An Analysis of Accelerator Data-Transfer Modes in NoC-Based SoC Architectures

Kuan-Lin Chiu, Davide Giri, Luca Piccolboni and Luca P. Carloni
Department of Computer Science
Columbia University, New York, U.S.A
{chiu, davide_giri, piccolboni, luca}@cs.columbia.edu

2023 IEEE High Performance Extreme Computing Conference (HPEC) | 979-8-3503-0860-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPEC58863.2023.10363546

***Abstract*—Data movement is a key factor impacting the performance of hardware accelerators. In a complex SoC architecture, multiple accelerators compete for accessing the resources of on-chip communication and off-chip memory interfaces. For a program that invokes many accelerators, orchestrating the data movement is critically important to avoid degrading the speedup that each standalone accelerator can achieve. We present a comparative analysis of the two main data-transfer modes among accelerators: memory-based and point-to-point (p2p) communication. We describe their implementation on FPGA for both single-thread and multi-thread software programs. We analyze the implications on programmability, performance, and energy efficiency by using a variety of synthetic benchmarks to evaluate the data-transfer modes in different scenarios and by accelerating two real-world image processing applications: Nightvision and Wide-Area Motion Imagery (WAMI). We demonstrate that for various configurations of a tile-based many-accelerator SoC, p2p outperforms memory-based communication.**

***Index Terms*—accelerators, data-transfer modes, point-to-point communication, system-on-chip, image processing.**

I. Introduction

Heterogeneous system-on-chip (SoC) architectures combine general-purpose processors with hardware accelerators [1], [2] that are specialized for given domains such as graph analytics [3], machine learning [4], and image processing [5].

There are two main categories of accelerators: tightly-coupled and loosely-coupled [6]. Tightly-coupled accelerators are integrated with the processor's pipeline and execute fine-grain tasks on small datasets. Loosely-coupled accelerators, instead, are computing engines that are independent from particular processor cores in terms of their design and their operations. They are typically integrated on a bus or a network-on-chip (NoC) and perform coarse-grain tasks on large datasets. In this paper, we focus on loosely-coupled accelerators and their integration into NoC-based SoC architectures.

A loosely-coupled accelerator is usually invoked by software programs through a device driver. The device driver configures the execution of the accelerator by specifying, for example, where the data are located in the main memory. Complex programs typically invoke multiple accelerators, which communicate by using a shared region of memory. The program copies the input data for the accelerator in the shared region of memory and invokes the first accelerator. The accelerator performs the task by accessing the data in the main memory, and then returns the control to software when the task is completed. Other accelerators can be invoked afterwards, thus resulting in a sequence of invocations of accelerators that communicate through the main memory.

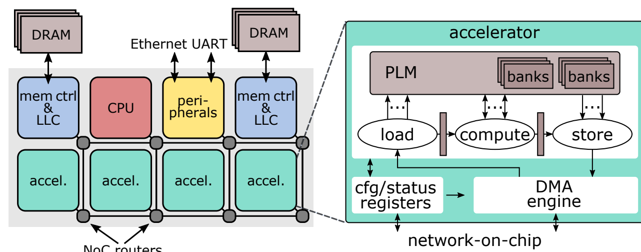

Fig. 1. A NoC-based SoC architecture with a heterogeneous tile organization.

Multiple accelerators can be invoked concurrently when their tasks are independent. While *memory-based communication* is the most used method for data transfers, Giri et al. propose using *point-to-point (p2p) communication* for accelerators [7]. With p2p communication, the accelerators can exchange data directly after being configured from software through device drivers. An accelerator can send data to another accelerator without using the off-chip main memory as a temporary buffer, thereby avoiding many costly data transfers.

Contributions. Our main contribution is a comprehensive analysis of these two main data-transfer modes in NoC-based heterogeneous SoC architectures. The analysis shows that the p2p communication mode provides more benefits in terms of performance, energy efficiency and programmability, compared to the memory-based communication mode. To complete our analysis, we make these additional contributions:

- We design many loosely-coupled accelerators including a synthetic accelerator, 3 accelerators for Nightvision, and 13 accelerators for Wide-Area Motion Imagery (WAMI).
- We develop three software programs for synthetic applications, four for Nightvision, and four for WAMI, all running on top of Linux.
- We deploy the heterogeneous SoC on an FPGA board and evaluate the performance and energy efficiency of the data-transfer modes.

To the best of our knowledge (Section VI), this is the first in-depth analysis of the accelerator data-transfer modes in NoC-based SoC architectures.

II. Hardware Architecture

To analyze the data movement of a NoC-based heterogeneous SoC, we utilize ESP, an open-source platform for agile SoC design [8], [9]. An SoC designed with ESP has a tile-based architecture with a configurable number of heterogeneous tiles. The left-hand side of Fig. 1 shows the example of

979-8-3503-0860-0/23/$31.00 ©2023 IEEE
Authorized licensed use limited to: Columbia University Libraries. Downloaded on March 07,2024 at 15:36:38 UTC from IEEE Xplore. Restrictions apply.

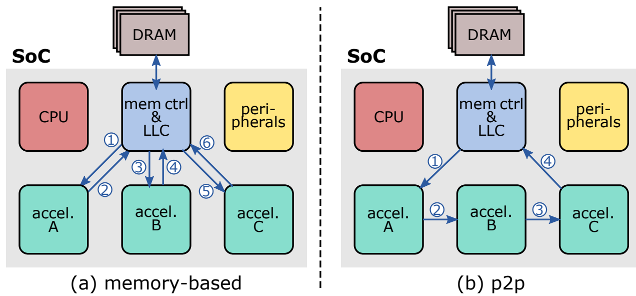

Fig. 2. Data movements required for memory-based and p2p communication for accelerator *A*, *B*, and *C*.

an SoC that has 8 tiles in a 4x2 matrix with: one processor tile (hosting a RISC-V CVA6 core [10]), four accelerator tiles (hosting loosely-coupled accelerators), two memory tiles (hosting memory controllers to communicate with off-chip DRAM banks), and an input/output tile (hosting various peripherals, such as Ethernet and UART). The core in the processor boots Linux and software programs use Linux device drivers to invoke the accelerators. Each tile is encapsulated in a socket that interfaces it to the router of a packet-switched 2D-mesh NoC with six physical planes [11]. Two of the planes are dedicated to DMA requests and responses between accelerator tiles and the memory tiles. Three other planes are for cache coherence messages and the last plane is for I/O, interrupts, monitoring, and debugging.

For our analysis, we design the accelerators in SystemC [12], and synthesize their RTL implementations by using commercial high-level synthesis (HLS) tools. We deploy the accelerators on FPGA and invoke them from a software program by using custom device drivers. Each device driver configures the corresponding accelerator through memory-mapped registers that define the location of the input and output data in memory and accelerator-specific parameters, for example, the size of the images to be processed. Once configured from software, the accelerators are autonomous computational engines that communicate directly with off-chip main memory (DRAM) via direct-memory access (DMA). Once an accelerator completes the execution of its task, it notifies the program via an interrupt.

The architecture of an accelerator is specified with three synthesizable SystemC processes: `load()`, `compute()`, and `store()` (right-hand side of Fig. 1). The `load()` process is responsible for loading the input data, while the `store()` process produces the output data that are the results of the accelerator computation. These two processes manage the external interface that allows the accelerator to communicate with the main memory or another accelerator. Because the ESP socket redirects the memory requests at runtime to the particular component as specified by the software program (Section III), the choice of communicating with the main memory or another accelerator is entirely transparent to the `load()` and `store()` processes. The `compute()` process implements the particular computational kernel that needs to be accelerated. It can be easily modified to include multiple kernels if some accelerators are merged into a single accelerator. These processes are organized in a sequence of loops that implement the highly-parallel datapath of the accelerator.

Each accelerator has a private local memory (PLM) to hold the data during the computation [13]. A PLM has a multi-bank memory architecture managed by the accelerator explicitly and can be used to hold the data that are frequently accessed. The data in the PLM can be accessed quickly (even in a single clock cycle if there are enough ports to manage concurrent requests). In contrast, communicating with the main memory can take from tens to hundreds of cycles.

III. Software Programs and Data-Transfer Modes

To evaluate the two data-transfer modes: memory-based and p2p communication, we develop different programs that invoke the accelerators. Fig. 2 shows the data transfers performed by the accelerators when they communicate (a) through memory or (b) via p2p. In the figure, we assume that we execute only three accelerators: *A*, *B*, and *C*.

In the case of memory-based communication (Fig. 2 (a)), each accelerator loads and stores the data by using some shared memory (in DRAM) as a "buffer". Each accelerator operates independently by reading and storing the data in the shared memory. The accelerators need to be configured and synchronized from software to enable the correct exchange of data. On the other hand, in the case of p2p communication (Fig. 2 (b)), once read from memory by the first accelerator, the data can be passed along to the next accelerators until the last accelerator stores the final results to memory. The software needs to define the dependencies among the accelerators (as explained later), but their synchronization is inherently supported by the hardware.

A. Memory-Based Communication

When a software program invokes multiple accelerators, they typically exchange data by using memory-based communication. The program first allocates a shared region of memory that can be used by both the program and the various accelerators. Then, it copies the input data that need to be processed before proceeding with the invocation of the first accelerator. As explained in Section II, the accelerator loads the input data, performs the computation, and stores back the results in the shared region of memory. When it has completed the assigned task, it notifies the program, which in turn invokes another accelerator that operates similarly to the first.

We develop two software programs that invoke the accelerators by using memory-based communication: a single-thread program and a multi-thread program. We use them in Section V to evaluate the performance and the energy efficiency of memory-based communication.

Single-Thread Program (stm). Implementing the single-thread program is relatively straightforward. ESP provides an API that allows software developers to access hardware accelerators [8]. After defining the location of the input and output data for each accelerator in the main memory, it is sufficient to call the primitive `esp_run()` in the API for invoking each accelerator in sequence. By default, `esp_run()` spawns a new thread to invoke the accelerator. This extra thread is not needed for the case of the single-thread program and, therefore, we remove it to improve the performance and have a stronger baseline in our comparative analysis.

979-8-3503-0860-0/23/$31.00 ©2023 IEEE
Authorized licensed use limited to: Columbia University Libraries. Downloaded on March 07,2024 at 15:36:38 UTC from IEEE Xplore. Restrictions apply.

Multi-Thread Program (mtm). Implementing the multi-thread program is, however, more complicated. ESP does not natively support the synchronization of multiple accelerator invocations: this is left to the programmer. Therefore, we develop a software library to synchronize the executions of the accelerators. The sequential invocation of the accelerators can be improved by pipelining their executions over time. A software thread is spawned for each accelerator. The library uses a queue data structure to store the data items exchanged by the accelerators (these correspond to the tokens in the dataflow [14] and Petri Net [15] models of computation). The access to a queue is protected by a corresponding mutex. Upon completing its execution, an accelerator produces its output tokens that are enqueued in the input queues of the accelerators whose computation depends on this accelerator. An accelerator cannot start its execution until all the accelerators that generate its input tokens have completed theirs.

B. Point-to-Point Communication

The other main data-transfer mode is p2p communication [7]. In this case, the program allocates a region of memory and copies the input data for the first accelerator that needs to be executed. The program configures all the accelerators by specifying where the data need to be accessed (either from the main memory or from another accelerator) and their accelerator-specific parameters. The program also defines the execution dependencies of the accelerators. Then, all the accelerators are invoked. The first accelerator (the one that accesses the data from the main memory) starts executing, while the subsequent accelerators wait for the input data from the previous accelerators in the pipeline.

Naturally, p2p supports *implicit pipelining*. The synchronization of the accelerators is entirely handled in hardware, without requiring the software to pipeline and synchronize the execution of the accelerators explicitly.

P2P Program (p2p). In order to implement the p2p program, we augment the original ESP API to support complex communication patterns in a dataflow, such as fork/join and feedback loops. This allows us to call `esp_run()` a single time to invoke the executions of all the accelerators of the dataflow kernels and have them processing a sequence of many frames. In the invocation, we specify the configuration parameters for all the accelerators, including the source of the input data (main memory or another accelerator), the destination of the output data (main memory or another accelerator), how many copies of the outputs, and so on.

The p2p communication is implemented as a pull mechanism: each accelerator sends request to the uplink accelerators in the dataflow (one or more than one) that produce its input data. When `esp_run()` is called in p2p mode, those accelerators that read inputs data only from main memory start executing, while all the others enter a polling cycle that makes them periodically send input-data requests. This implicitly handles the synchronization among the accelerators. Without our enhancement of the original ESP API, it would be necessary to invoke `esp_run()` many times, one for each accelerator processing each frame, thus suffering a larger invocation overhead.

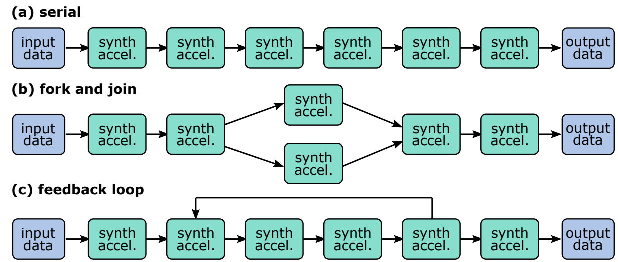

Fig. 3. Three main communication patterns in accelerator dataflows.

C. Programmability Considerations

The ability for accelerators to run concurrently is critical for performance. The choice of the data-transfer mode has substantial implication on concurrent programming with multiple threads. For the case of memory-based communication, we implement multiple threads with a library of functions that handles six main operations: create and join of the *pthreads*, lock and unlock of the *mutexes*, put and get of the *queues*. This requires writing approximately 800 lines of C code while considering the synchronization carefully among threads based on the topology of the dataflow application. This centralized communication control of programming effort is error-prone and leads to code that is difficult to debug.

On the contrary, the implementation of a multi-thread program utilizing p2p communication is a much simpler approach. Once the dependencies of the accelerators have been identified and specified in the configuration file, the process merely involves a single API call to initiate all accelerator invocations. Subsequent synchronization tasks are performed by the hardware itself without involving the CPU. Nevertheless, a drawback of this approach is that developers are required to modify the accelerators to ensure the smooth functioning of the overall workflow. For instance, in a 1-to-2 fork situation, it becomes necessary to duplicate the output of the upstream accelerator, enabling both downstream accelerators to receive a copy of the relevant data.

All the three mentioned programs (**stm**, **mtm** and **p2p**) are easy to scale up to accommodate more accelerators and execute a batch of many inputs.

IV. Target Application

Synthetic Accelerators. To first study the relationship between data-transfer modes and communication patterns in accelerator dataflow, we design *synth*, a synthetic hardware accelerator. This highly flexible hardware accelerator can be used as an actor to build dataflows, from a simple pipeline to a complex graph. The amount of input taken by the `load()` process and the amount of output data produced by the `store()` process can be both configured through the accelerator parameters by the software program. These allow us to tune the communication between accelerators and the main memory. In addition, the computation time in `compute()` of each *synth* can also be configured through the accelerator parameters. In this paper, we consider the three main communication patterns in dataflows shown in Fig. 3: (a) serial, (b) fork and join, (c) feedback loop. With these three patterns, we analyze the performance of the p2p

979-8-3503-0860-0/23/$31.00 ©2023 IEEE
Authorized licensed use limited to: Columbia University Libraries. Downloaded on March 07,2024 at 15:36:38 UTC from IEEE Xplore. Restrictions apply.

Table I
CHARACTERISTCS OF THE NIGHTVISION AND WAMI ACCELERATORS.

accelerator	% SW exec.	PLM (Byte)	LUTs	BRAMs	DSPs	Power (W)
NF	79.05	153,600	11,621	301	8	0.087
Hist	14.22	338,944	5,609	183	6	0.047
HistEq	6.73	415,744	12,222	381	9	0.103
debayer	8.86	135,232	19,874	35	70	0.069
grayscale	1.84	131,072	15,244	57	23	0.086
gradient	1.66	196,608	15,342	97	26	0.078
warp	25.82	131,096	49,381	71	0	0.094
matrix sub	1.72	196,608	17,424	97	7	0.082
steepest descent	3.84	524,288	28,164	321	0	0.131
hessian	18.88	393,360	23,011	195	0	0.106
matrix inv	0.03	288	27,305	7	0	0.066
sd update	7.54	458,776	24,796	227	0	0.111
matrix mult	0.04	192	19,381	7	0	0.056
matrix reshape	0.02	48	12,198	5	0	0.045
matrix add	0.01	72	14,828	4	0	0.053
change detection	29.74	131,072	32,199	497	310	0.344

Table II
SYSTEM CONFIGURATION FOR THE EXPERIMENTAL RESULTS.

Processor	CVA6, RISC-V, 64-Bit, 78 MHz
L1 Cache	ICache: 16KB, DCache: 32KB
Memory	2.5 GB DDR4-2400 memory
Accelerator Type	1 Synthetic, 3 Nightvision, and 13 WAMI
Evaluation Board	proFPGA Virtex UltraScale XCVU440
Operating System	Linux v4.20.0
HLS Tool	Cadence Stratus 20.25
Synthesis Tool	Xilinx Vivado 2019.2

communication and memory-based data-transfer modes across many different scenarios.

Nightvision and WAMI. In addition to the synthetic dataflows, we develop specialized accelerators for two distinct image processing applications: Nightvision [9] and Wide-Area Motion Imagery (WAMI) [16]. The Nightvision application serves the purpose of enhancing visibility in low-light or dark environments by improving the image contrast. Nightvision consists of three different kernels: noise filtering (*NF*), histogram (*Hist*), and histogram equalization (*HistEq*). These kernels are shown in Fig. 4 (a), which also showcases the fork/join pattern. We design an accelerator for each kernel.

WAMI processes the sequence of frames of an input video to detect changes in regions of interest, such as people or vehicles moving on the ground, while discarding the environmental noise, e.g. reflections. A software implementation of WAMI is available in the PERFECT Benchmark Suite [17]. As shown in Fig. 4 (b), WAMI consists of four main computational kernels: *Debayer*, *Grayscale*, *Lucas-Kanade*, and *Change-Detection*. We break the *Lucas-Kanade* into several image processing kernels to (i) further investigate the parallelism of the data transfer on a NoC-based SoC architecture and (ii) stress the two data-transfer modes in a scenario with complex dependencies (forks/joins and feedback loops).

We develop sixteen different hardware accelerators for each kernel in Nightvision and WAMI. Table I reports the main characteristics of these accelerators. The sizes of the PLMs

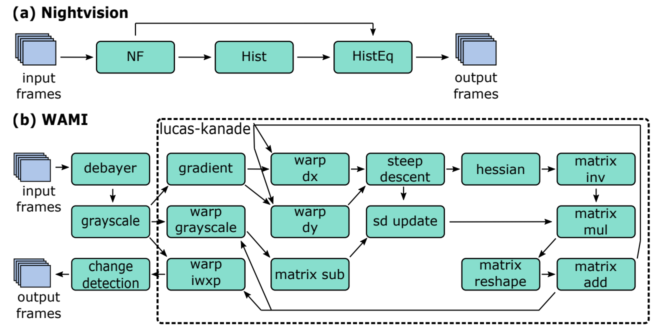

Fig. 4. Accelerator dataflow for the (a) Nightvision and (b) WAMI application.

Table III
PERFORMANCE COMPARISON OF SYNTHETIC DATAFLOWS.

(a) **serial**		**stm**	**mtm**	**p2p**
10 frames	*computation*	1.0	1.2	4.4
	communication	1.0	2.2	4.2
100 frames	*computation*	1.0	1.2	6.8
	communication	1.0	4.1	17.3
1000 frames	*computation*	1.0	1.2	7.2
	communication	1.0	4.3	25.0
(b) **fork and join**		**stm**	**mtm**	**p2p**
10 frames	*computation*	1.0	1.4	4.6
	communication	1.0	2.2	4.1
100 frames	*computation*	1.0	1.4	6.8
	communication	1.0	3.9	12.7
1000 frames	*computation*	1.0	1.4	7.2
	communication	1.0	4.2	16.6
(c) **feedback loop**		**stm**	**mtm**	**p2p**
10 frames	*computation*	1.0	1.2	5.1
	communication	1.0	2.2	3.6
100 frames	*computation*	1.0	1.2	7.0
	communication	1.0	3.7	9.9
1000 frames	*computation*	1.0	1.2	7.2
	communication	1.0	3.9	12.9

are based on the amount of data to be processed by each accelerator.

V. EXPERIMENTAL EVALUATION

We first evaluate the two data-transfer modes of the three different synthetic dataflows (Section V-A). Then, we complete an extensive evaluation of Nightvision and WAMI by measuring performance and energy efficiency (Section V-B) as well as the number of accesses to the off-chip memory (Section V-C). Finally, we do both a theoretical and an experimental analysis on the achievable speedup of the accelerators that we have developed for WAMI (Section V-D). For this analysis, we design several complete NoC-based SoCs with multiple accelerators, deploy them on an FPGA board, and collect results running software on top of Linux. Table II summarizes the common properties of our SoC.

A. Synthetic Dataflows

To evaluate the performance of the three synthetic dataflows of Section IV, we create two extreme scenarios: computation heavy and communication heavy. We arbitrarily assign 512 bytes for both `load()` and `store()`, and 1,048,576 cycles for `compute()` to represent a computation-bound scenario. For the communication-bound scenario, we pick 32,768 bytes

979-8-3503-0860-0/23/$31.00 ©2023 IEEE
Authorized licensed use limited to: Columbia University Libraries. Downloaded on March 07,2024 at 15:36:38 UTC from IEEE Xplore. Restrictions apply.

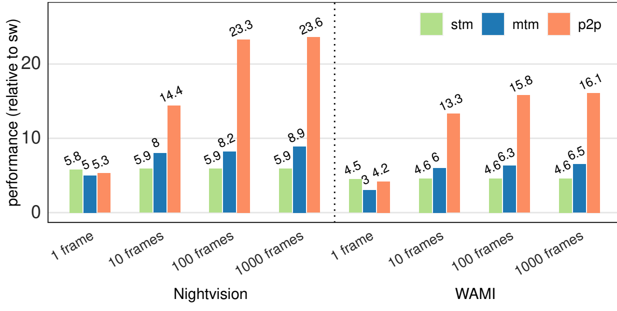

Fig. 5. Speedups: memory-based vs. p2p communication.

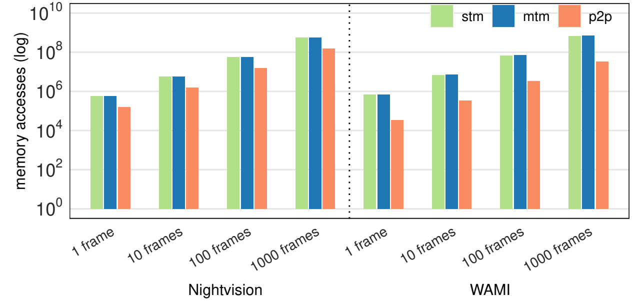

Fig. 6. Memory access savings with p2p communication.

of data to transfer and 16,384 cycles for computation. Table III reports the experimental results as speedups with respect to the single-threaded **stm**. We can observe that the multi-threaded processes (**mtm** and **p2p**) have better performance over **stm**, particularly when the accelerator of the application is communication-bound. Furthermore, **p2p** shows greater speedup comparing to **mtm** in all the cases, especially when the dataflow is communication-bound, as it optimizes the transfer of data between the different accelerators.

B. Performance and Energy Efficiency

Fig. 5 reports the results of the performance analysis of Nightvision and WAMI. We measure the performance in terms of throughput (calculated as frames per second) by varying the workload, which is defined as the total number of frames that are processed by the different programs. The results are normalized with respect to the software execution on the CVA6 core. We observe that, despite the overheads for accelerator invocation, the programs that use memory-based communication already outperform the CVA6 core. When the workload consists of processing only one frame, **stm** gives the best speedup compared to software execution because there is no need for pipelining, and thus having fewer software overheads results in better performance. In contrast, both **mtm** and **p2p** are slower because the overhead for spawning the software threads and handling their synchronization does not pay off when processing a single frame. As the number of frames increases, spawning multiple threads brings benefits while its overhead becomes negligible. In fact, **mtm** achieves better performance than **stm** thanks to the concurrent execution of multiple accelerators. Noticeably, **p2p** provides higher performance than memory-based communication when the number of frames is greater than one. For example, when invoking the sixteen accelerators to process a workload of WAMI that consists of a stream of 100 frames, **p2p** achieves a performance improvement of 15.8× compared to **sw**. This is mainly due to the reduction in the number of accesses to main memory compared to memory-based communication.

We also investigate the energy efficiency when executing the applications with different programs. From the post-synthesis report of Vivado, we obtain the dynamic power consumption of each tile, including CPU, memory, I/O, and accelerators. The ratio of the power dissipated by the SoC without accelerators divided by the power dissipated by the SoC with the accelerators is 0.91 for Nightvision and 0.66 for WAMI. The energy is defined as the product of power and latency. Therefore, the *energy efficiency gain* over the software execution is defined as the speedup times the ratio of the power. For example, when executing a workload of Nightvision that consists of 100 frames, the energy efficiency gain of **p2p** is 21.1× compared to **sw**. Overall, as the number of frames grows, the energy efficiency gain scales proportionally with the speedup of Fig. 5.

C. Accesses to the Off-chip Memory

To analyze the impact on the number of memory accesses, we use `esp_monitor()`. This function, made available in the ESP API, calls performance counters that keep track of the number of (read and write) accesses performed by each accelerator to the main memory. The results of this analysis are reported in Fig. 6, which shows the number of memory accesses performed by the accelerators when executing the Nightvision and WAMI application on workloads with different numbers of frames. We can observe that the number of accesses scales linearly with the number of frames, but in the case of **p2p**, the absolute value is significantly less. In fact, the utilization of **p2p** in the Nightvision leads to a reduction of memory accesses by a factor of 4 when compared to **stm** and **mtm**. Additionally, in the case of WAMI, the memory accesses is reduced by 20× due to the heavy data transactions between accelerators. This reduces the pressure on the memory subsystem, which is critically important for SoCs that run multiple applications.

D. Analysis on Achievable Speedup

After designing the WAMI accelerators, we want to confirm that we reach the maximum speedup at the application level. Amdahl's Law [18] provides a way to calculate the theoretical speedup of an entire application when a portion of it is parallelized. We develop sixteen simple variations of the program implementing the WAMI application. Each of these variations only invokes one accelerator to offload the execution of its corresponding kernel in hardware, while the rest of the kernels are executed in software on the CVA6 processor core. Each of the *experimental bars* in the diagram of Fig. 7 reports the value of the actual speedup measured on the FPGA while executing each of these variations of the WAMI application. Each of the *theoretical bars* in the diagram of Fig. 7 reports the value of the theoretical speedup for each accelerator. While a small degradation in speedup is to be expected, the actual speedup provided by each accelerator is very close to its

979-8-3503-0860-0/23/$31.00 ©2023 IEEE
Authorized licensed use limited to: Columbia University Libraries. Downloaded on March 07,2024 at 15:36:38 UTC from IEEE Xplore. Restrictions apply.

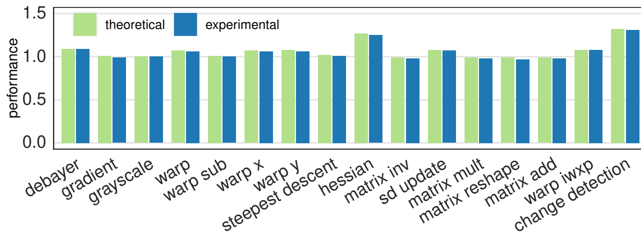

Fig. 7. Application-level speedup: experimental vs. theoretical results.

theoretical speedup. Although the performance speedups of these accelerators are limited when running standalone, their combined effect when running together is significant and it becomes very remarkable when they work in pipelining by exchanging data with p2p communication, as shown by the results presented in the previous sections.

E. Summary of Results and Lessons Learned

The main outcome of our analysis is a recommendation to use p2p communication instead of memory-based communication when designing an SoC that has multiple loosely-coupled accelerators that interact by exchanging large data amounts in a complex dataflow topology. Not only the p2p communication gives better performance through its on-chip data movement and implicit pipelining, but it also leads to smaller energy dissipation with respect to the memory-based communication mode. Next, we summarize the most important results obtained from our experiments:

- The **p2p** program has significantly better performance comparing to the memory-based programs when the dataflow is communication heavy.
- For two image processing applications, the **stm** program provides the best performance when only one frame is executed. The software overhead of **stm** is less than the others because spawning threads and software synchronization are not necessary.
- In the case of multiple frames, the **mtm** program achieves better performance than the **stm** program thanks to the parallelism that can be exploited.
- Overall, the **p2p** program outperforms **stm** and **mtm** both in terms of performance and energy efficiency.
- For the WAMI application, the measured speedups approach the theoretical maximum speedups.

In addition, we have some results that are not discussed in the previous sections due to limited space:

- In the case of memory-based communication, using two or more memory controllers can improve the performance (depending on the application). For **p2p** having more memory controllers is less important because the number of accesses to off-chip memory is reduced.
- The exploration of multiple SoC configurations by varying the positions of the accelerators does not substantially change the results of our analysis.

VI. Related Work

In an SoC, on-chip communication between CPUs, GPUs, accelerators, and memories is costly. Approaches like Wi-HetNoC by Choi et al. [19] and WNoC by Guirado et al. [20] address this problem by improving the NoC. Memory-based communication is the standard mechanism to allow accelerators to exchange data. Conversely, p2p communication is mostly used to allow different components of the same accelerator to communicate efficiently. For example, Cong et al. [21] propose an architecture with "accelerator building blocks" that can be freely composed to create more complex accelerators. Shao et al. [22] propose a multi-chip architecture to scale the performance of deep learning inference. Each chiplet has various processing elements (PE) that can communicate directly. However, p2p communication is more rarely used for accelerator communication on an NoC or a bus. Giri et al. [7] propose to use p2p communication to improve the performance of accelerators for machine learning and computer vision. Cong et al. [23] describe "accelerator chaining" as a technique for accelerator-to-accelerator communication, which is implemented by allowing the DMA controllers of the accelerators to communicate with each other.

Table IV
Comparison with Related Work

	Processing Unit Type	FPGA	OS	multi-thread	p2p	feedback loop
our work	17 ACCs	✓	✓	✓	✓	✓
[19]	28 GPUs	-	-	-	-	-
[20]	256 PEs	-	-	-	✓	-
[21]	15 ABBs	-	✓	✓	-	-
[22]	36 Chiplets	✓	-	-	✓	-
[7]	3 ACCs	✓	✓	-	✓	-
[23]	16 ACCs	-	✓	✓	✓	-

Table IV summarizes the experimental setups of these related works. Compared to these, the setup for our analysis is the only one that is based on FPGA-based prototypes of complete SoCs that execute complex multi-threaded applications (with feedback loops) running on top of the Linux operating system and exchanging data through p2p communications. We are not aware of any prior published analysis that compares memory-based and p2p communication for a heterogeneous SoC architecture executing a complex application that invokes many loosely-coupled accelerators.

VII. Conclusions

In this paper, we present a detailed comparative analysis of two data-transfer modes among many loosely-coupled accelerators in heterogeneous NoC-based SoC architectures: memory-based and p2p communication. We demonstrate the concept by using a variety of synthetic benchmarks to get a preliminary evaluation. Then we consider Nightvision and WAMI, real-world complex image processing applications [16], [17]. We develop sixteen distinct accelerators with HLS. For both applications, we integrate the accelerators into a Linux-capable SoC. We deploy and test the SoC on an FPGA development board to perform our experiments. Our analysis shows that p2p communication outperforms memory-based communication in terms of performance and energy efficiency. In addition, it simplifies the development of software by hiding the synchronization details from the programmers. We plan an open-source release of all the software and hardware artefacts in time for HPEC.

979-8-3503-0860-0/23/$31.00 ©2023 IEEE
Authorized licensed use limited to: Columbia University Libraries. Downloaded on March 07,2024 at 15:36:38 UTC from IEEE Xplore. Restrictions apply.

ACKNOWLEDGMENTS

This research was developed, in part, with funding from the Defense Advanced Research Projects Agency (DARPA), and in part with funding from the Army Research Office under Grant Number W911NF-19-1-0476. The views, opinions and/or other findings expressed are those of the authors and should not be interpreted as representing the official views or policies (either expressed or implied) of the Department of Defense, the Army Research Office, the National Science Foundation, or the U.S. Government. Distribution Statement "A": Approved for Public Release, Distribution Unlimited. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] W. J. Dally *et al.*, "Domain-Specific Hardware Accelerators," *Communications of ACM*, vol. 63, no. 7, 2020.

[2] J. Cong *et al.*, "Accelerator-rich Architectures: Opportunities and Progresses," in *Proc. of the Design Automation Conference (DAC)*, 2014.

[3] T. Oguntebi *et al.*, "GraphOps: A Dataflow Library for Graph Analytics Acceleration," in *Proc. of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[4] P. Srivastava *et al.*, "PROMISE: An end-to-end Design of a Programmable Mixed-Signal Accelerator for Machine-Learning Algorithms," in *Proc. of the International Symposium on Compute Architecture (ISCA)*, 2018.

[5] O. Reiche *et al.*, "Generating FPGA-based Image Processing Accelerators with Hipacc," in *Proc. of the International Conference On Computer Aided Design*, 2017.

[6] E. G. Cota *et al.*, "An Analysis of Accelerator Coupling in Heterogeneous Architectures," in *Proc. of the Design Automation Conference (DAC)*, 2015.

[7] D. Giri *et al.*, "ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning," in *Proc. of the ACM/IEEE Design, Automation and Test in Europe Conference (DATE)*, 2020.

[8] P. Mantovani *et al.*, "Agile SoC Development with Open ESP," in *Proc. of the International Conference On Computer Aided Design*, 2020.

[9] Embedded Scalable Platform (ESP), www.esp.cs.columbia.edu.

[10] F. Zaruba *et al.*, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 27, no. 11, 2019.

[11] L. P. Carloni, "The case for embedded scalable platforms," in *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 2016.

[12] D. Black *et al.*, *SystemC: From the Ground Up, Second Edition*. Springer, 2009.

[13] M. J. Lyons *et al.*, "The Accelerator Store: A Shared Memory Framework for Accelerator-based Systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, 2012.

[14] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[15] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[16] R. Porter *et al.*, "Wide-Area Motion Imagery," *IEEE Signal Processing Magazine*, vol. 27, no. 5, 2010.

[17] K. Barker *et al.*, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*, 2013.

[18] M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *afips*, 1967.

[19] W. Choi *et al.*, "On-chip communication network for efficient training of deep convolutional networks on heterogeneous manycore systems," *IEEE Transactions on Computers*, vol. 67, no. 5, pp. 672–686, 2018.

[20] R. Guirado *et al.*, "Understanding the impact of on-chip communication on dnn accelerator performance," in *International Conference on Electronics, Circuits and Systems (ICECS)*, 2019, pp. 85–88.

[21] J. Cong *et al.*, "Charm: A composable heterogeneous accelerator-rich microprocessor," in *Proc. of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2012.

[22] Y. S. Shao *et al.*, "Simba: Scaling Deep-learning Inference with Multi-chip-module-based Architecture," in *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2019.

[23] J. Cong *et al.*, "Architecture Support for Accelerator-rich CMPs," in *Proc. of the Design Automation Conference (DAC)*, 2012.

979-8-3503-0860-0/23/$31.00 ©2023 IEEE
Authorized licensed use limited to: Columbia University Libraries. Downloaded on March 07,2024 at 15:36:38 UTC from IEEE Xplore. Restrictions apply.