

FPGAs memory synchronization and performance evaluation using the open computing language framework

Abedalmuhdi Almomany, Amin Jarrah

Department of Computer Engineering, Hijjawi Faculty for Engineering Technology, Yarmouk University, Irbid, Jordan

Article Info

Article history:

Received Mar 24, 2023

Revised Sep 1, 2023

Accepted Sep 11, 2023

Keywords:

Field programmable gate arrays

Memory

Mutex

Open computing language

Synchronization

ABSTRACT

One advantage of the open computing language (OpenCL) software framework is its ability to run on different architectures. Field programmable gate arrays (FPGAs) are a high-speed computing architecture used for computation acceleration. This work develops a set of eight benchmarks (memory synchronization functions, explained in this study) using an OpenCL framework to study the effect of memory access time on overall performance when targeting the general FPGA computing platform. The results indicate the best synchronization mechanism to be adopted to synthesize the proposed design on the FPGA computation architecture. The proposed research results also demonstrate the effectiveness of using a task-parallel model approach to avoid using high-cost synchronization mechanisms within proposed designs that are constructed on the general FPGA computation platform.

This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.



Corresponding Author:

Abedalmuhdi Almomany

Department of Computer Engineering, Hijjawi Faculty for Engineering Technology, Yarmouk University
Irbid, Jordan

Email: emomani@yu.edu.jo

1. INTRODUCTION

A field programmable gate array (FPGA) is an integrated circuit that programmers can configure many times to achieve their goals [1], [2]. FPGAs include many low-level operations, such as shifts and additions. Usually, the intel FPGA incorporates several resources, such as random access memory (RAM) blocks and digital signal processors (DSPs), to perform various complex arithmetic functions and look-up tables (LUTs) [2]. LUTs are also used to implement several functions Figure 1. However, multiple LUTs can be combined to implement more complex functions. The DE5 (Stratix V) FPGA device is used in the present study. The adaptive logic module (ALM) resource allows a wide range of functions to be implemented efficiently. Each ALM contains several function units. The block diagram for ALM is shown in Figure 1 [3]. FPGA as a reconfigurable architecture provides better reconfiguration in which bit-level configuration is performed [4]–[7].

The flexible parallel hardware architecture is guaranteed by FPGA technology. It includes many logic components, such as adders, multipliers, and comparators. It also includes a lot of DSPs, LUTs, clocks, configurable I/O, memories, and wired connections between these components. Because these components operate concurrently, allowing for a large amount of computation to be done independently at once, we can achieve a high level of parallelization with this FPGA implementation [8]–[10]. Many semiconductor companies, including Xilinx, Altera, Actel, Lattice, Quick Logic, and Atmel, produced and improved FPGA.

Three types of FPGA-based spatially reconfigurable computing environments are now commercially available. They include commodity FPGA-based accelerator cards, stand-alone system on programmable chip (SoPC) environments, and cloud-based spatially reconfigurable platforms. Commodity FPGA-based

accelerator cards are the most common commercially available spatially reconfigurable computing environment and were chosen as the computing environment for this research. These cards are designed to be incorporated within a standard central processing unit (CPU)-based computing system as an add-on low-profile peripheral component interconnect express (PCIe)-based daughter card. They incorporate one or more high-end FPGAs and significant amounts of multi-banked double data rate synchronous dynamic random-access memory (DDR SDRAM) physical memory (8 GB to 32 GB), which are local to the card. The cards often also contain high speed network ports and flash memory that can be used to load default configurations within the FPGAs. Stand-alone SoPC configurations are also quite prevalent at the time of this writing. SoPC configurations also include high-end FPGAs that often contain built-in embedded CPU processing cores. SoPC configurations also contain varying amounts of DDR SDRAM physical memory and a host of I/O interfaces. SoPC configurations differ primarily from accelerator cards in that they are not designed to augment an existing CPU system [11].

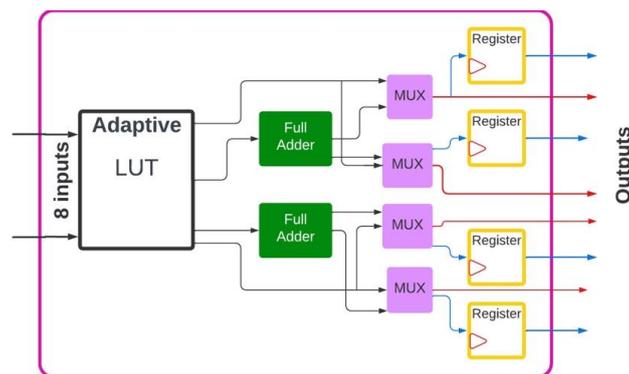


Figure 1. ALM block diagram

FPGA platforms are now becoming available on the cloud [12]. FPGA-based resources are accessible using OpenStack virtual machine environment, which provides tools for cloud resource management [13], [14]. In a related study, a framework that integrated Xilinx FPGAs into the cloud based on OpenStack showed great efficiency and scalability upon hosting multiple processes and virtual machines (VMs) [15]. FPGAs are also accessible as an F1 compute instance on the Amazon Elastic Cloud, where each instance contains up to eight FPGAs [16]. These instances could be used effectively to create a customized user's design in a wide range of commercial and scientific applications.

Commodity FPGA-based technology has several issues, though, which must be carefully considered. One important issue is that while it is possible to create specialized functional units and data paths that closely mirror the structure of the application, the FPGA resources that are required are usually only a fraction of those required to implement the application in its most optimized form. Thus, the intelligent time sharing of these resources is mandatory and is the system-wide focus of what is a very complex optimization problem. The time it takes to configure an FPGA is large compared to the time taken to perform a base operation. The reconfiguration time for large FPGAs can be in the order of seconds, whereas the internal clock speed can be greater than 300 MHz. This means that internal FPGA resource trade-offs may have to be made that will decrease the utilization and increase time sharing to reduce the number of FPGA reconfigurations required. Another possibility is to utilize partial reconfigurability, which is supported by most modern FPGAs. Partially reconfigurable devices allow for the logic functionality of a subsection of its programmable resources to be reconfigured without interrupting the operation of the other portion of the reconfigurable logic. Unfortunately, this feature is often poorly utilized. Another major issue is the time it takes to synthesize a design. The fine-grain complexity of FPGAs can result in extremely long design compilation times, which can take hours or days to complete. This problem is most apparent when the FPGA-based resources needed by the application get close to the actual resources that are present on the system. It becomes imperative in such cases that the high-level design environment allow for the functionality of the design to be verified quickly before it goes through this lengthy process. Fortunately, high-level synthesis environments, such as the open computing language (OpenCL) support an emulator mode where emulation can be performed on the CPU. Still, this constraint precludes the use of just-in-time compilation techniques that are possible in GPU and some CPU applications. This means all modules that are to be executed on the FPGA must therefore be progenerated in an offline manner [11].

OpenCL is used to harness the benefits of multi-processing elements. The wide variety of platforms that can be used for OpenCL makes it an attractive choice for heterogeneous systems in which several computations can be distributed among different computation architecture elements [10]. The OpenCL code written to run on the FPGA is implemented as a kernel, and the kernel code is compiled using the intel offline compiler (IOC). The kernel could be executed with one or multiple items (threads) [11], the choice depends on the code characteristics, and the goal is to achieve the highest degree of parallelism. The OpenCL standard naturally enables the ability to specify parallel algorithms to be implemented on FPGAs, at a far higher degree of abstraction than hardware description languages (HDLs) like VHDL or Verilog, in addition to providing a portable model.

Because FPGAs are not just processors using a typical software design flow, targeting FPGAs from OpenCL presents some special challenges. The FPGA architecture differs significantly from the typical platforms (such as CPUs and GPUs) that OpenCL implementations target. For instance, FPGA makers recently debuted programmable system-on-chips (SoCs), in which a SoC is connected to FPGA fabric to create a customizable platform for an embedded system environment, like the Zynq platform [17]. Additionally, there is plenty of room for OpenCL to adjust to this kind of platform due to the long compilation time, the programmable nature of FPGAs, and the capability for partial reconfiguration [18].

The notion of pipeline parallelism is an important concept of the IOC, which synthesizes the high-level abstracted OpenCL code on the target FPGA device. Pipelined architectures allow data to pass through various stages before the proposed result is attained. The IOC creates a customized pipeline architecture based on the proposed kernel code [19]. Figure 2 illustrates how the IOC creates the pipeline architecture for a given kernel code. Several optimization techniques, such as shift-register and loop unrolling can feasibly be used to create a powerful design architecture. Loop unrolling allows more operations to be performed per clock cycle by duplicating the necessary function units. Meanwhile, the shift-register technique helps reduce the dependency between consecutive statements, thereby reducing the number of stall cycles. The intel FPGA compiler provides several tools to modify the design's performance and solve possible critical issues that may reduce the effectiveness of the architecture before synthesizing the proposed FPGA device [20].

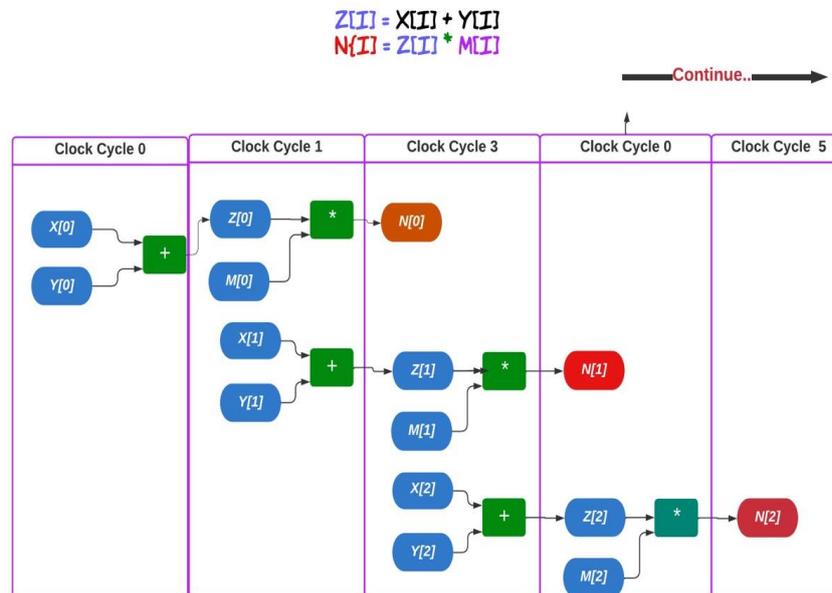


Figure 2. Illustration of the pipeline architecture created by the IOC for a given kernel code written in OpenCL

The code written in OpenCL to perform the acceleration process on the intel FPGA architecture has two parts. The first part is the host code, which is written in standard C/C++ code and compiled using the gcc/g++ compiler [8]. The host code is responsible for starting the acceleration process, deciding what data should be transferred between the host and the FPGA global memory, and deciding which parts of the code should be accelerated on the FPGA device. The overall host code is a sequence of steps taken before and after the kernel code is launched. The second part is the kernel code, which is implemented using the OpenCL application programming interfaces (APIs) and compiled with the IOC to generate the device executable code [10].

This work is based on the study titled “efficient synchronization primitives for GPU” [21], in which a set of eight benchmarks was developed using a compute unified architecture (CUDA) software framework to study the effect of memory access time on overall performance. The present study intends to replicate this work and study the effect of several synchronization functions on the overall performance when targeting the general FPGA computing platform. Several synchronization techniques can be used when multiple threads cooperate to perform related tasks and have access to commonly shared variables. The barrier is a common synchronization technique that allows all synchronized threads to stop at a certain point; then, once the last thread reaches this point, all threads resume their execution [22]. Mutex is another synchronization mechanism that allows only one thread to execute in the critical section to avoid the race condition issue. A binary semaphore is similar to a mutex mechanism when a single resource exists [23]. Meanwhile, the counting semaphore controls access to shared resources when there are multiple instances of a resource, and each instance cannot be used by more than one thread concurrently [24].

Although these synchronization techniques must be used to control access to shared resources, they introduce a significant time overhead due to the overall waiting time required for each thread to access the critical resource [25]. The present research recommends a technique that could have less overhead than other techniques when implemented on the FPGA platform. Several benchmarks are developed to analyze the memory access time of various implementations, thereby achieving the study’s purpose. These benchmarks are classified as atomic or non-atomic in the first layer and as having high or low contention in the second layer.

Finally, for read or write memory operations, we define atomic access as the access of only one thread to a distinct memory location, such that atomic accesses to the same memory location must be serialized. High contention means that all threads will access the same memory location. Meanwhile, in low contention instances, we generate multiple memory locations separated by at least 64 bytes, and there is a minimal chance that two or more threads will access the same memory location [21]. Lastly, in this design, we assume that each workgroup has 128 work items. Only one thread (the master thread) is given access to memory; for simplicity, thread zero is the master thread.

2. RESEARCH EXPERIMENTAL ENVIRONMENT

All benchmarks are compiled using the GCC V4.4 and the intel FPGA compiler V13.1, which are compatible with the Linux Centos operating system. The target FPGA used is the DE5 Stratix V device (5SGXEA7N2F45C2). This board contains enough resources, including 234 K ALMs, more than 250 DSP blocks, and 2.6 K RAM blocks to synthesize the user’s code in various heavy computation applications. The host CPU and the target board are connected via a PCIe connection, which enables extremely rapid data transfers between the processing units.

3. AVAILABLE RESOURCES VS THROUGHPUT TRADE-OFFS

Each FPGA contains a countable number of specific resources, such as ALMs, memory blocks, and DSPs. The proposed FPGA is usually connected to the host machine using the PCIe interface [6]. Each kernel is translated into a proposed hardware circuit using a fixed amount of resources. Typically, all kernels are combined into a single.cl (device code) file. While it takes only microseconds to milliseconds to run the kernel on the proposed FPGA (depending on the synthesized design), the overhead associated with switching the kernel during runtime is extremely large. The experiment was run 100 times to determine that it took approximately 1.612 seconds to configure the device at runtime. This outcome indicates that the configuration time is significant in most cases.

Another factor to consider is the additional resource consumption associated with using a high-level abstract OpenCL programming tool. Experiments demonstrate that approximately 16% of the ALMs, 11% of memory blocks, 3% of the total memory bits, and 53,893 registers are consumed to implement a blank (empty code) kernel. The extra resource overhead shown in Table 1 can be avoided by combining multiple kernels into a single file. Table 1 summarizes multiple kernels of vector addition, where the kernel is duplicated up to five times in a single file. Column 2 shows the resource usage of the blank kernel, column 3 shows the resource usage of a single vector addition kernel, and columns 4 to 7 show the resource usage of two, three, four, and five vector addition kernels. The experiment demonstrates the overhead associated with using a high-level abstracted OpenCL tool.

Loop unrolling can enhance performance by running several loop iterations in each clock cycle. However, the duplication function unit required to implement the loop unrolling technique consumes more resources. As such, the loop unrolling factor depends mainly on the number of resources available. Because of hardware limitations, we cannot fully unroll the loop in this work. Therefore, the loops in all benchmarks are unrolled 256 times. The same is true for different mutex implementations; all implementations are unrolled 10 times.

Table 1. FPGA resource usage for a single and multiple vector addition kernel

Resources/kernel	Blank Kernel	Vadd-1 kernel	Vadd-two kernels	Vadd-three kernels	Vadd-four kernels	Vadd-five kernels
Logic utilization (ALMs)	16%	20%	23%	26%	29%	31%
Total registers	53,893	72,397	84,439	97,485	106,650	115,269
M20k blocks (RAM blocks)	11%	14.72%	17.38%	20%	21.68%	22.62%
Total block memory bits	3%	3%	3%	4%	4%	4%

4. PROPOSED METHOD AND THE DEVELOPED BENCHMARKS

A set of eight benchmarks is created and compiled with the IOC to test the performance of FPGA memory systems. The benchmarks are classified as atomic or non-atomic, contentious or non-contentious, and read or write. For an atomic memory access operation, only one thread can access the desired memory location at a time, and no other thread can access the same memory location concurrently. In cases of contentious access, all threads access the same memory location, whereas in non-contentious access cases, different threads access different memory locations. Threads are divided into work groups, each of which contains 128 threads. However, only one thread in each workgroup (the first and master thread) can access the memory. Each master thread performs 1,024 memory access operations, which can be read or written. The “atomicadd” operation is used to implement the atomic read, and “atomicexchange” is used to implement the write operation. All benchmark loops are unrolled 256 times; this unrolling factor number is based on the available resources on the target FPGA to synthesize the proposed design.

Table 2 shows that atomic operations need more time to execute. Reducing the number of atomic operations will enhance the performance significantly. The effects of contention are not marked. The computing unit is saturated by running eight workgroups, each comprising 128 threads. Only the master thread can access the desired memory location.

Table 2. Benchmarks execution times for 1,000 memory operations, measured in milliseconds

Parameter	Read (ms)	Write (ms)	Average access time (ms)
Atomic contention	545	569	557
Atomic non-contention	422	428	425
Contention volatile	64	71	68
Non-contention volatile	68	73	71

The average execution times of various memory read/write operations are shown in Figure 3, and these are normalized to the execution time of a contention volatile memory operation. Figure 3 also shows the effects of atomic operations on memory access operations, which may increase the memory access time by more than eight times. However, the task-parallel model is more commonly used to construct the proposed design on the FPGA platform. The intel FPGA compiler has the capability to create an effective pipeline design where data can be shared among multiple loop iterations; this reduces the overall dependencies and the high cost of using several synchronization mechanisms.

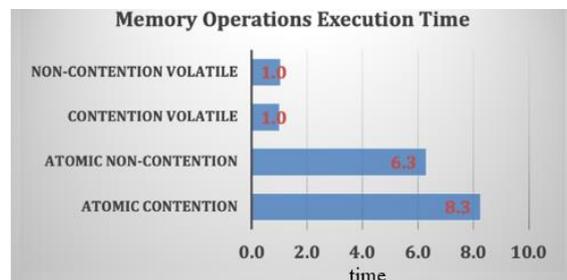


Figure 3. Average memory execution times normalized to the execution time of a contention volatile memory operation

5. MUTEX IMPLEMENTATION AND RESULTS DISCUSSION

After studying memory access benchmarks, several possible implementations of mutex are developed and tested on the intel FPGA architecture. All proposed algorithms perform atomic memory access, and only the master thread has access to memory. These suggested implementations [21] are described.

- a. Spinning: in this implementation, the target thread is in the waiting state until the status of the proposed memory location is changed. Two operations are considered here: i) lock function: the memory location is continually accessed using atomicexchange, which always returns the old value of the lock. If the returned value of the lock is 0, the thread can access the critical section; otherwise, it will continuously perform atomicexchange until it is granted access to the critical section; and ii) unlock function: the critical section is released by assigning a lock value of 0 using atomicexchange. This method is easy to implement; however, threads are not necessary to access critical sections in the same order in which they arrive (not first come first served (FCFS)).
- b. Backoff: the target thread continues doing non-useful work before getting access to the resource, two operations are carried out: i) lock function: the thread tries to gain access to the critical section if it is free. Otherwise, the thread sleeps for a certain time based on the thread group ID. This time increases after each trial until it reaches the maximum value, which is determined during the compilation time. This value is assigned to the minimum value if the incremented value is greater than the maximum value. This process is repeated until the thread accesses the critical section; and ii) unlock function: the unlock function assigns a lock value of 0 (nonatomic operation).
- c. Fetch and add using backoff: a well-known instruction supported by many processors to introduce an effective mutex implementation. The backoff is employed here to let the thread wait if the resource is not available, two operations are implemented here: i) lock function: each thread that should gain access to the critical section takes a ticket (the first variable), which is a number based on the thread's arrival order. The thread can access the critical section only if the value of the ticket is equal to the value of the turn (the second variable). If the ticket value is not equal to the turn value, the thread uses the backoff algorithm to sleep for a certain period of time; and ii) unlock function: increment the turn value (nonatomic operation).
- d. Fetch and add using sleeping: same as in "fetch and add using backoff", but with the sleeping technique is used instead of backoff to implement the thread waiting: i) lock function: this function is the same as that described in the fetch and add using the backoff algorithm, but if the ticket value is not equal to the turn value, the thread continuously polls the variables' memory locations to check if the equality condition is satisfied; and ii) unlock function: increment the turn value (nonatomic operation).

Several experiments with varying numbers of thread blocks are carried out to compare the performance of these algorithms. The performance of each algorithm is evaluated by measuring the number of memory operations completed per second. Table 3 shows the experimental results, which demonstrate that the highest throughput is achieved using the spinning implementation of mutex. Values represent millions of memory operations per second on the intel DE5 FPGA device. In this case, the target platform is a general intel FPGA device.

Table 3. The number of operations completed per second $\times 10^6$. Spinlock is the preferred implementation, and the fetch and add using the backoff algorithm has the lowest throughput.

Number of blocks	Spinning	Backoff	Fetch and add using Backoff	Fetch and add using sleeping
1	1.85	0.83	0.61	0.81
2	2.63	1.00	0.54	1.23
3	3.00	0.95	0.56	1.24
4	2.72	0.96	0.51	1.25
5	2.94	0.98	0.59	1.26
6	3.11	1.03	0.53	1.24
7	2.90	1.16	0.52	1.16
8	3.03	1.26	0.51	1.10
9	3.14	1.32	0.51	1.12
10	2.99	1.32	0.51	1.11

The preferred implementation is that which uses the fewest hardware resources. Table 4 shows some common resources used for each algorithm. The proposed synthesized architecture of the spinning algorithm consumes fewer resources than other algorithms. For all algorithms, each loop iteration contains 100 memory operations, and each operation has lock and unlock functions.

Table 4. Common hardware recourses used in each algorithm

Parameter	Spinning	Backoff	Fetch and add using backoff	Fetch and add using sleeping
LUTs (using parentage)	25%	67%	65%	38%
Registers	86.2 K	300.7 K	305.8 K	143.4 K
Total memory blocks (using parentage)	4%	12%	18%	5%

6. CONCLUSION

Several memory-access-based benchmarks are developed to study the effect of common synchronization techniques on the overall performance of the proposed synthesized design constructed on the intel FPGA platform. These benchmarks are developed using the abstracted high-level OpenCL programming tool. The results demonstrate that using atomic operations in the synthesized design leads to significant reductions in performance. Therefore, the task-parallel model, which improves the efficiency of the created design by generating an effective pipeline architecture, is a favorable choice when extra atomic operations are used. The present study also investigates several implementations of the widely used mutex synchronization mechanism and determines which implementation could be adopted by the proposed design to maximize the number of memory operations performed per second.

ACKNOWLEDGEMENTS

The Deanship of Scientific Research at Yarmouk University funded this work with Grant Number 43/2023. It is also supported by the intel FPGA University Program Ticket nos. LR4043 and BR 11211.

REFERENCES

- [1] A. Almomany, A. Jarrah, and A. Al Assaf, "FCM clustering approach optimization using parallel high-speed intel FPGA technology," *Journal of Electrical and Computer Engineering*, vol. 2022, pp. 1–11, 2022, doi: 10.1155/2022/8260283.
- [2] A. Abedalmuhdi, B. E. Wells, and K.-I. Nishikawa, "Efficient particle-grid space interpolation of an FPGA-accelerated particle-in-cell plasma simulation," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2017, pp. 76–79. doi: 10.1109/FCCM.2017.63.
- [3] V. J. K. K. Sonti, Y. Varthamanan, and D. Sivaraman, "Performance analysis of enhanced adaptive logic module for high performance FPGA architecture," *Research Journal of Applied Sciences, Engineering and Technology*, vol. 9, no. 3, pp. 215–223, Jan. 2015, doi: 10.19026/rjaset.9.1397.
- [4] A. Devrari and A. Kumar, "Reconfigurable linear feedback shift register for wireless communication and coding," *International Journal of Reconfigurable and Embedded Systems (IJRES)*, vol. 12, no. 2, p. 195, Jul. 2023, doi: 10.11591/ijres.v12.i2.pp195-204.
- [5] G. N. Chiranjeevi and S. Kulkarni, "Image processing using a reconfigurable platform: pre-processing block hardware architecture," *International Journal of Reconfigurable and Embedded Systems (IJRES)*, vol. 10, no. 3, pp. 230-236, Nov. 2021, doi: 10.11591/ijres.v10.i3.pp230-236.
- [6] G. R. Reddy, C. Perumal, P. Kodali, and B. V. Rajanna, "Design and memory optimization of hybrid gate diffusion input numerical controlled oscillator," *International Journal of Reconfigurable and Embedded Systems (IJRES)*, vol. 12, no. 1, pp. 78-86, Mar. 2023, doi: 10.11591/ijres.v12.i1.pp78-86.
- [7] D. K. Siddaraju and R. Munibyrapa, "Design and performance analysis of efficient hybrid mode multi-ported memory modules on FPGA platform," *International Journal of Reconfigurable and Embedded Systems (IJRES)*, vol. 11, no. 2, pp. 115–125, Jul. 2022, doi: 10.11591/ijres.v11.i2.pp115-125.
- [8] A. Almomany, W. R. Ayyad, and A. Jarrah, "Optimized implementation of an improved KNN classification algorithm using Intel FPGA platform: Covid-19 case study," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 6, pp. 3815–3827, Jun. 2022, doi: 10.1016/j.jksuci.2022.04.006.
- [9] W. Andre, "Efficient adaptation of the Karatsuba algorithm for implementing on FPGA very large scale multipliers for cryptographic algorithms," *International Journal of Reconfigurable and Embedded Systems (IJRES)*, vol. 9, no. 3, pp. 235-241, Nov. 2020, doi: 10.11591/ijres.v9.i3.pp235-241.
- [10] A. Almomany, A. Al-Omari, A. Jarrah, M. Tawalbeh, and A. Alqudah, "An OpenCL-based parallel acceleration of aSobel edge detection algorithm using IntelFPGA technology," *South African Computer Journal*, vol. 32, no. 1, Jul. 2020, doi: 10.18489/sacj.v32i1.749.
- [11] A. M. Almomany, "Efficient OpenCL-based particle-in-cell simulation of auroral plasma phenomena within a commodity spatially reconfigurable computing environment," Ph.D. dissertation, Dept. Elect and Comp. Eng., University of Alabama in Huntsville, USA, 2017.
- [12] M. Leeser, S. Handagala, and M. Zink, "FPGAs in the Cloud," *Computing in Science & Engineering*, vol. 23, no. 6, pp. 72–76, Nov. 2021, doi: 10.1109/MCSE.2021.3127288.
- [13] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "FPGAs in the cloud: booting virtualized hardware accelerators with OpenStack," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 109–116. doi: 10.1109/FCCM.2014.42.
- [14] S. S. Sahasrabudhe and S. S. Sonawani, "Comparing openstack and VMware," in *2014 International Conference on Advances in Electronics Computers and Communications*, 2014, pp. 1–4. doi: 10.1109/ICAEECC.2014.7002392.
- [15] N. Tarafdar, N. Eskandari, T. Lin, and P. Chow, "Designing for FPGAs in the cloud," *IEEE Design & Test*, vol. 35, no. 1, pp. 23–29, Feb. 2018, doi: 10.1109/MDAT.2017.2748393.
- [16] L. Kumar, P. Pooja, and P. Kumar, "Amazon EC2: (Elastic compute cloud) overview," in *Proceedings of Integrated Intelligence Enable Networks and Computing*, 2021, pp. 543–552. doi: 10.1007/978-981-33-6307-6_54.
- [17] N. Jacob et al., "Securing FPGA SoC configurations independent of their manufacturers," in *2017 30th IEEE International System-on-Chip Conference (SOCC)*, Sep. 2017, pp. 114–119. doi: 10.1109/SOCC.2017.8226019.
- [18] V. Mirian and P. Chow, "UT-OCL: an OpenCL framework for embedded systems using xilinx FPGAs," in *2015 International Conference on ReConfigurable Computing and FPGAs (ReConFig)*, 2015, pp. 1–6. doi: 10.1109/ReConFig.2015.7393366.
- [19] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, "OpenCL-based FPGA-platform for stencil computation and its optimization methodology," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1390–1402, 2017, doi: 10.1109/TPDS.2016.2614981.
- [20] H. M. Waidyasooriya, M. Hariyama, and K. Uchiyama, *Design of FPGA-based computing systems with OpenCL*. Cham: Springer International Publishing, 2018. doi: 10.1007/978-3-319-68161-0.

- [21] J. A. Stuart and J. D. Owens, "Efficient synchronization primitives for GPUs," *Prepr. arXiv.1110.4623*, 2011, [Online]. Available: <http://arxiv.org/abs/1110.4623>
- [22] S. Xiao and W. Feng, "Inter-block GPU communication via fast barrier synchronization," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Apr. 2010, pp. 1–12. doi: 10.1109/IPDPS.2010.5470477.
- [23] A. Carminati, R. S. de Oliveira, and L. F. Friedrich, "Implementation and evaluation of the synchronization protocol immediate priority ceiling in PREEMPT-RT linux," *Journal of Software*, vol. 7, no. 3, Mar. 2012, doi: 10.4304/jsw.7.3.516-525.
- [24] J. M. Garrido, *Performance modeling of operating systems using object-oriented simulations: A practical introduction*, 2000th ed. Boston: Kluwer Academic Publishers, 2002. doi: 10.1007/b116043.
- [25] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-port networks: preserving the timed behavior of synchronous systems for modeling on FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 2, no. 3, pp. 1–26, Sep. 2009, doi: 10.1145/1575774.1575775.

BIOGRAPHIES OF AUTHORS



Abedalmuhdi Almomany    His M.Sc. and Ph.D. in computer engineering from the University of Alabama in Huntsville in 2015 and 2017, respectively. He earned his B.Sc. in computer engineering from Yarmouk University in 2005. He started working as an assistant professor in the Computer Engineering Department of the College of Engineering at Yarmouk University in 2017. His areas of interest in research are embedded systems, parallel processing, and reconfigurable computing. He received grants from different institutes to pursue his research. He can be contacted at email: emomani@yu.edu.jo.



Amin Jarrah    His M.Sc. and Ph.D. in computer engineering were earned in 2010 and 2014, respectively, from the Jordan University of Science and Technology and the University of Toledo in the USA. He worked with firms in the TATA group and Magna Electronics between 2014 and 2016 as an embedded software engineer. As an assistant professor, he began working at Yarmouk University in January 2016. At the moment, he holds the position of associate professor at Yarmouk University in Irbid, Jordan's Department of Computer Engineering. His research focuses on the use of parallel processing with FPGAs, system on chips (SoC), and GPU in real-time embedded hardware implementation and high speed computing. He can be contacted at email: amin.jarrah@yu.edu.jo.