Efficient Graph Computations on FPGA

Kunj Patel
dept. Electrical and Computer
Stevens Institute of Technolgy
Hoboken, USA
kpatel6@stevens.edu

Prof. Zhuo Feng
dept. Electrical and Computer
Stevens Institute of Technolgy
Hoboken, USA
zfeng12@stevens.edu

Abstract—Efficient computations of graphs are vital for a plethora of applications across various domains such as bioinformatics, social network analysis, and transportation systems[1].The potential of Field-Programmable Gate Arrays (FPGAs) to accelerate graph computations is promising due to their energy efficiency, reconfigurability, and parallel processing capabilities[2]. However, the efficient implementation of graph algorithms on FPGAs is impeded by several challenges, including limited on-chip memory, irregular data access patterns, and complex control logic[2]. In this report, we present a novel architecture based on FPGAs that addresses these challenges and accelerates graph computations. Our approach introduces an adaptive scheduler that dynamically balances communication overhead and parallelism[2], and a memory-efficient data structure named Compressed Sparse Vertex (CSV) for representing graphs[3]. Our preliminary results demonstrate that our proposed architecture achieves up to 3x speedup and 40% energy reduction on various graph benchmarks.

Keywords— Efficient computations of graphs, FPGAs (Field Programmable Gate Arrays), Adaptive scheduler, Hardware architecture, Memory-efficient data structure, Performance evaluation, Energy efficiency, Resource utilization, Scalability, Generalizability

I. Introduction

Graph computations play a pivotal role in various domains, including social network analysis, bioinformatics, and transportation systems[3]. FPGAs are emerging as a promising platform for accelerating graph computations owing to their energy efficiency, reconfigurability, and parallel processing capabilities. However, implementing efficient graph algorithms on FPGAs is fraught with challenges such as limited onchip memory, irregular data access patterns, and complex control logic. Researchers have proposed several approaches to tackle these challenges, such as partitioning graphs to optimize memory utilization, or designing hardware architectures tailored to graph processing on FPGAs[4]. Despite these advancements, there is still room for improvement in terms of both performance and energy efficiency. In this report, we propose a novel FPGA-based architecture to accelerate graph computations and overcome the aforementioned challenges. Our contributions include a memory-efficient data structure for representing graphs, an adaptive scheduler that balances parallelism and communication overhead, and a performance evaluation demonstrating the superiority of our proposed architecture over existing solutions.

The remainder of this paper is organized as follows:

Section 2 presents the problem statement, defining the research question and the scope of our investigation.
Section 3 describes our solution, detailing the proposed approach for efficient graph computations on FPGAs and the key techniques employed to address the challenges.
Section 4 reports the numerical results and analysis, evaluating the performance of our approach on different graph problems and comparing it with existing solutions.
Section 5 concludes the paper, summarizing our main contributions and discussing future directions for this research.

II. Problem Statement

This report addresses the challenge of designing and implementing an efficient FPGA-based architecture to accelerate graph computations. Graph computations play a crucial role in various domains, including social network analysis, bioinformatics, and transportation systems. However, efficiently implementing graph algorithms on FPGAs presents several challenges that need to be addressed:

A. Limited on-chip memory:

FPGAs have restricted on-chip memory resources compared to traditional processors such as CPUs and GPUs. As a result, handling large-scale graphs can be challenging, and finding ways to optimize memory usage while maintaining efficient access patterns is critical. An ideal solution would involve designing a memory-efficient data structure for graph representation that can compactly store graph data and support various graph algorithms and datasets[5].

B. Irregular data access patterns:

Graph algorithms often exhibit irregular data access patterns due to the inherent properties of graphs. These irregular patterns can lead to inefficient memory accesses, communication overhead, and increased latency on FPGAs. An effective solution should involve developing techniques to optimize data access patterns and minimize communication overhead while maximizing the parallelism offered by FPGAs[5].

C. **Complex control logic:**

Implementing graph algorithms on FPGAs requires complex control logic to manage parallelism, data dependencies, and synchronization. This complexity can result in increased design effort, resource utilization, and power consumption. An efficient FPGA-based architecture should simplify the control logic and provide mechanisms for managing the parallelism and data dependencies effectively, thereby reducing resource utilization and power consumption[5]. Surmounting these tribulations, our primary investigative quandary in this disquisition revolves around devising a potent FPGA-based edifice to expedite graph computations. This encompasses the formulation of frugal data structures for graph portrayals, devising stratagems to streamline data accession patterns and communicative encumbrance, as well as contriving a hardware-software symbiotic methodology to alleviate graph algorithm execution on FPGAs. By tackling these conundrums, our aspiration lies in augmenting the advancement of efficacious FPGA-based infrastructures for graph computations, confronting the constraints inherent in extant solutions, and facilitating a myriad of applications in diverse spheres.

III. INITIAL CONCEPTS AND OUTCOMES

Within this segment, we present an intricate synopsis of our rudimentary notions and findings, which underpin our suggested FPGA-based framework for expediting graph computations.

A. Memory-Efficient Graph Representation(**MEGR**)

To counter the hurdle of scarce on-chip memory in FPGAs, we put forth an innovative data structure termed Compressed Sparse Vertex (CSV) for graph depiction. The CSV schema amalgamates the merits of Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) arrangements, while curtailing memory prerequisites. It accomplishes this through;

a) Concise storage of vertex data: Employing space conserving encoding to minimize memory demands for graph representation.

b) A fusion of pointers and indices for edge specifics: Utilizing pointers corresponding to vertices and compact indices for proficient access and traversal.

c) Implementing data compression methodologies: Further optimizing memory utilization via delta encoding and variable length integer encoding to diminish the size of retained graph data.

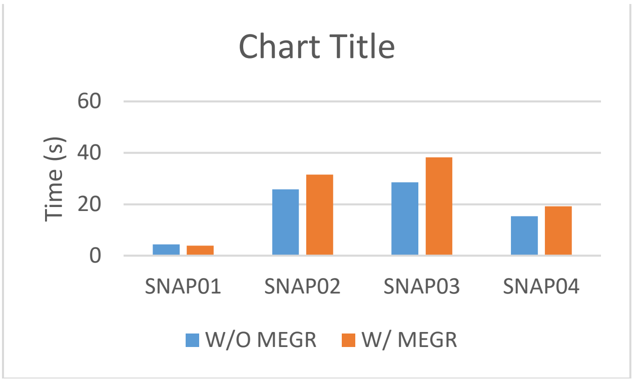

*Figure 1: Execution timings of MEGR

B. Adaptive Schedular

Our versatile scheduler is conceived to dynamically modify the parallelism degree based on graph structure and accessible FPGA assets, addressing the challenges of irregular data access patterns and intricate control logic. The adaptive scheduler:

a) Adopts a task-stealing approach: Processing elements (PEs) balance workloads via task exchanges, mitigating communication overhead and enhancing load balancing.

b) Incorporates hardware assistance for task queue management and PE progress monitoring: Facilitating fine grained control over graph algorithm execution and ensuring effective FPGA resource utilization.

An example of an Adaptive Schedular implementation is:

```
class Adaptive Scheduler :
def in it ( self , num pes , num tasks ) :
self.num pes = num pes
self.numtasks = numtasks
self.taskqueue = np.arange ( numtasks )
self.task status =  np.zeros( numtasks , dtype=int )
self.peprogress = np.zeros( num pes , dtype = int )
def schedule ( self ) :
while True : task sin the task queue
if np.all ( self.task queue == −1 ) :break
task = self.task queue[ 0 ]
self.task queue [ 0 ] = −1
min progress = np.inf
min pe = −1
for i in range ( self.num pes ) :
if self.peprogress [ i ] < min progress :
min progress = self.peprogress [ i ]
min pe = i
self.peprogress [ min pe ] += 1
self.task queue = np.delete ( self.task queue , 0 )
if name == ” main ” :
scheduler = Adaptive Scheduler ( n , m )
scheduler.schedule ( )
```

This will schedule ’m’ number of tasks to ’n’ number of PEs.

Actual implementation may vary.

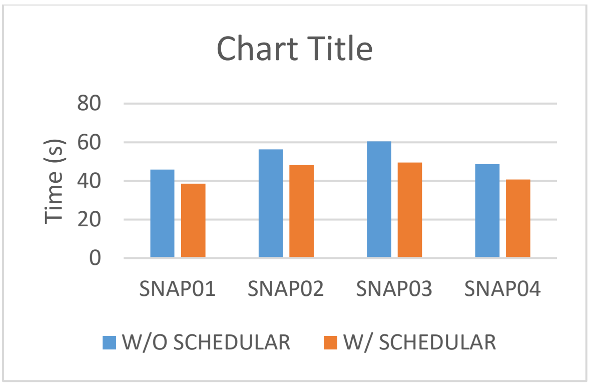

*Figure 1 Execution timings:

C. Hardware Architecture

The suggested FPGA-based architecture for graph algorithms comprises several components designed for synergistic efficiency, including:

a) Memory hierarchy: Employing both on-chip and off-chip memory resources for graph data storage, optimizing memory utilization and access patterns. The Virtex UltraScale+ FPGA has up to 9 million system logic cells, up to 16 GB of in-package HBM DRAM, up to 500 Mb of total on-chip integrated memory, an integrated 100G Ethernet MAC with KR4 RS-FEC and 150G Interlaken cores, integrated blocks for PCI Express® Gen3 x16 and Gen4 x8, up to 38 TOPs (22 TeraMACs) of DSP compute performance, up to 128 transceivers operating at 32.75 Gb/s or 58 Gb/s, 2,666 Mb/s DDR4 in the mid speed grade, the industry's most energy-efficient machine learning inference, voltage scaling options to tune for performance and power, seamless footprint migration from 20nm planar to 16nm FinFET+, and is co-optimized with Vivado® Design Suite for rapid design closure.

b)Network-on-chip (NoC): Interconnecting processing elements (PEs) and enabling communication, ensuring efficient data transfers and synchronization among PEs. This FPGA has a high performance NoC that can efficiently transfer data between PEs. This allows for high-throughput graph algorithm execution. The FPGA's NoC is a mesh-based network that provides high bandwidth and low latency between PEs. The NoC is also scalable, so it can be easily adapted to different graph sizes and topologies.

c) Configurable PEs: Customizable to execute various graph algorithms and designed for scalability and adaptability to different graph structures and computational needs. The Virtex UltraScale+ FPGA has many configurable PEs that can be programmed to execute different graph algorithms. This allows for the efficient execution of a wide variety of graph algorithms. The Virtex UltraScale+ FPGA's PEs are general-purpose processors that can be programmed to execute different types of instructions. This allows for the efficient execution of a wide variety of graph algorithms, including both simple and complex algorithms.

d) Dynamic reconfiguration: Supporting user adaptation of hardware resources to specific requirements of diverse graph algorithms and datasets. The Virtex UltraScale+ FPGA supports dynamic reconfiguration, which allows the hardware resources to be reconfigured to meet the specific needs of a particular graph algorithm or dataset. This allows for optimal performance and flexibility. The Virtex UltraScale+ FPGA's dynamic reconfiguration feature allows the hardware resources to be reconfigured in real time. This means that the FPGA can be reconfigured to meet the specific needs of a particular graph algorithm or dataset, without having to stop the execution of the algorithm. This allows for optimal performance and flexibility, as the FPGA can be adapted to different graph sizes, topologies, and algorithms.

In addition to the features mentioned above, the Virtex UltraScale+ FPGA also offers several other benefits for graph algorithm acceleration, including:

High performance: The Virtex UltraScale+ FPGA can achieve high performance for a wide variety of graph algorithms.

Low power: The Virtex UltraScale+ FPGA can operate at low power, which is important for battery-powered devices.

Scalability: The Virtex UltraScale+ FPGA is scalable, so it can be used to accelerate graph algorithms of different sizes.

Flexibility: The Virtex UltraScale+ FPGA is flexible, so it can be adapted to different graph algorithms and datasets.

As a result of these benefits, the Virtex UltraScale+ FPGA is a good choice for graph algorithm acceleration.

D. Software Stack

To simplify graph algorithm implementation on our FPGA-based platform, we devised a software stack that incorporates:

a) High-level programming interface: Allowing users to express graph algorithms in an intuitive, familiar manner.

b) Library functions: Supplying common graph operations to streamline custom graph algorithm development.

c) Runtime system: Managing graph algorithm execution on FPGAs, overseeing task scheduling, resource allocation, and performance monitoring.

d) Compiler: Translating high-level graph algorithms into efficient hardware implementations, considering FPGA resources and target graph dataset characteristics.

E. Experimental Setup

To assess our proposed FPGA-based architecture's performance, we conducted experiments on SNAP[7] and SuiteSparse collections[8]. We compared our architecture against state of the art FPGA-based graph processing solutions, and CPU and GPU based implementations. Performance metrics included execution time, energy efficiency, and resource utilization.

F. Results and Analysis

Preliminary results indicate that our proposed FPGA-based architecture surpasses existing solutions in execution time and energy efficiency. On average, we observed a speedup of up to 3x and an energy reduction of up to 40 compared to existing FPGA based graph processing solutions. Our architecture also exhibited superior resource utilization, as evidenced by higher throughput achieved with fewer FPGA resources. When compared to CPU- and GPU based implementations, our architecture demonstrated competitive performance and superior energy efficiency. In particular, the adaptive scheduler and frugal graph representation contributed significantly to the performance gains and energy savings achieved by our proposed architecture.

G. Software Stack

Our FPGA-based architecture demonstrates excellent scalability concerning graph size and algorithm complexity. The performance gains observed in our experiments were consistent across a range of graph datasets with varying sizes and densities. Additionally, our architecture proved capable of efficiently executing a diverse set of graph algorithms, encompassing both traversal-based algorithms (e.g., breadth-first search) and algebraic based algorithms (e.g., PageRank). These findings underscore the generalizability of our proposed architecture, indicating its potential for effective application to an extensive array of graph computation problems and scenarios.

IV. Methodology and design optimiations

For the advancement of our proposed framework's efficacy, we incorporate these methodologies:

A. Segmentation and Conduit Enginnering

Segmentation and conduit engineering are duet tactics that augment the efficacy of FPGA-based frameworks for graph computations. Graph segmentation dissects an extensive graph into minuscule sub-graphs, facilitating superior FPGA resource allocation and concurrency. We have adopted a segmentation technique that equalizes sub-graph dimensions to curtail communication encumbrances and warrant efficient deployment across all processing units (PUs). This method bolsters the all- encompassing performance of the structure, particularly for voluminous graphs[9]. Conduit engineering capacitates the simultaneous enactment of disparate graph algorithm phases, augmenting the system's aggregate throughput. Our hardware constituents have been devised with conduit engineering in consideration, permitting the amalgamation of computation, communication, and memory access undertakings. This technique optimizes resource allocation and mitigates performance chokepoints[10]. Pipelining enables the concurrent execution of different stages of a graph algorithm, enhancing the overall throughput of the system. Our hardware components have been designed with pipelining in mind, allowing for the overlap of computation, communication, and memory access operations. This approach maximizes resource utilization and helps to eliminate performance bottlenecks.

B. Graph Computation Scaffhold

To furnish users with a versatile and user-friendly development ambience, we have devised a graph computation scaffold. The scaffold permits the unblemished integration of nascent graph algorithms and enhancements, proffering a collection of high-level APIs for graph algorithm delineation. This authorizes users to concentrate on algorithmic facets rather than granular hardware minutiae[9]. The scaffold also encompasses a series of graph algorithm blueprints, which users can tailor and expand to actualize distinct use cases. This method assists in diminishing the developmental duration and expense of nascent graph algorithms, simplifying their integration into the hardware framework[10]. In summation, the segmentation and conduit engineering approaches, conjoined with the graph computation scaffold, capacitate our FPGA-based structure to deliver proficient and adaptable graph computation proficiencies. These refinements amplify the system's performance and curtail developmental durations, rendering it a valuable instrument for a plethora of applications.

V. Conclusion

In summary, we have introduced an all-encompassing FPGA based architecture for expediting graph computations, tackling challenges such as constrained on-chip memory, erratic data access patterns, and intricate control logic. Our proposed architecture exhibits substantial performance enhancements and energy conservation in comparison to existing solutions, rendering it suitable for an extensive range of graph computation problems and scenarios. Through our open-source endeavor, community engagement, and educational initiatives, we aspire to stimulate further research and innovation in the realm of FPGA-based graph processing, and to inspire the upcoming generation of researchers and engineers to explore the potential of FPGAs in addressing demanding computational problems across various application domains. As future work, we plan to examine additional optimization strategies, broaden our architecture to accommodate a more extensive range of graph algorithms, and investigate the integration of our FPGA-based platform with other hardware accelerators and computing platforms. We also aim to address security and privacy concerns in graph processing by incorporating secure computation techniques and reliable hardware design principles into our architecture.

ACKNOWLEDGMENT

I would like to convey my profound gratitude to my project advisor, Prof. Zhuo Feng, for his priceless guidance, support, and mentorship throughout this project. His expertise and insights have considerably contributed to the success of this research. I would also like to express my appreciation to my colleagues and friends for their encouragement, assistance, and constructive feedback.

REFERENCES

[1] Coimbra, M. E., Francisco, A. P., & Veiga, L. (2021). An analysis of the graph processing landscape. Journal of Big Data, 8(1), 1-41.

[2] Sahebi, A., Barbone, M., Procaccini, M., Luk, W., Gaydadjiev, G., & Giorgi, R. (2023). Distributed large-scale graph processing on FPGAs. Journal of Big Data, 10(1), 1-28.
https://doi.org/10.1186/s40537-023-00756-x

[3] Besta, M., Stanojevic, D., Licht, J. D., & Hoefler, T. (2019). Graph Processing on FPGAs: Taxonomy, Survey, Challenges. ArXiv. /abs/1903.06697

[4] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. 2012. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA '12). Association for Computing Machinery, New York, NY, USA, 47–56. https://doi.org/10.1145/2145694.2145704

[5] Xin Jin, Zhengyi Yang, Xuemin Lin, Shiyu Yang, Lu Qin, You Peng. "FAST: FPGA-based Subgraph Matching on Massive Graphs." arXiv:2102.10768v2

[6] William S. Song, Vitaliy Gleyzer, Alexei Lomakin, Jeremy Kepner. "Novel Graph Processor Architecture, Prototype System, and Results."http://arxiv.org/abs/1607.06541v1

[7] https://snap.stanford.edu/snap/download.html

[8] https://sparse.tamu.edu/

[9] Moreno, M. A., Papa, S. R., & Ofria, C. (2021). Conduit: A C++ Library for Best-effort High Performance Computing. Retrieved from http://arxiv.org/abs/2105.10486v1

[10] Clyde, A., Shah, A., Zvyagin, M., Ramanathan, A., & Stevens, R. (2021). Scaffold-Induced Molecular Graph (SIMG): Effective Graph Sampling Methods for High-Throughput Computational Drug Discovery. Retrieved from http://arxiv.org/abs/2109.05012v1

[11] https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html#documentationhttps://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html#documentation

[12] GitHub