FPGA-based Evaluation Platform for Disaggregated Computing

Dimitris Theodoropoulos
dtheodor@ics.forth.gr

Nikolaos Alachiotis
nalachio@ics.forth.gr

Dionisios Pnevmatikatos
pnevmati@ics.forth.gr

Computer Architecture and VLSI Systems Laboratory
Institute of Computer Science, Foundation for Research and Technology - Hellas (FORTH)
100 Plastira Avenue, Vassilika Vouton
GR-70013 Heraklion - Crete, GREECE

***Abstract*—Disaggregated computing aims at overcoming the problem of fixed resource proportionality in existing infrastructures while advancing resource allocation to virtual machines, which is currently restricted by the physical boundaries of a server tray. Organizing resources into large homogeneous pools (e.g compute, memory, accelerators, etc) enables the demand-driven, fine-grained allocation of resources, effectively leading to improved resource utilization and significant power savings. However, the success of this approach relies on how efficiently the underlying resources are utilized by the software application. To facilitate software development in disaggregated computing environments, we introduce a versatile multi-FPGA evaluation platform that can serve as an early exploration tool for the involved trade-offs and execution alternatives given the application at hand. To increase functionality of the proposed development/evaluation platform, we consider three types of building blocks, namely compute, memory, and accelerator ones, providing the developer with the option to instantiate and interconnect them in proportion to the application demands, thus facilitating both compute- and memory-intensive applications. We have implemented a fully fledged prototype platform, based on three interconnected Zynq boards, and rely on a thin user-level API to allocate compute and memory resources on remote blocks, transfer data, and deploy reconfigurable accelerators. As a case study, we employ one of the Seven Dwarfs of Symbolic Computation, the matrix multiply benchmark.**

I. Introduction

The increased requirements of compute and/or memory demanding applications from various domains (e.g., bioinformatics, finance, etc) have led to the exploration of new processing approaches, e.g., on the cloud. Processing on the cloud is conducted on third-party—typically general-purpose—datacenters, which are traditionally organized as collections of trays with a fixed amount of compute, memory, and peripherals. Such organization can frequently cause sub-optimal resource utilization at the tray level, leading to inefficient processing and increased energy consumption.

To this end, an increasingly explored solution relies on the concept of disaggregation, where compute, memory, and acceleration resources are organized as large homogeneous pools that can be allocated to specialized virtual machines (VMs) to better meet application-driven requirements. Model-based code execution on different configurations of disaggregated platforms can provide valuable insights in the various involved trade-offs related to performance scaling and resource allocation, consequently leading to more effective optimizations. Direct access to hardware-realized disaggregated evaluation platforms, supported by the required software-level infrastructure to minimize application porting effort, is of great significance to developers and Infrastructure-as-a-service (IaaS) providers in order to assess potential performance and Total Cost of Ownership (TCO) gains.

Our work represents an early effort toward this direction, providing a fully fledged prototype development platform for the thorough exploration of alternative execution strategies of applications in disaggregated computing environments. Given that generality and flexibility is of paramount importance, we propose a versatile architecture that employs three different types of processing nodes, with each node, henceforth referred to as a block, homogeneously comprising compute, memory, or accelerator resources. An arbitrary number of blocks of each type can be deployed on an as-needed basis to meet the requirements of compute- or memory-intensive applications. This models real-world disaggregated environments where resources are assigned to virtual machines driven by application demands. Therefore, in addition to facilitating software development and allowing to evaluate different optimization approaches, alternative hardware configurations can be explored. Our approach comprises a thin user-level API, implemented as a software library, to facilitate block configuration, remote memory allocation, data transfers, and accelerator deployment, collectively minimizing the required porting effort.

More specifically, we make the following contributions:

- We present a multi-FPGA evaluation platform to facilitate the exploration of alternative execution strategies and code optimizations for compute- and memory-intensive applications in disaggregated environments.
- We describe a light-weight software stack that provides basic functionality for remote memory allocation, data transfers, accelerator deployment and control, as well as platform debugging.
- We evaluate various software and hardware configurations for the matrix multiply benchmark, one of the Seven Dwarfs of Symbolic Computation [1].

The remainder of this paper is organized as follows. Section II revises related work and active research projects on acceleration infrastructure for datacenters. Section III presents our proposed platform architecture, while Section IV describes

the supporting software stack. Section V shows the multi-FPGA hardware prototype. Section VI presents experimental results for different configurations of a matrix multiplication benchmark. Finally, Section VII concludes this work and provides directions for future work.

II. Related Work

Current research projects on next-generation datacenters have proposed the employment of hardware accelerators to improve system performance. The Ecoscale project [2], for instance, proposes a novel architecture to automatically map and execute HPC applications to platforms, supported by reconfigurable modules. Vineyard [3] develops an integrated platform for heterogeneous accelerator-based servers, aiming at achieving improved performance and reduced energy consumption when compared with current solutions.

In the industrial domain, the HC series platforms from Micron [4] utilize QPI-connected x86 CPUs that are tightly coupled with an FPGA-based reconfigurable coprocessor. Maxeler offers dataflow computing platforms [5] that consist of so-called DFEs (Data Flow Engines), i.e., reconfigurable accelerators that are either tightly coupled with local Intel Xeon CPUs or shared with a host CPU over Infiniband.

Microsoft has already deployed the Catapult platform [6], a cloud-scale implementation based on Stratix V FPGAs, with applications in service (Web search ranking) and network (encryption) acceleration. Intel has launched the HARP (Intel-Altera Heterogeneous Architecture Research Platform) project [7], which tightly couples Xeon CPUs with an Altera Stratix V FPGAs, connected over a cache-coherent QPI. In addition, IBM proposes the CAPI architecture [8] for offloading algorithms to an FPGA; applications are executed on a Power8 CPU and can coherently exchange data with user-specific hardware accelerators.

Researchers have also proposed various works on coupling FPGA boards as processing platforms to host machines. For example, Fahmy et. al. [9] present a framework that integrates reconfigurable accelerators in a standard datacenters, which supports virtualised resource management and communication. The proposed framework utilizes the PCIe interface for hardware reconfiguration and data transfers, whereas the software infrastructure exposes an Application Programming Interface (API) at user level, that facilitates FPGA programming and management.

Vipin et. al. [10] present DyRACT, a platform that supports FPGA partial reconfiguration at runtime from a host PC, over a static PCIe interface. A static configuration controller is responsible for the user logic control and management via a set of AXI Stream-based FIFOs. Moreover, a supporting software infrastructure exposes a user library to the host machine, which facilitates partial reconfiguration and data transfers. DyRACT is implemented on a Virtex6 ML605 and a Virtex7 VC707 board; a video-processing application, consisting of different partial bitstreams, was used as a case study to validate and evaluate the proposed platform.

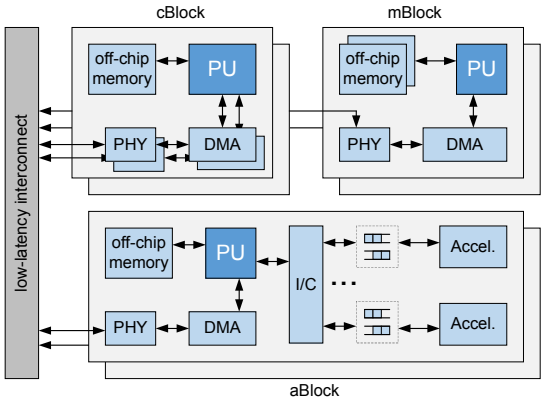

Fig. 1. High-level architecture of the supported processing platforms (blocks), namely compute block (cBlock), memory block (mBlock), and accelerator block (aBlock).

Finally, Kane et. al. [11] also present a framework for integrating FPGAs in datacenters. The proposed framework extends the OpenStack control, by introducing a set of custom modules for (a) abstracting resources to a manageable pool, (b) efficient sharing of available resources among different tenant threads to the host machine, (c) interfacing the underlying hardware, and (d) securing the host environment by illegal hardware transactions (e.g. memory accesses to kernel space, etc). A prototype was successfully mapped on an IBM server with an Intel Xeon host CPU and a Xilinx Kintex7 FPGA, communicating over a PCIe interface.

III. Disaggregated Platform

Within the context of resource disaggregation, compute, memory, and acceleration resources are organized as large pools that form VMs tailored to the application requirements. Compute-intensive applications, for instance, fit better to disaggregated configurations that utilize a large number of compute and/or accelerator resources; memory-demanding workloads, on the other hand, would benefit vastly from a disaggregated platform with increased memory resources.

We adopt an architecture that relies on three different types of processing platforms (blocks) to (a) facilitate general-purpose computing, the compute block or cBlock, (b) host large memory resources, i.e. the memory block or mBlock, and (c) support hardware acceleration, i.e. the accelerator block or aBlock, communicating over a low-latency interconnect. As illustrated in Figure 1, the cBlock integrates a local high-performance processing unit (PU), connected to an off-chip memory for instructions/data, and a set of DMA engines that directly interface high-speed transceivers (PHY) for block-to-block data transfers.

The mBlock integrates a large amount of memory resources, accessible by a local PU over wide high-speed interfaces. As in cBlock, the PU is also connected to a DMA engine that interfaces high-speed transceivers for block-to-block communication. Note that the mBlock opts for a general-purpose PU for accessing memory (instead of a custom hardware module), thus providing the flexibility to execute any software task close to it (described in Section V). Finally, the aBlock also hosts

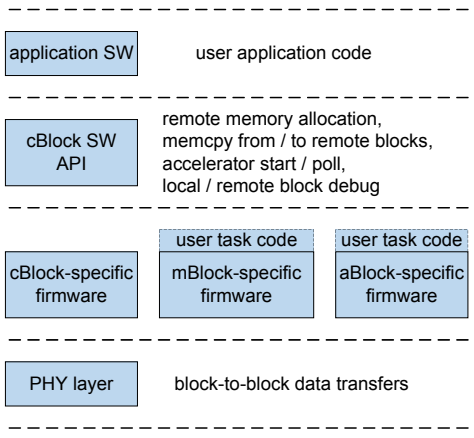

Fig. 2. Block software stack; the PHY layer is used for data transfers, each block type firmware implements a set of block-related primitives, the cBlock API provides block-to-block malloc, memcpy, accelerator invocation, and debug functions, while the application SW hosts the user code.

cBlock RV table

id	name	board id	start	size	type
1	a	mBlock	0x40000000	0xF	float
2	b	mBlock	0x40000010	0xF	float
3	c	aBlock	0x80000000	0xF	float
			...		

RV = remote variable

aBlock RV table

id	start	size	type
3	0x80000000	0xF	float
	...		

mBlock RV table

id	start	size	type
1	0x40000000	0xF	float
2	0x40000010	0xF	float
	...		

Fig. 3. Tables across each block type that are used to store all required metadata of remotely allocated variables.

a local PU connected to an off-chip memory and a DMA engine that interfaces high-speed transceivers for block-to-block data transfers. Furthermore, the PU can interface locally instantiated hardware accelerators via memory-mapped queues that are attached to a high-performance interconnect (I/C).

IV. Software Stack

To facilitate block-to-block communication and data transfers, we have developed a thin software layer executed on each block type, illustrated in Figure 2. The "PHY layer" directly interfaces the hardware PHY module (the Aurora IP in our current prototype), and is responsible for data transfers from/to other blocks at the physical level. On top of the "PHY layer", each block hosts a custom firmware that includes a set of common functions for initialization, internal testing, connectivity probing and debug (described below). Moreover, it provides software primitives for remote memory allocation and data copies. The aBlock firmware also interfaces available accelerators for data transfers, control and status polling. Developers also have the option to execute code over the mBlock and aBlock middleware, in order to explore the possibility for near-data processing using the local PU.

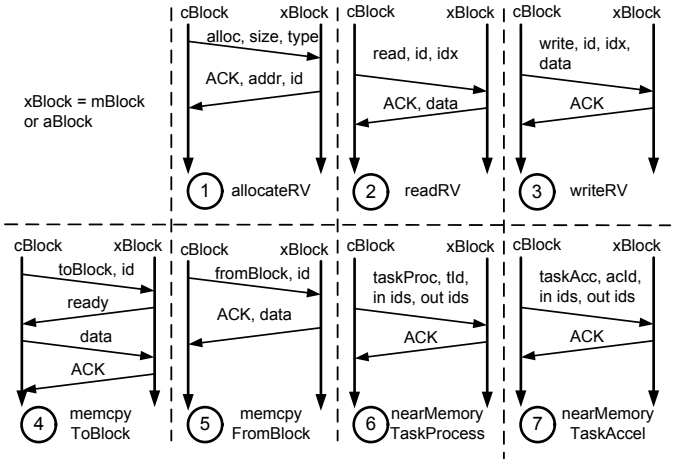

Fig. 4. Flow of block-to-block operations for (1) RV allocation, (2) reading a RV, (3) writing a RV, (4) perform a memcpy from a cBlock to an xBlock (x = m or a), (5) perform a memcpy from an xBlock to a cBlock, (6) invoke a remote task, and (7) a remote hardware accelerator.

The cBlock software stack additionally includes the "cBlock SW API". It uses the custom *remote variable* (RV) structure, which keeps all required metadata (unique id, readable name, hosting board id, starting address, size, type) for remote memory / variables management, and data transfers. The cBlock SW API currently supports the most frequently-used data types (sp-float, int, unsigned int, short, unsigned short, char).

Moreover, mBlock and aBlock maintain a table that stores all data of locally allocated variables. As an example, Figure 3 illustrates how the required metadata of three remotely allocated variables (a, b, and c) are stored across each block type. For each variable the "cBlock RV table" keeps its id, name, the board id where its space is allocated, the remote starting address, its size, and finally its type. On the other hand, the "mBlock RV table" has two entries (for a and b), each keeping the variable id, starting address, size, and type. Finally, the "aBlock RV table" has a single entry for variable c. As it is described later, the aforementioned structure reduces network control overheads, since it requires only variable ids to be exchanged for block-to-block operations.

Figure 4 shows the flow of the supported block-to-block operations that the software stack exposes at user-level:

- *allocateRemoteVariable()*: Instructs an xBlock (x = m or a) middleware to allocate space in its local memory for a remote variable of a specified size. The cBlock transmits a message that contains an allocation instruction id, the requested size and variable type. The xBlock tries to perform a local malloc, and if succeeds, it returns a message with an ack, the variable starting address, and its assigned id, else returns a failed command.
- *readRemoteElement()*: Caches and prints the i-th element of a remote variable allocated in an xBlock. The cBlock transmits a message that contains a read instruction id, the variable id, and the element index (i). The xBlock replies back a message with an ack and the requested

i-th element, or a failed command.

- *writeRemoteElement()*: Writes the i-th element of a remote variable in an xBlock. The cBlock transmits a message with the write instruction id, the variable id, and the element index (i). The xBlock updates the i-th element of the requested variable, and replies back a message that contains an ack if succeeds, or a failed command.
- *memcpyToBlock()*: Copies data from the cBlock to the pre-allocated memory space of a remote variable in a destination xBlock. The cBlock transmits a message with the "toBlock" instruction id, and the variable id that will be written. The xBlock configures the local PHY link to be ready to accept data equal to the variable size, and transmits back a ready message. The cBlock then transmits a message with the variable data to the xBlock. If it succeeds the latter sends an ack, or a failed command.
- *memcpyFromBlock()*: Copies the remote variable data from the specified xblock to a pre-allocated buffer on the local memory of the source cBlock. The cBlock sends a message with the "fromBlock" instruction id, and the variable id that will be read. The xBlock replies back with a message that contains an ack, and all variable data if it succeeds, or a failed command id.
- *nearMemoryTaskProcess()*: Forwards the task id and its remote variable input / output argument ids to an xBlock for task execution on the local PU; cBlock processing is paused until the function returns. The cBlock sends a message with the "taskProc" instruction id, the task tId that will be executed, and the input/output variable ids. The xBlock calls the task tId with arguments the received input/output variable ids. When done, if succeed, it sends an ack to the cBlock, or a failed command.
- *nearMemoryTaskAccel()*: Forwards the remote variable input / output argument ids to an accelerator integrated in a aBlock; cBlock processing is paused until the function returns. The cBlock sends a message with the "taskAcc" instruction id, the accelerator acId that will be invoked, and the input/output variable ids that will be used as calling arguments for the accelerator. The xBlock starts the acId, and when done, if succeed, it sends an ack to the cBlock, or a failed command.
- *printRemoteVariable()*: Iterates a remote variable in an xBlock, and prints its elements (not shown in Figure 4).
- *testBoardConnections()*: Probes boards connectivity; the cBlock exchanges a set of test/ack messages with the connected xBlocks (not shown in Figure 4).

To summarize, the cBlock SW API essentially provides all the necessary functions for memory allocation, data transfers, matrix iteration and debugging. This allows developers to swiftly change application mappings and perform tests on an actual multi-FPGA evaluation hardware platform, thus considerably reducing porting efforts. Finally, note that the cBlock SW API can be used in both OS-based and bare-metal environments.

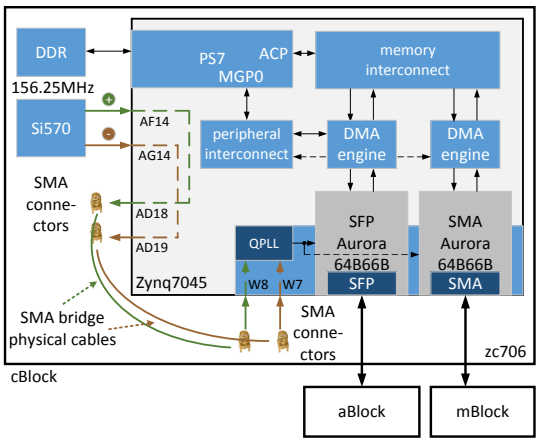

Fig. 5. The experimental platform consists of a cBlock, an mBlock, and an aBlock, implemented on three ZC706 interconnected boards respectively. The mBlock and aBlocks are connected to the cBlock using four SMA and one SFP cables, respectively.

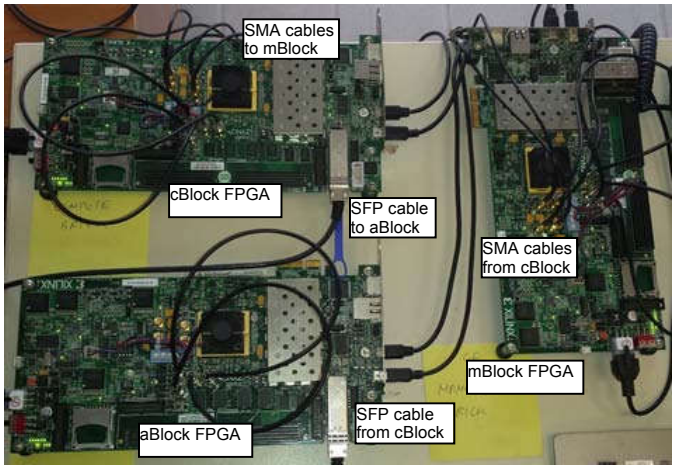

Fig. 6. The three ZC706 interconnected boards; the upper left board (cBlock) connected with an SFP cable to the lower left one (aBlock) and 4 SMA cables with the right one (mBlock).

V. Hardware Prototype

To validate our work, we implemented the cBlock, mBlock, and aBlock architectures to hardware and mapped them to three ZC706 Xilinx FPGA boards. In all cases, we used the integrated processing system (PS7) as PU, which packs two ARM Cortex A9 CPUs with 32KB I/D L1 cache, and 512KB L2 cache. We note that the supporting SW stack (described below) is highly modular, hence can easily target different FPGA-based SoCs (e.g., Xilinx Zynq Ultrascale+) with minimal modifications.

Figure 5 illustrates how the three ZC706 boards are connected, while focusing on the cBlock internal implementation. The PS7 is connected to the local off-chip DDR memory for data/instructions. Its Accelerator Coherency Port (ACP) interfaces with a high-performance memory interconnect to

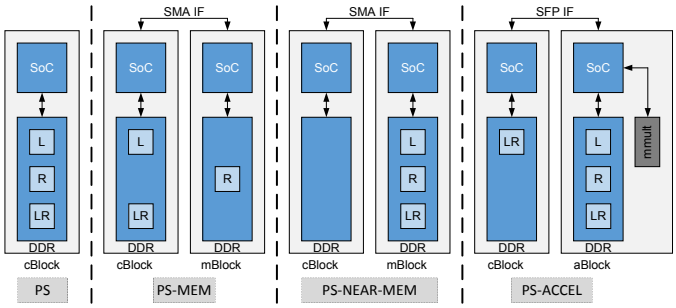

Fig. 7. Matrix multiplication mappings; in PS all matrices are stored in the cBlock, in PS-MEM L, LR matrices are stored in the cBlock, while R is stored in the mBlock, in PS-NEAR-MEM and PS-ACCEL all matrices are stored in the mBlock and aBlock respectively.

support two DMA Engines mapped to programmable logic (PL) for cache-coherent data transfers between the Aurora SMA/SFP physical interfaces and the local DDR memory. The PS7 can configure both DMA engines via its MGP0 port.

The cBlock SFP link is connected to the aBlock board, which integrates the SFP $\leftrightarrow$ DMA Engine $\leftrightarrow$ PS7 hardware path (i.e. not SMA-related hardware), and a memory-mapped FIFO for data exchange over AXI-Stream with a matrix-multiplication accelerator [12]. Furthermore, the cBlock SMA link is connected to the mBlock board, which only integrates the illustrated SMA $\leftrightarrow$ DMA Engine $\leftrightarrow$ PS7 hardware path.

Regarding clock generation on each FPGA board, the utilized FPGA transceivers (GTX) connected to the SMA and SFP interfaces, are clocked as shown in Figure 5: The on-board Si570 Oscillator generates a 156.25MHz clock, which is forwarded to the GTX quad PLL using two physical SMA bridge cables. Note that in the current prototype, both links achieved an effective 3.6 Gb/sec data throughput. Finally, the actual complete disaggregated multi-FPGA platform is shown in Figure 6. The top left, bottom left and right boards are classified as cBlock, mBlock, and aBlock respectively, connected as described above.

VI. Experimental Results

Platform evaluation: To evaluate our platform, we use the matrix multiplication Dwarf of Symbolic Computation, since it is one of the most widely used computations required in linear algebra algorithms [1]. Moreover, we use an HLS-based matrix multiplication IP provided by Xilinx [12] as hardware accelerator (mmult) implemented on the aBlock.

To demonstrate the platform flexibility for testing different application configurations, we execute four different $n \times n$ floating-point single-precision matrix multiplication mappings, as illustrated in Figure 7. In the first (outer left) mapping, the PS allocates input L, R and output LR matrices in the cBlock DDR, then the PS-MEM allocates L, LR matrices in the cBlock DDR and R matrix in the mBlock DDR, the PS-NEAR-MEM allocates all L, R, and LR matrices in the mBlock DDR, and finally the PS-ACCEL allocates both L, R matrices in the aBlock DDR and LR matrix in the cBlock DDR. Both PS and PS-MEM mappings execute all operations

```
void mmult(float **L, remoteVariable *rvR, float **LR) {
  int i,j,k; float* leftRow, cell; u32 tmpData[MSIZE];

  for (i = 0; i < MSIZE; i ++ ) {
    for (j = 0; j < MSIZE; j ++ ) {
      cell = 0.0f;
      leftRow = L[i];
      //cache the j-th line to cBlock
      memcpyFromBrick(MEM_BLOCK,tmpData,&rvR[j]);
      for (k = 0; k < MSIZE; k ++ )
        cell += leftRow[k] * ((float*)&tmpData)[k];
      LR[i][j] = cell;
    }
  }
}

void main() {
  float **L, **LR; int i,j;
  remoteVariable rvR[MSIZE];
  //malloc, and initialize L, LR matrices
  ...
  //allocate remotely MSIZE 1-d variables, each of MSIZE
  for (i=0;i<MSIZE;i++) {
    strcpy(rVName,strcat("rvR",sprintf(buff,"%d",i)));
    allocateRemoteVariable(&rvR[i], MEM_BLOCK, MSIZE,
    FLOAT_T,rVName);
  }
  //copy R data to MEM_BLOCK
  for (i=0;i<MSIZE;i++) {
    for (j=0;j<MSIZE;j++) {
      tmpBuffer[j] = (float)(i+j+12);
    }
    memcpyToBrick(MEM_BLOCK,(u32 *)tmpBuffer,&rvR([i]);
  }
  //do the L*rvR = LR multiplication
  mmult(L,rvR,LR);
}
```

Fig. 8. Code excerpt that shows the PS-MEM mapping code execution to the cBlock; the code allocates L and LR matrices locally to the cBlock, then allocates remotely MSIZE 1-d variables, each one representing one line of the rvR matrix, and after L and rvR are initialized, the multiplication output is locally stored to the LR matrix.

on the cBlock, however PS-MEM first caches the j-th column of R matrix from the mBlock before calculating the i-th LR output line. On the other hand, the PS-NEAR-MEM processes all data in the mBlock. Finally the PS-ACCEL perform all calculations in the hardware mmult IP.

The code excerpt shown in Figure 8, shows how one can use the cBlock API to implement the PS-MEM mapping (i.e., allocate a remote variable to an mBlock, and use it throughout data calculations done in the cBlock). The mmult function arguments are the input local 2-d "L" matrix and remote variable "rvR", and an output local 2-d "LR" matrix. Moreover, within the nested for-loops, the code simply does a memcpyFromBrick to cache the j-th line of rvR to the tmpData local variable, before calculating the i,j element of the output LR matrix.

Within the main function, apart from allocating memory and initializing local variables L, and LR, the code declares a 1-d array of remote variables, each entry keeping the data of the i-th line, i=0,...,MSIZE-1. The first for-loop, calls MSIZE times the "allocateRemoteVariable" function to allocate memory in the mBlock for each rvR[i] variable of type float. As described in Figure 3, the allocateRemoteVariable will create MSIZE entries in the "cBlock RV table", and the mBlock middleware

```
void main() {
  remoteVariable rvL[MSIZE],rvR[MSIZE],rvLR[MSIZE];
  //allocate remotely MSIZE 1-d variables, each of MSIZE
  for (i=0;i<MSIZE;i++) {
    strcpy(rVName,strcat("rvL",sprintf(buff,"%d",i)));
    allocateRemoteVariable(&rvL[i], MEM_BLOCK, MSIZE,
    FLOAT_T,rVName);
    strcpy(rVName,strcat("rvR",sprintf(buff,"%d",i)));
    allocateRemoteVariable(&rvR[i], MEM_BLOCK, MSIZE,
    FLOAT_T,rVName);
    strcpy(rVName,strcat("rvLR",sprintf(buff,"%d",i)));
    allocateRemoteVariable(&rvLR[i], MEM_BLOCK, MSIZE,
    FLOAT_T,rVName);
  }
  //copy L,R data to MEM_BLOCK
  for (i=0;i<MSIZE;i++) {
    for (j=0;j<MSIZE;j++) {
      tmpBuffer[j] = (float)(i+j+12);
    }
    memcpyToBrick(MEM_BLOCK,(u32 *)tmpBuffer,&rvL([i]);
    for (j=0;j<MSIZE;j++) {
      tmpBuffer[j] = (float)(i+j+20);
    }
    memcpyToBrick(MEM_BLOCK,(u32 *)tmpBuffer,&rvR([i]);
  }
  //do the rvL*rvR = rvLR multiplication
  nearMemoryTaskProcess(MEM_BLOCK,MMULT0_SW_ID, &rvL[0],
  &rvR[0],&rvLR[0])
}
```

Fig. 9. Code excerpt that shows the PS-NEAR-MEM mapping code execution to the cBlock; the code first allocates remote matrices rvL, rvR, and rvLR to the mBlock memory, then initializes all matrices by performing memcpyToBrick transactions, and finally calls the remote task with id MMULT0_SW_ID to do the matrix multiplication.

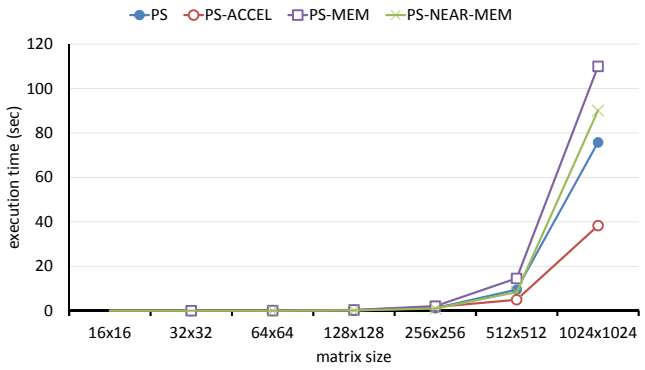

Fig. 10. Execution time of the matrix multiplication on mappings PS, PS-MEM, PS-NEAR-MEM and PS-ACCEL.

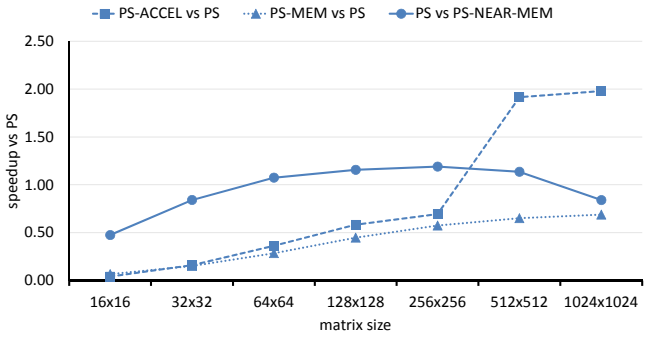

Fig. 11. Speedup of the matrix multiplication on mappings PS-MEM, PS-NEAR-MEM and PS-ACCEL against PS.

will update its own "mBlock RV table" structure with the allocated memory boundaries for each local variable. Within the last two for-loops, the code initializes a buffer with the data of each line in matrix rvB, and performs a memcpyToBrick to transfer the data to the corresponding allocated memory space in the mBlock. Once all data are transferred the code calls the mmult function to calculate the output matrix.

The code excerpt in Figure 9 shows also how one can implement the PS-NEAR-MEM mapping (i.e., instruct the cBlock code to perform near-memory software task processing in an mBlock). The mBlock-specific firmware needs first to be updated with the user task code (and assign a unique id - MMULT0_SW_ID) that will be invoked by the cBlock code (as shown in Figure 2).

Within the main function, as before, the code declares three remote variables, namely "rvL", "rvR", and "rvLR", and within the first for-loop allocates the required space for each matrix line in the mBlock memory. Once the memory allocation succeeds, the code performs consecutive memcpyToBrick's, in order to initialize the rvL and rvR matrices. Finally, it calls the "nearMemoryTaskProcess" with arguments that designate the mBlock, the user task code to be invoked (MMULT0_SW_ID), and all input / output remove variables.

The cBlock code can follow a similar approach to implement the PS-ACCEL mapping (i.e. invoke hardware accelerators that are already mapped to reconfigurable logic in an aBlock). Within the main function, the code first allocates the desired space for each matrix line in the aBlock memory, and then initialize all matrix data with consecutive memcpyToBrick's. Finally it calls the "nearMemoryTaskAccel" function with arguments that designate the target aBlock, a pre-assigned hardware accelerator id (defined in the aBlock-specific firmware), and all input / output remove variables.

Exploring performance: Figure 10 and Figure 11 show the overall execution time and speedup respectively of all tested mappings on our hardware prototype. As expected, up to 128×128, where all data ($128 \times 128 \times 4 = 64KB/matrix$) fit within the PS7 L1 (32KB) and L2 (512KB) caches, PS outperforms the PS-MEM and PS-ACCEL mappings. However, as the matrix sizes increase, the application fits better to an accelerator-oriented disaggregated configuration. This is verified by the 1024×1024 case, where PS-ACCEL becomes up to 2x faster compared to the baseline PS mapping. On the other hand, PS-MEM becomes up to 30% slower compared to PS, however this mapping would be useful in cases, when the cBlock local memory could not fit all matrix data.

Interestingly, the PS mapping outperforms the PS-NEAR-MEM mapping for sizes 16×16, 32×32 and 1024×1024. In contrast, for sizes ranging from 64×64 up to 512×512, the PS-NEAR-MEM mapping performs calculations %15 on average faster than the PS one. Up to 32×32 sizes, the cBlock PS7 caches are large enough to fit all matrices plus other code data, without incurring significant overheads due to misses, as in the mBlock PS7 for the PS-NEAR-MEM mapping.

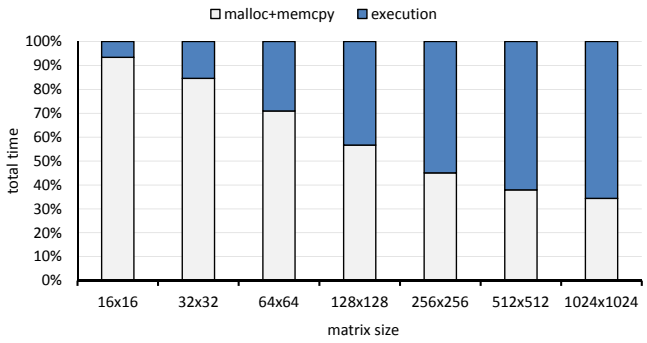

Fig. 12. % utilization of the memory allocation, data transfers and actual execution time with respect to the overall elapsed time for the PS-MEM mapping.

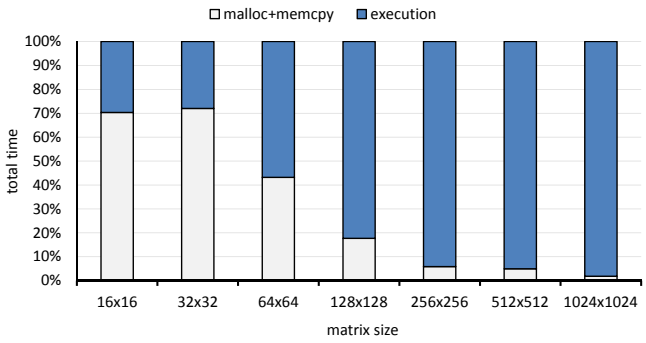

Fig. 13. % utilization of the memory allocation, data transfers and actual execution time with respect to the overall elapsed time for the PS-ACCEL mapping.

However, the latter sustains an interconnect overhead, while data are transferred from the cBlock to the mBlock memory. From 64×64 up to 512×512 (i.e. data size are increasing), the interconnect overhead is partially hidden by the fact that incoming data directly update (and mostly fit) to the mBlock PS7 caches, since the local DMA engines are connected to the PS7 Accelerator Coherency Port (ACP). For 1024×1024 sizes though, matrix data transferred to the mBlock do not fit to the PS7 caches, leading to misses, which in combination with the actual data transfer overhead, result into the PS mapping outperforming the PS-NEAR-MEM one by approximately 16%.

In addition, Figures 12, 13, and 14 break down the overall time for PS-MEM, PS-ACCEL, and PS-NEAR-MEM mappings into portions spent for remote memory allocation/data transfers (malloc+memcpy) and the actual execution time (execution) respectively. As expected, for small matrix sizes (e.g. 16×16, 32×32), most of the overall time is spent on board-to-board communication for control messages and data transfers. However, as sizes tend to increase, the actual execution time portion increases, and eventually becomes larger compared to the board-to-board communication overheads.

Finally, an interesting observation for the PS-MEM mapping is that for 1024×1024 matrices multiplication, it still spends more than 30% of its time on data transfers. This leads to the observation that, given the fixed PS7 performance, there is still potential for improvement by upgrading the board-to-board interconnect link. On the other hand, in both PS-NEAR-MEM and PS-ACCEL mappings the overall time is dominated by the actual data processing; approximately 95% of its time is spent on calculations, thus potential performance improvements are expected by more efficient processing systems (applicable to the mBlock) or faster hardware mmult IPs (applicable to the aBlock).

Exploring energy consumption: Apart from exploring the performance potential of different application mappings, one can also examine their energy consumption on a real disaggregated platform. As an example case, Table I shows the static, dynamic (includes the PS) and transceiver power consumption of the three different block types (considering the

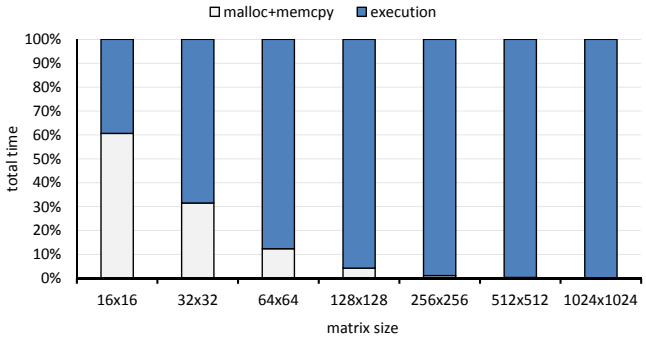

Fig. 14. % utilization of the memory allocation, data transfers and actual execution time with respect to the overall elapsed time for the PS-NEAR-MEM mapping.

TABLE I
REPORTED POWER CONSUMPTION OF EACH BLOCK TYPE (CASE OF 256×256)

Resource	cBlock	mBlock	aBlock
static power (W)	0.27	0.27	0.30
dynamic power (inc. PS) (W)	1.91	1.83	2.25
GTXs (W)	0.75	0.38	0.38

case of 256×256) matrix sizes for the mmult IP in aBlock). All types require approximately the same static power; both mBlock and aBlock require 0.38W for their SMA and SFP transceiver respectively, whereas the cBlock requires double power for its SMA and SFP transceivers. As expected, the aBlock requires the most dynamic power, since it includes the 256×256 mmult IP.

Figure 15 breaks down the energy consumption for the application mappings on all blocks, when considering the case of 256×256) matrix sizes. As observed, the least energy is spent on data transfers (GTX), whereas the most energy is spent on the PS and available reconfigurable logic. For *the considered case* (256×256) matrix sizes), the PS mapping is more energy efficient than both PS-MEM and PS-ACCEL ones. In fact, the PS-MEM requires the most energy, since the R matrix is at first stored to the mBlock, and during processing, each line is being cached to the cBlock. On the other hand, PS-NEAR-MEM consumes slightly less energy compared to

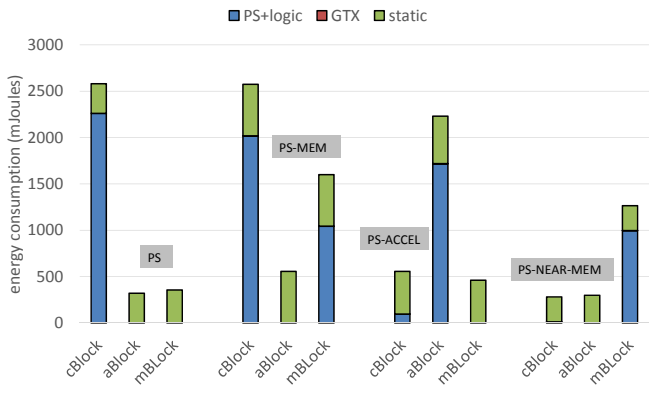

Fig. 15. Breakdown of the consumed energy for the application mappings on all blocks, when considering the case of 256×256) matrix sizes.

the PS mapping, since (as described before) during transfers, data are directly placed to the mBlock PS7 cache, resulting to shorter processing time.

Comparison with related work: The presented experiments demonstrate the platform flexibility and feasibility in terms of exploring and executing different application configurations on actual hardware with minimal effort. A quantitative comparison of the case study benchmark when executed on our prototype (performance and/or energy consumption), against other platforms (e.g. GPUs, highly optimized hardware multipliers), would simply compare the concept of resources disaggregation against other approaches, which is out of the scope of this work.

On the available features perspective, commercial platforms that also utilize FPGAs [6] [7] [4] [8] primarily target the traditional CPU - workload offload to accelerators paradigm. Similarly, recent research works [11], [9], [10] target processing environments that consider FPGAs as auxiliary platforms on host machines / datacenters, communicating via a high-speed interface (e.g. PCIe) for (partial) configuration / data transfers.

In contrast, our work focuses on platforms with truly disaggregated resources (compute, memory, configurable hardware), allowing users to explore different memory management (e.g. local vs remote variables) and processing (e.g. near-memory SW or HW processing) strategies. Such features - not available to any of the aforementioned platforms - may provide valuable insights to developers in terms of performance bottlenecks / power consumption, ultimately leading to reduced overall platform TCO.

VII. Conclusions and Future Work

We presented an FPGA-based evaluation platform for exploring application execution on disaggregated environments. Towards flexibility, our platform supports different processing types, which developers can instantiate and interconnect, as well as find optimal configurations that fit the application requirements. As future work, we plan to (a) support partial reconfiguration, (b) further improve interconnect latency, (c) enhance the cBlock SW API with more functionalities (e.g. more variable types), and (d) integrate in the aBlock an interface for direct access from the accelerators to local off-chip memory on the PL side.

References

[1] Erich L. Kaltofen, "The Seven Dwarfs of Symbolic Computation," in *Numerical and Symbolic Scientific Computing: Progress and Prospects*. Springer Vienna, 2012, pp. 95–104.

[2] I. Mavroidis, et.al., "Ecoscale: Reconfigurable computing and runtime system for future exascale systems," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 696–701.

[3] C. Kachris, et. al., "The VINEYARD approach: Versatile, Integrated, Accelerator-based, Heterogeneous Data Centres," in *International Symposium on Applied Reconfigurable Computing (ARC 2016)*, 2016, pp. 3–13.

[4] Micron Technology, Inc., "Convey HC2," https://www.micron.com/about/about-the-convey-computer-acquisition/hc-series, [Online; accessed 31-Oct-2016].

[5] Maxeler technologies, "MPC Series," https://www.maxeler.com/products, [Online; accessed 31-Oct-2016].

[6] A. Caulfield, et. al., "A cloud-scale acceleration architecture," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[7] Intel Corporation, "Intel QuickAssist Technology," http://www.intel.com/content/www/us/en/embedded/technology/quickassist/overview.html, [Online; accessed 31-Oct-2016].

[8] B. Wile, "Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems," Tech. Rep., 2014.

[9] S. A. Fahmy, et. al., "Virtualized FPGA Accelerators for Efficient Cloud Computing," in *International Conference on Cloud Computing Technology and Science (CloudCom)*, November 2015, pp. 430–435.

[10] K. Vipin and S. A. Fahmy, "DyRACT: A partial reconfiguration enabled accelerator and test platform," in *International Conference on Field Programmable Logic and Applications (FPL)*, September 2014, pp. 1–7.

[11] F. Chen, et. al., "Enabling FPGAs in the Cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, May 2014, pp. 1–10.

[12] Daniele Bagni, et. al., "A Zynq Accelerator for Floating Point Matrix Multiplication Designed with Vivado HLS," Application note, January 2016.