, Amazon and Microsoft defer the end-customer to the use of the FPGA vendor's HLS tool to handle the data after it crosses the IO controller boundary. And if HLS fails to provide optimal data partitioning and management of IO at the on-die buffering level for a unique application's data path, then the end-user is left to implement their own custom solution in RTL; if they have the skills and development time to do so.

We on the other hand have demonstrated an embedded system solution for data transfer, and more importantly, have shown that for a class of applications that have physical IO devices as their system level bottleneck, the system level performance is not degraded. For other classes of applications that have no such IO bottlenecks to begin with, our embedded solution does degrade system performance, however, we have also shown that in such cases scaling the acceleration is enough to counterbalance the IO performance loss caused by our solution.

Within this work, we have also compared the development time effort, performance, and area tradeoffs between the custom RTL design methodology used when end-users have full access to their FPGAs, and our embedded solution to be offered by data centres when such levels of access can not be granted to the end-users. Through our comparisons of experimental results, we make an argument in favour of the embedded-system approach. That conclusion was derived based not only on the ease of automation available within the current embedded-system EDA space of FPGA vendors but by the experimentally verified validity in aggregate performance and throughput of the accelerators not being compromised. The total effect on power consumption through the use of our embedded system approach to data transfers is predicted to be higher; however, it was beyond the scope of this paper to perform any detailed power analysis, and that remains to be completed as a future work in progress.

Authors' contributions

TM performed all of the RTL and embedded systems design that were necessary to conduct the experiments discussed in this article. SG conceived of the study to contrast embedded systems based IO control with purely RTL based approaches. Both authors participated in the selection and framing of the applications chosen for this study. Both authors read and approved the final manuscript.

About the Authors

Theepan Moorthy (S'04) received the B.A.Sc. in Computer Engineering from Queen's University in Kingston, ON, Canada in 2004. After spending two years in industry working on wireless chipset designs he returned to academia to pursue graduate degrees, and earned the M.A.Sc. degree from Ryerson University, Toronto, ON, Canada in 2008.

He later started his doctoral program at the University of British Columbia, Vancouver, Canada and transitioned from previously having worked on FPGA acceleration for video encoding to acceleration of bio-informatics algorithms. During his PhD he has held various internships at PMC-Sierra and Xilinx Research Labs.

Sathish Gopalakrishnan is an Associate Professor of Electrical & Computer Engineering at the University of British Columbia. His research interests center around resource allocation problems in several contexts including real-time, embedded systems and wireless networks. Prior to joining UBC in 2007, he obtained a PhD in Computer Science and an MS in Applied Mathematics from the University of Illinois at Urbana-Champaign. He has received awards for his work from the IEEE Industrial Electronics Society (Best Paper in the IEEE Transactions on Industrial Informatics in 2008) and at the IEEE Real-Time Systems Symposium (in 2004).

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Received: 2 April 2017 Accepted: 8 August 2017
Published online: 22 August 2017

References

1. Haidar A, Cao C, Yarkhan A, Luszczek P, Tomov S, Kabir K, Dongarra J (2014) Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. pp 491–500
2. Liu C, Ng HC, So HKH (2015) Quickdough: A rapid fpga loop accelerator design framework using soft cgra overlay. In: Field Programmable Technology (FPT), 2015 International Conference on. pp 56–63
3. Byma S, Steffan JG, Bannazadeh H, Garcia AL, Chow P (2014) Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In: Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on. pp 109–116
4. Putnam A, Caulfield AM, Chung ES, Chiou D, Constantinides K, Demme J, Esmaeilzadeh H, Fowers J, Gopal GP, Gray J, Haselman M, Hauck S, Heil S, Hormati A, Kim JY, Lanka S, Larus J, Peterson E, Pope S, Smith A, Thong J, Xiao PY, Burger D (2014) A reconfigurable fabric for accelerating large-scale datacenter services. In: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA). pp 13–24
5. Lin Z, Chow P (2013) Zcluster: A zynq-based hadoop cluster. In: Field-Programmable Technology (FPT), 2013 International Conference on. pp 450–453
6. Norm Jouppi, Distinguished Hardware Engineer, Google, Google supercharges machine learning tasks with TPU Custom Chip. [Online]. Available: https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html Jouppi, Distinguished Hardware Engineer, Google, Google supercharges machine learning tasks with TPU Custom Chip. [Online]. Available: https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html
7. Cong J, Liu B, Neuendorffer S, Noguera J, Vissers K, Zhang Z (2011) High-level synthesis for fpgas: From prototyping to deployment. IEEE Trans Comput. Aided Des Integr Circ Syst 30(4):473–491
8. Ma S, Andrews D, Gao S, Cummins J (2016) Breeze computing: A just in time (jit) approach for virtualizing fpgas in the cloud. In: 2016 International Conference on, ReConFigurable Computing and FPGAs (ReConFig). pp 1–6
9. Chen F, Lin Y (2015) FPGA accelerator virtualization in OpenPOWER cloud. In: OpenPower Summit
10. Milford M, Mcallister J (2016) Constructive synthesis of memory-intensive accelerators for fpga from nested loop kernels. IEEE Trans Signal Process 99:4152–4165
11. Munshi A (2009) The OpenCL Specification. Khronos OpenCL Working Group
12. Altera ALTERA SDK FOR OPENCL. [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html
13. Morris GW, Thomas DB, Luk W (2009) Fpga accelerated low-latency market data feed processing. In: 2009 17th IEEE Symposium on, High Performance Interconnects. pp 83–89
14. Chrysos G, Sotiriades E, Rousopoulos C, Pramataris K, Papaefstathiou I, Dollas A, Papadopoulos A, Kirmitzoglou I, Promponas VJ, Theocharides T, Petihakis G, Lagnel J (2014) Reconfiguring the bioinformatics computational spectrum: Challenges and opportunities of fpga-based bioinformatics acceleration platforms. IEEE Design Test 31(1):62–73
15. Lockwood JW, Monga M (2015) Implementing ultra low latency data center services with programmable logic. In: 2015 IEEE 23rd Annual, Symposium on High-Performance Interconnects. pp 68–77
16. Erdmann C, Lowney D, Lynam A, Keady A, McGrath J, Cullen E, Breathnach D, Keane D, Lynch P, Torre MDL, Torre RDL, Lim P, Collins A, Farley B, Madden L (2014) 6.3 a heterogeneous 3d-ic consisting of two 28nm fpga die and 32 reconfigurable high-performance data converters. In: 2014 IEEE International Solid-State, Circuits Conference Digest of Technical Papers (ISSCC). pp 120–121
17. Morgenstern B, Frech K, Dress A, Werner T (1998) DIALIGN: Finding Local Si milarities by Multiple Sequence Alignment. Bioinformatics 14(3):290–294
18. Boukerche A, Correa Jan M, Cristina A, de Melo MA, Ricardo Jacobi P (2010) A Hardware Accelerator for the Fast Retrieval of DIALIGN Bilogical Sequence Alignments in Linear Space. IEEE Trans Comput 59(6):808–821
19. Xilinx Xilinx University Program XUPV5-LX110T Development System. [Online]. Available: http://www.xilinx.com/univ/xupv5-lx110t.htm
20. Moorthy T, Gopalakrishnan S (2014) Gigabyte-scale alignment acceleration of biological sequences via ethernet streaming. In: Field-Programmable Technology (FPT), 2014 International Conference on. pp 227–230
21. Woods L, Eguro K (2012) Groundhog - A Serial ATA Host Bus Adapter (HBA) for FPGAs. IEEE 20th Int. Symp Field-Programmable Cust. Comput. Mach:220–223
22. Xilinx MicroBlaze Soft-Processor IP Protocol Specification. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf
23. Eguro K (2010) SIRC: An Extensible Reconfigurable Computing Communication API. 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines:135–138
24. Mendon AA, Huang B, Sass R (2012) A high performance, open source SATA2 core. Field Programmable Logic Appl. (FPL), 2012 22nd Int. Conf. 421–428
25. ARM AMBA AXI Protocol Specification. [Online]. Available: http://www.arm.com/products/system-ip/amba-specifications.php
26. Xilinx Fast Simplex Link IP Protocol Specification. [Online]. Available: http://www.xilinx.com/products/intellectual-property/fsl.html
27. Multi-Port Memory Controller IP Protocol Specification. [Online]. Available: http://www.xilinx.com/products/intellectual-property/mpmc.html
28. Processor Local Bus IP Protocol Specification. [Online]. Available: http://www.xilinx.com/products/intellectual-property/plb_v46.html
29. Summers J, Brecht T, Eager D, Gutarin A (2016) Characterizing the workload of a netflix streaming video server. In: 2016 IEEE International Symposium on, Workload Characterization (IISWC). pp 1–12
30. Delimitrou C, Kozyrakis C (2013) The netflix challenge: Datacenter edition. IEEE Comput. Archit. Letters 12(1):29–32
31. Aaron A, Li Z, Manohara M, Lin JY, Wu ECH, Kuo CCJ (2015) Challenges in cloud based ingest and encoding for high quality streaming media. In: Image Processing (ICIP) 2015 IEEE International Conference on. pp 1732–1736

32. Wiegand T, Sullivan GJ, Bjontegaard G, Luthra A (2003) Overview of the h.264/avc video coding standard. IEEE Trans. Circ. Syst. Video Technol 13(7):560–576
33. ITU Telecom. Standardization Sector of ITU., Advanced video coding for generic audiovisual services. ITU-T Recommendation H.264, May 2003
34. Liu Z, Huang Y, Song Y, Goto S, Ikenaga T (2007) Hardware-Efficient Propagate Partial SAD Architecture for Variable Block Size Motion Estimation in H.264/AVC. Proc 17th Great Lakes Symp. VLSI. 160–163
35. Moorthy T, Ye A (2008) A Scalable Computing and Memory Architecture for Variable Block Size Motion Estimation on Field-Programmable Gate Arrays. Proc. 2008 IEEE conf. Field Programmable Logic Appl:83–88
36. Stone JE, Gohara D, Shi G (2010) OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Comput. Sci. Eng 12(3):66–73
37. Amazon Amazon EC2 F1 Instances. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/
38. Microsoft Microsoft Azure. [Online]. Available: https://azure.microsoft.com/en-us/resources/videos/build-2017-inside-the-microsoft-fpga-based-configurable-cloud/
39. Moorthy T (2008) Scalable FPGA Hardware Acceleration for H.264 Motion Estimation. Ryerson University, Theses and Dissertations