

FSHMEM: Supporting Partitioned Global Address Space on FPGAs for Large-Scale Hardware Acceleration Infrastructure

Yashaël Faith Arthanto
School of Electrical Engineering
KAIST
Daejeon, Republic of Korea
yashaël.faith@kaist.ac.kr

David Ojika
Flapmax
Austin, USA
dave@flapmax.com

Joo-Young Kim
School of Electrical Engineering
KAIST
Daejeon, Republic of Korea
jooyoung1203@kaist.ac.kr

Abstract—By providing highly efficient one-sided communication with globally shared memory space, Partitioned Global Address Space (PGAS) has become one of the most promising parallel computing models in high-performance computing (HPC). Meanwhile, FPGA is getting attention as an alternative compute platform for HPC systems with the benefit of custom computing and design flexibility. However, the exploration of PGAS has not been conducted on FPGAs, unlike the traditional message passing interface. This paper proposes FSHMEM, a software/hardware framework that enables the PGAS programming model on FPGAs. We implement the core functions of GASNet specification on FPGA for native PGAS integration in hardware, while its programming interface is designed to be highly compatible with legacy software. Our experiments show that FSHMEM achieves the peak bandwidth of 3813 MB/s, which is more than 95% of the theoretical maximum, outperforming the prior works by 9.5 \times . It records 0.35 μ s and 0.59 μ s latency for remote write and read operations, respectively. Finally, we conduct a case study on the two Intel D5005 FPGA nodes integrating Intel’s deep learning accelerator. The two-node system programmed by FSHMEM achieves 1.94 \times and 1.98 \times speedup for matrix multiplication and convolution operation, respectively, showing its scalability potential for HPC infrastructure.

Index Terms—Parallel Programming Model, PGAS, FPGA, GASNet, SHMEM

I. INTRODUCTION

High Performance Computing (HPC) employs computer clusters to solve advanced computational problems, primarily centered around scientific applications including molecular modeling [1], weather modeling [2], and genomics [3]. Recently, as artificial intelligence (AI) and machine learning (ML) technology are transforming major industries with highly beneficial applications such as image captioning [4], [5], virtual assistant [6], stock market prediction [7], and self-driving cars [8], HPC systems have evolved to accommodate AI workloads [9], [10]. Consisting of hundred-thousands of CPU-GPU nodes, a state-of-the-art HPC infrastructure performs several hundreds of peta floating-point operations per second (PFLOPS) [11], [12]. *Parallel programming model* is the key to run large-scale AI services on distributed computing nodes.

Figure 1 shows three main parallel programming models: Shared Memory, Message Passing, and Partitioned Global

Address Space (PGAS). In the shared memory model, multiple processes can directly access the shared memory space. On the other hand, the message passing model only allows the processes to exchange messages based on the Message Passing Interface (MPI) [13]. MPI has become popular in HPC as it enables data communication between remote nodes with the same inter-process mechanism. Since version 2, it also supports one-sided data communication operations. PGAS uses one-sided data communication with the concept of partitioned but globally shared memory. Leveraging the benefit of the shared memory model, each node has direct access to another node’s memory space without interfering with the corresponding process. This is possible because all the nodes in the network form a single global address space, in which each node has its own part. In addition, each node has a private memory for its local processing. With this concept, PGAS brings a few advantages to parallel programming. First, its shared view of memory simplifies parallel programming, inspired by the shared memory paradigm. Second, PGAS’s one-sided communication [14] that does not interfere with the remote process reduces the communication overhead. Third, the clear decoupling of private and shared memory gives programmers a better perspective on utilizing locality.

Meanwhile, the HPC community has a surge of interest in adopting an alternative acceleration platform beyond GPU. FPGA is one strong candidate due to the flexibility in design, proving its feasibility in production environments [15]–[17]. For the use of FPGA in HPC, several works have implemented MPI on FPGAs [18]–[22]. However, research on PGAS support for FPGA is still preliminary [23], as the protocol is relatively new. In this paper, we present FSHMEM, a software/hardware framework to support PGAS on FPGAs,

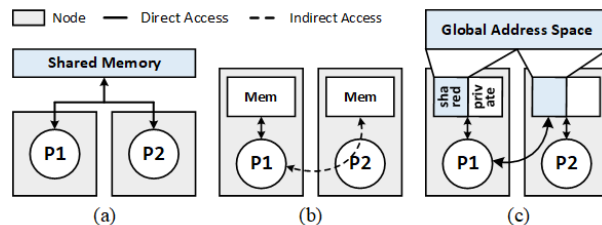


Fig. 1: Three parallel programming models: (a) Shared Memory, (b) Message Passing, and (c) PGAS

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2020-0-01847) supervised by the IITP (Institute of Information & Communications Technology Planning & Evaluation).

named after the shared memory library (SHMEM). Along with the trend of FPGA adoption in HPC, we believe that FSHMEM will play a fundamental role in building FPGA-based hardware infrastructure at scale. Our main contributions are as follows:

- FSHMEM implements core functions of GASNet protocol [24] on FPGA to enable native PGAS integration in hardware. It also provides an easy-to-use software interface for high adaptability.
- We benchmark the FSHMEM’s bandwidth and latency performance for Remote Direct Memory Access (RDMA) and compare it against the previous results. Our maximum bandwidth of 3813 MB/s outperforms prior works by $9.5\times$ with an average latency of $0.47\mu s$.
- We build a practical multi-FPGA acceleration system using FSHMEM and Intel’s Deep Learning Accelerator (DLA) as computing core. We parallel-program the AI computations and evaluate the performance to show the framework’s potential for performance scaling.

II. BACKGROUND

A. GASNet

Global Address Space Networking (GASNet) is a language-independent networking middleware that describes PGAS’s one-sided communication interface. Its interface is built around Active Message (AM) protocol. In AM, each message head specifies the address of a handler function that will be called upon the message’s arrival, and each message body provides the arguments for the function along with the data to be transferred [25]. GASNet does not provide function implementations because it may vary among network interfaces or devices, but it must support one-way data communication. Recently, GASNet-EX, an upgraded version of GASNet, was also introduced [26]. It achieves an average of $1.77\mu s$ latency and saturates to maximum bandwidth at 4-8KB, which are $1.05\text{-}5\times$ faster in latency and $4\times$ faster to saturation than MPI.

B. MPI on FPGA

TMD-MPI [27] implemented MPI on FPGA by creating a software library for embedded processors and TMD-Message Passing Engine (TMD-MPE) for hardware kernels. This engine brings MPI functionality to a hardware kernel by handling MPI’s protocol and packet generation. They perform bandwidth and latency benchmarks under a 2-rank system and discover a maximum bandwidth at 400 MB/s, achieving 75% of its peak bandwidth. Zivras *et al.* implemented one-sided MPI primitives on embedded FPGA [28]. Tested with two FPGAs on a single board, their implementation reaches 141 MB/s, or 70.6% of the peak bandwidth. Other works [18], [29] try to improve the system bottleneck caused by extensive use of collective communications by offloading the data processing algorithms such as *reduce*, *allReduce*, *bcast* to the FPGAs within network switches. They attain the latency speedup of $3.9\times$ on average.

C. PGAS on FPGA

The GASNet (Toronto Heterogeneous GASNet) [23] is the latest work that implemented GASNet on FPGA by introducing Global Address Space Core (GASCore), an RDMA

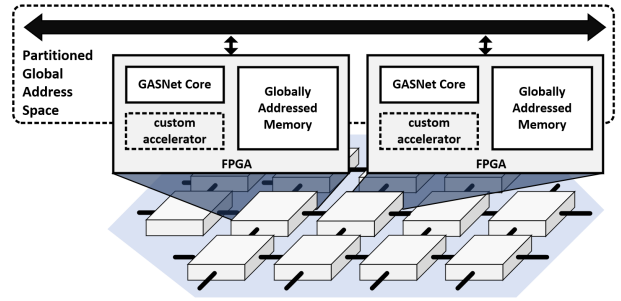


Fig. 2: FSHMEM-based hardware acceleration infrastructure

core, and Programmable Active Message Sequencer (PAMS). PAMS is responsible for interfacing the hardware kernel to the GASCore by handling messages and synchronization. The GASCore can be programmed directly through software when used by embedded processors. Meanwhile, hardware kernels program the GASCore through PAMS, containing a code memory for GASNet operations. In addition, they suggest a software stack to transform an application’s compute portion to hardware kernels and load the GASNet portion into PAMS.

D. Multi-FPGA Infrastructure for HPC

Axel [22] showcased an early adoption of FPGAs in HPC infrastructure, in which each node includes CPU, GPU, and FPGA through a PCIe switch. It uses a Map-Reduce framework to accelerate N-Body Simulation using heterogeneous resources achieving an impressive $22.7\times$ performance improvement. However, Axel does not scale out well, getting only $4.4\times$ improvement from 1 to 16 nodes because of the all-to-all functions that do not scale linearly with the number of nodes. On the other hand, Microsoft’s BrainWave platform [15] deployed FPGAs in the data center at scale to provide real-time AI services. In this architecture, FPGA accelerators are integrated between the CPUs and the top-of-rack switch, allowing them to communicate with many servers in the data center. Each FPGA contains a specialized AI accelerator for low latency inference services.

Cygnus [17] is a heterogeneous supercomputer utilizing FPGAs in 40% of their compute nodes. These FPGAs are additionally interconnected in a 2D-torus Infiniband network. Cygnus achieved 2.4 PFLOPS for double-precision, while the FPGAs contributed only 0.6 PFLOPS for single-precision.

III. FSHMEM: INTER-FPGA INFRASTRUCTURE FOR PGAS PROGRAMMING MODEL

This work proposes FSHMEM, an infrastructural framework that enables PGAS on FPGAs by implementing the GASNet interface in hardware and supporting the corresponding API in software. Figure 2 shows the conceptual diagram of an FSHMEM-based hardware acceleration infrastructure made of fabrics of FPGAs. FSHMEM enables the PGAS parallel computing model on a pool of FPGAs by interconnecting them through the GASNet interface. Each FPGA node in this infrastructure includes the GASNet core, a globally addressed memory space, and a custom accelerator. Note that the hard-

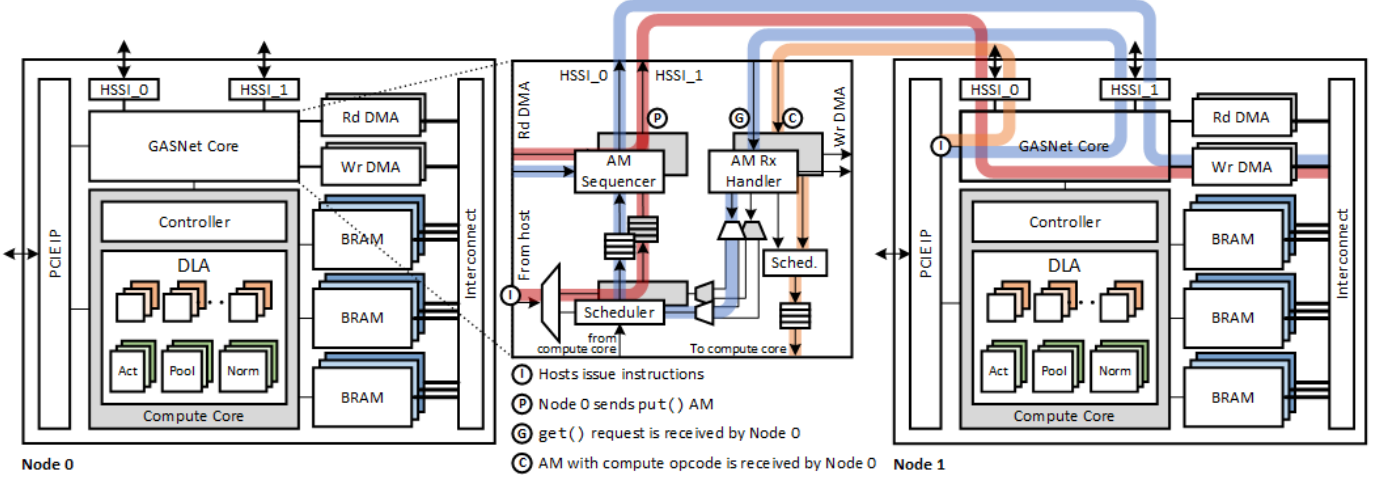


Fig. 3: FSHMEM node architecture with dataflow for *gasnet_put* (red), *gasnet_get* (blue) and *gasnet_AMRequest** (orange)

ware infrastructure can adopt any network topology, while the diagram shows an example mesh topology.

The FSHMEM based infrastructure facilitates mainly three benefits. First, FSHMEM provides low latency, direct FPGA-to-FPGA communication by implementing GASNet’s lightweight interface on high-speed transceiver links. Second, FSHMEM is highly flexible with globally shared memory space and local memories. It allows users to deliberately manage memory transfer and internal data flow using GASNet’s memory-to-memory functions and custom hardware function handlers. Third, FSHMEM’s software API is highly compatible with the existing PGAS frameworks so that many legacy HPC applications can be easily adapted to use FSHMEM. In the following subsections, we will describe the GASNet core, an essential hardware module to implement the GASNet protocol on FPGA, the custom accelerator’s interface to the GASNet core, and the software interface.

A. GASNet Core

GASNet core implements the fundamental communication protocol of the GASNet specification, which is the AM interface. There are two key features to consider when implementing AM on hardware:

- AM invokes a handler function on the destination node that may compute the data or reply with the requested data. This is done by passing a handler function pointer in a software implementation. Instead, the GASNet core directly passes the function opcode.
- AM carries arguments for the function to be invoked. Arguments can be integers or data payloads to be stored in the destination address. Therefore, to provide such functionality, the GASNet core must contain an RDMA.

Table I shows the list of GASNet functions that GASNet core currently supports on the FPGA: *gasnet_AMRequest*, *gasnet_AMReply*, *gasnet_put*, and *gasnet_get*. Other functions from the specifications such as job controls, job environments, and barrier functions are implemented on the software side. Meanwhile, atomicity control of the handler function is natively supported by hardware.

TABLE I: Implemented Functions on GASNet Core

Function	Description
<i>gasnet_AMRequest*</i> ((Short/Medium/Long)	Send messages to destination node
<i>gasnet_AMReply*</i> ((Short/Medium/Long)	Reply messages to requesting node
<i>gasnet_put</i> ()	Store data in the target node
<i>gasnet_get</i> ()	Request data from the desired node

As part of the specification, AM request functions have three variants based on the length of arguments: short, medium, and long. The short message does not send a payload to the destination, making it generally suitable for configuration update functions. Both medium and long messages include payload, but medium messages go to the local memory address while long messages go to the globally shared address space. GASNet’s AM reply functions are essentially the same as the request functions, except they can only reply to the requesting node. *gasnet_put* and *gasnet_get* function, which comes from the GASNet extended API, can be implemented using the request and reply functions. For example, we use the AM request function that invokes the PUT and GET handler for *gasnet_put* and *gasnet_get* function, respectively. Note that the GET handler will invoke an AM reply function. Another necessary handler is for the compute core.

Figure 3 illustrates the FSHMEM’s node architecture, mainly consisting of the host interface (PCIe), GASNet core, and its underlying memory and accelerator subsystems. The GASNet core is composed of two sets of AM sequencer, AM receiver handler, and schedulers with FIFOs. Each set corresponds to the High-Speed Serial Interface (HSSI) port for inter-FPGA communication. In this module, the AM sequencer forms the active message by generating the header and reading the message body from the memories via DMA. Since the requests can come from multiple sources, e.g., host, compute core, or a remote node, the scheduler is necessary. The AM receiver handler writes the incoming data to the memories.

The figure also shows the fundamental data flows between

two FSHMEM nodes: *gasnet_put* in red, *gasnet_get* in blue, and *gasnet_AMRequest* in orange. For the *gasnet_put* operation, which is a remote write from Node 0 to Node 1, the host of Node 0 initially issues the *gasnet_put* command (①). Going through the scheduler and FIFO, it arrives at the AM sequencer. The sequencer then fetches the necessary data using the read DMA and sends the formed message to Node 1 (②). In Node 1, the AM receive handler checks the opcode of the received message, which should be a PUT opcode, and uses write DMA to store the payload to the destination address.

Let us assume that Node 1 performs *gasnet_get* operation this time, which is a remote read from Node 0. Similar to the *gasnet_put* case, the host of Node 1 issues the *gasnet_get* command, yet without carrying any payload. Upon the arrival of the GET request in Node 0 (③), the AM receiver handler immediately issues a PUT reply command and forwards it to the scheduler. After that, the execution is exactly the same as the PUT operation described above. As a result, the requested data are read and packed by the AM sequencer and are sent to Node 1, which accomplishes the GET command. The orange color in the figure highlights the dataflow of a *gasnet_AMRequest** function, especially with the case that carries a compute opcode without data payloads. Upon receiving this type of message (④), the AM receiver handler sends the function’s arguments to the compute command scheduler to get it queued for the compute core’s execution. If the received message carries the data payload, the AM receiver handler writes it into the memory before instructing the compute core to execute.

Due to the simple and optimized design, the GASNet core’s logic usage is extremely low, around 0.2% logic for two HSSI ports on Intel Stratix 10. Its logic size will increase with the number of available HSSI ports in the FPGA. This low design overhead allows the compute core to utilize most of the device’s resources for high application performance. However, as the GASNet core is not designed for any specific network topology, it may need a router for an extensive network setting.

B. Deep Learning Accelerator

For the primary compute core of this work, we customize the Intel DLA [30] for major AI computations such as convolution and matrix multiplication. It uses a 1-dimensional systolic array architecture to accelerate these computations with high flexibility. Although optimized for convolution, the DLA can also perform matrix-vector and matrix-matrix multiplication by properly mapping the data. It is highly flexible as the computation types and tensor sizes are exposed as arguments. The GASNet’s active message can easily instruct the DLA via its handler interface by passing a few arguments.

A typical interaction between the host and DLA for parallel execution is a repetition of compute command, acknowledgment, and PUT command. With a powerful compute core like the DLA, this workflow requires an extra host intervention and a considerable communication bandwidth if the PUT command is performed upon the final result data. Therefore, we devise the Automatic Result Transfer (ART) mechanism

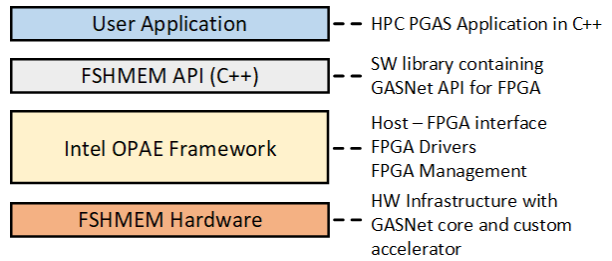


Fig. 4: FSHMEM software stack

to let the DLA initiate the transfer. Instead of sending a large-sized message at the end of the computation, ART splits it into several smaller-sized messages in the middle of the computation. ART enables this by issuing a PUT command for every N valid result, in which N is configurable. Thus, ART hides the communication latency with the computation execution time while removing extra host interventions.

C. Software Interface

FSHMEM supports GASNet API for compatibility with legacy HPC applications. Figure 4 shows FSHMEM’s entire software stack that starts from the user’s PGAS-based HPC application. The application is programmed using the GASNet-compatible FSHMEM API in C++. The stack utilizes Intel’s Open Programmable Acceleration Engine (OPAE) library [31] for host communication, device driver, and FPGA management. Although FSHMEM is currently based on Intel’s FPGA platform, we can extend it to Xilinx’s FPGA platform by integrating Xilinx Runtime (XRT) library [32]. On the bottom of the stack, FSHMEM provides a hardware layer that implements remote data communications across devices. Each FSHMEM device instantiates the GASNet core module that works alongside the software interface. Compared to THE GASNet, our FSHMEM API directly commands the GASNet core to initiate data transfer without needing to compile or translate code into dedicated instructions. In this way, FSHMEM promotes a ‘plug-and-play’ notion of using FPGA with GASNet API.

IV. COMMUNICATION PERFORMANCE RESULTS

A. Methodology

We use Intel Acceleration Stack 2.0.1 framework for logic synthesis and layout and implement the architecture on Intel’s D5005 Programmable Acceleration Card (PAC). Intel D5005 PAC is a high-performance PCIe-attached FPGA acceleration card that includes Stratix 10 SX FPGA (part number: 1SX280HN2F43E2VG) with 32GB DDR memory and 2 QSFP+ network interfaces. For experiments, we build a prototype machine that harnesses an Intel CPU as the host and two PACs as the FSHMEM devices. The host CPU drives the testing/application program using FSHMEM API, while both PACs, interconnected via QSFP+ cables in a ring fashion, are responsible for actual execution. We profile FSHMEM’s PUT and GET active messages via *gasnet_put* and *gasnet_get* function by measuring the read/write bandwidth and latency between the two FPGAs. For accurate measurement, we add a hardware performance counter to measure the time taken from

TABLE II: FPGA Resource Utilization

Module	LUT + Register	BRAM	DSP
GASNet core	1995.3 (0.21%)	17 (0.15%)	0 (0%)
DLA	102276 (10.96%)	8 (0.07%)	1409 (24.46%)

when a command is given until the corresponding message is returned.

B. FPGA Resource Utilization

Table II shows the resource utilization result of the GASNet core and DLA implemented on Intel D5005 PAC at 250 MHz operating frequency. The GASNet core takes minimal logic resources to implement, making it suitable for use with deep learning accelerators or other compute-demanding accelerators. The DLA used for our experiments contains 16×8 PEs, utilizing a quarter of the available DSPs.

C. Communication Bandwidth

Figure 5 shows the communication bandwidth measurement results of FSHMEM when the packet size varies among 128, 256, 512, and 1024 bytes, with increasing transfer size from 4 bytes to 2 MB. We measure both PUT and GET bandwidth and achieve a peak bandwidth of 3813 MB/s for the packet size of 512 and 1024 bytes, which is more than 95% of the theoretical maximum bandwidth. The 128 and 256-byte packet size achieves a 2621 and 3419 MB/s peak bandwidth, which is 65% and 85% of the theoretical maximum, respectively. Smaller packet sizes show lower peak bandwidths because they need to generate more packets for the same transfer size, causing network overhead and throughput degradation.

We have two observations in the bandwidth curves. First, FSHMEM’s communication reaches the half-maximum at around 2KB. It saturates around the transfer size of 32KB, reaching 95% of the peak bandwidth for all cases, telling us that we must transfer at least 32KB to fully utilize the FSHMEM’s available bandwidth. Second, we observe that GET functions (i.e., read operation) have lower bandwidth than PUT functions (i.e., write operation). Although the GET bandwidth is almost similar to PUT when the transfer size is large enough (> 32 KB), their gap tends to be large for small-to-medium-sized transfers. For example, the GET bandwidth is 20% and 8% lower than the PUT bandwidth at the transfer size of 2KB and 8KB, respectively. This is because the GET function consists of a short request message and a long reply message with data, while the PUT function only consists of a long message containing the data. Therefore, the GET function includes an additional short request message that does not contain any payload, causing a constant overhead regardless of the transfer size. As a result, the overhead cost is more apparent for small-to-medium-sized transfers when the GET reply message is small enough, while the overhead is amortized in large-sized transfers. Figure 5 also depicts the results of the previous works for comparison. Both TMD-MPI [27] and THE GASNet [23] achieved the peak bandwidth of 400 MB/s, while FSHMEM achieved 3813 MB/s, which is a $9.5 \times$ improvement. Compared to the one-sided MPI [28], FSHMEM achieves $26 \times$ bandwidth improvement.

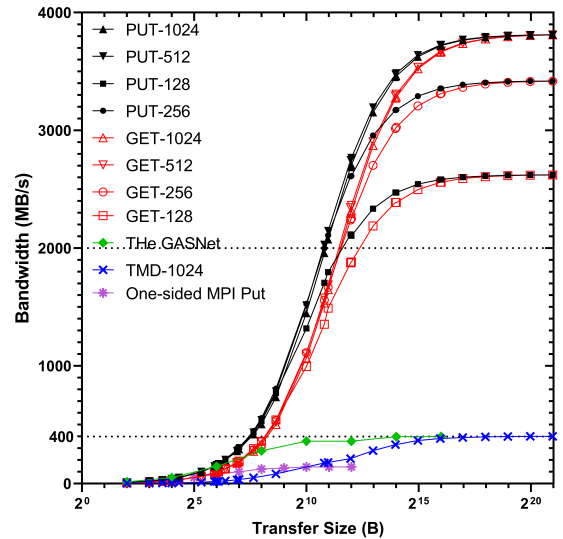


Fig. 5: Communication bandwidth measurement results

TABLE III: Latency Comparison

Implementations	PUT (μ s)	GET (μ s)
TMD-MPI (inter-m2b)	2	
One-sided MPI	0.36	0.62
THE GASNet (short message)	0.17	0.35
THE GASNet (single word)	0.29	0.47
FSHMEM (short message)	0.21	0.45
FSHMEM (long message)	0.35	0.59

D. Communication Latency

We measure FSHMEM’s communication latency with the same experimental setting. From the time a command is given to the initiator FPGA, the PUT latency is measured until the message header is received in the remote FPGA, and the GET latency is measured until the reply message’s header is returned to the initiator FPGA. Table III summarizes the latency measurement results of the various implementations for PUT and GET function. The FSHMEM’s average latency for short messages (no payload) is measured at 0.21μ s and 0.45μ s for PUT and GET functions, while the average latency for long messages (payload size: 4 B to 2 MB) is measured at 0.35μ s and 0.59μ s for PUT and GET function, respectively. The GET latency is by nature longer than that of PUT as the function requires two-way communication where a request is replied with the requested data.

Table III also shows a huge difference in latency between the TMD-MPI with two-sided communication and the other one-sided communication protocols including FSHMEM. THE GASNet shows lower latency than FSHMEM through onboard wires. However, such channels are less scalable than FSHMEM’s QSFP+ cables, commonly used in data centers.

E. Comparison

Table IV summarizes FSHMEM’s implementation details compared to the previous implementations. FSHMEM achieves the highest communication bandwidth of 3813 MB/s with 95% efficiency by utilizing a high-speed QSFP+ interface and lightweight GASNet core implementation.

TABLE IV: Comparison with Prior Works

	TMD-MPI [27]	One-sided MPI [28]	The GASNet [23]	This Work
FPGA	Xilinx XC5VLX110	Xilinx XC2V6000	Xilinx XC5VLX155T	Intel Stratix-10
Clock	133.33 MHz	50 MHz	100 MHz	250 MHz
Data width	32-bit	32-bit	32-bit	128-bit
Physical channel	Intel Front Side Bus	On-board wires	On-board wires	QSFP+
Max BW	400 MB/s	141 MB/s	400 MB/s	3813 MB/s
Efficiency	0.75	0.706	1.00	0.95

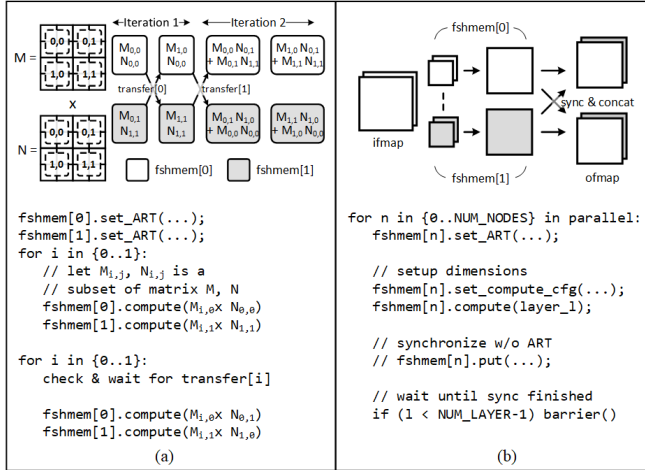


Fig. 6: Parallel programs for (a) matrix multiplication and (b) convolution with their pseudo codes

V. CASE STUDY

We conduct two case studies using FSHMEM to show its feasibility in parallel programming, especially for AI applications. We map a general matrix multiplication and convolution workload on the two FPGA nodes where each node integrates the Intel DLA with 16×8 PEs.

Figure 6(a) shows the implementation of parallel matrix multiplication ($M \times N$) using FSHMEM. Each of the two input matrices is partitioned into four sub-matrices, and the sub-matrices are split across the two FPGA nodes. The computation starts by iterating the row of matrix M to multiply its sub-matrix with that of matrix N (e.g., $N_{0,0}$, $N_{1,1}$), followed by exchanging the partial sum result to another node. After the first iteration, it checks if the first partial sum is transferred and does the same operation with the next set of matrix N sub-matrices (e.g., $N_{0,1}$, $N_{1,0}$). Each node accumulates the partial sum to the previously transferred partial sum to get the final result. The result are stored across the two FPGAs like matrix M , where each FPGA holds sub-matrices of the same column. Note that the command to transfer the partial sum is expressed by setting up the ART instead of explicitly using a PUT function in the pseudo code, allowing FSHMEM to send results simultaneously with computations to maximize the speedup. We perform experiments on three different matrix sizes: 256×256 , 512×512 , and 1024×1024 .

For convolution, a widely used operation in convolutional

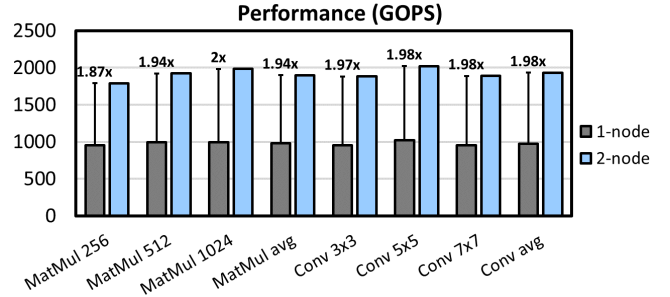


Fig. 7: Performance results for matrix multiplication and convolution and the speedups achieved using 2 nodes

neural networks (CNN), we split the weight kernels into two groups for parallel computation, as shown in Figure 6(b). After each convolution, both nodes must synchronize their results and concatenate them, producing a complete result in both nodes. We use 64×64 input feature maps for the experiments and vary the number and size of kernels: 256 , $3 \times 3 \times 256$, 192 , $5 \times 5 \times 192$, and 128 , $7 \times 7 \times 128$.

Figure 7 shows the experimental results on the two workloads comparing the single-node and two-node performance. For matrix multiplication, the single-node FPGA achieves an average of 979.4 GOPS, reaching 95.6% of the theoretical maximum, while the two-node implementation achieves 1898.5 GOPS, which is a $1.94 \times$ performance gain. We can see that the speedup increases as the matrix size increases because the longer accumulation in a larger matrix size gives more time to transfer the partial sum from one node to another. The convolution operation's average performance gain is about $1.98 \times$ with 1931.3 GOPS. In general, convolution requires longer accumulation than matrix multiplication, resulting in a higher average speedup. However, none of the convolution results reach $2 \times$ speedup. One of the matrix multiplication results reaches $2 \times$ speedup as its algorithm hides the communication latency in-between the process, while the synchronization process in convolutions happens at the end of the process, causing an inevitable latency. Overall, all performances exceed the throughput of a single node by around $1.95 \times$ on average, suggesting a nearly linear speedup as the number of nodes increases.

VI. CONCLUSION AND FUTURE WORK

To conclude, we propose FSHMEM, a software/hardware framework for GASNet-enabled FPGA hardware acceleration infrastructure, by implementing the GASNet core and the supporting API in software. We describe how this framework implements GASNet's AM functions in hardware and the software stack that enables high compatibility. The benchmark results show that FSHMEM's bandwidth outperforms the prior works with competitive latency. Our case study on parallel matrix multiplication and convolution also shows great potential for scaling up. For future work, we plan to build a scaled-up server that contains up to 8 FPGA acceleration cards and build an FSHMEM-based hardware infrastructure using it. We also plan to accelerate various machine learning models using the PGAS programming model for AI-enabled HPC.

REFERENCES

- [1] D. E. Shaw *et al.*, “Anton 2: Raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer,” in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 41–53.
- [2] J. G. Powers *et al.*, “The weather research and forecasting model: Overview, system efforts, and future directions,” *Bulletin of the American Meteorological Society*, vol. 98, no. 8, pp. 1717 – 1737, 2017. [Online]. Available: <https://journals.ametsoc.org/view/journals/bams/98/8/bams-d-15-00308.1.xml>
- [3] J. T. Simpson and R. Durbin, “Efficient construction of an assembly string graph using the FM-index,” *Bioinformatics*, vol. 26, no. 12, pp. i367–i373, 06 2010. [Online]. Available: <https://doi.org/10.1093/bioinformatics/btq217>
- [4] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, “Show and tell: Lessons learned from the 2015 mscoco image captioning challenge,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 652–663, 2017.
- [5] M. Cornia, M. Stefanini, L. Baraldi, and R. Cucchiara, “Meshed-memory transformer for image captioning,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 10578–10587.
- [6] T. Capes *et al.*, “Siri on-device deep learning-guided unit selection text-to-speech system,” in *Proc. Interspeech 2017*, 2017, pp. 4011–4015. [Online]. Available: <http://dx.doi.org/10.21437/Interspeech.2017-1798>
- [7] S. Mehtab, J. Sen, and A. Dutta, *Stock Price Prediction Using Machine Learning and LSTM-Based Deep Learning Models*, S. M. Thampi, S. Piramuthu, K.-C. Li, S. Berretti, M. Wozniak, and D. Singh, Eds. Singapore: Springer Singapore, 2021.
- [8] E. Talpes *et al.*, “Compute solution for tesla’s full self-driving computer,” *IEEE Micro*, vol. 40, no. 2, pp. 25–35, 2020.
- [9] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, “Deep learning with cots hpc systems,” in *International conference on machine learning*. PMLR, 2013, pp. 1337–1345.
- [10] D. Kalamkar, E. Georganas, S. Srinivasan, J. Chen, M. Shiryayev, and A. Heinecke, “Optimizing deep learning recommender systems training on cpu cluster architectures,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.
- [11] D. Monroe, “Fugaku takes the lead,” *Commun. ACM*, vol. 64, no. 1, p. 16–18, dec 2020. [Online]. Available: <https://doi.org/10.1145/3433954>
- [12] J. Hines, “Stepping up to summit,” *Computing in science & engineering*, vol. 20, no. 2, pp. 78–82, 2018.
- [13] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, Jun. 2021. [Online]. Available: <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>
- [14] K. Yelic *et al.*, “Productivity and performance using partitioned global address space languages,” in *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, ser. PASC07. New York, NY, USA: Association for Computing Machinery, 2007, p. 24–32. [Online]. Available: <https://doi.org/10.1145/1278177.1278183>
- [15] E. Chung *et al.*, “Serving dnns in real time at datacenter scale with project brainwave,” *IEEE Micro*, vol. 38, pp. 8–20, March 2018. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/>
- [16] R. Tornero-Gavilá, J. Flich Cardo, J. M. Martínez Martínez, T. Picornell-Sanjuan, and V. Scotti, “The mango process for designing and programming multi-accelerator multi-fpga systems,” in *Fourth International Workshop on Heterogeneous High-Performance Reconfigurable Computing (H2RC'18)*. ACM, 2018.
- [17] B. Taisuke *et al.*, “Cygnus: A multi-hybrid supercomputing platform with gpus and fpgas,” URL: https://www.isc-hpc.com/agenda2019/conferences/isc_hpc/assets/2019/posters/proj138.pdf, June 2019, poster on ISC2019.
- [18] Q. Xiong, C. Yang, P. Haghi, A. Skjellum, and M. Herbordt, “Accelerating mpi collectives with fpgas in the network and novel communicator support,” in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 215–215.
- [19] M. Saldana and P. Chow, “Tmd-mpi: An mpi implementation for multiple processors across multiple fpgas,” in *2006 International Conference on Field Programmable Logic and Applications*, 2006, pp. 1–6.
- [20] A. Patel *et al.*, “A scalable fpga-based multiprocessor,” in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 111–120.
- [21] D. L. Ly, M. Saldaña, and P. Chow, “The challenges of using an embedded mpi for hardware-based processing nodes,” in *2009 International Conference on Field-Programmable Technology*, 2009, pp. 120–127.
- [22] K. H. Tsoi and W. Luk, “Axel: A heterogeneous cluster with fpgas and gpus,” in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 115–124. [Online]. Available: <https://doi.org/10.1145/1723112.1723134>
- [23] R. Willenberg and P. Chow, “A heterogeneous gasnet implementation for fpga-accelerated computing,” in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: <https://doi.org/10.1145/2676870.2676885>
- [24] D. Bonachea and P. Hargrove, “Gasnet specification, v1.8.1,” Lawrence Berkeley National Laboratory, Tech. Rep., 8 2017.
- [25] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer, “Active messages,” *ACM SIGARCH Computer Architecture News*, vol. 20, pp. 256–266, 5 1992.
- [26] D. Bonachea and P. H. Hargrove, “Gasnet-ex: A high-performance, portable communication library for exascale,” in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2018, pp. 138–158.
- [27] M. Saldaña *et al.*, “Mpi as a programming model for high-performance reconfigurable computers,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, nov 2010. [Online]. Available: <https://doi.org/10.1145/1862648.1862652>
- [28] S. G. Ziavras, A. V. Gerbessiotis, and R. Bafna, “Coprocessor design to support mpi primitives in configurable multiprocessors,” *Integration*, vol. 40, no. 3, pp. 235–252, 2007. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167926005000519>
- [29] P. Haghi *et al.*, “Fpgas in the network and novel communicator support accelerate mpi collectives,” in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–10.
- [30] M. S. Abdelfattah *et al.*, “Dla: Compiler and fpga overlay for neural network inference acceleration,” in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 411–418.
- [31] Intel Corporation, “Open programmable acceleration engine.” [Online]. Available: <https://01.org/opae>
- [32] Xilinx, Inc., “Xilinx runtime.” [Online]. Available: <https://github.com/Xilinx/XRT>